Technical Guide

Security Design Patterns

by Bob Blakley, Craig Heath, and members of The Open Group Security Forum

The Open Group

Technical Guide

Security Design Patterns

ISBN: 1-931624-27-5
Document Number: G031

Published in the U.K. by The Open Group, April 2004.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Figures

Preface

The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including UNIX certification.

Further information on The Open Group is available at *www.opengroup.org*.

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at *www.opengroup.org/testing*.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at *www.opengroup.org/pubs*.

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards-compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published at *www.opengroup.org/corrigenda*.

This Document

The Open Group Security Forum decided to develop design patterns for information security design because its members saw that a new, more flexible approach to security architecture is needed.

There is a long history of The Open Group creating security specifications, providing structural guidelines, and defining application programming interface definitions (APIs) in C and other languages.

This approach no longer addresses the real needs of security system architects and designers, because:

- Most information systems are already in existence.
- The C language is decreasingly relevant as the useful way to express interface definitions.
- In modern software design the designers need instructional guidance that is language-independent; not prescriptive definitions written in C or any other programming language.

Design patterns are language-independent, flexible, adaptable, and scalable to all information system design problems.

This Technical Guide provides a pattern-based security design methodology and a system of security design patterns. This methodology, with the pattern catalog, enables system architects and designers to develop security architectures which meet their particular requirements. The introductory chapters of this Technical Guide provide background information on the design patterns approach to software architecture, describe how patterns are discovered and documented, and explain how to use patterns to design security into a system.

It is inherent in the nature of design patterns that they evolve with experience, and the security design patterns in this Technical Guide are no exception. It is therefore possible that a second edition will be forthcoming over time. We welcome feedback, which should be sent to *OGSpecs@opengroup.org* and will be included in preparation of a future edition. We also invite parties interested in working with us on progressing a future edition to get in touch using the same email address.

About the Authors

Bob Blakley
Chief Scientist, Security and Privacy, IBM Tivoli Software

Bob Blakley is chief scientist for Security and Privacy at IBM Tivoli Software. He is general chair of the 2003 IEEE Security and Privacy Conference and has served as General Chair of the ACM New Security Paradigms Workshop. He serves on the National Academy of Science's study group on Authentication Technologies and Their Privacy Implications. He was named Distinguished Security Practitioner by the 2002 ACM Computer Security and Applications Conference (ACSAC), and serves on the editorial board for the International Journal of Information Security (IJIS).

Bob Blakley was the editor of the OMG CORBA Security specification, and is the author of *CORBA Security: An Introduction to Safe Computing with Objects*, published by Addison-Wesley. Blakley was also the editor of The Open Group Authorization API Specification and the OASIS Security Services Technical Committee's SAML Specification effort. Blakley has been involved in cryptography and data security design work since 1979 and has authored or co-authored seven papers on cryptography, secret-sharing schemes, access control, and other aspects of computer security. He holds nine patents on security-related technologies.

Blakley received an A.B. in classics from Princeton University, and a Master's Degree and Ph.D. in computer and communications sciences from the University of Michigan.

Craig Heath
Product Strategist, Core OS and Security, Symbian

Craig has been working in IT security since 1988, including positions at The Santa Cruz Operation as security architect for SCO UNIX, and at Lutris Technologies as security architect for the Enhydra Enterprise Java Application Server.

He has been a member of The Open Group Security Forum (originally the X/Open Security Working Group) since 1993, sitting on the Steering Committee since 1999. He has contributed to several published security standards, including XBSS (baseline system security requirements), XDAS (distributed audit), and XSSO (single sign-on). He has also participated in standards work within POSIX, IETF, the Java Community Process, and the Open Mobile Alliance.

Craig graduated from the University of Warwick with a B.Sc. in computer science in 1984. He has two patents pending on security-related technologies.

Trademarks

Acknowledgements

The Open Group gratefully acknowledges:

- The major contribution of George Robert (Bob) Blakley III, Chief Scientist, Security and Privacy, IBM Tivoli Software, for his leadership and expertise in writing, editing, and verifying the technical content of this Technical Guide
- The assistance of Craig Heath, Product Strategist, Core OS and Security, Symbian, for preparation of new material (principally in Chapter 8) and editing for consistency.
- The significant contributions of pattern drafts, and of review comments and verification of the security design patterns in this catalog provided by other members of The Open Group Security Forum, particularly:

  Steve Jenkins, NASA Jet Propulsion Laboratory, CalTech
  Chris Milsom, Hewlett-Packard Company
  Eliot Solomon, SIMC
  Steve Whitlock, Boeing Corporation
  Ian Dobson, The Open Group

Referenced Documents

The following documents are referenced in this Technical Guide:

[APLRAC]
A Pattern Language for Designing and Implementing Role-based Access Control, Saluka R. Kodituwakku and Peter Bertok (Dept. of Computer Science, RMIT University, Australia) and Liping Zhao (Dept. of Computation, UMIST, UK).

[Appleton]
Patterns and Software: Essential Concepts and Terminology, Brad Appleton: *www.cmcrossroads.com/bradapp/docs/patterns-intro.html*.

[Brown-Fernandez]
The Authenticator Pattern, F. Lee Brown and Eduardo B. Fernandez, 1999.

[CA_OE]
The Oregon Experiment, Christopher Alexander, Oxford University Press, New York, 1975, ISBN: 0-19-501824-9.

[CA_NOPCL]
The Nature of Order Book 2: The Process of Creating Life, Christopher Alexander, Patternlanguage.com, August 2003, ISBN: 0-97-265292-2.

[CA_PL]
A Pattern Language: Towns, Buildings, Construction, Christopher Alexander, Oxford University Press, New York, 1979, ISBN: 0-19-501919-9.

[CA_TWB79]
The Timeless Way of Building, Christopher Alexander, Oxford University Press, 1979, ISBN: 0-19-502402-8.

[CESG Memorandum No.1]
CESG Memorandum No.1, Issue 1.2, October 1992, Glossary of Security Terminology.

[Coplien]
Links to James O. Coplien works on design patterns: *http://www1.bell-labs.com/user/copy/*.

[ECMA TR/46]
ECMA TR/46, Security in Open Systems, A Security Framework, July 1988, European Computer Manufacturers Association.

[Federal Criteria V1.0]
Federal Criteria Version 1.0, December 1992, Federal Criteria for Information Technology Security.

[Fernandez-Pan]
A Pattern Language for Security Models, Eduardo B. Fernandez and Rouyi Pan, Dept. of Computer Science & Engineering, Florida Atlantic University, 2001.

[GoF]
Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, October 1994, ISBN: 0-201-63361-2.

[IEEE Std 1003.0/D15]
IEEE Std 1003.0/D15, June 1992, Draft Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 0.

[ISO 7498-2]
ISO 7498-2: 1989, Information Processing Systems — Open Systems Interconnection — Basic Reference Model — Part 2: Security Architecture.

[ISO / IEC 10181-1]
ISO / IEC 10181-1: 1996, Information Technology — Open Systems Interconnection — Security Frameworks for Open Systems: Overview.

[ISO / IEC 10181-2]
ISO / IEC 10181-2: 1996, Information Technology — Open Systems Interconnection — Security Frameworks for Open Systems: Authentication Framework.

[ISO / IEC 10181-3]
ISO / IEC 10181-3: 1996, Information Technology — Open Systems Interconnection — Security Frameworks for Open Systems: Access Control Framework.

[ITSEC]
Information Technology Security Evaluation Criteria (ITSEC), Provisional Harmonized Criteria, June 1991, Version 1.2, published by the Commission of the European Communities.

[Lea]
Links to Doug Lea's checklist on how to write good patterns: *www.hillside.net/patterns/writing/writingpatterns.htm*.

[Mahmoud]
Security Policy: A Design Pattern for Mobile Java Code, Qusay H. Mahmoud, School of Computer Science, Carleton University, Ontario, Canada, 2000.

[NAI]
Security Patterns Repository, Version 1.0, Darrell M. Kienzle, Matthew C. Elder, David Tyree, and James Edwards-Hewitt. (Available from: *www.scrypt.net.˜celer/securitypatterns/*.)

[Object-Filter]
The Object Filter and Access Control Framework, Viviane Hays, Marc Loutrel, and Eduardo B. Fernandez, Dept. of Computer Science & Engineering, Florida Atlantic University, 2000.

[Role-Object]
The Role Object Pattern, Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf, 1997.

[Romanowsky]
Security Design Patterns Part 1, Sasha Romanowsky, Morgan Stanley Online, September 2001.

[TCSEC]
Trusted Computer System Evaluation Criteria, National Computer Security Center, Technical Report DoD 5200.28-STD, U.S. Department of Defense, 1985.

[TG_SDP]
Security Design Patterns, Bob Blakley, Craig Heath, and The Open Group Security Forum, Technical Guide, 2004, Doc. No. G031, ISBN: 1-931624-27-5 (this document).

[Tropyc]
A Pattern Language for Cryptographic Software, Alexandre M. Braga, Cecilia M.F. Rubira, and Ricardo Dahab, Institute of Computing, State University of Campinas, Brazil, 1998.

[Yoder-Barcalow]
Architectural Patterns for Enabling Application Security, Joseph Yoder and Jeffrey Barcalow, 1998.

The following provide further information:

- Informative links to design patterns information on the Hillside web site: *www.hillside.net*.
- Security Patterns and Related Concepts, and Security Engineering with Patterns (Markus Schumacher and Utz Roedig), Darmstadt University of Technology, Dept. of Computer Science, IT Transfer Office (ITO): *www.ito.tu-darmstadt.de/securitypatterns*.
- *Courier Patterns*, Robert Switzer, Goettingen Mathematisches Institut, Germany, July 1998.

Chapter 1

Introduction

Why Security Patterns?

There are many excellent Security Architecture publications available. They have variously appeared and rapidly become out-of-date for different reasons, but mostly because they either do not allow a sufficiently flexible architectural model to keep up with the evolving needs of a business and the evolving landscape of available information technology, or they limit the scope of the architecture that the business needs in other ways.

We see that in the current diversity of enterprise business requirements, software architects and designers increasingly need a way to design their own architectures—and then to be able to maintain their integrity and coherence as they evolve to match the changing needs of the business they serve. In the spirit of the proverb:

> ‘‘It’s better to teach a man how to fish than to give him fish.’’

we believe it is better to explain how to use a proven methodology—design patterns—to design security architectures that accurately fit the needs of a business, than to prescribe a range of fixed architectures that system architects and designers then have to modify and mould to make a best approximate fit with their systems.

What are Patterns?

Brad Appleton [Appleton] has defined a ‘‘pattern’’ as:

> ‘‘A named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.’’

Design patterns tell their readers how to design a system, given a statement of a problem and a set of forces that act upon the system. In the information technology environment, patterns give information system architects a method for defining reusable solutions to design problems without ever having to talk about or write program code; they are truly programming language-independent.

Origins of Design Patterns Technique

The design patterns approach was invented by Christopher Alexander. Alexander is a buildings architect, who devised the idea of a design pattern in the course of his design work. He realized that in buildings design there are certain well-defined components that occur repeatedly and which can be described in design terms, and reused. He defined these repeating components as design patterns, and each time the problems recurred he reused the same design pattern to provide the solution. This technique maintained accuracy and consistency in design for the common components in his buildings designs.

Alexander first launched his ideas on the concept of design patterns in *The Oregon Experiment* [CA_OE], published in 1975. He followed this with *A Pattern Language* [CA_PL], published in 1977, and described his design patterns technique in his landmark book *The Timeless Way of Building* [CA_TWB79], published in 1979.

After Alexander’s book first described patterns for physical architecture, software architects saw parallels between architectural issues in buildings and in software; it was natural that they should adopt (and adapt) the design patterns approach to their own work.

Many papers and books have been published on design patterns since Alexander's 1979 book first appeared. The landmark patterns book for software architects is *Design Patterns: Elements of Reusable Object-Oriented Software* [GoF] published in October 1994. The authors (Gamma, Helm, Johnson, and Vlissides) of that book are known in the software patterns community as the ''Gang of Four''. Their book describes simple and elegant solutions to specific problems in object-oriented software design. It established design patterns as a method for software architecture.

Mature software design patterns, like patterns in any other discipline, capture solutions that have developed and evolved over time. Hence they are not the designs that people tend to generate initially. Mature patterns reflect many iterations of untold redesign and recoding, as developers have struggled for greater reuse and flexibility in their software. Design patterns capture refined solutions in a succinct and easily applied form.

The purpose of using patterns is to create a re-usable design element. Each pattern is useful in and of itself. The combination of patterns assists those responsible for implementing security to produce sound, consistent designs that include all the operations required, and so assure that the resulting implementations can be completed efficiently and will perform effectively.

Writing good design patterns is hard. We recommend reading the ''Gang of Four'' book to gain an appreciation of just how much expertise and skill is required to do it well. Equally, the flaws in inadequate patterns become evident when they are put to real use; unfortunately there is no such thing as a self-checking facility for a design pattern.

Today, the design patterns technique for designing software architecture to suit any organization's business needs is firmly established, and many design patterns for software components have been published. The good ones have been proven as ''good'' because they have stood the test of time through repeated successful application.

In this spirit, we want the security design patterns in this Technical Guide to be proven as good, so expect them to evolve with experience. It is therefore possible that a second edition will be forthcoming over time. We welcome feedback, which should be sent to *OGSpecs@opengroup.org* and will be included in preparation of a future edition. We also invite parties interested in working with us on progressing a future edition to get in touch using the same email address.

Objectives of this Technical Guide

Many software design patterns have been published. Some security patterns exist; the Referenced Documents section (see **Referenced Documents**) lists a number of them. What doesn't exist yet is a comprehensive system of security patterns together with a methodology for constructing a secure system by combining the correct patterns in the correct way. The Open Group Security Forum established the Security Patterns initiative which produced this Technical Guide in order to fill this gap.

The aim of this Technical Guide is to provide a catalog of security design patterns, and a methodology for using those patterns to design a secure system, which will enable software architects and system designers who:

- Have a specific security problem in a specific context
- Want to develop an architecture to solve that security problem in that context
- Would like to know how The Open Group security experts would approach their task

to produce a system architecture which meets their security requirements, and which is maintainable and extensible from the smallest to the largest systems

When to Use this Technical Guide

There is a huge installed base of computing systems throughout the world. Business increasingly depends on the secure operation of its computing systems. It is the task of information security architects and systems designers to design and upgrade all these computer systems to incorporate increasingly more sophisticated security mechanisms (often off-the-shelf mechanisms whose properties can't be changed), so as to provide the increasing levels of protection that business needs.

In a world of diverse computing systems, security architects and systems designers need sound guidance rather than prescribed language-dependent solutions, because they need to be able to work out their own system architecture solutions that will provide the right system designs for their business requirements in their contexts, and then to be able to upgrade those system designs to match the evolving needs of their business.

Use of design patterns enables them to do this. The design patterns in this catalog provide a sufficient set to enable design of software system architectures that address security concerns. They can also be used to verify that integrating and evolving extensions and upgrades is achieved in a secure manner. This design patterns approach provides a framework for continued secure system evolution to match the changing needs of the business.

Chapter 2

The Nature of Patterns

A comprehensive source for information about patterns, and a bibliography of reference publications, is available at: *www.hillside.net/patterns*.

2.1 Minimal Definition for a Pattern

Accepting the definition of (in Chapter 1) a ‘‘pattern’’ as:

> ‘‘... a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces ...’’

we can construct the essential content and features of a pattern definition. It’s clear that a pattern definition must include a name; a description of the problem, its context, and the system of forces which must be addressed; and a description of the structure of the family of successful solutions the pattern recommends. In other words, a pattern’s definition must include at least the following:

Pattern Name Provides a memorable and descriptive way to refer to the pattern.

This formal process to name and describe a pattern captures the key elements of knowledge about it that will remind programmers who will use it of its intent, and attaches a label to it that makes it easy to recognize.

The Problem A description of the contexts and situations in which the pattern is useful.

This is essentially the design rationale, describing under what conditions this pattern should be used.

The Solution A specific but flexible approach to solving the problem.

This provides the program structure for the pattern, and also serves as a language-independent description of a typical implementation.

Consequences Implementing the solution described in the pattern will require making specific trade-offs among competing forces. These trade-offs and their consequences are described here.

2.2 How to Recognize a Pattern

You might be wondering how you recognize a pattern if someone else hasn't already written it down. Jim Coplien [Coplien] recommends asking whether a solution to a problem has the following properties (if it has, it might be a pattern!):

- Is it a solution to a problem in a context?
- Can you tell the problem solver what to do in order to solve the problem?
- Is it a mature, proven solution?

  In this context, ''proven'' means it has been used multiple times by architects and designers who are familiar with proper use of design patterns and on all occasions has not been found to be flawed in any way.
- Is it something you did not invent yourself?
- Does the solution build on the insight of the problem solver, and can it be implemented many times without ever being the same twice?
- Can the solution be formalized or automated?

  If it can be formalized or automated, then do that instead of writing it as a pattern.
- Does it have a dense set of interacting forces that are independent of the forces in other patterns?
- Is writing it down hard work?

  If it is easy to write, it may not be a pattern, or it is likely that you have not thought hard enough about the forces that bear down on the situation.

2.3 Defining a Good Pattern

If you have identified a pattern and tried your hand at writing it down, you might be wondering how you know whether you've done it right. Doug Lea [Lea] has identified the following checklist for verifying that you have written a good pattern:

- It describes a single kind of problem.
- It describes the context in which the problem occurs.
- It describes the solution as a constructable software entity.
- It describes design steps or rules for constructing the solution.
- It describes the forces leading to the solution.
- It describes evidence that the solution optimally resolves forces.
- It describes details that are allowed to vary, and those that are not.
- It describes at least one actual instance of use.
- It describes evidence of generality across different instances.
- It describes or refers to variants and subpatterns.
- It describes or refers to other patterns that it relies upon.
- It describes or refers to other patterns that rely upon this pattern.

- It relates to other patterns with similar contexts, problems, or solutions.

Above all, a good pattern is one that has been proven to be effective through repeated use, with experiences of usage resulting in appropriate refinements to its definition within the context of the problem it addresses.

Chapter 3

The Open Group Pattern Template

In this Technical Guide, we have chosen to adopt a variant of the format used by the ‘‘Gang of Four’’ [GoF] to describe our patterns. Our format for describing a pattern includes the following elements:

Pattern Name (Scope, Purpose)

The pattern’s name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

Intent

A short statement that answers the following questions:

- What does the design pattern do?
- What is its rationale and intent?
- What particular design issue or problem does it address?

Also Known As

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates a design problem, and how the structures in the pattern solve the problem. The scenario will help understanding of the more abstract description of the pattern that follows.

Applicability

- What are the situations in which the design pattern can be applied?
- What are examples of design problems that the pattern can address?
- How can you recognize these situations?
- What forces must be reconciled when solving the problem?

An applicable situation should be included in the description.

Structure

A diagram illustrating the structure of the solution.

Participants

The entities participating in the design pattern, and their responsibilities.

Collaborations

How the participants collaborate to carry out their responsibilities.

Consequences

- How does the pattern support its objectives?
- What are the trade-offs and results of using the pattern?
- What aspect of system structure does it let you vary independently?

Implementation

What are the pitfalls, hints, or techniques that you should be aware of when implementing the pattern? Are there language-specific issues?

Known Uses

Examples of the pattern found in real systems. We have included at least two examples from different domains.

Related Patterns

- What design patterns are closely related to this one?
- What are the important differences?
- With which other patterns should this one be used?

Chapter 4

The System of Security Patterns

The Patterns Catalog (see Chapter 7 and Chapter 8) in this Technical Guide provides a coherent set of security design pattern definitions that can be used to provide a framework for building a secure system.

The patterns fall into two parts:

1. Available System Catalog

   Contains structural design patterns which facilitate construction of systems which provide predictable uninterrupted access to the services and resources they offer to users.

2. Protected System Catalog

   Contains structural design patterns which facilitate construction of systems which protect valuable resources against unauthorized use, disclosure, or modification.

4.1 Available System Patterns

The Available System patterns include:

- Checkpointed System
- Standby
- Comparator-Checked Fault-Tolerant System
- Replicated System
- Error Detection / Correction

4.2 Protected System Patterns

The Protected System patterns include:

- Protected System
- Policy
- Authenticator

  (This is a placeholder pattern for possible future expansion for the Authenticator class referred to in the Policy pattern.)

- Subject Descriptor
- Secure Communication
- Security Context
- Security Association
- Secure Proxy

Chapter 5

Design Methodology

This chapter describes a systematic methodology for using the security pattern catalogs which appear in Chapter 7 and Chapter 8 to design a system which has good availability and protection properties.

The methodology presented here is based on the notion of ‘‘generative sequences’’ developed by Christopher Alexander, and explained in his book *The Nature of Order Book 2: The Process of Creating Life* [CA_NOPCL].

