Security Analysis of Core J2EE Patterns Project

INTRODUCTION

Most application security experts focus on a single activity for integrating design into the SDLC: threat modeling . Threat modeling is excellent at approximating an application's attack surface but, in our experience, developers sometimes do not have the time, budget or security know-how to build an adequate threat model. Perhaps more importantly, developers cannot create a comprehensive threat model until they complete the application design.

This reference guide aims at dispensing security best practices to developers to make security decisions during design. We focus on one of the most important concepts in modern software engineering: design patterns. Ever since the publication of the seminal Design Patterns Book , developers have reused common patterns such as Singleton and Factory Method in large-scale software projects. Design patterns offer a common vocabulary to discuss application design independent of implementation details. One of the most critically acclaimed pattern collections in the Java Enterprise Edition (JEE) community is the Core J2EE Patterns book by Deepak Alur, Dan Malks and John Crupi . Developers regularly implement patterns such as "Application Controller", "Data Access Object" or "Session Façade" in large, distributed JEE applications and in frameworks such as Spring and Apache Struts . We aim to dispense security best practices so that developers can introduce security features and avoid vulnerabilities independent of their underlying technology choices such as which Model View Controller (MVC) framework to use.

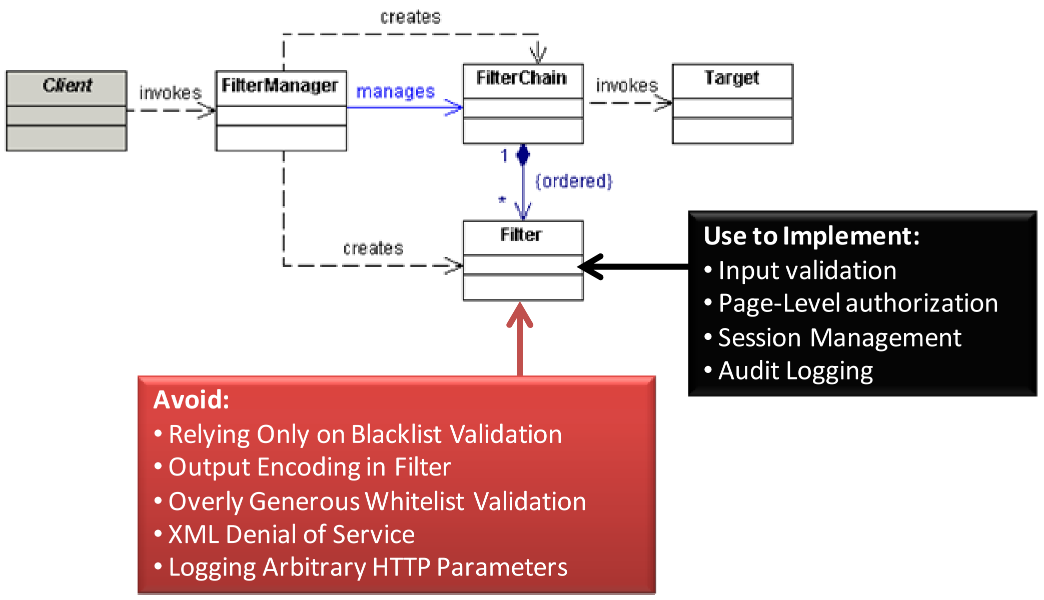
Java developers currently have access to patterns for security code (e.g. how to develop authentication, how to implement cryptography) such as the Core Security Patterns book. We hope our guide will help address the critical shortage of advice on securely coding using existing design patterns. Your feedback is critical to improving the quality and applicability of the best practices listed in the Security Analysis of Core J2EE Design Patterns. Please contact the mailing list at owasp_security_analysis_j2ee@lists.owasp.org with comments or questions and help improve the guide for future developers.

PRESENTATION-TIER PATTERNS

INTERCEPTING FILTER

The *Intercepting Filter* pattern may be used in instances where there is the need to execute logic before and after the main processing of a request (pre and postprocessing). The logic resides in `Filter` objects and typically consist of code that is common across multiple requests. The Servlet 2.3 Specification provides a mechanism for building filters and chaining of `Filters` through configuration. A `FilterManager` controls the execution of a number of loosely-coupled `Filters` (referred to as a `FilterChain`), each of which performs a specific action. This Standard Filter Strategy can also be replaced by a Custom Filter Strategy which replaces the Servlet Specification's object wrapping with a custom implementation.

