Security Pattern Catalog

Application Firewall

Pattern documentation

Quick info

Intent: To filter calls and responses to/from enterprise applications, based on an institution access control policy.

Aliases: Content Firewall

Problem

Enterprise applications in an organization's internal network are accessed by a broad spectrum of users that may attempt to abuse its resources (leakage, modification or destruction of data). These applications can be numerous, thus implementing access control independently in ad hoc ways and may make the system more complex and thus less secure.

Moreover, traditional network firewalls (application layer firewalls or packet filters), do not make it possible to define high level rules (role-based or individual-based rules)that could make the implementation of business security policies easier and simpler.

Forces

- There may be many users (subjects)that need to access an application in different ways; the firewall must adapt to this variety.
- There are many ways to filter application inputs, we need to separate the filtering code from the application code.
- There may be numerous applications that may require different levels of security. We need to define appropriate policies for each application.
- The business policies are constantly changing and they need to be constantly updated; hence it should be easy to change the firewall filtering configuration.
- The number of users and applications may increase significantly; adding more users or applications should be done transparently and at proper cost.
- Network firewalls cannot understand the semantics of applications and are unable to filter out potentially harmful messages.

Example

Consider a medical record application in a Hospital. One of the services it provides is to allow patients to lookup their personal medical records from home. To ensure that only patients can access this service, the patient must first be authenticated and then she must be identified as a patient. Finally, the application must ensure that only the medical records belonging to the patient are returned (i. e., match the name in the medical record with that of the user).

One way to provide this security is to let application maintain list of all valid patients with their authentication credentials, and implement the code for blocking unauthorized access from within the application level code of the application. This approach has several problems. In the future, if the hospital decides to allow patients to be able to schedule appointments, it will have to repeat the implementation of the access control code for the scheduling application as well. Furthermore, if there are changes in hospital business policies, e. g. they want to allow external primary care physicians access the medical records of their own patients, these applications will have to be rewritten. In this changing scenario, a new access control list for authorized primary care physicians will have to be added to medical record application, and a list of patients will have to be associated with each physician to indicate the patients belonging to a doctor. Such application modifications are time consuming, difficult to manage, expensive and error prone.

Solution

Interpose a firewall that can analyze incoming requests for application services and check them for authorization. A client can access a service of an application only if a specific policy authorizes it to do so. Policies for each application are centralized within the Application Firewall, and they are accessed through a PolicyAuthorizationPoint. Each application is accessed by a client through a PolicyEnforcementPoint that enforces access control by looking for a matching policy in the PolicyBase. This enforcement may include authenticating the client through its identity data stored in the IdentityBase.

Structure

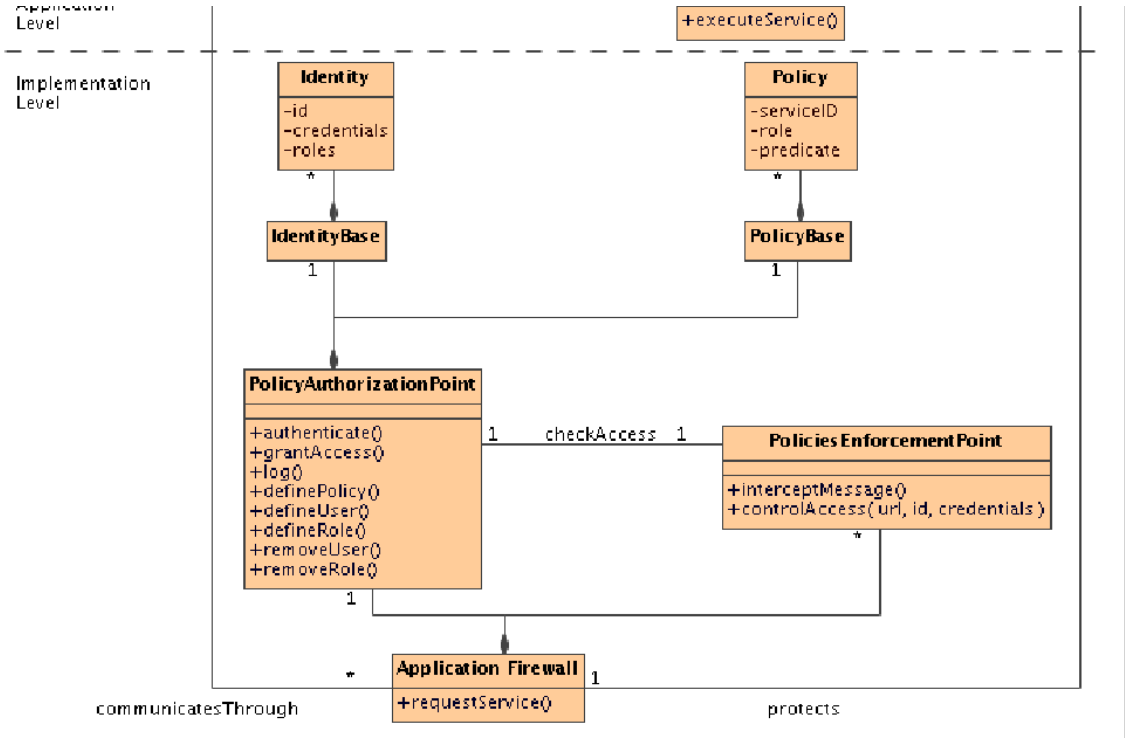


Figure 1: Application firewall class diagram.

Figure 1 shows the class diagram for the Application Firewall.

Dynamics

We describe the dynamic aspects of the Application Firewall using sequence diagrams for two use cases: filtering a Client's request with user authentication and adding a new policy.

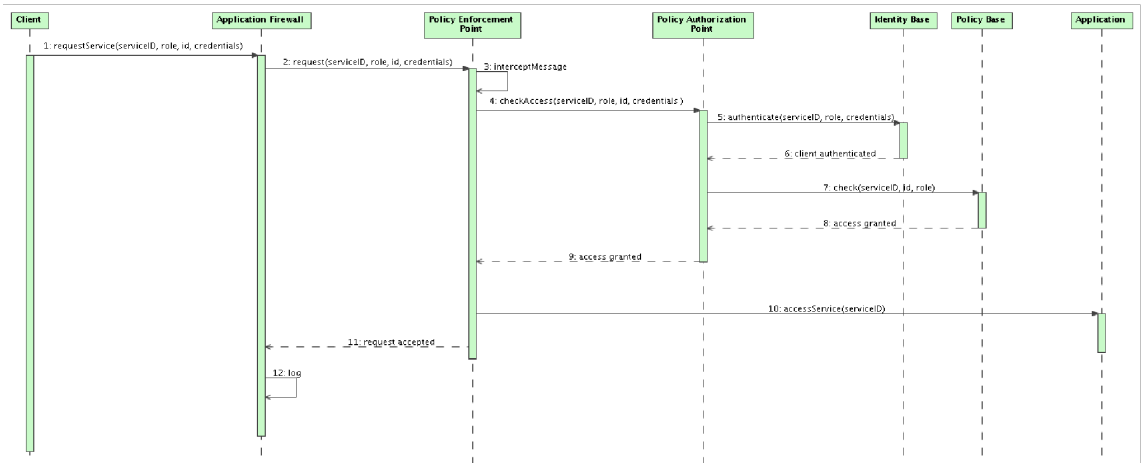


Figure 2: Application firewall sequence diagram: filtering a request.

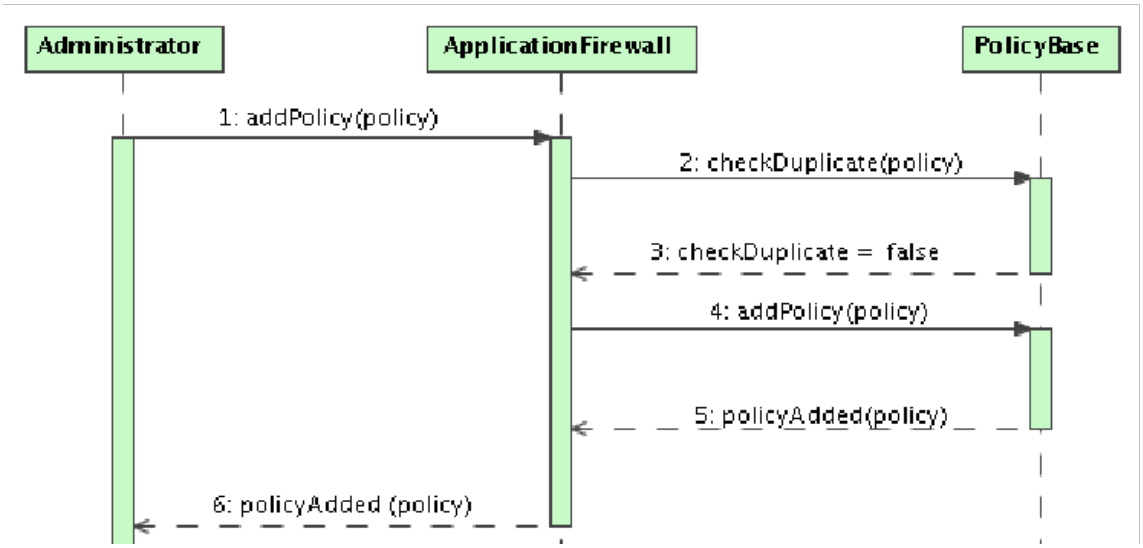


Figure 3: Application firewall sequence diagram: adding a new policy.

Participants

Classes Client and Service have the usual meaning. A Client accesses a service provided by an application. The access requests are controlled by authorization rules (denoted here as policies to follow the usual industrial notation), and represented by class Policy. Policies are collected in a policy base (PolicyBase).

The firewall consists of one PolicyAuthorizationPoint which centralizes the definition of the policies and identities throughout the institution, and several PolicyEnforcementPoints, which are intended to actually check the accesses to the applications. The enterprise applications are represented by the class Application that is made up of Services. A service is identified by a serviceId, which is usually and URI or an URL.

Collaborations

A Client requests access to a service of an application to either input or retrieve information. The access request is made through the PolicyEnforcementPoint, which accesses the PolicyAuthorizationPoint to determines whether to accept or deny the request. Figure 2 corresponds to this basic use case.

- Actors: A Client
- Precondition: Existing IdentityBase and PolicyBase classes must be in place in the firewall. The IdentityBase contains the data necessary to authenticate a Client. The PolicyBase contains specific policies defined by the organization.
- Description:
    - A Client requests access to an application.
    - An Application Firewall, through its PolicyEnforcementPoint, intercepts the request and accesses the PolicyAuthorizationPoint.
    - The PolicyAuthorizationPoint authenticates the Client through its IdentityBase. This step may be avoided for each request through the use of a Session class.
    - Once the Client is authenticated and identified, the PolicyAuthorizationPoint filters the request according to the PolicyBase. The request is accepted or denied according to the defined policies.
    - If the request is accepted, the firewall allows access to the service of the application and the access is logged into the Application Firewall.
- Alternate Flows: If the Client is not recognized or if no policy allows the specific Client to access the specified service, the firewall rejects the access request to the service. If the user has already been authenticated, the Client may not be authenticated again (Single Sign-On use).
- Postcondition: The firewall has provided the access of a Client to a service, based on verifying the identity of the Client, and the existence of a matching policy.In the case of adding a new policy: the security administrator intends to add a new policy to the set of policies. Before adding it, the firewall checks whether the new policy to be added does not already exist in the rule set. Figure 3 illustrates this use case.
- Actors: Administrator.
- Precondition: The administrator must have authorization to add rules.
- Description:
    - The administrator initiates the addition of a new rule.
    - If the rule does not already exist in the rule set then it is added.
    - The firewall acknowledges the addition of the new rule.
- Alternate Flow: The rule is not added because it already exists in the rule set.
- Postcondition: A new rule is added to the rule set of the firewall.

Implementation

To implement the Application Firewall, the following tasks need to be done:

- Define users and their roles.
- Define role rights and implement them as policies (Use Case2).
- Add/Remove policies when needed.Moreover, two architectural configurations are possible: one is where the application firewall takes the position of a reverse proxy (i. e. one firewall for a number of applications), in the other configuration each application is protected by one firewall. With the reverse proxy implementation, the input flow is intercepted on a single point. There is only one PolicyEnforcementPoint, and all the flow should go through it. With the second implementation, several PolicyEnforcementPoints are distributed on the network, close to the different applications that have to be controlled. These enforcement points intercept every request to the application. It is also possible to control access for requests coming from internal networks.

Pitfalls

(Nothing given)

Consequences

This pattern presents the following advantages:

- The institution policies to control access are easily defined and administered, as the policies have centralized administration. This makes the whole system less complex, and thus more secure.
- This firewall could be combined with an Intrusion Detection System to facilitate the prevention of some attacks.
- The firewall lends itself to a systematic logging of incoming and outgoing messages.
- As authentication of Clients is performed, users can be held responsible for their actions.
- New applications are easily integrated into the system by adding their specific policies.
- New clients can be accommodated by adding new policies to the policy base of an application.
- Because of their separation, the application and the filtering policies can evolve independently.The pattern also has some (possible)liabilities:
- The application could affect the performance of the protected system as it is a bottleneck in the network. This can be improved by considering the firewall a virtual concept and using several machines for implementation.
- The solution is intrusive for existing applications that already implement their own access control.
- The application itself must be built in a secure way or normal access to commands could allow attacks through the requests.
- We still need the operating system and the network infrastructure to be secure.

Known uses

This pattern is used in several commercial products, such as Cerebit InnerGuard, or Netegrity SiteMinder. This model is also used as an underlying architecture for XML Application Firewalls. There are also some products called Application security gateways that incorporate these functions plus others.

Audit Interceptor

Pattern documentation

Quick info

Intent: You want to intercept and audit requests and responses to and from the Business tier, in a flexible and modifyable way.

Problem

Auditing is an essential part of any security design. Most enterprise applications have security-audit requirements. A security audit allows auditors to reconcile actions or events that have taken place in the application with the policies that govern those actions. In this manner, the audit log serves as a record of events for the application. This record can then be used for forensic purposes following a security breach. That record must be checked periodically to ensure that the actions that users have taken are in accordance with the actions allowed by their roles. Deviations must be noted from audit reports, and corrective actions must be taken to ensure those deviations do not happen in the future, either through code fixes or policy changes. The most important part of this procedure is recording the audit trail and making sure that the audit trail helps proper auditing of appropriate events and user actions associated. These events and actions are often not completely understood or defined prior to construction of the application. Therefore, it is essential that an auditing framework is able to easily support additions or changes to the auditing events.

Forces

- You want centralized and declarative auditing of service requests and responses.
- You want auditing of services decoupled from the applications themselves.
- You want pre-and post-process audit handling of service requests, response errors, and exceptions.

Example

(Nothing given)

Solution

Use an Audit Interceptor to centralize auditing functionality and define audit events declaratively, independent of the Business tier services. An Audit Interceptor intercepts Business tier requests and responses. It creates audit events based on the information in a request and response using declarative mechanisms defined externally to the application. By centralizing auditing functionality, the burden of implementing it is removed from the back-end business component developers. Therefore, there is reduced code replication and increased code reuse. A declarative approach to auditing is crucial to maintainability of the application. Seldom are all the auditing requirements correctly defined prior to implementation. Only through iterations of auditing reviews are all of the correct events captured and the extraneous events discarded. Additionally, auditing requirements often change as corporate and industry policies evolve. To keep up with these changes and avoid code maintainability problems, it is necessary to define audit events in a declarative manner that does not require recompilation or redeployment of the application. Since the Audit Interceptor is the centralized point for auditing, any required programmatic change is isolated to one area of the code, which increases code maintainability.

Structure

The classes of the Audit Interceptor are depicted in Figure 1 . The Client attempts to access the Target. The AuditInterceptor class intercepts the request and uses the AuditEventCatalog to determine if an audit event should be written to the AuditLog.

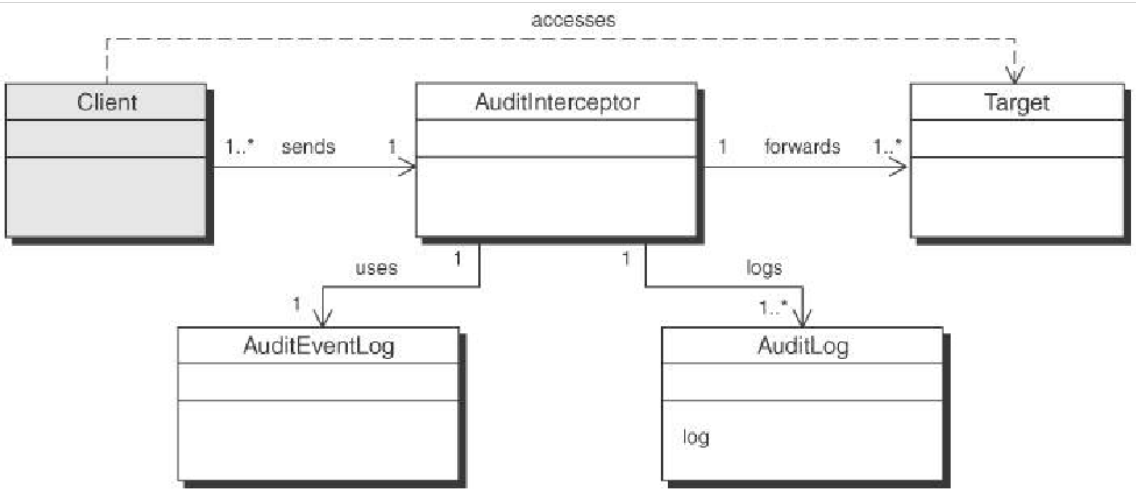


Figure 1: Class layout of the Audit Interceptor.

Dynamics

Figure 2 shows the sequence of events for the Audit Interceptor pattern. The Client attempts to access the Target, not knowing that the Audit Interceptor is an intermediary in the request. This approach allows clients to access services in the typical manner without introducing new APIs or interfaces specific to auditing that the client would otherwise not care about.

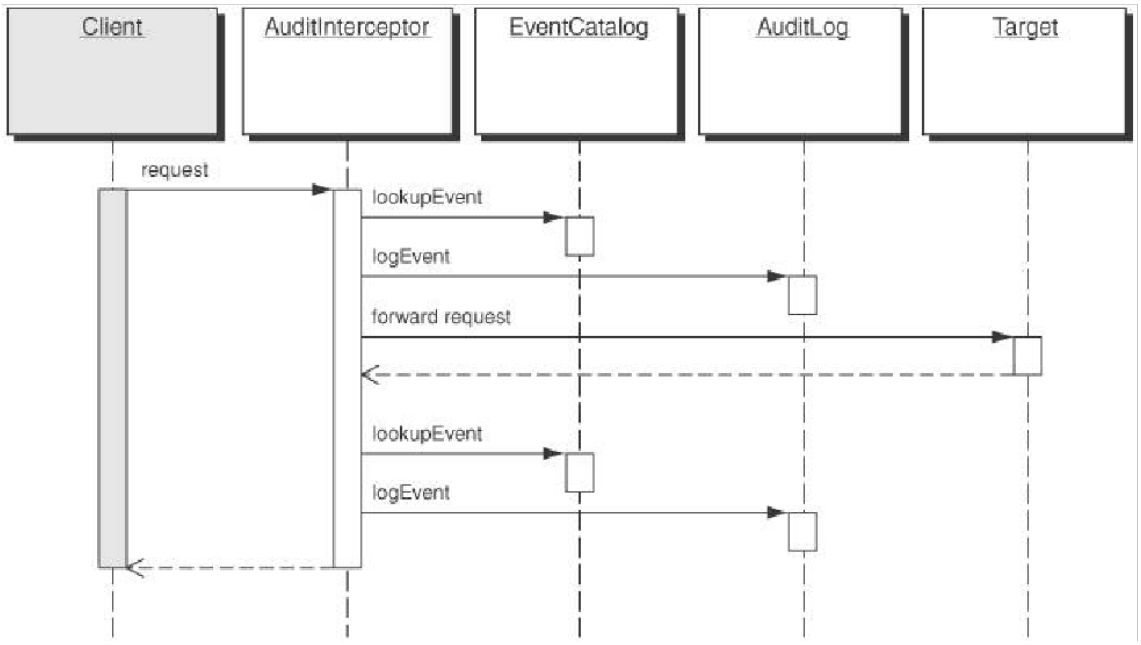


Figure 2: Event sequence for the Audit Interceptor.

The diagram in Figure 2 does not reflect the implementation of how the request is intercepted, but simply illustrates that the AuditInterceptor receives the request and then forwards it to the Target.

Participants

- Client: A client sends a request to the Target.
- AuditInterceptor: The AuditInterceptor intercepts the request. It encapsulates the details of auditing the request.
- EventCatalog: The EventCatalog maintains a mapping of requests to audit events. It hides the details of managing the life cycle of a catalog from an external source.
- AuditLog: AuditLog is responsible for writing audit events to a destination. This could be a database table, flat file, JMS queue, or any other persistent store.
- Target: The Target is any Business-tier component that would be accessed by a client. Typically, this is a business object or other component that sits behind a SessionFacade, but not the SessionFacade itself, because it would mostly be the entry point that invokes the AuditInterceptor.

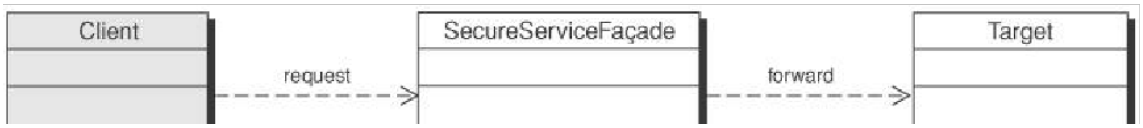
Collaborations

The Audit Interceptor pattern is illustrated in the following steps (see Figure 2):

- Client attempts to access Target resource.
- AuditInterceptor intercepts request and uses EventCatalog to determine which, if any, audit event to generate and log.
- AuditInterceptor uses AuditLog to log audit event.
- AuditInterceptor forwards request to Target resource.
- AuditInterceptor uses EventCatalog to determine if the request response or any exceptions raised should generate an audit event.
- AuditInterceptor uses AuditLog to log generated audit event.

Implementation

The Audit Interceptor requires that it be inserted into the message flow to intercept requests. The Intercepting Session Facade strategy designates the Session Facade as the point of interception for the Intercepting Auditor. The Session Facade receives the request and then invokes the Audit Interceptor at the beginning of the request and again at the end of the request. Figure 3 depicts the class diagram for the Secure Service Facade Interceptor Strategy.



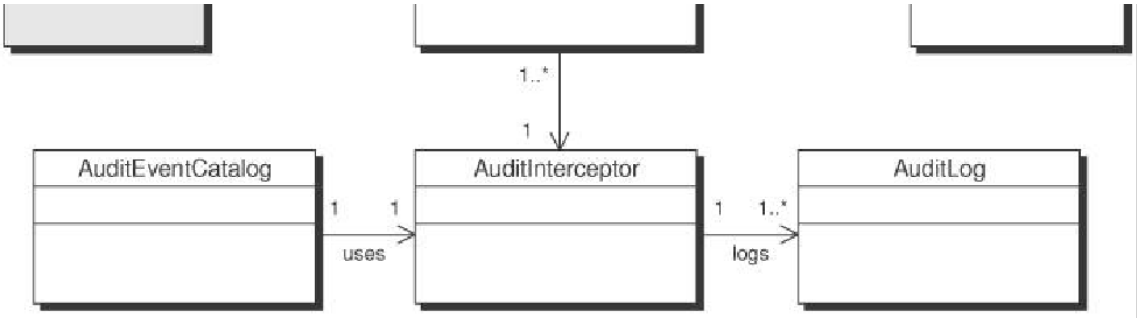


Figure 3: Secure Service Facade Interceptor strategy class diagram.

Using a Secure Service Facade Interceptor strategy, developers can audit at the entry and exit points to the Business tier. The SecureServiceFacade is the appropriate point for audit interception, because its job is to forward to the Application Services and Business Objects. Typically, a request consists of several Business Objects or Application Services, though only one audit event is required for that request. For example, a credit card verification service may consist of one Secure Service Facade that invokes several Business Objects that make up that service, such as an expiration date check, a LUN10check, and a card type check. It is unlikely that each individual check generates an audit event; it is likely that only the verification service itself generates the event. In Figure 3 , the SecureServiceFacade is the entry to the Business tier. It provides the remote interface that the Client uses to access the target component, such as another EJB or a Business Object. Instead of forwarding directly to the target component, the SecureServiceFacade first invokes AuditInterceptor. The AuditInterceptor then consults the EventCatalog to determine whether to generate an audit event and, if so, what audit event to generate. If an audit event is generated, the AuditLog is then used to persist the audit event. Afterward, the SecureServiceFacade then forwards the request as usual to the Target. On the return of invocation of the Target, the SecureServiceFacade again calls the AuditInterceptor. This allows auditing of both start and end events. Exceptions raised from the invocation of the Target also cause the SecureServiceFacade to invoke the AuditInterceptor. More often than not, you want to generate audit events for exceptions. Figure 4 depicts the Secure Service Facade Interceptor strategy sequence diagram.

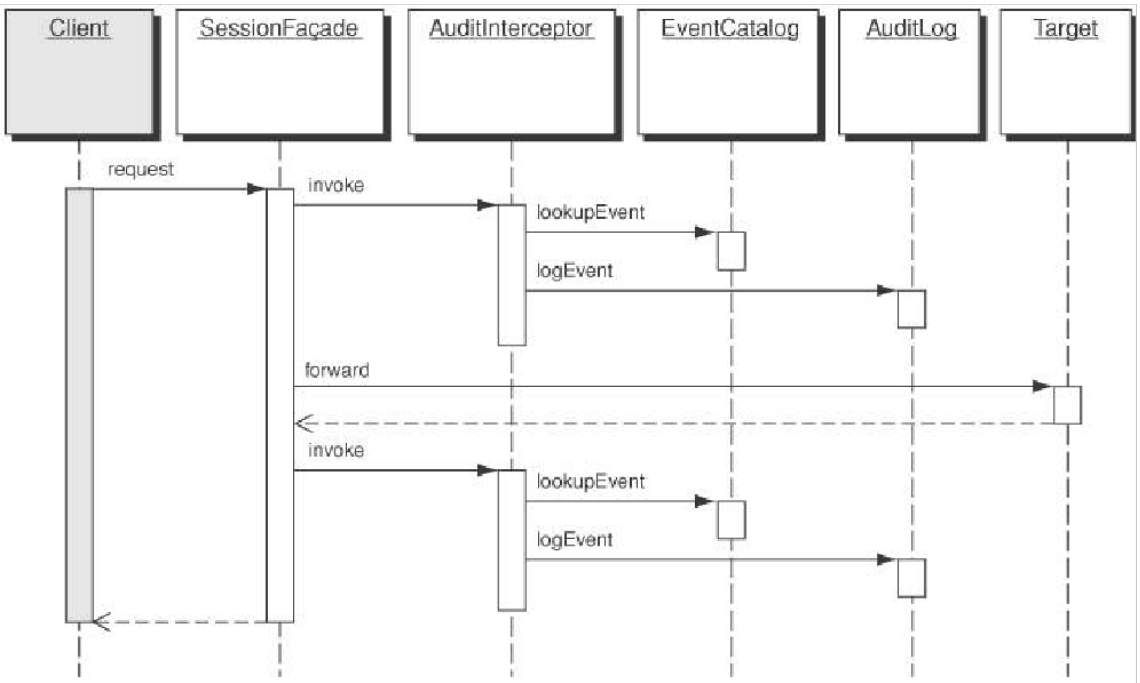


Figure 4: Secure Service Facade Interceptor strategy sequence diagram.

Pitfalls

The Audit Interceptor pattern provides developers with a standard way of capturing and auditing events in a decoupled manner. Auditing is an essential part of any security architecture. Audit events enable administrators to capture key events that they can later use to reconstruct who did what and when in the system. This is useful in cases of a system crash or in tracking down an intruder if the system is compromised.

- **business tier** Auditing. The Audit Interceptor pattern is responsible for providing a mechanism to capture audit events using an Interceptor approach. It is independent of where the audit information gets stored or how it is retrieved. Therefore, it is necessary to understand the general issues relating to auditing. Typically, audit logs (whether flat files or databases)should be stored separately from the applications, preferably on another machine or even off-site. This prevents intruders from covering their tracks by doctoring or erasing the audit logs. Audit logs should be writable but not updateable, depending on the implementation.
- **distributed security** JMS. The Audit Interceptor pattern is responsible for auditing potentially hundreds or even

thousands of events per second in high-throughput systems. In these cases, a scalable solution must be designed to accommodate the high volume of messages. Such a solution would involve dumping the messages onto a persistent JMS queue for asynchronous persistence. In this case, the JMS queue itself must be secured. This can be done by using a JMS product that supports message-level encryption or using some of the other strategies for securing JMS described in Chapter5. Since the queue must be persistent, you will also need to find a product that supports a secure backing store.

Consequences

Auditing is one of the key requirements for mission-critical applications. Auditing provides a trail of recorded events that can tie back to a Principal. The Audit Interceptor provides a mechanism to audit Business-tier events so that operations staff and security auditors can go back and examine the audit trail and look for all forms of application-layer attacks. The Audit Interceptor itself does not prevent an attack, but it does provide the ability to capture the events of the attack so that they can later be analyzed. Such an analysis can help prevent future attacks. The Audit Interceptor pattern has the following consequences for developers:

- Centralized, declarative auditing of service requests. The Audit Interceptor centralizes the auditing code within the application. This promotes reuse and maintainability.
- Pre-and post-process audit handling of service requests. The Audit Interceptor enables developers to record audit events prior to a method call or after a method call. This is important when considering the business requirements. Auditing is often required prior to the service or method call as a form of recording an"attempt."In other cases, an audit event is required only after the outcome of the call has been decided. And finally, there are cases where an audit event is needed in the event of an exception with the call.
- Auditing of services decoupled from the services themselves. The Audit Interceptor pattern decouples the business logic code from the auditing code. Business developers should not have to consider auditing requirements or implement code to support auditing. By using the Audit Interceptor, auditing can be achieved without impacting business developers.
- Supports evolving requirements and increases maintainability. The Audit Interceptor supports evolving auditing requirements by decoupling the events that need to be audited from the implementation. An audit catalog can be created that defines audit events declaratively, thus allowing different event types for different circumstances to be added without changing code. This improves the overall maintainability of the code by reducing the number of changes to it.
- Reduces performance. The cost of using an interceptor pattern is that performance is reduced anytime the interceptor is invoked. Every time that Audit Interceptor determines that a request or response does not require generation of an audit event, it unnecessarily decreases performance.

Known uses

Authentication Enforcer

Pattern documentation

Quick info

Intent: You need to verify that each service request is from an authenticated entity.

Problem

You need to verify that each request is from an authenticated entity, and since different classes handle different requests, authentication code is replicated in many places and the authentication mechanism can't easily be changed. Choice of user authentication mechanisms often require changes based on changes in business requirements, application-specific characteristics, and underlying security infrastructures. In a coexisting environment, some applications may use HTTP basic authentication or form-based authentication. In some applications, you may be required to use client certificate-based authentication or custom authentication via JAAS. It is therefore necessary that the authentication mechanisms be properly abstracted and encapsulated from the components that use them. During the authentication process, applications transfer user credentials to verify the identity requesting access to a particular resource. The user credentials and associated data must be kept private and must not be made available to other users or coexisting applications. For instance, when a user sends a credit card number and PIN to authenticate a Web application for accessing his or her banking information, the user wants to ensure that the information sent is kept extremely confidential and does not want anyone else to have access to it during the process.

Forces

- Access to the application is restricted to valid users, and those users must be properly authenticated.
- There may be multiple entry points into the application, each requiring user authentication.
- It is desirable to centralize authentication code and keep it isolated from the presentation and business logic.

Example

(Nothing given)

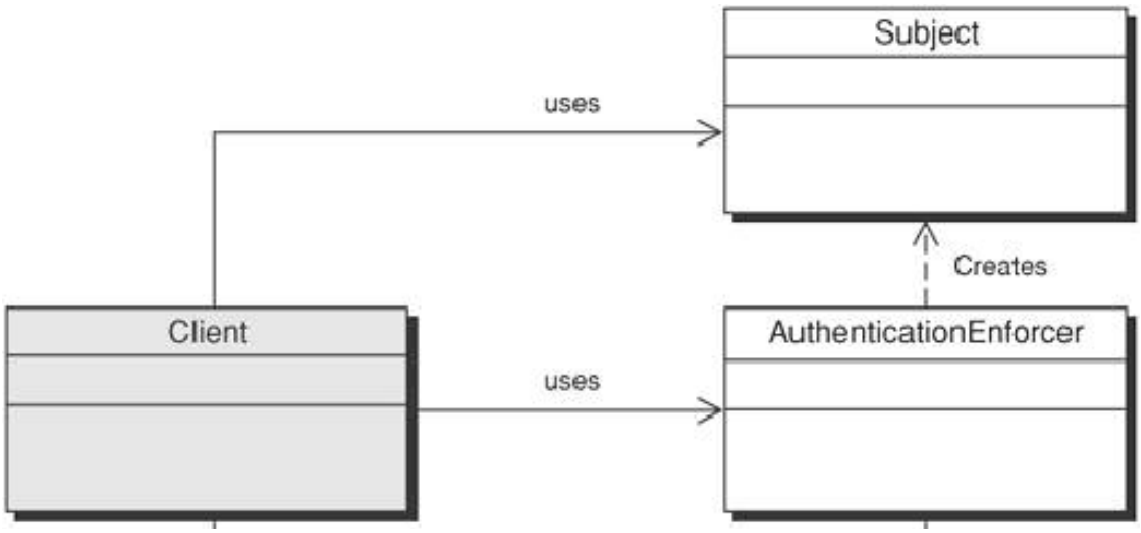
Solution

Create a centralized authentication enforcement that performs authentication of users and encapsulates the details of the authentication mechanism.

The Authentication Enforcer pattern handles the authentication logic across all of the actions within the Web tier. It assumes responsibility for authentication and verification of user identity and delegates direct interaction with the security provider to a helper class. This applies not only to password-based authentication, but also to client certificate-based authentication and other authentication schemes that provide a user's identity, such as Kerberos. Centralizing authentication and encapsulating the mechanics of the authentication process behind a common interface eases migration to evolving authentication requirements and facilitates reuse. The generic interface is protocol-independent and can be used across tiers. This is especially important in cases where you have clients that access the Business tier or Web Services tier components directly.

Structure

Figure 1 shows the participants in the Authentication Enforcer pattern. The core Authentication Enforcer consists of three classes: AuthenticationEnforcer, RequestContext, and Subject.



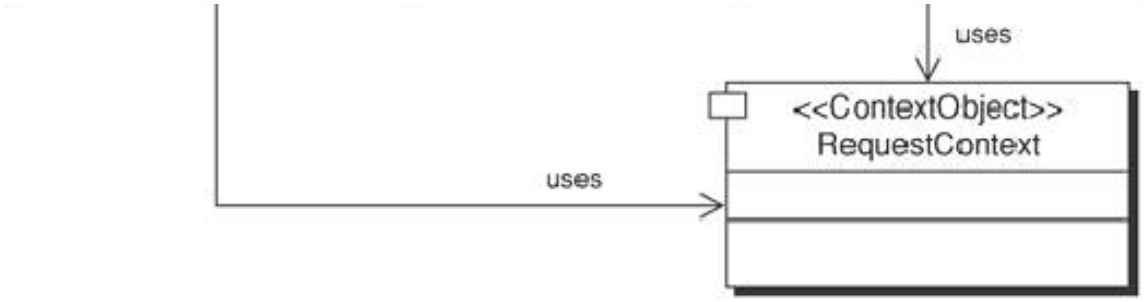


Figure 1: Class layout of the Authentication Enforcer.

Dynamics

The dynamics of an authentication enforcer are depicted in Figure 2 .

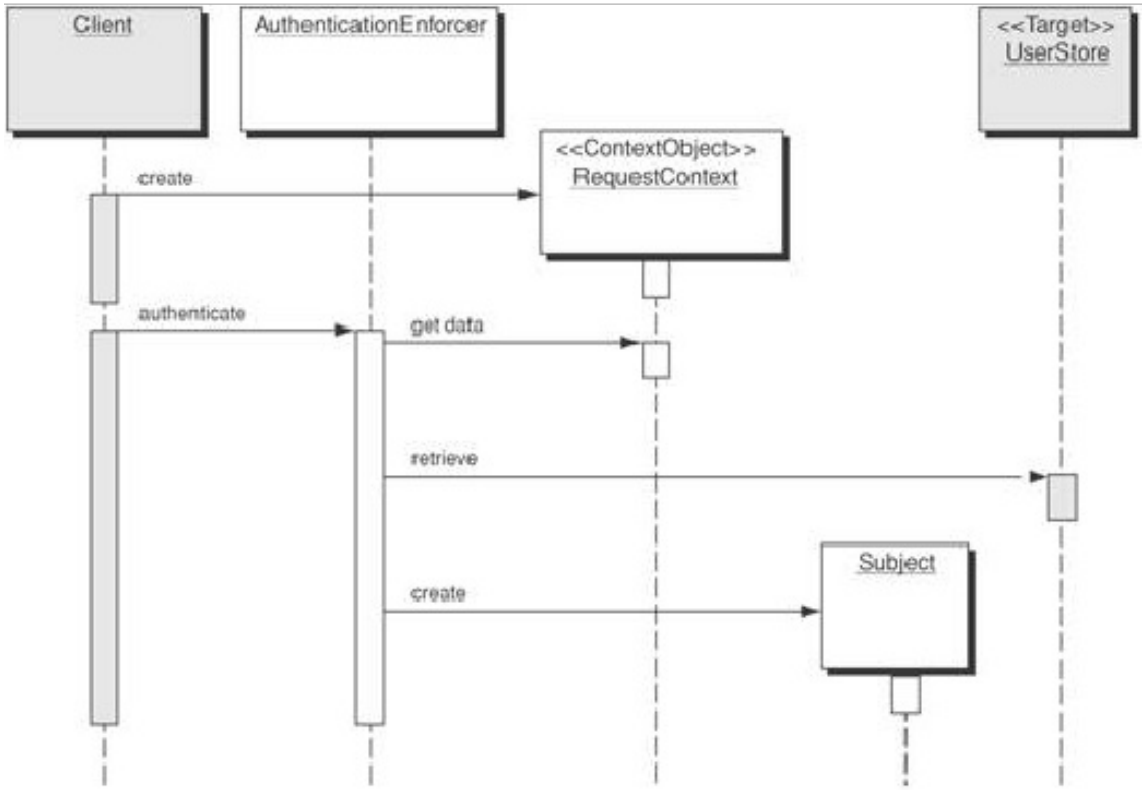


Figure 2: Sequence diagram of the Authentication Enforcer.

Participants

Figure 1 is a class diagram of the Authentication Enforcer pattern participant classes. Their responsibilities are:

- **client** A client uses the AuthenticationEnforcer to authenticate a user.
- **authenticationenforcer** The AuthenticationEnforcer authenticates the user using the credentials passed in the RequestContext.
- **requestcontext** The RequestContext contains the user's credentials extracted from the protocol-specific request mechanism.
- **subject** The AuthenticationEnforcer creates a Subject instance that represents the authenticated user.

Collaborations

Figure 2 depicts a typical client authentication using Authentication Enforcer. In this case, the Client is a SecureBaseAction that delegates to the AuthenticationEnforcer, which retrieves the appropriate user credentials from the UserStore. Upon successful authentication, the AuthenticationEnforcer creates a Subject instance for the requesting user and stores it in its cache.

Client (such as a FrontController or ApplicationController)creates RequestContext containing user's credentials.

Client invokes AuthenticationEnforcer's authenticate method, passing the RequestContext.

AuthenticationEnforcer retrieves the user's credentials from the RequestContext and attempts to locate user's Subject instance in its cache based upon the supplied user identifier in the credentials. This identifier may vary depending upon the authentication mechanism and may possibly require some form of mapping, for example, if an LDAP DN retrieved from a client

certificate is used as a credential. Unable to locate an entry in the cache, the AuthenticationEnforcer retrieves the user's corresponding credentials in the UserStore. (Typically this will contain a hash of the password.)The AuthenticationEnforcer will verify that the user-supplied credentials match the known credentials for that user in the UserStore and upon successful verification will create a Subject for that user. The AuthenticationEnforcer will then place the Subject in the cache and return it to the SecureBaseAction.

Implementation

The Authentication Enforcer pattern provides a consistent and structured way to handle authentication and verification of requests across actions within Web-tier components and also supports Model-View-Controller (MVC)architecture without duplicating the code. The three strategies for implementing an Authentication Enforcer pattern include Container Authenticated Strategy, Authentication Provider Strategy (Using Third-party product), and the JAAS Login Module Strategy.

Three implementation strategies are described here: the Container Authenticated Strategy, the Authentication Provider-Based Strategy and the JAAS Login Module Strategy.

The Container Authenticated Strategy is usually considered to be the most straightforward solution, where the container performs the authentication process on behalf of the application. The J2EE specification mandates support for HTTP Basic Authentication, Form Based Authentication, Digest-based Authentication, and Client-certificate Authentication. The J2EE container takes the responsibility for authenticating the user using one of these four methods. These mechanisms don't actually define the method to verify the credentials, but rather they show how to retrieve them from the user. How the container performs the authentication with the supplied credentials depends on the vendor-specific J2EE container implementation. Most J2EE containers handle the authentication process by associating the current HTTPServletRequest object, and its internal session, with the user. By associating a session with the user, the container ensures that the initiated request and all subsequent requests from the same user can be associated with the same session until that user's logout or the authenticated session expires. Once authenticated, the Web application can make use of the following methods provided by the HTTPServletRequest interface. **getRemoteUser ()**

- Determines the user name with which the client authenticated. **isUserInRole (String username)**
- Determines the given user is in a specified security role. **getUserPrincipal ()**
- Returns a java. security. Principal object.The Authentication Provider-Based Strategy adopts a third-party authentication provider for providing authentication for J2EE applications. Figure 3 illustrates how the Authentication provider is responsible for the authentication of the user, and the Authentication Enforcer extracts the user's Principal and creates a Subject instance with that Principal.

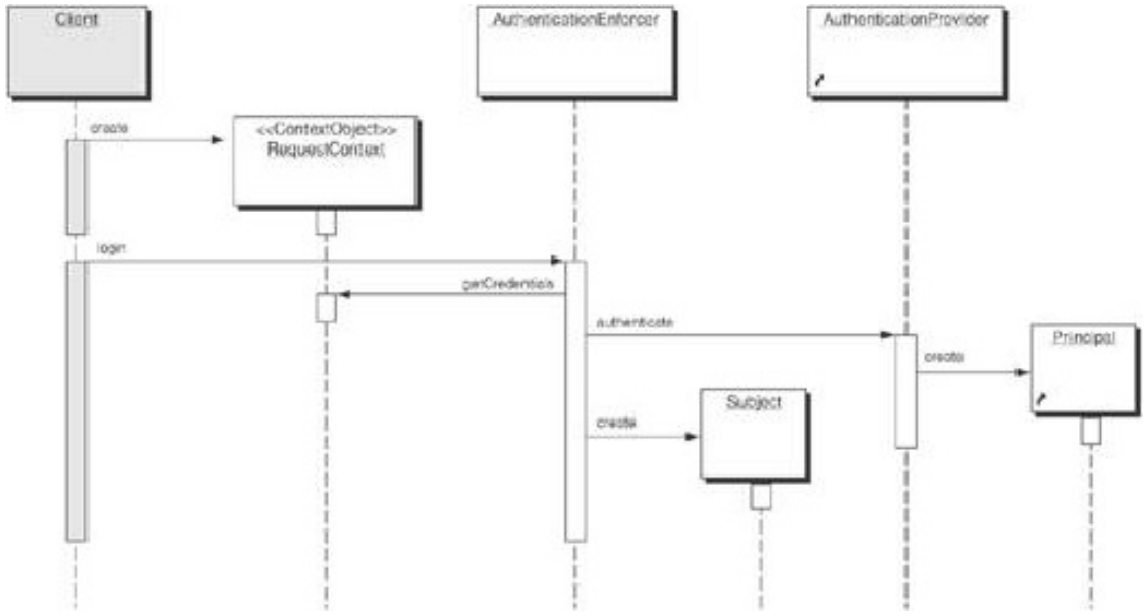


Figure 3: Sequence diagram for Authentication Provider strategy.

As you can see in Figure 3 , the authentication provider takes care of the authentication and creation of the Principal. The Authentication Enforcer simply creates the Subject and adds the Principal and the Credential to it. The Subject then holds a collection of permissions associated with all the Principals for that user. The Subject object can then be used in the application to identify, and also to authorize, the user.

The JAAS Login Module Strategy is more involved, because it takes responsibility for authentication from the container and moves it to the application that uses an authentication provider. This provides a pluggable approach and more programmatic control offering more flexibility to applications that require additional authentication mechanisms not supported by the J2EE specification. In essence, JAAS provides a standard programmatic approach to nonstandard authentication mechanisms. It also allows incorporation of multifactor authentication using security providers based on smart cards and biometrics. Figure 4 shows

the additional components required by this strategy.

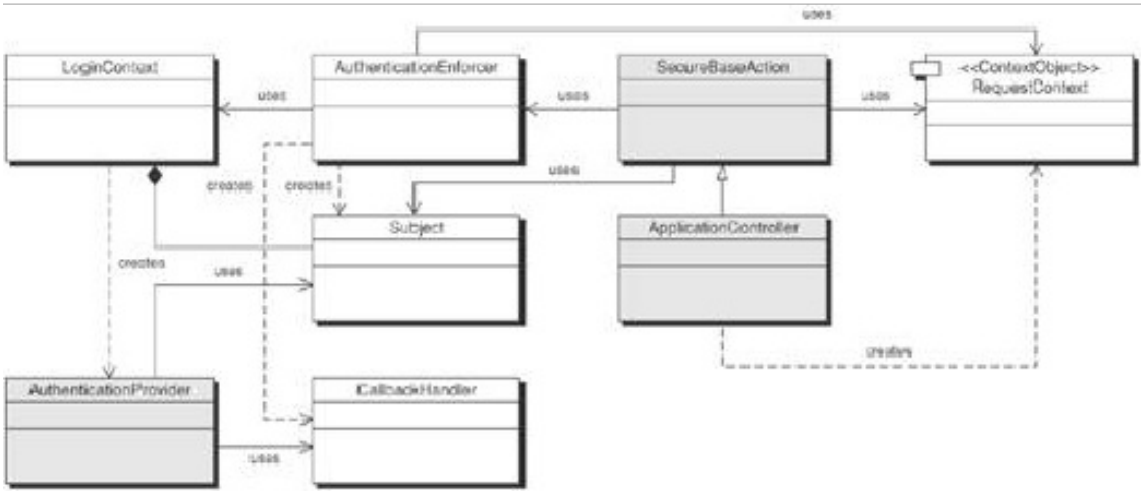


Figure 4: Class diagram for JAAS Login Module strategy.

In this strategy, the AuthenticationEnforcer is implemented as a JAAS client that interacts with JAAS LoginModule (s)for performing authentication. The JAAS LoginModules are configured using a JAAS configuration file, which identifies one or more JAAS LoginModules intended for authentication. Each LoginModule is specified via its fully qualified class name and an authentication Flag value that controls the overall authentication behavior. The flag values (such as Required, Requisite, Sufficient, Optional)defines the overall authentication process. The authentication process proceeds down the specified list of entries in the configuration file based on the flag values. The AuthenticationEnforcer instantiates a LoginContext class that loads the required LoginModule (s), specified in the JAAS configuration file. To initiate authentication the AuthenticationEnforcer invokes the LoginContext. login ()method which in turn calls the login ()method in the LoginModule to perform the login and authentication. The LoginModule invokes a CallbackHandler to perform the user interaction and to prompt the user for obtaining the authentication credentials (such as username/password, smart card and biometric samples). Then the LoginModule authenticates the user by verifying the user authentication credentials. If authentication is successful, the LoginModule populates the Subject with a Principal representing the user. The calling application can retrieve the authenticated Subject by calling the LoginContext's getSubject method. Figure 5 shows the sequence diagram for JAAS Login Module strategy.