A generative sequence is similar to an algorithm; it is a recipe for creating a design, with steps which need to be performed in order. Generative sequences tell you which patterns to start with, and which ones to apply in what order as you refine a design.

A generative sequence tells you how to apply patterns from a pattern catalog:

- Starting with a high-level sequence of operations defining a set of steps to apply, in order
- With no backtracking
- Making well-identified choices at each step
- In which each step involves a choice of which patterns to apply
- Where each step is likely to include making design tradeoffs among alternative patterns which could be used

The generative sequence for applying The Open Group security patterns to a design problem consists of a main sequence and two sub-sequences. The remainder of this chapter lists the steps in the main sequence and the sub-sequences. Chapter 6 further explains each step by way of an example.

5.1 System Security Sequence

The System Security Sequence defines the order in which security concerns are addressed. Note that this sequence assumes that you are not building a totally new system from scratch (although we think it would work in that case too!). Rather, the sequence assumes that your design is starting from a collection of components (perhaps products, or pre-existing code modules), many of which won’t be changed significantly during the process of constructing the overall system.

The System Security Sequence is as follows:

1. Choose a design which satisfies your functional objectives for the system (you might use patterns or any other design technique you choose to perform this step). Identify the components of your design, including their interfaces, data repositories, and communications links and protocols.
2. Apply the Available System Sequence.
3. Apply the Protected System Sequence.

In practice you should anticipate that you will not be able to meet all your availability and protection requirements. Experience suggests that adding protection to a system can adversely affect the availability. Balancing competing requirements is a judgement the designer must make the on the overall system requirements.

5.2 Available System Sequence

The Available System Sequence is as follows:

1. Identify critical components; specifically, identify critical processing elements and critical repositories (a critical processing element is a processing element whose services are subject to an availability requirement; a critical repository is a data store whose contents must remain correct and up-to-date because of an availability requirement).
2. If data corruption is a likely cause of system failures, apply the Error Detection / Correction pattern to critical repositories to reduce the risk of data corruption.
3. If single component failures must be detected and corrected immediately in order to maintain a known level of confidence in the availability of the system, and if service outages are unacceptable, use the Comparator-Checked Fault-Tolerant System pattern to address the availability requirement. Otherwise, continue to the next step.
4. If single component failures are acceptable but service outage is unacceptable, use the Standby pattern to address the availability requirement. Otherwise, continue to the next step.
5. If partial or localized service outages are acceptable, as long as service is available from alternate components, use the Replicated System pattern to address the availability requirement.

5.3 Protected System Sequence

The Protected System Sequence is as follows:

1. Identify Resources (things to be protected—more familiar to those with Orange Book [TCSEC] backgrounds as ‘‘Objects’’), and Actors (entities whose access to the resource is to be restricted—more familiar to those with Orange Book backgrounds as ‘‘Subjects’’).
2. Define one or more Protected System (PS) pattern instances. You may identify only one instance, or you may identify more than one, but every resource which needs to be protected must be inside at least one PS instance. If you identify more than one instance, you must ensure that every pair of instances is in one of two relationships:

Disjoint	The two PS instances contain no resources in common.
Nested	Every resource contained in one PS instance is also contained in the other instance (but not necessarily *vice versa*).

   Designers should note that the current pattern catalog does not provide a way to deal with designs in which two PS instances overlap, but in which neither instance completely contains the other. When using the present version of the Protected System catalog, it is best to avoid configurations with overlapping PS instances.

   Note that the ‘‘nested’’ relationship could be degenerate, in the sense that two PS instances contain the same resources, but still interesting because the two PS instances have different guards.
3. For each pair of nested PS instances, choose a Secure Proxy pattern to define the relationship between the two PS instances.
4. For each PS guard, define a Policy.
5. For each Policy, define the required Secure Communication pattern instance. Instances of Secure Communication arise whenever disjoint PS instances need to communicate, or

whenever an actor needs to communicate to a PS instance.

6. Decide how the Policy will describe and identify Resources.
7. Derive Security Context from the Policy at each end of the Secure Communication.
8. For each Security Context, derive a Security Association and Subject Descriptor.
9. Examine each Security Context to determine whether it needs to be factored into a PS with multiple Security Contexts.

5.4 Review

Once you've completed the System Security Sequence, stop and reflect. Did you learn anything while you were applying the patterns? Is there something about your basic design which makes it difficult to secure? If so, could you do something different which would simplify the task of building a secure system?

In particular, you might want to think about two things:

1. Distribution issues: does your design co-locate different kinds of resources with different policy enforcement requirements? Does this complicate your security design? Does your design spread protected resources over several or many different systems, thereby requiring lots of protected system instances and lots of redundant components for availability? If so, could you simplify the security design by centralizing protected resources?
2. Composition issues: do you have a lot of nested protected system instances? Does this require you to define a lot of instances of Secure Communication, or make a lot of use of the Secure Proxy patterns? Does it make proper design of Policies very difficult? If so, could you simplify your design by ‘‘collapsing’’ several nested protected systems into a single protected system with a single guard?

Chapter 6

Example: Secure Mail

The following example illustrates the use of the Protected System generatives sequence to solve an example problem of how to define a system for securing email. The diagram below illustrates the secure email problem: two actors wish to communicate—a sender wishes to send a document to a receiver. The sender and the receiver wish to eliminate the risk that an attacker might be able to examine or modify the contents of the document while it is in transit between the sender and the receiver.

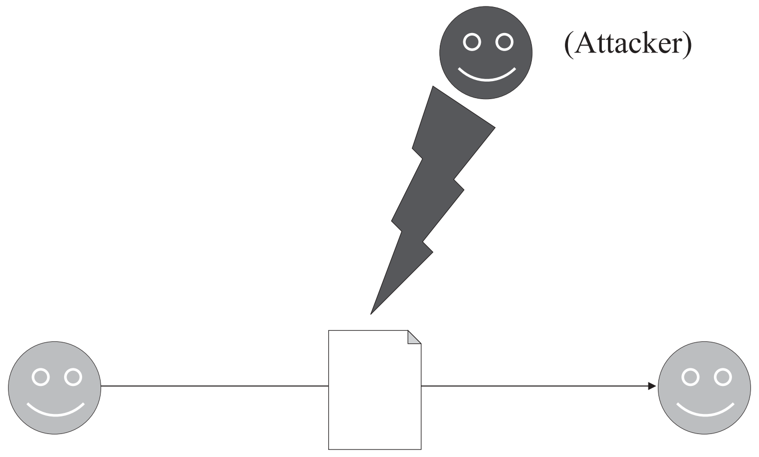


Figure 6-1 Example Problem: Secure Email

Step 1

In Step 1 of the sequence, we identify resources and actors. The resource is the email, and the actors are the Sender and the Target Addressee, plus the Attacker.

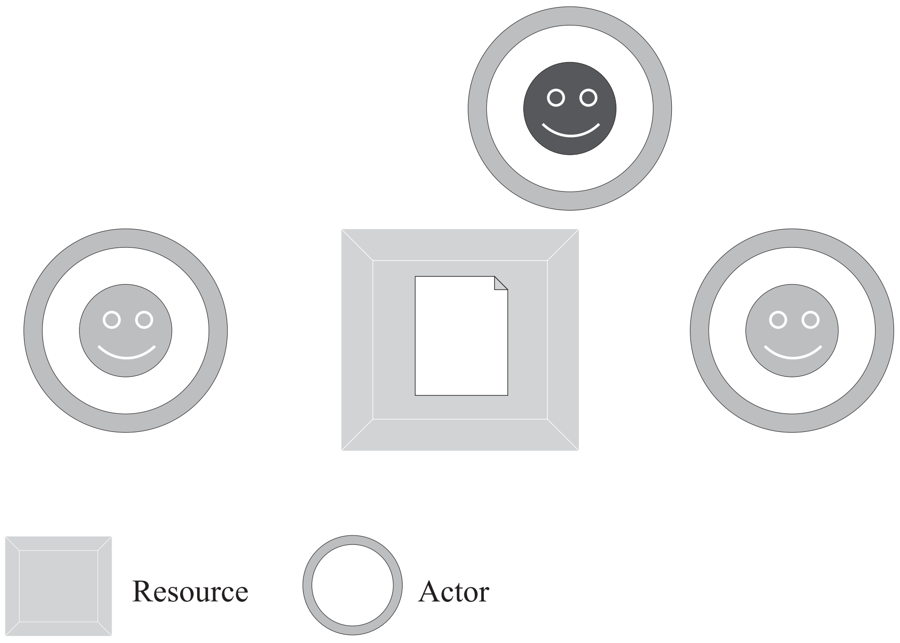


Figure 6-2 Identify Resources and Actors

Step 2

In Step 2 of the sequence, we identify instances of the Protected System pattern. The Protected System instance is a Guard enveloping the email document. For the present, we will assume that the Sender of the email is inside the Protected System boundary. This violates one of our rules (that all the actors should be outside the Protected System boundary) but we will fix this problem later in this example.

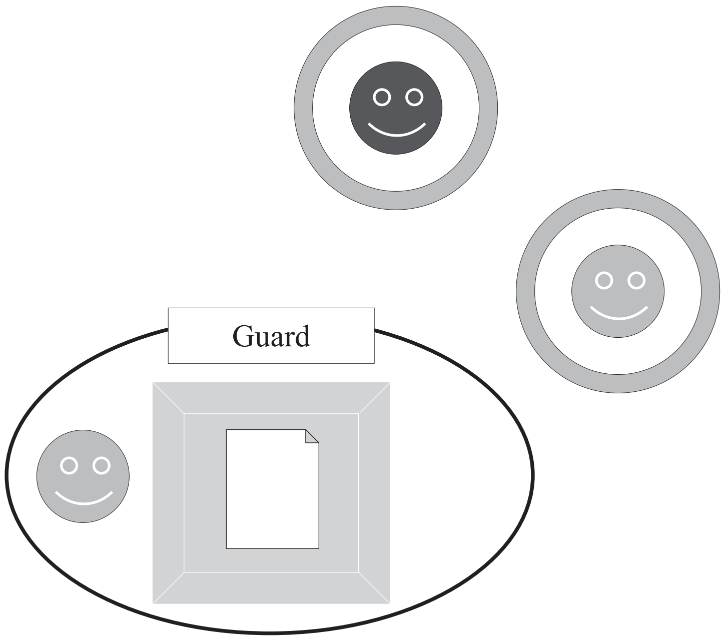


Figure 6-3 Define Protected System Instances

Step 3

Since we don't have any nested PS instances, we skip Step 3.

Step 4

In Step 4, we define the policy to be enforced by the Guard of our PS instance. The Guard needs to have a Policy that relates subject control information, target control information, and context information, in order to make a decision. Because this is a secure email, we define a Policy that allows only the intended Recipient to read the email.

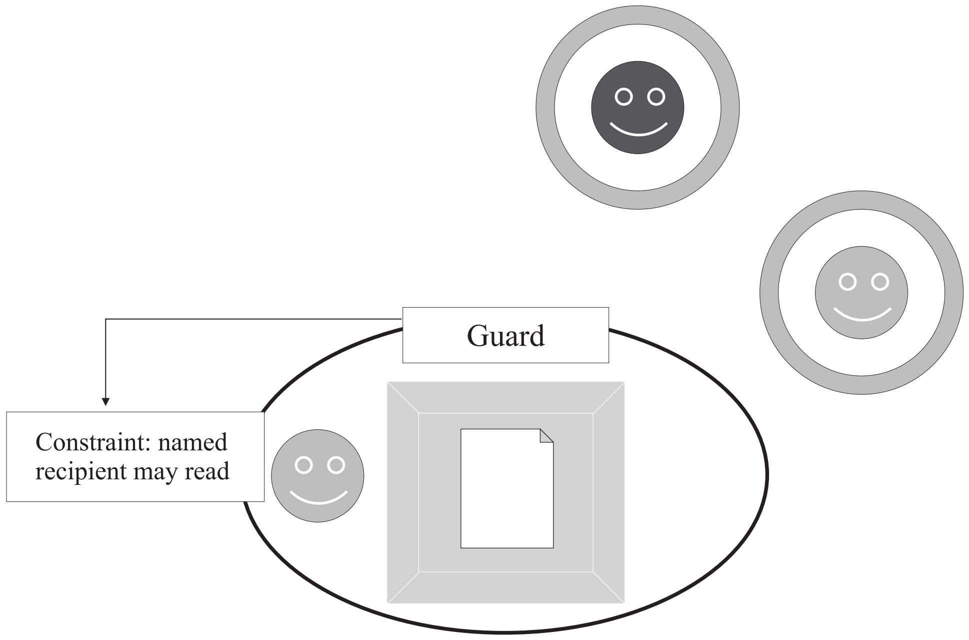


Figure 6-4 Define Policy

Step 5

Next we have to define the Secure Communication between the Guard and the target Recipient of the email. The security protection applied to the communication depends on the Guard's Policy, and in this case this Policy requires the application of confidentiality and authentication services.

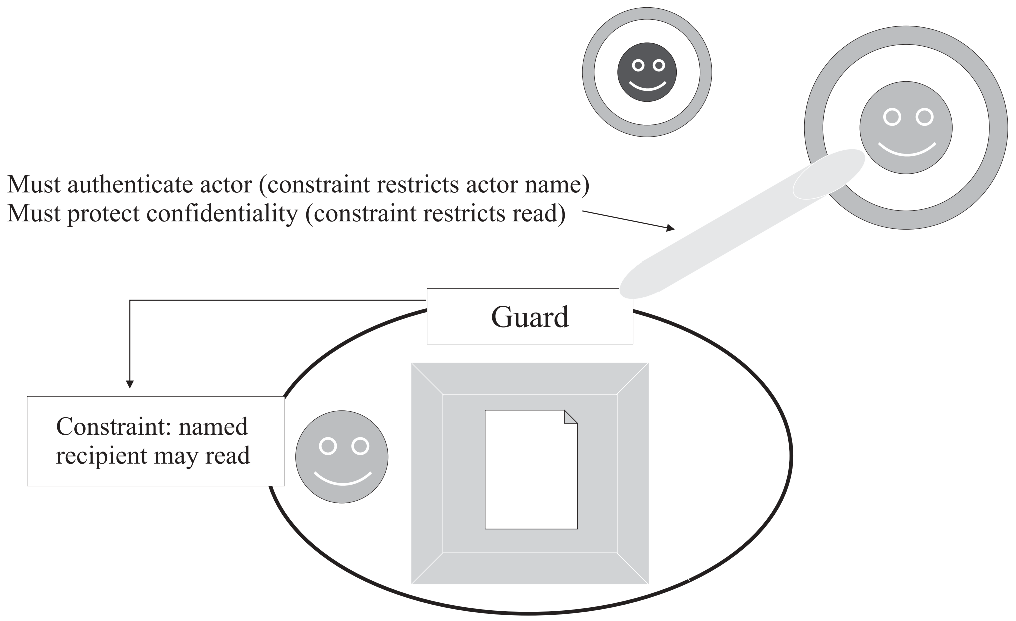

Must authenticate actor (constraint restricts actor name)
Must protect confidentiality (constraint restricts read)
Guard
Constraint: named recipient may read

Figure 6-5 Define Secure Communications

Step 6

We now decide how to describe Resources in the Policy. The Policy Rule needs to contain a field for the Recipient name because the Policy refers to the Recipient's name.

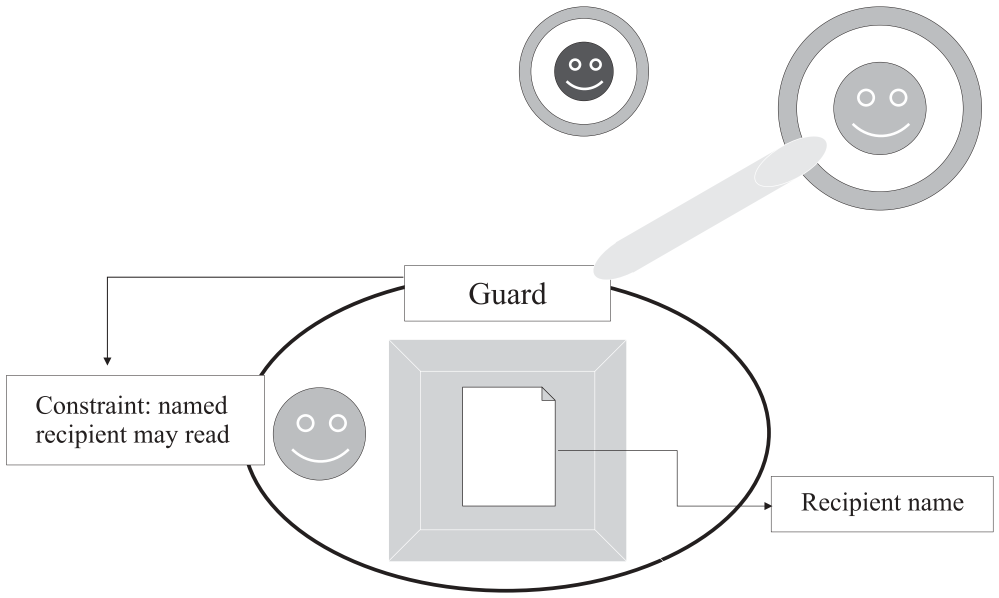


Figure 6-6 Derive Target Descriptor

Step 7

What needs to be done next is to describe how an actor convinces the Guard that it has the right subject control information to pass the policy test. This is where we use the Security Context pattern. The Guard uses the information in the Secure Communication to determine the partner name, and puts this into the Security Context.

First we derive from the policy what Security Context(s) is required at each end of the Secure Communication.

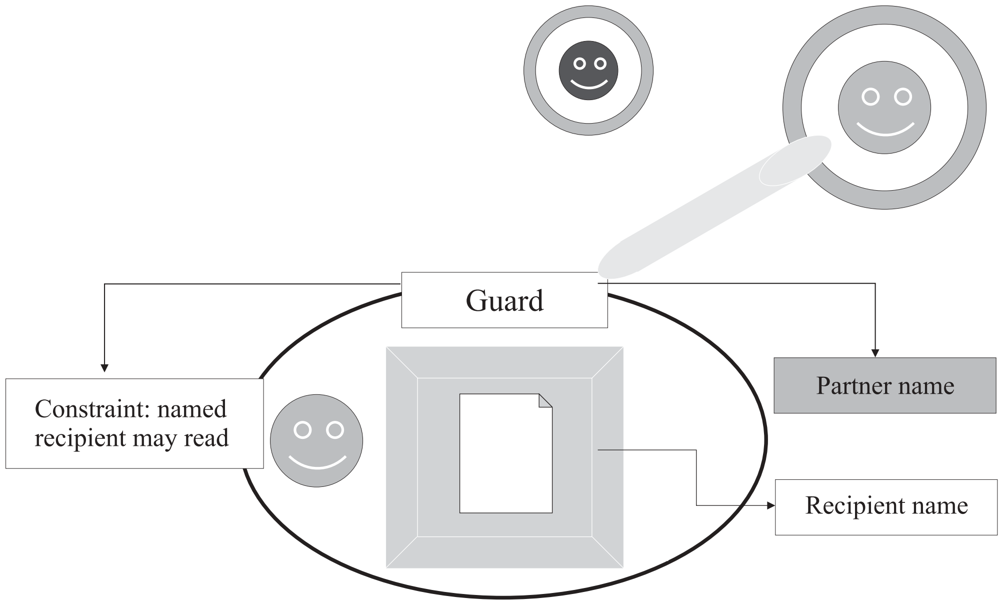


Figure 6-7 Derive Security Contexts

Step 8

In this step we define Subject Descriptors and Security Associations.

The Secure Communication uses an instance of the Secure Association to send the name for the client to the Guard. In order to do this it needs to get the Client name — in this case from the Subject Descriptor.