DIAGRAM



ANALYSIS

Avoid

Relying Only on a Blacklist Validation Filter

Developers often use blacklists in `Filters` as their only line of defense against input attacks such as Cross Site Scripting (XSS). Attackers constantly circumvent blacklists because of errors in canonicalization and character encoding[i]. In order to sufficiently protect applications, do not rely on a blacklist validation filter as the sole means of protection; also validate input with strict whitelists on all input and/or encode data at every sink.

Output Encoding in Filter

Encoding data before forwarding requests to the `Target` is too early because the data is too far from the sink point and may actually end up in several sink points, each requiring a different form of encoding. For instance, suppose an application uses a client-supplied e-mail address in a Structured Query Language (SQL) query, a Lightweight Directory Access Protocol (LDAP) lookup, and within a Hyper Text Markup Language (HTML) page. SQL, LDAP, and HTML are all different sinks and each requires a unique form of encoding. It may be impossible to encode input at the `Filter` for all three sink types without breaking functionality. On the other hand, performing encoding after the `Target` returns data is too late since data will have already reached the sink by the time it reaches the `Filter`.

Overly Generous Whitelist Validation

While attempting to implement whitelist validation, developers often allow a large range of characters that may include potentially malicious characters. For example, some developers will allow all printable ASCII characters which contain malicious XSS and SQL injection characters such as less than signs and semi-colons. If your whitelists are not sufficiently restrictive, perform additional encoding at each data sink.

XML Denial of Service

If you use *Intercepting Filter* to preprocess XML messages, then remember that attackers may try many different Denial of Service (DOS) attacks on XML parsers and validators. Ensure either the web server, application server, or the first `Filter` on the chain performs a sanity check on the *size* of the XML message **prior** to XML parsing or validation to prevent DOS conditions.

Logging Arbitrary HTTP Parameters

A common cross-cutting application security concern is logging and monitoring of user actions. Although an *Intercepting Filter* is ideally situated to log incoming requests, avoid logging entire HTTP requests. HTTP requests contain user-supplied parameters which often include confidential data such as passwords, credit card numbers and personally identifiable information (PII) such as an address. Logging confidential data or PII may be in violation of privacy and/or security regulations.

Use to Implement

Input Validation

Use an *Intercepting Filter* to implement security input validation consistently across all presentation tier pages including both Servlets and JSPs. The `Filter's` position between the client and the front/application controllers make it an ideal location for a blacklist against all input. Ideally, developers should always employ whitelist validation rather than blacklist validation; however, in practice developers often select blacklist validation due to the difficulty in creating whitelists. In cases where blacklist validation is used, ensure that additional encoding is performed at each data sink (e.g. HTML and JavaScript encoding).

Page-Level Authorization

Use an *Intercepting Filter* to examine requests from the client to ensure that a user is authorized to access a particular page. Centralizing authorization checks removes the burden of including explicit page-level authorization deeper in the application. The Spring Security framework[ii] employs an *Intercepting Filter* for authorization.

Remember that page-level authorization is only one component of a complete authorization scheme. Perform authorization at the command level if you use `Command` objects, the parameter level such as HTTP request parameters, and at the business logic level such as *Business Delegate* or *Session Façade*. Remember to propagate user access control information such as users' roles to other design layers like the *Application Controller*. The OWASP Enterprise Security Application Programming Interface (ESAPI)[iii] uses `ThreadLocal`[iv] objects to maintain user authorization data throughout the life of a thread.

Session Management

Session management is usually one of the first security controls that an application applies to a request. Aside from container-managed session management controls such as idle timeout and invalidation, some applications implement controls such as fixed session timeout, session rotation and session-IP correlation through proprietary code. Use an *Intercepting Filter* to apply the additional session management controls before each request is processed.

Invalidating the current session token and assigning a new session token after authentication is a common defense against session fixation attacks. This control can also be handled in an *Intercepting Filter* specifically configured to intercept authentication requests. You may alternatively use a generic session management `Filter` that intercepts all requests, and then use conditional logic to check, specifically, for authentication requests in order to apply a defense against session fixation attacks. Be aware, however, that using a generic `Filter` introduces maintenance overhead when you implement new authentication paths.