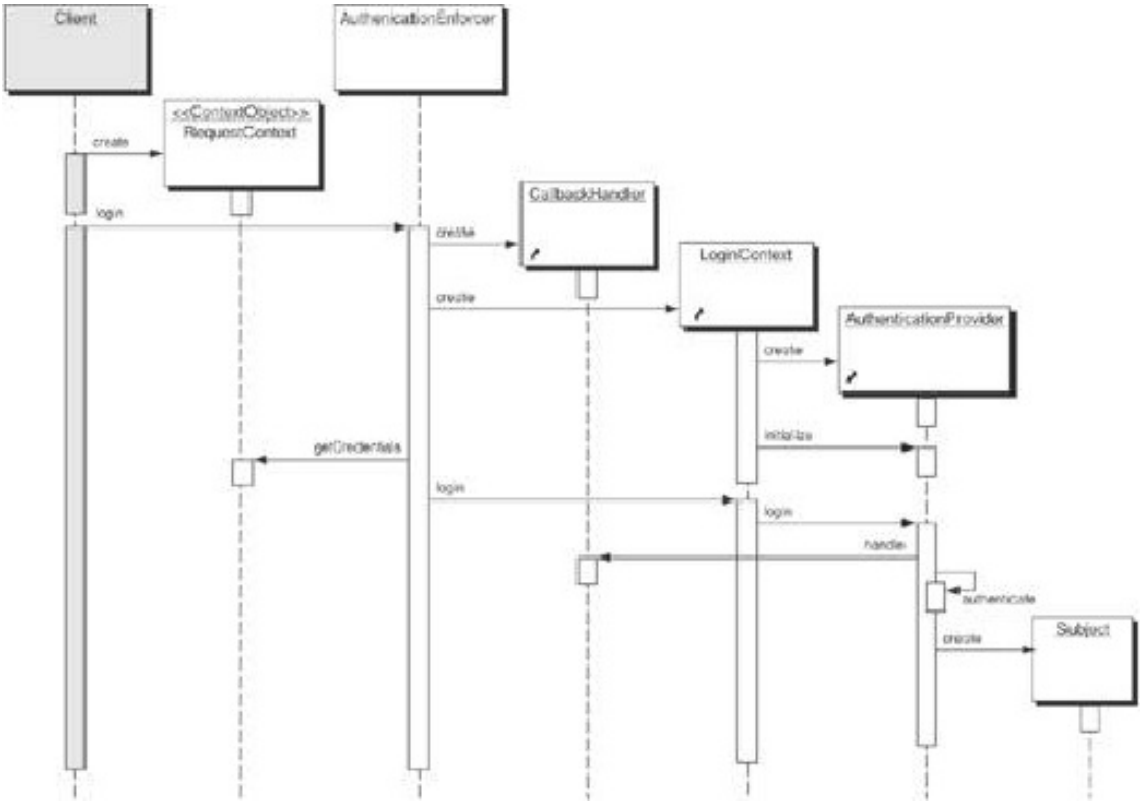


Figure 5: Sequence diagram for JAAS Login Module strategy.

- SecureBaseAction creates RequestContext containing user's credentials.
- SecureBaseAction invokes AuthenticationEnforcer's login method, passing in the RequestContext.
- AuthenticationEnforcer creates a CallbackHandler object that contains the username and password extracted from the

RequestContext
- AuthenticationEnforcer creates a LoginContext.
- LoginContext loads the AuthenticationProvider implementation of a LoginModule.
- LoginContext initializes the AuthenticationProvider.
- AuthenticationEnforcer invokes login method on the AuthenticationProvider.
- AuthenticationProvider retrieves the username and password by calling handle on the CallbackHandler.
- AuthenticationProvider uses username and password to authenticate user.
- Upon successful invocation AuthenticationProvider, upon commit, sets the Principal in the Subject and returns the Subject back up the request chain.

Pitfalls

The following security factors and risks apply when using the Authentication Enforcer pattern and its strategies.

- Authentication: Keep all user login code in classes separate from your application classes so you can re-implement them if you port the application or change your user authentication mechanism. The J2EE platform expects that developers will not be writing authentication functionality directly into their applications; the authentication mechanisms must remain independent from the application functionality. However, developers will do so as long as the container-provided mechanisms aren't adequate to suit the needs of an application. If this is done, it would be wise for the developer to isolate the code so that it can be easily removed as containers become more capable. The risk is this: As long as developers are writing authentication code in the application, they are opening up the possibility for bugs that attackers may be able to exploit.
- Standardization: Whenever possible, employ a JAAS Login Module Strategy. It promotes modularity and standardization. Most major application servers support JAAS, and it has become the industry-recognized standard. By using a proprietary approach, you increase the risk of creating security holes that can be exploited to subvert the application.
- Web authentication: Choose the right approach for your security requirements. Basic HTTP authentication is usually highly vulnerable to attacks and provides unacceptable exposure. On the other hand, requiring client certificates for authentication may deter potential users of the system, which is an abstract form of a denial of service attack.
- Confidentiality: During the authentication process, sensitive information is sent over the wire, so confidentiality becomes a critical requirement. Use a Secure Pipe pattern during the user login process to protect the user's credentials. Not securing transmission of the user credentials presents a risk that they may be captured and used by an attacker to masquerade as a legitimate user.

Consequences

By employing the Authentication Enforcer pattern, developers will be able to benefit from reduced code and consolidated authentication and verification to one class. The Authentication Enforcer pattern encapsulates the authentication process needed across actions into one centralized point that all other components can leverage. By centralizing authentication logic and wrapping it in a generic Authentication Enforcer, authentication mechanism details can be hidden and the application can be protected from changes in the underlying authentication mechanism. This is necessary because organizations change products, vendors, and platforms throughout the lifetime of an enterprise application.

A centralized approach to authentication reduces the number of places that authentication mechanisms are accessed and thereby reduces the chances for security holes due to misuse of those mechanisms. The Authentication Enforcer enables authenticating users by means of various authentication techniques that allow the application to appropriately identify and distinguish user's credentials. A centralized approach also forms the basis for authorization that is discussed in the Authorization Enforcer pattern. The Authentication Enforcer also provides a generic interface that allows it to be used across tiers. This is important if you need to authenticate on more than one tier and do not want to replicate code. Authentication is a key security requirement for almost every application, and the Authentication Enforcer provides a reusable approach for authenticating users.

Known uses

Authorization Enforcer

Pattern documentation

Quick info

Intent: Verify that requests for services are properly authorized at the method and link level.

Problem

Many components need to verify that each request is properly authorized at the method and link level. For applications that cannot take advantage of container-managed security, this custom code has the potential to be replicated.

In large applications, where requests can take multiple paths to access multiple business functionality, each component needs to verify access at a fine-grained level. Just because a user is authenticated does not mean that user should have access to every resource available in the application. At a minimum, an application makes use of two types of users; common end users and administrators who perform administrative tasks. In many applications there are several different types of users and roles, each of them require access based on a set of criterion defined by the business rules and policies specific to a resource. Based on the defined set of criterion, the application must enforce that a user can be able to access only the resources (and in the manner)that user is allowed to do.

Forces

- You want to minimize the coupling between the view presentation and the security controller.
- Web applications require access control on a URL basis.
- Authorization logic required to be centralized and should not spread all over the code base in order to reduce risk of misuse or security holes.
- Authorization should be segregated from the authentication logic to allow for evolution of each without impacting the other.

Example

(Nothing given)

Solution

Create an Access Controller that will perform authorization checks using standard Java security API classes.

The AuthorizationEnforcer provides a centralized point for programmatically authorizing resources. In addition to centralizing authorization, it also serves to encapsulate the details of the authorization mechanics. With programmatic authorization, access control to resources can be implemented in a multitude of ways. Using an AuthorizationEnforcer provides a generic encapsulation of authorization mechanisms by defining a standardized way for controlling access to Web-based applications. It provides fine-grained access control beyond the simple URL restriction. It provides the ability to restrict links displayed in a page or a header as well as to control the data within a table or list that is displayed, based on user permissions.

Structure

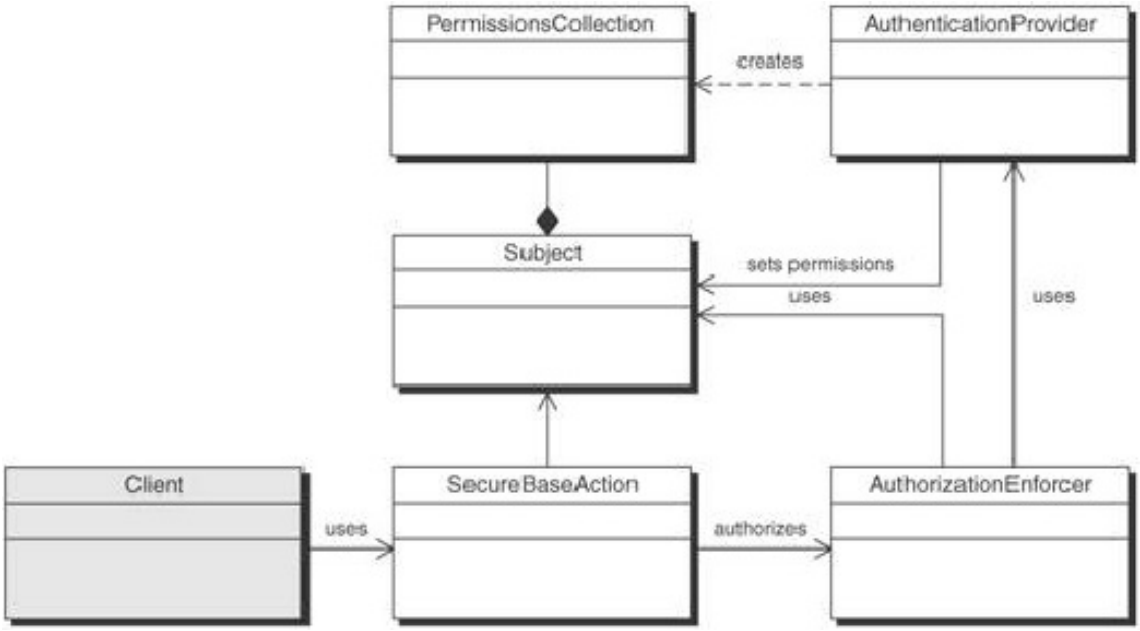


Figure 1: Class diagram for the Authorization Enforcer.

Figure 1 shows the AuthorizationEnforcer class diagram.

Dynamics

See the implementation section for the dynamics of the different implementation strategies.

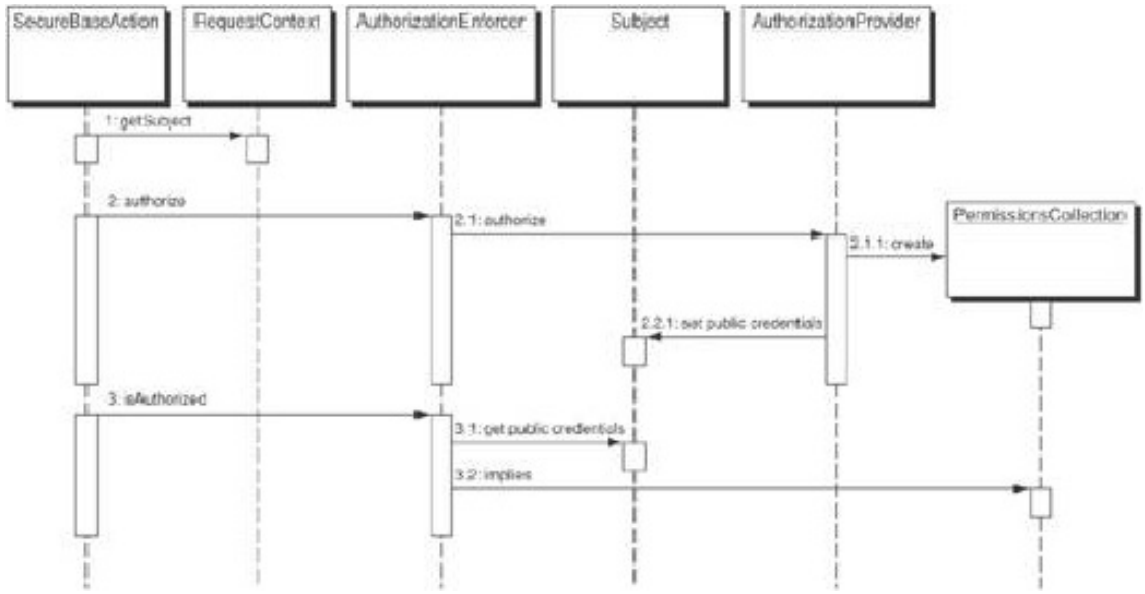


Figure 2: Sequence diagram for the Authorization Enforcer.

Participants

SecureBaseAction

- is an action class that gets the Subject from the RequestContext, and checks whether it is authorized for various permissions. **RequestContext**
- is a protocol-independent object used to encapsulate protocol-specific request information. **AuthorizationEnforcer**
- is an object used to generically enforce authorization in the Web tier. **Subject**
- is a class used to store a user's identities and credential information. **AuthorizationProvider**
- is a security provider that implements the authorization logic. **PermissionCollection**
- is a class used to store permissions, with a method for verifying whether a particular permission is implied in the collection.

Collaborations

Figure 2 shows a sequence diagram depicting the authorization of a user to a permission using the Authorization Enforcer.

Implementation

There are three commonly adopted strategies that can be employed to provide authorization using Authorization Enforcer pattern. The first is using an authorization provider, using a third-party security solution that provides authentication and authorization services. The second is purely programmatic authorization strategy which makes use of the Java2security API classes and leveraging the Java2Permissions class. The third is a JAAS authorization strategy that makes use of the JAAS principal based policy files and takes advantage of the underlying JAAS programmatic authorization mechanism for populating and checking a user's access privileges.

Not discussed further here is the J2EE container-managed authorization strategy. This strategy, or more correctly, the implementation, was found to be too static and inflexible. In the Authorization Provider strategy, the Authorization Enforcer makes use of a third-party security provider which handles authentication and provides policy based access control to J2EE based application components. In a typical authorization scenario (see Figure 2), the client (an Application Controller or extended action class)wants to perform a permission check on a particular user defined in the Subject class retrieved from his or her session. Prior to the illustrated flow, the SecureBaseAction class would have used the AuthenticationEnforcer to authenticate a user and then placed that user's Subject into the session. The Subject object can then be subsequently retrieved from the RequestContext. In the flow above through the following process:

- SecureBaseAction retrieves the Subject from the RequestContext.
- SecureBaseAction invokes AuthorizationEnforcer's authorize method, passing in the Subject.
- AuthorizationEnforcer calls the AuthorizationProvider's authorize method, again passing the Subject.
- AuthorizationProvider retrieves the appropriate permissions for the Principals defined in the Subject class and creates a

PermissionsCollection.

- AuthorizationProvider stores the PermissionsCollection in the Subject's public credential set.Sometime later, the SecureBaseAction needs to check that a user has a specific Permission and calls the isAuthorized method of the AuthorizationEnforcer.
- AuthorizationEnforcer retrieves the PermissionsCollection from the Subject's public credential set.
- AuthorizationEnforcer calls the implies method of the PermissionCollection, which passes in the checked Permission and returns the response.The Programmatic Authorization Strategy has the advantage of being flexible enough to easily accommodate new types of permissions for the variety of resources that you want to protect. The programmatic authorization strategy is a purely programmatic approach to authorization. It allows developers to arbitrarily create permissions and store them in the PermissionsCollection class, as demonstrated in Figure 2 . These permissions could be dynamically created at runtime as resources are created. For example, consider an application that allows administrators to upload new forms. Those forms may have access-control requirements that do not correspond to existing roles. You may need to create a resource permission that allows you to specify the name of the form and to then assign that permission to a user or group of users as necessary. It is often necessary to not only deny access to a particular link on a page, but to hide it from the view of those users without appropriate permissions to view its contents. In this case, a custom tag library can be constructed to provide tags for defining permission-based access to links and other resources in the JSPs. The JAAS Authorization Strategy is less flexible than a purely programmatic authorization strategy but provides the benefit of offering a standard JAAS LoginModule-based approach to authorization. It also utilizes a declarative means of mapping permissions to resources. This is a good approach for applications that do not support dynamic resource creation. Developers can map permissions to resources and roles to permissions declaratively at deployment time, thus eliminating programmatic mappings that often result in bugs and cause security vulnerabilities.

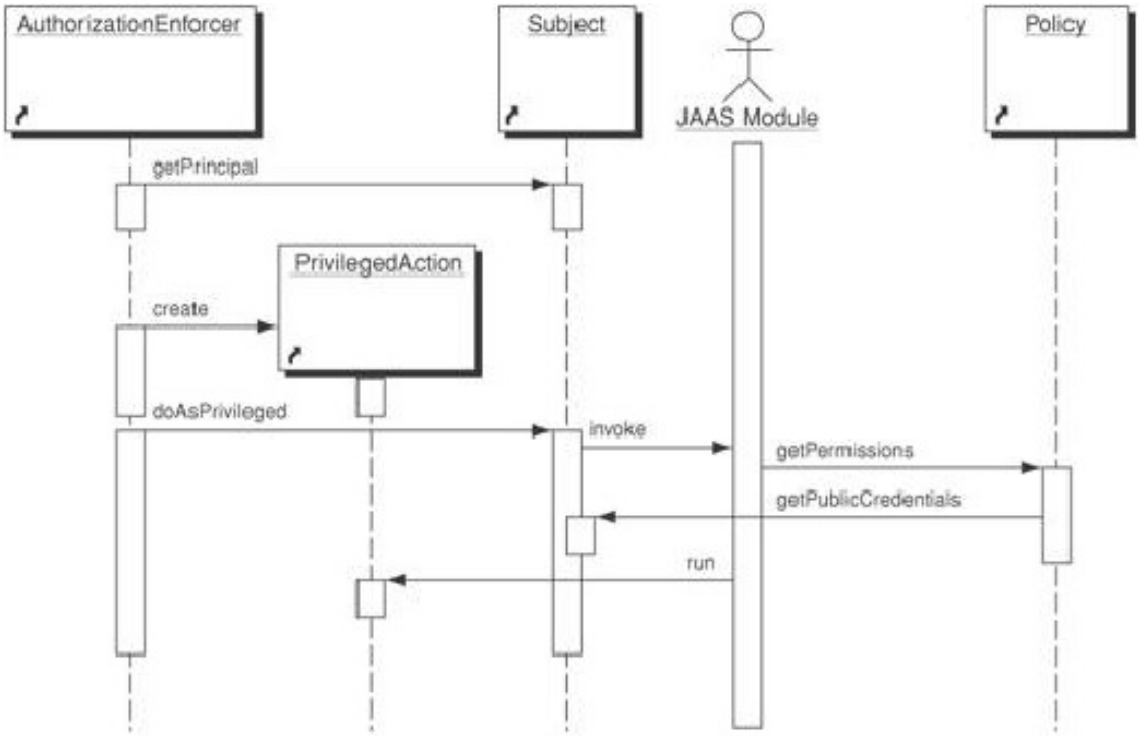


Figure 3: JAAS strategy for the Authorization Enforcer.

Figure 3 shows the sequence diagram of the Authorization Enforcer implemented using the JAAS Authorization Strategy. The key participants and their roles are as follows: AuthorizationEnforcer. An object used to generically enforce authorization in the Web tier. **Subject**

- is a Java2security class used to store a user's identities and security-related information. **PrivilegedAction**
- is a computation to be performed with privileges enabled. **Policy**
- is a JAAS Principal-based policy file, which defines the Principals with designated permissions to execute the specific application code or other privileges associated with the application or resources. **JAAS Module**
- is responsible for enforcing access control by enforcing the JAAS Policy and verifying that the authenticated Subject has been granted the appropriate set of permissions before invoking the PrivilegedAction.

Pitfalls

Authorization. Protect resources on a case by case basis. Fine-grained authorization allows you to properly protect the application without imposing a one-size-fits-all approach that could expose unnecessary security vulnerabilities. A common security vulnerability arises from access-control models that are too coarse-grained. When the model is too coarse-grained, you inevitably have users that do not fit nicely into the role-permission mappings defined. Often, administrators are forced to give these users elevated access due to business requirements. This leads to increased exposure. For instance, you have two groups of users that you break into two roles (staff and admin). The staff role only has the ability to read form data. The admin

role has the ability to create, read, update, and delete (CRUD)form data. You find that you have a few users that need to update the form data, though they should not be able to create or delete it. You are now forced to put them into the admin role, giving them these additional permissions because your model is too coarse-grained.

Consequences

- Centralizes control. The Authorization Enforcer allows developers to encapsulate the complex intricacies of implementing access control. It provides a focus point for providing access control checks, thus eliminating the chance for repetitive code.
- Improves reusability. Authorization Enforcer allows greater reuse through encapsulation of disparate access-control mechanisms through common interfaces.
- Promotes separation of responsibility. Partitions authentication and access-control responsibilities, insulating developers from changes in implementations.

Known uses

Checkpointed System

Pattern documentation

Quick info

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

Aliases: Snapshot, Undo

Problem

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.

Forces

- 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.

Example

(Nothing given)

Solution

The Checkpointed System pattern (see Figure 1)consists of a Recovery Proxy (Proxy:)and a Recoverable Component which periodically saves a recoverable version of the component's state as a Memento. The Memento can be used to restore the component's state when required.

Structure

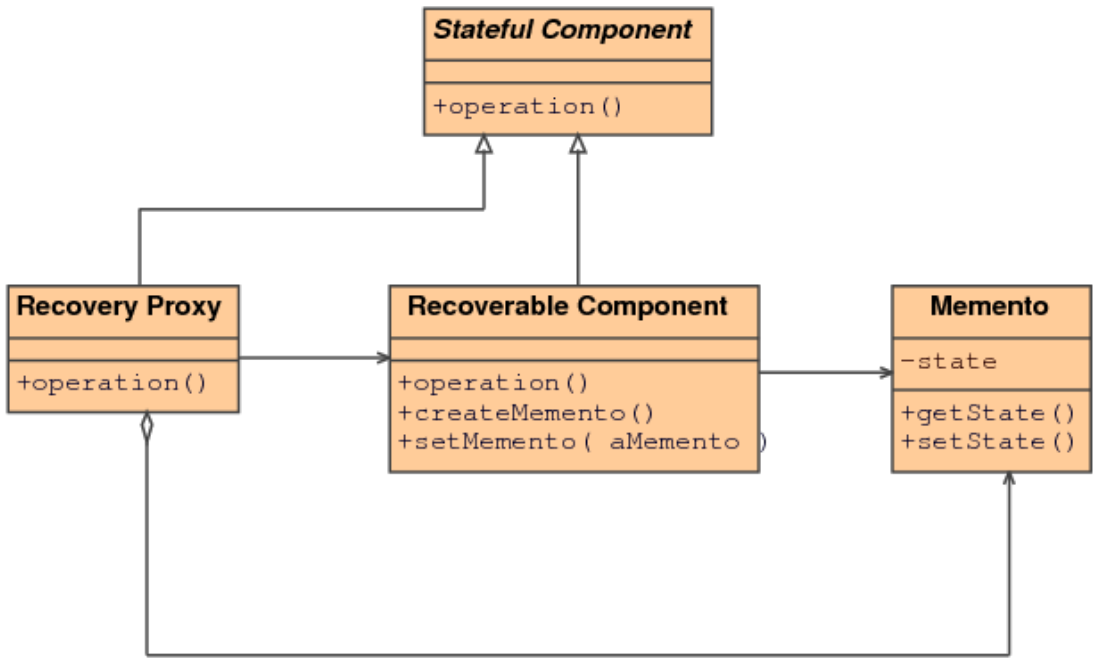


Figure 1: Class layout of the Checkpointed System.

See Figure 1 .

Dynamics

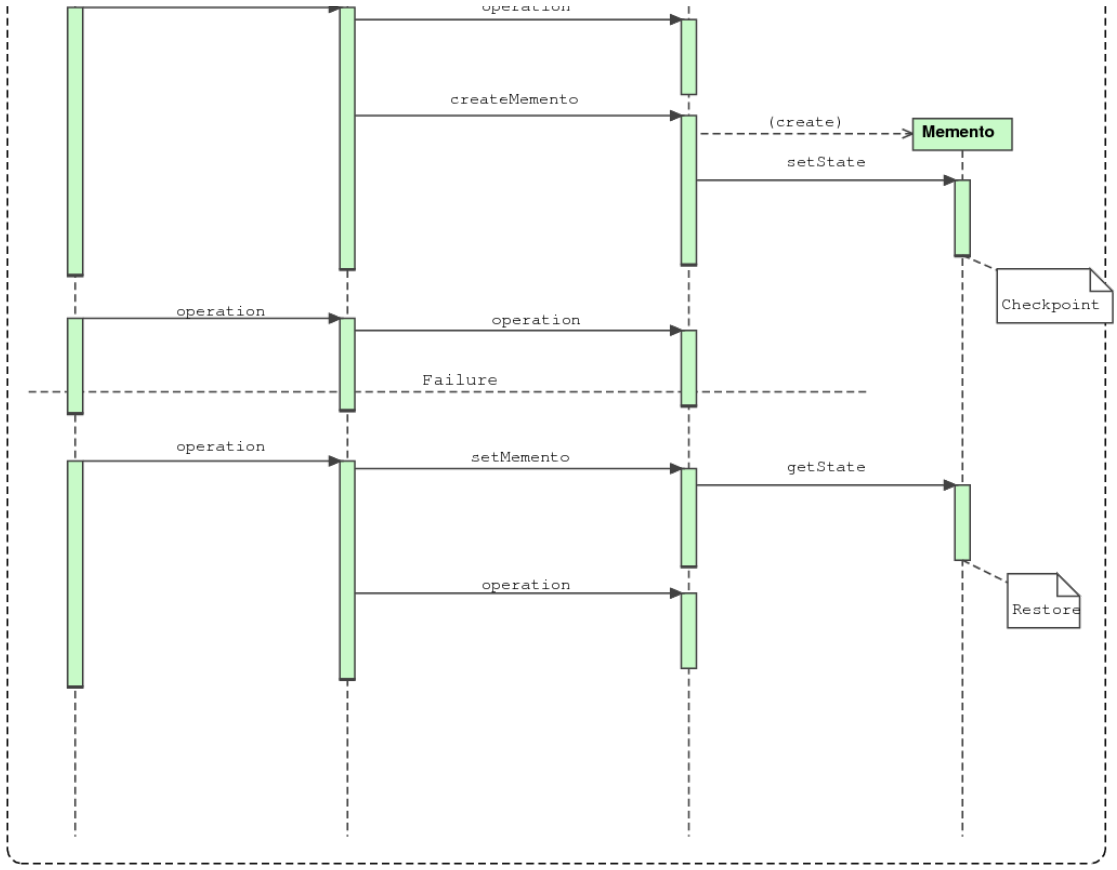


Figure 2: Event sequence for the Checkpointed System.

See Figure 2 .

Participants

The relations between the participants are shown in Figure 1 . The participants and their responsabilities are:

- Stateful Component Abstract class. Defines component operations.
- Recovery Proxy Proxyfor 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.
- MementoThe Recoverable Component's externalized state.

Collaborations

The interactions between the participants are shown in Figure 2 .

- 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.

Implementation

(Nothing given)

Pitfalls

(Nothing given)

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.

Known uses

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

Comparator Checked Fault Tolerant System

Pattern documentation

Quick info

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.

Aliases: Tandem System

Problem

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.

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.

Forces

The Comparator-Checked Fault-Tolerant System must resolve the following forces:

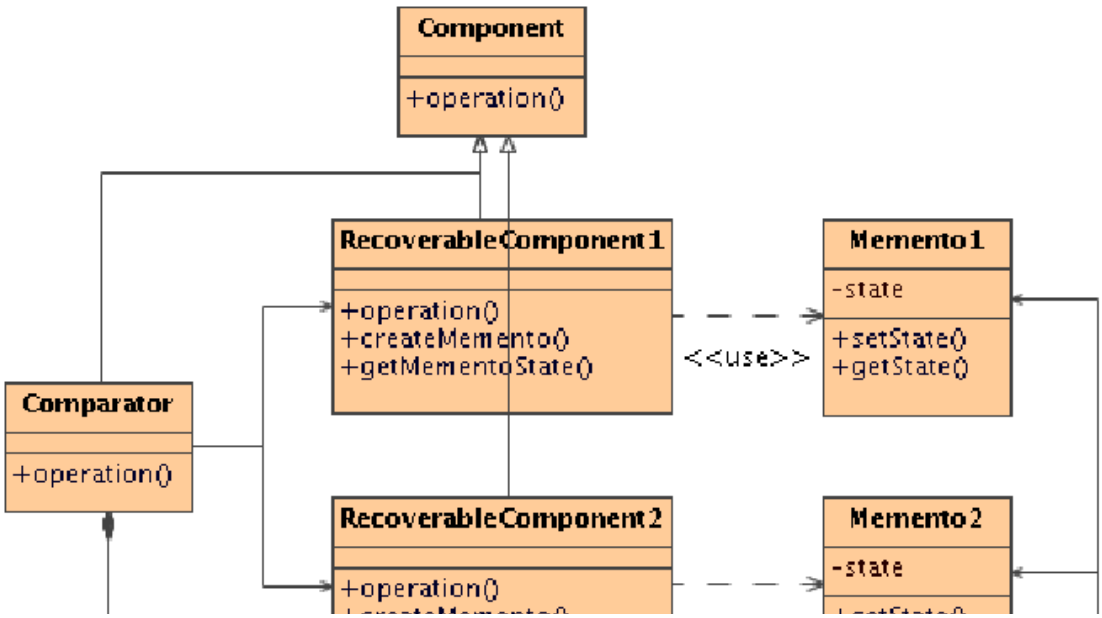
- Each component may fail at random times, without prior warning.
- It is possible that erroneous output is produced that is still within an acceptable output range for the component.
- For the system to remain functional, faults need to be detected with little or no delays.

Example

Solution

Structure a system so that an even number of recoverable components execute tasks concurrently. Let the components produce memento's as a means to compare their results. If a discrepancy between the results of a pair of components is detected, take corrective action.

Structure



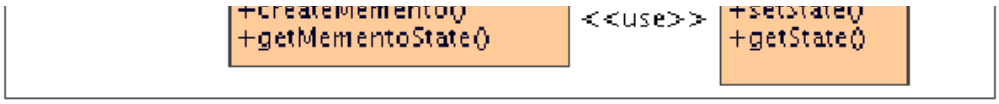


Figure 1: Comparator checked fault tolerant system class diagram.

A Comparator-Checked Fault-Tolerant System consists of an even number of Recoverable Components (often four or more), organized as sets of pairs, together with a Comparator for each pair. Each comparator examines Mementos 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. An overview is given in Figure 1 .

Dynamics

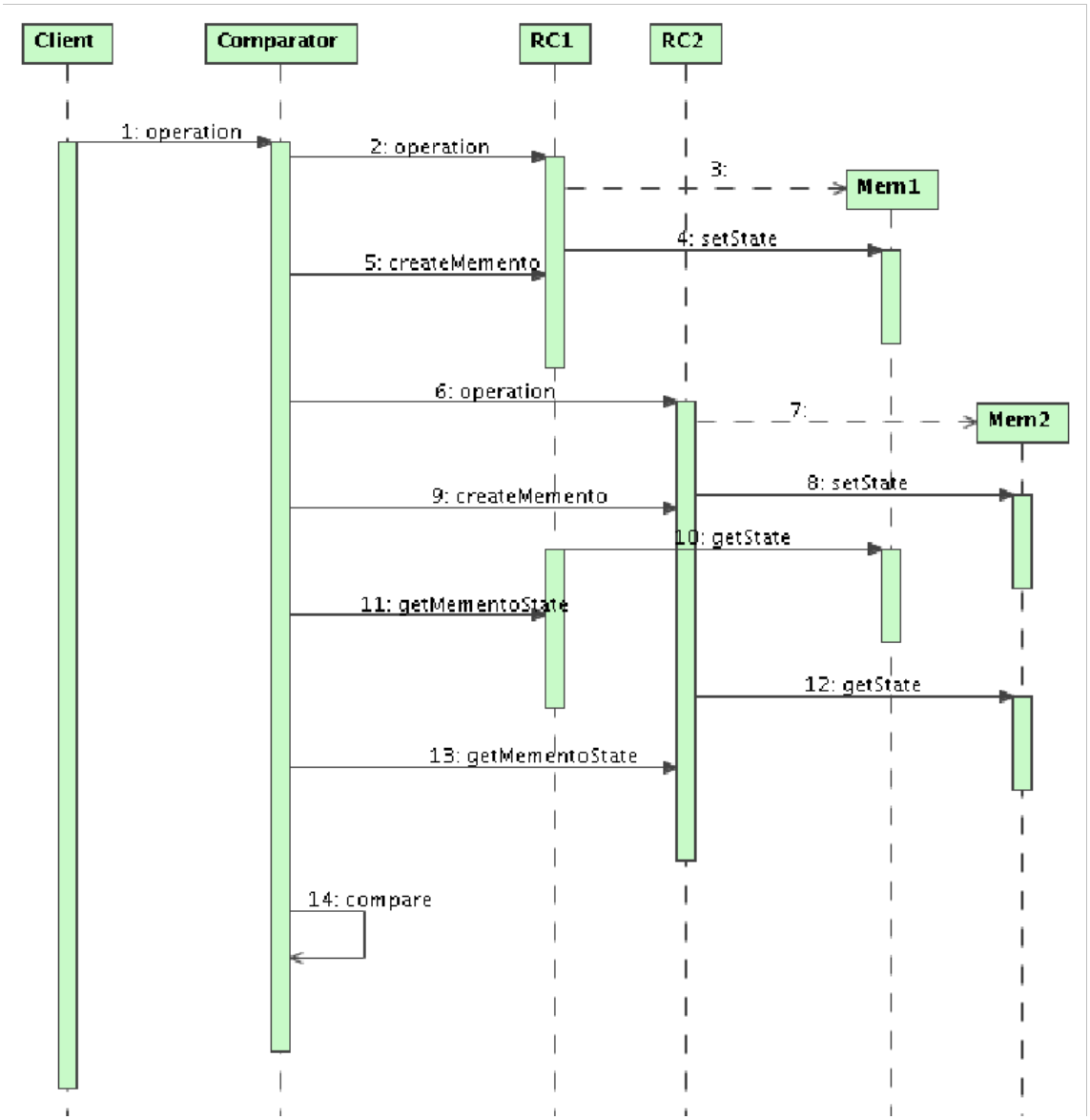


Figure 2: Comparator checked fault tolerant system sequence diagram.

The dynamics of a comparator checked fault tolerant system are described in Figure 2 .

Participants

Recoverable Components

- perform operations on behalf of clients. Each Recoverable Component is a member of a pair. **Comparator**
- is afor a pair of Recoverable Components. The Caretaker for Recoverable Components'. 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 operationÂ¿s result to the client; otherwise (if the states do not match), Comparator initiates recovery actions.

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.

Pitfalls

(Nothing given)

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.

Known uses

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

Container Managed Security

Pattern documentation

Quick info

Intent: You need a simple, standard way to enforce authentication and authorization in your J2EE applications and don't want to reinvent the wheel or write home-grown security code. Using a Container Managed Security pattern, the container performs user authentication and authorization without requiring the developer to hard-wire security policies in the application code.

Problem

Using a Container Managed Security pattern, the container performs user authentication and authorization without requiring the developer to hard-wire security policies in the application code. It employs declarative security that requires the developer to only define roles at a desired level of granularity through deployment descriptors of the J2EE resources. The administrator or deployer then uses the container-provided tool to map the roles to the users and groups available in the realm at the time of deployment. A realm is a database of users and their profiles that includes at least usernames and passwords, but can also include role, group, and other pertinent attributes. The actual enforcement of authentication and authorization at runtime is handled by the container in which the application is deployed and is driven by the deployment descriptors. Most containers provide authentication mechanisms by configuring user realms for LDAP, RDBMS, UNIX, and Windows.

Declarative security can be supplemented by programmatic security in the application code that uses J2EE APIs to determine user identity and role membership and thereby enforce enhanced security. In cases where an application chooses not to use a J2EE container, configurable implementation of security similar to Container Managed Security can still be designed by using JAAS-based authentication providers and JAAS APIs for programmatic security.

Forces

- You need to authenticate users and provide access control to business components.
- You want a straightforward, declarative security model based on static mappings.
- You want to prevent developers from bypassing security requirements and inadvertently exposing business functionality.

Example

(Nothing given)

Solution

Use Container Managed Security to define application-level roles at development time and perform user-role mappings at deployment time or thereafter.

In a J2EE application, both ejb-jar. xml and web. xml deployment descriptors can define container-managed security. The J2EE security elements in the deployment descriptor declare only the logical roles as conceived by the developer. The application deployer maps these application domain logical roles to the deployment environment. Container Managed Security at the Web tier uses delayed authentication, prompting the user for login only when a protected resource is accessed for the first time. On this tier, it can offer security for the whole application or specific parts of the application that are identified and differentiated by URL patterns. At the Enterprise Java Beans tier, Container Managed Security can offer method-level, fine-grained security or object-level, coarse-grained security.

Structure

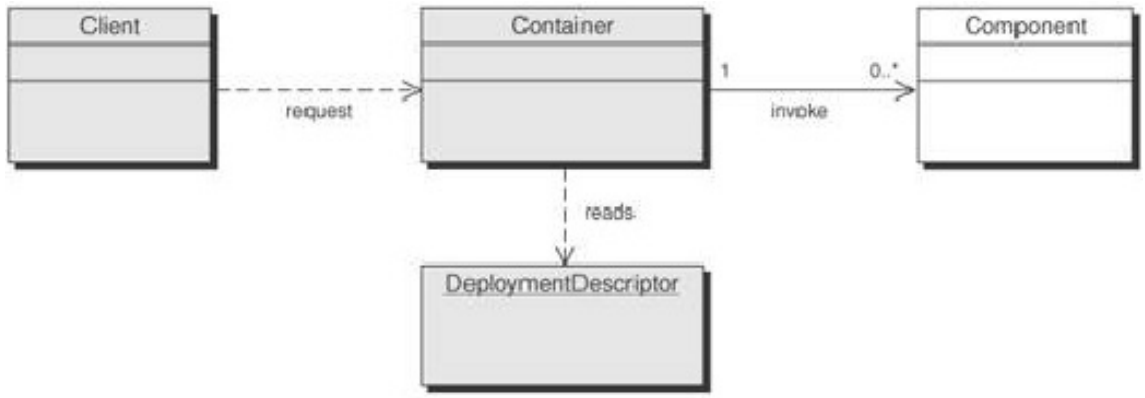


Figure 1: Structure of container managed security enforcement.

Figure 1 depicts a generic class diagram for a Container Managed Security implementation. Note that the class diagram can only be applicable to the container's implementation of Container Managed Security. The J2EE application developer would not use such a class structure, because it is already implemented and offered by the container for use by the developer.

Dynamics

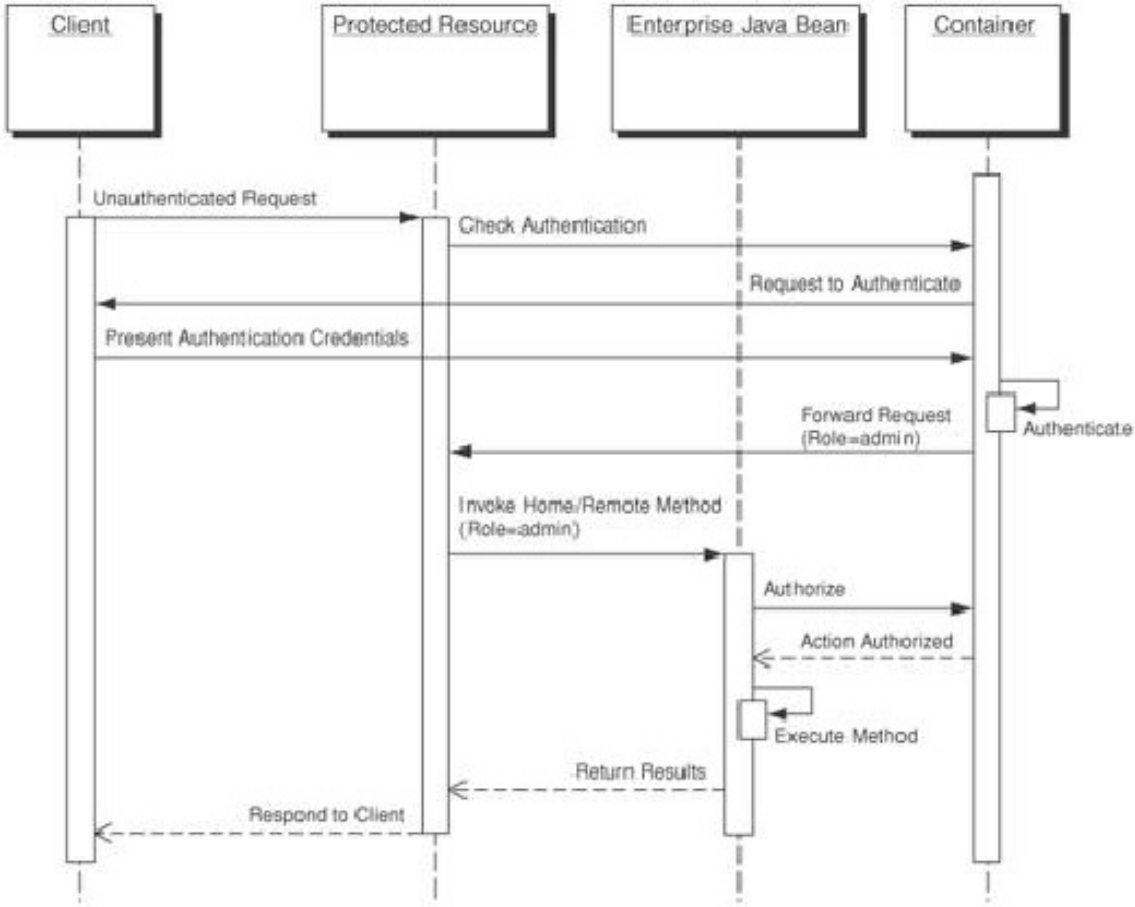


Figure 2: Container managed security dynamics.

Figure 2 depicts a sequence of operations involved in fulfilling a client request on a protected resource on the Web tier that uses an EJB component on the Business tier. Both tiers leverage Container Managed Security for authentication and access control.

Participants

- Client: A client sends a request to access a protected resource to perform a specific task.
- Container: The container intercepts the request to acquire authentication credentials from the client and thereafter authenticates the client using the realm configured in the J2EE container for the application.
- Protected Resource: The security policy of the protected resource is declared via the Deployment Descriptor. Upon authentication, the container uses the Deployment Descriptor information to verify whether the client is authorized to access the protected resource using the method, such as GET and POST, specified in the client request. If authorized, the request is forwarded to the protected resource for fulfillment.
- Enterprise Java Bean: The protected resource in turn could be using a Business Tier Enterprise Java Bean that declares its own security policy via the ejb. jar deployment descriptor. The security context of the client is propagated to the EJB container while making the EBJ method invocation. The EJB container intercepts the requests to validate against the security policy much like it did in the Web tier. If authorized, the EJB method is executed, fulfilling the client request. The results of execution of the request are then returned to the client.

Collaborations

- When a Client makes an unauthenticated request for a Protected Resource, this Protected Resource will first check the authentication of the Client with the Container.
- At this point, the Container will take over the authentication process and forward a request to authenticate to the Client.
- The client presents appropriate authentication credentials to the Container, which then performs the authentication.
- The Container forwards the request to the Protected Resource, together with the role of the authenticated Client.
- The Protected Resource is now able to invoke the appropriate method on the EJB, together with the role of the Client.
- The EJB checks the authorization with the Container, which replies that the action is indeed authorized.
- The EJB now executes the invoked method, and returns the results.

Implementation

Container Managed Security can be used in the Web and Business tiers of a J2EE application, depending on whether a Web container, an EJB container, or both are used in an application. It can also be supplemented by Bean Managed/Programmatic Security for fine-grained implementations. The various scenarios are described in this section. In the **Web Tier Container Managed Security** strategy, security restraints are specified in the web. xml of the client/user-facing Web application (that is, the Web tier of the J2EE application). If this is the only security strategy used in the application, an assumption is made that the back-end Business tier is not directly exposed to the client for direct integration. The web. xml declares the authentication method via the<auth-method>node of the web. xml to mandate either BASIC, DIGEST, FORM, or CLIENT-CERT authentication modes whenever authentication is required. It also declares authorization for protected resources that are identified and distinguished by their URL patterns. The actual enforcement or security is performed by the J2EE-compliant Web container in this strategy.

In the **Service Tier Container Managed Security** strategy, the developer configures the EJB's deployment descriptors to incorporate security into the service backbone of the application. A security role in EJB's ejb-jar. xml is defined through a<security-role-ref>element. These bean-specific logical roles can be associated to a security role defined with a different name in the<role-name>elements of the application deployment descriptor via a<role-link>element. The<assembly-descriptor>section of ejb-jar. xml, which is the application-level deployment descriptor, lists all the logical application-level roles via<role-name>elements, and these roles are mapped to the actual principals in the realm at the time of deployment.

Declarative Security for EJBs can either be at the bean level or at a more granular method level. Home and Remote interface methods can declare a<method-permission>element that includes one or more<role-name>elements that are allowed to access one or more EJB methods as identified by the<method>elements. One can also declare<exclude-list>elements to disable access to specific methods. To specify an explicit identity that an EJB should use when it invokes methods on other EJBs, the developer can use<use-caller-identity>or<run-as>/<role-name>elements under the<security-identity>element of the deployment descriptor. For finer granularity or to meet requirements unfulfilled by Container Managed Security, a developer could choose to use programmatic security in bean code or Web tier code in conjunction with Container Managed Security. For example, in the EJB code, the caller principal as a java. security. Principal instance can be obtained from the EJBContext. getCallerPrincipal ()method. The EJBContext. isCallerInRole (String)method can determine if a caller is in a role that is declared with a<security-role-ref>element. Similarly, on the Web tier, HttpServletRequest. getUserPrincipal ()returns a java. security. Principal object containing the name of the current authenticated user, and HttpServletRequest. isUserInRole (String)returns a Boolean indicating whether the authenticated user is included in the specified logical role. These APIs are very limited in scope and are confined to determining a user's identity and role membership. This approach is useful where instance-level security is required, such as permitting only the admin role to perform account transfers exceeding a certain amount limit.

Pitfalls

The extent of security offered by this pattern is limited to the security mechanisms offered by the container where the application code is deployed. It is also constrained by the limited subset of security aspects covered in the J2EE specification. As a result, the pattern elicits several risks:

- Limitations to fine-grained security. Use of Container Managed Security limits the ability of the application to incorporate fine-grained security such as that based on an object's run-time attribute values, time of day, and physical location of the client. These deficiencies could be overcome by programmatic security inside business components, but the security context information accessible to the component code is limited to principal information and the role association.
- Requires preconceived granularity of roles. Container Managed Security necessitates a preestablished notion of roles at the granularity level required by the application over the foreseeable future. This is because roles need to be defined in the deployment descriptor for each Web tier resource, ejb-tier business objects, or business methods before the application is packaged and deployed. Retrofitting additional roles after deployment would require repackaging the application with new deployment descriptors.
- Too limiting. Container Managed Security of the J2EE specification omits many aspects of integration between the container and the existing security infrastructure and limits itself to authentication and role-based access control. This may be too limiting for certain requirements, making programmatic security inevitable.Is Container Managed Security comprehensive at the Web tier?If the granularity of security enforcement is not matched by the granularity offered by the resource URL identifiers used by Container Managed Security to distinguish and differentiate resources, this pattern may not fulfill the requirements. This is particularly true in applications that use a single controller to front multiple resources. In such cases, the request URI would be the same for all resources, and individual resources would be identified only by way of some identifier in the query string (such as/myapp/controller?page=resource1). Container Manager Security by URL patterns is not applicable in such cases unless the container supports extensive use of regular expressions. Resource-level security in such scenarios requires additional work in the application.

Is Container Managed Security required at the service tier?If all the back-end business services are inevitably fronted by a security gateway such as Secure Service Proxy or Secure Service Fa Ã§ ade, having additional security enforcement via Container Managed Security on EJBs may not add much value and may incur unnecessary performance overhead. The choice must be carefully made in such cases.