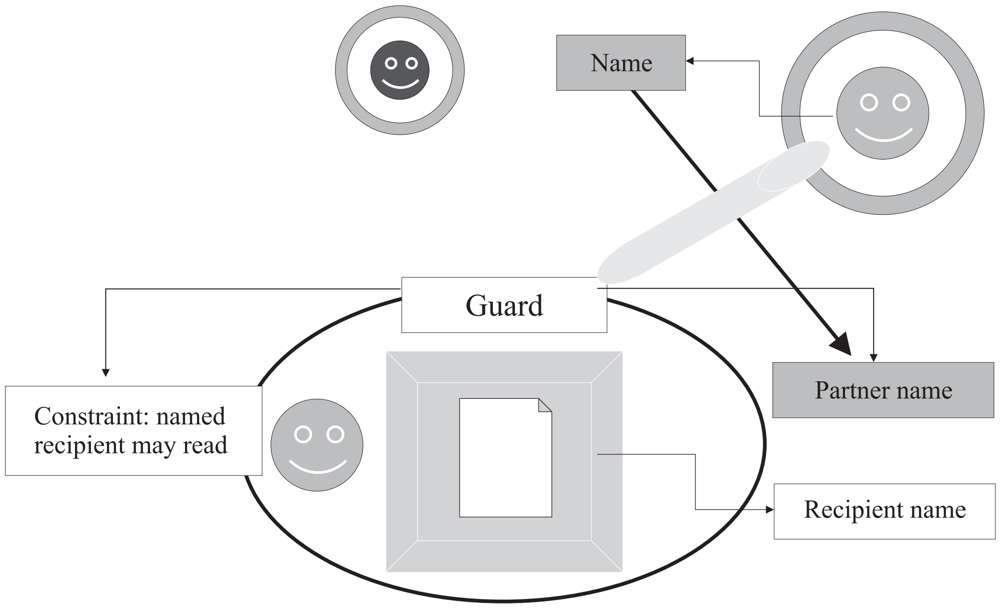


Figure 6-8 Derive Secure Associations and Subject Descriptors

Those who know email systems may recognize this scheme as similar to ‘‘hushmail’’.[1] In this system, the Protected System is enforced by encryption, and thus the message is inside the Protected System boundary even when it is not on the Sender’s machine.

This represents a problem where both parties must share a key.

1. Copyright © 2001 Hush Communications

   Refer to the Hush Encryption Engine^{TM} white paper available at: *corp.hush.com/info_center/document_library/hush_patent_wp.pdf*.

Step 9

In this step we check to see whether we need to introduce additional instances of Protected System or Secure Communication, in order to refine the design. The solution is to introduce a mutually trusted third party with whom all parties share a key. To do this we have to factor the Protected System.

Our rules for the Protected System define that all the actors must be outside the Protected System. So we have to move the Sender outside the Protected System, and add a constraint to the Guard's Policy saying that anyone can write a message so long as the Sender identifies the Recipient.

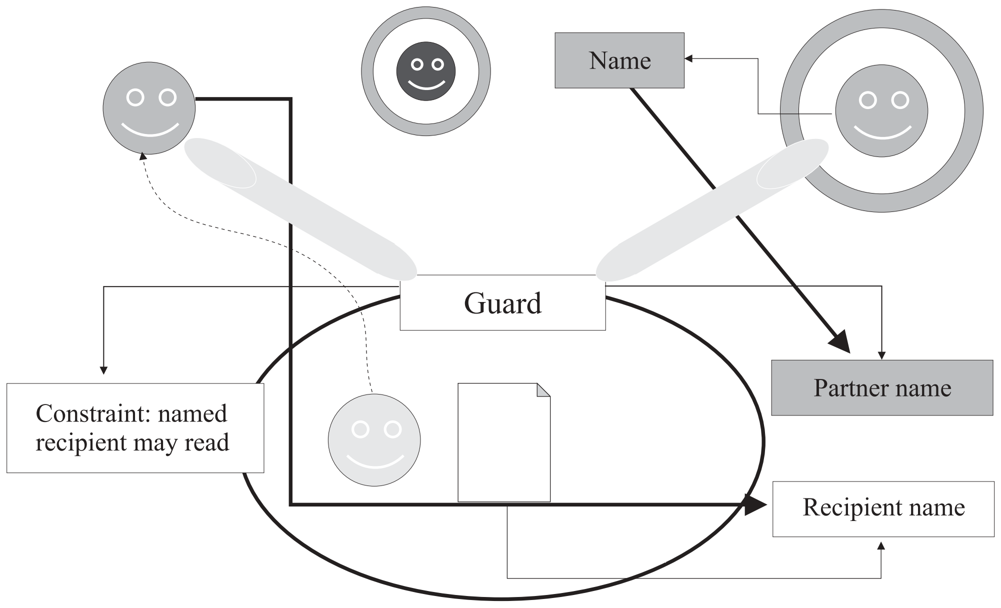


Figure 6-9 Consider Factoring Protected Systems

To support this Policy change, we need to define a second Secure Communication, which does not need to authenticate the partner (the Sender) but does need to require the Sender to identify the Recipient and also needs to prevent the content of the message from being revealed to an Attacker.

The definition for this Secure Communication instance will be developed in future Writers' Workshops.

Chapter 7

Available System Patterns

The Available System patterns are a coherent set of security design pattern definitions that facilitate construction of systems which provide predictable uninterrupted access to the services and resources they offer to users.

7.1 Checkpointed System

Intent

Structure a system so that its state can be recovered and restored to a known valid state in case a component fails.

Also Known As

Snapshot, Undo

Motivation

A component failure can result in loss or corruption of state information maintained by the failed component. Systems which rely on retained state for correct operation must be able to recover from loss or corruption of state information.

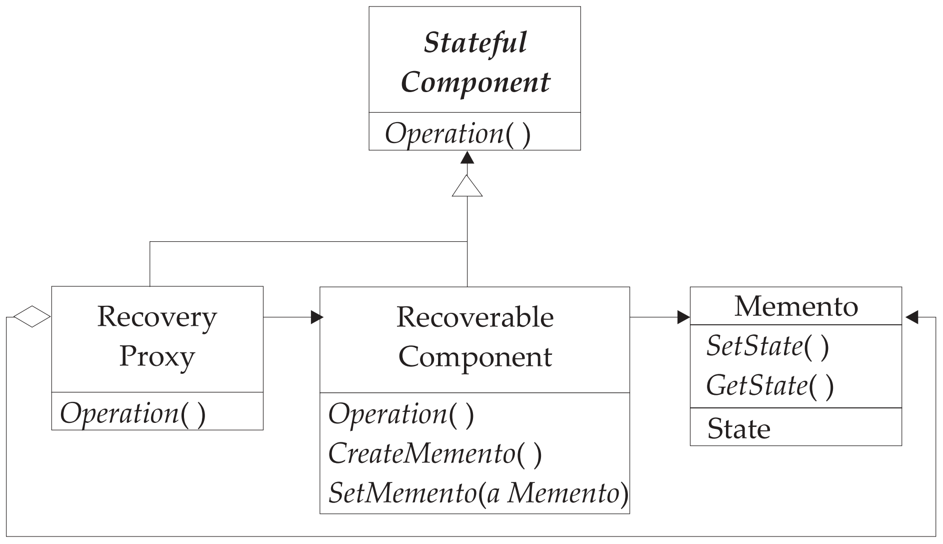
Applicability

Use Checkpointed System when:

- Operations on a component update its state.
- Correctness of the system's operation depends on correctness of its components' state.
- Component failures could cause loss or corruption of a component's state.
- Transactions which occurred between the time a state snapshot is taken and the time the system is rolled back to the snapshot state are irrelevant or inconsequential, or can be reapplied.

Structure

The Checkpointed System pattern consists of a Recovery Proxy [Proxy: GoF] and a Recoverable Component which periodically saves a recoverable version of the component's state as a Memento [GoF]. The Memento can be used to restore the component's state when required.



Participants

- Stateful Component

  Abstract class. Defines component operations.

- Recovery Proxy

  Proxy [GoF] for Recoverable Component. A Stateful Component. Caretaker for Recoverable Component's Mementos. Initiates creation of Mementos when Recoverable Component state changes. Detects failures and initiates state recovery by instructing Recoverable Component to restore state from Memento.

- Recoverable Component

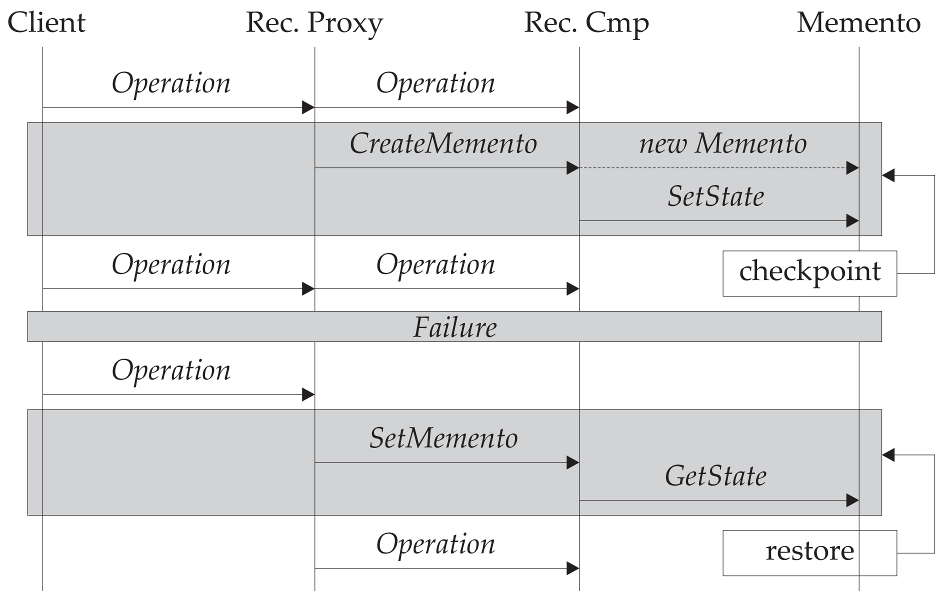
  A Stateful Component. Implements component operations. Periodically saves component state to Memento to support later recovery operations. Restores component state when required.

- Memento [GoF]

  The Recoverable Component's externalized state.

Collaborations

- The Recovery Proxy responds to requests to perform operations.
- The Recovery Proxy periodically instructs the Recoverable Component to create a new Memento to save the Recoverable Component's current state.
- In the event of a failure, the Recovery Proxy instructs the Recoverable Component to restore its state using the information stored in the Memento, and then instructs the Recoverable Component to execute requested operations. Note that any state resulting from operations performed after the most recent state save will be lost.



Consequences

Use of the Checkpointed System pattern:

- Improves component fault tolerance.
- Improves component error recovery.
- Increases system resource consumption (extra resources are required for the Memento).
- Increases system complexity; creating a Memento may require the creation of work queues or other transaction management constructs to ensure consistency of the state data stored in the Memento.
- May increase system latency or decrease throughput if creation of the Memento requires processing to pause or stop.
- Allows loss of a small number of transactions and their associated state.
- Increases system cost per unit of functionality.

Implementation

A wide variety of implementation approaches are possible. Examples include:

- A wide variety of configurations that provide the ability to ‘‘restart’’ the system from a known valid state, either on the same platform or on different platforms.

Known Uses

The periodic save feature of many applications (for example, Microsoft Word) is an instance of the Checkpointed System pattern.

Related Patterns

Recovery Proxy is a Proxy [GoF].
Recovery Proxy is the Caretaker for a Memento [GoF].

7.2 Standby

Intent

Structure a system so that the service provided by one component can be resumed from a different component.

Also Known As

Disaster Recovery, Backup Site

Motivation

In many system implementations it is only cost-effective to implement a single, coarse 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).

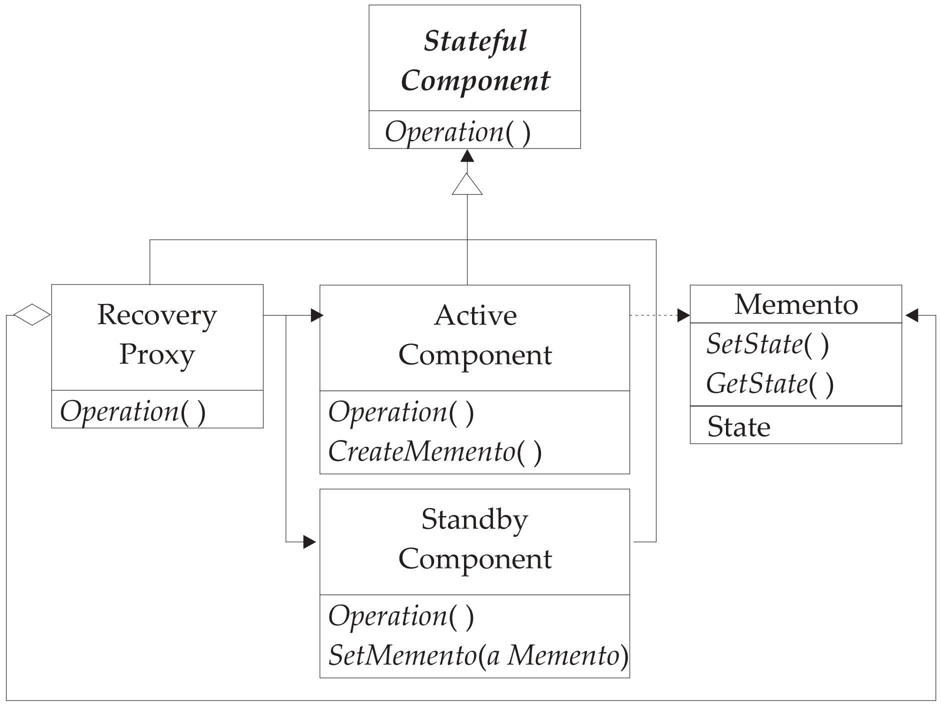
Applicability

Use Standby when:

- It is not acceptable for a single component failure to cause a system service outage.
- It is anticipated that a failed component may not be recoverable, but a similar or identical backup component is available.
- A small number of transactions occurring between the time a component fails and the time service is restored using a backup component are irrelevant or inconsequential, or can be recovered and reapplied.
- Employing a duplicate component is economical.
- Externalizing component state is feasible.

Structure

The Standby pattern consists of one active Recoverable Component and at least one Standby Recoverable Component. When the Standby is activated, the Memento (or Mementos) of the active component are consumed by the State Recovery facility of the Standby component, which ‘‘restores’’ the state to the Standby component and activates it.



Participants

- Active Component

  A Stateful Component. Performs operations on behalf of clients. Periodically saves state to Memento.

- Recovery Proxy

  Proxy [GoF] for Active and Standby Components. A Stateful Component. Caretaker for Active Component's Mementos. Initiates creation of Mementos when Active Component state changes. Detects failures and initiates recovery by instructing Standby Component to restore state from Memento and routing operations to Standby Component.

- Standby Component

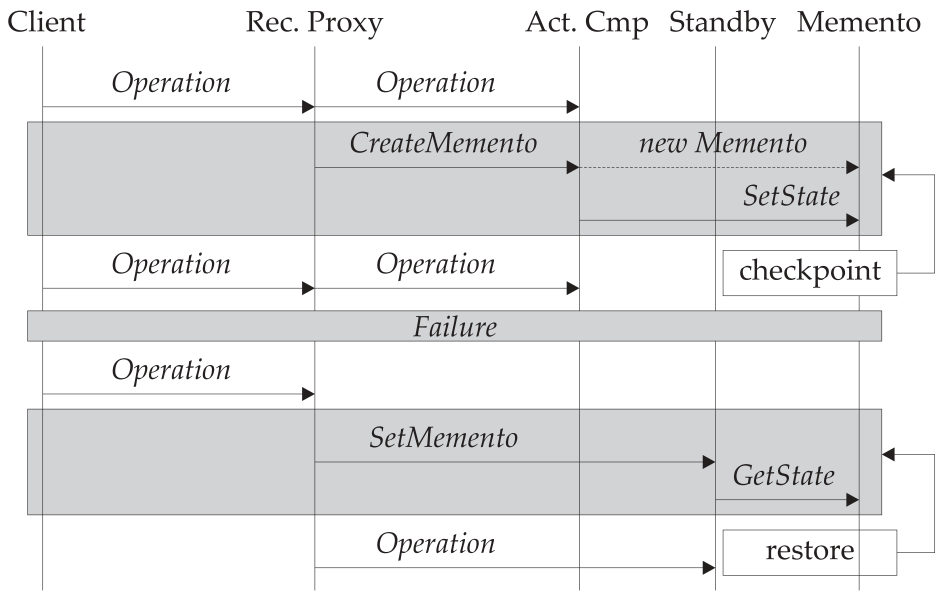
  Waits for failure of Active Component. Upon failure, restores state from Memento and activates.

- Memento

  A Memento [GoF]. Encapsulates the state of the Active Component. Used by the Standby Component to restore the system's state and resume operations.

Collaborations

- Recovery Proxy responds to client requests for operations and dispatches them to the Active Component.
- From time to time, Recovery Proxy instructs Active Component to checkpoint its state by creating a Memento.
- In case of a failure of the Active Component, Recovery Proxy activates the Standby Component by restoring the Memento's state to it and routing requests to it instead of the failed Active Component. Note that any transactions which the failed Active Component executed after its last checkpoint will be lost.



Consequences

Use of Standby:

- Improves system resistance to component failures.
- May introduce a substantial delay between component failure and standby activation.
- Increases system complexity. Creating the Memento may require the creation of work queues or other transaction management constructs to ensure consistency of the state data stored in the Memento.
- May impair system latency or throughput if creation of a checkpoint requires processing to pause or stop.
- Allows loss of a small number of transactions.
- May require substantial resources for storage of Memento information.
- Increases system cost by requiring at least one non-operational component.

Implementation

A wide variety of implementation approaches are possible. Examples include:

- Offsite backup

Known Uses

Offsite disaster recovery services often implement instances of the Standby pattern.

Related Patterns

Standby is a Checkpointed System [TG_SDP] with an identical spare Recoverable Component.

Standby uses a Memento [GoF] to communicate state information from the active component to the Standby Component when recovery is required.

7.3 Comparator-Checked Fault-Tolerant System

Intent

Structure a system so that an independent failure of one component will be detected quickly and so that an independent single-component failure will not cause a system failure.

Also Known As

Tandem system

Motivation

It is sometimes very important to detect component faults quickly, or to detect component faults at a specific point during processing, to prevent component faults from causing system failures. Inspection of the output of a component may not directly reveal whether a fault has occurred or not. Some mechanism is required to support detection of faults which have not yet caused a failure.

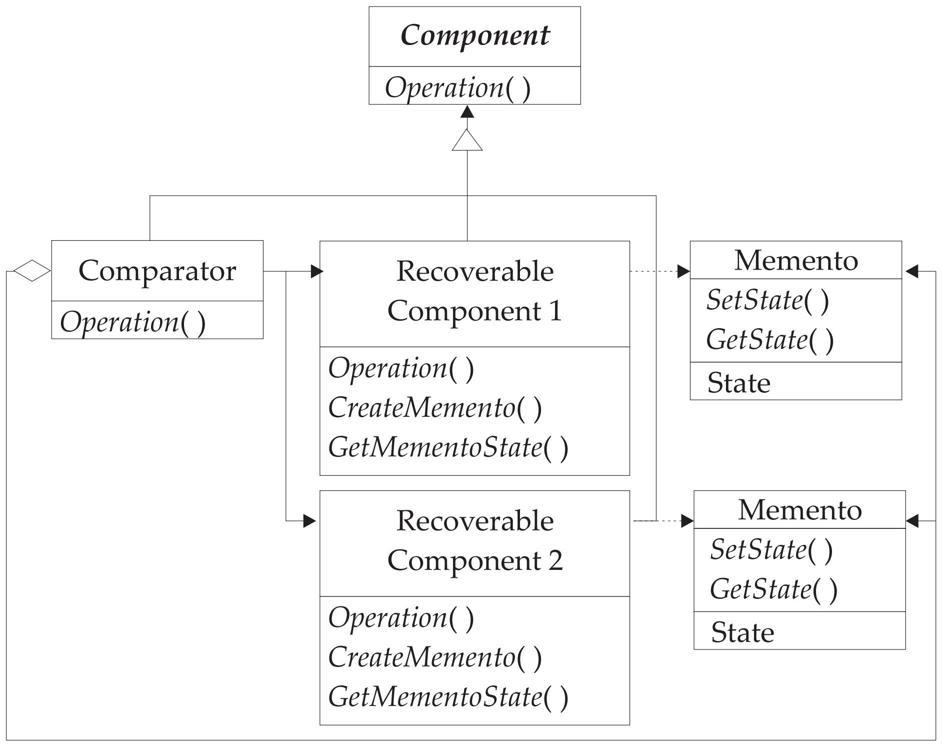
Applicability

Use Comparator-Checked Fault-Tolerant System when:

- Faults in one component are not expected to be strongly correlated with similar or identical faults in another component (this will usually be the case when faults are caused by factors external to components; it will often not be the case when faults are caused by component design or implementation errors).
- It is feasible to compare the outputs or internal states of components.
- Component faults must be detected soon after they occur, or at specific points during processing, but in any case before they lead to a system failure.
- Duplicating system components is economical.

Structure

A Comparator-Checked Fault-Tolerant System consists of an even number of Recoverable Components [TG_SDP] (often four or more), organized as sets of pairs, together with a Comparator for each pair. Each comparator examines Mementos [GoF] produced by each member of its pair to determine whether they match. If the Mementos do not match, the Comparator concludes that a fault has occurred in one of the components and takes corrective action.