Audit Logging

Since *Intercepting Filters* are often designed to intercept all requests, they are ideally situated to perform logging of user actions for auditing purposes. Consider implementing a `Filter` that intercepts all requests and logs information such as:

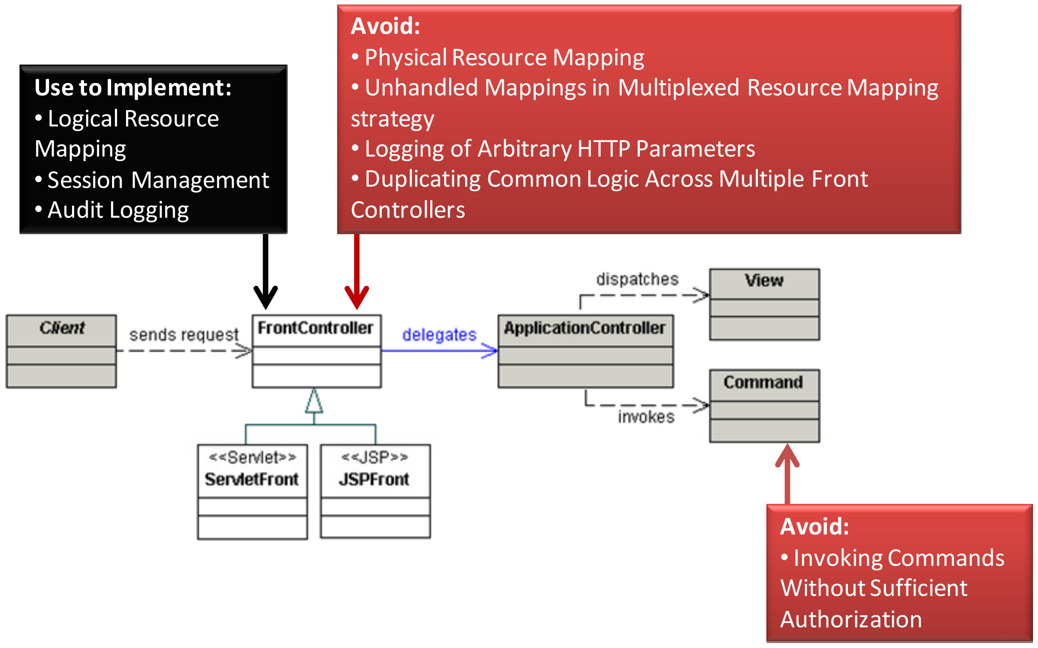
- Username for authenticated requests
- Timestamp of request
- Resource requested
- Response type such as success, error, etc.

The logging filter should be configured as the first `Filter` in the chain in order to log all requests irrespective of any errors that may occur in `Filters` further down the chain. Never log confidential or PII data.

FRONT CONTROLLER

Processing a request typically consists of a number of request handling activities such as protocol handling, navigation and routing, core processing, and dispatch as well as view processing[vii]. A controller provides a place for centralizing this common logic performed for each request. Typically implemented as a Servlet, the *Front Controller* can perform this common logic and further delegate the action management (servicing the request) and view management (dispatching a view for user output) activities to an *Application Controller*. This pattern provides centralization of request control logic and partitioning of an application between control and processing.

DIAGRAM



Use to Implement:
• Logical Resource Mapping
• Session Management
• Audit Logging
Avoid:
• Physical Resource Mapping
• Unhandled Mappings in Multiplexed Resource Mapping strategy
• Logging of Arbitrary HTTP Parameters
• Duplicating Common Logic Across Multiple Front Controllers
Client
sends request
FrontController
delegates
ApplicationController
dispatches
View
invokes
Command
<<Servlet>> ServletFront
<<JSP>> JSPFront
Avoid:
• Invoking Commands Without Sufficient Authorization

ANALYSIS

Avoid

Physical Resource Mapping

The Physical Resource Mapping strategy maps user-supplied parameters directly to physical resources such as files residing on the server. Attackers often take advantage of this strategy to gain illicit access to resources. In a directory traversal exploit, for example, clients supply the server with the physical location of a file such as "file=statement_060609.pdf". Attackers attempt to access other files on the server by supplying malicious parameters such as "file=../../../../../etc/password". If the application blindly accepts and opens any user-supplied filename then the attacker may have access to a whole array of sensitive files, including properties and configuration files that often contain hard-coded passwords.

Developers sometimes mitigate directory traversal attacks by checking for the presence of a specific prefix or suffix, such as verifying that the file parameter begins with “statement” and ends with “.pdf”. A crafty attacker can take advantage of null character injection and enter “file=statement_060609.pdf/../../../../etc/password%00.pdf”. Java will see that the resource beings with “statement” and ends with “.pdf” whereas the operating system may actually drop all remaining characters after the %00 null terminator and open the password file.