Consequences

Container Managed Security offers flexible policy management at no additional cost to the organization. While it allows the

developer to incorporate security in the application by way of simply defining roles in the deployment descriptor without writing any implementation code, it also supports programmatic security for fine-grained access control. The pattern offers the following other benefits to the developer:

- Straightforward, declarative security model based on static mappings. The Container Managed Security pattern provides an easy-to-use and easy-to-understand security model based on declarative user-to-role and role-to-resource mappings.
- Developers are prevented from bypassing security requirements and inadvertently exposing business functionality. Developers often advertently or inadvertently bypass security mechanisms within the code. Using Container Managed Security prevents this and ensures that EJB methods are adequately protected and properly restricted at deployment time by the application deployer.
- Less prone to security holes. Since security is implemented by a time-tested container, programming errors are less likely to lead to security holes. However, the security functionality offered by the container could be too limited and inflexible to modify.
- Separation of security code from business objects. Since the container implements the security infrastructure, the application code is free of security logic. However, developers often end up starting with Container Managed Security and then using programmatic security in conjunction with it, which leads to mangled code with a mixture of declarative and programmatic security that is difficult to manage.

Known uses

Probably the best known use is the declarative container managed security offered by the **J2EE** platform.

Controlled Object Factory

Pattern documentation

Quick info

Intent: This pattern addresses how to specify the rights of processes with respect to a new object. When a process creates a new object through a factory, the request includes the features of the new object. These features include a list of rights to access the object.

Problem

A computing system that needs to control access to its created objects because of their different degrees of sensitivity. Rights for these objects are defined by authorization rules or policies that are enforced when a process attempts to access an object.

In a computing environment, executing applications need to create objects for their work. Some objects are created at program initialization, while others are created dynamically during execution. The access rights of processes with respect to objects must be defined when these objects are created, or there may be opportunities for the processes to misuse them. Applications also need resources such as I/O devices and others that may come from resource pools: when these resources are allocated, the application must be given rights to them.

Forces

The solution to this problem must resolve the following forces:

- Applications create objects of many different types, but we need to handle them uniformly with respect to their access rights, otherwise it would be difficult to apply standard security policies.
- We need to allow objects in a resource pool to be allocated and have their rights set dynamically: not doing so would be too rigid.
- There may be specific policies that define who can access a new object, and we need to apply these when creating the rights for an object. This is a basic aspect of security.

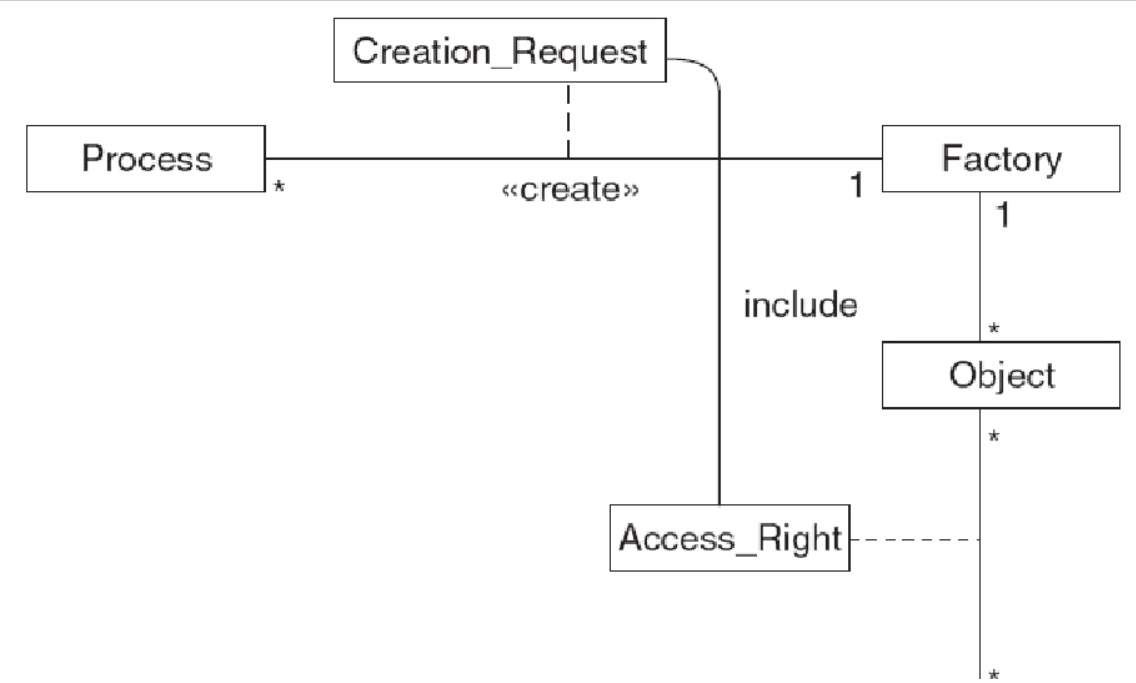
Example

In many operating systems the creator of an object gets all possible rights to the object. Other operating systems apply predefined sets of rights: for example, in Unix all the members of a file ownerÂ¿s group may receive equal rights for a new file. These approaches may result in unnecessary rights being given to some users, violating the principle of least privileges.

Solution

Whenever a new object is created, define a list of subjects that can access it, and in what way.

Structure



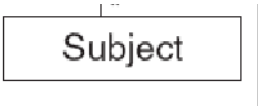


Figure 1: The structure of a Controlled Object Factory.

The structure of ais shown in Figure 1 .

Dynamics

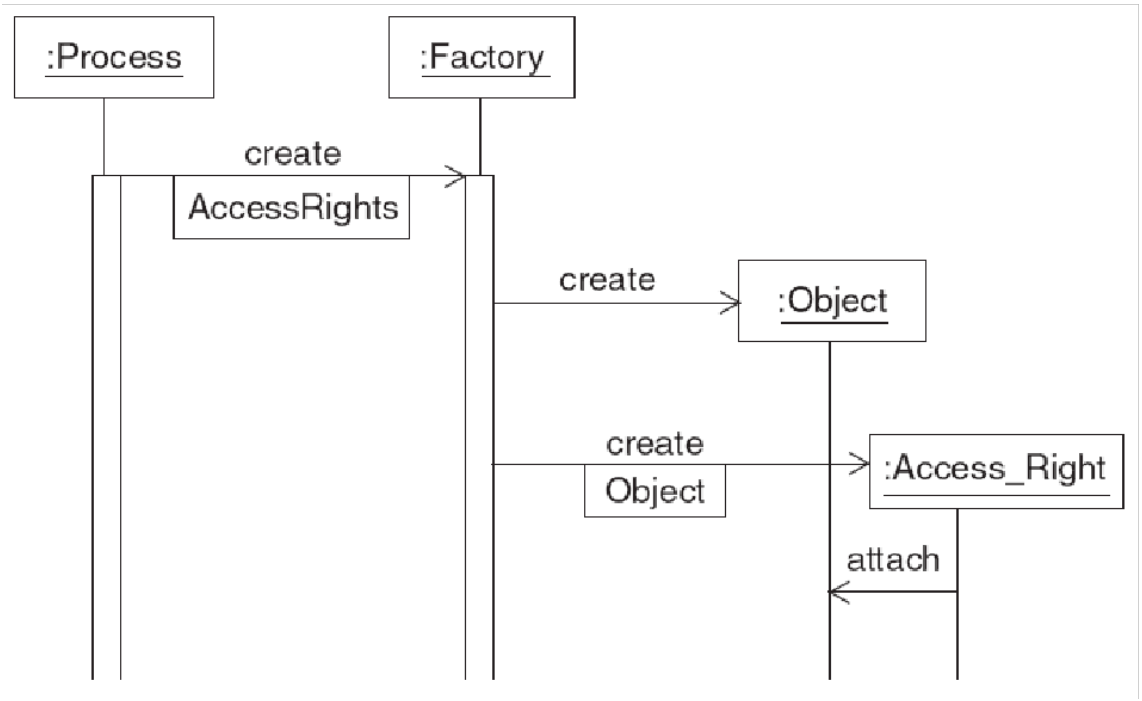


Figure 2: Controlled object factory dynamics.

The dynamics of aare shown in Figure 2 . A process creating an object through adefines the rights for other subjects with respect to this object.

Participants

When a Process creates a new object through a Factory, the CreationRequest includes the features of the new object. Among these features is a list of rights that define the access rights for a Subject to access the created Object. This implies that we need to intercept every access request: this is done by.

Collaborations

- A **Process** invokes the **Factory** to create a new **Object** with the provided **Access Rights** .
- The **Factory** then creates a new **Object** .
- The **Factory** also creates a new **Access_Right** object, to hold the access rights.
- The **Access_Right** object then attaches itself to the new **Object** .

Implementation

Each object may have an associated access control list (ACL). This will list the rights each user has for the associated object. Each entry specifies the rights that any other object within the system can have. In general, each right can be an ``allow''or a ``deny''. These are also known as Access Control Entries (ACE)in the Windows environment. The set of access rules is also known as the Access Control List (ACL)in Windows and most operating systems. Capabilities are an alternative to an ACL. A capability corresponds to a row in an access matrix. This is in contrast to the ACL, which is associated with the object. The capability indicates to the secure object that the subject does indeed have the right to perform the operation. The capability may carry some authentication features in order to show that the object can trust the provided capability information. A global table can contain rows that represent capabilities for each authenticated user, or the capability may be implemented as a lists for each user which indicates which object each user has access to.

Pitfalls

(Nothing given)

Consequences

The following benefits may be expected from applying this pattern:

- There will be no objects that have default access rights because somebody forgot to define rights to access them
- It is possible to define access rights to an object based on its sensitivity
- Objects allocated from a resource pool can have rights attached to them dynamically
- The operating system can apply ownership policies: for example, the creator of an object may receive all possible rights to the objects it creates. The following potential liabilities may arise from applying this pattern:
- There is a process creation overhead
- It may not be clear what initial rights to define

Known uses

The Win32API allows a process to create objects with various Create system calls using a structure that contains access control information (DACL)passed as a reference. When the object is created, the access control information is associated with the object by the kernel. The kernel returns a handle to the caller to be used for access to the object.

Controlled Object Monitor

Pattern documentation

Quick info

Intent: This pattern addresses how to control access by a process to an object. Use a reference monitor to intercept access requests from processes. The reference monitor checks whether the process has the requested type of access to the object.

Problem

The context of the problem lies in an operating system that consists of many users, objects that may contain sensitive data, and where we need to have controlled access to resources. When objects are created we define the rights processes have to them. These authorization rules or policies must be enforced when a process attempts to access an object.

Forces

The solution to this problem must resolve the following forces:

- There may be many objects with different access restrictions defined by authorization rules: we need to enforce these restrictions when a process attempts to access an object.
- We need to control different types of access, or the object may be misused.

Example

Our operating system does not check all user requests to access resources such as files or memory areas. A hacker discovered that some accesses are not checked, and was able to steal customer information from our files. He also left a program that randomly overwrites memory areas and produces serious disruption to the other users.

Solution

Use ato intercept access requests from processes. Thechecks whether the process has the requested type of access to the object according to some access rule.

Structure

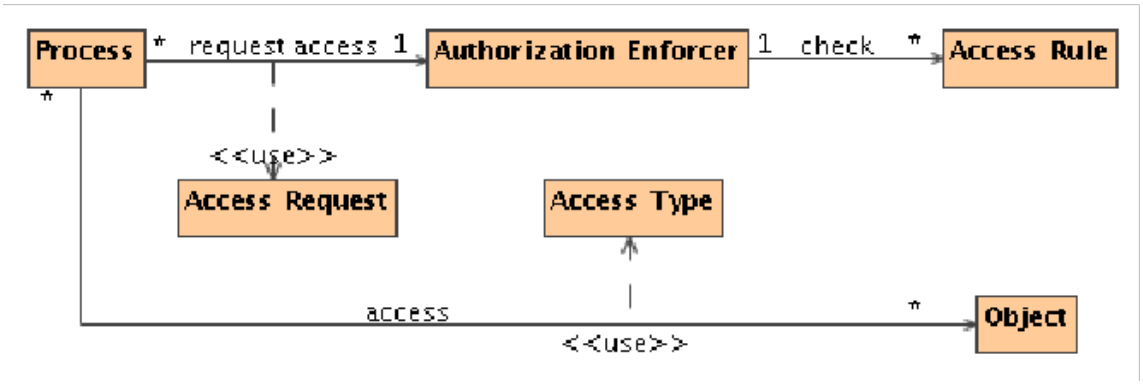


Figure 1: Class structure of a controlled object monitor.

Figure 1 shows the class diagram for this pattern. This is a specific implementation of a. The modification shows how the system associates the rules to the secure object in question.

Dynamics

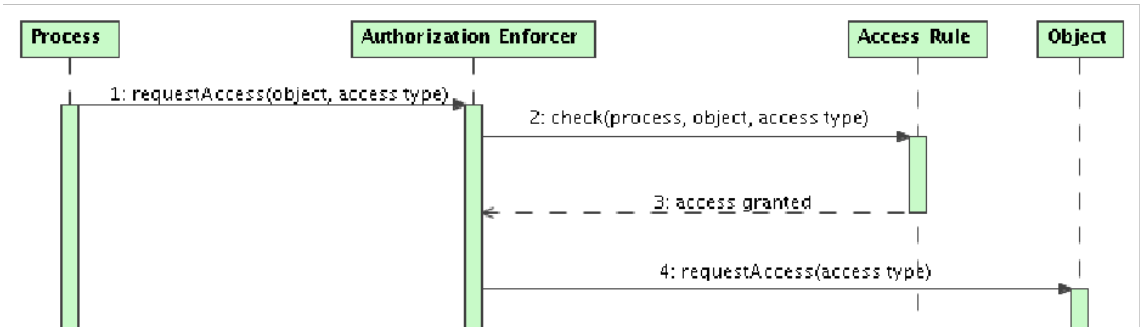


Figure 2: Dynamics of a controlled object monitor.

The dynamics of aare depicted in Figure 2 .

Participants

- A **Process** uses **Access Requests** to request access to an **Authorization Enforcer** .
- The **Authorization Enforcer** checks access requests against a number of **Access Rules** .
- The process accesses **Objects** . It therefore uses a certain **Access Type** .

Collaborations

Figure 2 shows the dynamics of secure subject access to a secure object. Here the request is sent to thewhere it checks the Access Rules. If the access is allowed, it is performed and result returned to the subject. Note that here, a handle or ticket is returned to the Subject so that future access to the secure object can be directly performed without additional checking.

Implementation

(Nothing given)

Pitfalls

(Nothing given)

Consequences

The following benefits may be expected from applying this pattern:

- Each access request can be intercepted and accepted or rejected depending on the authorization rules.
- The access rules can implement an access matrix defining different types of access for each subject. We can add content-dependent rules if required. The following potential liabilities may arise from applying this pattern:
- There is a need to protect the authorization rules. However, the same mechanism that protects resources can also protect the rules.
- There is an overhead involved in controlling each access. This is specially heavy for content-dependent rules. However, some accesses may be compiled for efficiency.

Known uses

Windows NT. The Windows NT security subsystem provides security using the patterns described here. It has the following three components:

- Local Security Authority
- Security Account Manager
- Security Reference MonitorThe Local Security Authority (LSA)and Security Account Manager (SAM)work together to authenticate the user and create the user's access token. The security reference monitor runs in kernel mode and is responsible for the enforcement of access validation. When an access to an object is requested, a comparison is made between the file's security descriptor and the Secure ID (SID)information stored in the user's access token. The security descriptor is made up of Access Control Entries (ACE's)included in the object's Access Control List (ACL). When an object has an ACL the SRM checks each ACE in the ACL to determine if access is to be granted. After the Security Authorization Enforcer (SRM)grants access to the object, further access checks are not needed, as a handle to the object that allows further access is returned the first time.

Types of object permissions are no access, read, change, full control, and special access. For directory access, the following are added: list, add, and read. Windows use the concept of a handle for access to protected objects within the system. Each object has a Security Descriptor (SD)that contains a Discretionary Access Control List (DACL)for the object. Each also process has a security token that contains an SID which identifies the process. This is used by the kernel to determine whether access is allowed. The ACL contains Access Control Entries (ACE's)that indicate what access is allowed for a particular process SID. The kernel scans the ACL for the rights corresponding to the requested access.

A process requests access to the object when it asks for a handle using, for example, a call to CreateFile (), which is used both to create a new file or open an existing file. When the file is created, a pointer to an SD is passed as a parameter. When an existing file is opened, the request parameters, in addition to the file handle, contain the desired access, such as GENERIC_READ. If the process has the desired rights for the access, the request succeeds and an access handle is returned, so that different handles to the same object may have different accesses. Once the handle is obtained, additional access to read a file will not require further authorization. The handle may also be passed to another trusted function for further processing.

Java1.2Security. The Java security subsystem provides security using the patterns described here. The Java Access Controller builds access permissions based on permission and policy. It has a checkPermission method that determines the codesource object of each calling method and uses the current Policy object to determine the permission objects associated with it. Note that the checkPermission method will traverse the call stack to determine the access of all calling methods in the stack. The java. policy file is used by the security manager that contains the grant statements for each codesource.

Controlled Process Creator

Pattern documentation

Quick info

Intent: This pattern addresses how to define and grant appropriate access rights for a new process, in an operating system in which processes or threads need to be created according to application needs.

Problem

A user executes an application composed of several concurrent processes. Processes are usually created through system calls to the operating system. A process that needs to create a new process gets the operating system to create a child process that is given access to some resources. A computing system uses many processes or threads. Processes need to be created according to application needs, and the operating system itself is composed of processes. If processes are not controlled, they can interfere with each other and access data illegally. Their rights for resources should be carefully defined according to appropriate policies, for exampleÂ¿need-to-know.Â¿

Forces

The solution to this problem must resolve the following forces:

- There should be a convenient way to select a policy to define processÂ¿rights. Defining rights without a policy brings contradictory and non-systematic access restrictions that can be easily circumvented.
- A child process may need to impersonate its parent in specific actions, but this should be carefully controlled, otherwise a compromised child could leak information or destroy data.
- The number of child processes created by a process must be restricted, or process spawning could be user to carry out denial-of-service attacks.
- There are situations in which a process needs to act with more than its normal rights, for example to access data in a file to which it doesnÂ¿t normally have access.

Example

Most operating systems create a process with the same rights as its parent. If a hacker can trick an operating system into creating a child of the supervisor process, this runs with all the rights of the supervisor.

Solution

Because new processes are created through system calls or messages to the operating system, we have a chance to control the rights given to a new process. Typically, operating systems create a new process as a child process. We let the parent assign a specific set of rights to its children, which is more secure because a more precise control of rights is possible.

Structure

The structure of the controlled process creator is depicted in Figure 1 .

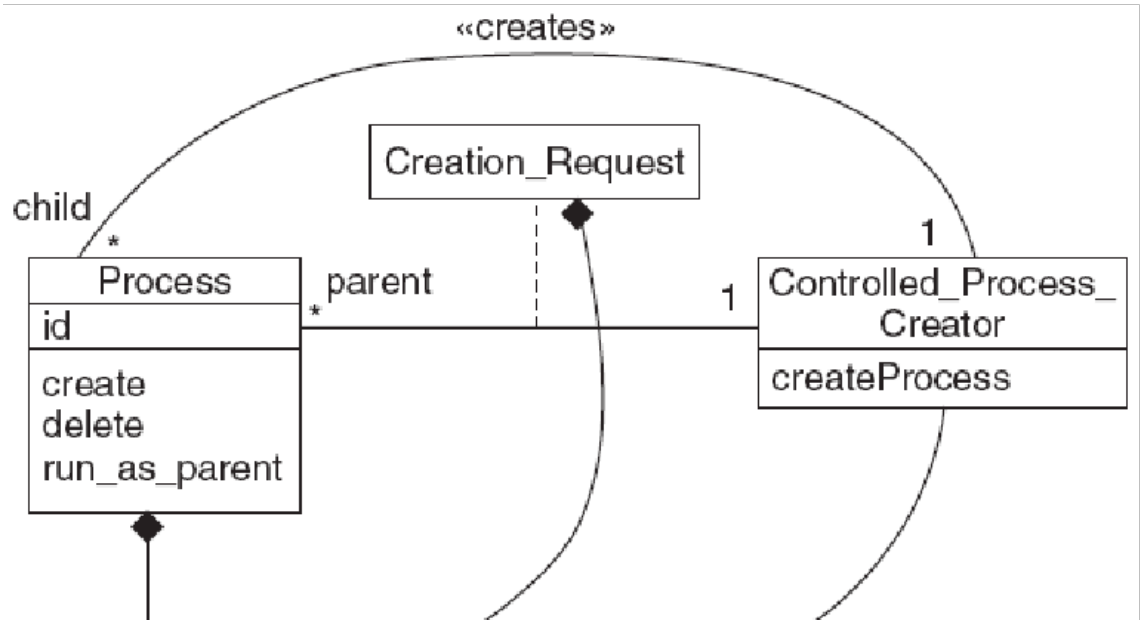


Figure 1: Structure of the controlled process creator.

Dynamics

The dynamics of the controlled process creator are depicted in Figure 2 .

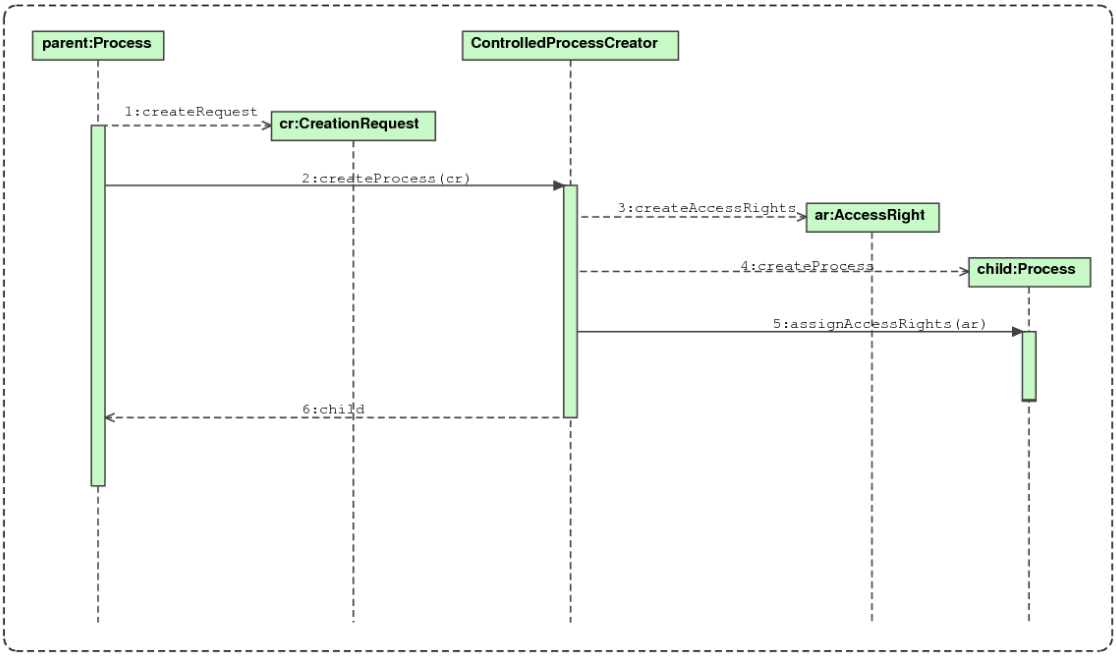


Figure 2: Dynamics of the controlled process creator.

Participants

The Controlled Process Creator is a part of the operating system in charge of creating processes. The Creation Request contains the access rights that the parent defines for the created child. These access rights must be a subset of the parentÂ¿s access rights.

Collaborations

A process requests the creation of a new process by submitting a Creation Request to the Process Creator. The Process Creator then creates both a new process and a set of access rights, based on this Creation Request, and assigns the access rights to the new process.

Implementation

For each required application of kernel threads, define their rights according to their intended function.

Pitfalls

(Nothing given)

Consequences

The following benefits may be expected from applying this pattern:

- The created process can receive rights according to required security policies.
- The number of children produced by a process can be controlled. This is useful to control denial of service attacks.
- The rights may include the parent's id, allowing the child to run with the rights of its parent.The following potential liability may arise from applying this pattern:
- Explicit rights transfer takes more time than using a default transfer.

Known uses

Credential Tokenizer

Pattern documentation

Quick info

Intent: You need a flexible mechanism to encapsulate a security token that can be used by different security infrastructure providers.

Problem

There are different forms of user credentials (also referred to as security tokens), such as username/passwords, binary security tokens (for example, X.509v3certificates), Kerberos tickets, SAML tokens, smart card tokens and biometric samples. Most security tokens are domain-specific. To encapsulate these user credentials for use with different security product architectures, developers have to modify the security token processing routine to accommodate individual security product architectures, which depends on the specific security specification the security product uses. A user credential based on a digital certificate will be processed differently than that of a Kerberos ticket. There is no consistent and flexible mechanism for using a common user credential tokenizer that supports different types of security product architectures supporting different security specifications.

Forces

- You need a reusable component that helps to extract processing logic to handle creation and management of security tokens instead of embedding them in the business logic or the authentication process.
- You want to shield off the design and implementation complexity using a common mechanism that can accommodate a security credential and interface with a supporting security provider that makes use of them.

Example

(Nothing given)

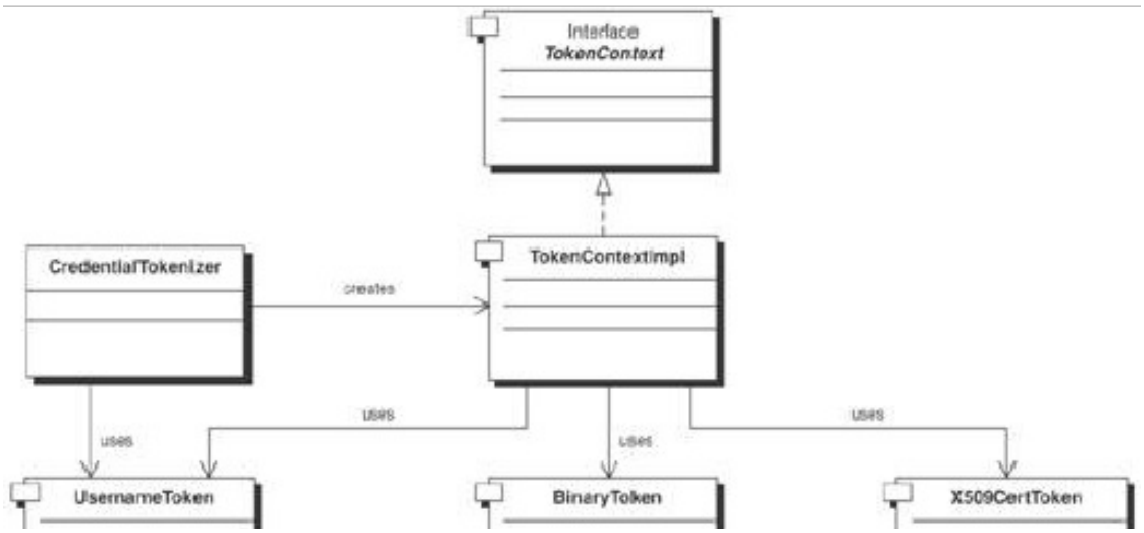
Solution

Use a Credential Tokenizer to encapsulate different types of user credentials as a security token that can be reusable across different security providers. A Credential Tokenizer is a security API abstraction that creates and retrieves the user identity information (for example, public key/X.509v3certificate)from a given user credential (for example, a digital certificate issued by a Certificate Authority). Each security specification has slightly different semantics or mechanisms to handle user identity and credential information. These include the following characteristics:

- Java applications that need to access user credentials or security tokens from different application security infrastructures.
- Web Services security applications that need to encapsulate a security token, such as username token or binary token, in the SOAP message.
- Java applications that support SAML or Liberty that need to include an authentication credential in the SAML assertion request or response.
- Java applications that need to retrieve user credentials for performing SSO with legacy applications.

Structure

The structure of the Credential Tokenizer is depicted in Figure 1 .



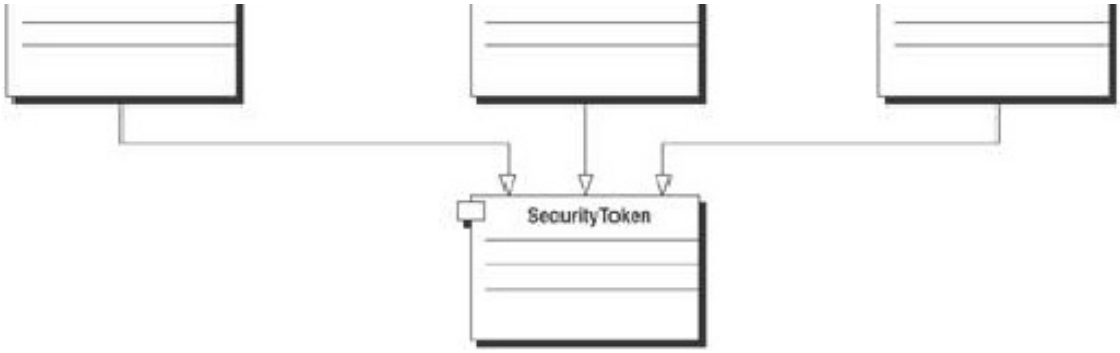


Figure 1: Structure of the credential tokenizer.

Dynamics

The dynamics of a Credential Tokenizer are depicted in Figure 2 .

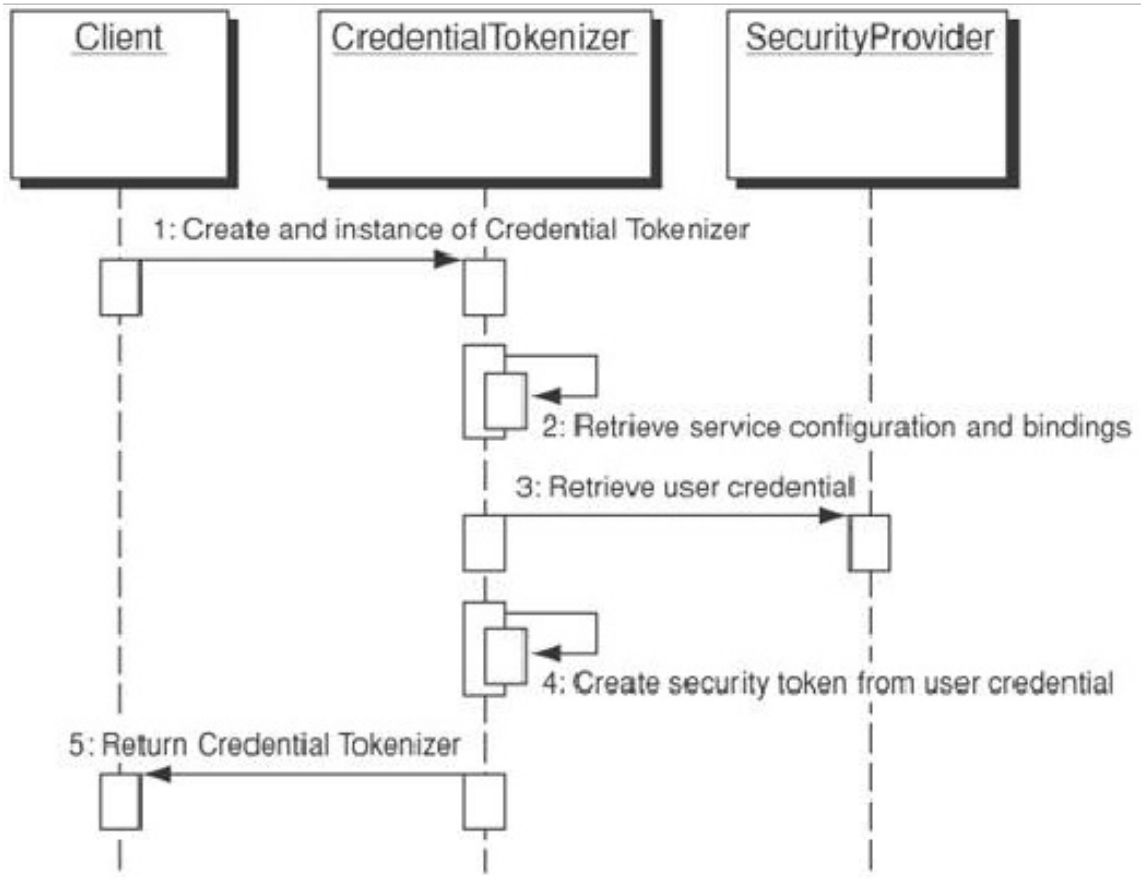


Figure 2: Dynamics of the credential tokenizer.

Participants

The Credential Tokenizer can be used to create different security tokens (SecurityToken), including username token and binary tokens (X.509v3certificate. When creating a security token, the Credential Tokenizer creates a system context (TokenContext)that encapsulates the token type, the name of the principal, the service configuration, and the protocol binding that the security token supports.

There are two major objects in the Credential Tokenizer: SecurityToken and TokenContext. The SecurityToken is a base class that encapsulates any security token. It can be extended to implement username token (UsernameToken), binary token (BinaryToken), and certificate token (X509v3CertToken). In this pattern, Username token is used to represent a user identity using Username Password. Binary tokens are used to represent a variety of security tokens that resemble a user identity using binary text form (such as Kerberos Tickets). Certificate tokens denote digital certificates issued to represent a user identity. An X.509v3certificate is a common form of certificate token.

The TokenContext class refers to the system context used to create security tokens. It includes information such as the security token type, service configuration, and protocol binding for the security token. This class defines public interfaces only to set or get the security token information. TokenContextImpl is the implementation for TokenContext.

Collaborations

As an illustration of the collaborations in a credential tokenizer, the Client may be a service requester that is required to create the Username Password-token to represent in the WS-Security headers of a SOAP message. The CredentialTokenizer denotes the credential tokenizer that creates and manages user credentials. The UserCredential denotes the actual Credential Token, such as username/password or a X.509v3digital certificate. The following sequences describe the interaction between the Client, CredentialTokenizer, and UserCredential:

- Client creates an instance of CredentialTokenizer.
- CredentialTokenizer retrieves the service configuration and the protocol bindings for the target service request.
- CredentialTokenizer retrieves the user credentials from SecurityProvider according to the service configuration. For example, it extracts the key information from an X.509v3certificate.
- CredentialTokenizer creates a security token from the user credentials just retrieved.
- Upon successful completion of creating the security token, CredentialTokenizer returns the security token to Client.

Implementation

To build a Credential Tokenizer, developers need to identify the service, authentication scheme, application provider, and underlying protocol bindings. For example, in a SOAP communication model, the service requestor is required to use a digital certificate as a binary security token for accessing a service end-point. In this case, the service configuration specifies the X.509v3digital certificate as the security token and SOAP messages and SOAP over HTTPS as the protocol binding. Similarly, in a J2EE application, the client is required to use a Client-certificate for enabling mutual authentication. In this case, the authentication requirements specify an X.509v3digital certificate as the security token and SOAP over HTTPS as the protocol binding, but the request is represented as HTML generated by a J2EE application using a JSP or a servlet.

Credential Tokenizer provides an API abstraction mechanism for constructing security tokens based on a defined authentication requirement, protocol binding, and application provider. It also provides API mechanisms for retrieving security tokens issued by a security infrastructure provider. Different strategies to implement a credential tokenizer exist:

- Service Provider Interface Approach: using a service provider interface approach to define the public interfaces for different security tokens will be more flexible and adaptive for different security tokens and devices. For example, certificate tokens may differ in vendor implementation. Developers can use the same public interfaces to support different credential token implementations and meet the requirements of different platforms and service providers without customizing the APIs for specific devices.
- Protocol Binding Strategy: as with the Assertion Builder pattern, it is possible that the same client may be using the Credential Tokenizer to encapsulate user credentials as a security token in a SOAP message. To accommodate such use, developers can employ a custom service configuration look-up function (for example, refer to getProtocolBinding method in the SSOContext discussed in SSO Delegator pattern)to determine the data transport and application environment requirements. In this way, the common processing logic of the user credential processing and security token encapsulation can be reused.

Pitfalls

The Credential Tokenizer pattern is essential to encapsulating user credentials and user information to meet authentication and non-repudiation security requirements. One important security factor for building reliable credential tokenizers is the identity management infrastructure and whether the keys are securely managed prior to the credential processing. The following are security factors and risks associated with the Credential Tokenizer pattern.

- Username password token. Username password tokens are highly vulnerable to attacks by using a password dictionary.
- X.509v3certificate token. Certificate token is a reliable security token and is stronger than Username Password token. However, it may be susceptible to human error during the management of the distribution of digital certificates and the timely revocation of certificates.
- Key management strategy. The security factor of key management strategy defines the process of generating key pairs, storing them in safe locations, and retrieving them. The generation of SAML assertion statements and signed SOAP messages using WS-Security is key management strategy. If the key management strategy and the infrastructures are not in place, the user credential token processing will be at risk.Should we use username/password as a security token? Some security architects insist that the username/password pair is not secure enough and should not be used as a security token. To mitigate the potential risk of a weak password, security architects should reinforce strong password policies and adopt a flexible security token mechanism such as Credential Tokenizer to accommodate different types of security tokens for future extension and interoperability.

What other objects can be encapsulated as security token?You can embed different types of security tokens in the Credential Tokenizer, not just username/password or digital certificate. For example, you can embed binary security tokens, because they can be encapsulated as a SAML token for an authentication assertion statement. In addition, you can also add the REL token (which denotes the rights, usage permissions, constraints, legal obligations, and license terms pertaining to an electronic document)based on the eXtensible Rights Markup Language (XrML).

Consequences

- Supports SSO. The Credential Tokenizer pattern helps in capturing authentication credentials for multifactor

authentication. It also helps in using"shared state"(the"shared state"mechanism allows a login module to put the authentication credentials into a shared map and then passes it to other login modules)among authentication providers in order to establish single sign-on, where the Credential Tokenizer can be used for retrieving the SSO token and providing SSOToken on demand for requesting applications.

- Provides a vendor-neutral credential handler. The Credential Tokenizer pattern wraps vendor-specific APIs using a generic mechanism in order to create or retrieve security tokens from security providers.
- Enables transparency by encapsulating multiple identity management infrastructures. The Credential Tokenizer pattern encapsulates any form of security token as a credential token and thus eases integration and enables interoperability with different identity management infrastructures.

Known uses

(Nothing given)

Demilitarized Zone

Pattern documentation

Quick info

Intent: Any organization conducting e-commerce or publishing information over Web technologies must make their service easily accessible to their users. However, any form of Web site or e-commerce system is a potential target for attack, especially those on the Internet. A Demilitarized Zone (DMZ)separates the business functionality and information from the Web servers that deliver it, and places the Web servers in a secure area. This reduces the ``surface area"of the system that is open to attack.

Problem

Internet technology systems, particularly those facing the public Internet, are regularly subject to attacks against their functionality, resources and information. How do we protect our systems from direct attacks?

Forces

Solving this problem requires you to resolve the following forces:

- The cost of an extensive security solution will be high, but the cost of an intrusion may also be high in terms of system damage, theft and loss of customer confidence. If the potential rewards from the attack are high in terms of financial gain or publicity, the risk of such an attack will be higher. The scope, and hence cost, of any countermeasure must be commensurate with the level of perceived threat and the potential cost of the intrusion.
- To prevent attack, we must make intrusion into any part of the system as difficult as possible, especially an organization's internal business systems. However, increasing the level of security will generally make the system more difficult to use, which conflicts with the goal of making the system open and easy for legitimate users.

Example

A commercial Internet system holds customer profiling information, dealer order information and commercially-sensitive sales information, any of which could be stolen or corrupted by an attacker. This information must be shared with the organizationÂ¿s corporate systems, making them liable to attack as well. You could use a firewall to control access to your systems from the outside world as shown below.

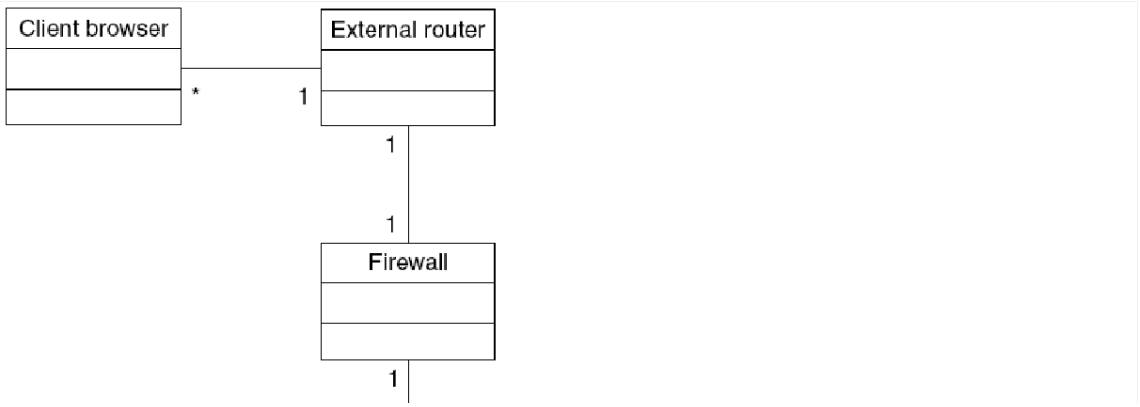
The firewall would be configured to allow only inbound traffic to access the Web server. However, this places a large onus on the system administrators to configure the firewall correctly, and on the firewall software to operate correctly. If the firewall fails, an attacker could potentially have direct access to other business resources such as the SAP system or mainframe shown in the diagram. The configurationof the firewall is further complicated by the fact that for any highly-available Webbased system, multiple servers must be exposed to support either load balancing or failover. If the Web-based system is also high-functionality, additional protocols must be allowed through the firewall. All of this makes a configuration error more likely.

Solution

Provide a region of the system that is separated from both the external users and the internal data and functionality---commonly known as a demilitarized zone (DMZ). This region will contain the servers, such as Web servers, that expose the functionality of the Web-based application. Restrict access to this region from the outside by limiting network traffic flow to certain physical servers. Use the same techniques to restrict access from servers in the DMZ to the internal systems.

Structure

The structure of a DMZ is depicted in Figure 1 .



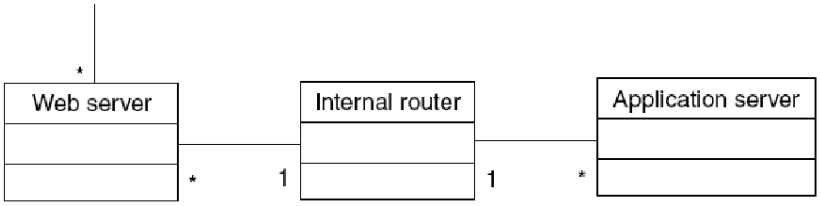


Figure 1: Structure of a demilitarized zone.

Dynamics

The dynamics of a DMZ are depicted in Figure 2 and Figure 3 .

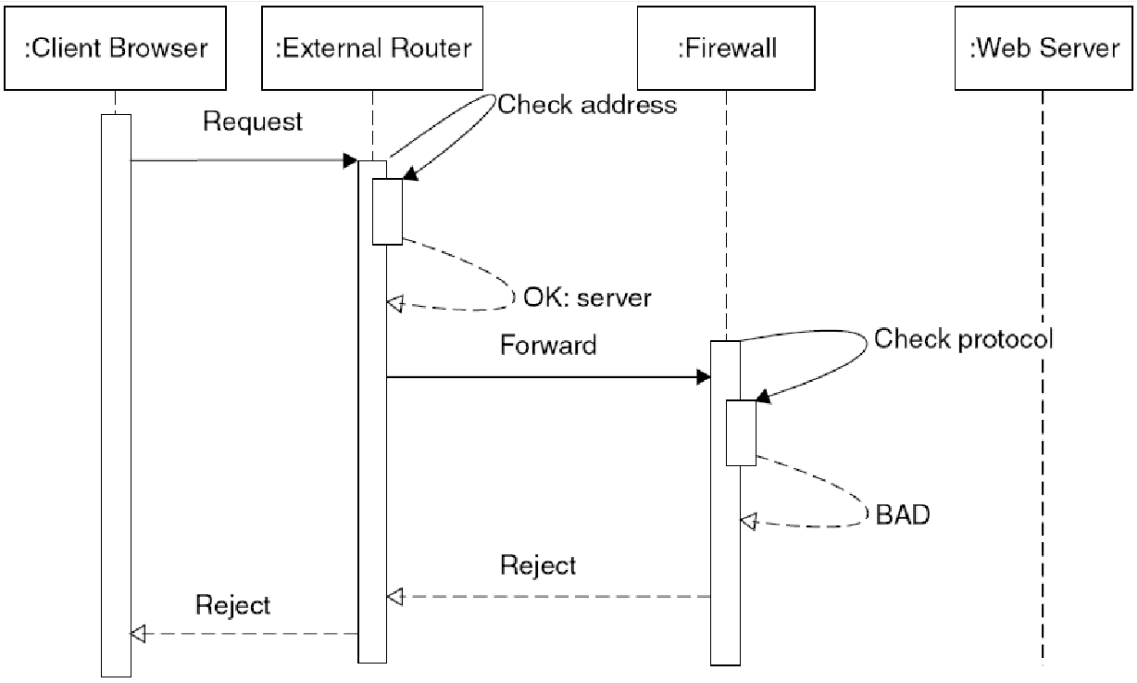


Figure 2: Dynamics of a DMZ: rejecting a request.

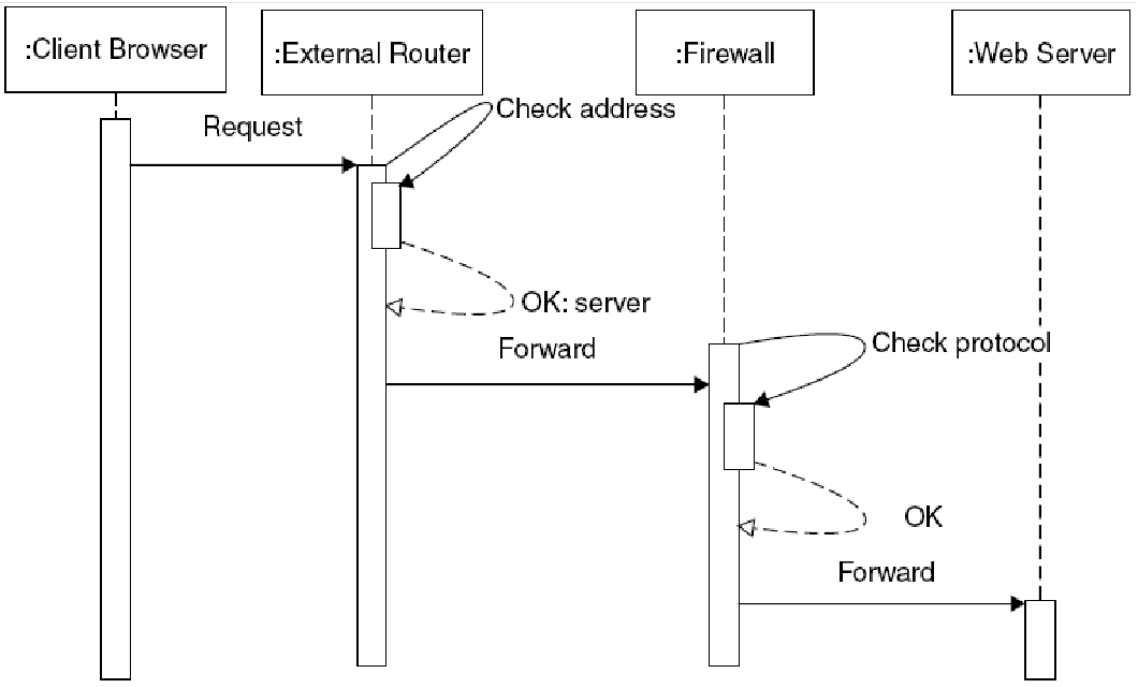


Figure 3: Dynamics of a DMZ: allowing a request.