Participants

- Recoverable Components

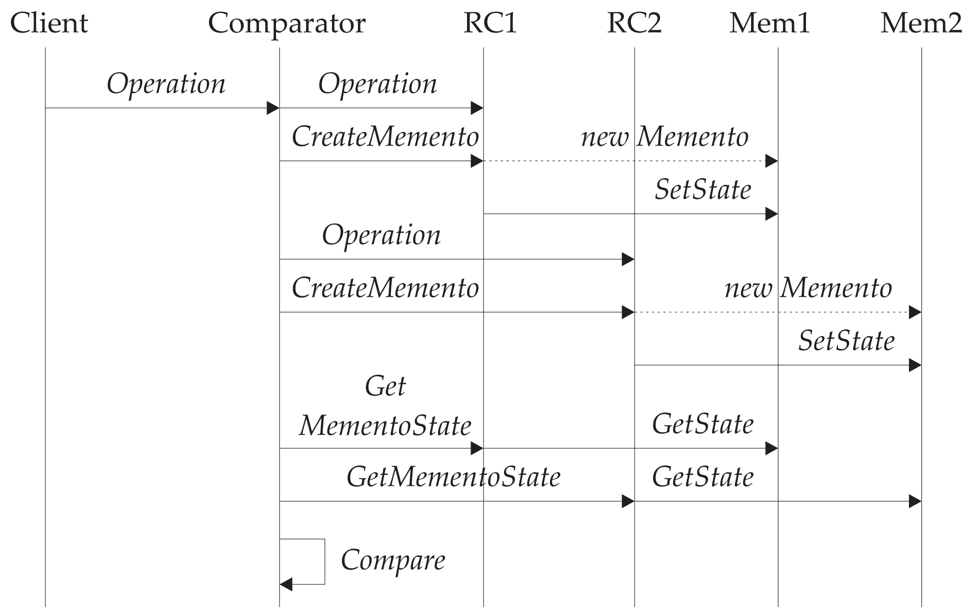
  Perform operations on behalf of clients. Each Recoverable Component is a member of a pair.

- Comparator

  A Proxy [GoF] for a pair of Recoverable Components. The Caretaker for Recoverable Components' Mementos [GoF]. Checks Mementos created by the members of its pair of Recoverable Components. If the Mementos do not match, the Comparator concludes that a fault has occurred in one of its Recoverable Components and initiates corrective action. In systems consisting of two or more pairs, the usual corrective action is to take the faulted pair offline.

Collaborations

- Comparator responds to requests for operations.
- Comparator routes each request to both Recoverable Components, each of which creates a Memento externalizing its state upon completion of the operation.
- Comparator retrieves state from both Mementos and compares them.
- If the states of the Mementos match, Comparator returns the operation's result to the client; otherwise (if the states do not match), Comparator initiates recovery actions.



Consequences

Use of the Comparator-Checked Fault-Tolerant System pattern:

- Improves system tolerance of component faults.
- Substantially increases component costs.
- Increases system complexity. Creating the Memento may require the creation of work queues or other transaction management constructs to ensure consistency of the state data stored in the Memento. Creating the Comparator and its recovery function will also add complexity.
- May impair system latency or throughput if creation of a checkpoint requires processing to pause or stop.

Implementation

- The Comparator's error checking mechanism works by comparing the two Mementos. If the state comparison shows any difference, the pair is taken offline. In some implementations, the ‘‘failed’’ pair continues processing inputs but presents no outputs. Continued processing allows the next collaboration.
- The Comparator of a failed pair may collaborate with the error checking mechanisms of the surviving pair's Comparator to identify which Recoverable Component of the failed pair has actually failed. This function can be used to guide manual or automatic intervention, correction, and restart.
- A Comparator may use its Mementos to maintain a consistent externalized image of the ‘‘correct’’ state. This can be used to enable the restart of a failed element or its replacement.

Known Uses

The Tandem Nonstop operating system is an example of the Comparator-Checked Fault-Tolerant System pattern.

Related Patterns

Comparator is a Proxy [GoF] and the Caretaker for the Mementos [GoF] of its Recoverable Components.

7.4 Replicated System

Intent

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.

Also Known As

Redundant Components, Horizontal Scalability

Motivation

Transactional systems often susceptible to outages because of failure of communication links, communication protocols, or other system elements. Nevertheless, it is important to assure availability of transaction services in the face of such failures.

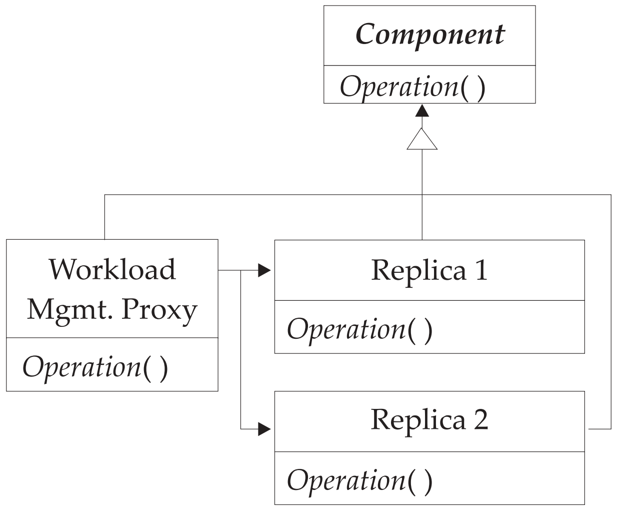
Applicability

Use Replicated System when:

- A system's state is updated via a series of individual transactions.
- The completion state and result of each transaction must be accurately reflected in the system state.
- Equivalent services must be provided simultaneously from multiple ''points of presence'', each of which must rely on and consistently update the same system state.
- Link failures are more likely than component failures.
- Each point of presence can be provided with reliable access to a master copy of the system state.
- Operational procedures call for a service to be periodically relocated from one platform or site to another, and brief pauses in processing for the purpose of relocation are acceptable. (Relocation might be desired to match the point of provision of the service to the locality of the offered load, or when the service may need to be relocated to a more capable (''larger'') platform to meet peak load demands.) Service must continue to be provided in the face of component or link failures.

Structure

Replicated System consists of two or more Replicas and a Workload Management Proxy which distributes work among the components. The Replicas must all be capable of performing the same work. The Replicas may be stateless or stateful. If they are stateful, they may be allowed to be inconsistent. If the Replicas are stateful and must be kept consistent, the Standby pattern may be used to ensure consistency of state across components.



Participants

- Replica

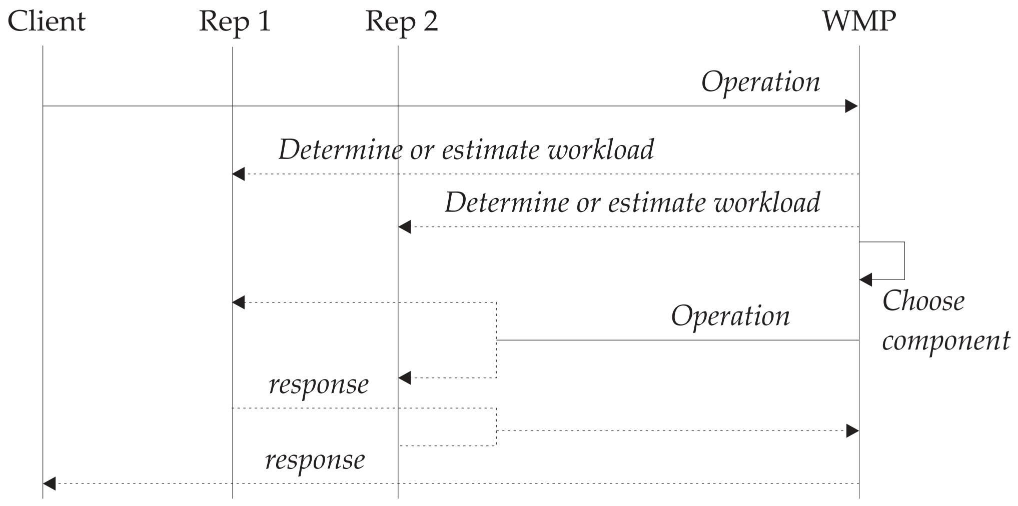
  Implements operations. All Replicas in a replicated system must support the same set of operations.

- Workload Management Proxy

  Dispatches operations to components based on workload scheduling algorithm.

Collaborations

- Workload Management Proxy responds to requests for operations.
- Workload Management Proxy dispatches operation requests to Replicas which are best able to handle them.



Consequences

Use of the Replicated System pattern:

- Improves system tolerance to component failures.
- Improves system ability to handle distributed load and link failures.
- Makes the Workload Management Proxy a single point of failure; may make the persistent data store a single point of failure.

Implementation

Future Writer's Workshops will identify appropriate text for this section.

Known Uses

Network Load Balancers (fronting replicated Web Servers, for example) are instances of the Replicated System pattern.

Related Patterns

Replicated System may use Standby [TG_SDP] to ensure consistency of state among its Replicas if this is required.

Replicated System's Workload Management Proxy is a Proxy [GoF].

7.5 Error Detection/Correction

Intent

Add redundancy to data to facilitate later detection of and recovery from errors.

Also Known As

Redundancy Check, Error-correcting Code, Parity

Motivation

Data residing on storage media or in transit across communication links is often susceptible to small, local errors.

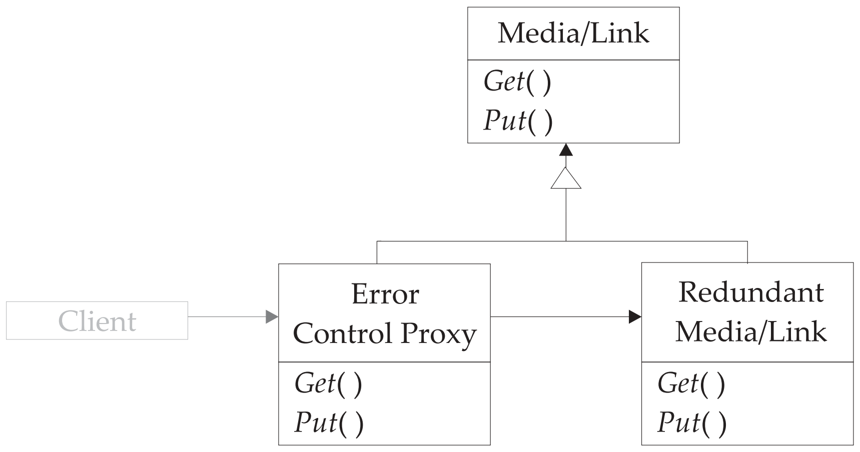
Applicability

Use Error Detection / Correction when:

- Storage media or communication links are susceptible to undetected or uncorrected errors.
- The format of data stored on media or communicated across a link can be modified to incorporate redundant error-control information.
- Some data expansion is acceptable.
- Data corruption is likely to be limited to a known number of errors per bit of data, and the distribution of errors is likely to be predictable in advance.

Structure

Error Detection / Correction consists of a Media device or a communications Link together with an Error Control Proxy.



Participants

- Redundant Media / Link

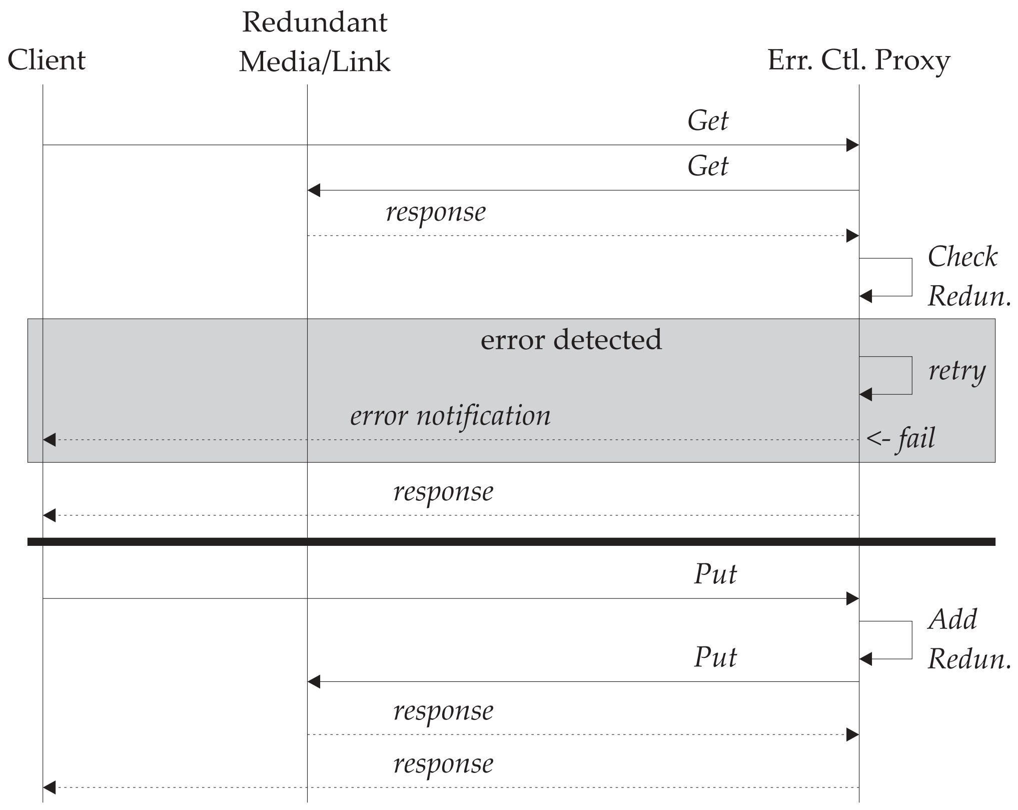
  The storage medium or communications link to which data will be written or from which data will be read.

- Error Control Proxy

  Adds redundancy to data written to a storage medium or communications link; uses redundant information to check integrity of (and, if possible, repair integrity of) data read from a storage medium or communications link.

Collaborations

- Error Control Proxy responds to client requests to put / get information to / from a communications link or media device.
- Error Control Proxy adds redundancy to information which is written to the link or device.
- Error Control Proxy checks previously added redundancy in order to verify the integrity of data retrieved from the communications link or media device. In the event that the verification fails, Error Control Proxy may retry the get operation before returning a failure code to the client.



Consequences

Use of the Error Detection / Correction pattern:

- Protects against loss of data integrity by detecting and, in some cases, correcting errors.
- Expands data by a known factor.
- Introduces a startup delay into data storage / transmission and retrieval / reception operations.

Implementation

Error-Control Code (for example, Cyclic Redundancy Check (CRC)), Cryptographic Hash, Digital Signature.

Note that performance overhead can be reduced to a constant startup latency by using streaming and parallelism.

Known Uses

RAID array, Disk storage CRC

Related Patterns

Error Control Proxy is a Proxy [GoF].

Chapter 8

Protected System Patterns

The Protected System patterns are a coherent set of security design pattern definitions that facilitate construction of systems which protect valuable resources against unauthorized use, disclosure, or modification.

8.1 Protected System

Intent

Structure a system so that all access by clients to resources is mediated by a guard which enforces a security policy.

Also Known As

Reference Monitor, Enclave

Motivation

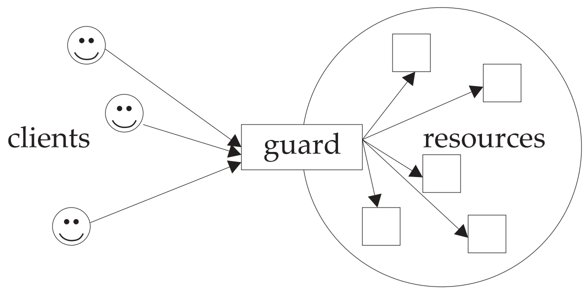
It is often desirable or imperative to protect system resources against unauthorized access. In order to do this it is necessary to evaluate requests to determine whether or not they are permitted by a policy. All requests must be evaluated against the policy; otherwise, unchecked requests might violate the policy.

To assure that all access requests are evaluated against the system's policy, a policy enforcement mechanism with the following properties must exist:

- The mechanism must be invoked on every access request.
- The mechanism must not be bypassable.
- The mechanism must correctly evaluate the policy.
- The mechanism's correct functioning must not be corruptible.
- The previous four properties must be verifiable to some stated level of confidence.

This pattern includes three elements:

1. An ''outside'', from which all access requests originate
2. An ''inside'', in which all resources are located
3. A correct, verifiable, incorruptible, non-bypassable ''guard'', which enforces policy on all requests from ''outside'' for resources ''inside''

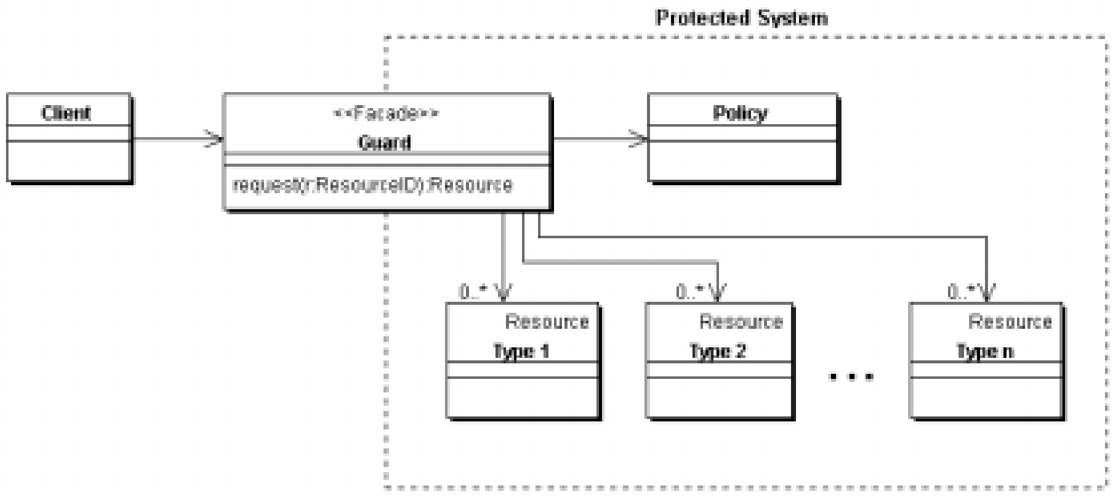


Applicability

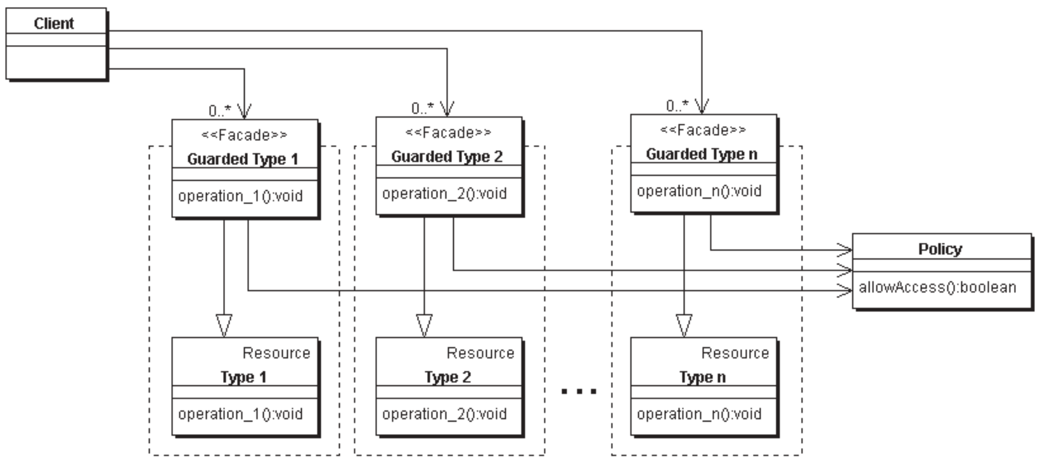
Use this pattern whenever access to resources must be granted selectively based on a policy. When designing secure systems by refinement, Protected System should be considered a candidate for the starting point of the refinement.

Structure

There are two main variants of this pattern. The first variant has a single centralized guard which mediates requests for all resources in the system.



The second variant distributes the responsibilities of the guard, such that there is a separate guard instance for each distinct type of resource.