As a rule, avoid using the Physical Resource Mapping strategy altogether. If you must use this strategy, ensure that the application operates in a sandboxed environment with the Java Security Manager and/or employs sufficient operating system controls to protect resources from unauthorized access.

Invoking Commands Without Sufficient Authorization

In the Command and Controller strategy, users supply a `Command` object which the `Application Controller` subsequently handles by invoking an action. Developers who rely on client-side controls and page-level access control often forget to check if the user is actually *allowed* to invoke a given `Command`.

Attackers take advantage of this vulnerability by simply modifying a parameter. A common example is a Create Read Update Delete (CRUD) transaction, such as http://siteurl/controller?command=viewUser&userName=jsmith. An attacker can simply modify “viewUser” to “deleteUser”. Often developers assume that if clients cannot see a link to “deleteUser” on a web page then they will not be able to invoke the “deleteUser” command. We like to call this *GUI-based Authorization* and it is a surprisingly common vulnerability in web applications.

Ensure that clients are actually allowed to invoke the supplied command by performing an authorization check on the application server. Provide the `Application Controller` sufficient data about the current user to perform the authorization check, such as roles and username. Consider using a `Context` object to store user data.

Unhandled Mappings in the Multiplexed Resource Mapping Strategy

The Multiplexed Resource Mapping strategy maps sets of logical requests to physical resources. For example, all requests that end with a “.ctrl” suffix are handled by a `Controller` object. Often developers forget to account for non-existent mappings, such as suffixes not associated with specific handlers.

Create a default `Controller` for non-existent mappings. Ensure the `Controller` simply provides a generic error message; relying on application server defaults often leads to propagation of detailed error messages and sometimes even reflected XSS in the error message (e.g. “The resource <script>alert(‘xss’)</script>.pdf could not be found”).[v]

Logging of Arbitrary HTTP Parameters

A common cross-cutting application security concern is logging and monitoring of user actions. Although a *Front Controller* is ideally situated to log incoming requests, avoid logging entire HTTP requests. HTTP requests contain user-supplied parameters which often include confidential data such as passwords and credit card numbers and personally identifiable information (PII) such as an address. Logging confidential data or PII may be in violation of privacy and/or security regulations.

Duplicating Common Logic Across Multiple Front Controllers

If you use multiple *Front Controllers* for different types of requests, be sure to use the Base Front strategy to centralize security controls common to all requests. Duplicating cross-cutting security concerns such as authentication or session management checks in multiple *Front Controllers* may decrease maintainability. In addition, the inconsistent implementation of security checks may result in difficult-to-find security holes for specific use cases. If you use the BaseFront strategy to encapsulate cross-cutting security controls, then declare all security check methods as *final* in order to prevent method overriding and potentially skipping security checks in subclasses. The risk of overriding security checks increases with the size of the development team.

Use to Implement

Logical Resource Mapping

The Logical Resource Mapping strategy forces developers to explicitly define resources available for download and prevents directory traversal attacks.

Session Management

Session management is usually one of the first security controls that an application applies to a request. Aside from container-managed session management controls such as idle timeout and invalidation, some applications implement controls such as fixed session timeout, session rotation, and session-IP correlation through proprietary code. Use a *Front Controller* to apply the additional session management controls before each request is processed.

Invalidating the current session token and assigning a new session token after authentication is a common defense against session fixation attacks. This control can also be handled by *Front Controller* .

Audit Logging

Since *Front Controllers* are often designed to intercept all requests, they are ideally situated to perform logging of user actions for auditing purposes. Consider logging request information such as:

- Username for authenticated requests
- Timestamp of request
- Resource requested