Participants

- External router, a filtering router whose principal responsibility is to ensure that all inbound traffic is directed to the firewall. Its secondary responsibility may be to keep out random traffic generated by attackers.
- Firewall, responsible for receiving inbound requests from the external router and subjecting them to more sophisticated

analysis, such as stateful inspection. If a request is judged to be legitimate, it will be forwarded to an appropriate Web server.

- Web servers, providing access to the applicationÂ¿s functionality and information. There may be multiple Web servers that are accessed through a load balancer. A Web server will receive a request from the firewall and service that request. A request for a static resource, such as a fixed page of HTML or an image, may be delivered from a cache held on a local disk. A request for a dynamic resource will be proxied through to an application server that is shielded from the outside world in the style of a protection reverse proxy. No application functionality, such as servlets or ASP. NET pages, will run on the Web servers, as this makes them open to direct attack. Although described here as ``Web"servers, these servers may support access through other protocols such as FTP.
- Internal router, a filtering router whose principal responsibility is to ensure that it only passes legitimate traffic from the Web servers through to the internal network.
- Application servers, a platform on which the application's code runs, typically in the form of Web components such as servlets and business components such as EJBs.

Collaborations

The first scenario in Figure 3 shows a successful client request for some business functionality. The client browser request is filtered by the external router to ensure that it is destined for a valid server. The request is forwarded to the firewall to undergo more rigorous checking. If the firewall is happy with the protocol use, the request goes onwards to the server requested by the client.

The second scenario shows a malicious client call being blocked by the firewall. The client browser request is again filtered by the external router to ensure that it is destined for a valid server. The request is then forwarded to the firewall to undergo more rigorous checking. At this stage, the firewall detects invalid protocol use---maybe some form of protocol-based attack, or an attempt to flood the server. The request is rejected and the suspicious activity is logged. See Figure 2 .

Implementation

Since the request handling and business functionality must be separated by a filter, it is best to use dedicated web and application servers where any programmatic functionality, whether business or presentation, is deployed on an application server that is physically separate from the Web server. These application servers can be placed on a more protected network than the Web servers. This protected network will have easier (possibly direct)access to the corporate information and services required by the Web-based application.

The external router should be configured to deny any attempted access to any network addresses outside of those known in the DMZ. To increase security, any requests with a destination address that does not match the Web server address (or that of the Web server cluster)may be rejected. The external router may also reject requests based on the port number of the request, for example rejecting any request that is not for port80. The external router will therefore block direct attacks on the internal router, and possibly the firewall.

The Web servers will be built solely for the purpose of delivering static Web content or proxying requests through to the application servers. These Web servers should be locked down (or ``hardened")by removing unnecessary functionality. Such hardening helps to prevent other, unintended, access to the servers.

The internal router will limit network traffic to connections between the Web servers on the DMZ and specific internal servers, such as the application servers, using a fixed set of protocols. This restriction reduces the risk of attack on other internal systems. The use of an internal router helps to reduce the risk of attack should the external router be breached. Because of this threat, no traffic should be allowed directly from the external router to the internal router. The whole operation of the routers and the traffic filtering may be controlled from a machine running specific firewall software. This makes it easier to apply consistent rules to the routers and to use statistical analysis to detect potential attacks. The firewall applies more sophisticated traffic filtering rules to detect more complex attacks. Depending on the type of firewall, the network traffic may or may not pass through the firewall itself.

Because the number of servers exposed to the outside world is reduced, it means that fewer parts of the system need a high level of security. In the scenario described, the application servers will not need to be hardened to the same level as the Web servers. To access those servers not directly exposed (and hence less securely configured), any attacker will have to breach several security elements that form part of the DMZ. Hopefully, they will set off various intruder alerts as they do soÂ¿if, indeed, they are capable of doing so.

Applying a DMZ to a system is a good way to provide protection for the system. However, you must remember that protecting the platforms on which the system is built is only part of the solution. Since security is a matter of policy as well as technology, all protection mechanisms---such as a DMZ---must be backed up with appropriate procedures and processes to ensure that the level of security remains high. If there is a high level of concern about possible attacks on the system, an intrusion detection system (IDS)may also be used. An IDS monitors the traffic on the network, or on specific hosts, looking for suspicious activity. If the IDS identifies a pattern of network or host traffic that indicates an attack is underway, it will notify the system administrators. An IDS could be used on the DMZ itself, on the internal network, or both.

Implementation variants include:

- Multi-homed firewall. The number of machines involved in implementing the DMZ will vary according to the level of protection required (based on anticipated risk)and the amount of money available. In the simplest case, the DMZ may be partitioned using a single firewall machine. This machine will have three network cards: one connected to the Internet, one connected to the internal network and one connected to a dedicated LAN containing only the Web servers and any other ``public facing"parts of the system. The firewall software running on the machine will manage the traffic between the three networks to maintain three separate security zones. The benefits of such an ``multi-homed host"implementation include reduced cost and ease of maintenance. However, this system creates a single point of failure, both in terms of security and availability. It also means that any attacker is only one system away from gaining access to the sensitive internal systems.
- Firewall as filter. A multi-homed firewall host may be used in place of the external or internal router. This means that all traffic must pass through the firewall (and its filtering rules)to reach the internal network or the DMZ itself.
- Stealth firewall. Rather than relaying traffic, the firewall may simply be attached to the demilitarized network and act inÂ¿stealthÂ¿mode, simply monitoring traffic for potential intrusion. This can make the firewall itself more difficult for an intruder to detect.

Pitfalls

(Nothing given)

Consequences

The following benefits may be expected from applying this pattern:

- Security is improved, because fewer systems are exposed to attack and multiple firewall artefacts must be breached to compromise security.
- The level and depth of protection can be varied to match the anticipated risk and the cost limitations.
- The additional security is transparent to the users of the system functionality and to the developers of such functionality.
- Fewer hosts must be hardened to withstand attack than if they were all exposed to the outside world.The following potential liabilities may arise from applying this pattern:
- Availability may be impacted, because the firewall becomes a single point of failure. The standard procedure is therefore for a firewall to ``fail closed"---that is, in the event of failure, it will deny all connections to the protected systems.
- Manageability is impacted, because the very restrictions that limit access to internal data may make it difficult to access the application from an internal monitor.
- Cost is increased, because extra elements must be procured to build the DMZ. These include not only the filtering routers, firewall software and firewall host, but also the extra network equipment, such as switches and cabling, used on the DMZ itself.
- Performance is impacted due to the overhead of network traffic filtering. Performance is also impacted as it becomes necessary physically to separate the Web servers from the application servers. If this has not already been done to improve another non-functional characteristic, it must be done to implement a DMZ, and so will add multiple extra network hops for each user transaction.

Known uses

DMZs are extremely common for almost all Internet sites and advice on the creation of DMZ configurations is offered by almost all major network hardware and software vendors, such as:

- Sun http://www. sun. com/executives/iforce/solutions/SecuritySolnIIFinal3. pdf
- Microsoft http://www. microsoft. com/windows2000/techinfo/reskit/en-us/default. asp?url=/windows2000/techinfo/reskit/en-us/deploy/dgcf_inc_icku. asp
- Cisco (variously described as part of their SAFE Blueprint)

Encrypted Storage

Pattern documentation

Quick info

Intent: The Encrypted Storage pattern ensures that even if data is stolen, the most sensitive data will remain safe from prying eyes.

Problem

Web applications are often required to store a great deal of sensitive user information, such as credit card numbers, passwords, and social security numbers. Although every effort can be taken to defend the Web server, one can never be sure that some new vulnerability wonâ€™t be discovered, leading to the compromise of the server. Hackers are known to specifically target this sort of information.

Historically, Web sites that have experienced the loss of sensitive customer data have found it very difficult to recover from the adverse publicity. While many sites have recovered from the shame of being defaced, the large-scale loss of credit card numbers is a catastrophic failure. Ultimately, it is always preferable not to store sensitive data. However, sometimes it is not avoidable. For example, credit card transactions are often not a single event. If an item is back ordered or the user requires a refund, the site must be able to access the credit card number that was used. Similarly, many government and financial sites rely on the social security number as the primary identifier for American users. These sites need a better approach to protecting this data.

Forces

(Nothing given)

Example

The UNIX password file hashes each user's password and stores only the hashed form. Several Web sites with which we are familiar use encryption to protect the most sensitive data that must be stored on the server. All use variations on this pattern.

Solution

The Encrypted Storage pattern encrypts the most critical user data before it is ever committed to disk. Before it can be used, it is decrypted in memory. If the Web server is compromised, an attacker may be able to steal the data store, but will not be able to gain access to the sensitive data.

In the most straightforward approach, each userâ€™s data is protected using a single key. Under this solution, the application server maintains a single key that is used to encrypt and decrypt all critical user data. The key should be stored in a secure fashion, and especially not in the same data store as the protected data.

The key should be changed occasionally. This requires that the system be able to decrypt data using the old key and re-encrypt it using the new. Because of the complexity of encrypting and decrypting data on the fly, this should be performed with the database off-line during a period of downtime. If downtime is not possible, a large key should be selected with the expectation that it will not be changed.

Structure

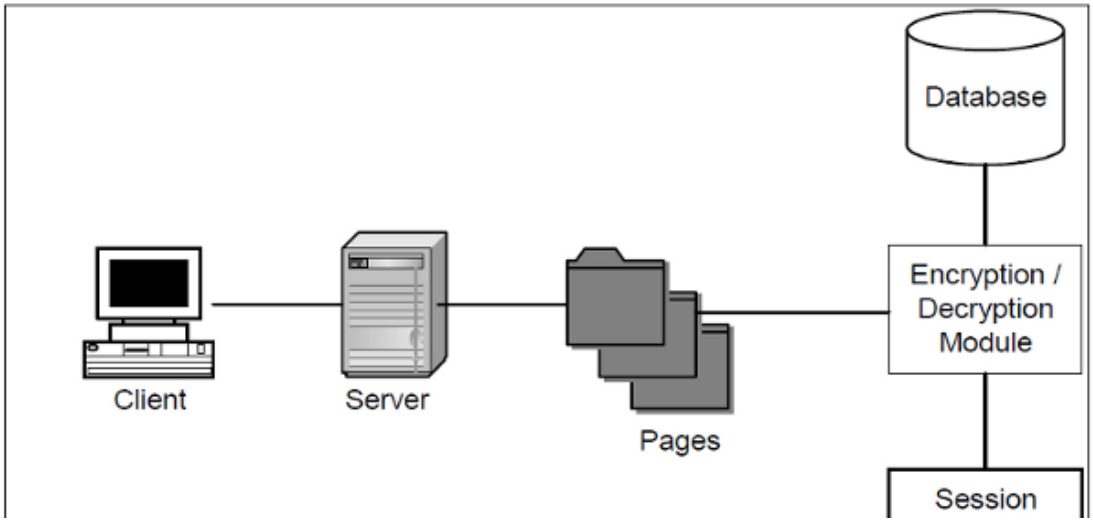


Figure 1:Encrypted storage

Dynamics

Server startup:

- The server loads the key into the encryption module
- The server takes protective measures to ensure that the key cannot be further accessed

Receipt of sensitive data:

- The client submits a transaction containing sensitive data
- The server submits the data to the encryption module
- The server overwrites the cleartext version of the sensitive data
- The sensitive data is stored in the database with other user data and an identifier for the sensitive information

Use of sensitive data:

- A transaction requiring the key is requested (usually from the client)
- The transaction processor retrieve the user data from the database
- The sensitive data is submitted to the encryption module for decryption
- The transaction is processed
- The cleartext sensitive data is overwritten
- The transaction is reported to the client without any sensitive data

Key refreshing:

- A utility program is started and loaded with both the old and the new key
- Each user record in the database is converted individually.

Participants

(Nothing given)

Collaborations

(Nothing given)

Implementation

(Nothing given)

Pitfalls

Never echo the sensitive data to the user. If you need to differentiate among several credit card numbers, display only the last four digits of the card. These should be stored in the database along with the encrypted card number. Both performance and security could suffer if the card numbers are decrypted every time the last four digits are required.

Do not rely on any Operating System-level encrypting file system. Encrypting file systems are adequate for defending against a lost hard drive. But if the system is compromised by a remote attacker, the attacker will gain some sort of toehold on the system. In that case, the operating system will dutifully decrypt all data as it is requested from the file system and deliver it to the attacker.

The following general principles should be followed:

- Never attempt to invent an encryption algorithm. Use a tested algorithm from Applied Cryptography.
- If possible, use a freely available library rather than coding one from scratch.
- After sensitive data is used, the memory variables containing it should be overwritten.
- Care must be taken to insure that sensitive data is not written into virtual memory during processing.
- Use only symmetric encryption algorithms. Asymmetric (public/private) algorithms are too computationally expensive and could easily result in processor resources being exhausted during normal usage.

Protection of the Key

If at all possible, the key should not be stored on the file system. There are COTS devices available that provide the system with a physical encryption card. These devices offer performance benefits, and also guarantee that the key cannot be read off the device. The key is manually loaded into the card, the encryption takes place on the card, and the key cannot be extracted from the card. The only downside to this approach is the cost â€“ both the cost of purchasing the hardware and the development and maintenance costs of programming around it. A cheaper alternative to loading the key is to require that an administrator load the key at system start, either from removable media or using a strong passphrase. This reduces the risk of having the key on-line, but does expose the key to a fair number of different people. This approach may sacrifice some availability because an operator

must manually intervene whenever the system restarts.

If neither of these approaches is feasible, the Web server can read the key value from a file at server startup. This approach ensures that the server can restart unattended, but puts the key at risk if the system is compromised. To reduce this risk, use one or more of the Server Sandbox pattern techniques, even going so far as to chroot the server so it cannot see the key file once it has completed its initialization process.

Unless a hardware encryption device is used, the server will have to maintain a copy of the encryption key in RAM in order to decrypt data as it is needed. Minimize the code modules that have access to the key. And if the operating system supports it, mark the decryption module so that it will never be swapped to disk. Also be aware that a coredump file might contain the key â€“ these should be disabled on a production server.

In addition to protecting the key from attackers, the key must also be protected from conventional loss. The loss of the key would be catastrophic, since all user data would become inaccessible to the server. Maintain multiple backups of the key at various off-premises locations. Recognize that multiple keys increase the risk that one could be stolen, and take care to protect them all.

Variation: One Key Per User

This alternative is similar to the Password Propagation pattern in that it requires that the individual userâ€™s password be available in order to gain access to that userâ€™s data. The server itself does not even have a key that will allow access to a userâ€™s data. It is not really applicable to the protection of credit card numbers, as those numbers must be available to the server even when the user is not connected.

In this approach, the userâ€™s password is used to encrypt the data that is sensitive to that user. To decrypt the data, the user must again provide their password, which is never stored in decrypted form. Because decryption of the data requires the user to provide his/her password, and because that password is not known outside of the context of an authenticated user transaction, the site administrator has no access to that data.

If the password itself is stored in the data, it should be stored in hashed form, using a different algorithm than the hash function used to encrypt the sensitive data. If the password is stored in plaintext or hashed using the same algorithm, the attacker will have the key needed to decrypt the data.

If the user changes his/her password, the data must be decrypted using the old password and reencrypted using the new. If the user loses his/her password, encrypted data will be lost. Data protected in this way must be data that can be recovered through some other means, such as the user providing it again.

Possible Attacks

There are a number of possible attacks that could be perpetrated against this pattern:

- Search of virtual memory â€“ if sensitive data is paged out of RAM into a disk-based virtual memory file, it may be possible for an attacker to search the pagefile for obvious data patterns (such as numeric strings that are recognized as credit card numbers).
- Key capture â€“ an attacker will attempt to gain access to the key used to encrypt the data
- Dictionary attack â€“ when encryption keys are generated from passwords, password-guessing attacks are generally much less difficult than exhaustive search of all possible keys.
- Race condition attack â€“ an attacker may be able to capture the data before it has been encrypted.

Consequences

(Nothing given)

Known uses

(Nothing given)

Firewall

Pattern documentation

Quick info

Intent: Control incoming and outgoing network connections, restrict access to certain hosts on the network level.

Problem

Beside regular users, attackers can probe, access and misuse any system inside of the internal network.

It is unlikely that the access control facilities of all internal systems are activated and configured appropriately. In particular, out-of-the box installations offer standard services which can be misused by an attacker. Even if there are access restrictions it is unlikely that they are consistent, especially when more than one administrator is involved and there are no ``global''guidelines.

Even worse, we can assume that most internal systems are not hardened: experience shows that patches are not applied in time and that many, often unneeded services are running. This makes internal system also vulnerable to attacks at a rather low level: script-kiddies download exploitation software and conduct random attacks (not to speak of deliberate attacks carried out by an experienced attacker).

Another basic threat is that the overall network topology is visible, i. e. an attacker can analyze possible targets without further burden.

Furthermore, it might happen that attacks can not even be detected as one cannot ensure that the audit facilities of the internal systems are activated and configured appropriately.

Therefore, you should restrict the ingoing and outgoing traffic at the border between the internal and the external network.

The regular router has to be replaced with a firewall system (or extended accordingly)that implements the following functions: analysis, access control, filtering and modification.

The firewall must be able to analyze the messages which are sent through it. The analysis itself can be conducted in several ways which differ in parameters such as technical implementation, quality, or granularity. For example, it is possible to analyze the semantics of specific protocol headers, the different states of a protocol, or the relationship between several parallel connections.

Based on the information gathered before, a firewall will be able to make an access control decision. First, the message has to be identified somehow, e. g. by the identity of the user, the process Id of the corresponding application or network addresses. With the given information the firewall can now decide whether a particular message is dangerous or not (hereby, a set of rules is usually evaluated). Possible access control actions are to grant, reject, discard, or modify messages.

Forces

The following forces contribute to the problem:

- Authorized users want to be able to use the network without being hindered or blocked in doing their work.
- Unauthorized users have to be denied access to the network.
- A network can have multiple access points, making security policy enforcement hard.
- We want to be able to provide a basic level of protection, even (especially?)with regard to unsecured and unpatched hosts.
- Some services are intended for internal use only, we want to limit external access to these services (but keeping them accessible from the internal net).
- Abusive network traffic, e. g. denial-of-service attacks, should be blocked at the perimeter of the network.

Example

A real world example for a firewall is the guard of a prison in a more oppressive country. The guard can examine all letters for and from the prisoners. He is basically in a position to decide whether a letter is passed or not.

Solution

An interceptor, the, is placed between the internal and external network. All incoming or outgoing connections pass through this host, i. e. it is a ``choke point''. The network administrator can then, by configuring the firewall with a policy, decide which network connections are allowed and which are not and enforce these decisions.

Structure

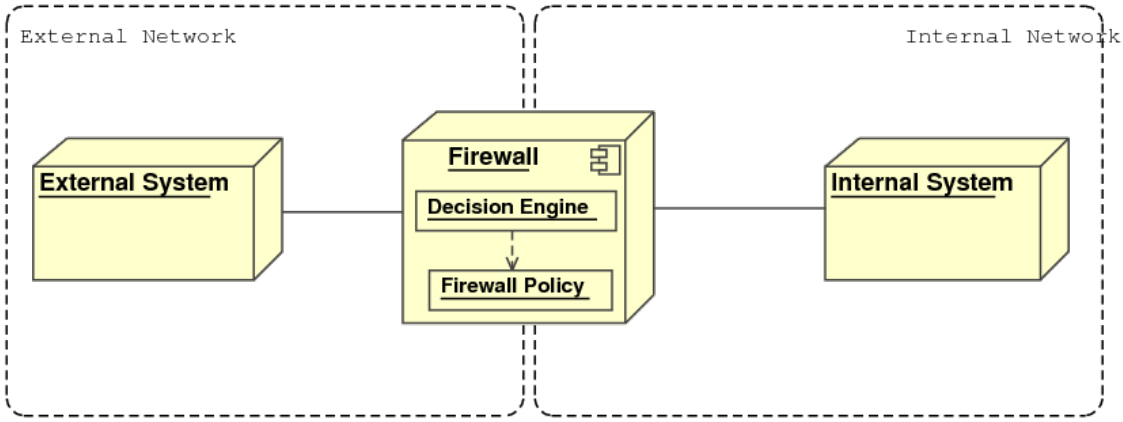


Figure 1: Structure of a firewall, separating the internal from the external network.

The high-level structure of a firewall is shown in Figure 1 . In it, the external network (containing the external system)is separated from the internal network (containing the internal system). This forces all communication between the internal and external system to pass through the firewall.

The firewall contains a decision engine which is configured with a policy (provided by the network administrator). Depending on this policy and information gathered from the network level, the firewall will allow or deny certain communications between the internal and external system.

Dynamics

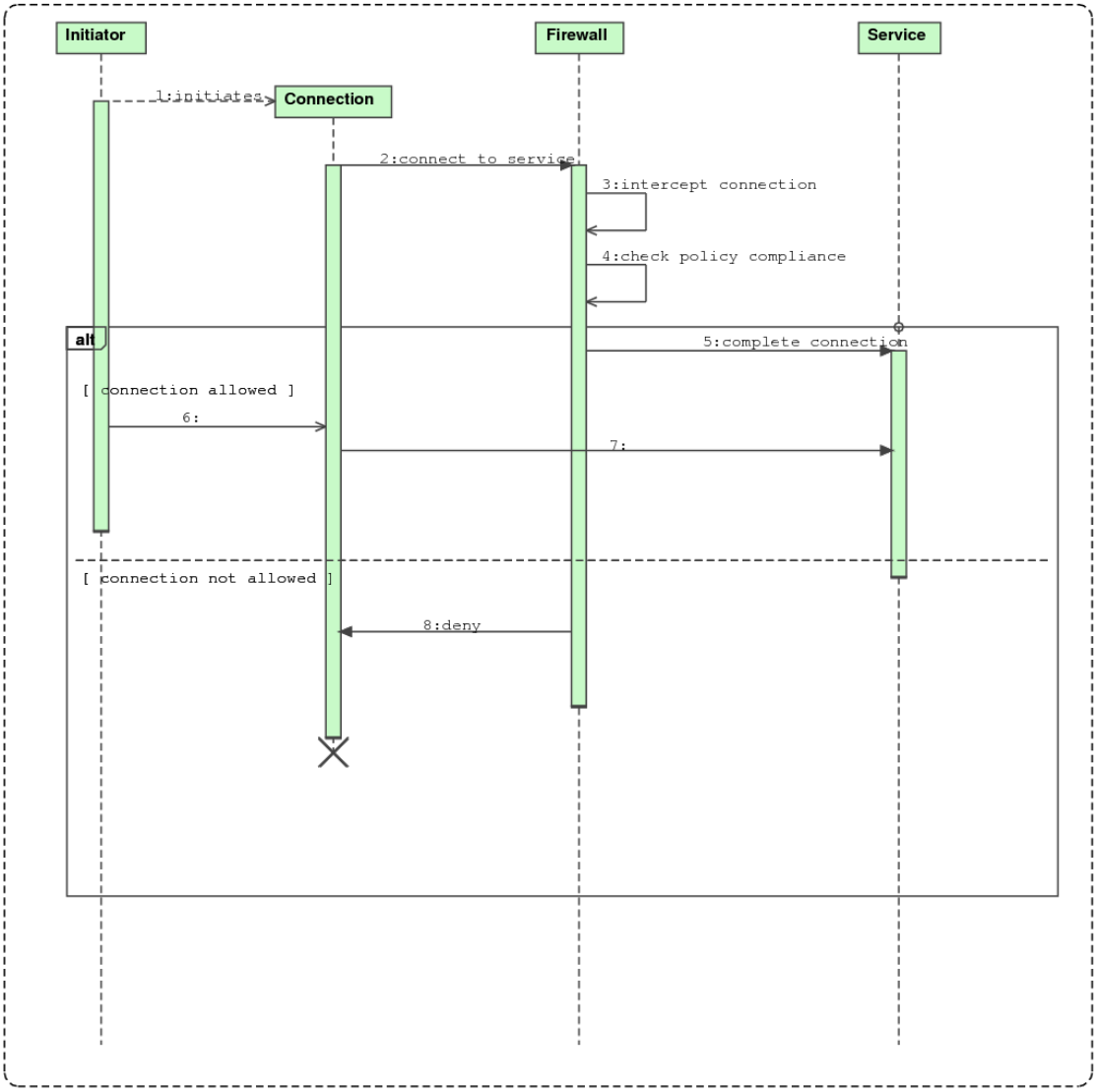


Figure 2: Intercepting and allowing/denying a connection.

A graphical overview of the process of intercepting and allowing or disallowing a network connection is given in Figure 2 .

Participants

- The **Initiator** is a network host that an be on the internal or the external network. This host initiates communications between itself and the service it wants to invoke by creating a new connection.
- The **Firewall** intercepts the connection before it is established, and then either allows its creation or prevents it from being established.
- The **Service** is a network host on the opposite side of the firewall; i. e. if the initiator is internal, then the service is external and vice versa. It offers a service to the initiator.

Collaborations

The **Initiator** initiates a new **Connection** to the **Service** . This connection is intercepted by the **Firewall** , and then checked if it conforms to the firewall policy. If the connection is allowed, the firewall completes the connection and the **Initiator** is allowed to communicate with the **Service** . Otherwise, if the connection is not allowed, the **Firewall** terminates it.

Implementation

Two implementation variants of a firewall are mentioned: the stateless firewall and the statefull firewall. In general, the stateless firewall is more performant and the statefull firewall is more advanced, i. e. it is able to make more intelligent decisions.

The stateless firewall does not take the state of the network connection into account. In the case of a TCP/IP network, for example, certain packets (either TCP or UDP)to and from a certain IP address and port are either allowed or disallowed. However, the stateless firewall is not able to distinguish a packet that belongs to a valid TCP connection from a ``rogue"packet.

The statefull firewall, on the other hand, maintains a table of current network connections and their state, i. e. (in the case of a TCP connection (description from the Linux manpage on netstat)):

- **syn_sent** The socket is actively attempting to establish a connection.
- **syn_recv** A connection request has been received from the network.
- **established** The socket has an established connection.
- **fin_wait1** The socket is closed, and the connection is shutting down.
- **fin_wait2** Connection is closed, and the socket is waiting for a shutdown from the remote end.
- **time_wait** The socket is waiting after close to handle packets still in the network.
- **closed** The socket is not being used.This statefull information allows the statefull firewall to distinguish packets that belong to an active connection from other packets, enabling it to make more intelligent enforcement decisions.

Pitfalls

Before deploying a firewall, the network administrator must be aware of the following pitfalls:

- Since the firewall operates on the network level, it is application-agnostic. While this design decision keeps the firewall platform and application independent, it also entails that the firewall is unaware of the events on the application level: an allowed connection can still contain malicious traffic, such as buffer overflows or SQL injections.
- While certain protocols are commonly associated with a known range of ports (i. e. HTTP connections are TCP connections to port80), this need not be the case: an HTTP service could be configured to listen to another port as well. As such, blocking connections to and from a certain port is no guarantee that communication using this protocol is prevented.
- Certain protocols, such as FTP, use different connections for transmitting both data and control sequences. It can be particularly tricky to write policies that enforce control over these protocols correctly.
- A firewall is impaired by a reverse proxy (or potentially all non-transparent intermediaries), as the network connections do not appear to be originating from the proxied servers anymore, but from the proxy itself. The firewall should be positioned so that it is able to make the necessary distinctions between authorized and unauthorized connections.

Consequences

The consequences of applying a firewall between two networks can be summarized as follows:

- Accountability: As a firewall analyzes all messages which pass through it, rather fine-grained log information can be generated easily. Thus it is possible to detect possible attacks and to hold regular users responsible for their actions.
- Availability: A firewall also helps to increase the availability of internal systems as the ``attack surface"is made smaller significantly.
- Confidentiality/Integrity: As there is an additional line of defense, the confidentiality and integrity of information hosted at the internal systems is increased, too.
- Manageability: A firewall is an additional, complex component of the network infrastructure. Thus, the efforts for managing the network are higher.
- Usability: Integrating a firewall often requires to change applications or the users behavior. In either way, a firewall has an impact on the usability.

- Performance: The analysis and access control decision process consumes additional processing time. Thus, every firewall decreases the performance. Especially in high-bandwidth environments, this is an important issue.
- Cost: There will be additional costs for setting up and maintaining a firewall.

Known uses

Some widely deployed firewalls are, Linux'IPTABLES, IPFW and PF from the BSD family.

Full View with Errors

Pattern documentation

Quick info

Intent: Prevent users to perform illegal operations by showing an error message when the user tries to perform an illegal operation.

Aliases: Full View With Exceptions, Reveal All and Handle Exceptions, Notified View

Problem

Users should not be allowed to perform illegal operations.

Forces

- Users may be confused when some options are either not present or disabled.
- If options pop in and out depending upon roles, the user may get confused on what is available.
- Users should not be able to see operations they are not allowed to do.
- Users should not view data they do not have permissions for.
- Users do not like being told what they cannot do.
- Users get annoyed with security errors, permission denied, and illegal operation messages.

Example

(Non-security)Once an officer is allowed on a military base, he or she could go to any building on the base. In effect, the officer has a full view of the buildings on the base. If the officer tries to enter a restricted area without proper clearance, either someone would stop and check them noting that they are not allowed in the restricted area, or alarms would sound and guards would show up to arrest the officer.

Graphical applications often provide many ways to view data. Users can dynamically choose which view on which data they want. When an application has these multiple views, the developer must always be concerned with which operations are legal given the current state of the application and the privileges of the user. The conditional code for determining whether an operation is legal can be very complicated and difficult to test. By giving the user a complete view to what all users have access to can make teaching how to use the system easier and can make for more generic GUIs.

Solution

Design the application so users see everything that they might have access to. When a user tries to perform an operation, check if it is valid. Notify them with an error message when they perform illegal operations.

This pattern is very useful when a user can perform almost any operation. It is easier to show the user everything and just provide an error message when an illegal operation is attempted.

The solution for this pattern is simple when only a few error message need to be displayed. Just display the error message to standard error or in a dialog box. If many error messages are spread throughout the application, a separate error reporting mechanism may be useful. This mechanism could also be used for error logging.

Typically, an error-reporting framework would have two principal components. The log event object has a message describing the error condition and a severity level indicating if the event is a warning, an error, or just user information. When a log event is created it can automatically register itself with the logger. The logger is a Singleton that automatically receives and processes log events. The logger can be configured to display dialogs or write to a file depending on the severity of the event.

Structure

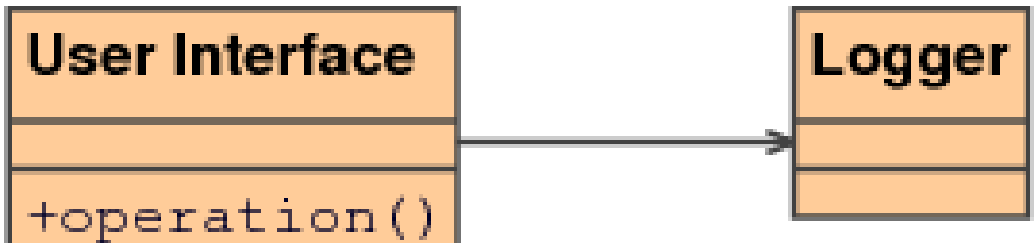


Figure 1: Structure of the Full View with Errors pattern.

See Figure 1 for an overview of the pattern when a separate error reporting mechanism is used. When no such mechanism is used, the user interface itself incorporates the logger functionality.

Dynamics

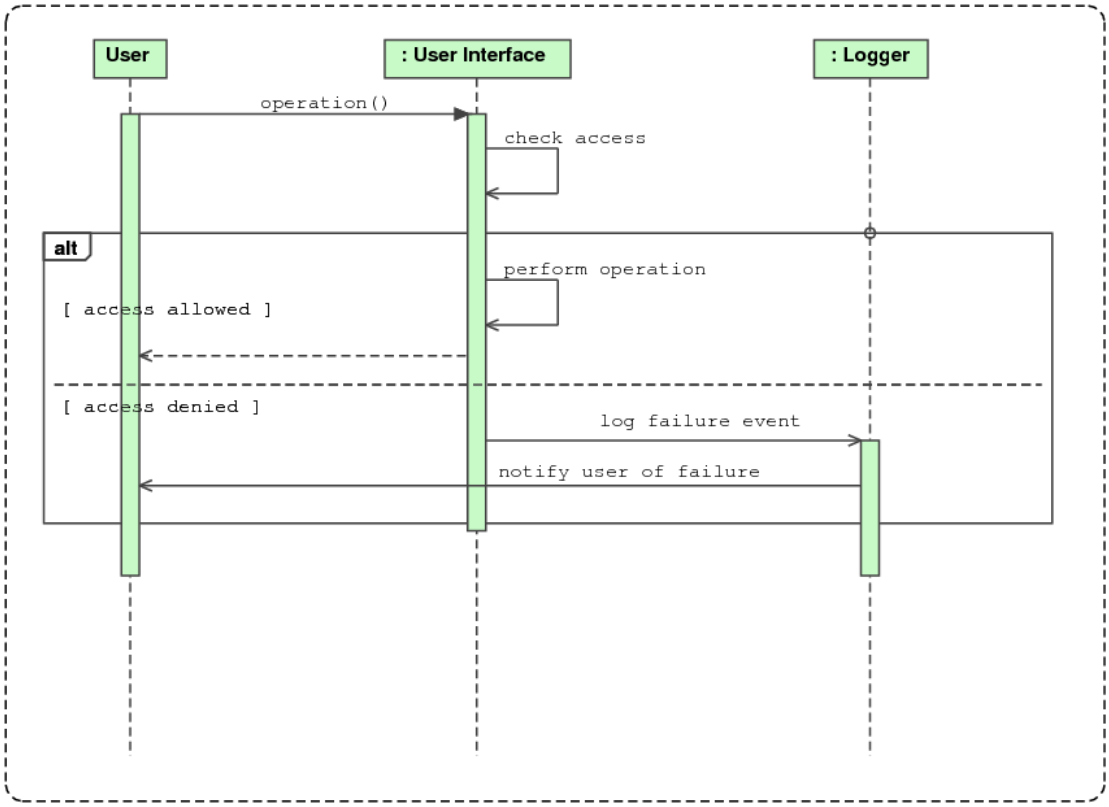


Figure 2: Event sequence for the Full View with Errors pattern.

See Figure 2 for an overview of the event sequence when a separate error reporting mechanism is used. When no such mechanism is used, the user interface itself incorporates the logger functionality.

Participants

See also Figure 1 .

- User Interface The user interface is responsible for offering all operations to the user.
- Logger The logger is responsible of taking appropriate action when a log event is

Collaborations

See also Figure 2 .

- The user performs an operation using the user interface.
- The user interface validates whether the execution of the operation should be allowed or not.
- If the operation is allowed, the user interface executes the operation and possibly returns the result to the user.
- If the operation is disallowed, the user interface notifies the logger of the failure. The logging component then takes appropriate action (for example based on the severity of the failure). This action may be displaying a notification to the user or logging the event to a file.

Implementation

(Nothing given)

Pitfalls

(Nothing given)

Consequences

- Training materials for the application are consistent for each type of user.

- Retrofitting this pattern into an existing system is straightforward. Just write a GUI that will handle all options and whenever a problem happens with an operation, simply exit the operation and open an error dialog.
- It is easier to dynamically change privileges on an operation because authorization is performed when the operation is attempted.
- It is easier to implement since you don't have to have multiple views on the data. This also improves maintainability.
- Users may get confused with a constant barrage of error dialogs. This hampers usability.
- Operation validation can be more difficult when users can perform any operation.
- Users will get frustrated when they see options that they cannot perform. This also decreases usability.

Known uses

Full View with Errors is used in Oracle databases. When you are using SQLPlus to access the data, you can execute any command. However, if you try to access data you don't have permission to see, an appropriate error message will be displayed.

Login windows inform users when they enter incorrect passwords.

Most word processors and text editors, including Microsoft Word and vi, let the user try to save over a read-only file. The program displays an error message after the save has been attempted and has failed.

Reuters SSL Developers Kit has a framework for reporting error, warning, and information events. It can be configured to report errors to standard error, a file, or a system dependent location such as a dialog.

Input Guard

Pattern documentation

Quick info

Intent: Protect components from input that does not conform to the system specification.

Problem

The Input Guard pattern applies to a system which has the following characteristics:

- The system is composed from distinguishable components, which can play the role of fault compartments and which interact with each other by feeding one's output into another's input.
- The errors that can be propagated into a system component have the form of erroneous input, i. e. input whose content or timing does not conform to the system specification.The second characteristic implies that internal errors (e. g. changes to the internal state due to electromagnetic disturbances in the environment where the system operates)are not considered by this pattern since they are not expressed as erroneous input according to the system specification. Moreover, this pattern does not deal with cases where the input to the system conforms with the system specification but it still contains errors according to the specification of the system's environment.

Forces

In the above context, the Input Guard pattern solves the problem of stopping the propagation of an error from the outside to the inside of the guarded component by balancing the following forces:

- Input that does not conform to the specification of the receiving component must be identified.
- Different systems have different requirements regarding size impact of the fault containment mechanism.
- Different systems have different requirements regarding the time penalty of the fault containment mechanism.
- Fault containment is usually integrated with other solutions provided for other fault tolerance constituents (e. g. error masking, error detection, fault diagnosis and the others mentioned in Section1)in order to provide wider fault tolerance guarantees.

Example

A system S consists of two components A and B. Component A takes as input two pairs of integers and provides as output a pair of integers representing the sum of the first and the second input pair. Component B takes as an argument a pair of integers and provides as output one integer representing the integer quotient of the division of the first input integer by the second one. The specification of component B imposes that both input values must be integers and the second input value cannot be zero. Component A receives its input form the user and feeds its output to component B, which in turn delivers its output to the user (see Figure 1).

Now, let's suppose that the system S is fed with with the following two pairs of integers (10,20)and (2,4). This input is legitimate according to the specification of system S and the expected output is5. The input pairs of integers are also legitimate input for the component A and the expected output is the pair (30,6). Figure1shows the case where an error has occurred inside component A and as a result the output (60,6)is delivered, which is erroneous according to the specification of component A. Notice however that the same pair is not erroneous input for the component B according to its specification (both60and6are integers and6does not equal zero). Hence, applying the Input Guard pattern on component B would not prevent the propagation of this error inside B. This kind of error can be contained inside the component that produced it (i. e. A)by applying theon component A.

Solution

To stop erroneous input from propagating the error inside a component a guard is placed at every access point of the component to check the validity of the input. Every input to the guarded component is checked by the guard against the component specification. If and only if the input conforms with that specification then it is forwarded to the guarded component.

Notice that the above solution does not define the behavior of the guard in the presence of erroneous input, besides the fact that it does not forward it to the guarded component. This is intentionally left undefined in order to allow implementations of the Input Guard to be combined with error detection mechanisms (e. g. when a check fails, an error notification is sent to the part of the system responsible for fault diagnosis)or with the implementations of error masking mechanisms (e. g. the comparator entity of the Active Replication pattern). Hence, the behavior of the guard when the checks performed on the input fail depends on the other fault tolerance constituents with which the input guard is combined.

Structure

Figure 1 (a)illustrates graphically the structure of the Input Guard pattern for a guarded component with a single access point. Figure 1 (b)contains the activity diagram that describes the functionality of the guard.

Dynamics

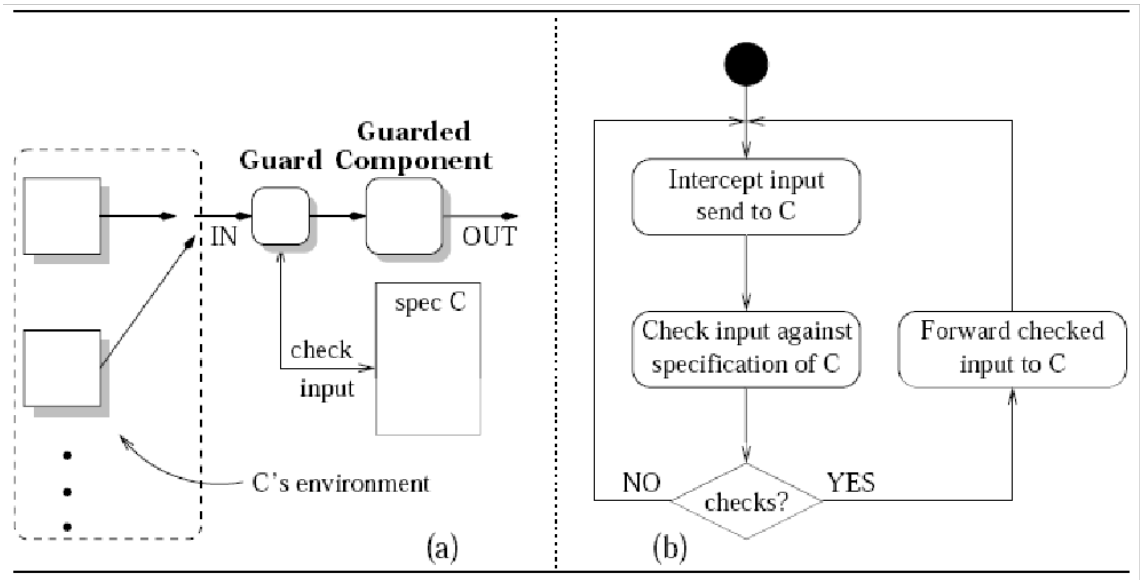


Figure 1: The structure [a]and the activity diagram [b]of the Input Guard pattern.

See Figure 1 .

Participants

The Input Guard pattern introduces two entities:

- The guarded component which is the part of the system that is protected against the fault contamination from external errors propagated to it through its input.
- The guard which is responsible to check for errors the input to the guarded component against its specification.There may be many instances of the guard entity for the same guarded component, depending on the system design and on the number of different access points the guarded component may have. For example, a software component with a number of interfaces and a number of operations declared in each interface may have one guard per interface or one guard per operation declared in its interfaces or any possible combination of those.

Collaborations

When the **Guarded Component** receives input from the environment, the **Guard** intercepts the input. The input is then checked against a specification of the interface of the **Guarded Component** , to see if the input is indeed valid input for the component. If the input is valid, it is forwarded to the **Guarded Component** . If not, the input is dropped.

Implementation

One possibility is to implement the guards as separate components in the system. This approach allows to have a number of guards proportional only to the number of the access points of the guarded component. The time overhead introduced by this approach is quite high since it includes the invocation of an additional component (i. e. the guard). Also, the space overhead of this approach is rather elevated since it increases the number of the components in a system by the number of guards that are implemented. Furthermore, in the case where components are mapped to individual units of failure (i. e. each component can fail as a whole and independently of other components)this approach introduces a well-known dilemma in fault tolerance: ``QUIS CUSTODIET IPOS CUSTODES?"(``who shall guard the guards?").

Despite the above inconveniences, this implementation approach is valuable in the case of COTS-based systems composed from black-box components where the system composer does not have access to the internals of the components. Also, this approach can be applied when fault containment comes as a late-or after-thought in the system development and a quick fix is needed in form of a patch. This implementation approach does not require any modification on existing components of a system; rather, guards are introduced as separate add-on components to the existing system.

Another implementation approach is to make the guard part of the implementation of the guarded component. This practice is often employed in programming where a method checks its arguments before using them to perform its designated task. This allows the coupling of the guard (s)and the guarded component. By integrating the guard with theguarded component the space overhead of the Input Guard implementation is kept low since it does not introduce another component in the system. Coupling the guard and guarded component implementation is usually applied in the development of COTS software where the developer has no knowledge about the rest of the system in which the component will be integrated. Hence, in order to assure

robust functioning of a component, the developer checks the input of the component on every call. The drawback of this implementation approach is the fact that the time overhead is high and fixed. This is because the guard is engaged on every call to the guarded component, even when the supplied input has already been checked by other fault tolerance means.

A third implementation possibility is to place the guard inside each of the components which may provide input to the guarded component. This approach allows the integration of the guard with other fault tolerance mechanisms (e. g. the guard of the Output Guard pattern for each component that provides input to the guarded component; see Section3for more details). Furthermore, this approach allows the elimination of redundant checks for errors which can increase the time and space overhead of fault tolerance solutions in a system. On the other hand, this approach is not applicable to COTS software. Third party developers may not have information about the specification of the other components to which they will feed their output, hence they do not know what conditions to check in the guard. A drawback of this implementation approach is the elevated space overhead; the number of guards is not only proportional to the access points of the guarded component but also to the number of components that provide input to the guarded component. Another drawback is that this guard cannot protect the guarded component from communication errors that occurred during the forward of the checked input from the guard to the guarded component. On the positive side however, this approach allows the guard to be selectively integrated only with those components that considered not robust enough and subject to produce erroneous input for the guarded component. This can be used to reduce the elevated space overhead of the approach.

Pitfalls

(Nothing given)

Consequences

The Input Guard pattern has the following benefits:

- It stops the contamination of the guarded component from erroneous input that does not conform to the specification of the guarded component.
- The undefined behavior of the guard in the presence of errors allows its combination with error detection and error masking patterns, and fault diagnosis mechanisms. Whenever this is applicable, the system benefits in terms of reduced run-time overhead introduced by the implementation of the fault tolerant mechanism (e. g. the combination of fault containment and error detection in the context of system recovery from errors).
- The similarities between the guard entities of the Input Guard pattern and Output Guard pattern (see Section3)allow the combination of the two in a single entity. This entity will operate on the same data and will perform two checks: one against the specification of the component that produced the data as output and the other against the specification of the component that will consume the data as input. When applicable, this combination can provide significant benefits in terms of time and space overhead since two separate checks will be performed by the same piece of code.
- There are various ways that the Input Guard pattern can be implemented, each providing different benefits with respect to the time or space overhead introduced by the guard. It is also possible to integrate the guard with an existing system without having to modify the internals of the system components (first implementation alternative). That reduces significantly the amount of system re-engineering required for applying the Input Guard pattern to COTS-based systems made of black-box components.The Input Guard pattern imposes also some liabilities:
- It is not possible to minimize both the time and the space overhead of this pattern. To keep low the time overhead introduced by the Input Guard pattern, the functionality of the guard must not be very time consuming. This results in a tendency to introduce a separate guard for each different access point (e. g. one guard per interface or even per operation declared in an interface)of the guarded component. Each such guard checks only a small part of the specification of the guarded component, minimizing thus the execution time of an individual guard. However, this results in a large number of guards, hence in an elevated space overhead. On the other hand, to keep low the space overhead introduced by the Input Guard pattern, the number of guards needs to remain as small as possible. This implies that each guard will have to check a lager number of input for the guarded component, becoming a potential bottleneck and thus penalizing the performance of the system with elevated time overhead.
- For certain systems that require guards to be implemented as components (e. g. systems composed from black-box COTS software), the Input Guard pattern results unavoidably to an elevated time and space overhead. The space overhead is due to the introduction of the new components implementing the guards. The time overhead is due to the fact that passing input to the guarded component requires one additional indirection through the component implementing the guard that check the given input.
- The Input Guard pattern cannot prevent the propagation of errors that do conform with the specification of the guarded component. Such errors may contaminate the state of the guarded component if it has one. Although these errors cannot cause a failure on the guarded component since it operates according to its specification, they can cause a failure on the rest of the system. Such a failure of the entire system will be traced back to an error detected in the contaminated guarded component. Unless the error detection and fault diagnosis capabilities of the system allow to continue tracing the error until the initial fault that caused it, it is possible that inappropriate recovery actions will be taken targeted only at the guarded component, which, nonetheless, has been operating correctly according to its specification.
- The Input Guard pattern can effectively protect a component from being contaminated by erroneous input according to its specification. However, unless it is combined with some error detection and system recovery mechanisms, this pattern will result in a receive-omission failure (i. e. failure to receive input)of the guarded component. For certain systems, such a failure of one of their components may cause a failure on the entire system. Hence, the Input Guard pattern has limited applicability to such systems if it is not combined with other fault tolerance patterns.

Known uses

Replicated System

Pattern documentation

Quick info

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.

Aliases: Redundant Components, Horizontal Scalability

Problem

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.

Forces

- 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.

Example

(Nothing given)

Solution

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.