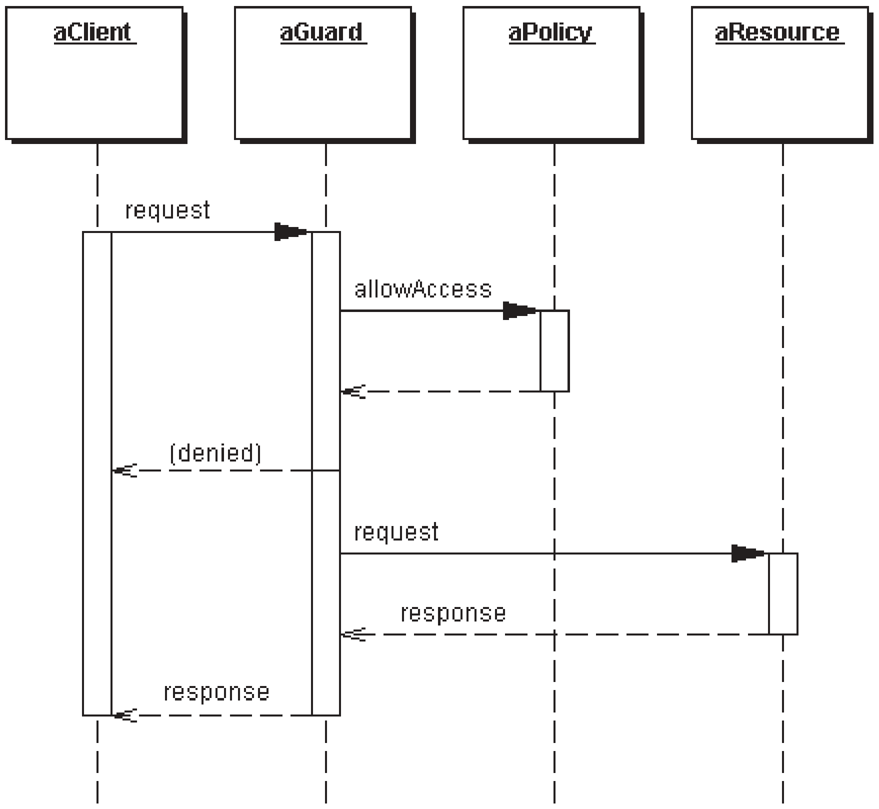
Hybrids of these variants are possible (see Implementation below).

Participants

- Client
  — Submits access requests to guard.
- Guard
  — Mediates requests to access protected resources. The Guard is a Facade [GoF].
  — Evaluates each access request against a policy; grants requests which are permitted by the policy, and denies requests which are forbidden by the policy.
  — Cannot be bypassed (no direct access by Clients to Resources is possible).
- Policy
  — Determines whether access to the resource should be granted.
  — Encapsulates the rules defining which Clients may access which Resources.
- Resource
  — Services requests for access to protected resources.

Collaborations

- Clients submit access requests to the Protected System's Guard.
- The Guard determines whether access requests should be granted or denied by consulting the Policy.
- If the Guard determines that an access request should be denied, it discards the request and returns. If the Guard determines that the access request should be granted, it passes the request on to the Resource for fulfillment and returns the response to the Client.



Consequences

Use of the Protected System pattern:

- Isolates resources.

  The system's resources are isolated by the Guard from any accesses which do not conform to the security policy enforced by the Guard.

- Loosens coupling between security policy and Resource implementation.

  Resource implementations do not need to be aware of security policy and do not have to be modified when security policy changes, since the policy is enforced by the Guard.

- Improves system assurability.

  Only the Guard implementation needs to be evaluated for correctness in order to ensure that the system correctly enforces its security policy.

- Degrades performance.

  In almost all implementations, interposing a Guard between the Client and the Resource imposes a performance penalty; this penalty may be significant. In operating system kernel implementations, the performance cost to cross the kernel boundary is often much higher than the cost to make a procedure call when the caller and the called routine are both in ‘‘user space’’ or both in ‘‘system space’’. In network configurations, Guards are usually network proxies (for example, routers) and their use requires an extra network message for each resource access.

Implementation

Several issues need to be considered when implementing the Protected System pattern:

- Isolation

  In order to provide complete protection of resources, the Guard must be non-bypassable.

  A firewall is a good example of a Guard which may be bypassable; if modems permit intermittent dial-up access to machines ‘‘inside’’, but access to the modems does not go through the firewall, then it will be possible to bypass the firewall and its associated security policy.

  Many microprocessor designs do not support complete address-space isolation between programs running in ‘‘system state’’ and programs running in ‘‘user state’’. It is difficult or impossible to design operating system kernels which are not bypassable to run on such microprocessors.

  Virtual machine architectures also suffer from failures of address-space isolation; several versions of the Java Virtual Machine, for example, shared public static variables between thread address spaces, which violated the thread isolation property assumed by the Java security model.

- Guard self-protection

  In order to protect resources, the Guard must function correctly. Among other things, this means that the Guard must be incorruptible.

  Corruptibility is often a consequence of a failure to validate input data.

  Many systems (including many Internet servers) are vulnerable to buffer overflow attacks. Buffer overflow attacks result from the Guard’s failure to check the size of input parameters provided by the client. A buffer overflow attack causes the Guard to execute malicious code provided by the client.

  Some systems are vulnerable to data poisoning attacks; data poisoning attacks which result from the Guard’s designers failing to define error responses for all possible invalid input data values. Data poisoning attacks exploit the Guard’s unanticipated response to an ‘‘improper’’ input value.

- Assurance

  It must be possible to demonstrate that the Guard functions correctly, and that the Guard is non-bypassable and incorruptible.

  Assurance is very difficult, and its difficulty scales super-linearly with increasing system size. The Protected System pattern contributes to assurability by minimizing the amount of code which must be assured, and by modularizing it to the Guard.

  Typical assurance activities include: disciplined design processes; documentation of all aspects of system design, implementation, production, delivery, and operation; assurance inspections; rigorous testing; and formal correctness verification.

- Alternative structures for the Guard

  The first variant of this pattern shown in Structure above uses a single Guard instance which mediates access to all Resources. The client must invoke this Guard in order to perform operations on any Resource.

  The second variant shows separate Guards for each resource type. This is still considered a single protected system, as there is a common Policy controlling access to all resources, although the access control decisions are triggered in multiple Guarded Types. Note that the interfaces presented by a Guarded Type may be identical to the Type which is being protected; in this case, the Guarded Type is a Proxy [GoF] for the underlying Type, and the Client need not be aware that it is invoking a Guarded Type.

A hybrid between these two variants is possible, such that the Client obtains its initial reference to the Resource from a centralized Guard, but the reference returned is to a Guarded Type which will perform supplementary access checks for each operation on the Resource. In this case, the centralized Guard is acting in a similar manner to an Abstract Factory [GoF].

Guards can be implemented as Proxies, or they can be embedded directly in the Resource implementation. Proxy Guards are easier to assure (because policy enforcement functionality is strongly separated from operational functionality), but they impose a larger performance penalty, because of the requirement for additional procedure calls or network messages to communicate requests and responses between the Guard and the Resource.

- Feasibility of externalizing policy enforcement

  In some systems, business rules are strongly integrated with policy enforcement, or policy is strongly dependent on the specific details of a resource request or of the resources to which access is being requested. In such systems, it may be very difficult or very inefficient to separate policy enforcement from processing of resource access requests.

  For example, if the desired policy depends on the specific values of all parameters of a resource access request, moving policy enforcement from the Resource to a centralized Guard may require essentially total duplication of the Resource manager's request processing code (and thus will impose substantial performance overhead without any corresponding gain in assurability of the policy enforcement code).

Known Uses

A very large number of secure system designs are instances of this pattern.

The Anderson Report first defined the structure described in the first variant of this pattern. It refers to the Guard together with the Resource managers it protects as a ‘‘Reference Monitor’’, and to the Guard itself as a ‘‘Reference Mediation Mechanism’’. In an operating system whose kernel is a reference monitor, the Guard is the operating system kernel (syscall) boundary, and the protected resources are files, pipes, and other operating system objects.

The Java 2 security architecture is an example of the second variant of the pattern. There is a single AccessController object which is responsible for making the decisions on whether operations should be permitted, based on Permissions granted to executing code. The checkPermission method of the AccessController is however invoked separately by each resource class, specifying the Permission that is to be checked for the requested operation.

The Java 2 AccessController design has one notable advantage over a monolithic reference monitor: new types of Resource can be introduced into the Protected System by simply defining a new Permission class. No change is necessary to the AccessController implementation (that is, the Policy component of this pattern).

A firewall is a Protected System whose guard is a router and whose resources are IP addresses and ports of systems ‘‘inside’’ the firewall.

A bank vault is a Protected System whose guard is the walls and vault door and whose resources are cash and gold bars.

Related Patterns

- As noted above, the Guard of a Protected System may be a Proxy [GoF]. GoF refers to a proxy used for this purpose as a ''Protection Proxy''.
- Further details relating to the Policy class are covered in the Policy [TG_SDP] pattern below.
- Role-based access control [APLRAC].
- Guard for Protected System [Brown-Fernandez].
- Security policy [Mahmoud].
- Security models, multi-level security [Fernandez-Pan].
- Object Filter access control framework [Object-Filter].
- Enabling application security, single access point, checkpoint [Yoder-Barcalow].

8.2 Policy

Intent

Isolate policy enforcement to a discrete component of an information system; ensure that policy enforcement activities are performed in the proper sequence.

Also Known As

Access Decision Function, Policy Decision Point [ISO / IEC 10181-3]

Motivation

Many systems (and components of systems) need to enforce policy. In such systems, the policy enforcement functions must be invoked, in the correct sequence, every time access is attempted to a resource which is subject to the policy.

If policy-enforcement code is intermingled with code which services resource requests, it may be difficult to verify that the policy enforcement functions are always invoked when necessary and that they are correctly implemented. It may therefore be desirable to isolate policy-enforcement code from other code in order to simplify verification of the correctness of the policy-enforcement code.

It may be desirable to use the same policy-enforcement code to protect more than one system component.

It may also be desirable to support changes in the code which makes policy decisions without requiring changes to code which enforces policy decisions.

Applicability

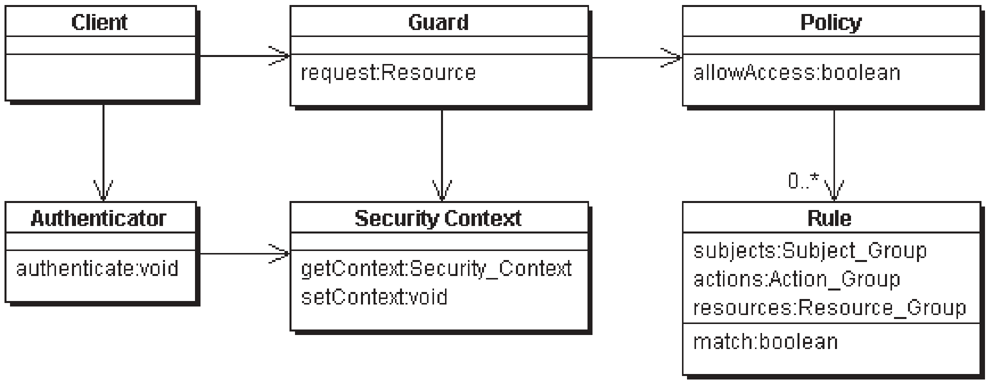
Use this pattern when:

- It is desirable to decouple the implementation of security policy from resource manager implementation.
- It is desirable to isolate policy-enforcement code to a minimum number of simple modules to simplify verification of correctness.
- It is desirable to isolate policy-enforcement code from policy decision evaluation code.

Do not use this pattern if:

- It is infeasible to make policy decisions outside the context of the resource manager which responds to requests.

Structure



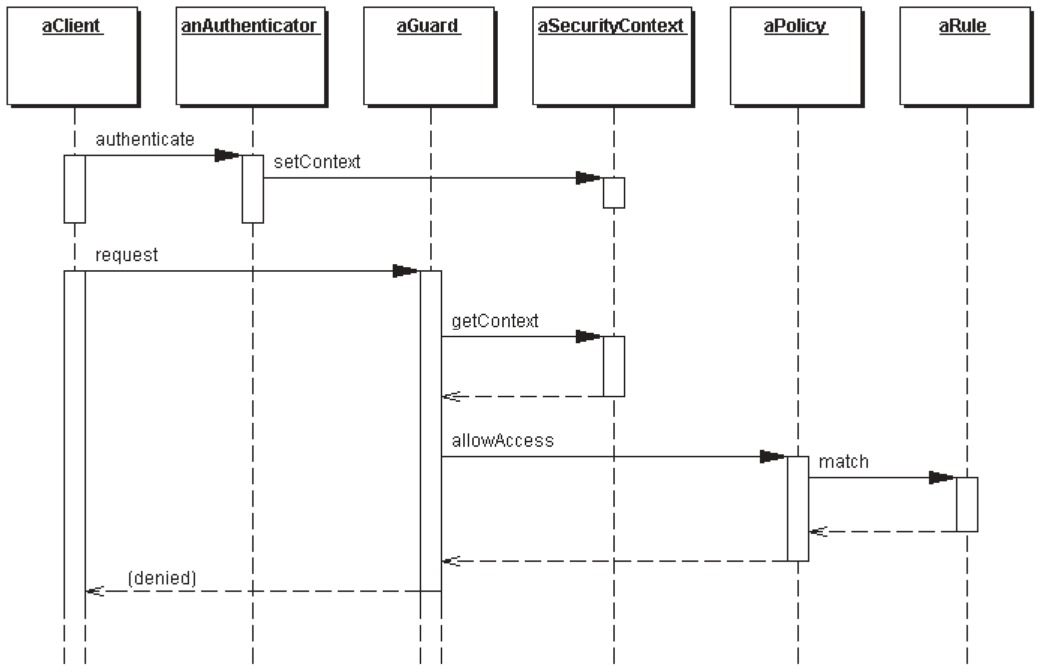
Participants

- Client
  - — Represents any subject governed by the system's policy.
  - — Gains access to resources by submitting requests to the Guard.
  - — May initiate user authentication if it is to operate on behalf of a human user.
- Authenticator
  - — Authenticates users.
  - — An optional component (the Security Context may be implicit or inherited).
- Guard (also known as Policy Enforcement Point (PEP))
  - — Collects client, request, target, and context attributes needed to make access control decisions.
  - — Requests access control decisions from the Policy.
  - — Rejects requests which are not permitted by the Policy.
  - — Sequences operations related to policy enforcement.
  - — May cache client identity and attribute information to optimize performance in cases where a single client submits multiple requests.
- Security Context
  - — Maintains credentials and security attributes for use in Policy decisions.
- Policy (also known as Policy Decision Point (PDP))
  - — Makes decisions to grant or deny access to resources based on client attributes, request attributes, target attributes, context attributes, and policy.
  - — Encapsulates a set of Rules determining which Clients can perform which operations on which Resource.

- Rule
  - — A component of the Policy expressing the permission for a specified set of Clients to perform a specified set of operations on a specified set of Resources.
  - — Structure of rules will vary depending on the Guard, the resources the guard protects, the structure of Subject Descriptors, naming of resources, and other factors.

Collaborations

- If the Client is operating on behalf of a specific user, and that user's security attributes have not previously been retrieved and cached, the Client may invoke the Authenticator to establish and verify that user's identity.
- If used, the Authenticator causes the user's security attributes to be included in the effective Security Context.
- The Client submits an access request to the Guard.
- The Guard determines the request, target, and context attributes.
- The Guard requests a policy decision (passing in the Client, request, target, and context attributes) from the Policy.
- The Policy checks which of the set of Rules matches the security attributes, requested operation, and the targeted Resource.
- If the Rules indicate that the request should be denied, the Policy returns a notification to the Guard, which then passes a failure notification to the Client.
- If the Policy result indicated that the request should be granted, the Guard passes the request for fulfillment to the Resource and relays the response back to the Client.



Consequences

Use of the Policy pattern:

- Loosens coupling between policy enforcement and resource implementation.

  This permits resource managers to be built with no awareness of authentication, attribute collection, policy evaluation, and policy enforcement.

- Ensures that security policy is checked before client requests for resources are fulfilled.

  Sequencing security policy checks and resource request fulfillment can be used to ensure that policy checks are always performed in the correct order, and that they are always performed before resource requests are fulfilled.

- Provides a single point of control and management for policy-related activities.

  Localizing policy-related operations improves assurability of the system by limiting the amount of system code which needs to be examined during assurance activities.

  Localizing policy-related operations may also create a single point of failure or attack; designers should take care to address these problems by hardening the Policy component and by addressing availability of the Policy (perhaps by use of the Redundant System pattern).

- Requires matching of the Rules parameter signature with resource namespace and operation signatures.

  The policy expressed by the Rules needs to ‘‘speak the same language’’ as the Guard uses when checking requests for access to resources.

  Adding new resources or new operations to a system without changing the Rules policy language and enforcement capability can weaken the protection provided by the Guard.

  For example, adding relational queries to a system whose Policy recognizes and protects only the files in which the relational tables are stored will leave the system vulnerable to unauthorized disclosures of information through inference.

- Imposes performance overhead.

  Separating policy enforcement from resource request fulfillment will usually introduce additional procedure calls or network overhead.

Implementation

- Time-of-check *versus* time-of-use

  Many policies are time-sensitive. Security authorization policies, for example, are sensitive to the status of a user’s account.

  In a system which uses the Policy pattern to separate policy enforcement from resource access request fulfillment, there will be a delay between evaluation of policy and fulfillment of requests. If this delay is long, there is a possibility that the user’s authorization status will have changed after the policy is evaluated but before the request is fulfilled (for example, the user’s account may have been suspended or revoked).

  Designers using the Policy pattern should take care to minimize the interval between the time the Policy makes a decision and the time the request is fulfilled by the Resource.

- Policy and Rule interface design

  Designing Policies which can be extended to support new types of resources and new operations on existing resources (so that it is not necessary to replace the system's Policy whenever a new type or version of a Resource is added to the system) is difficult. Very broadly extensible Policies are also very hard to assure, because of the difficulty of analyzing the policy which is actually enforced.

  In general, Policies consisting of only monotonic Rules (such as Rules which only express the granting of access, with denial of access being represented by an absence of any matching Rules) are easier to manage and assure.

  As it is a characteristic of a Protected System that a single Policy makes all access decisions, use of the Singleton [GoF] pattern may be appropriate to ensure that only a single Policy object will be instantiated.

  Rules are commonly expressed as triplets of (subject, operation, resource).

Known Uses

Firewall Routing Filter, Content Scanner, Reference Monitor, Credit Authorization

CORBASecurity provides three instances of Policy:

1. Client Secure Invocation Policies describe the client's minimum requirement for Quality of Protection (QoP) when initiating a Secure Communication.
2. Server Secure Invocation Policies define the range of Quality of Protection options which are supported when accepting a Secure Communication and also the minimum requirement for Quality of Protection when accepting a Secure Communication.
3. Object References contain a copy of the object's Server Secure Invocation Policy information.

Related Patterns

- Protected System [TG_SDP]
- Redundant System [TG_SDP]

  Introducing a centralized Policy implementation may, if done carelessly, create a single point of failure in a system which otherwise would not have one. Use of the Redundant System pattern is recommended to avoid this happening.
- Authenticator [Brown-Fernandez]
- Security models, authorization, multi-level security [Fernandez-Pan]
- Security policy [Mahmoud]

8.3 Authenticator

Intent

Note: We believe it may be useful to include a separate pattern expanding on the Authenticator class referred to in the Policy pattern above. This section is reserved as a placeholder. The utility of including such a pattern will be investigated in upcoming patterns workshops.]

Known Uses

PAM, JAAS

8.4 Subject Descriptor

Intent

Provide access to security-relevant attributes of an entity on whose behalf operations are to be performed.

Also Known As

Subject Attributes. The entity described may be referred to as a subject or principal.

Motivation

There are many security-relevant attributes which may be associated with a subject; that is, an entity (human or program). Attributes may include properties of, and assertions about, the subject, as well as security-related possessions such as encryption keys. Control of access by the subject to different resources may depend on various attributes of the subject. Some attributes may themselves embody sensitive information requiring controlled access.

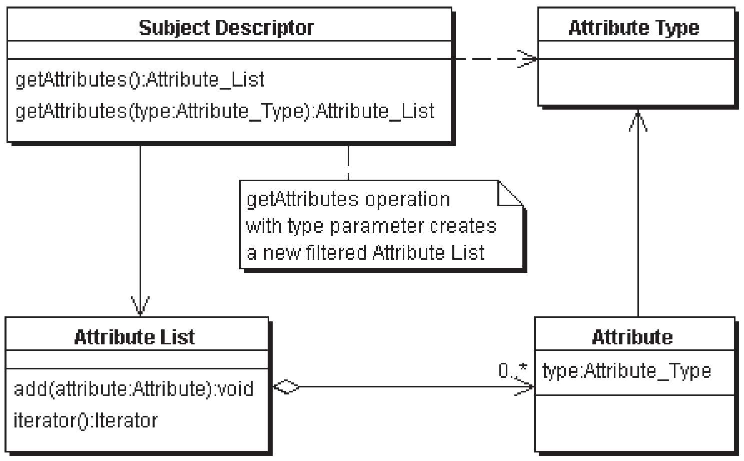
Subject Descriptor provides access to subject attributes and facilitates management and protection of those attributes, as well as providing a convenient abstraction for conveying attributes between subsystems. For example, an authentication subsystem could establish subject attributes including an assertion of a user's identity which could then be consumed and used by a separate authorization subsystem.