Structure

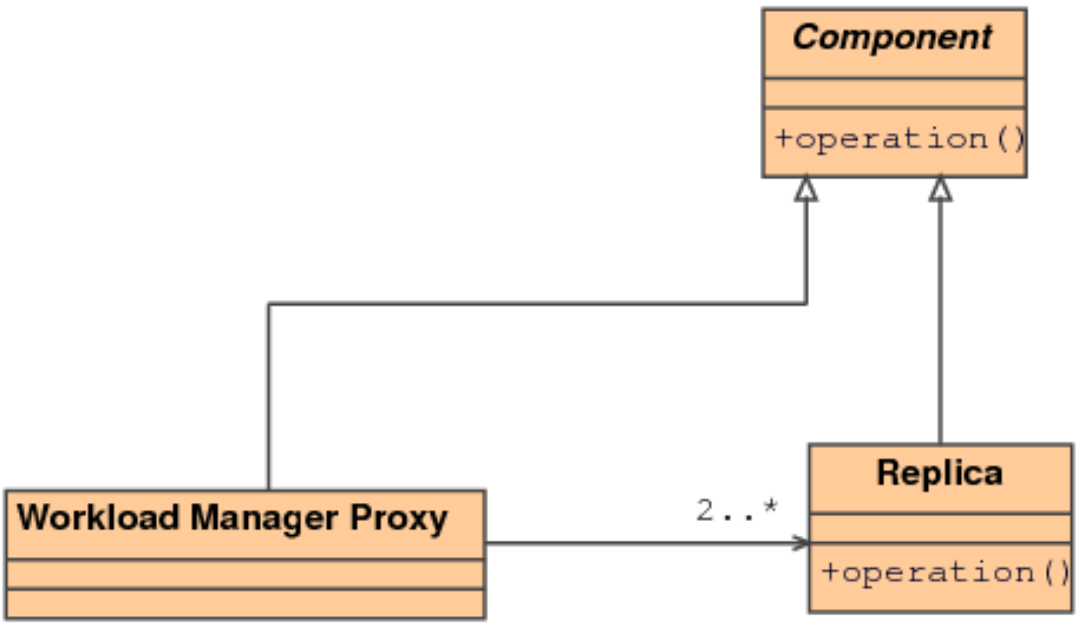


Figure 1: Structure of the Replicated System.

See Figure 1 .

Dynamics

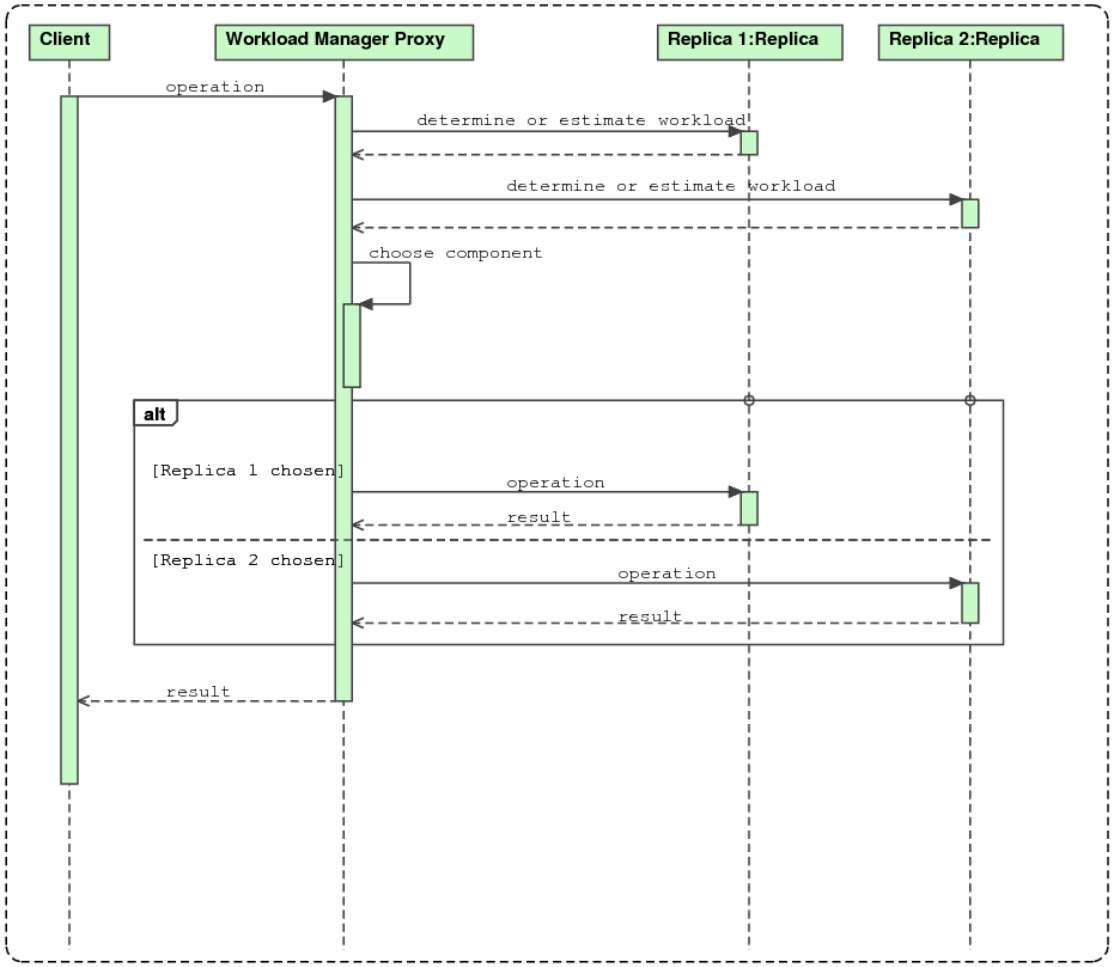


Figure 2: Event sequence of the Replicated System.

See Figure 2 .

Participants

The relations between the participants are shown in Figure 1 . The participants and their responsabilities are:

- 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

The interactions between the participants are shown in Figure 2 .

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

Implementation

(Nothing given)

Pitfalls

(Nothing given)

Consequences

- 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.
- Making sure that the replica's are real replica's (i. e., they would respond identically to each request at all times)can be hard.

Known uses

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

Reverse Proxy

Pattern documentation

Quick info

Intent: Protect your web server infrastructure on an application protocol level, without hindering accessibility.

Problem

Putting a web server or an application server directly on the Internet gives attackers direct access to any vulnerabilities of the underlying platform (application, web server, libraries, operating system). However, to provide a useful service to Internet users, access to your server is required. Firewalls shield your server from attacks on the network level, but a Security Reverse Proxy protects also on an application protocol level.

Forces

- A simple firewall is not enough to protect your web server, since some access must be provided to the Internet.
- Attack scenarios often employ extra long, or extra crafted request parameters to exploit buffer overflows. Most firewalls work on the network packet level and cannot prohibit attacks using such invalid requests.
- Installing patches to your web server platform helps to avoid exploitation of known vulnerabilities. But with each patch you risk that your system extensions cease to work. You need to rerun your integration tests at each patch level and might need to keep your extensions up to date with each patch level. It might even be impossible to upgrade your web server in a timely manner, because the extensions aren't ready.
- Switching to another web server software by a different source is expensive, risky and time consuming, too. A new web server might have fewer vulnerabilities, but you are less familiar with it. In addition it might also require to adapt your own system extensions.
- You cannot know about vulnerabilities detected in the future.

Example

You are running your web site using a major software vendorÂ¿s web server software. Your web site uses this vendor's proprietary extensions to implement dynamic content for your visitors and you have invested heavily in your website's software. Your server is protected by a regular firewall.

You must open this firewall to allow access to the public port (80)of your web server. Attacks from the Internet exploiting vulnerabilities of your server software burden your system administrator with installing patches frequently. Switching to another vendor's web server is not possible because of the existing investment in the web server platform, its content and your own software extensions. In addition, with every new patch you install, you run the risk of destabilizing your configuration so that your system extensions cease to work, that your software extensions cease to work. How can you escape the dilemma to keeping your web site up without compromising its security and integrity?

Solution

Change your network topology to use a protection reverse proxy that shields your real web server. Configure this reverse proxy to filter all requests, so that only (mostly)harmless requests will reach the real web server. Two firewalls ensure that no external network traffic reaches the real web server. The resulting network topology provides a demilitarized zone (DMZ)containing only the reverse proxy machine and a secured server zone containing the web server.

Structure

The structure of a reverse proxy is depicted in Figure 1 .

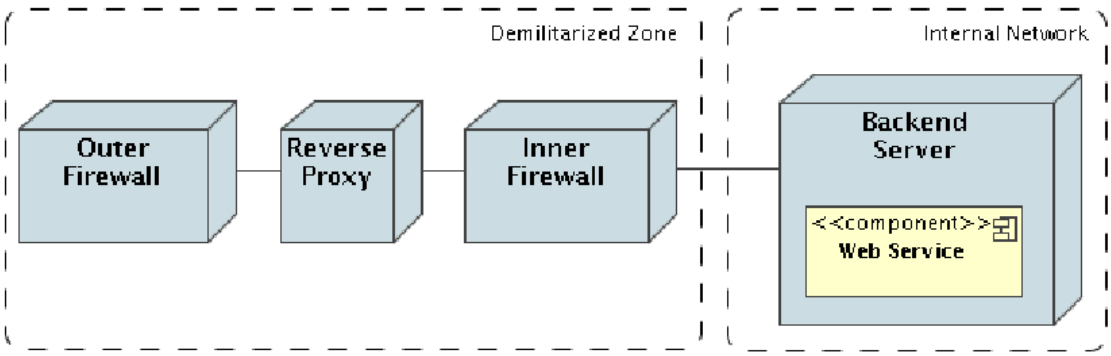


Figure 1: Reverse proxy structure.

Dynamics

The dynamics of a Reverse Proxy are included in Figure 2 .

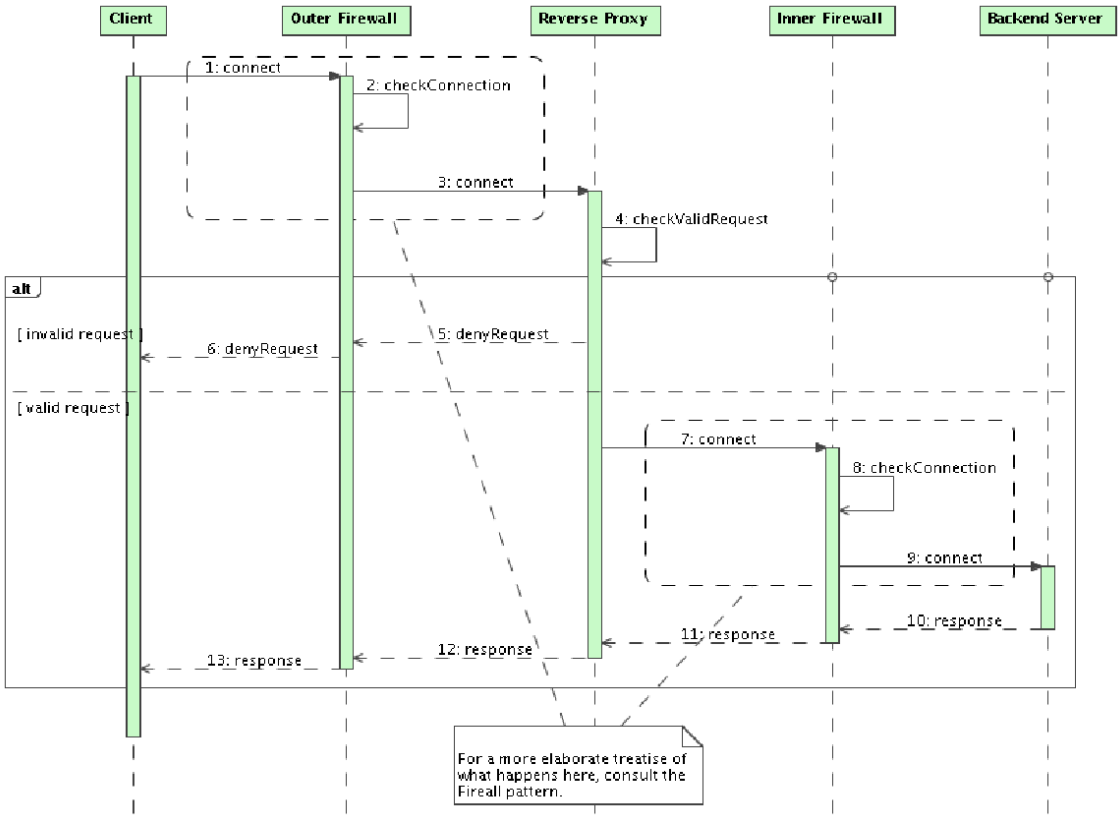


Figure 2: Reverse proxy dynamics.

Participants

- The **outer firewall** filters incoming network traffic and allows only HTTP port access to the Reverse Proxy. Furthermore, it denies outbound connection from the Reverse Proxy to the Internet.
- The **reverse proxy** accepts requests from browsers and forwards only valid requests to the backend server. Furthermore, it passes the reply from the backend server back to the originating browser.
- The **inner firewall** separates the server zone from the DMZ. It denies inbound connections except from the Reverse Proxy and denies outbound connection from the backend servers.
- The **backend server** provides the real web service. It accepts requests from the reverse proxy and returns replies.

Collaborations

- A client connects to the reverse proxy. This connection is intercepted by the outer firewall, and checked against the firewall configuration to see if it is allowed to be established. We refer to thepattern for additional info.
- If the outer firewall allows the connection, the reverse proxy then checks if the connection contains a valid request for the backend server. If not, the request of the client is denied. Otherwise, the reverse proxy forwards the request to the backend server.
- The inner firewall intercepts and checks the connection from the reverse proxy to the backend server, and ensures that the connection originates from the proxy (i. e., no other external parties are allowed to communicate with the backend server directly). Again, we refer to thepattern.
- The backend server receives the request of the client, acts accordingly, and returns a response.

Implementation

To implement the Security Reverse Proxy several tasks need to be done:

- Plan your firewall and network configuration. Even if the firewall update is done after every other part is in place, it is good to start with a plan, so that configuration of the other components can rely on the firewall plan. Often the concrete configuration needs to consider more than just one protocol and some explicit ``holes"in your firewall may be needed. Find out what protocol your reverse proxy solution needs to support. Typically only HTTP (port80)is needed, but you might want to allow other protocols as well through your reverse proxy.
- Select a Reverse Proxy platform. You might create your own reverse proxy, for example by configuring the Apache web server with mod_rewrite and mod_proxy modules, several vendors offer professional reverse proxy solutions, or you

might be brave and implement your own reverse proxy. Showing the details of implementing your own reverse proxy server software is beyond the scope of this pattern. If your reverse proxy needs to support more specialized or seldom used protocols, building one yourself might be the only option. When selecting a vendor or source for your security reverse proxy you should opt for a simple and proven solution. For example, using Apache you risk all Apache web server vulnerabilities to be present in your security reverse proxy. On the other hand, the Apache web server is deployed so often, that most vulnerabilities and countermeasures are known.

- Configure your backend web server (s). The web content should rely on relative path names and not use its internal name or IP address to refer to itself. Otherwise, links might not work, because the browser can no longer directly access the machine it is running on.
- Configure your Security Reverse Proxy. For the security to work, you need to define what requests should be mapped towards your backend web server, and define what to happen if invalid requests occur. For example, you might want to log what request were denied by the reverse proxy. For request filtering there exists two approaches: black lists and white lists.
    - A black list filter only blocks requests that its list of malicious requests knows of, but passes on all others. Black list filters are easier to deploy but riskier. They are often used by ``higher-level''firewalls.
    - A white list filter is more restrictive and only lists allowed requests. It needs to be configured with detailed knowledge of the backend server and allowed URLs. A white list filter needs to be adapted every time your backend server changes significantly in its URL space. Nevertheless, it is the better choice for a Security Reverse Proxy.If your backend server relies on redirects or other mechanisms using its host address and you cannot change that, you need to configure your reverse proxy to modify server responses accordingly.
- Deploy everything. Initial deployment with setting up firewalls, network and routers, host IP addresses and so on requires good planning. If you have something up and running already, this reconfiguration might mean some service interruption. Nevertheless, later changes to the topology need only consider the reverse proxy and eventually the inner firewall.

Pitfalls

(Nothing given)

Consequences

The pattern implies the following benefits:

- Your backend server is protected. Attackers can no longer directly exploit vulnerabilities of the backend server. Even when the backend server gets compromised, the firewalls hinder further spreading of Internet worms, etc., by blocking outgoing requests from the backend server.
- Even with known vulnerabilities, you might be able to keep your web server configuration stable, because the Security Reverse Proxy with its request filtering can prohibit exploitation of the web server's vulnerabilities.
- Easier patch administration. Only one machine remains connected to the Internet directly and needs to be monitored for potential vulnerabilities and existing patches to be applied. However, you cannot blindly trust your Security Reverse Proxy. A backend server still needs to be configured with your brain on, to avoid exploitation of vulnerabilities with ``allowed''requests.However, the Security Reverse Proxy pattern also has its liabilities:
- Black list filtering can give you a false sense of security. Like patches, black lists can only be constructed after a vulnerability is known.
- White list filtering can be fragile, when backend servers change. Adding functionality, or re-arranging content structure on the backend web server, can imply additional work to re-configure the white list filter of the Security Reverse Proxy.
- Latency. A reverse proxy adds latency to the communication, not only because of the additional network traffic, but also for the filtering and validation of requests.
- Some loss of transparency. Some restrictions are imposed on the backend servers. However, these are typically good practice anyway, like relative paths in URLs. Nevertheless, the backend servers no longer see the communication end partner directly on the network level. So the protocol may need to provide a means to identify the original communication end point (which HTTP allows).
- Additional point of failure. If the reverse proxy stops working, also any access to your web site is impossible. Any additional component that can fail increases the overall risk of system failure. To reduce this risk, you can provide a hot or cold stand by installation with hardware or software fail-over switches.

Known uses

Security Reverse Proxies are popular. Some organizations in the financial industry have the guideline to use a reverse proxy for every protocol provided over the Internet (with some exceptions, like DNS). Thus they can ensure that never a vulnerable server is directly accessible from the ``wild''.

You can use stunnel to provide a secure reverse proxy for some protocol. However, this only ensures a one on one connection, not like a ``normal''reverse proxy provides.

Secure Access Layer

Pattern documentation

Quick info

Intent: Application security will be insecure if it is not properly integrated with the security of the external systems it uses. On top of the lower-level security, build a secure access layer for communicating in and out of the program.

Aliases: Using Low-level security, Using Non-application security, Only as strong as the weakest link

Problem

When secure documents are transferred from one secure area to another in the military base, it is important that the security of the documents is not violated during the transfer. If the document is being transferred via a computer disk, the data could be encrypted and then locked in a briefcase and handcuffed to the arm of the courier during transfer. This will provide an isolation layer to protect the secure information during the transfer.

Most applications tend to be integrated with many other systems. The places where system integration occurs can be the weakest security points and the most susceptible to break-ins. If the developer is forced to put checks into the application wherever the applications communicates with these systems, then the code will be very convoluted and abstraction will be difficult. An application that is built on an insecure foundation will be insecure. In other words, it doesn't do any good to bar your windows when you leave your back door is wide open.

Forces

- Application development should not have to be developed with operating system, networking, and database specifics in mind. These can change over the life of an application.
- Putting low-level security code throughout the whole application makes it difficult to debug, modify, and port to other systems.
- Even if the application is secure, a good hacker could find a way to intercept messages or go under the hood to access sensitive data.
- Interfacing with external security systems is sometimes difficult.
- An external system may not have sufficient security, and implementing the needed security may not be possible or feasible.

Example

The PLoP registration program uses a Secure Access Layer. A layer was created where all communications is processed for registering through the web. This communications layer is positioned on top of Apache's Secure Socket Layer. This prevents any information from being sniffed during the entry of data such as credit card numbers. Also, a layer on the database side was also created to provide additional security by encrypting the credit card information in the database. The secure layer uses a key for encrypting and decrypting the data when needed. Thus, even if someone was able to access the database through some back door, the credit card data is still protected.

Solution

Build your application security around existing operating system, networking, and database security mechanisms. If they do not exist, then build your own lower-level security mechanism. On top of the lower-level security, build a secure access layer for communicating in and out of the program.

Usually an application communicates with many pre-existing systems. For example, a financial application on a Windows NT client might use an Oracle database on a remote server. Given that most systems already provide a security interface, develop a layer in your application that encapsulates the interfaces for securely accessing these external systems. All communication between the application and the outside world will be routed through this secure layer.

The important point to this pattern is to build a layer to isolate the developer from change. This layer may have many different protocols depending upon the types of communications that need to be done. For example, this layer might have a protocol for accessing secure data in an Oracle database and another protocol for communicating securely with Netscape server through the Secure Sockets Layer (SSL). The crux of this pattern is to componentize each of these external protocols so they can be more easily secured. The architecture for different Secure Access Layers could vary greatly. However, the components'organization and integration is beyond the scope of this pattern.

By creating a Secure Access Layer with a standard set of protocols for communicating with the outside world, an application developer can localize these external interfaces and focus primarily on applications development. Communicate in and out of

the application will pass through the protocols provided by this layer.

This pattern assumes a convenient abstraction is possible. For example, VisualWorks'LensSession does not support Microsoft Access, so QueryDataManager cannot be used with a Microsoft Access database. Secure Access Layer, however, provides a location for a more general database abstraction. Third party drivers have been developed for ODBC that can communicate with Microsoft Access. By using the Secure Access Layer, it is easy to extend your application to use the ODBC protocol, thus allowing your application to communicate with any database that supports ODBC.

Structure

The structure of a secure access layer is depicted in Figure 1

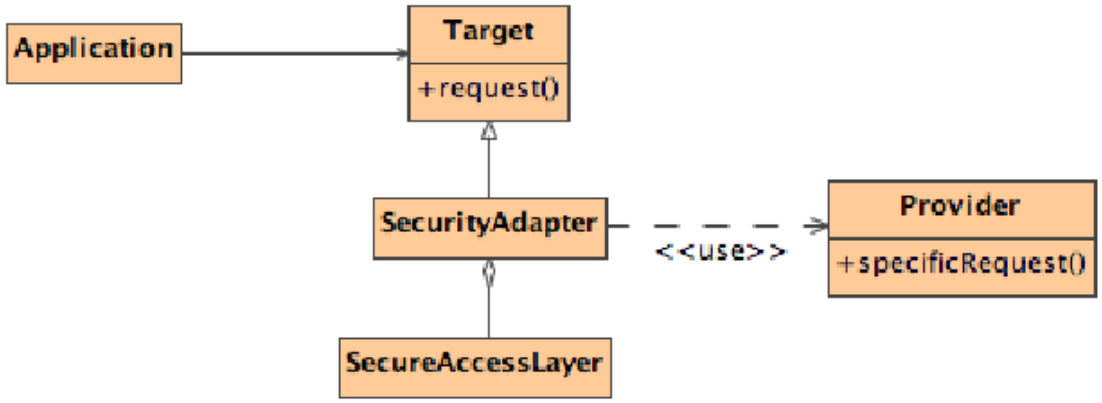


Figure 1: Secure access layer class diagram.

Dynamics

The dynamics of a secure access layer are depicted in Figure 2

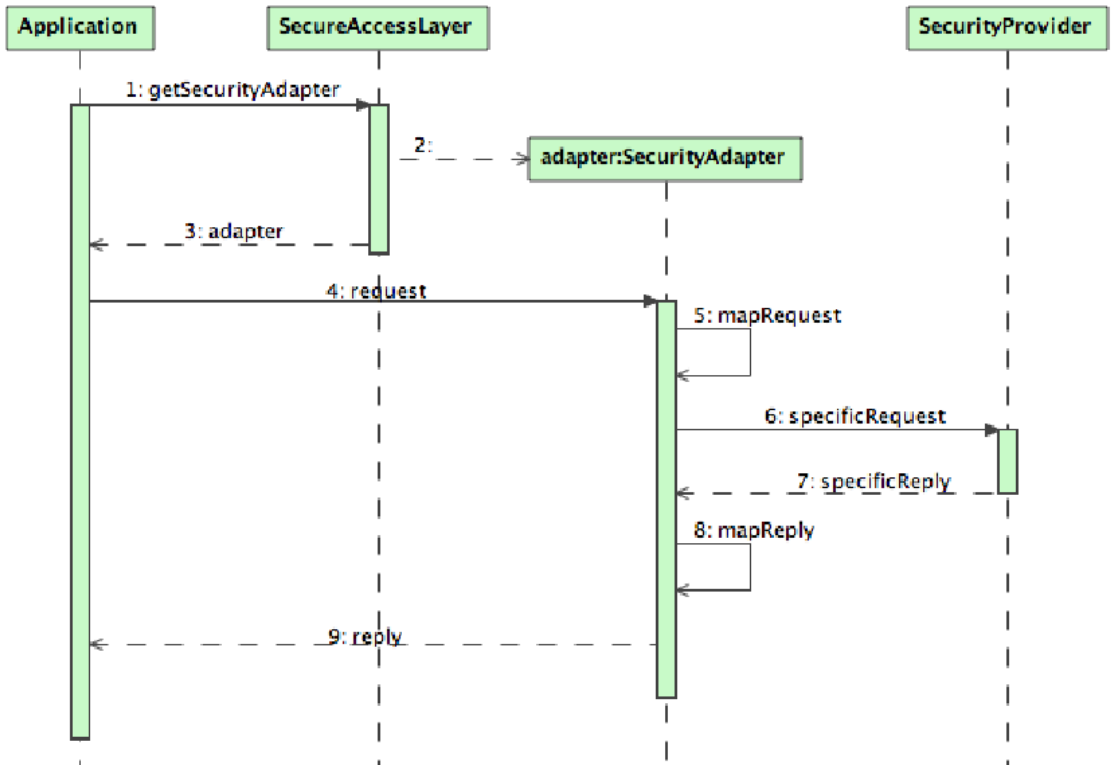


Figure 2: Secure access layer sequence diagram.

Participants

- The **Application** is wrapped by the secure access layer. It depends on functionality offered by a certain provider.
- The **Target** offers an interface to the application which is a general abstraction of the specific interface as offered by the provider.
- Theextends the target by wrapping invocations to it in a secure way, and mapping them to specific requests to a provider.
- The **SecureAccessLayer** is an aggregation of these SecurityAdapters, and contains functionality to find and instantiate the appropriate adapter.

Collaborations

When the **Application** wants certain functionality (for example, communicating with an Oracle database), it locates the appropriate **SecurityAdapter** by issuing a request to the **SecureAccessLayer** . The **SecureAccessLayer** will ensure that an instantiation of the appropriate adapter is available, and return a reference to the **Application** .

The **Application** then issues requests to this adapter, which will map these general requests to provider-specific requests. It will also ensure that these requests are wrapped in a secure way and that the interaction with the provider happens securely (i. e., the correct interaction protocols are respected).

Once the provider has issued a reply, the provider-specific reply is in turn mapped to a general format. This general reply is then returned to the **Application** .

Implementation

(Nothing given)

Pitfalls

(Nothing given)

Consequences

- **+** A Secure Access Layer can help isolate where an application communicates with external security systems. Isolating secure access points make it easier to integrate new security components and upgrade existing ones, improving maintainability.
- **+** A Secure Access Layer can make an application more portable. If the application later needs to communicate with Sybase rather than Oracle, then the access to the database is localized and only needs to be changed in one place. QueryObjects uses this approach by having all accesses to the database go through the QueryDataManager, which is built on top of the LensSession. The LensSession can map to either Oracle or Sybase. Therefore the application developer does not need to be concerned with either choice or future changes.
- **-** Different systems that your application may need to integrate with use different security protocols and schemes for accessing them. This can make it difficult to develop a Secure Access Layer that works for all integrated systems, and it also may cause the developer to keep track of information that many systems do not need.
- **-** It can be very hard to retrofit a Secure Access Layer into an application which already has security access code spread throughout.

Known uses

- Secure Shell includes secure protocols for communicating in X11sessions and can use RSA encryption through TCP/IP connections.
- SSL (Netscape Server)provides a Secure Access Layer that web clients can use for insuring secure communication.
- Oracle provides its own Secure Access Layer that applications can use for communicating with it.
- CORBA Security Services specifies how to authenticate, administer, audit and maintain security throughout a CORBA distributed object system. Any CORBA application's Secure Access Layer would communicate with CORBA's Security Service.

Secure Logger

Pattern documentation

Quick info

Intent: Application events must be logged in a centralized way, and it should be impossible to alter log files.

Problem

All application events and related data must be securely logged for debugging and forensic purposes. This can lead to redundant code and complex logic.

All trustworthy applications require a secure and reliable logging capability. This logging capability may be needed for forensic purposes and must be secured against stealing or manipulation by an attacker. Logging must be centralized to avoid redundant code throughout the code base. All events must be logged appropriately at multiple points during the application's operational life cycle. In some cases, the data that needs to be logged may be sensitive and should not be viewable by unauthorized users. It becomes a critical requirement to protect the logging data from unauthorized users so that the data is not accessible or modifiable by a malicious user who tries to identify the information trail. Without centralized control, sometimes the code usually gets replicated, and it becomes difficult to maintain the changes and monitor the functionality.

One of the common elements of a successful intrusion is the ability to cover one's tracks. Usually, this means erasing any tell-tale events in various log files. Without a log trail, an administrator has no evidence of the intruder's activities and therefore no way to track the intruder. To prevent an attacker from breaking in again and again, administrators must take precautions to ensure that log files cannot be altered. Cryptographic algorithms can be adopted to ensure data confidentiality and the integrity of the logged data. But the application processing logic required to apply encryption and signatures to the logged data can be complex and cumbersome, further justifying the need to centralize the logger functionality.

Forces

- You need to log sensitive information that should not be accessible to unauthorized users.
- You need to ensure the integrity of the data logged to determine if it was tampered with by an intruder.
- You want to capture output at one level for normal operations and at other levels for greater debugging in the event of a failure or an attack.
- You want to centralize control of logging in the system for management purposes.
- You want to apply cryptographic mechanisms for ensuring confidentiality and integrity of the logged data.

Example

(Nothing given)

Solution

Use a Secure Logger to log messages in a secure manner so that they cannot be easily altered or deleted and so that events cannot be lost. The Secure Logger provides centralized control of logging functionality that can be used in various places throughout the application request and response. Centralizing control provides a means of decoupling the implementation details of the logger from the code of developers who will use it throughout the application. The processing of the events can be modified without impacting existing code. For instance, developers can make a single method call in their Java code or JSP code. The Secure Logger takes care of how the events are securely logged in a reliable manner.

Structure

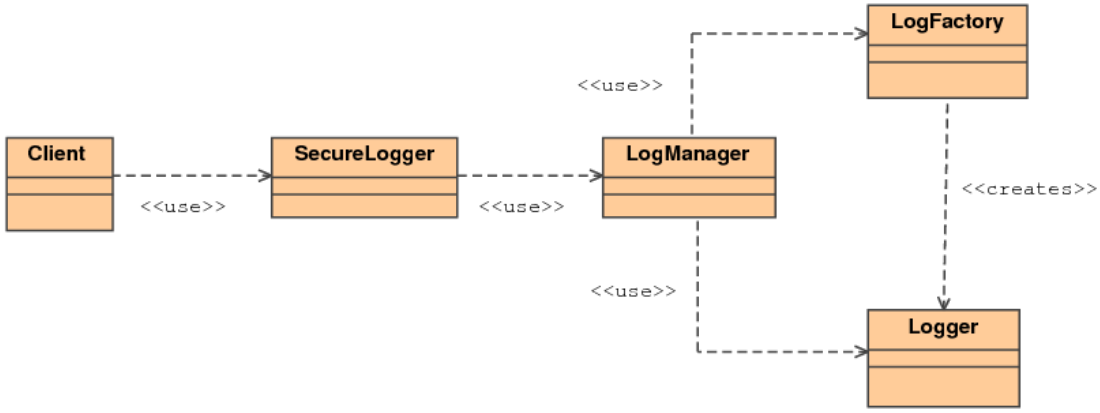


Figure 1: Class layout of the Secure Logger

See Figure 1 .

Dynamics

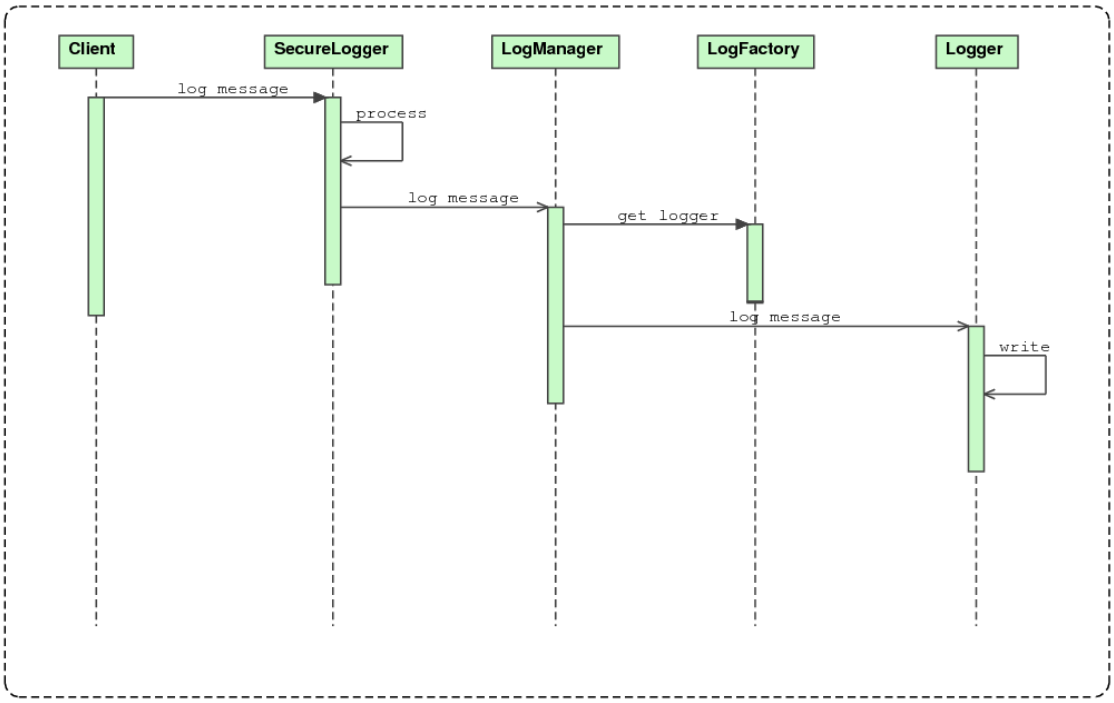


Figure 2: Event sequence for the Secure Logger

See Figure 2 .

Participants

- Client A client sends a request to a particular target resource.
- SecureLogger SecureLogger is a class used to manage logging of data in a secure, centralized manner.
- LogManager LogManager obtains a Logger instance from LogFactory and uses it to log messages.
- LogFactory A LogFactory is responsible for creating and returning Logger instances.
- Logger A Logger writes log messages to a target destination.

Collaborations

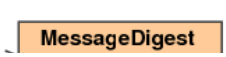
A client uses the SecureLogger to log events. The SecureLogger centralizes logging management and encapsulates the security mechanisms necessary for preventing unauthorized log alteration.

- Client wants to log an event using SecureLogger.
- SecureLogger generates a sequence number and prepends it to the message.
- SecureLogger passes the LogManager the modified event string to log.
- LogManager obtains a handle to a Logger instance from a LogFactory.
- LogFactory creates a Logger instance.
- LogManager delegates actual logging of the event to the Logger.There are two parts to this logging process. The first part involves securing the data to be logged and the second part involves logging the secured data. The SecureLogger class takes care of securing the data and the LogManager class takes care of logging it.

Implementation

There are two basic strategies for implementing a Secure Logger. One strategy is to secure the log itself from being tampered with, so that all data written to it is guaranteed to be correct and complete. This strategy is the Secure Log Store Strategy. The other strategy, the Secure Data Logger Strategy, secures the data so that any alteration or deletion of it can be detected. This works well in situations where you cannot guarantee the security of the log itself.

Secure Data Logger Strategy The Secure Data Logger Strategy entails preprocessing of the data prior to logging it. After the data is secured in the preprocessing, it is sent to the logger in the usual manner. There are four new classes introduced to help secure the data. Figure 3 illustrates the structure of the Secure Logger implemented using a Secure Data Logger Strategy.



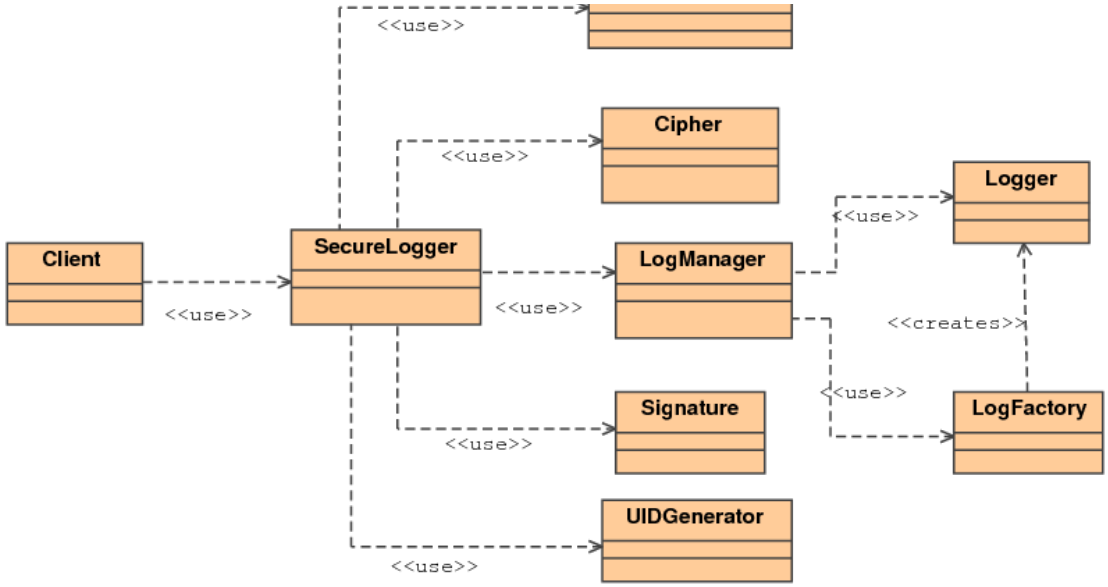


Figure 3: Secure Logger with Secure Data Logger Strategy class diagram

We use the MessageDigest, Cipher, Signature, and UIDGenerator classes for applying cryptographic mechanisms and performing various functions necessary to guarantee the data logged is confidential and tamperproof. Figure 4 shows the sequence of events used to secure the data prior to being logged.

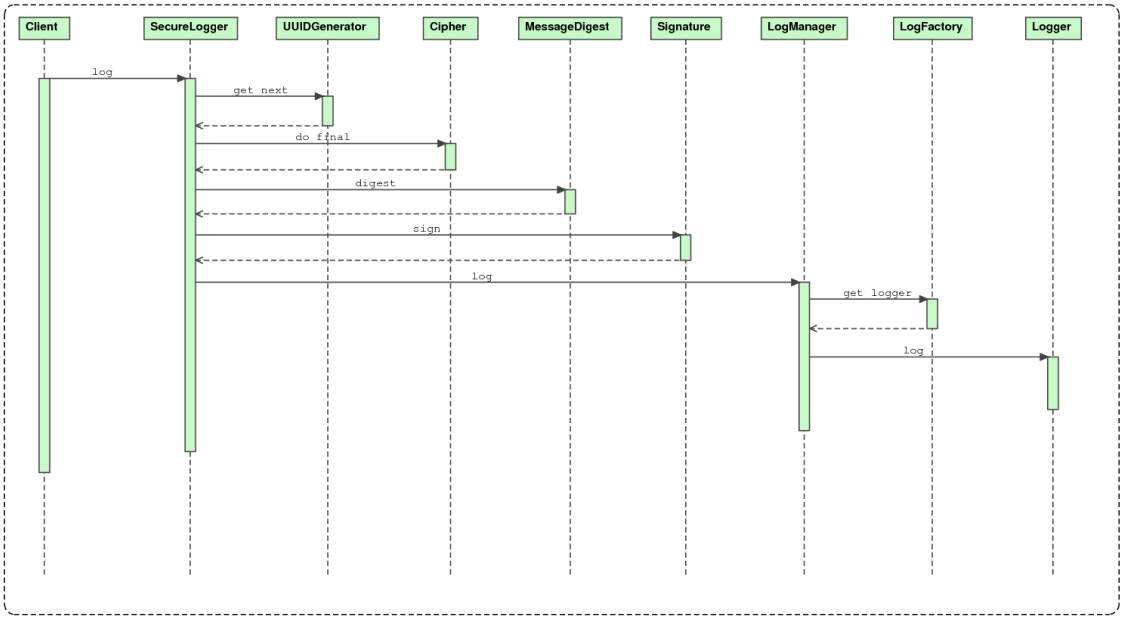


Figure 4: Secure Logger with Secure Data Logger Strategy sequence diagram

When you have sensitive data or fear that log entries might be tampered with and can't rely on the security of the infrastructure to adequately protect those entries, it becomes necessary to secure the data itself prior to being logged. That way, even if the log destination (file, database, or message queue)is compromised, the data remains secure and any corruption of the log will become clearly evident.

There are three elements to securing the data:

- Protect sensitive data. Ensure all sensitive data are stored and remain confidential throughout the process. For example, Credit card numbers should not be viewed directly by unauthorized personnel.
- Prevent data alteration. Make sure that data is tamperproof. For example, user IDs, transaction amounts, and so forth should not be changed.
- Detect deletion of data. Detect if events have been deleted from the log, a tell-tale sign that an attacker has compromised the system.To protect sensitive data, encrypt it using a symmetric key algorithm. Public-key algorithms are too CPU-intensive to use for bulk data. They are better for encrypting and protecting a symmetric key for use with a symmetric key algorithm. Properly protecting the symmetric key can ensure that attackers cannot access sensitive data even if they have access to the logs. For this, the SecureLogger can use an EncryptionHelper class. This class is responsible for encrypting a given string but not for decrypting it. This is an extra security precaution to make it harder for attackers to gain access to that sensitive data. Decryption should only be done outside the application, using an external utility that is not accessible

from the application and its residing host. Data alteration can be prevented by using digitally signed message digests in the same manner that e-mail is signed. A message digest is generated for each message in the log file and then signed. The signature prevents an attacker from modifying the message and creating a subsequent message digest for the altered data. For this operation, the SecureLogger uses MessageDigestHelper and DigitalSignatureHelper classes. Finally, to detect deletion of data, a sequence number must be used. Using message digests and digital signatures is of no use if the entire log entry, including the signed message, is deleted. To prevent deletion, each entry must contain a sequence number that is part of the data that gets signed. That way, it will be evident if an entry is missing, since there will be a gap in the sequence numbers. Because the sequence numbers are signed, an attacker would be unable to alter subsequent numbers in the sequence, making it easy for an administrator reviewing the logs to detect deletions. To accomplish this, the SecureLogger uses a UUIDpattern.

Secure Log Store Strategy In the Secure Log Store Strategy, the log itself is secured from tampering. A secure repository houses the log data and can be implemented using a variety of off-the-shelf products or various techniques such as a Secure Pipe. A Secure Pipe pattern is used to guarantee that the data is not tampered with in transit to the Secure Store. Figure 5 illustrates the structure of the Secure Logger pattern implemented using a Secure Log Store Strategy.

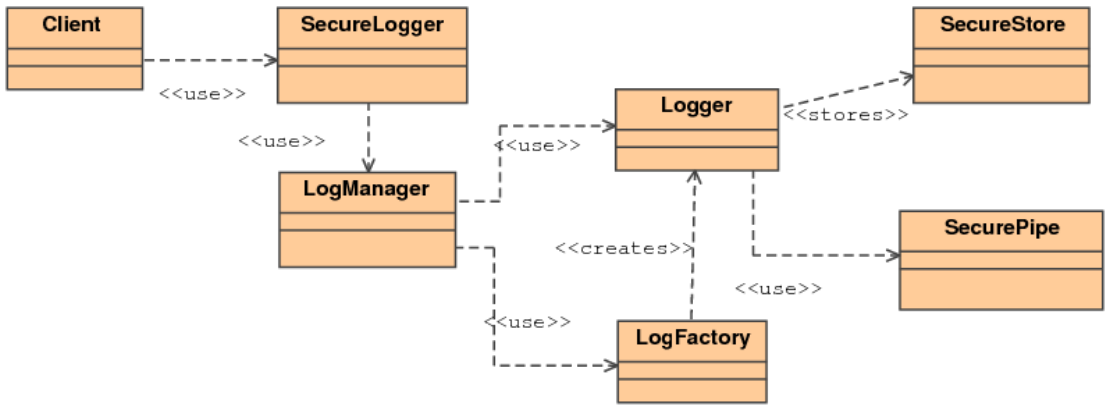


Figure 5: Secure Logger Pattern with Secure Log Store Strategy class diagram

The Secure Log Store strategy does not require the data processing that the Secure Data Logger Strategy introduced. Instead, it makes use of a Secure Pipe pattern and a secure datastore (such as a database), represented as the SecureStore object in Figure 5 . In Figure 6 , the only change from the main Secure Logger pattern sequence is the introduction of the Secure Pipe pattern.

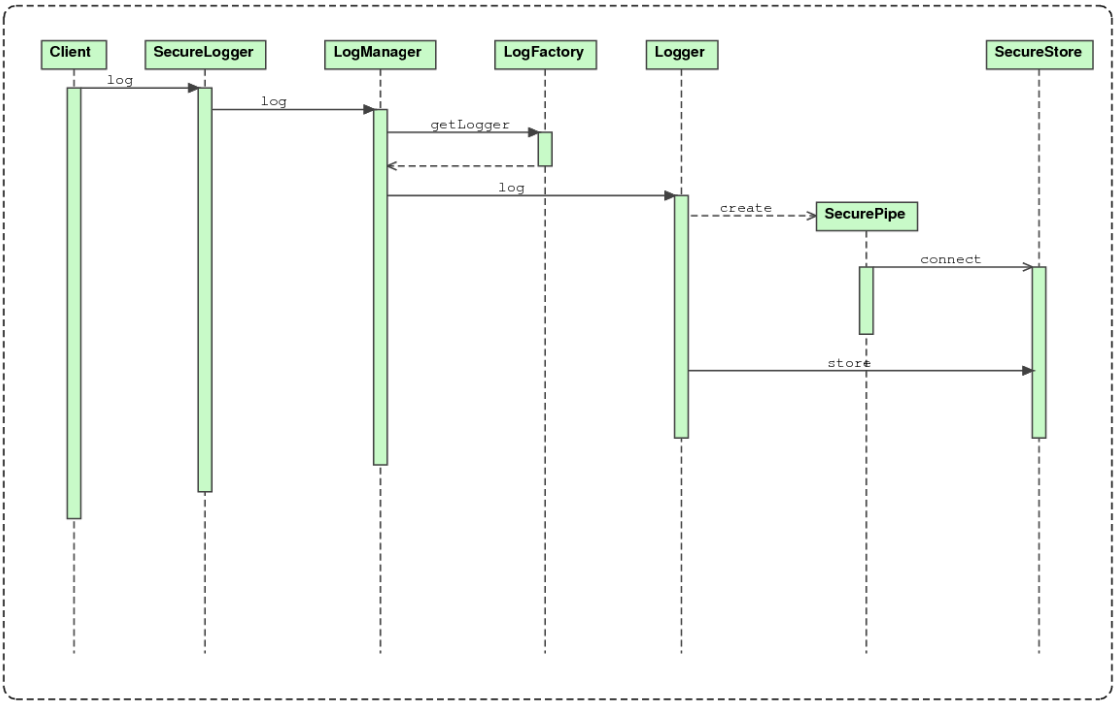


Figure 6: Secure Logger pattern using Secure Pipe

In the Secure Log Store Strategy sequence diagram, depicted in Figure 6 , Logger establishes a secure connection to the SecureStore using a SecurePipe. The Logger then logs messages normally. The SecureStore is responsible for preventing tampering with the log file. It could be implemented as a database with create-only permissions for the Logger user; a listener on

a separate, secure box with write only capabilities; or any other solution that prevents deletion, modification, or unauthorized creation of log entries.