Applicability

Use the Subject Descriptor pattern when:

- A subsystem responsible for checking subject attributes (for example, rights or credentials) is independent of the subsystem which establishes those attributes.
- Several subsystems establish attributes applying to the same subject.
- Different types or sets of subject attributes may be used in different contexts.
- Selective control of access to particular subject attributes is required.
- Multiple subject identities need to be manipulated in a single operation.

Structure


Subject Descriptor
getAttributes():Attribute_List
getAttributes(type:Attribute_Type):Attribute_List
Attribute Type
getAttributes operation with type parameter creates a new filtered Attribute List
Attribute List
add(attribute:Attribute):void
iterator():Iterator
0..*
Attribute
type:Attribute_Type

Participants

- Subject Descriptor

  Encapsulates a current set of attributes for a particular subject.

  Supports operations to provide access to the complete current set of attributes, or a filtered subset of those attributes.

- Attribute List

  Controls access to and enables management of a list of attributes for a subject.

  A new Attribute List can be created to reference a filtered subset of an existing set of attributes.

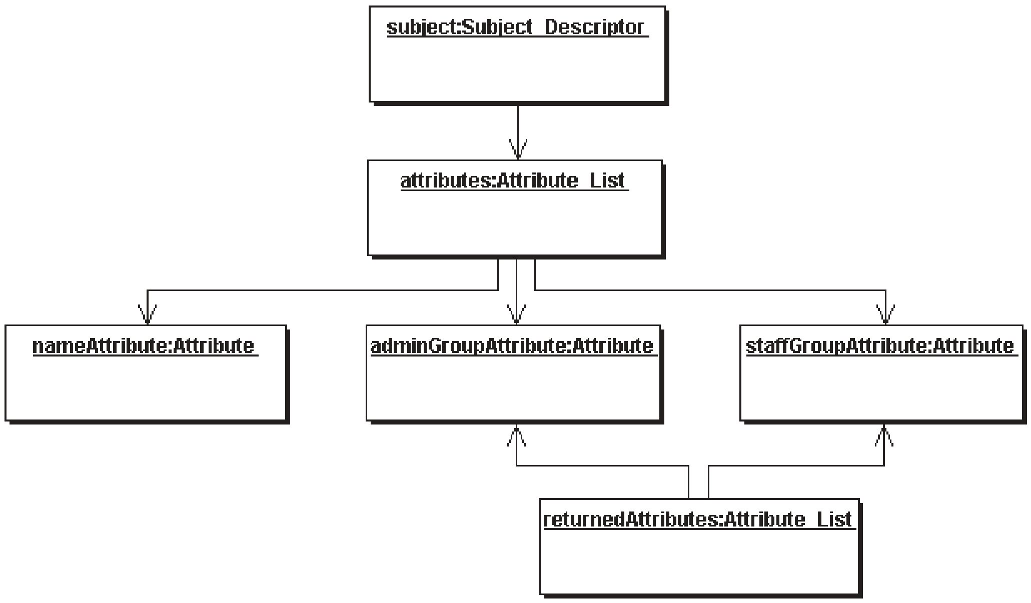
- Attribute

  Represents a single security attribute.

- Attribute Type

  Allows related attributes to be classified according to a common type.

The following object diagram shows an example instantiation of a Subject Descriptor, with its internal Attribute List, and a constructed Attribute List referencing a subset of the Attributes.


subject:Subject Descriptor
attributes:Attribute List
nameAttribute:Attribute
adminGroupAttribute:Attribute
staffGroupAttribute:Attribute
returnedAttributes:Attribute List

Collaborations

Attribute List returns an Iterator [GoF] allowing the caller to operate on the individual Attributes referenced in the list.

Attribute List may be a Guarded Type (see the Protected System pattern above), consulting Policy in order to determine whether the caller is permitted to access attributes within the list. A filtered Attribute List can be a way for a caller to pre-select only those attributes which it is permitted to access.

Consequences

Use of the Subject Descriptor pattern:

- Encapsulates subject attributes

  Subject Descriptor allows a collection of attributes to be handled as a single object. New types of attributes can be added without modifying the Subject Descriptor or code which uses it.

- Provides a point of access control

  Subject Descriptor allows construction of Attribute Lists including access control functionality to ensure that unauthorized callers will not have access to confidential attributes (such as authentication tokens).

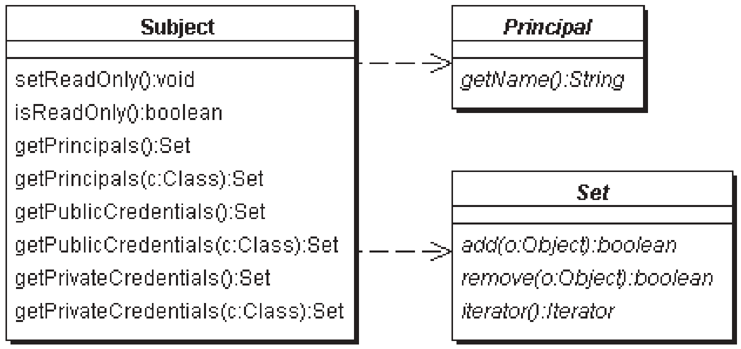
Implementation

When implementing Subject Descriptor, it may be helpful to choose a hierarchical representation for the attribute type. This helps extensibility in that you can have broad categories of attributes (for example, ‘‘identity’’ for all attributes which are some type of name) which can be subdivided into more specific categories (for example, ‘‘group identity’’, or even more specific ‘‘UNIX group ID number’’). Callers can then select attributes at varying levels of abstraction choosing which is most suitable for their specific purpose.

Class names are a ready-made hierarchy which may be suitable.

Known Uses

1. JAAS (Java Authentication and Authorization Service) javax.security.auth.Subject

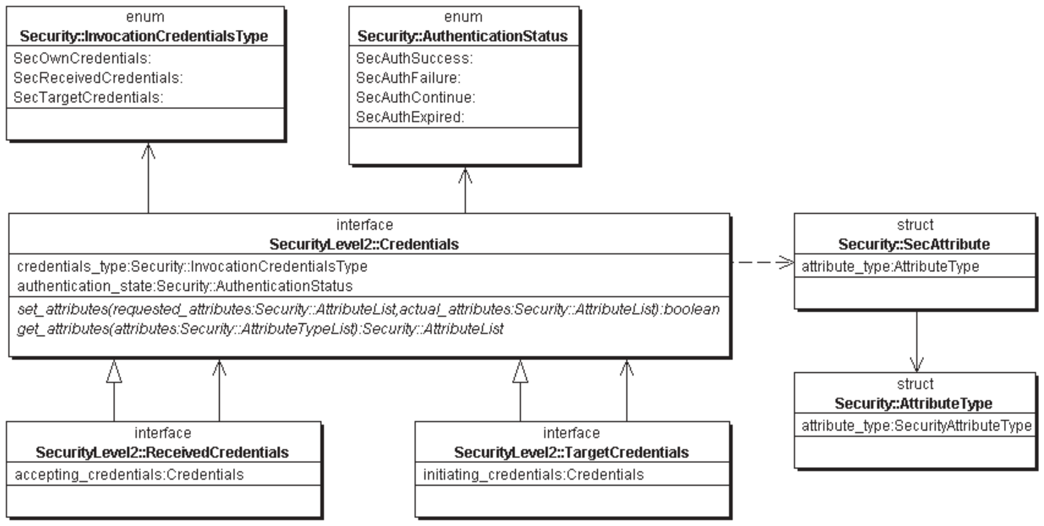


JAAS divides the subject attributes into three collections: principals, public credentials, and private credentials. Principals (which might be better called identities, but the class name ‘‘Identity’’ was already taken) are used to represent user identities and also groups and roles. There is a defined interface to Principal objects, allowing a name to be retrieved without requiring the specific implementing class to be known. Public and private credentials, on the other hand, are arbitrary Java objects and have no defined interface.

Principals and public credentials may be retrieved by any caller which has a reference to the Subject object. Private credentials require a permission to be granted in order to access them, which may be specified down to the granularity of a particular credential object class within Subjects having a particular Principal class with a particular name.

The JAAS Subject class includes a method to set a read-only flag which specifies that the Sets of Principals returned will be read-only (that is, the *add*( ) and *remove*( ) methods will fail). This is useful where a privileged caller gets a reference to a Subject object which it then wishes to pass on to an untrusted recipient.

2. CORBASecurity SecurityLevel2::Credentials


enum
Security::InvocationCredentialsType
SecOwnCredentials:
SecReceivedCredentials:
SecTargetCredentials:
enum
Security::AuthenticationStatus
SecAuthSuccess:
SecAuthFailure:
SecAuthContinue:
SecAuthExpired:
interface
SecurityLevel2::Credentials
credentials_type:Security::InvocationCredentialsType
authentication_state:Security::AuthenticationStatus
set_attributes(requested_attributes:Security::AttributeList,actual_attributes:Security::AttributeList):boolean
get_attributes(attributes:Security::AttributeTypeList):Security::AttributeList
struct
Security::SecAttribute
attribute_type:AttributeType
struct
Security::AttributeType
attribute_type:SecurityAttributeType
interface
SecurityLevel2::ReceivedCredentials
accepting_credentials:Credentials
interface
SecurityLevel2::TargetCredentials
initiating_credentials:Credentials

CORBASecurity credentials lists encapsulate subject attributes. CORBASecurity associates a set of credentials with each execution context; *OwnCredentials* represent the security attributes associated with the process itself; *ReceivedCredentials* represent the security attributes associated with a communications session within which the process is the receiver; and *TargetCredentials* represent the security attributes which will be used to represent the process to a partner in a communications session within which the process is the sender.

Related Patterns

- Security Context [TG_SDP] uses Subject Descriptor to represent the attributes of subjects.
- Subject Descriptor uses Iterator [GoF] to return a subject's attributes to callers.
- Role-based access control [APLRAC].
- Security models, authorization, role-based access, multi-level security [Fernandez-Pan].
- The role object pattern [Role Object].
- Enabling application security, roles [Yoder-Barcalow].

8.5 Secure Communication

Intent

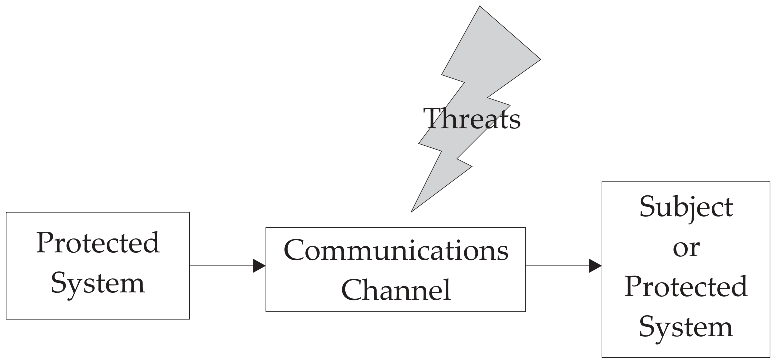
Ensure that mutual security policy objectives are met when there is a need for two parties to communicate in the presence of threats.

Also Known As

None known.

Motivation

A communications channel between two Protected Systems or between a subject and a Protected System may be subject to various security threats. The security provided by the sending Protected System will not be effective if it can be subverted by attacks on the communications channel. Therefore it may be desirable or imperative to protect the channel.



Threats against the communications channel may include:

- Unauthorized disclosure of traffic
- Impersonation of a party to the communication
- Unauthorized modification of traffic
- Diversion or interdiction of traffic

The Secure Communication pattern protects against threats by employing security countermeasures to protect traffic in the communications channel.

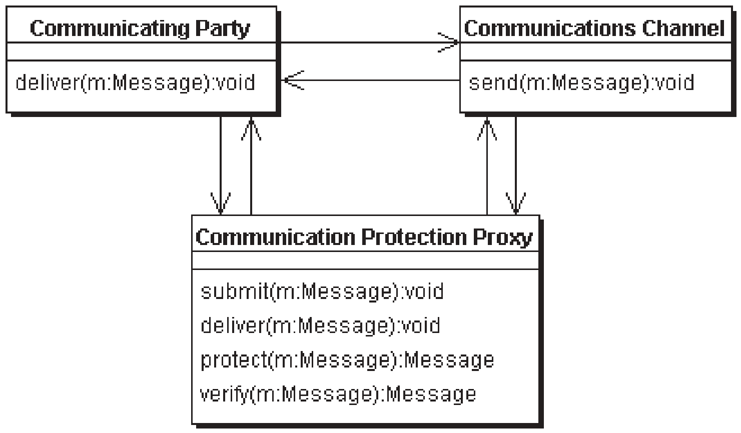
Applicability

Consider using the Secure Communication pattern when:

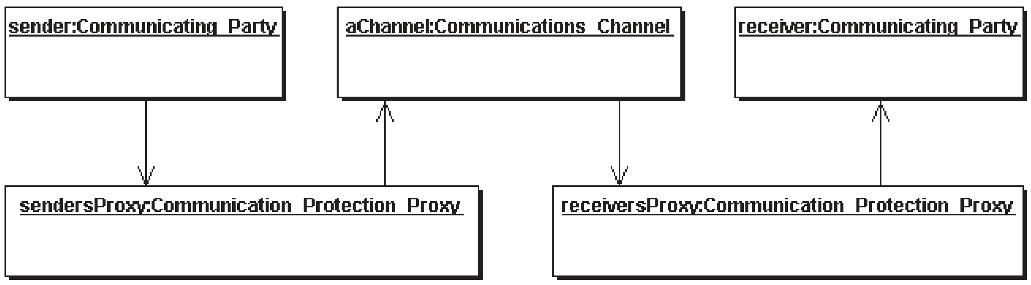
- A Protected System needs to communicate sensitive information with subjects or with other Protected Systems over a communications channel.
- Traffic in the communications channel may be subject to security threats.

Structure

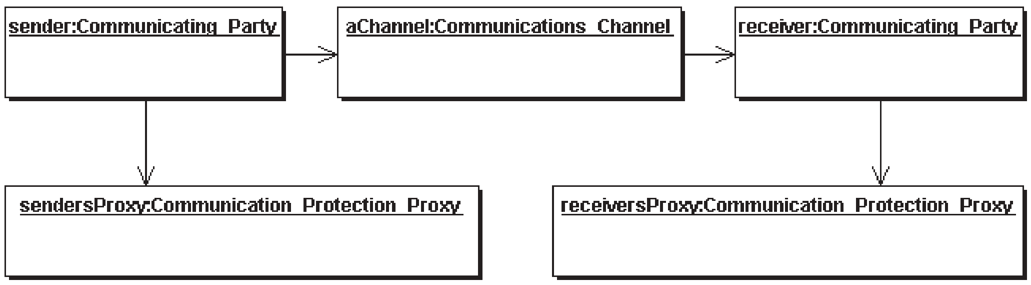
The Secure Communication Pattern has two structural variants.



In the first variant, a Communication Protection Proxy is an inline proxy between the sender/receiver and the communications channel.


sender:Communicating Party
aChannel:Communications Channel
receiver:Communicating Party
sendersProxy:Communication Protection Proxy
receiversProxy:Communication Protection Proxy

In the second variant, a Communication Protection Proxy is an out-of-band service which is used by senders and receivers to protect traffic which they submit to or receive from the Communications Channel. (Note that this variant is more appropriate for use with non-session-oriented or store-and-forward communication protocols.)


sender:Communicating Party
aChannel:Communications Channel
receiver:Communicating Party
sendersProxy:Communication Protection Proxy
receiversProxy:Communication Protection Proxy

Participants

- Communicating Party

  The source and/or destination of messages to be sent over a communications channel.

- Communications Channel

  Carries information exchanged between a message sender and receiver.

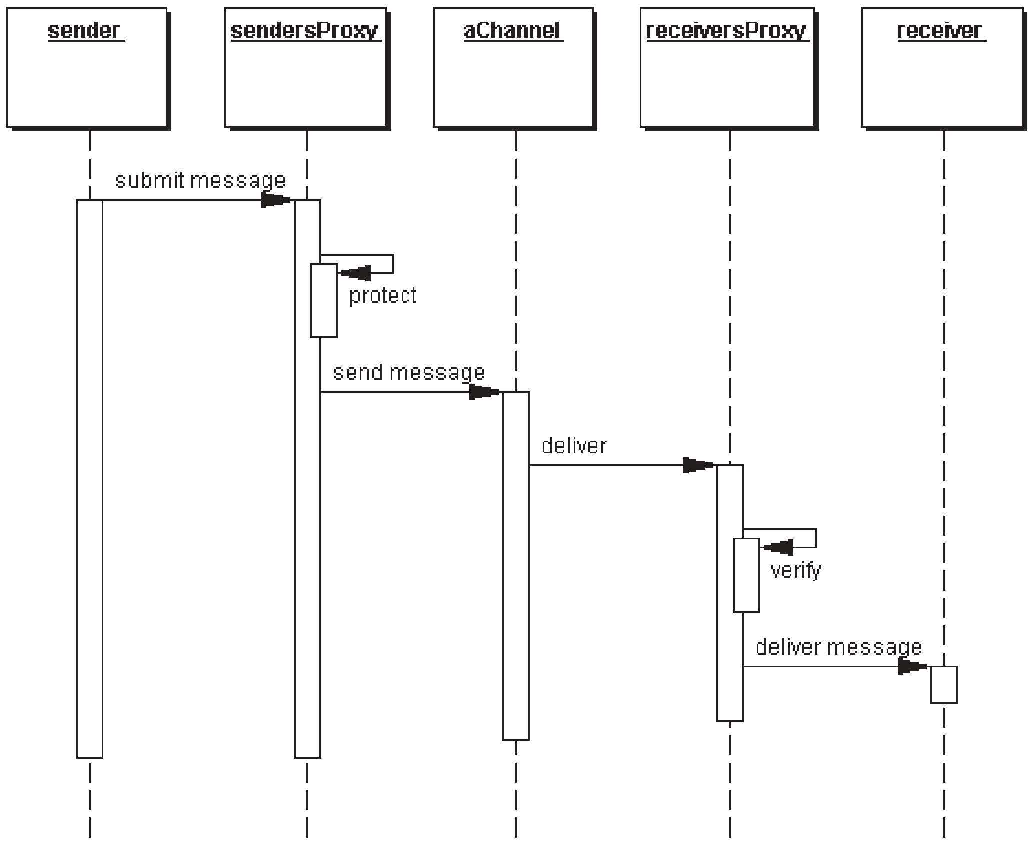
- Communication Protection Proxy

  Protects traffic sent over the communications channel using one of a variety of protection mechanisms.

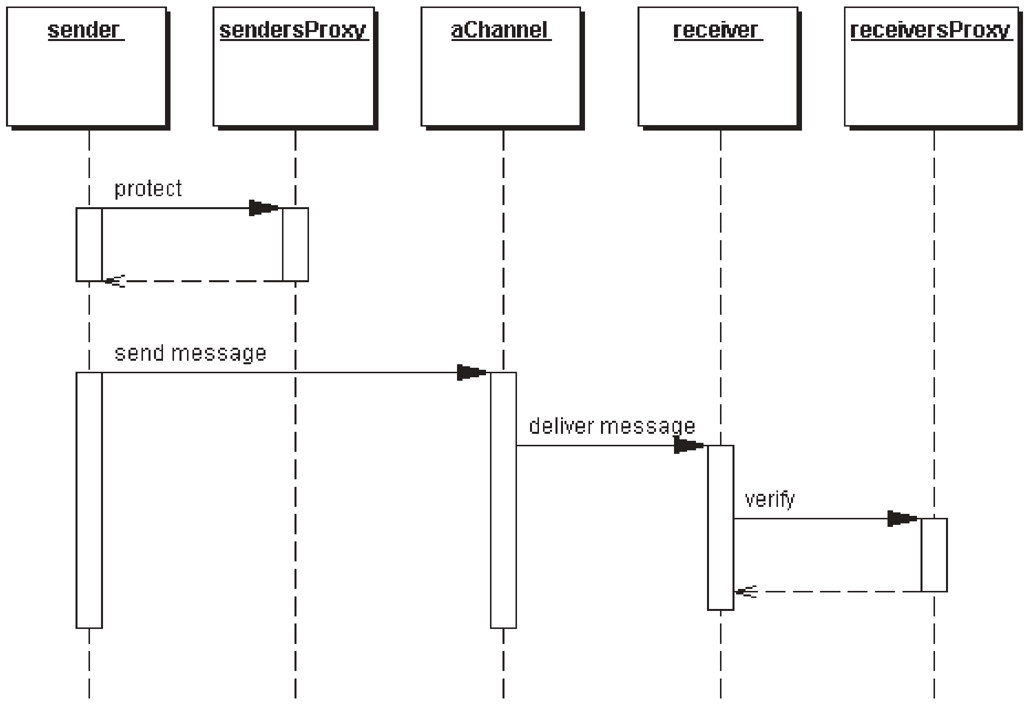
Collaborations

- A sending Communicating Party submits a message to its Communication Protection Proxy for protection.
- The Communication Protection Proxy applies appropriate protection to the message.
- If the Communication Protection Proxy is functioning as an inline proxy (variant 1 above) then it uses the Communications Channel to transmit the message to the Communication Protection Proxy of the receiving Communicating Party.
- If the Communication Protection Proxy is functioning as an out-of-band service (variant 2 above) then it returns the protected message to the sending Communicating Party, which uses the Communications Channel to transmit the protected message to the receiving Communicating Party.
- The receiver obtains messages sent over the Communications Channel; if the receiver's Communication Protection Proxy is serving as an inline proxy, then the message's protection will already have been verified and any necessary decryption will already have been done by the Communication Protection Proxy. If the receiver's Communication Protection Proxy is serving as an out-of-band service, then the receiver will pass the protected message to its Communication Protection Proxy, which will verify the message's protection, do any necessary decryption, and return the verified message to the receiver.