Pitfalls

The Secure Logger pattern provides the entry point for logging in the application. As such, it has the following security factors and risks associated with it:

- Key Management. The Secure Logger must either encrypt data itself or establish a secure channel to a secure log store. Either way, there are key management issues that must be addressed. If the key or password for retrieving the key (such as for a keystore)must be kept in code, make sure that the code is obfuscated. Failure to properly protect the key will render the Secure Logger useless.
- Integrity. The Secure Logger must provide integrity when communicating with the secure store in the Secure Data Store strategy. If the communication channel is not secure, it opens up the possibility that an attacker can compromise the communication channel and modify the data in transit.Should everything be logged from Web tier?No. The Secure Logger pattern is applicable across tiers. It should be implemented on each tier that requires logging.

Too much performance overhead. Using the Secure Data Store Strategy incurs severe performance overhead. Expect a significant slowdown due to the extensive use of cryptographic algorithms. The Secure Data Logger Strategy is the preferred strategy for performance, but it also incurs the same overhead associated with use of Secure Pipe.

How likely is log tampering?Log modifications to cover an attacker's tracks is not only common, it is the hallmark of a good hacker. It is difficult to determine how prevalent it is due to its very nature. Log files that have been successfully altered usually mean that the last trace of evidence that a system has been compromised is now gone.

Shouldn't log security be the responsibility of the system administrators?In many cases, system administrators can effectively secure the log, and additional security is unnecessary. It depends on the skill of your operations staff along with the requirements of the application. Like any other security, log security is only as strong as the weakest link. By consolidating and encapsulating log functionality using the Secure Logger, you provide the capability to add additional security, such as in the Secure Data Strategy, if and when you find external mechanisms are not sufficient.

Consequences

Using the Secure Logger pattern helps in logging all data-related application events, user requests, and responses. It facilitates confidentiality and integrity of log files. In addition, it provides the following benefits:

- Centralizes logging control. The Secure Logger improves reusability and maintainability by centralizing logging control and decoupling the implementation details from the API. This allows developers to use the logging facilities through the API independent of the security functionality built into the logger itself. This reduces the possibility that business developers will inadvertently circumvent security by misusing it.
- Prevents undetected log alteration. The key to successfully compromising a system or application is the ability to cover your tracks. This involves alteration of log files to ensure that an administrator cannot detect that a breach has occurred. By employing a Secure Logger, security developers can prevent log alterations, ensuring that a breach can be detected through log file forensics, which is the first step in tracking down an intruder and preventing security breaches.
- Reduces performance. The Secure Logger impacts performance due to the use of cryptographic algorithms. Operations such as message digests, digital signatures, and encryption are computationally expensive and add additional performance overhead. Use only the necessary functionality to avoid unwanted performance overhead. Reduced performance can lead to a self-inflicted denial of service attack.
- Promotes extensibility. Security is a constantly evolving process. To protect against both current and future threats, code must be adaptable and extensible. The Secure Logger provides the requisite extensibility by hiding implementation details behind a generic interface. By increasing the overall lifespan of the code, you increase its reliability by having tested it and worked out all of its bugs.
- Improves manageability. Since all of the logging control is centralized, it is easier to manage and monitor. The Secure Logger performs all of the necessary security processing prior to the actual logging of the data, which allows management of each function independently of the others without risk of impacting overall security.

Known uses

Secure Message Router

Pattern documentation

Quick info

Intent: Securely communicate with multiple partner endpoints using message-level security and identity-federation mechanisms.

Problem

Using Web services communication in an organizational workflow or across the Internet with multiple partners poses a lot of challenges. If the message sender signs and encrypts the message in its entirety, the message sender restricts the possibility of further message changes by the message recipient in the workflow. This becomes a critical issue when each recipient of the message in a workflow has a responsibility for a selected portion of that message and must modify or add to it. If the message-level security, such as signature and encryption, were applied to the entire message, any modification made by the initial recipient would invalidate the original message as well as expose the entire message, which was not intended for the initial recipient. In some cases, if a message is intended for multiple recipients and only selected fragments need to be revealed for each recipient, then it becomes more complex to convert each fragment as a message and then compile them together at the end of workflow. Let's consider an example scenario; a patient visits a hospital that handles all documents and communication electronically using XML Web services. All patient information (for example, contact information, insurance data, health analysis, lab results, doctor observations, prescriptions, visit schedule, credit card information, and so on)is represented in XML. During the visit, the patient's health record is maintained via a workflow involving doctors, pharmacists, insurance providers, and so on. Each individual participating in the workflow does not require complete access to the patient's record. Only selected portions of the message are required and applicable to each workflow participant. For example, the billing department only requires knowing the insurance provider and the co-payment and credit card information; it does not need to know the patient's health history. Although the information workflow happens within an organization, it is a violation of specific legal regulations to unnecessarily disclose information to personnel. Web services promise easier integration of applications, business partners, and consumers. With multiple parties involved, it often becomes more difficult to communicate with a standardized infrastructure representing a common scheme of authentication and authorization. Sometimes, each service needs a unique representation of credentials and message formats. In a trusted inter-organization or multi-partner communication scenario, eliminating point-to-point security and enabling interoperable mechanisms for single sign-on (SSO), global logout, identity registration, and termination are mandated. This is accomplished by adopting Liberty alliance standards, which define rules and guidelines for defining federated identities, identity registration and revocation, SSO with multiple partner services, global logout, and so forth. Thus, it becomes very important to provide a security intermediary infrastructure that can handle multiple recipients using a standards-based framework, that can provide message-level configuration security mechanisms, and that can support SSO for accessing disparate security infrastructures.

Forces

- You want to use a security intermediary to support Web servicesbased workflow applications or to send messages to multiple service endpoints.
- You want to configure element-level security and access control that apply message-level security mechanisms, particularly authentication tokens and signatures and encrypted portions using XML digital signature or XML Encryption.
- You want to make sure to reveal only the required portions of a protected message to a target recipient.
- You want to implement SSO by interacting with an identity provider authority to generate SAML assertions and XACML-based access control lists for accessing Web services providers and applications that rely on SAML assertions.
- You want to incorporate a global logout mechanism that sends a logout notification to all participating service endpoints.
- You want to notify participating service providers when an identity is registered, revoked, and terminated.
- You want to dynamically apply security criteria through message transformations and canonicalizations before forwarding them to their intended recipients.
- You want to filter incoming message headers for security requirements and dynamically apply context-specific rules and other required security mechanisms before forwarding the messages to an endpoint.
- You want to support document-based Web services, particularly by checking document-level credentials and attributes.
- You want to enforce centralized logging for incoming messages, faults, messages sent, and intended recipients of the messages.
- You want to configure multiple message formats and support XML schemas that guarantee interoperability with intended service endpoints without compromising message security.
- You want to meet the mandated regulatory requirements defined by Web-services partners.
- You want to use a centralized intermediary that provides mechanisms for configuring message-level security headers supporting XML security specifications such as OASIS WS-Security, XML Signature, XML Encryption, SAML, XACML, and Liberty Alliance.

Example

(Nothing given)

Solution

The Secure Message Router pattern is used to establish a security intermediary infrastructure that aggregates access to multiple application endpoints in a workflow or among partners participating in a Web-services transaction. It acts on incoming messages and dynamically provides the security logic for routing messages to multiple endpoint destinations without interrupting the flow of messages. It makes use of a security configuration utility to apply endpoint-specific security decisions and mechanisms, particularly configuring message-level security that protects messages in entirety or reveals selected portions to its intended recipients. During operation, the Secure Message Router pattern works as a security enforcement point for outgoing messages before sending them to their intended recipients by providing endpoint-specific security services, including SSO, access control, and message-level security mechanisms. In addition, it can also provide identity-federation mechanisms that notify service providers and identity providers upon SSO, global logout, identity registration, and termination. In effect, a Secure Message Router must handle tasks such as:

- Configuring message-level security that allows signing and encrypting an XML message or its selected elements intended for multiple service endpoints.
- Configuring SSO access with multiple Web-services endpoints using SAML tokens and XACML assertions that can act as SSO session tickets.
- Supporting the use of XKMS-based PKI services to retrieve keys for signing and encrypting appropriate message parts specific to a service endpoint or to participate in workflow.
- Notifying all participating service providers and identity providers of SSO and global logouts.
- Notifying all participating service providers and identity providers of identity registration, revocation, and termination.
- Dynamically applying message transformation and canonicalization algorithms to meet recipient endpoint requirements or standards compliance.
- Reconfiguring incoming messages to destination-specific message formats and supporting XML schemas that guarantee interoperability with the target service endpoint.
- Centralizing logging of messages and recording of auditable trails for incoming messages, faults, and their ultimate endpoints.
- Supporting use of a Liberty-compliant identity provider and agents for identity federation and establishing a circle of trust among participating service providers.

Structure

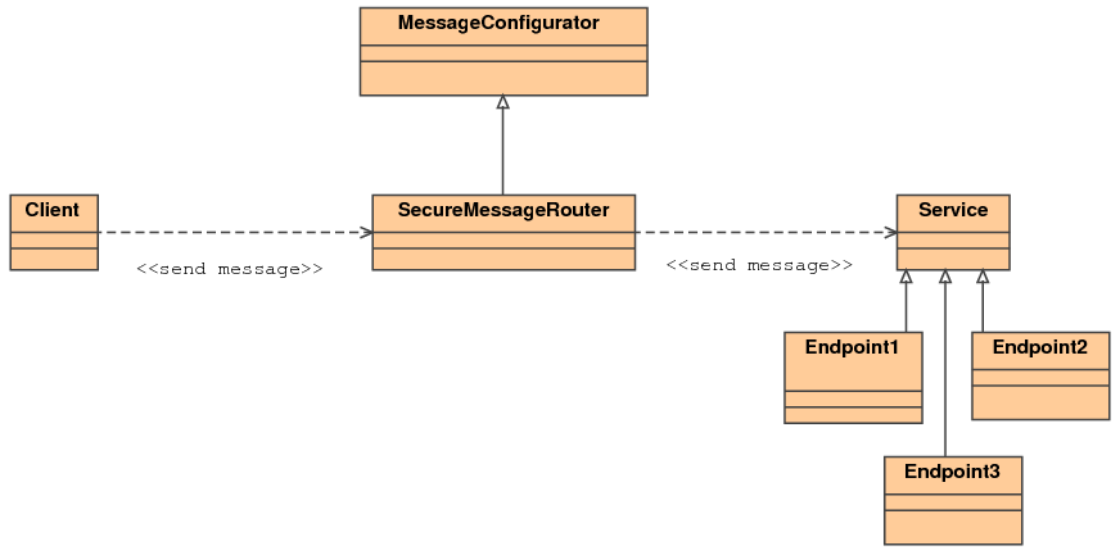
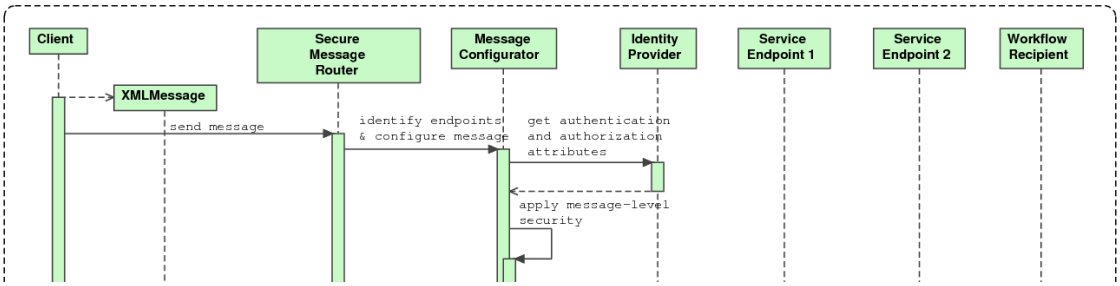


Figure 1: Class layout of the Secure Message Router.

See Figure 1

Dynamics



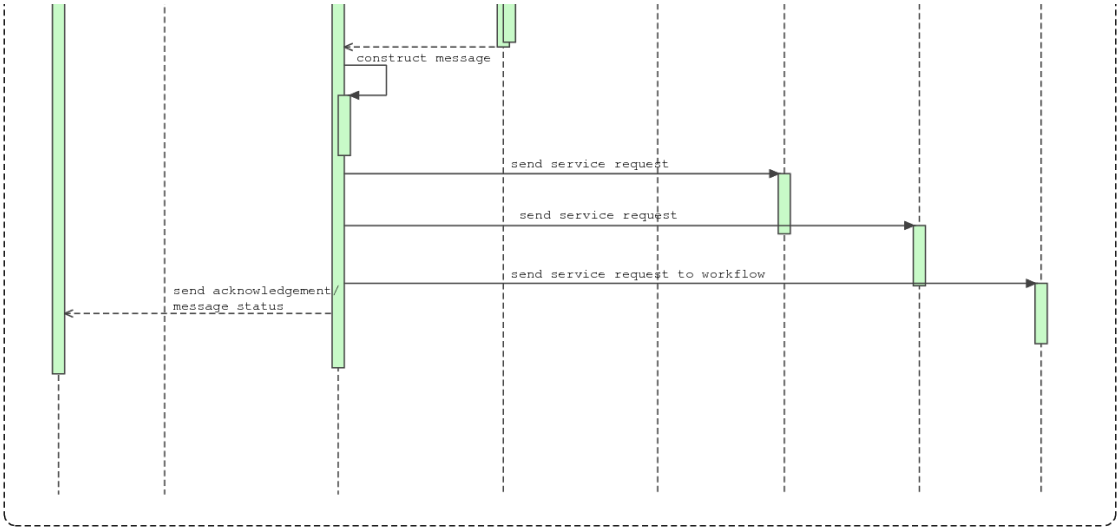


Figure 2: Event sequence for the Secure Message Router.

See Figure 2

Participants

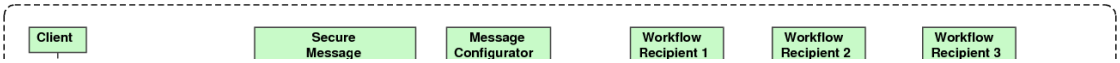
- Client. The client of the Secure Message Router pattern can be any application that initiates a service request to access a single endpoint or multiple service endpoints. Typically, it can be any application component or a Message Interceptor Gateway that sends requests or responds to a Web-services transaction.
- Secure Message Router. The Secure Message Router allows configuring message-level security mechanisms and provides support for Liberty-enabled services such as Federated SSO, global logout, identity registration, and termination services by interacting with a Liberty-enabled identity provider.
- Message Configurator. The Message Configurator plays a secondary role as the Secure Message Router pattern. It implements all the methods intended for configuring message-level security intended for a specified endpoint. It makes use of configuration tables that identify the message, service endpoint and intermediaries, message-level access privileges, validating XML schemas, transformations, and compliance requirements. It signs and encrypts messages in their entirety or selected portions, as specified in the configuration table.
- Identity Provider. The identity provider represents a Liberty-compliant service provider that delivers federated-identity services such as federated single sign-on, global logout, identity registration, termination, authentication, authorization, and auditing.
- Request. The Request message represents an XML document that is verified by all the required security-processing tasks carried out by the Secure Message Router.
- ServiceEndpoint. The ServiceEndpoint represents the target object and the ultimate consumer of the message that the client uses to do message processing. In the case of the Secure Message Router pattern, the ServiceEndpoint can be a single provider or multiple service providers or applications that implement the business logic and processing of the client request.
- WorkflowRecipient. The WorkflowRecipient represents an endpoint that participates in a workflow or in collaboration. It is an intermediary endpoint representing an identity or business logic designated for processing the entire document or selected portions of an incoming message and then forwarding it to the next recipient in the workflow chain.

Collaborations

- The client creates an XML message and sends it to the Secure Message Router.
- The Secure Message Router hands over the message to the Message Configurator, to identify the endpoints and configure the message.
- The Message Configurator applies message-level security to the message, according to its configuration tables.
- The Secure Message Router then constructs the message that has to be sent to the recipients.
- The message is sent to the various recipients (which may be service endpoints or workflow recipients).

Implementation

XML Messaging Provider Strategy In this strategy, the Secure Message Router pattern adopts an XML-based messaging provider or message-broker infrastructure that facilitates sending and receiving of XML messages (such as SOAP or ebXML)using synchronous and asynchronous delivery mechanisms. The XML messaging provider acts as a SOAP intermediary providing message-level security-mechanism support for RPC and document-style Web-services interactions among multiple service endpoints involved in a workflow or collaboration. Figure 3 represents the sequence diagram illustrating the Secure Message Router pattern using the XML Messaging Provider Strategy.



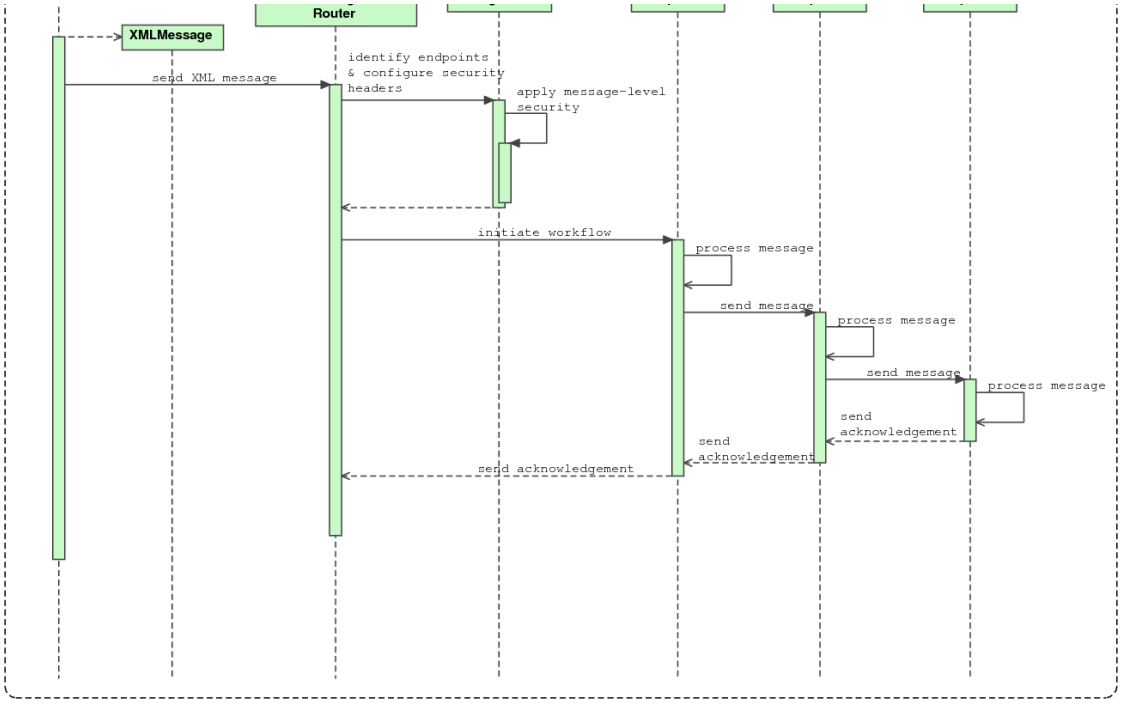


Figure 3: XML Messaging Provider sequence diagram

The Client initiates XML message requests intended for processing at multiple service endpoints in a Workflow. These messages are forwarded to the messaging provider, which acts as a SOAP security intermediary that allows configuring and applying security-header mechanisms before sending the messages to its workflow participants. Upon receipt of a request message from the client, the messaging provider processes the message and then identifies and determines its intended recipients and their message-level security requirements. It makes use of a Message configurator that provides the required methods and information for applying the required message-level security mechanisms and defining endpoint-specific requirements. The Message configurator follows a security configuration table that specifies the message identifier, endpoints, and message-level security requirements related to representing the identity, signature, encryption, timestamps, correlation ID, and other endpoint-specific attributes. After configuring the message, the messaging provider initiates the workflow by dispatching configured message to its first intended endpoint (that is, a workflow participant). The dispatched message ensures that only the privileged portions of the message are allowed to be viewed or modified by workflow participants, based on their identities and other information; all other portions of the message remain integral and confidential throughout the workflow process.

Liberty SSO Strategy The Liberty SSO Strategy adopts a federated network identity architecture based on the Liberty Alliance specifications. Using a Liberty-enabled identity provider, this strategy allows establishing circle-of-trust (CoT)relationships via identity federation to enable secure data communication among the service providers over the Internet. The service providers rely on a Liberty-enabled identity provider, which acts as a trust provider that defines and establishes identity federation-based trust relationships and also plays the role of an authority for issuing security assertions that represents authentication, authorization, and other attribute information. In this strategy, the Secure Message Router pattern makes use of a Liberty-enabled identity provider to link service endpoints, and issue XML-based security assertions. Using the security assertions provided by the service provider, it initiates SSO with partner service endpoints and also uses authorization and other attribute assertions to support message-level security mechanisms for sending XML messages. Figure 4 represents the sequence diagram illustrating the Secure Message Router using the Liberty SSO strategy.

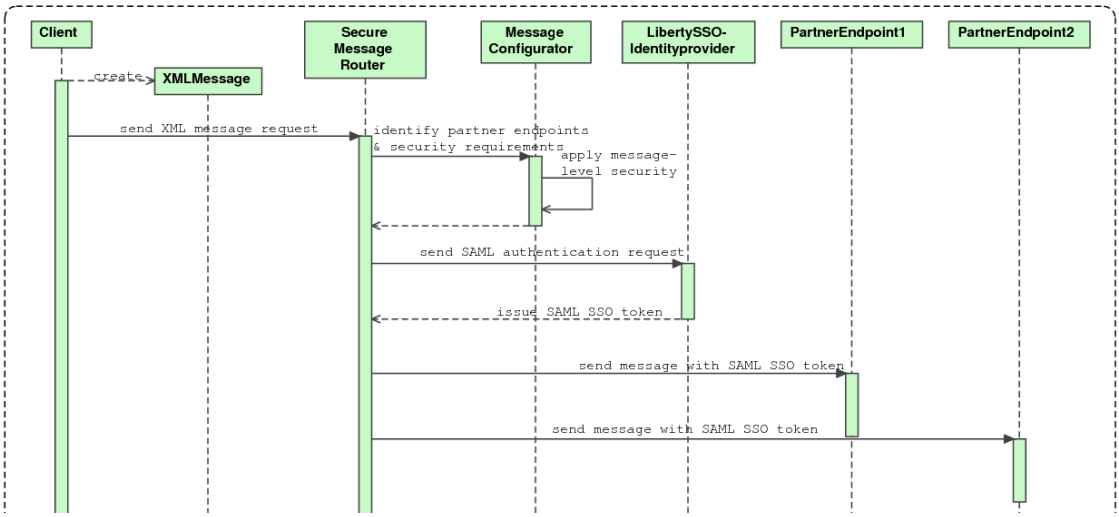


Figure 4: Liberty SSO sequence diagram

During operation, the client will make use of Secure Message Router to process the message, determine its intended endpoint recipients using a message Configurator, and then interact with a Liberty-enabled identity provider to establish SSO with partner endpoints. The Secure Message Router communicates with the Liberty-enabled identity provider using a Liberty-agent via a request and response protocol that works as follows:

- The Secure Message Router initiates a request to the service provider, which sends a SAML authentication request to an identity provider that instructs the identity provider to provide an authentication assertion.
- The identity provider responds with a SAML authentication response containing SAML artifacts or an error.
- The Secure Message Router uses the SAML artifacts as an SSO token to interact with all partner endpoints and to initiate the transaction. The partner endpoints trust the SSO tokens issued by the Liberty-enabled identity provider that established the identity federation.In addition to the above, the Secure Message Router also facilitates other Liberty-enabled services and tasks, such as notification of identity registration, termination, and global logout to all partner endpoints.

Pitfalls

- High availability and reliability. The Secure Message Router infrastructure and every component that interacts with it must provide high availability and reliability. It becomes very important to ensure100percent availability so that the message router can be better than the weakest link in the Web-services architecture. Failure to provide high availability may result in monetary loss and security vulnerabilities.
- Fault tolerance. The Secure Message Router is also expected to be fault tolerant in order to support security and to enhance reliability and performance of the infrastructure. When a failure is detected, it must be transparently replaced with a redundant infrastructure. The failure should not jeopardize any existing outbound requests or responses or their intermediate processing states. There must be a recovery mechanism that can read all outstanding service requests and paused requests with intermediate states and forward them for further processing with the Secure Message Router without skipping any existing security mechanisms.
- Provider issues. From an implementation standpoint, there are not many messaging providers that facilitate standards-based XML message workflow, multi-hop Web-services communication, and Liberty SSO. Using nonstandard implementations affects the secure message-router-based architecture with noticeable problems related to incompatible messages, routing failures, longer latencies, and lack of guaranteed message delivery. In general, these issues directly affect security and reliability of Web services or workflow communication using multiple Web-services endpoints. The adoption of emerging Web-services standards such as BPEL4WS, WS-Reliability, WS-Reliable Messaging, WS-*, and their compliant products is expected to provide interoperable workflow collaboration, reliability, and guaranteed message delivery protocols.
- Enabling interoperability in a workflow?The Secure Message Router must pre-verify the messages for interoperability before sending them to participants in a workflow or intended recipients. The interoperability requirements of the recipient endpoint with regard to WS-I profiles, XML schemas, transformations, canonicalizations, and other endpoint-specific attributes must be specified using the Message Configurator.
- Scalability?It is important to verify the Secure Message Router solution architecture for scalability to eliminate bottlenecks when communicating with multiple endpoints. This is critical to the success of every Message Router to perform resource-intensive tasks such as applying signatures, encryptions, and transformations without the expense of scalability and overall performance.

Consequences

Adopting the Secure Message Router pattern facilitates applying SSO mechanisms and trusted communication when the target message is exchanged among multiple recipients or intended to be part of a workflow. It also allows selectively applying XML Encryption and XML Signature at the element level by ensuring that content is not exposed to everyone unless the recipient has privileges to access the selected fragments of the message. This helps in securely sending messages to multiple recipients and ensuring that only selected fragments of the message are revealed or modified by the privileged recipients. With the support for Liberty-enabled identity providers, it establishes a circle of trust among participation endpoints and facilitates SSO by securely sharing identity information among the participating service endpoints. The Secure Message Router also ensures seamless integration and interoperability with all participating endpoints by sending destination-specific messages. In addition, the Secure Message Router pattern provides the following benefits:

- Centralized routing. The Secure Message Router delivers a centralized message intermediary solution for applying message-level security mechanisms and enabling SSO access to multiple endpoints. This allows configuring a centralized access control and processing subsystem for incorporating all security-related operations for sending

messages to multiple service endpoints. It offers centralized management of related services, including authentication, authorization, faults, encryption, audit trails, metering, billing, and so on. This improves manageability.

- Modularity and maintainability. Centralizing all security mechanisms and configuring access-control policies using a single intermediary keep the message-sender application interfaces separated from security operations. This enhances a service with a modular subsystem designated for security and reduces complex tasks at the service endpoint of a Web services provider. This also saves significant application processing time and resources at the message-sending application endpoint.
- Reusability and extensibility. The Secure Message Router pattern encapsulates all direct access to participating service endpoints, facilitating a common reusable solution that is necessary for protecting multiple service endpoints. It also offers extensibility by allowing you to incorporate more message-level security mechanisms and functionalities specific to the target endpoints.
- Improved testability. The Secure Message Router infrastructure separates the security architectural model from the underlying message-sender's service endpoint. This improves ease of testability and extensibility of the security architecture.

Known uses

Secure Pipe

Pattern documentation

Quick info

Intent: You need to provide privacy and prevent eavesdropping and tampering of client transactions caused by man-in-the-middle attacks.

Problem

Web-based transactions are often exposed to eavesdropping, replay, and spoofing attacks. Anytime a request goes over an insecure network, the data can be intercepted or exposed by unauthorized users. Even within the confines of a VPN, data is exposed at the endpoint, such as inside of an intranet. When exposed, it is subject to disclosure, modification, or duplication. Many of these types of attacks fall into the category of man-in-the-middle attacks. Replay attacks capture legitimate transactions, duplicate them, and resend them. Sniffer attacks just capture the information in the transactions for use later. Network sniffers are widely available today and have evolved to a point where even novices can use them to capture unencrypted passwords and credit card information. Other attacks capture the original transactions, modify them, and then send the altered transactions to the destination. This is a common problem shared by all applications that do business over an untrusted network, such as the Internet. For simple Web applications that just serve up Web pages, it is not cost-effective to address these potential attacks, since there is no reason for attackers to carry out such an attack (other than for defacement of the pages)and therefore the risk is relatively low. But, if you have an application that requires sending sensitive data (such as a password)over the wire, you need to protect it from such an attack.

Forces

- You want to avoid writing application logic to provide the necessary protection; it is better to push this functionality down into the infrastructure layer to avoid complexity.
- You want to make use of hardware devices that can speed up the cryptographic algorithms needed to prevent confidentiality-and integrity-related issues.
- You want to adopt tested, third-party products for reliable data and communication security.
- You want to limit the protection of data to only sensitive data due to the large processing overhead and subsequent delay due to encryption.

Example

(Nothing given)

Solution

Use a Secure Pipe to guarantee the integrity and privacy of data sent over the wire. A Secure Pipe provides a simple and standardized way to protect data sent across a network. It does not require application-layer logic and therefore reduces the complexity of implementation. In some instances, the task of securing the pipe can actually be moved out of the application and even off of the hardware platform altogether. Because a Secure Pipe relies on encrypting and decrypting all of the data sent over it, there are performance issues to consider. A Secure Pipe allows developers to delegate processing to hardware accelerators, which are designed especially for the task.

Structure

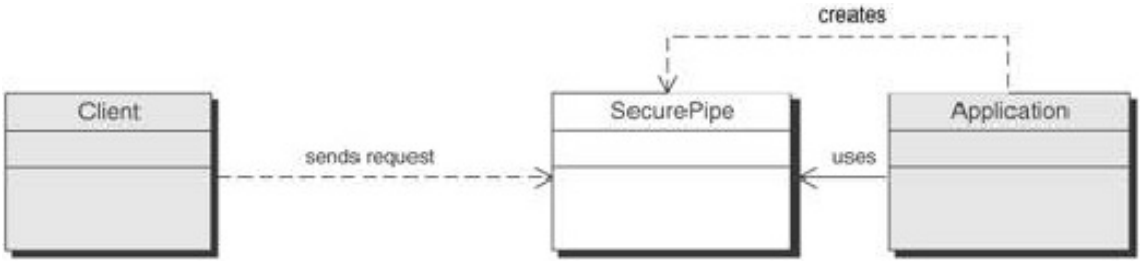


Figure 1: Secure Pipe class diagram.

Figure 1 depicts a class diagram of the Secure Pipe pattern in relation to an application.

Dynamics

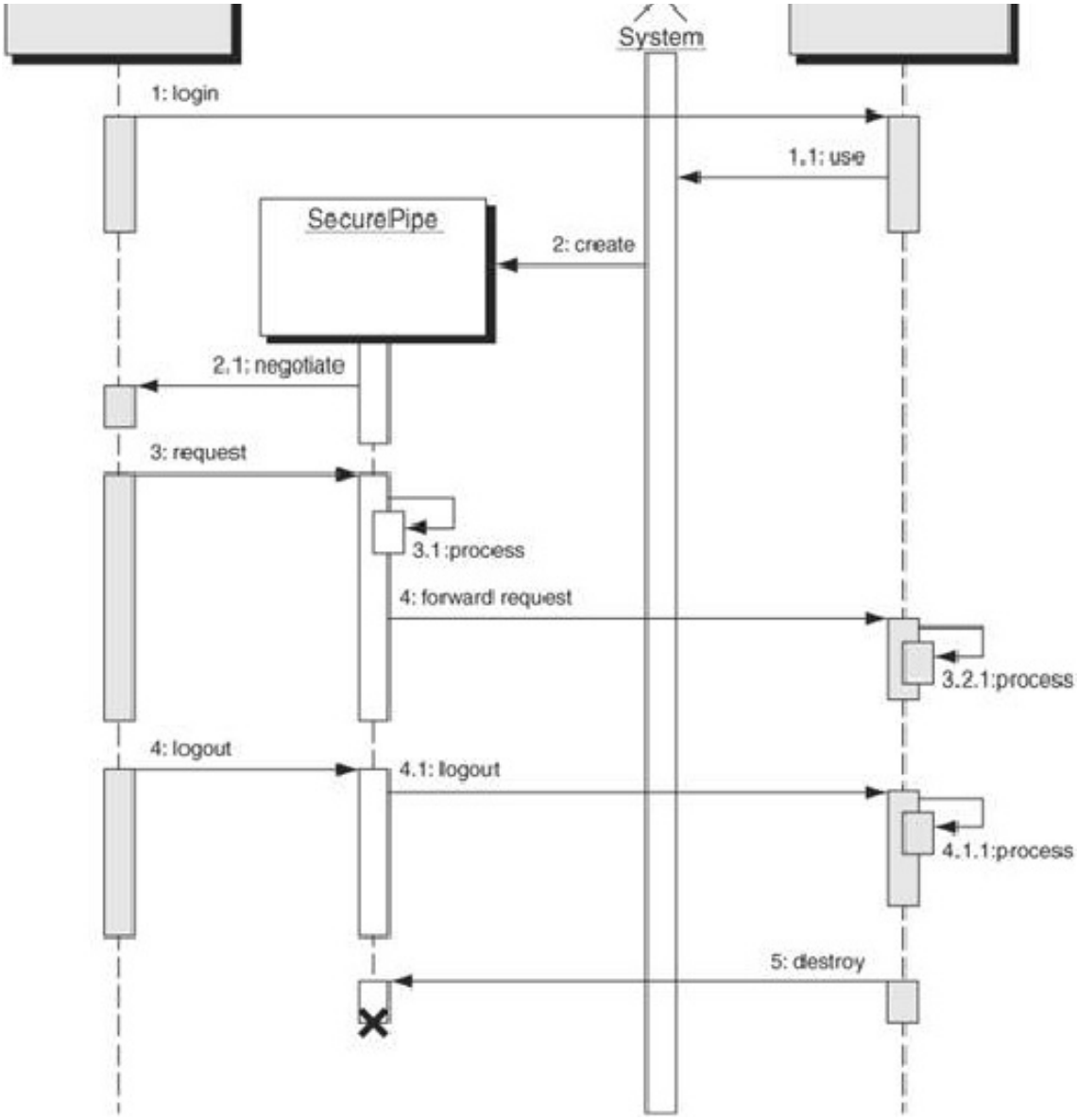


Figure 2: Secure Pipe sequence diagram.

The sequence diagram of the operation of a secure pipe is given in Figure 2 .

Participants

The following participants are illustrated in the sequence diagram shown in Figure 2 .

- **client** Initiates a login with the application.
- **application** Creates a system level SecurePipe over which to communicate with the client.
- **securepipe** A SecurePipe is an encrypted communications channel that provides data privacy and integrity between two endpoints.

Collaborations

Figure 2 shows a sequence diagram depicting use of the Secure Pipe pattern. In the sequence shown, a client needs to connect to an application over a secure communication line. The diagram shows how the client and the application communicate using the Secure Pipe. The interaction is as follows.

- Client sends login request to the Application.
- Application uses System to create a SecurePipe.
- SecurePipe negotiates parameters of the secure connection with the Client.
- Client sends request to the Application.
- SecurePipe processes the request and creates a secure message by encrypting the data. It sends the message over the wire to the corresponding SecurePipe components on the Application.
- SecurePipe on the Application processes the request received from the Client by decrypting it and then forwards the decrypted message to the Application.
- Client sends a logout request.
- Application destroys the SecurePipe.There are two components of the Secure Pipe pattern: the client-side component and

the server-side component. These components work together to establish a secure communication. Typically, these components would be SSL or TLS libraries that the client's Web browser and the application use for secure communications.

Implementation

There are several strategies for implementing a Secure Pipe pattern, each with its own set of benefits and drawbacks. Those strategies include: web-server-based SSL/TLS, hardware-based cryptographic accelerator cards, application-layer encryption using the Java Cryptography Extension (JCE).

All major Web-server vendors support SSL. All it takes to implement SSL is to obtain or create server credentials from a CA, including the server X.509certificate, and configure the Web server to use SSL with these credentials. Before enabling SSL, the Web server must be security-hardened to prevent compromise of the server's SSL credentials. Since these credentials would be stored on the Web server, if that server were compromised, an attacker could gain access to the server's credentials (including the private key associated with the certificate)and would then be able to impersonate the server. To enhance SSL performance, a specialized hardware referred to as SSL accelerators can be used to assist with cryptographic computations. When a new SSL session is established, the Web server will use the SSL accelerator hardware to accept the SSL connection and perform the necessary cryptographic calculations for verifying certificates, encrypting session keys, and so forth instead of having the server CPU perform these calculations in software. SSL acceleration improves Web application performance by relieving servers of complex public key operations, bulk encryption, and high SSL traffic volumes. A network appliance is a stand-alone piece of hardware dedicated to a particular purpose. In this strategy, we refer to network appliances that act as dedicated SSL/TLS endpoints. They make use of hardware-based encryption algorithms and optimized network ports. Network appliances move the responsibility for establishing secure connections further out into the perimeter and provide greater performance. They sit out in front of the Web servers and promote a greater degree of reusability, since they can service multiple Web servers and applications. However, the security gap between the Secure Pipe endpoint and the application has widened as the appliance is moved logically and physically further away from the application endpoint on the network. In some cases, Secure Pipe can be implemented in the application layer by making use of Java Secure Socket Extensions (JSSE)framework. JSSE allows enabling secure network communications using Secure Sockets Layer (SSL)and Transport Layer Security (TLS)protocols. It includes functionality for data encryption, server authentication, message integrity, and optional client authentication. Example9-18shows how to create secure RMI connections by implementing an RMI Secure Socket Factory that provides SSL connections for the RMI protocol, which provides a secure tunnel.

Pitfalls

The Secure Pipe pattern is an integral part of most Web server infrastructures because we make use of SSL/TLS between the client and the Web Server. Without it, mechanisms for ensuring data privacy and integrity must be performed in the application itself, leading to increased complexity, reduced manageability, and the inability to push the responsibility down into the infrastructure. Infrastructure:

- Infrastructure for ensuring data privacy and integrity. Any communication over the Internet or an intranet are subject to attack. Attackers can sniff the wire and steal data, alter it, or resend it. Developers need to protect this data by encrypting it and using digitally signed timestamps, sequence numbers, and checksums. Using industry standards, such as SSL and TLS, developers can secure data that is interoperable with Web browsers and other client applications.
- Data encryption performance. Encryption is an expensive processing task. Hardware devices can increase throughput and response times by performing the necessary cryptographic functions in hardware, freeing up CPU cycles for the application.Web Tier:
- Server certificates. One of the requirements with SSL is public key management and trust models. To solve this problem, certificate authorities were established to act as trusted third parties responsible for the authentication and validation of public keys through the use of digital certificates. Several CA's certificates are packaged in Web browsers and in the Java Runtime Environment's cacerts file. This allows developers to take advantage of client certificate chains to ensure that the requesting client was properly authenticated by a trusted third party.

Consequences

- Ensures data confidentiality and integrity during communication. The Secure Pipe pattern enforces data confidentiality and integrity using a mixture of encryption and digital signatures. Using SSL/TLS mechanisms, all point-to-point communications links can be secured from man-in-the-middle attacks.
- Promotes interoperability. Using industry-standard infrastructure components to implement the Secure Pipe pattern allows application owners to achieve greater interoperability with clients and partners. By taking advantage of infrastructure products and standard protocols like SSL/TLS, IPSEC, application-level interoperability can be achieved between Web browser clients and Web-server-based applications.
- Improves performance. Delegating CPU-intensive cryptographic operations into hardware infrastructure often shows performance benefits. Strategies such as SSL accelerators and network appliances often demonstrated quadruple performance over application layer processing.
- Reduces complexity. The Secure Pipe pattern reduces complexity by separating complex cryptographic algorithms and procedures from application logic. The details associated with providing secure communications can be pushed down into the infrastructure, thus freeing up the application to focus on business logic rather than security.Will Secure Pipe impact

performance?Using a Secure Pipe will certainly impact performance noticeably. Do not use it when it is not required. Many business cases dictate securing sensitive information and therefore a Secure Pipe must be used. If your Web application mandates the need for protecting passwords and sensitive information in transit, use a Secure Pipe (such as HTTPS)just for those operations. Otherwise, you may conduct all other transactions over standard HTTP communication.

Are there any compatibility issues with Secure Pipe?Implementing a Secure Pipe requires an agreement between the communicating peers. The client and the server must support the same cryptographic algorithms and key lengths as well as agree upon a common protocol for exchanges keys. SSL and TLS provide standard protocols for ensuring this compatibility by providing handshake mechanisms that allow clients and servers to negotiate algorithms and key lengths.

Known uses

Secure Service Facade

Pattern documentation

Quick info

Intent: You need a secure gateway mandating and governing security on client requests, exposing a uniform, coarse-grained service interface over fine-grained, loosely coupled business services that mediates client requests to the appropriate services.

Problem

Having more access points in the Business tier leads to more opportunities for security holes. Every access point is then required to enforce all security requirements from authentication and authorization to data validation and auditing. This becomes exacerbated in applications that have existing Business-tier services that are not secured.

Retrofitting security to security-unaware services is often difficult. Clients must not be made aware of the disparities between service implementations in terms of security requirements, message specifications, and other service-specific attributes. Offering a unified interface that couples the otherwise decoupled business services makes the design more comprehensible to clients and reduces the work involved in fulfilling client requests.

Forces

- You want to off-load security implementations from individual service components and perform them in a centralized fashion so that security developers can focus on security implementation and business developers can focus on business components.
- You want to impose and administer security rules on client requests that the service implementers are unaware of in order to ensure that authentication, authorization, validation, and auditing are properly performed on all services.
- You want a framework to manage the life cycle of the security context between interactive service invocations by clients and to propagate the security context to appropriate servers where the services are implemented.
- You want to reduce the coupling between fine-grained services but expose a unified aggregation of such services to the client through a simple interface that hides the complexities of interaction between individual services while enforcing all of the overall security requirements of each service.
- You want to minimize the message exchange between the client and the services, storing the intermittent state and context on the server on behalf of the client instead.

Example

(Nothing given)

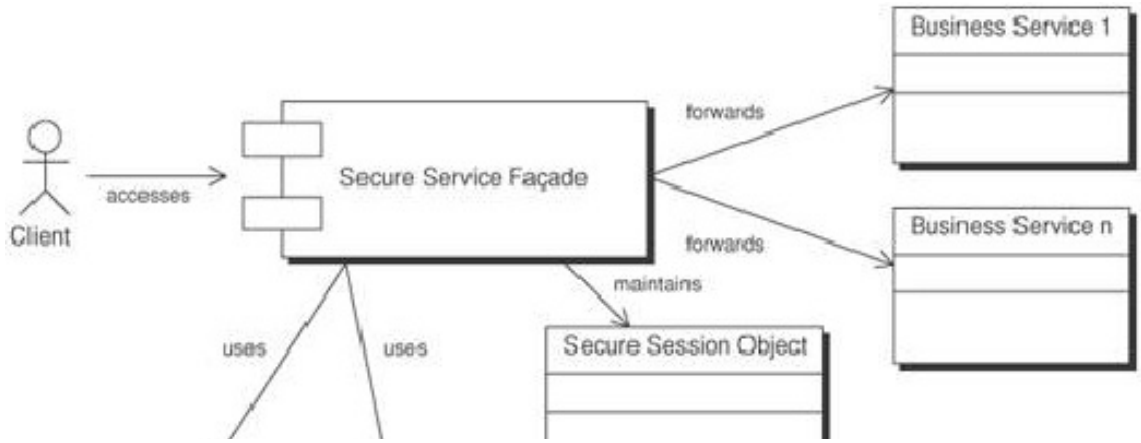
Solution

Use a Secure Service Facade to mediate and centralize complex interactions between business components under a secure session.

Use a Secure Session Facade to integrate fine-grained, security-unaware service implementation and offer a unified, security-enabled interface to clients. The Secure Service Facade acts as a gateway where client requests are securely validated and routed to the appropriate service implementations, often maintaining and mediating the security and workflow context between interactive client requests and between fine-grained services that fulfill portions of the client requests.

Structure

Figure 1 illustrates a Secure Service Facade class diagram.



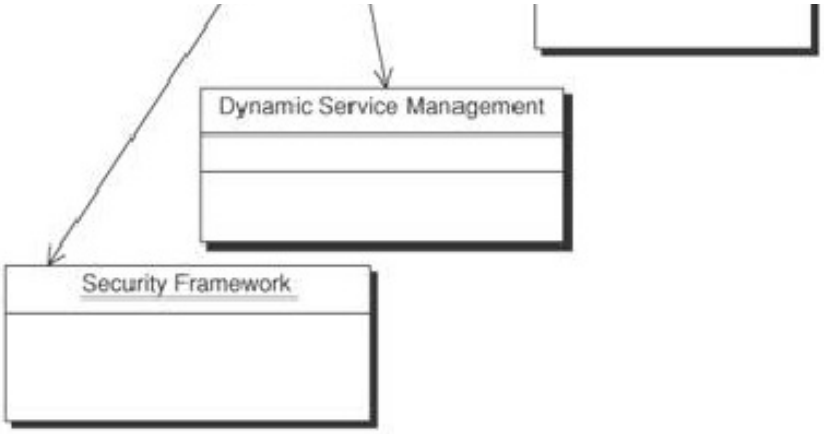


Figure 1: Class layout of the Secure Service Facade.

Dynamics

Figure 2 depicts a sequence diagram for a typical Secure Service Facade implementation that corresponds to the structure description in the preceding section.

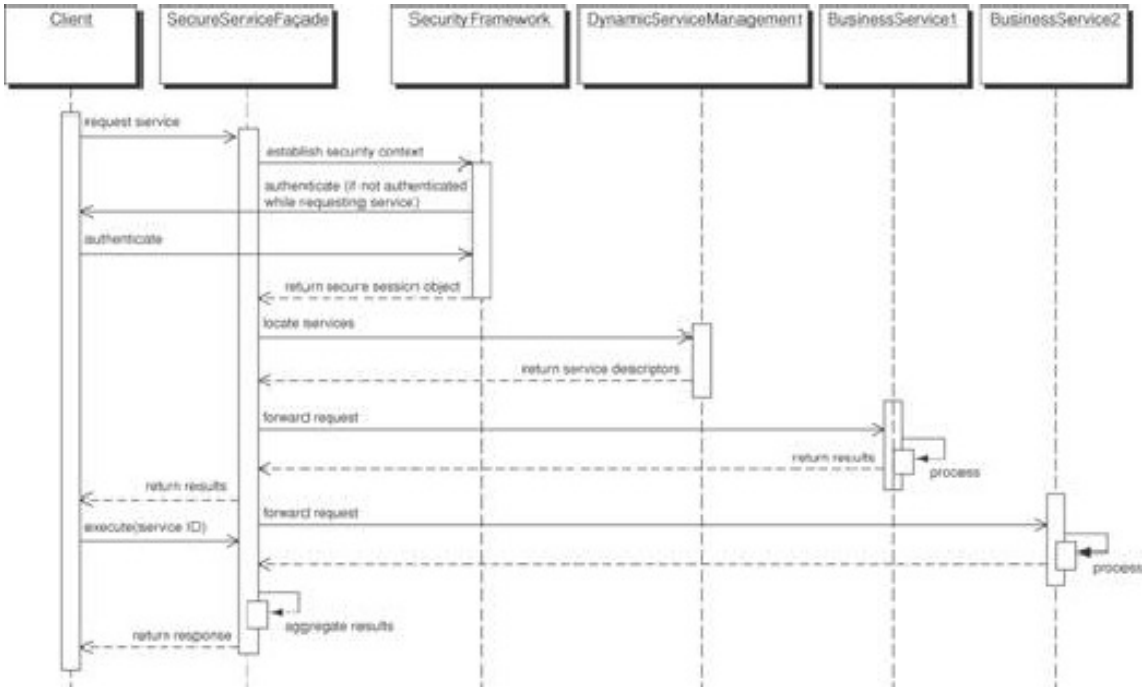


Figure 2: Sequence diagram for the Secure Service Facade.

The fine-grained business services are not directly exposed to the client. The services themselves maintain loose coupling between each other and the fa Ã§ ade. The fa Ã§ ade takes the responsibility of unifying the individual services in the context of the client request. The service fa Ã§ ade contains no business logic itself and therefore requires no protection.

Participants

- Client. A client sends a request to perform a specific task with the appropriate service descriptors to the Secure Service Facade, optionally incorporating the decision-tree predicates to determine the sequence services to be invoked.
- The Secure Service Facade deciphers the client request, verifies authentication, fulfills the request, and returns the results to the client. In doing so, it may use the following components:
    - Security Framework. The fa Ã§ ade uses the existing enterprise-wide security framework implemented using other security patterns discussed in this book. Such a framework can be leveraged for authentication, authorization and access control, security assertions, trust management, and so forth. If the request is missing any credentials, the client request could be terminated or the client could be asked to furnish further credentials.
    - Dynamic Service Framework/Service Locator. The fa Ã§ ade uses the Dynamic Service Framework or Service Locator to locate the services that are involved in fulfilling the request. The services could reside on the same host or be distributed throughout an enterprise. In either case, the fa Ã§ ade ensures that the security context established using the security framework is correctly propagated to any service that expects such security attributes. The fa Ã§ ade then establishes the execution logic and invokes each service in the correct order.

Collaborations

The Facade is the endpoint exposed to the client and could be implemented as a stateful session bean or a servlet endpoint. It uses the security framework (implemented using other patterns)to perform security-related tasks applicable to the client request. The framework may request the client to present further credentials if the requested service mandates doing so and if those credentials were not found in the initial client request. The Facade then uses the Dynamic Service Management pattern to locate the appropriate service-provider implementations. The request is then forwarded to the individual services either sequentially, in parallel, or in any complex relationship order as specified in the request description.

If the client request represents an aggregation of fine-grained services, the return messages from previous sequential service invocations can be aggregated and delivered to the subsequent service to achieve a sequential workflow-like implementation. If those fine-grained services are independent of each other, then they can be invoked in parallel and the results can be aggregated before delivering to the client, thus achieving parallel processing of the client request.

Implementation

The Secure Service Facade manages the complex relationships between disparate participating business services, plugs in security to request fulfillment, and provides a high-level, coarse-grained abstraction to the client. The nature of such tasks opens up multiple choices for implementation flavors, two of which are briefly discussed now. Facade with static relationships between individual service components. The relationship between participating fine-grained services is permanently static in nature. In such cases, the fa Ã§ ade can be represented by an interface that corresponds to the aggregate of the services and can be implemented by a session bean that implements the interface. The session bean life cycle method Create can preprocess the request for security validations.

Facade with dynamic, transient relationships between individual service components. When the sequence of service calls to be invoked by the fa Ã§ ade is dependent upon the prior invocation history in the execution sequence, the decision predicates can be specified in the request semantics and used in the fa Ã§ ade implementations to determine the next service to be invoked. Such an implementation can be highly dynamic in nature, and the decision predicates can incorporate security class and compartment information to enable multilevel security in the fa Ã§ ade implementation. A different flavor can use a simple interface in the fa Ã§ ade, such as a command pattern implementation, and can mandate that the service descriptors be specified in the request message. This allows new services to be plugged-and-played without requiring changes to the fa Ã§ ade interface and is widely used in Web services.

Pitfalls

The Secure Service Facade pattern is susceptible to code bloating if too much interaction logic is incorporated. However, this can be minimized by appropriate design of the fa Ã§ ade using other common design patterns. As the gateway into the Business tier, the Secure Service Facade serves to limit the touch points between the Web and Web Services tiers and the Business tier. This means that there are fewer entry points that need to be secured and therefore fewer opportunities for security holes to be introduced. Does the Service Facade need to incorporate security?The Secure Service Proxy uses the existing security framework while aggregating fine-grained services. However, security context validation may not be required if other means of authentication and access control are pertinently enforced on the client request before it reaches the fa Ã§ ade.

Does the Secure Service Facade need to perform service aggregation?If the client requests will mostly be fulfilled by a single, fine-grained service component, there is no necessity for aggregation. In such cases, Secure Service Proxy may well suit the purpose.

Does the Secure Service Facade reduce security code duplication?If security context validation is performed by each service component, the validation at the fa Ã§ ade level may turn out to be redundant and wasteful. A planned design could reduce such duplication.

Consequences

The Secure Service Facade pattern protects the Business-tier services and business objects from attacks that circumvent the Web tier or Web Services tier. The Web tier and the Web Services tier are responsible for upfront authentication and access control. An attacker who has penetrated the network perimeter could circumvent these tiers and access the Business tier directly. The Secure Service Facade is responsible for protecting the Business tier by enforcing the security mechanisms established by the Web and Web Services tiers. By employing the Secure Service Facade pattern, developers and clients can benefit in the following ways:

- Exposes a simplified, unified interface to a client. The Secure Service Facade shields the client from the complex interactions between the participating services by providing a single unified interface for service invocation. This brings the advantages of loose coupling between clients and fine-grained business services, centralized mediation, easier management, and reduces the risks of change management.
- Off-loads security validations from lightweight services. Participating business services in a fa Ã§ ade may be too lightweight to define security policies and incorporate security processing. Secure Service Facade off-loads such responsibility from business services and offers a centralized policy management and administration of centralized security processing tasks, thereby reducing code duplication and processing redundancies.

- Centralizes policy administration. The centralized nature of the Secure Service Facade eases security policy administration by isolating it to a single location. Such centralization also makes it feasible to retrofit infrastructure security to otherwise security-unaware or existing services.
- Centralizes transaction management and incorporates security attributes. As with a generic session fa Ã§ ade, a Secure Service Facade allows applying distributed transaction management over individual transactions of the participating services. Since security attributes are accessible at the same place, transaction management can incorporate such security attributes, offering multilevel, security-driven transaction management.
- Facilitates dynamic, rule-based service integration and invocation. As explained in the preceding"Strategies"section, multiple flavors of fa Ã§ ade implementations offer a very dynamic and flexible integration of business services. Integration rules can incorporate security and message attributes in order to dynamically determine execution sequence. An external Business Rules Engine can also be plugged into such a dynamic fa Ã§ ade.
- Minimize message exchange between client and services. Secure Service Facade minimizes message exchange by caching the intermittent state and context on the server rather than on the client.The following security factors are addressed by the Secure Service Facade:
- **authentication** The Secure Session Facade pattern authenticates requests coming into the Business tier. This is often necessary when clients connect directly to the Business tier through a remote interface or in cases where the Web tier cannot be trusted to perform authentication appropriately for the Business tier.
- **auditing** The Secure Session Facade enables developers to insert auditing at the entry and exit points of the Business tier. This enables them to put an Audit Interceptor pattern, discussed earlier in this chapter, in place and decouple auditing from business logic while ensuring that no requests can be initiated without first being audited.

Known uses

Secure Session Object

Pattern documentation

Quick info

Intent: You need to facilitate distributed access and seamless propagation of security context and client sessions in a platform-independent and location-independent manner.

Problem

A multi-user, multi-application distributed system needs a mechanism to allow global accessibility to the security context associated with a client session and secure transmission of the context among the distributed applications, each with its own address space. While many choices are possible, the developer must design a standardized structure and interface to the security context. The security context propagation is essential within the application because it is the sole means of allowing different components within the application to verify that authentication and access control have been properly enforced. Otherwise, each component would need to enforce security and the user would wind up authenticating on each request. The Secure Session Object pattern serves this purpose.

Forces

- You want to define a data structure for the security context that comprises authentication and authorization credentials so that application components can validate those credentials.
- You want to define a token that can uniquely identify the security context to be shared between applications to retrieve the context, thereby enabling single sign-on between applications.
- You want to abstract vendor-specific session management and distribution implementations.
- You want to securely transmit the security context across virtual machines and address spaces when desired in order to retain the client's credentials outside of the initial request thread.

Example

(Nothing given)

Solution

Use a Secure Session Object to abstract encapsulation of authentication and authorization credentials that can be passed across boundaries. You often need to persist session data within a single session or between user sessions that span an indeterminate period of time. In a typical Web application, you could use cookies and URL rewriting to achieve session persistence, but there are security, performance, and network-utilization implications of doing so. Applications that store sensitive data in the session are often compelled to protect such data and prevent potential misuse by malicious code (a Trojan horse)or a user (a hacker). Malicious code could use reflection to retrieve private members of an object. Hackers could sniff the serialized session object while in transit and misuse the data. Developers could unknowingly use debug statements to print sensitive data in log files. Secure Session Object can ensure that sensitive information is not inadvertently exposed. The Secure Session Object provides a means of encapsulating authentication and authorization information such as credentials, roles, and privileges, and using them for secure transport. This allows components across tiers or asynchronous messaging systems to verify that the originator of the request is authenticated and authorized for that particular service. It is intended that this serves as an abstract mechanism to encapsulate vendor-specific implementations. A Secure Session Object is an ideal way to share and transmit global security information associated with a client.

Structure

The class diagram of the secure session object is included in Figure 1 .

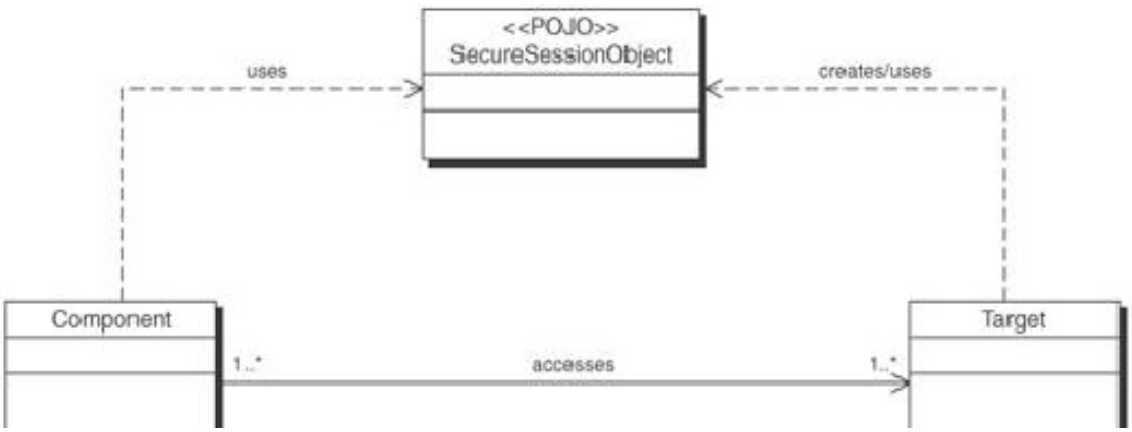


Figure 1: Secure session object class diagram.

Dynamics

The dynamics of the secure session object are depicted in Figure 2 .

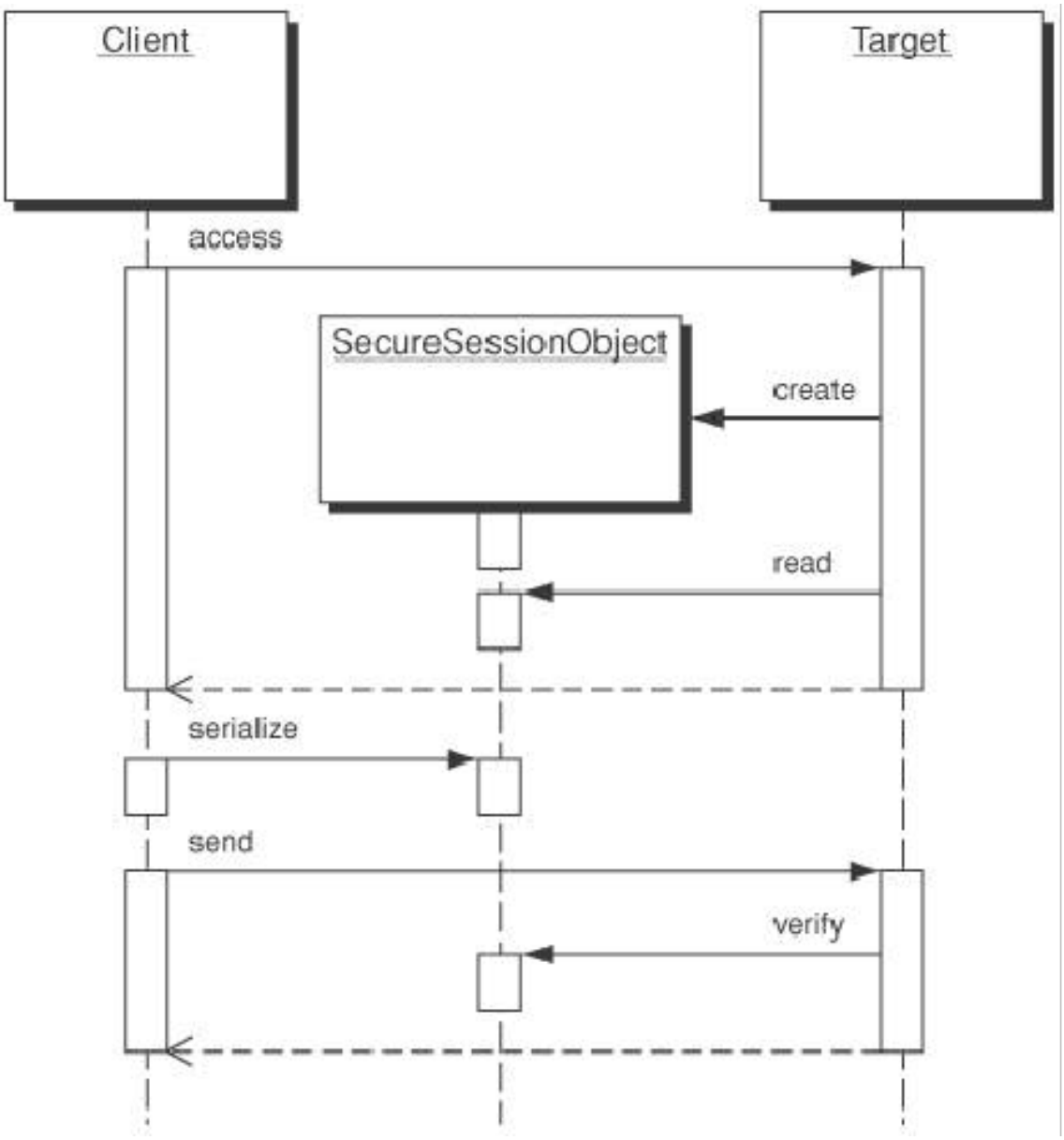


Figure 2: Secure session object dynamics.

Participants

- Client. The Client sends a request to a Target resource. The Client receives a SecureSessionObject and stores it for submitting in subsequent requests.
- SecureSessionObject. SecureSessionObject stores information regarding the client and its session, which can be validated by consumers to establish authentication and authorization of that client.
- Target. The Target creates a SecureSessionObject. It then verifies the SecureSessionObject passed in on subsequent requests.

Collaborations

The Secure Session Object is implemented through the following steps:

- Client accesses a Target resource.
- Target creates a SecureSessionObject.
- Target serializes SecureSessionObject and returns it in response.
- Client needs to access Target again and serialize SecureSessionObject from the last request.
- Client accesses Target, passing the SecureSessionObject created previously in response to the request.
- Target receives the request and verifies the SecureSessionObject before completing the request.

Implementation

You can use a number of strategies to implement Secure Session Object. The first strategy is using a Transfer Object Member, which allows you to use Transfer Objects to exchange data across tiers. The second strategy is using an Interceptor, which is applicable when transferring data across remote endpoints, such as between tiers. In the **Transfer Object Member** strategy (see Figure 3), the Secure Session Object is passed as a member of the more generic Transfer Object. This allows the target component to validate the Secure Session Object wherever data is passed using a Transfer Object. Because the Secure Session Object is contained within the Transfer Object, the existing interfaces don't require additional instances of the Secure Session Object. This keeps the interfaces from becoming brittle or inflexible and allows easy integration of the Secure Session Object into existing applications with established interfaces.

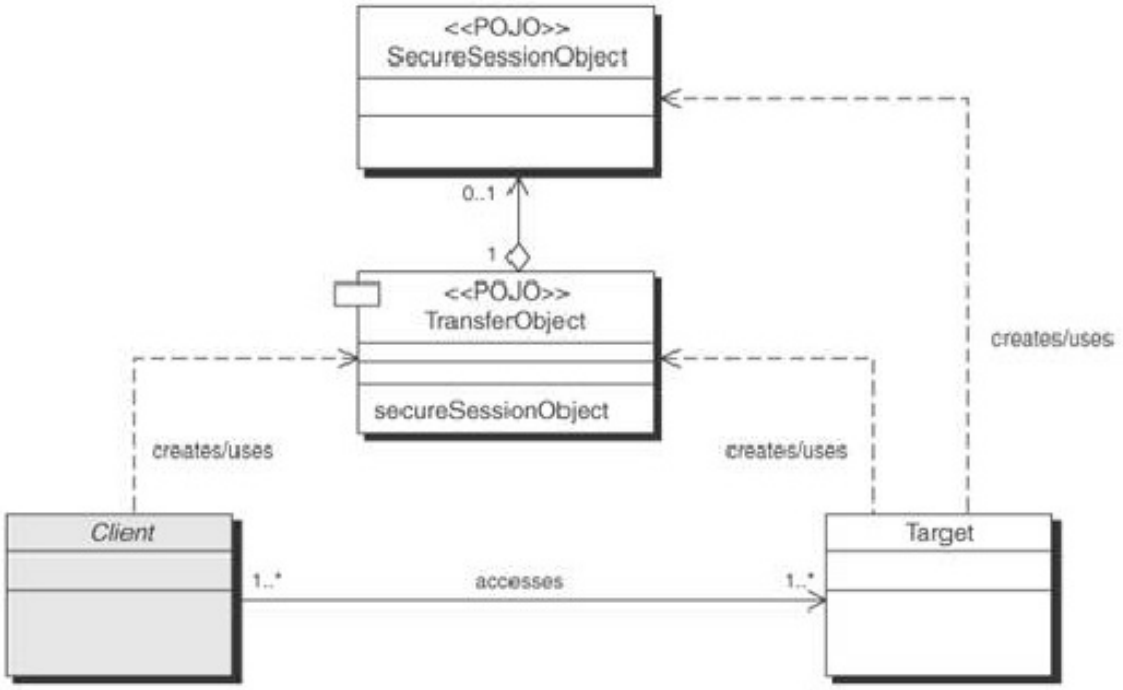


Figure 3: Secure session object---Transfer object member strategy.

In the **Interceptor Strategy** (see Figure 4), which is mostly applicable to a distributed client-server model, the client and the server use appropriate interceptors to negotiate and instantiate a centrally managed Secure Session Object. This session object glues the client and server interceptors to enforce session security on the client-server communication. The client and the server interceptors perform the initial handshake to agree upon the security mechanisms for the session object. The client authenticates to the server and retrieves a reference to the session object via a client interceptor. The reference could be as simple as a token or a remote object reference. After the client has authenticated itself, the server interceptor uses a session object factory to instantiate the Secure Session Object and returns the reference of the object to the client. The client and the server interceptors then exchange messages marshalled and unmarshalled according to the security context maintained in the Secure Session Object.

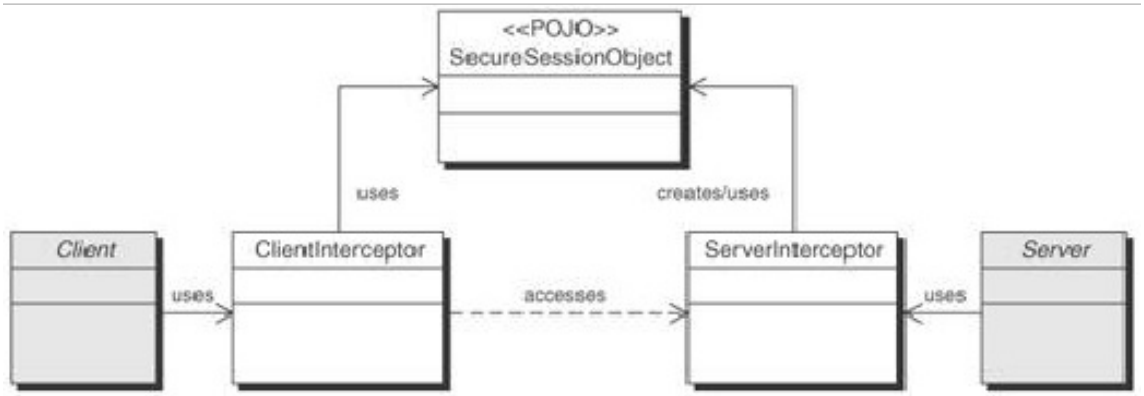


Figure 4: Secure session object---Interceptor strategy.

This strategy offers the ability to update or replace the security implementations in the interceptors independently of one another. Moreover, any change in the Secure Session Object implementation causes changes only in the interceptors instead of the whole application.

Pitfalls

- Authentication. The Secure Session Object enforces authentication of clients requesting Business-tier components. Target components or interceptors for those components can validate the Secure Session Object passed in on request and therefore assure that the invoking client was properly authenticated.
- Authorization. The Secure Session Object can enforce authorization on Business-tier clients as well. While it provides a coarse-grained level of authorization, just by being in the request or not it can be extended to include and enforce fine-grained authorization.Is Secure Session Object too bloated?Abstracting all session information into a single composite object may increase the object size. Serializing and de-serializing such an object quite frequently degrades performance. In such cases, one could revisit the object design or serialization routines to alleviate the performance degradation.

Concurrency implications. Many components associated with the client session could be competing to update and read session data, which could lead to concurrency issues such as long wait times or deadlocks. A careful analysis of the possible scenarios is recommended.

Consequences

The Secure Session Object prevents a form of session hijacking that could occur if session context is not propagated and therefore not checked in the Business tier. This happens when the Web tier is distributed from the Business tier. This also applies to message passing over JMS as well. The ramifications of not using a Secure Session Object are that impersonation attacks can take place from inside the perimeter. By employing the Secure Session Object pattern, developers benefit in the following ways: Controlled access and common interface to sensitive information. The Secure Session Object encapsulates all sensitive information related to session management and communication establishment. It can then restrict access to such information, encrypt with complete autonomy, or even block access to information that is inappropriate to the rest of the application. A common interface serves all components that need access to the rest of the session data and offers an aggregate view of session information.

Optimized security processing. Since Secure Session Object can be reused over time, it minimizes repetition of security tasks such as authentication, secure connection establishment, and encryption and decryption of shared, static data.

Reduced network utilization and memory consumption. Centralizing management and access to a Secure Session Object via appropriate references and tokens minimizes the amount of session information exchanged between clients and servers. Memory utilization is also optimized by sharing security context between multiple components.

Abstract vendor-specific session management implementations. The Secure Session Object pattern provides a generic data structure for storing and retrieving vendor-specific session management information. This reduces the dependency on a particular vendor and promotes code evolution.

Known uses

Security Association

Pattern documentation

Quick info

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.

Problem

Instantiating thepattern 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 thefor each message would be prohibitive. Therefore it is desirable to enable an instance ofto 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 thepattern.

Forces

- Thepattern is used to protect messages in a communications channel.
- Some security parameters of thepattern 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 communication channel without renegotiating the security parameters of the channel for each message.

Example

(Nothing given)

Solution

(Nothing given)

Structure

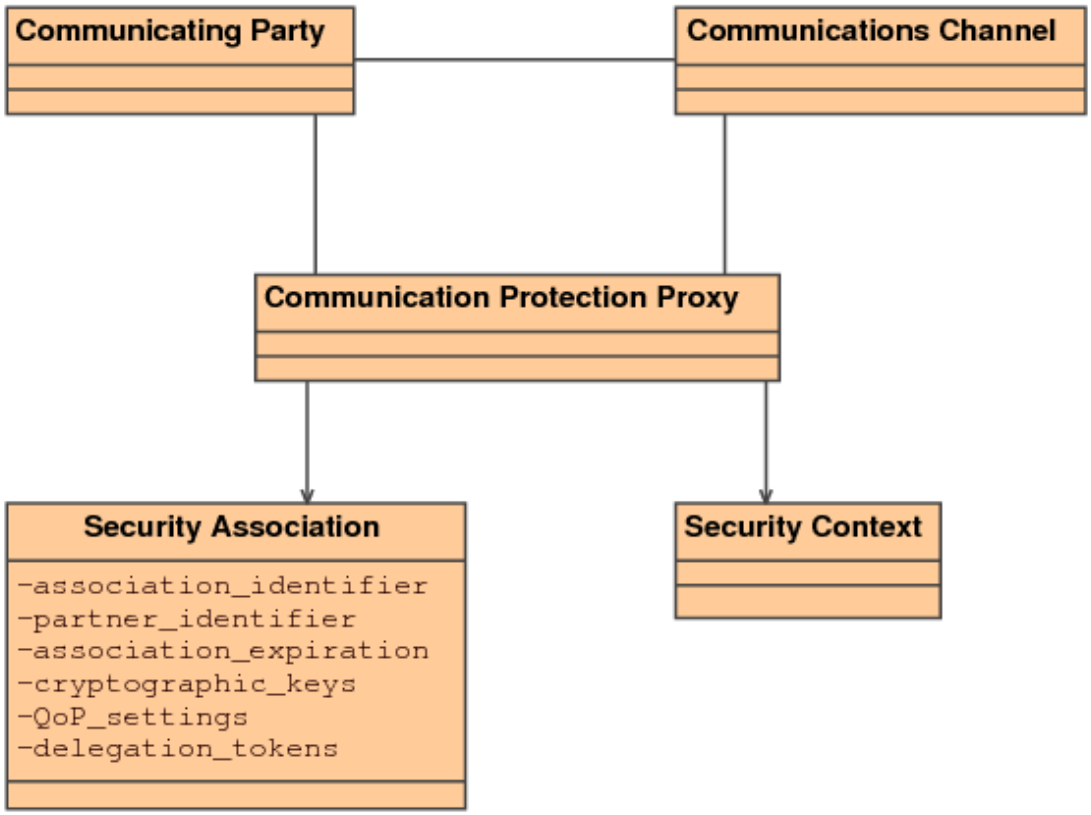


Figure 1: The structure of the Security Association

The structure of the Security Association is shown in Figure 1 . 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 Pipe pattern owning this instance of Security Association to protect messages.
- Quality of Protection (QoP)Settings Used by the Secure Pipe pattern to determine which security services need to be applied to messages.
- Delegation Tokens Used by the Secure Pipe pattern to implement delegation functionality.

Dynamics

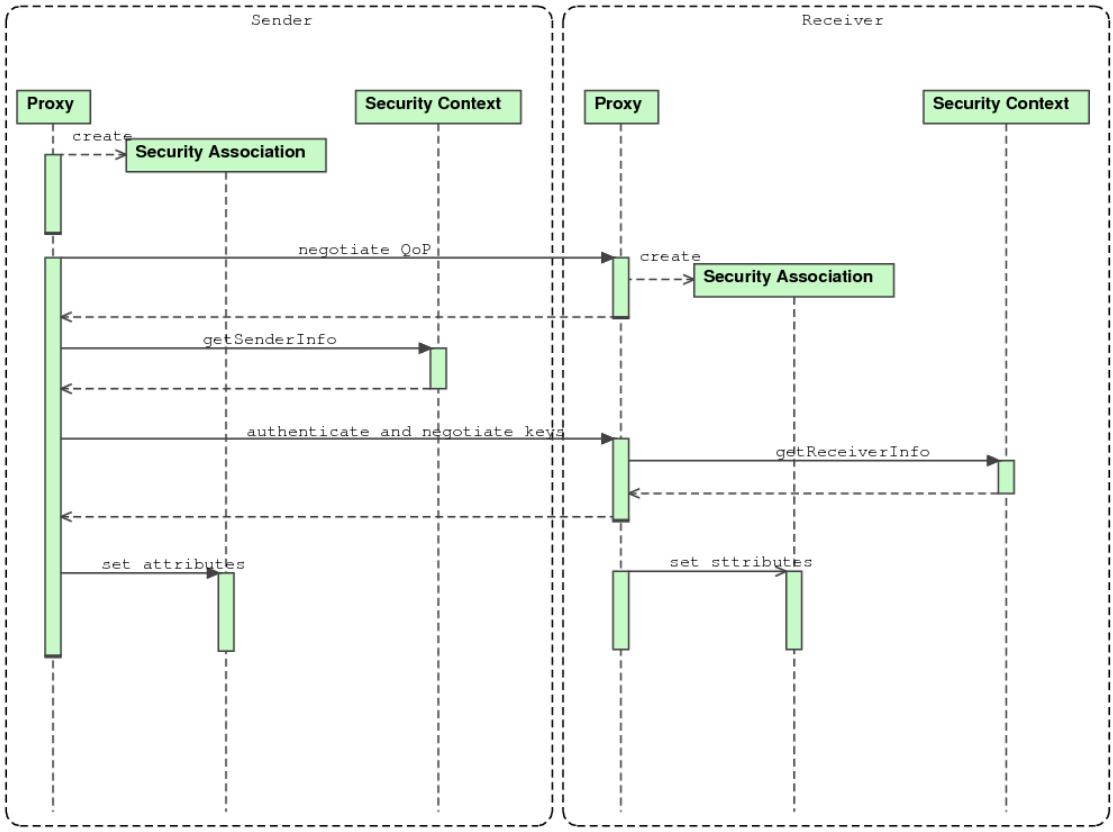


Figure 2: Event sequence for the Security Association

See Figure 2 .

Participants

The relations between the participants are shown in Figure 1 . The participants and their responsabilities are:

- 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

The interactions between the participants are shown in Figure 2 .

- 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.

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.

Pitfalls

(Nothing given)

Consequences

- Permits re-use of a single instance of Secure Pipe to protect more than one message.
- Reduces the time required to set up a Secure Pipe by eliminating the need to renegotiate 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.

Known uses

- Generalized Security Service (GSS-API); the Security Association instances are called ``Security Contexts''.
- OMG CORBASecurity; Security Association instances are called ``Security Contexts''.

Security Context

Pattern documentation

Quick info

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

Problem

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.

Forces

- 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.

Example

(Nothing given)

Solution

Structure

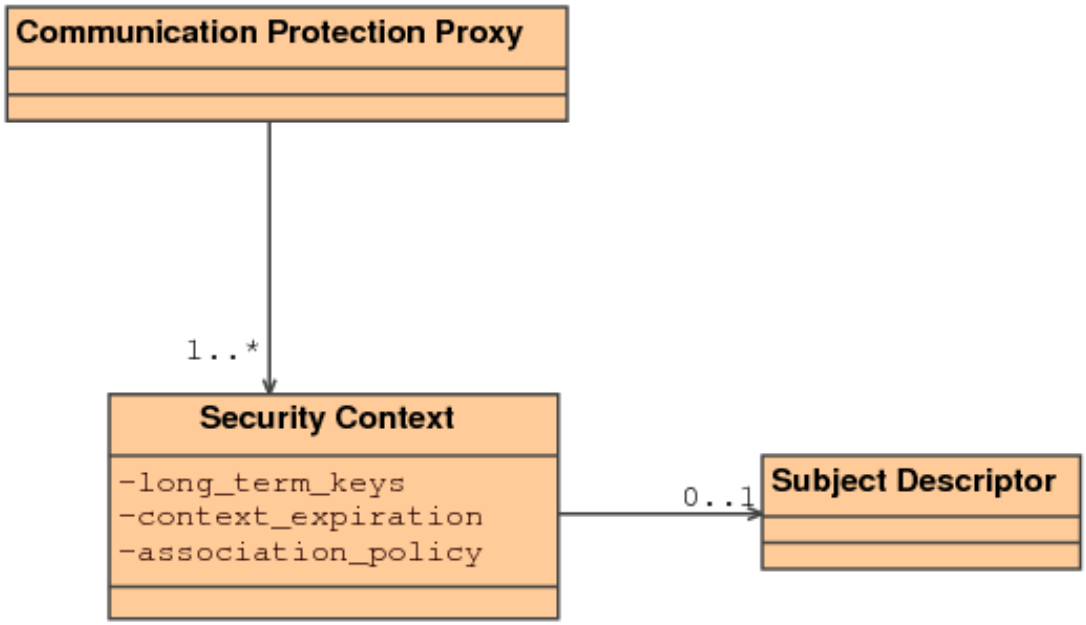
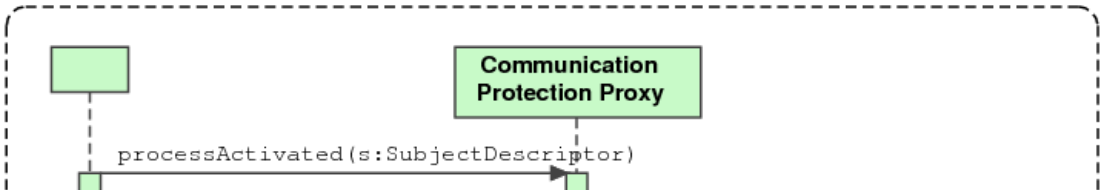


Figure 1: Class layout of the Security Context.

See Figure 1 .

Dynamics



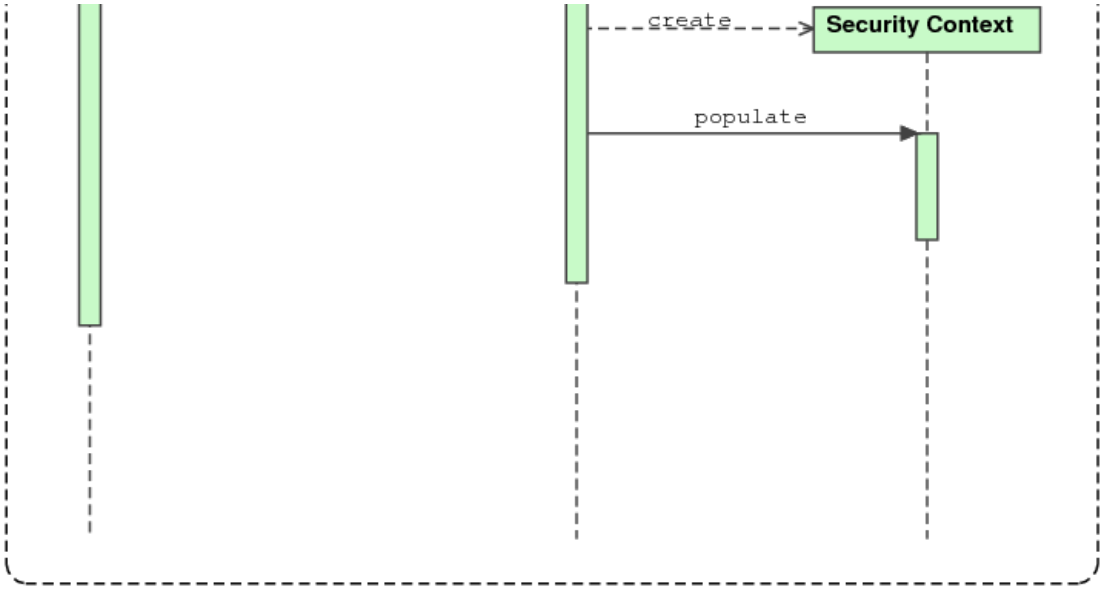


Figure 2: Event sequence for the Security Context.

See Figure 2 .

Participants

See also Figure 1 .

- 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.

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.

Pitfalls

(Nothing given)

Consequences

- 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. The encapsulation improves maintainability.
- 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.

Known uses

- 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.

- Java2Standard Edition--java. security. AccessControlContext The Java2Access 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.
- 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.
- 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 the Secure Pipe pattern)to create CORBASecurity Context objects (which are instances of our Security Association pattern).

Server Sandbox

Pattern documentation

Quick info

Intent: Many site defacements and major security breaches occur when a new vulnerability is discovered in the Web server software. Yet most Web servers run with far greater privileges than are necessary. The Server Sandbox pattern builds a wall around the Web server in order to contain the damage that could result from an undiscovered bug in the server software.

Aliases: Privilege Drop, Untrusted Server, Constrained Execution Environment, Unprivileged/Restricted User Account, Run as Nobody

Problem

A server-based application is typically exposed to a huge number of potentially malicious users. Any application that processes user input could potentially be tricked into performing actions that it was never intended to perform. For example, many Web servers contain logic errors that can be exploited to allow private files to be served over the Internet. Other servers contain undiscovered buffer overflow errors that can allow client-provided malicious code to be executed on the server.

While every attempt should be made to prevent these types of errors, it is impossible to anticipate every possible attack beforehand. Therefore, it is prudent to deploy a server application in a manner that minimizes the damage that can occur if the server is compromised by a hacker.

Web applications generally require little in the way of privileges once they are started. But by default, many servers and applications install in a manner that gives them unnecessary and dangerous privileges, that if compromised could lead to significant security breach.

For instance, Web servers running on the UNIX operating system must be started with administrative privileges in order to listen on port80---the standard HTTP port---which is a privileged port. Likewise, the Microsoft IIS default installation executes the Web server using the privileged SYSTEM user. If a Web server running with administrative privileges is compromised, an attacker will have complete access to the entire system. This is widely considered the single greatest threat to Web site security.

Forces

- A process needs enough privileges to perform its function correctly.
- The more privileges a process has, the more privileges a compromised process has.
- Implementing fine-grained least-privilege policies in real systems can be cumbersome.

Example

(Nothing given)

Solution

The Server Sandbox pattern strictly limits the privileges that Web application components possess at run time. This is most often accomplished by creating a user account that is to be used only by the server. Operating system access control mechanisms are then used to limit the privileges of that account to those that are needed to execute, but not administer or otherwise alter, the server.

This approach accommodates systems that require administrative privileges to start the application, but do not need those privileges during normal operation. The most common example of this is a UNIX server application that must listen on a privileged port. The application can start with additional privileges, but once those privileges are no longer needed, it executes a privilege drop, from which it cannot return, into the less privileged operating mode.

There are a number of different operating system specific privilege drop mechanisms. Some of the more common are:

- An application can switch the user account under which it is executing at run-time. For example, a UNIX application can switch from running with administrator privileges to a specific server account or even the nobody account.
- An application can inform the operating system that it wishes to drop certain privileges dynamically. This is common in capability-based systems, where the operating system dynamically maintains a list of application capabilities. In Linux, an application can ask the operating system to make entire APIs invisible for the remainder of the lifetime of that process.
- An application can instruct the operating system to no longer accept any changes that it requests. For example, once a Linux system has fully booted, it can instruct the operating system to no longer allow kernel modules to be dynamically loaded, even by the administrative account.
- An application can be executed within a virtualized file system. The UNIX chroot option allows the application to think it can see the actual file system, when in fact it only sees a small branch set aside for that application. Any changes to the

system files it sees will not affect the actual system files.The Server Sandbox pattern also requires that the remainder of the system hosting the server be hardened. Many operating systems allow all user accounts to access certain global resources. A server sandbox should remove any global privileges that are not essential and replace them specific user and group privileges. A compromised Web server will allow an external hacker to gain access to all global resources. Eliminating the global privileges will ensure that the hacker will not have access to useful (and potentially vulnerable)utilities and operating system features.

The Server Sandbox pattern partitions the privileges required by the server between those needed at server startup and those needed during normal operation. For example, UNIX systems require administrative privileges to create a server listening on port80, the standard HTTP port. However, the server should not possess administrative privileges at run-time. A server sandbox allows dangerous privileges to be used to create the server but then revoked before the server is exposed to client input.

While the most common implementation of the Server Sandbox pattern relies on a restricted user account, other (additional)implementations are possible, including:

- Creating a virtual file system and restricting the server so that it cannot see files outside of this space (chroot).
- Putting wrappers around dangerous components that limit the application's ability to access resources and operating system APIs
- Using operating system network filtering to prevent the server from initiating connections to other machines

Structure

See Figure 1 .

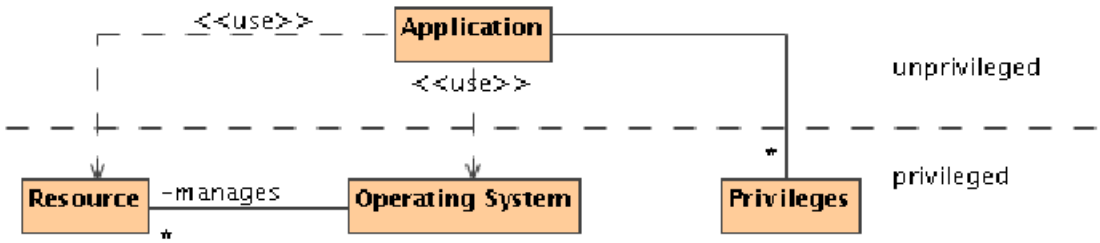


Figure 1: Server sandbox structure.

Dynamics

See Figure 2 .

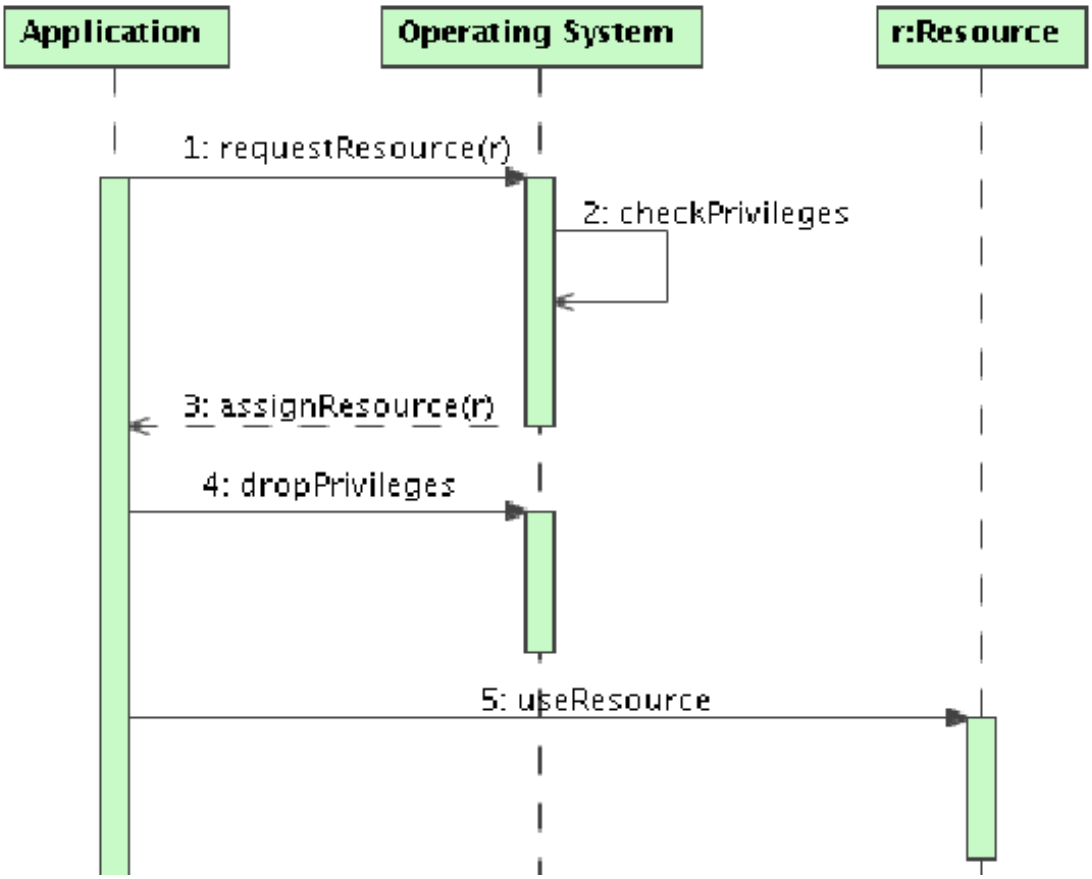


Figure 2: Server sandbox dynamics.

Participants

- The **Application** needs a number of **Resources** to function correctly.
- Each of the **Resources** are managed by the **Operating System** . **Applications**
- also have associated **Privileges** , that allow them to request certain **Resources** from the **Operating System** .

Collaborations

- The Application requests a Resource from the Operating System.
- The Operating System checks if the Application is privileged to use this Resource.
- The Operating System assigns the Resource to the Application.
- The Application then drops its extra privileges. As explained in the solution above, this can happen in multiple ways. We assume here that the Application is able to request of the Operating System to drop its privileged state.
- Next, the application uses the assigned Resource and continues operating in an unprivileged state.

Implementation

(Nothing given)

Pitfalls

It is critical that the application be developed within the envisioned constrained environment. Attempting to add the constrained environment after the fact generally breaks the application and often results in the constrained environment being unnecessarily relaxed in order to resolve the problem. For example, most IIS applications are developed using the standard, insecure configuration, in which IIS executes as SYSTEM. If an individual administrator attempts to configure his or her server more securely and run IIS using a less privileged account, many of these applications will fail to execute properly.

Building the application within the constrained environment also ensures that any performance or resource usage impact will be uncovered early in development.

It is important to document the security configuration in which the system is expected to execute. If the application requires specific privileges to specific files and services, this information must be provided to the administrator configuring the system. It is not sufficient to merely provide an installation script that sets all the appropriate options, because many administrators need to finetune the installation afterwards or install other applications that may alter the security configuration of the system. If the administrator is not aware of the minimum required privileges, he or she may give the application unneeded---and potentially dangerous---privileges. This often translates to executing the application with full administrative privilege.

Many operating systems install in an insecure state. Employ general hardening techniques to eliminate weaknesses. On many systems, the Operating System access control model can be bypassed. If an outsider is able gain control over a general user account, it can be fairly straightforward to exploit a weakness in a system application to gain root/administrator privileges. If possible, the restricted user account should be limited to executing only those programs that it requires.

There are a number of possible attacks that could be perpetrated against this pattern:

- **buffer overflow attacks** buffer overflow attacks on the server are the most common approach to remote compromise of the server. The sandbox is intended to contain the damage of such an attack.
- **privilege escalation** ---if an attacker is able to compromise a Web server, even one running as nobody, they will be able to execute code on the system. Attackers typically attempt to break out of the sandbox by exploiting vulnerabilities in other privileged applications, such as sendmail. If a vulnerable, privileged application is accessible to the restricted user account, a privilege escalation attack is possible.
- **breaking out of the sandbox** ---if the sandbox mechanism contains bugs, an attacker may be able to exploit them to break out of the sandbox. If the attacker can somehow gain root privilege, many sandbox features (such as chroot)are reversible.
- **snooping** ---if an attacker is able to exploit a server vulnerability and gain a toehold on the system, they may have enough privilege to monitor further server operations. They could capture passwords or other sensitive data. If the server has privileges to access a back-end database, the attacker will have those same privileges.
- **application level exploits** ---even if the server is perfectly sandboxed, it may still suffer from application-level vulnerabilities. The remote attacker may not have to compromise the server in order to misuse its services.

Consequences

- Manageability: This pattern will affect the manageability of the software in question because constrained execution environments often incur overhead to setup and maintain.
- Performance: This pattern will often have a negative effect on performance, but this will depend on the specific techniques

used. Using chroot or unprivileged user accounts do not affect performance. Other techniques that impose additional runtime validity checks will incur a performance penalty.
- Cost: This pattern will increase development costs somewhat. This can be minimized if the application is developed with the constraints already in place. Retrofitting an existing application is much more difficult.

Known uses

At the code level, Java provides the most widely known implementation of a sandbox. It prevents the user from using features and functions that are outside of the Java security policy.

At the system level, the canonical example of this pattern is the Apache Web server, which by default runs as user nobody. Although root privileges are required to start the server on port80, the server drops into the nobody account after initialization. The nobody account is able to read (but not write)all of the public html files on the server. But a well-configured server will disallow the nobody account from executing any commands or reading any other files.

Similarly, the Netscape Enterprise Server (iPlanet Web server)for UNIX uses the nobody account. If it is instructed to listen on a privileged (<1024)port, it must be started as root. However, once the port is established, it switches to the nobody account before accepting client connections.

At the network level, it is common practice to place a Web server outside the corporate firewall, or in a Demilitarized Zone (DMZ)between the Internet and the internal network. In either case, a firewall separates the Web server from the rest of the internal network. This is an example of a network-level server sandbox: the Web server is only allowed to connect to a handful of specific ports on one or more specific trusted machines on the internal network. In some configurations, the connections must be initiated from the internal network---in this case, the DMZ represents a sandbox in the purest sense.

Session Failover

Pattern documentation

Quick info

Intent: Avoid inconveniencing users that lose session data in a system restart.