The figure below illustrates the interactions for variant 1 (inline proxy) of the Secure Communication pattern.



The figure below illustrates the interactions for variant 2 (out-of-band service) of the Secure Communication Proxy.



Consequences

Use of the Secure Communication pattern:

- Ensures that data communicated over a potentially insecure communication channel is protected against a known set of threats.
- May reduce communications throughput or increase communications latency.
- May require the use of cryptography (and therefore may require consideration of international deployment issues related to cryptography).
- May interfere with the use of other services (for example, content scanners, proxies, filtering routers) which depend on access to message content between communications endpoints.

Implementation

Secure Communication Proxies may need to apply one or more of the following types of protection to messages in order to counter threats anticipated in the Communications Channel:

- Data Origin Authentication protects against misrepresentation of the identity of a sender of a message.
- Peer Entity Authentication protects against impersonation of parties to the communication.
- Data Integrity protects against undetected, unauthorized modification of data in transit in the communications channel. Data integrity services may provide additional services, including:
  — Replay detection

    The ability to detect that some unauthorized party that captured a sequence of communication exchanges subsequently tried to replay that exchange.

  - — Sequence ordering

    The ability to detect missing or reordered elements of a communication.

- Data Confidentiality protects against disclosure of message contents to unauthorized parties.

One or more of following mechanisms is used to implement the protection features listed above:

- Cryptography
- Cryptographic Key Management
- Hash Functions
- Secure Protocol Handshake Exchanges

For the data content that is to be passed across a communication channel the pattern implementor will need to identify:

1. The protection services and mechanisms that need to be applied in the context of a security policy appropriate to use of the communication channel, and the strength of mechanisms which will be required to counter anticipated threats.
2. The granularity of protection services and mechanisms to be applied (for example, whether protection characteristics will be able to be changed on a per-message basis or only on a per-session basis).
3. What key management and key exchange functionality will be required to support the necessary protection services and mechanisms.

Known Uses

The following are instances of Secure Communication (variant 1, inline proxy):

- Secure Sockets Layer (SSL) and Transport Layer Security (TLS) [IETF RFC 2246 and others]
- Internet Protocol Security [IETF RFC 2401 and others]
- IEEE Standard for Interoperable LAN / MAN (SILS) [IEEE Std 802.10-1998]

The following are instances of Secure Communication (variant 2, out-of-band service):

- Secure Multipurpose Internet Mail Extensions (S / MIME) [IETF RFC 2311]
- Cryptographic Message Syntax [IETF RFC 2630]
- Generic Security Service (GSS-API) [IETF RFC 1508 and others]
- Independent Data Unit Protection (IDUP-GSS-API) [IETF RFC 2479]

Related Patterns

- Protected System [TG_SDP] instances use Secure Communication to protect messages transmitted between their guards.
- Secure Communication uses Security Association [TG_SDP] to store state information about the security parameters to be used to protect messages.
- Single sign-on, role-based access control [APLRAC].
- Authenticated session [NAI].
- Pattern language for cryptographic software [Tropyc].
- Enabling application security, session [Yoder-Barcalow].

8.6 Security Context

Intent

Provide a container for security attributes and data relating to a particular execution context, process, operation, or action.

Also Known As

None known.

Motivation

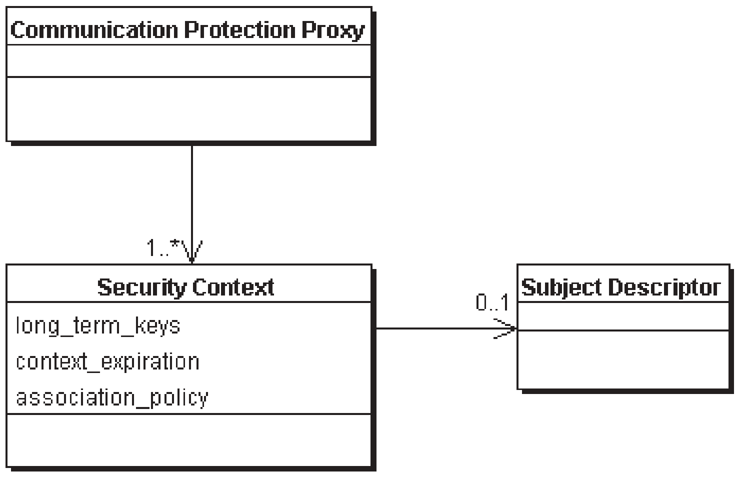
When a single execution context, program, or process needs to act on behalf of multiple subjects, the subjects need to be differentiated from one another, and information about each subject needs to be made available for use. When an execution context, program, or process needs to act on behalf of a single subject on multiple occasions over a period of time, it needs to be able to have access to information about the subject whenever it needs to take an action. The Security Context pattern provides access to subject information in these cases.

Applicability

Use the Security Context pattern when:

- A process or execution context acts on behalf of a single subject over time but needs to establish secure communications with a variety of different partners on behalf of this single subject.
- A process or execution context is able to act on behalf of different subjects and needs to manage which subject is currently active.

Structure



Participants

- Communication Protection Proxy

  Responsible for establishing Security Associations; used by Secure Communication to apply protection described in Security Association to messages.

- Security Context

  Stores information about a single subject, including secret attributes such as long-term keys to be used to establish Security Associations. A Communication Protection Proxy may create and retain several security contexts simultaneously, but it must always know which Security Context is active (that is, will be used to establish Security Associations).

- Subject Descriptor

  Stores the identity-related attributes of a subject.

Collaborations

Whenever a process becomes active in an execution context, the execution context's Communication Protection Proxy creates an instance of Security Context and populates it with the necessary information about the process. The execution context may perform some authentication challenge to verify the identity of the subject before creating a Security Context; the execution context may also set an expiration time for the Security Context to ensure that it is not re-used by a party other than the subject it refers to.

Consequences

Use of the Security Context pattern:

- Encapsulates security attributes relating to a process and user.

  Use of Security Context allows a user's security attributes, cryptographic keys, and process security attributes to be handled as a single object.

- Provides a point of access control.

  The Security Context will include attributes or accessors allowing callers to retrieve extremely sensitive information (such as long-term cryptographic keys belonging to the subject). This information must be protected against disclosure or misuse.

Implementation

As noted above, the Security Context implementation will need to protect the sensitive information contained within it.

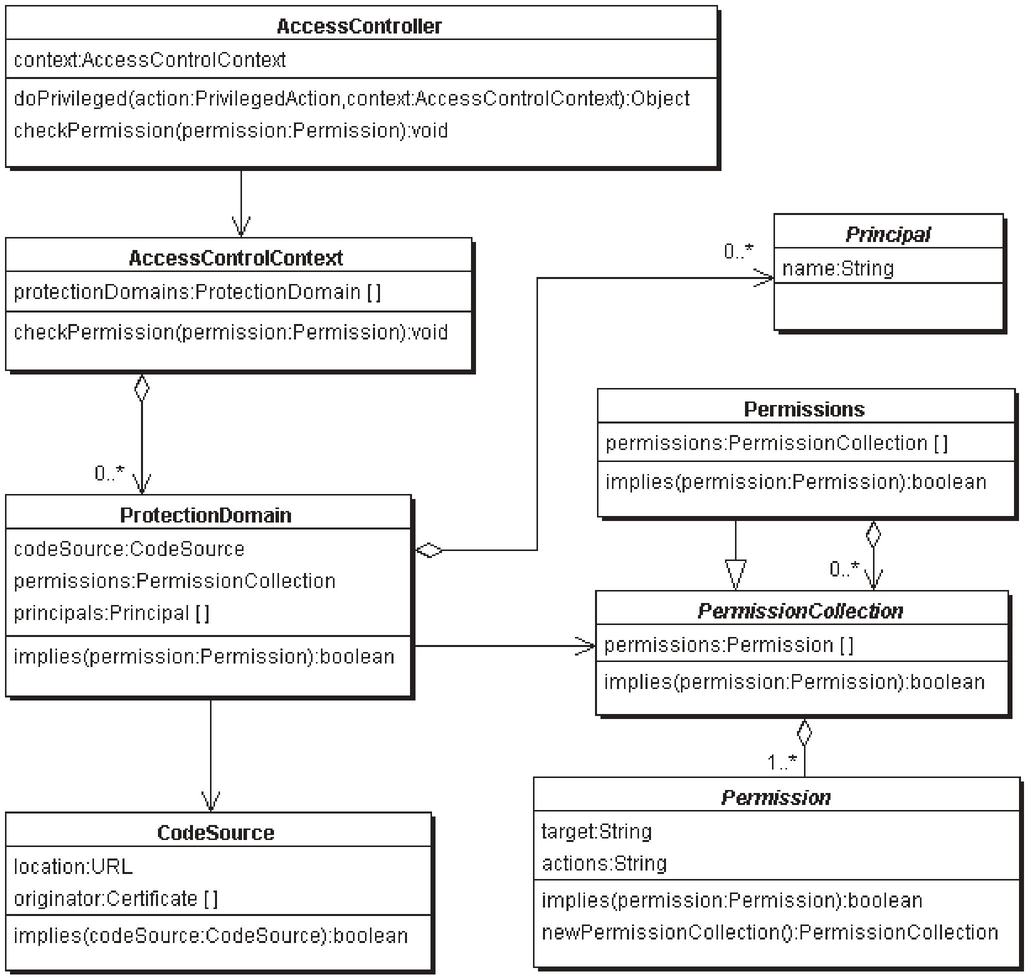
Access control can be implicit, if the system is architected such that only authorized callers can obtain a reference to a Security Context. If it is possible for unauthorized callers to discover references to Security Contexts, the implementation will need to provide accessors which check the authorization of the caller before returning sensitive information (see the Guard class in the Protected System pattern).

Known Uses

1. UNIX — Per-process User Information (‘‘u area’’)

   The UNIX process table includes a ‘‘u area’’ which stores the identity of the logged-on user as well as the identity of an ‘‘effective user’’; the real user and the effective user are the same unless the user identity has been modified by executing a *setuid* operation. Retention of the real user ID allows switching back to the user’s original account after performing operations under the effective (*setuid*) identity.

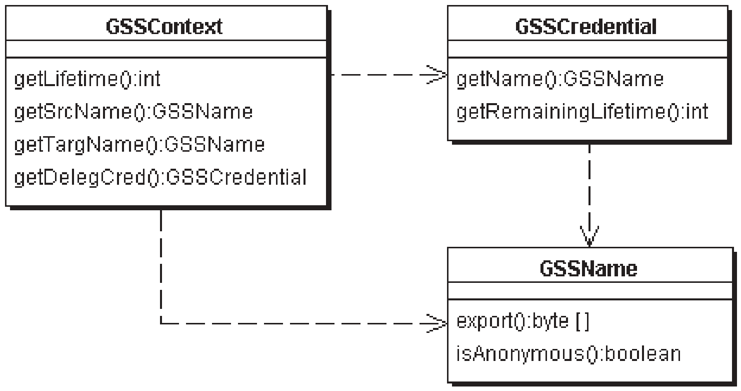
2. Java 2 Standard Edition — java.security.AccessControlContext



AccessController
context:AccessControlContext
doPrivileged(action:PrivilegedAction,context:AccessControlContext):Object
checkPermission(permission:Permission):void
AccessControlContext
protectionDomains:ProtectionDomain []
checkPermission(permission:Permission):void
Principal
name:String
0..*
Permissions
permissions:PermissionCollection []
implies(permission:Permission):boolean
0..*
ProtectionDomain
codeSource:CodeSource
permissions:PermissionCollection
principals:Principal []
implies(permission:Permission):boolean
0..*
PermissionCollection
permissions:Permission []
implies(permission:Permission):boolean
1..*
CodeSource
location:URL
originator:Certificate []
implies(codeSource:CodeSource):boolean
Permission
target:String
actions:String
implies(permission:Permission):boolean
newPermissionCollection():PermissionCollection

The Java2 Access Control Context records the identity of the source of the executing code, together with the identity of the active user. The code source is recorded in a *ProtectionDomain* object, while the user identity is stored in a *Principal* object.

3. GSS-API — org.ietf.jgss.GSSContext



What GSS-API calls a ‘‘Security Context’’ is an instance of our Security Association pattern. The GSS-API structure which instantiates the Security Context pattern is the GSS Credential, which records the name and cryptographic key of the subject, together with an indication of whether the GSS Credential can be used to initiate outgoing GSS Security Contexts, or only to accept incoming GSS Security Contexts.

4. CORBA — SecurityLevel2::Current

   CORBASecurity’s Current object (which represents an execution context) creates and stores three CORBA Credential objects; these objects are instances of Security Context; each Credential object contains information about a subject; the *InvocationCredential* object always refers to the active subject, and it is used by the Communications Protection Proxy (called a Security Interceptor) of the CORBA ORB (which is an instance of our Secure Communication pattern) to create CORBASecurity Context objects (which are instances of our Security Association pattern).

Related Patterns

Security Context uses Subject Descriptor [TG_SDP] to store identity-related information about subjects.

- Secure Association [TG_SDP] uses Security Context to store information about subjects and processes.

8.7 Security Association

Intent

Define a structure which provides each participant in a Secure Communication with the information it will use to protect messages to be transmitted to the other party, and with the information which it will use to understand and verify the protection applied to messages received from the other party.

Also Known As

None known.

Motivation

Instantiating the Secure Communication pattern to protect messages in a communications channel is expensive and often slow, because it requires cryptographic operations to authenticate partners and exchange keys, and it often requires negotiating which protection services need to be applied to the channel. When two parties want to communicate securely they often want to send more than one message, but the cost of creating an instance of Secure Communication for each message would be prohibitive. Therefore it is desirable to enable an instance of Secure Communication to protect more than one message. Doing this requires storing a variety of security-related state information at each end of the communications channel. The Security Association pattern defines what state information needs to be stored, and how it is created during the establishment of an instance of the Secure Communication pattern.

Applicability

Use this pattern when:

- The Secure Communication pattern is used to protect messages in a communications channel.
- Some security parameters of the Secure Communication pattern are established by negotiation each time communication is initiated, rather than being pre-configured at each endpoint of the communication link out-of-band.
- It is desirable to send multiple messages over a secure communications channel without re-negotiating the security parameters of the channel for each message.

Structure

A Security Association may contain some or all of the following information:

- Association Identifier

  Used to distinguish this instance of the Security Association pattern from other instances.

- Partner Identifier

  Used to identify the entity with which this instance of the Security Association pattern enables communication.

- Association Expiration

  The time after which the instance of the Security Association pattern is no longer valid and must not be used to protect messages.

- Cryptographic Keys

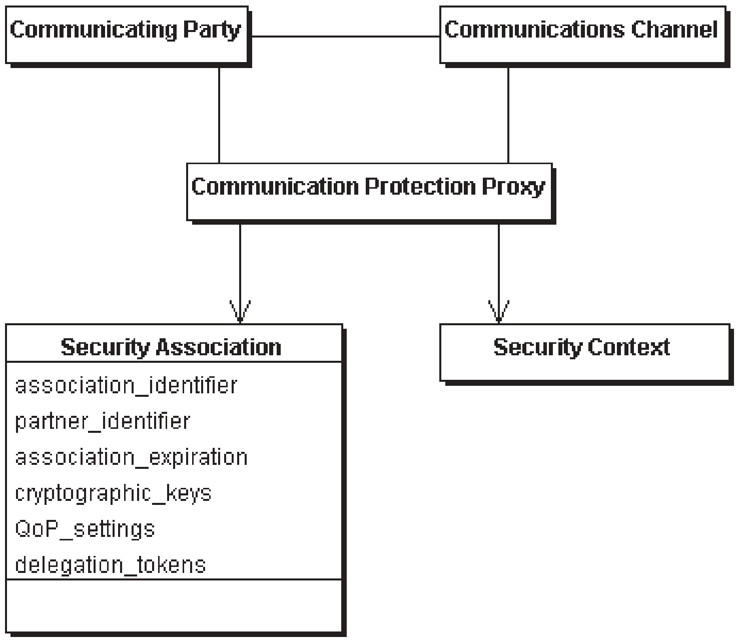
  Used by the Secure Communication pattern owning this instance of Security Association to protect messages.

- Quality of Protection (QoP) Settings

  Used by the Secure Communication pattern to determine which security services need to be applied to messages.

- Delegation Tokens

  Used by the Secure Communication pattern to implement delegation functionality.



Participants

- Protection Proxy

  Creates Security Associations and protects messages using information in Security Associations.

- Security Association

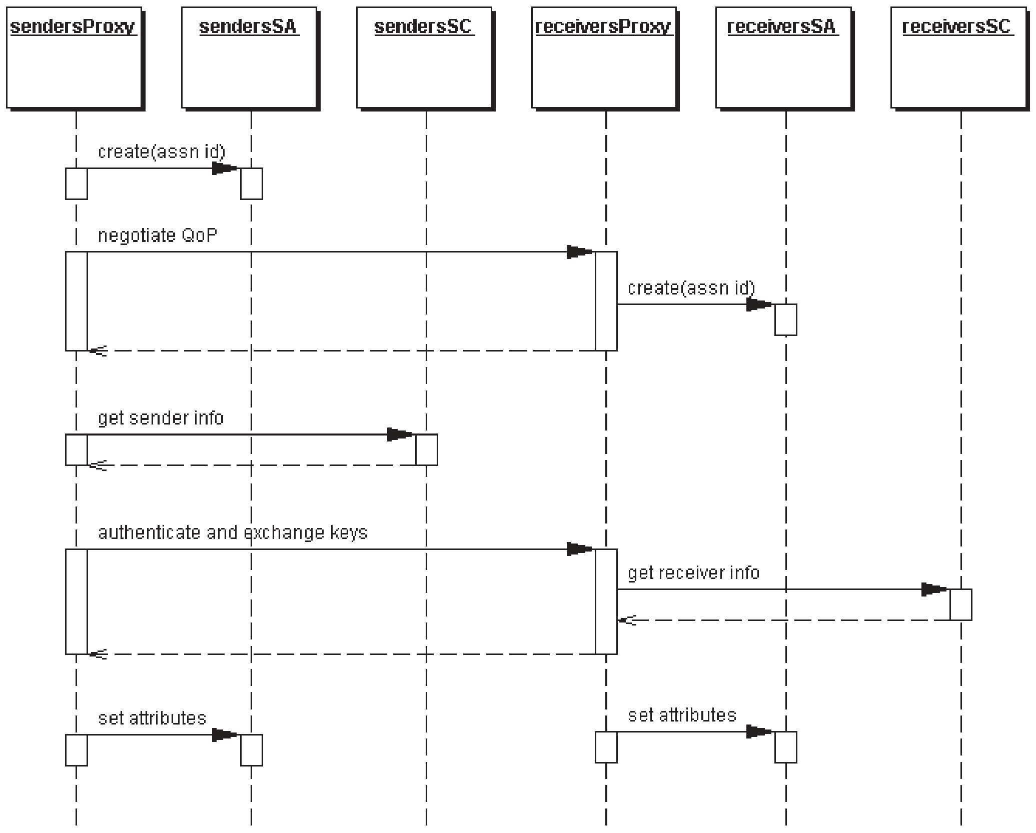
  Defines parameters used to protect messages.

- Security Context

  Contains information used to set up Security Association.

Collaborations

- Each Protection Proxy creates an instance of Security Association and assigns it a unique Association Identifier.
- The Protection Proxies determine the required QoP by reading configuration information or by negotiation with one another.
- If necessary, the Protection Proxies authenticate partner identifiers.
- If necessary, the Protection Proxies exchange session keys.
- Each Protection Proxy determines an expiration time for its Security Association (this will typically be a pre-configured interval, though it might be limited by a variety of factors including remaining key lifetimes).
- The sender's Protection Proxy transmits delegation tokens to the receiver's Protection Proxy, if appropriate.