Problem

While using a system with Sessions and Keep Session Data in the Server, all requests from a user within a session needs to be handled by the server instance that holds the session data. If the server becomes unavailable, for instance due to a crash or scheduled maintenance, the user will lose all session data and have to start over. How can we avoid inconveniencing users in this way?Can we avoid it without making the system overly complicated?

Forces

- Users don't want interruptions in their use of the system.
- Few if any systems are free from needing to be taken off-line for maintenance and upgrades to hardware and software once in a while during their lifetime.
- Connectivity between users and system can be lost with little or no warning due to network problems, hardware failure and software crashes and other problems outside of the control of the system or the user.
- For most systems, it's straightforward to start several instances of the system on one or several computers. However they will run independent of each other, sharing no data, neither system state nor session data.
- Session data from a users session is kept on one instance of the server, so no matter how many server instances are running, only one of them can handle the requests from the user. Should that server instance become unavailable to the user, the session and its associated session data becomes unavailable too, and the user will have to start over.

Example

(Nothing given)

Solution

Add the requirement to Sessions that all session related data must be capable of being made persistent, so it can be transmitted over the network between server instances.

Keep several instances of the system running at all times, preferable on more than one computer and in more than one geographical location. Assign each session to one server instance that will handle all requests on that session. Keep a copy of all session specific data on another instance of the system. When assigning servers to keep copies of Session specific data, keep an eye to minimizing the likelihood that both the server keeping the copy and the server actively servicing the user will be affected by the same network problem.

When a server instance breaks down or becomes inaccessible, let all further events related to its sessions be handled by the system instance that keeps copies of the session scooped data.

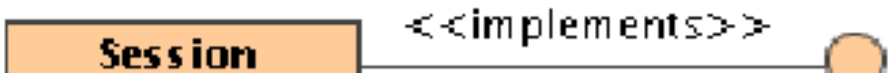
To handle the situation where a sizable portion of the server instances becomes unavailable, you can keep more servers online than are needed for handling the load. The extra servers will hold copies of session data, but won't handle requests, until a server handling a session they are holding a copy for becomes unavailable. The smallest possible installation is two server instances running on the same computer, one handling requests, and the other holding copies of all session data. A large scale installation will have servers parked in several cities in different countries and on different continents.

Make sure that in case of server failure, session related transactions on other systems are automatically rolled back. Make sure that when the session is moved to a new server and some roll back back has taken place, the user is somehow made aware of this, and given the opportunity to re-do the steps needed to resubmit it. If the user is another program, this can be done in the form of throwing an exception.

Use a Load Balancer to control when to move a Session, and to direct accesses to a Session to the server instance currently being responsible for it.

Structure

The structure of a session failover is included in Figure 1 .



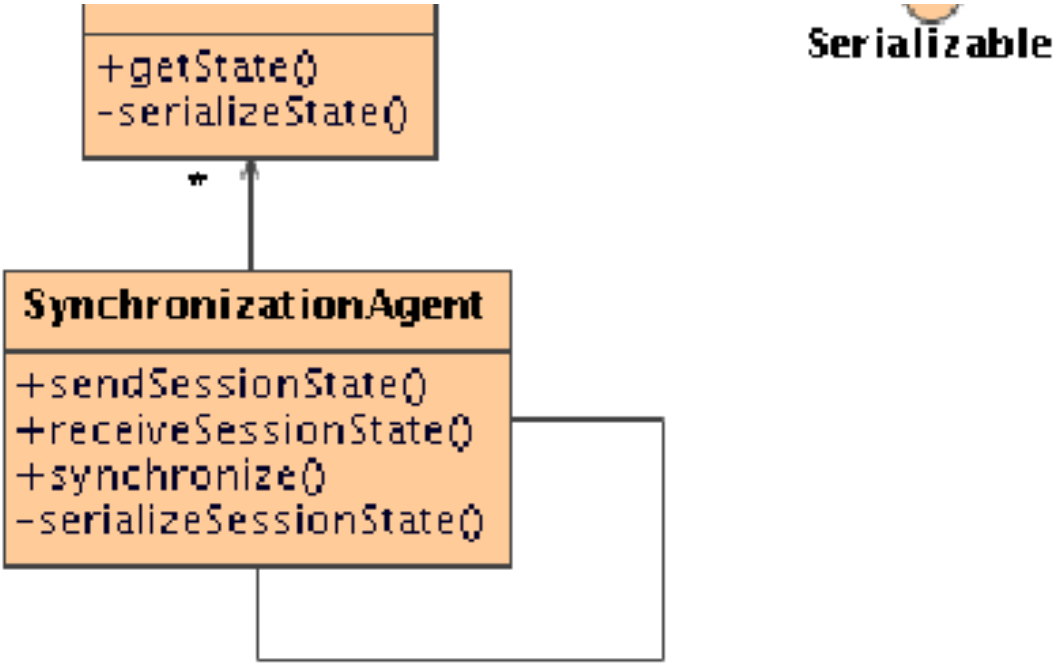


Figure 1: Session Failover structure.

Dynamics

The dynamics of a session failover are depicted in Figure 2 .

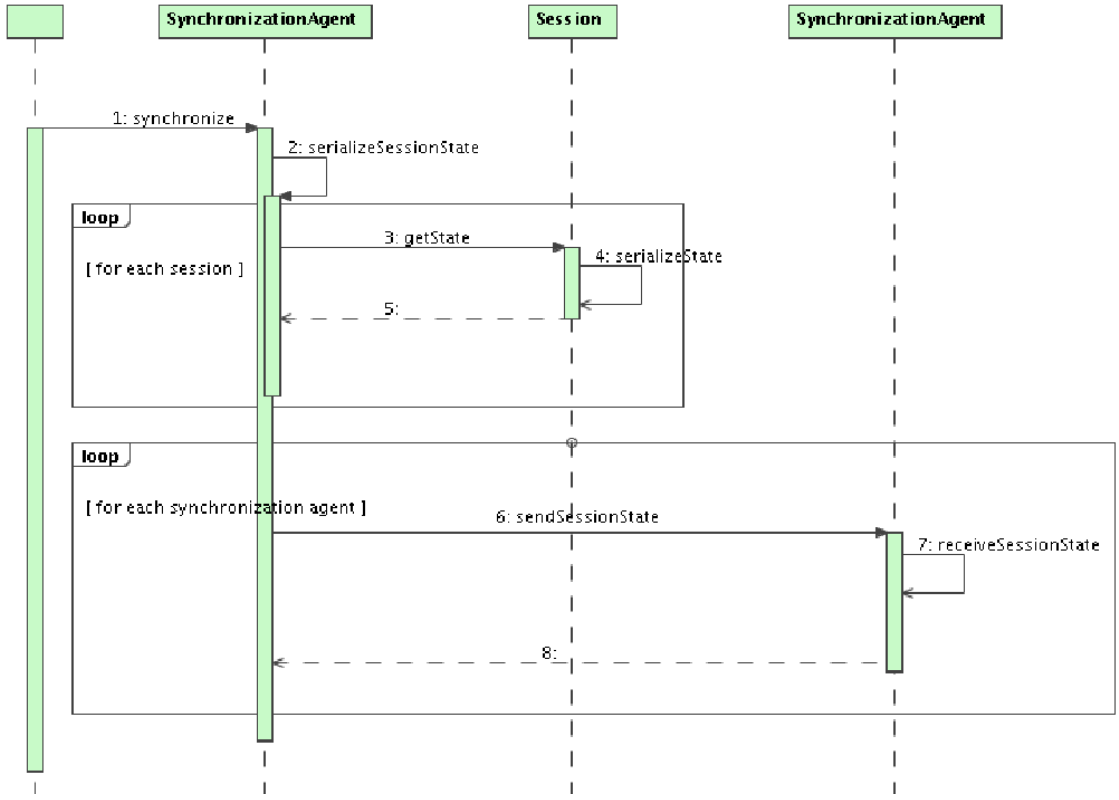


Figure 2: Session Failover dynamics.

Participants

Session

- implements the session pattern. In order for the state of the session to be transmitted to other systems, this session must be serializable. **SynchronizationAgent**
- is the workhorse of the session failover, in that it is responsible for collecting the serialized state of all sessions and exchanging it with neighbouring SynchronizationAgents.

Collaborations

- When the SynchronizationAgent receives a message to synchronize its state (this message could be sent periodically, or because of some internal session state change), it serializes the session states by requesting the serialized state from each session it is aware of.
- After the SynchronizationAgent has received all session states, it iterates over all other SynchronizationAgents and sends the collected serialized session states to them.
- The other SynchronizationAgents receive the new session states, and update their internal states accordingly.

Implementation

(Nothing given)

Pitfalls

(Nothing given)

Consequences

Benefits:

- The likelihood that users will see uninterrupted service from the system is increased.
- Scheduling maintenance on servers becomes far less stress-full, because the impact on users is diminishes or even totally removed. As a fringe benefit this can lead to better maintained servers that runs more stable than had Session Failover not been implemented.Liabilities:
- The session data replication functionality is an additional piece of nontrivial code that has to be written, tested and maintained.
- At run time the on-going copying of session data between server instances increases the workload and memory requirement of the servers, which leads to increased costs. If server instances are placed in separate geographical locations, e. g. in two different cities, to guard against network failure, the constant copying of session data between cities takes up bandwidth on either Internet or leased lines. In both cases this comes at a price.
- Handling the case where a session is moved to another server while it was in the middle of a transaction to another system, is at best tricky and in many cases impossible to implement correct. It might be a better choice to drop the session, rather than trying and failing at recovering it on another server instance.
- The requirements for a Load Balancer to handle a system with Session Failover are the same as for a system with Keep Session Data in the Server, and then some. It needs to also be capable of handling the situation where requests within a session must be sent to a new server, after Session Failover has kicked in.

Known uses

Application servers: The ATG Dynamo and the BEA WebLogic J2EE application servers and some but not all of their competitors implements Session Failover.

At one point Scandinavian Online ran on four servers, one in Copenhagen, one in Oslo, one in Stockholm and one in Helsinki. Each server handled the requests originating from the country it was situated within, as well as being fail over server for one of the other servers.

Session

Pattern documentation

Quick info

Intent: Many objects need access to shared values, but the values are not unique throughout the system.

Aliases: User's Environment, Namespace, Threaded-based Singleton, Localized Globals

Problem

Military personnel's activities are tracked while they are in a high-security military installation. Their entry and exit are logged. Their badges must be worn at all times to show they are only where they are supposed to be. Guards inside of the base can assume personnel with a badge have been checked thoroughly at the base entrance. Therefore they only have to perform minimal checks before allowing them into a restricted area. Many people are working in a base at the same time. Each security badge uniquely identifies who that person is and what they can do. It also tracks what the carrier of the badge has been doing.

Secure applications need to keep track of global information used throughout the application such as username, roles, and their respective privileges. When an application needs to keep one copy of some information around, it often uses the Singleton pattern. The Singleton is usually stored in a single global location, such as a class variable. Unfortunately, a Singleton can be difficult to use when an application is multi-threaded, multi-user, or distributed. In these situations, each thread or each distributed process can be viewed as an independent application, each needing its own private Singleton. But when the applications share a common global address space, the single global Singleton cannot be shared. A mechanism is needed to allow multiple ``Singletons'', one for each application.

Forces

- Referencing global variables can keep code clean and straightforward.
- Each object may only need access to some of the shared values.
- Values that are shared could change over time.
- Multiple applications that run simultaneously might not share the same values.
- Passing many shared objects throughout the application make APIs more complicated.
- While an object may not need certain values, it may later change to need those values.

Example

A Session can be used to store many different kinds of information in addition to security data. The Caterpillar/NCSA Financial Model Framework has a FMState class (See http://www. uiuc. edu/ph/www/j-yoder/financial_framework). An FMState object serves as a Session. It provides a single location for application components to access a Limited View of the data, the current products that can be selected, the user's Role, and the state of the system. Most of the classes in the Financial Model keep a reference to an FMState. A true Singleton could not be used because a user can open multiple sessions with different selection criteria, each yielding a different Limited View. Figure 1 shows FMState from the Financial Model. Security info includes username and role. The security info and selection criteria define the limited views. Each ReportView and ReportModel has a reference back to the FMState so it can access other data.

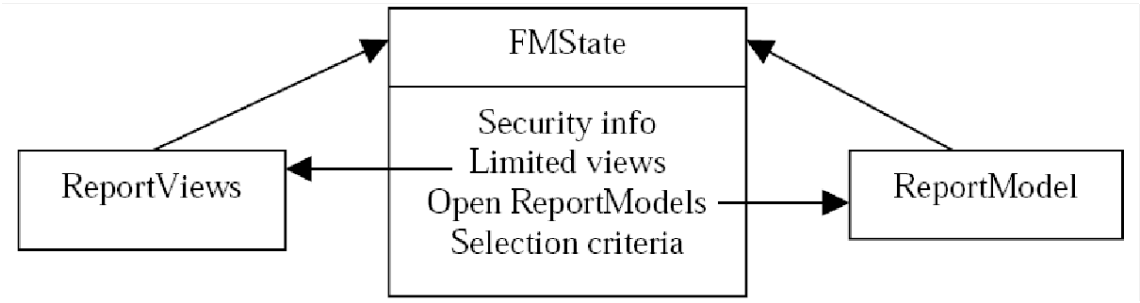


Figure 1: FMState, a Session example.

Solution

Create a Session object, which holds all of the variables that need to be shared by many objects. Each Session object defines a namespace, and each variable in a single Session shares the same namespace. The Session object is passed around to objects which need any of its values. Certain user information is used throughout a system. Some of this information is security related, such as the user's role and privileges. A Session object is a good way for sharing this global information. This object

can be passed around and used as needed.

Depending on the structure of the class hierarchy, an instance variable for the Session could be added to a superclass common to every class that needs the Session. Many times, especially when extending and building on existing frameworks, the common superclass approach will not work, unless of course you want to extend object which is usually not considered a good design. Thus, usually an instance variable needs to be added to every class that needs access to the Session.

All of the objects that share the same Session have a common scope. This scope is like the environments used by a compiler to perform variable lookups. The principle differences are that the Session's scope was created by the application and that lookups are performed at runtime by the application.

Since many objects hold a reference to the Session, it is a great place to put the current Stateof the application. The State pattern does not have to be implemented inside of the Session for general security purposes, however. Limited View data and Roles can also be cached in a Session. It is important to note that the user should not be allowed to access any security data that may be held within a Session such as passwords and privileges. It can be a good idea to structure any application with a Session object. This object holds onto any shared information that is needed while a user is interacting with the application.

Structure

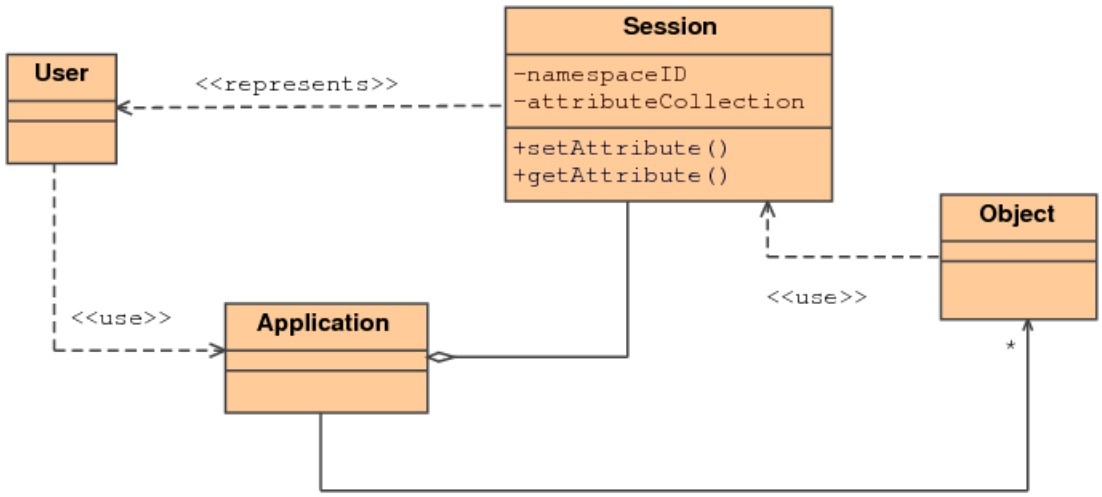
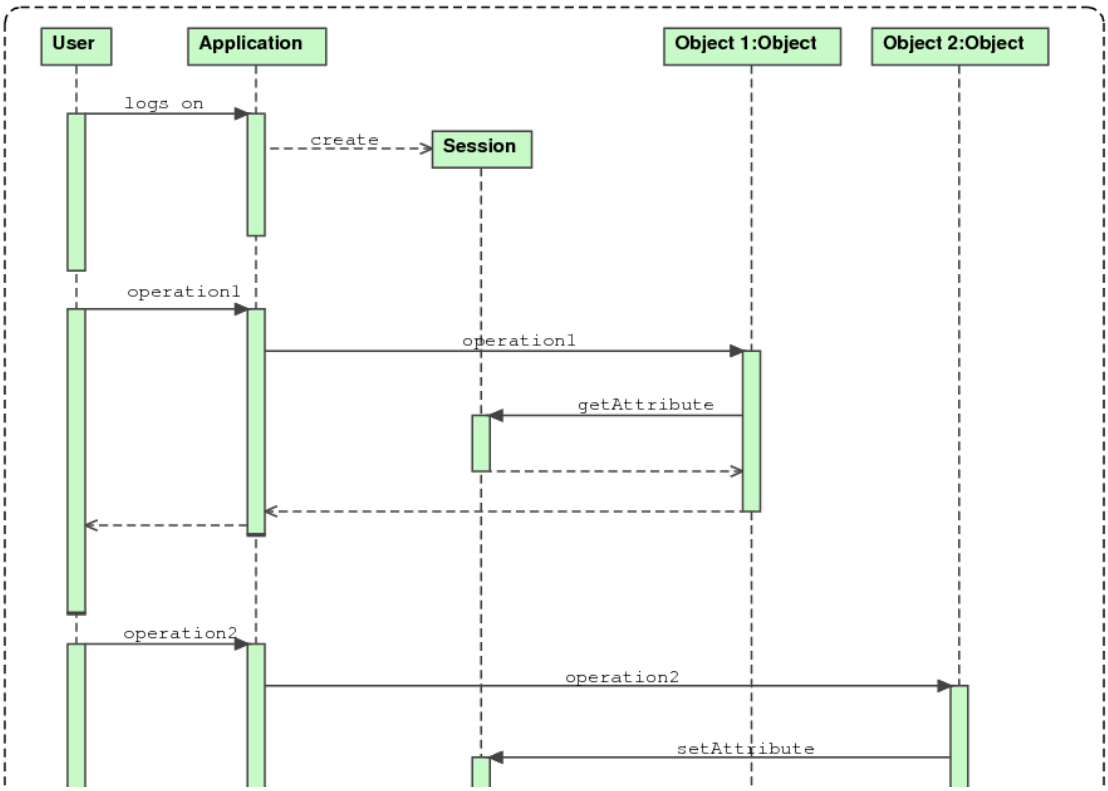


Figure 2: Session structure.

See Figure 2 .

Dynamics



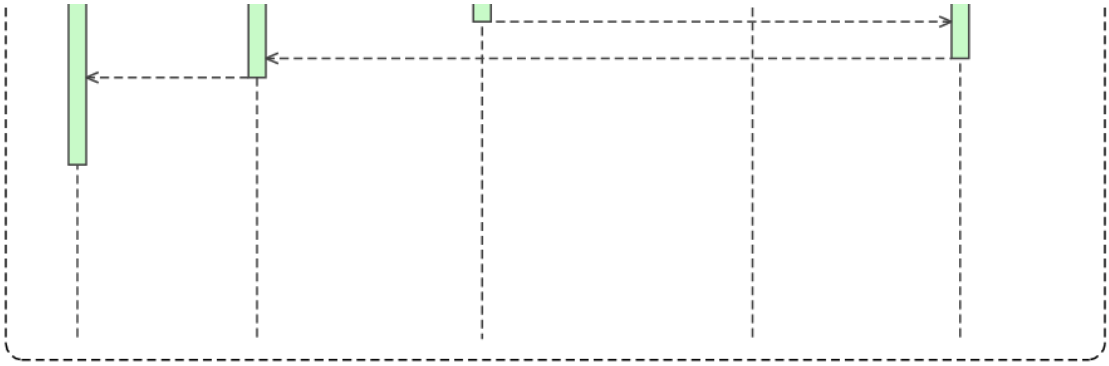


Figure 3: Session dynamics.

See Figure 3 .

Participants

- User The user of the application. A session will be a representation of the relevant attributes of or related to this user.
- Application The application which the user uses. The application defines the context of the session, and can deal with the sessions of multiple, concurrent users.
- Session The session object encapsulates the attributes of or related to a user. A session typically has a unique ID within its context.
- Objects The objects are objects executing operations while the user uses the application. The objects may need to retrieve or store information in the session object.

Collaborations

- When a user starts using an application, a session object for this user is created.
- For each subsequent request of the user, the session object is passed along to the object (s)which will handle the request.
- An object can then use or update the information stored in the session object.

Implementation

Two strategies for storing the session data are possible:

- Keep Session Data in Server. Keep all session specific data on the server. Assign a unique token to each session, and create the protocols used in communication between users and system so that this token is made part of every interaction. Use this token as a key into the data structure in the server that holds the session specific data for all clients.

Keeping all data on the server and making sure it will never leave the server, means you have no need to write elaborate error checking code to validate data every time it reenters the system from the client. It also frees you from implementing code that converts from the form the data is stored in while in the server (eg. hierarchies of objects)to a form that can be transmitted over the wire between client and server.

Equip the system with mechanisms to validate that the token a user identifies himself with during an access belongs to the user performing the access. Add mechanisms commensurate with the threat and value of identity theft in the system.

Consider the following when choosing this strategy.

- Delay and limited bandwidth between client and server can make it impractical to transfer session specific data from client to server as part of each request.
- If session data is to be provided with each request, it has to be checked for errors that can make the server code fail. If the data is stored in the server this check can be omitted.
- The client might not have enough memory to store the data for its session, even if the amount of data is small. A good example is a tiny embedded system. On the other hand the client might have ample memory (and swap space)to store its session specific data. A common example is a desktop PC running a webbrowser.
- If session specific data is to be kept on the server, a system for mapping incoming requests to the correct bundle of session specific data has to be in place. If looking at other clients session specific data is of value to malicious entities, this mapping system must be capable of withstanding identity spoofing attacks.
- Keeping session data with the client and transmitting it to the server as part of requests can be inconvenient because of the communication protocol being used. An example is data hidden in the html for a web page. The data must be formatted in a certain way that's cumbersome and not very space efficient. The data has to be duplicated for each and every link on the page that must supply session specific data to the server. This leads to slow page loads.

- Keep Session Data in Client. Keep the session specific data in the client. Transfer all or the necessary subset of it to the server along with each request. Logging out is as simple as no longer contacting the server, because the server is not keeping track of clients. If the client crashes before logging out, no session data is left orphaned anywhere, because it went away with

the crashed client.

Consider the following when choosing this strategy.

- The server is too simple to be able to hold on to session data in between requests, or it has too little memory to do so for all concurrent users. An example of the first case is cgi scripts on a webserver.
- The server will be restarted, taken offline or suffer crashes during runtime. Session data will be lost when this happens. Enhancing the server with the ability to let session data live on across system restarts is not feasible for practical or economical reasons.
- The server is really several servers behind a load balancer that directs requests to the least loaded server, so the requests from a client can go to different servers at each request.
- The client making requests on the server must be reliably identified if the request is to be mapped to client specific data in the server. This can very well be hard, unreliable or even impossible, for instance because of the the protocol used for communication (eg. HTTP).

Pitfalls

If the session specific data is kept in the client, the client can modify this data. Make sure that no critical data can be modified without detection.

Consequences

A Session offers the following advantages:

- The Session object provides a common interface for all components to access important variables.
- Instead of passing many values around the application separately, a single Session object can be passed around.
- Whenever a new shared variable or object is needed, it can be put in the Session object and then all components that have access to the object will have access to it.
- Change propagation is simplified because each object in a thread or process is dependent on only a single, shared Session object.A Session offers the following disadvantages:
- While an object may not need a Session, it may later create an object that needs the Session. When this is the case, the first object must still keep a reference to the Session so it can pass it to the new object. Sometimes, it may seem as if every object has a Session. The proliferation of Session instance variables throughout the design is an unfortunate, but necessary, consequence of the Session pattern.
- Adding Session late in the development process can be difficult. Every reference to a Singleton must be changed. The authors have experience retrofitting Session in place of Singleton and can attest that this can very tedious when Singletons are spread among several classes. This is also true when trying to consolidate many global variables that were being passed around as parameters into a Session.
- When many values are stored in the Session, it will need some organizational structure. While some organization may make it possible to breakdown a Session to reduce coupling, splitting the session requires a detailed analysis of which components need which subsets of values.

Known uses

- For VisualWorks, the Lens framework for Oracle and GemBuilder for GemStone have OracleSession and GbsSession classes respectively. Each keeps information such as the transaction state and the database connection. The Sessions are then referenced by any object within the same database context.
- The Caterpillar/NCSA Financial Model Framework has a FMState class. An FMState object serves as a Session, while keeping a Limited View of the data, the current product/family selection, and the state of the system. Most of the classes in the Financial Model keep a reference to an FMState.
- The PLoP'98registration program has a Session object that keeps track of the user's global information as they are accessing the application.
- Most databases use a Session for keeping track of user information.
- VisualWave has a Session for its httpd services, which keeps track of any web requests made to it.
- UNIX ftp and telnet services use a Session for keeping track of requests and restricting user actions.

Session Timeout

Pattern documentation

Quick info

Intent: Prevent the system from running out of resources because abandoned sessions are not cleaned up.

Problem

You have a system with Sessions where users might abandon their sessions without the system being notified.

After a session has been created, its user will access it zero or more times. At the time of each access the session, and the data stored in the related session scope must be available. However the user might not signal to the system, when he no longer intends to access the session (log out), since the system can be long lived, data related to sessions abandoned in this way, can slowly fill up all available memory on the system.

Forces

- Each session in the server takes up some memory and/or permanent storage.
- The system doesn't have an infinite amount of memory and permanent storage in which to store session specific data.
- Session data must be available when a request is made within its session.
- Some sessions will see only infrequent requests. An example is an e-commerce system that polls an inventory tracking system for updates to the inventory once an hour, within a session in the inventory tracking system.
- For some sessions frequent use is the norm, so such a session that has not been used for significantly longer than the typical delay between uses, can be considered abandoned with a high probability.
- Looking at the data in another user's session specific data, or even performing requests on the server using another user's session, can be of some value to malicious entities. If the session being ``hijacked''is abandoned, the likelihood of being caught in the act is lowered.

Example

Non software example. At a cafeteria where you pay when you order and therefore can leave without notifying a waiter, there is one or more persons whose job it is to periodically go from table to table and clean up leftover service and food from tables whose occupants appears to have left.

If the clean up work is not getting performed fast enough, the cafeteria will run out of free and clean tables, which will turn customers away. If too much cleaning staff is assigned, they will often sit idle, which is a waste of the cafeterias money.

Solution

Make a session manager, that runs in the system as a long lived process. Periodically it will iterate over all sessions in the system looking for sessions that has not been accessed for more than a set amount of time. When such a session is found, it's immediately flagged as invalid, so it won't be used any more. Asynchronously another process can delete its session specific data. This can be implemented both for sessions kept entirely in memory, as well as for session stored on permanent media.

A pre-requirement for this solution is that a well defined method for invalidating a session and for deleting its data is implemented in all sessions. These methods can also be used when the client explicitly logs out.

If a user tries to access a session that has been flagged as invalid, or is not known to the system, the user can be informed that his session has expired and all related data removed. The user can be given easy means to get a new session created. In many cases it will be practical to automatically create a new session for the user, and inform him that this has been done, by means of a status flag for machine users, and by means of a textual message in a dialog box for human users.

In the common application servers the time out value is the same for all sessions. However it would be straigtforward to add a property to the session class, so the time out could be set individually for each session.

Structure

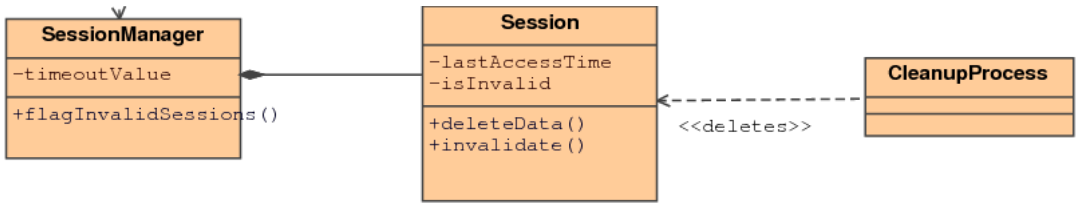


Figure 1: Class layout of the Session Timeout pattern

See Figure 1 .

Dynamics

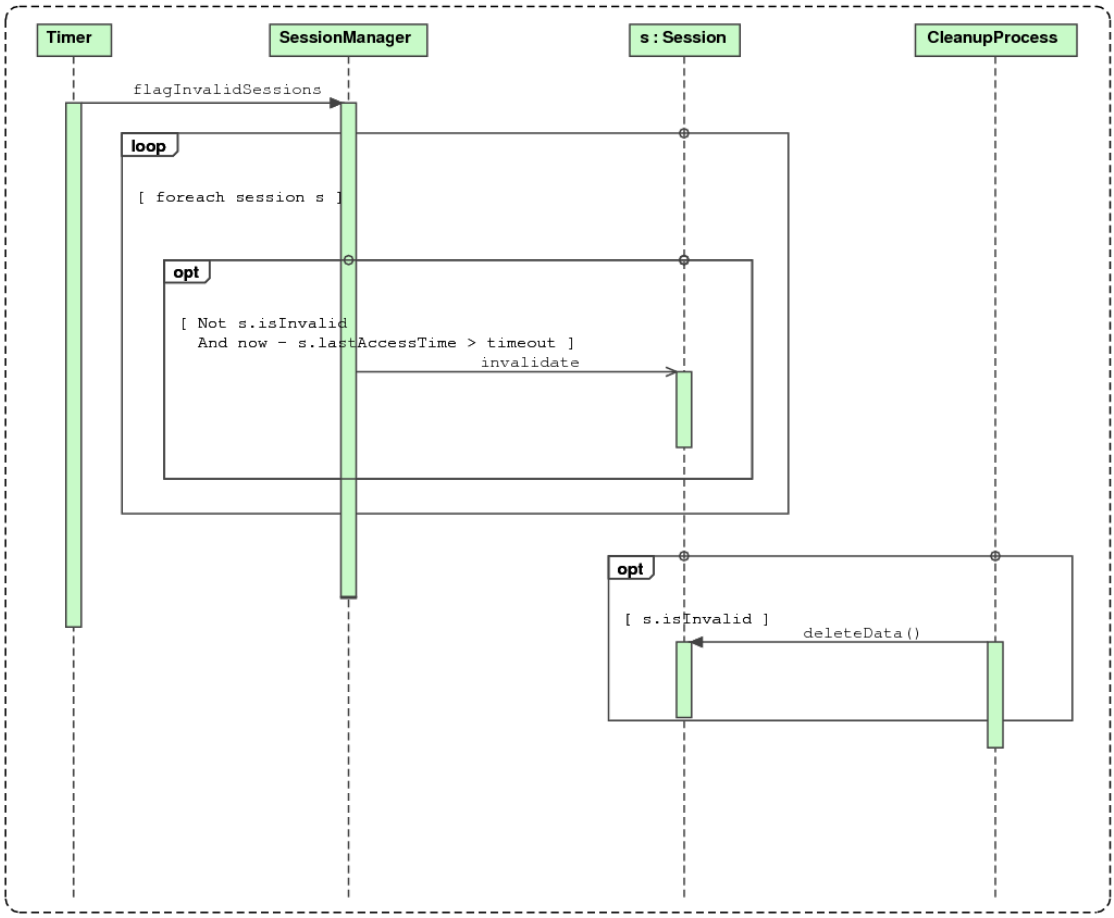


Figure 2: Event sequence for the Session Timeout pattern

See Figure 2 .

Participants

- SessionManager The session manager holds a reference to every session in the system.
- Session The session represents a user session. It has the possibility to be invalidated, and can delete its data when needed.
- Timer The timer periodically calls the operation to invalidate inactive sessions.
- CleanupProcess The cleanup process deletes invalidated sessions.

Collaborations

- The Timer triggers the SessionManager's flagInvalidSessions method.
- The SessionManager iterates over all sessions, comparing the session's last access time to the current time.
- If the difference is larger than the session timeout. value, it invalidates the session.
- The CleanupProcess looks for invalidated sessions, and deletes their data.

Implementation

(Nothing given)

Pitfalls

(Nothing given)

Consequences

- Automatically expiring a user's session might lead to confusion or frustration at the end of the user. This decreases the usability.
- On the other hand, automatically closing a user's session improves the the user's privacy. Imagine a user performing an operation using a public computer. If this user herself does not log out and the session would not expire, other people can continue using the original user's session. This might compromise the user's privacy.
- A session timeout is simple to implement, and thus does not lead to high implementation costs. Moreover, by using the pattern other costs (for example for memory or data storage)might be avoided or at least limited.

Known uses

Session Timeout is a standard feature of J2EE application servers servers such as ATG Dynamo, Tomcat and BEA WebLogic J2EE.Unix has automatic session expiration capability, not on process level but with regard to user logins. It is mostly activated for the users of systems like public terminals at universities and the like where users might get up and leave, forgetting to log out, and in high security environments where the cost and risk of an abandoned login session being misused are too high to ignore. Unix will expire the login session after a relatively short period of time, such as15minutes. In UNIX the process does the job of Sessions and much more. Code running within a process can not allocate memory itself, it has to obtain it from the system. This is often done by calling the malloc library call, which will obtain memory from the UNIX kernel. Memory is freed up in a similar fashion, calling free, which gives the memory back to the UNIX kernel. This way all memory allocations and deallocations are tracked by the OS, so whenever a process is terminated, all memory that's allocated to code running within it, can be freed.

Single Access Point

Pattern documentation

Quick info

Intent: Reduce the ``attack surface''by imposing a single access point on the system, providing an ideal place to do access control and policy enforcement.

Aliases: Login Window, One Way In, Guard Door, Validation Screen

Problem

A military base provides a prime example of a secure location. Military personnel must be allowed in while spies, saboteurs, and reporters must be kept out. If the base has many entrances, it will be much more difficult and expensive to guard each of them. Security is easier to guarantee when everyone must pass through a single guard station. It is hard to provide security for an application that communicates with networking, operating systems, databases, and other infrastructure systems. The application will need a way to log a user into the system, to set up what the user can and can not do, and to integrate with other security modules from systems that it will be interacting with. Sometimes a user may need to be authenticated on several systems. Additionally, some of the user-supplied information may need to be kept for later processing. Single Access Point solves this by providing a secure place to validate users and collect global information needed about users who need to start using an application.

A security model is difficult to validate when it has multiple ``front doors,"``back doors,"and ``side doors"for entering the application.

Forces

Having multiple ways to open an application makes it easier for it to be used in different environments.

- An application may be a composite of several applications that all need to be secure.
- Different login windows or procedures could have duplicate code.
- A single entry point may need to collect all of the user information that is needed for the entire application.
- Multiple entry points to an application can be customized to collect only the information needed at that entry point. This way, a user does not have to enter unnecessary information.

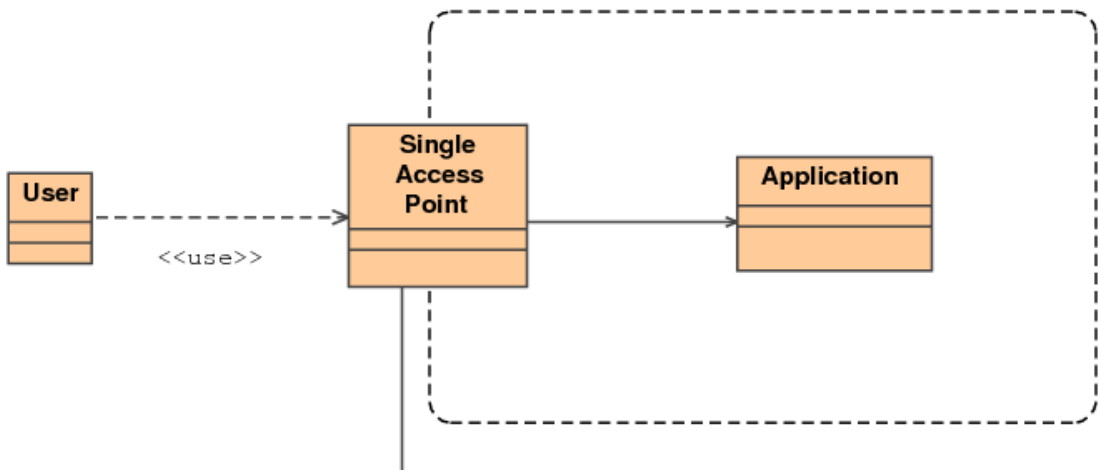
Example

There are many examples of Single Access Point. In order to access an NT workstation, there is a single login screen which all users must go through to access the system. This Single Access Point validates the user and insures that only valid users access the system and also provides Roles for only allowing users to see and do what they have permissions to do. Most UNIX systems also have a Single Access Point for getting a console shell. Oracle applications also have many applications such as SQLPlus and the like that provide a Single Access Point as the only means for running those applications.

Solution

Set up only one way to get into the system, and if necessary, create a mechanism for deciding which sub-applications to launch.

Structure



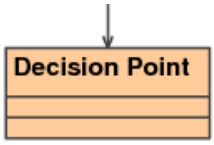


Figure 1: Single Access Point structure.

See Figure 1 .

Dynamics

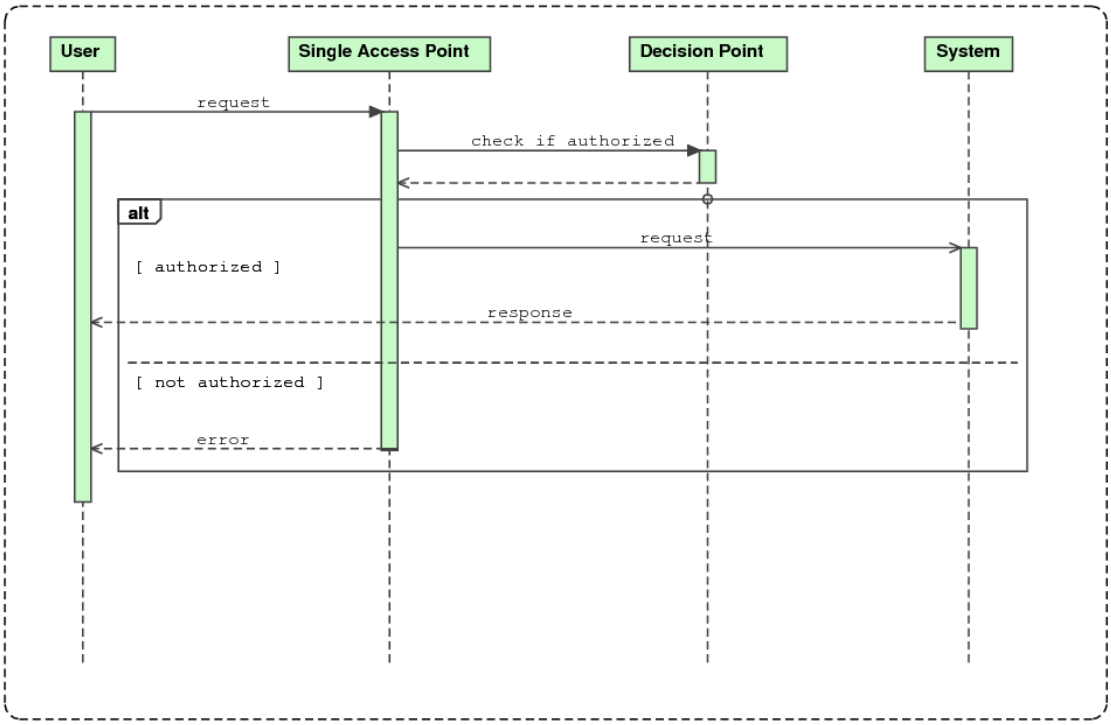


Figure 2: Event sequence for the Single Access Point.

See Figure 2 .

Participants

- User The user which will use the system.
- Single Access Point The only entrance point to the application.
- Decision Point The decision point can decide both about whether a user is allowed into the system or not, and if needed about which sub-applications to launch for te request.
- System The system (possibly composed of sub-systems)which is protected by the single access point.

Collaborations

- The user sends a request through the Single Access Point.
- The Single Access Point contacts the decision point to decide whether or not the request is allowed.
- If the request is allowed, it is sent through to the system. Otherwise, an error is returned to the user.

Implementation

(Nothing given)

Pitfalls

(Nothing given)

Consequences

- **+** A Single Access Point provides a place where everything within the application can be setup properly. This single location can help ensure all values are initialized correctly, application setup is performed correctly, and the application does not reach an invalid state.
- **+** Control flow is simpler since everything must go through a single point of responsibility in order for access to be allowed. Note, Single Access Point is only as secure as the steps leading up to it.
- **-** The application cannot have multiple entry points to make entering an application easier and more flexible.

Known uses

The following are security-related known uses:

- UNIX telnet and Windows NT login applications use Single Access Point for logging into the system. These systems also create the necessary Roles for the current Session.
- Most application login screens are a Single Access Point into programs because they are the only way to startup and run the given application.
- The Caterpillar/NCSA Financial Model Frameworkhas a FMLogin class, which provides both Single Access Point and Check Point.
- The PLoP'98registration programprovides a Single Access Point for logging into the system and entering in credit card information when users registered for PLoPÂ¿98.
- Secure web servers, such as Java DeveloperÂ¿s Connection appear to have multiple access points for each URL. However, the web server forces each user through a login window before letting them download early access software.Other, non-security related uses are:
- Any application that launches only one way, ensuring a correct initial state.
- Windows95, also uses a login window which is a Single Access Point, but it is not secure because it allows any user to override the login screen.
- Single creational methods provide for only one way to create a class. For example, Points in VisualWorks Smalltalkguides you to creating valid points by providing a couple of creational methods that ensure the Object is initialized correctly. Kent BeckÂ¿s describes Constructor Methods as a single way to create well-formed instances of objects. These are put into a single ``instance creation''protocol. This becomes the Single Access Point to create new objects.
- Constructor Parameter Methodinitializes all instance variables through a single method, which is really a Single Access Point for that class to initialize its instance variables.
- Concurrent programs can encapsulate non-concurrent objects inside an object designed for concurrency. Synchronization is enforced through this Single Access Point. Pass-Through Host designdeals with synchronization by forwarding all appropriate methods to the Helper using unsynchronized methods. This works because the methods are stateless with respect to the Host class.

Subject Descriptor

Pattern documentation

Quick info

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

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

Problem

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.

Forces

- 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.

Example

(Nothing given)

Solution

Encapsulate the attributes for a subject in a Subject Descriptor, and support operations to provide access to the complete current set of attributes, or a filtered subset of those attributes.

Structure

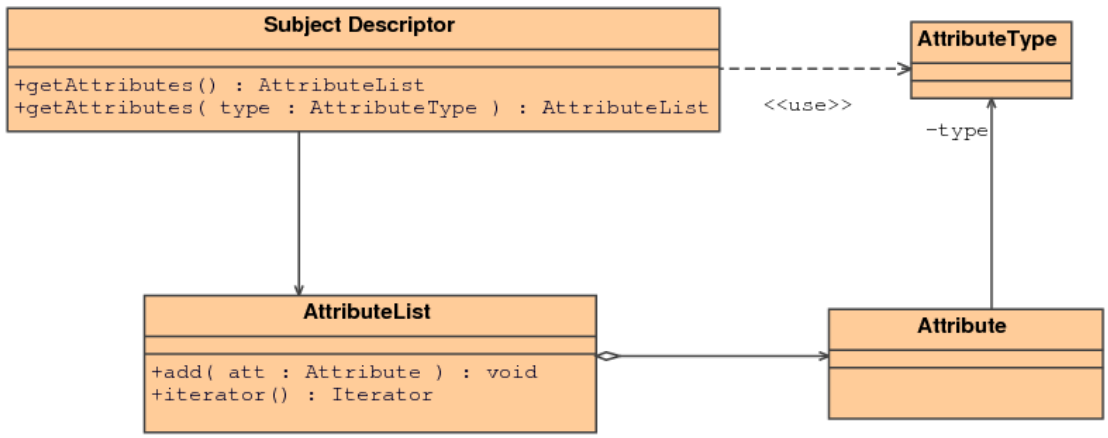
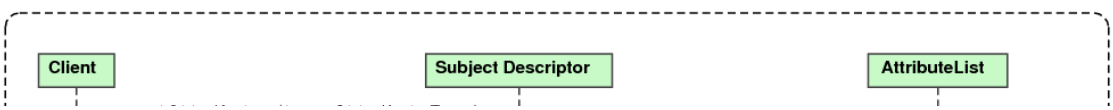


Figure 1: Class layout of the Subject Descriptor

See Figure 1 .

Dynamics



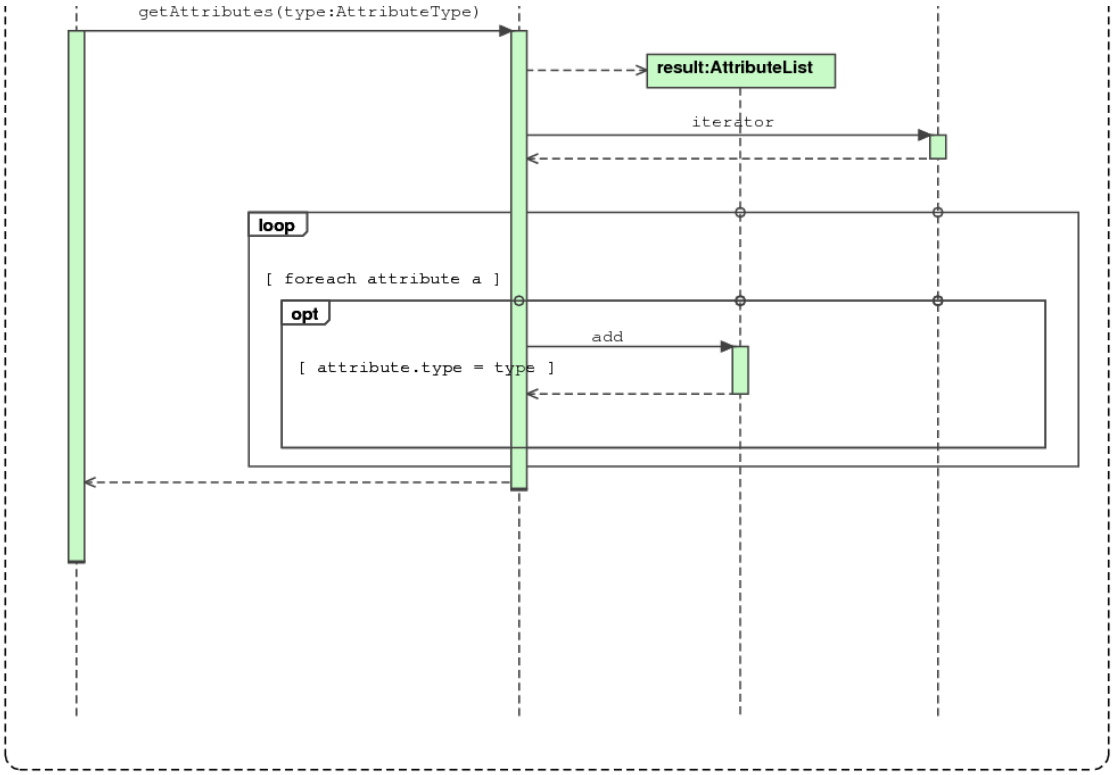


Figure 2: Retrieving a filtered set of attributes

See Figure 2 .

Participants

See also Figure 1 .

- 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.

Collaborations

Attribute List returns an Iteratorallowing the caller to operate on the individual Attributes referenced in the list.

Attribute List may be a a guarded type, consulting a 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.

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.

Pitfalls

(Nothing given)

Consequences

- 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).

Known uses

- 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.

- CORBASecurity SecurityLevel2:: 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.