Consequences

Use of the Secure Association pattern:

- Permits re-use of a single instance of Secure Communication to protect more than one message.
- Reduces the time required to set up Secure Communications by eliminating the need to re-negotiate protection parameters and cryptographic keys.
- Creates a data structure which stores cryptographic key material; this structure needs to be strongly protected against disclosure of keys and against modification of identity information associated with keys.

Implementation

Security Association can be used to protect both session-oriented and store-and-forward message traffic, but the negotiation and key distribution mechanisms differ for the two types of messaging environments. In general, Security Association instance information can be developed via online, real-time negotiations in session-oriented protocol contexts, whereas they typically need to be derived from configuration information, target object reference information, or information in a directory or other repository in non-session-oriented protocol contexts.

Known Uses

Generalized Security Service (GSS-API) [IETF RFC 1508 and others]; the Security Association instances are called ‘‘Security Contexts’’.

OMG CORBASecurity; Security Association instances are called ‘‘Security Contexts’’.

Related Patterns

Secure Communication [TG_SDP] uses Security Association to store information used to protect message traffic.

Security Context [TG_SDP] contains information used by Secure Communication to create Security Association instances.

8.8 Secure Proxy

Intent

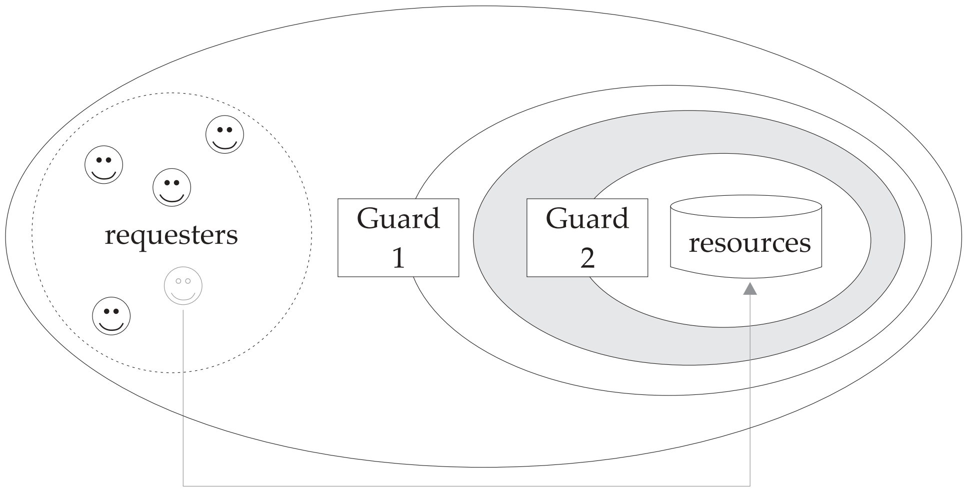
Define the relationship between the guards of two instances of Protected System in the case when one instance is entirely contained within the other.

Also Known As

Defense In Depth, Single Sign-on, Delegation, Security Protocol Encapsulation, Tunneling, Nested Protected Systems

Motivation

Security properties, especially authentication, often do not compose. Nevertheless, information systems are often built by composition. When composition results in one instance of Protected System being encapsulated inside another instance of Protected System, the requirements of both instances of Protected System need to be satisfied before resources inside the inner instance can be accessed.



A number of forces constrain solutions to this problem:

- The user would like to sign on only once.
- Both guards would like to authenticate the user.
- Both guards would like to enforce a policy based on the user's identity.
- The authentication protocol may not authenticate the user to more than one partner.
- The user may not want (or be allowed, or be able) to divulge a password or other authentication data to Guard 1.

Applicability

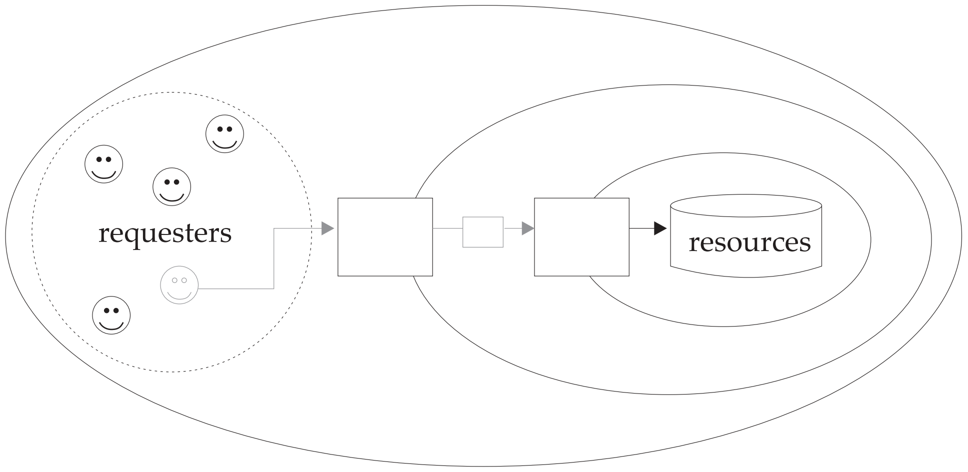
Use Secure Proxy when:

- One instance of Protected System is encapsulated inside another.

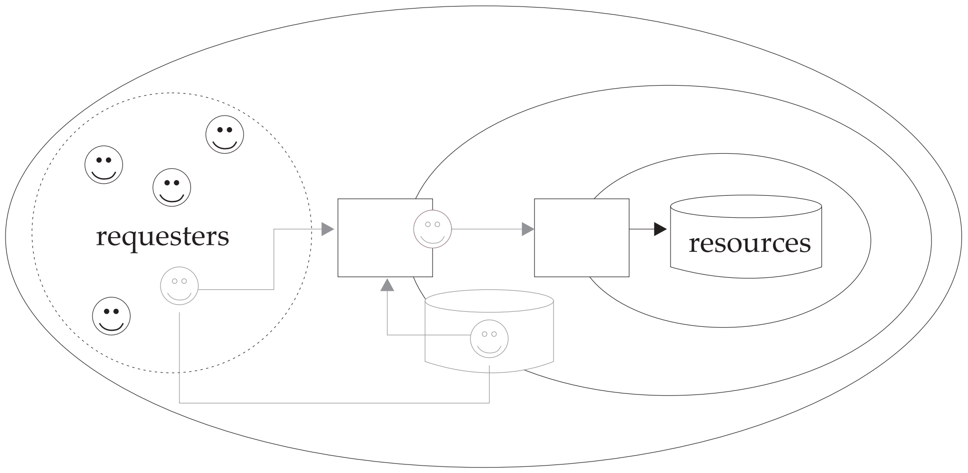
Structure

There are a number of approaches to resolving the forces described in the Motivation section; see Consequences for a description of how each of the Secure Proxy variants below addresses the various forces.

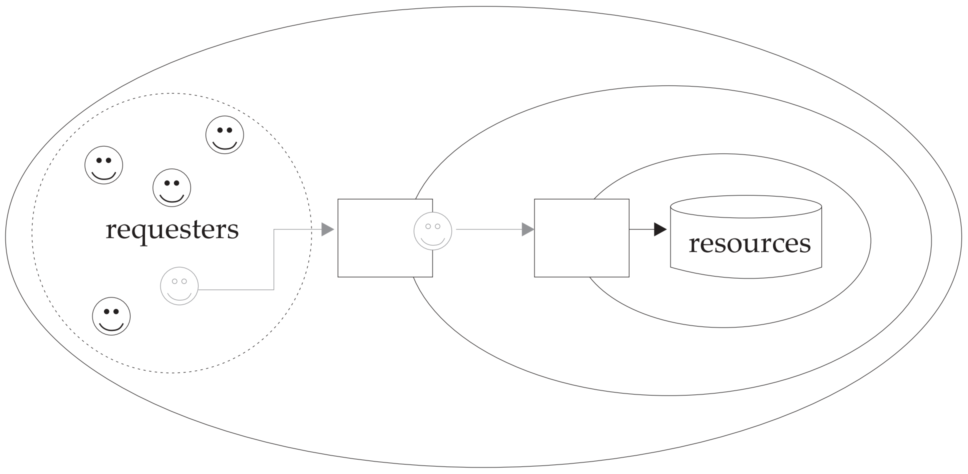
The first variant is the Trusted Proxy; in this variant, the user authenticates to Guard 1. Guard 1 then authorizes the user to access resources owned by Guard 2, and passes the request on to Guard 2 under its own (that is, Guard 1's) identity:



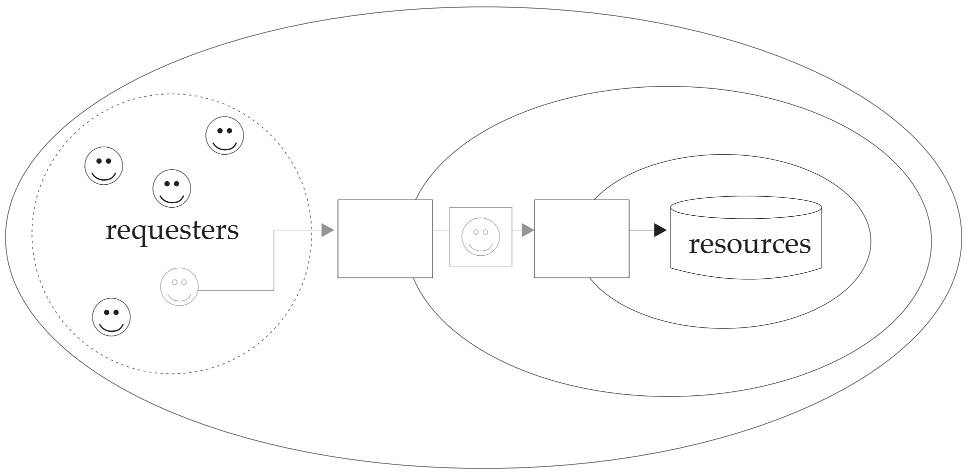
The second variant is the Authenticating Impersonator; in this variant, the user authenticates to Guard 1 and provides to Guard 1 the password (or other secret authentication data) which serves to authenticate the user to Guard 2; Guard 1 uses the user's password to authenticate ''as the user'' to Guard 2. Guard 2 authorizes the user's access to resources.



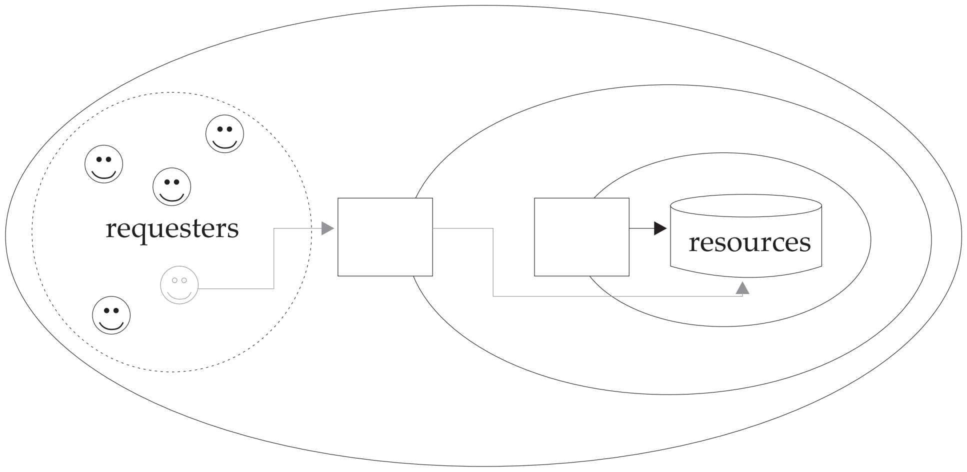
The third variant is the Identity-Asserting Impersonator. This variant is the same as the Authenticating Impersonator, except that Guard 2 ''trusts'' Guard 1 to assert the user's correct identity and does not require the presentation of a password by Guard 1. This means that Guard 1 does not need the user's password, but it does not eliminate the risk that Guard 1 will impersonate the user in unauthorized transactions with Guard 2.



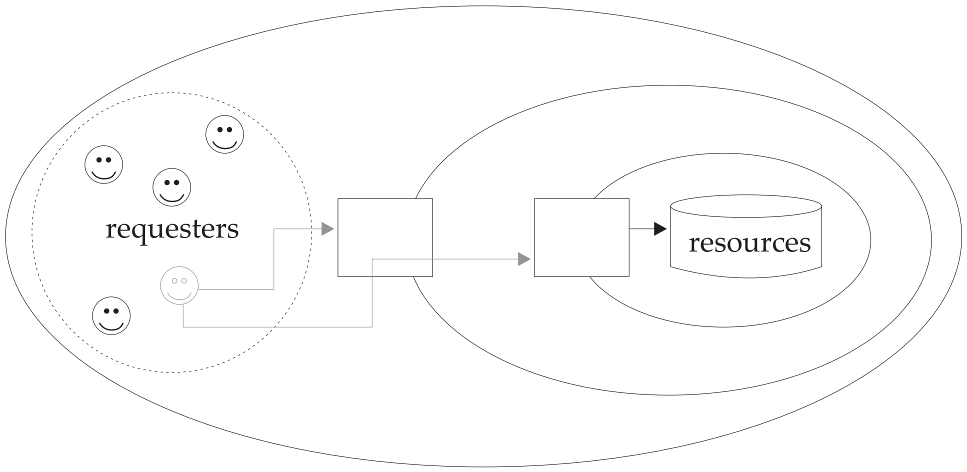
The fourth variant is the Delegate. In this variant, the security protocol shared by the user's system, Guard 1, and Guard 2, supports ''delegation''; that is, the ability of the user to authorize Guard 1 to assume the user's identity for the purposes of a single transaction with Guard 2 (and not with any other party).



The fifth variant is the Authorizing Proxy. In this variant, Guard 1 authenticates and authorizes the user, and Guard 2 simply ''steps out of the way'' and allows all requests originating from Guard 1 to pass through.



The sixth and last variant is the Login Tunnel. In this case, Guard 1 authenticates the user and then permits traffic to flow through to Guard 2 unimpeded. Guard 2 then authenticates the user for a second time and authorizes access to resources.



Participants

- User requests access to resources.
- Protected System 1 guards access to its own resources but also encapsulates Protected System 2.
- Guard 1 enforces Protected System 1's policy and authenticates users.
- Protected System 2 guards access to resources the user wishes to use.
- Guard 2 enforces Protected System 2's policy and authenticates users.

Collaborations

See Structure.

Consequences

No solution to the Secure Proxy problem is ideal; each of the variants described imposes a tradeoff between desired properties. The table below summarizes the tradeoffs.

	passwd to guard1	**userid to guard2**	**guard2 authn**	**guard2 authz**	**sso**	**delegation protocol**
ideal	no	yes	user	user	yes	no
trusted proxy	no	***no***	***guard1***	***guard1***	yes	no
authn impers	***yes***	yes	user	user	yes	no
id-assert impers	no	yes	***no***	user	yes	no
delegate	no	yes	user	user	yes	***yes***
authz proxy	no	***no***	***no***	***no***	yes	no
login tunnel	no	yes	user	user	***no***	no

The table is read as follows: the column labels describe desirable properties of a solution to the Secure Proxy problem.

- *passwd to guard1* means that the user must disclose a password or other secret authentication data to *guard1*, thus creating a risk that *guard1* will later impersonate the user in an unauthorized transaction.
- *userid to guard2* means that the user's Subject Descriptor information is provided to *guard2*, so that *guard2* can use it to make policy decisions.
- *guard2 authn* means that *guard2* authenticates a particular party — either the user, *guard1*, or no one at all.
- *guard2 authz* means that *guard2*'s authorization policy is based on Subject Descriptor information for a particular party — either the user, *guard1*, or no one at all.
- *sso* means that the user only needs to engage in one authentication dialog.
- *delegation protocol* means that the user's system, *guard1*, and *guard2* all have access to the same security protocol, and that protocol implements delegation functionality.

The first row of the table describes the ''ideal'' solution to the Secure Proxy problem; this solution meets all of the user's usability and risk mitigation requirements, meets all of *guard1*'s policy requirements, meets all of *guard2*'s policy requirements, and does not require ubiquitous implementation of complicated, inefficient, and difficult-to-manage delegation protocols.

Each subsequent row of the table describes the characteristics of one of the variant solutions to the problem described in Structure. Undesirable characteristics are highlighted in ***Bold-Italic*** font.

Implementation

Future Writer's Workshops will identify appropriate text for this section.

Known Uses

- Identity-Asserting Impersonator: IBM SNA LU6.2 ‘‘Already Verified’’
- Delegate: OSF DCE Kerberos
- Login Tunnel: Many VPN servers

Related Patterns

- Protected System [TG_SDP]
- Policy Enforcement Point [TG_SDP]
- Secure Communication [TG_SDP]
- Trusted Proxy [NAI]
- Layered Security [Romanowsky]

Glossary

API
: Application Programming Interface. The interface between the application software and the application platform, across which all services are provided. The application programming interface is primarily in support of application portability, but system and application interoperability are also supported by a communication API. See IEEE Std 1003.0 / D15.

assertion
: Explicit statement in a system security policy that security measures in one security domain constitute an adequate basis for security measures (or lack of them) in another. See CESG Memorandum No.1.

authentication
: Verify claimed identity; see data origin authentication and peer entity authentication in ISO / IEC 10181-2.

authorization
: The granting of rights, which includes the granting of access based on access rights. See ISO 7498-2.

authorization policy
: A set of rules, part of an access control policy, by which access by security subjects to security objects is granted or denied. An authorization policy may be defined in terms of access control lists, capabilities or attributes assigned to security subjects, security objects, or both. See ECMA TR / 46.

availability
: The property of being accessible and usable upon demand by an authorized entity. See ISO 7498-2.

confidentiality
: The property that information is not made available or disclosed to unauthorized individuals, entities, or processes. See ISO 7498-2.

contextual information
: Information derived from the context in which an access is made (for example, time of day). See ISO / IEC 10181-3.

data integrity
: The property that data has not been altered or destroyed in an unauthorized manner. See ISO 7498-2.

design pattern
: See Chapter 1 and *www.hillside.net*.

identification
: The assignment of a name by which an entity can be referenced. The entity may be high-level (such as a user) or low-level (such as a process or communication channel).

Hillside
: The well-established web reference site at *www.hillside.net* for information at all levels and on all aspects of design patterns.

PLoP
Pattern Languages of Programming.

privacy
The right of individuals to control or influence what information related to them may be collected and stored and by whom and to whom that information may be disclosed. Note that because this term relates to the right of individuals, it cannot be very precise and its use should be avoided except as a motivation for requiring security. See ISO 7498-2.

repudiation
Denial by one of the entities involved in a communication of having participated in all or part of the communication. See ISO 7498-2.

secure association
An instance of secure communication (using communication in the broad sense of space and/or time) which makes use of a secure context.

secure context
The existence of the necessary information for the correct operation of the security mechanisms at the appropriate place and time.

security architecture
A high-level description of the structure of a system, with security functions assigned to components within this structure. See CESG Memorandum No.1.

security attribute
A security attribute is a piece of security information which is associated with an entity.

security domain
A set of elements, a security policy, a security authority, and a set of security-relevant operations in which the set of elements are subject to the security policy, administered by the security authority, for the specified operations. See ISO/IEC 10181-1.

security policy
The set of laws, rules, and practices that regulate how assets including sensitive information are managed, protected, and distributed within a user organization. See ITSEC.

security service
A service which may be invoked directly or indirectly by functions within a system that ensures adequate security of the system or of data transfers between components of the system or with other systems.

security state
State information that is held in an open system and which is required for the provision of security services.

security vulnerabilities
The weaknesses in the construction, capability, and operation of information systems that expose them to the accidental or intentional realization of security threats.

system security function
A capability of an open system to perform security-related processing. See CESG Memorandum No.1.

target
An entity to which access may be attempted. See ISO/IEC 10181-3.

threat
A potential violation of security; see ISO 7498-2. An action or event that might prejudice

security; see ITSEC.

Security threats include unauthorized disclosure, unauthorized use of resources, denial of service, and repudiation. These threats represent security vulnerabilities. They are often evaluated by considering the methods of attack that each involves.

trojan horse

Computer program containing an apparent or actual useful function that contains additional (hidden) functions that allow unauthorized collection, falsification, or destruction of data. See Federal Criteria V1.0.

trust

A relationship between two elements—a set of operations and a security policy—in which element *X* trusts element *Y* if and only if *X* has confidence that *Y* behaves in a well-defined way (with respect to the operations) that does not violate the given security policy. See ISO / IEC 10181-1.

Index