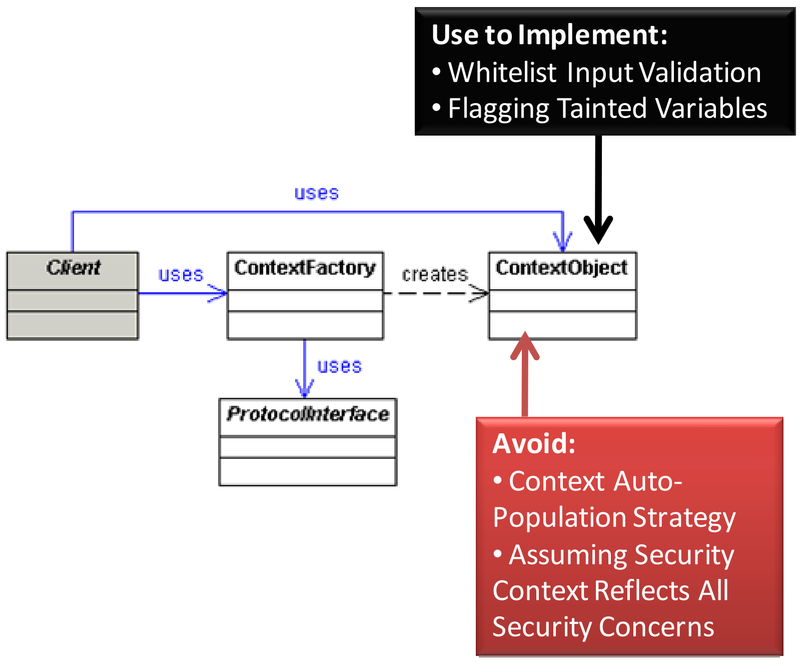
- Response type such as success, error, etc.

Never log confidential or PII data.

Context Object

In a multi-tiered applications, one tier may retrieve data from an interface using a specific protocol and then pass this data to another tier to be used for processing or as input into decision logic. In order to reduce the dependency of the inner tiers on any specific protocol, the protocol-specific details of the data can be removed and the data populated into a `ContextObject` which can be shared between tiers. Examples of such data include HTTP parameters, application configuration values, or security data such as the user login information, defined by the Request Context, Configuration Context, and Security Context strategies, respectively. By removing the protocol-specific details from the input, the *Context Object* pattern significantly reduces the effort required to adapt code to a change in the application's interfaces.

Diagram


Use to Implement:
• Whitelist Input Validation
• Flagging Tainted Variables
uses
Client
uses
ContextFactory
creates
ContextObject
uses
ProtocolInterface
Avoid:
• Context Auto-Population Strategy
• Assuming Security Context Reflects All Security Concerns

Analysis

Avoid

Context Object Auto-Population Strategy

The Context Object Auto-Population strategy uses client-supplied parameters to populate the variables in a `Context` object. Rather than using a logical mapping to match parameters with `Context` variables, the Context Object Auto-Population strategy automatically matches `Context` variable names with parameter names. In some cases, developers maintain two types of variables within a `Context` object: client-supplied and server supplied. An e-commerce application, for example, might have a ShoppingCartContext object with client-supplied product ID and quantity variables and a price variable derived from a server-side database query. If a client supplies a request such as "http://siteurl/controller?command=purchase&prodID=43quantity=2" then the Context Object Auto-Population strategy will automatically set the ShoppingCartContext.prodId=43 and ShoppingCartContext.quantity=2. What if the user appends "&price=0.01" to the original query? The strategy automatically sets the ShoppingCarContext.price=0.01 even though the price value should not be client controlled. Ryan Berg and Dinis Cruz and of Ounce labs documented this as a vulnerability in the Spring Model View Controller (MVC) framework[vi].

Avoid using the Context Object Auto-Population strategy wherever possible. If you must use this strategy, ensure that the user is actually allowed to supply the variables to the context object by performing explicit authorization checks.

Assuming Security Context Reflects All Security Concerns

The Security Context strategy should more precisely be called an Access Control Context strategy. Developers often assume that security is comprised entirely of authentication, authorization and encryption. This line of thinking often leads developers to believe that using the Secure Socket Layer (SSL) with user authentication and authorization is sufficient for creating a secure web application.

Also remember that fine-grained authorization decisions may be made further downstream in the application architecture, such as at the *Business Delegate*. Consider propagating roles, permissions, and other relevant authorization information via the `Context` object.

Use to Implement

Whitelist Input Validation

The Request Context Validation strategy uses the `RequestContext` object to perform validation on client-supplied values. The Core J2EE Patterns book provides examples for form and business logic level validation, such as verifying the correct number of digits in a credit card. Use the same mechanism to perform security input validation with regular expressions. Unlike *Intercepting Filter*s, `RequestContexts` encapsulate enough context data to perform whitelist validation. Many developers employ this strategy in Apache Struts applications by opting to use the Apache Commons Validator[vii] plugin for security input validation.

In several real-world implementations of Request Context Validation, the `RequestContext` only encapsulates HTTP parameters. Remember that malicious user-supplied input can come from a variety of other sources: cookies, URI paths, and other HTTP headers. If you do use the Request Context Validation strategy for security input validation then provide mechanisms for security input validation on other forms of input. For example, use an *Intercepting Filter* to validate cookie data.

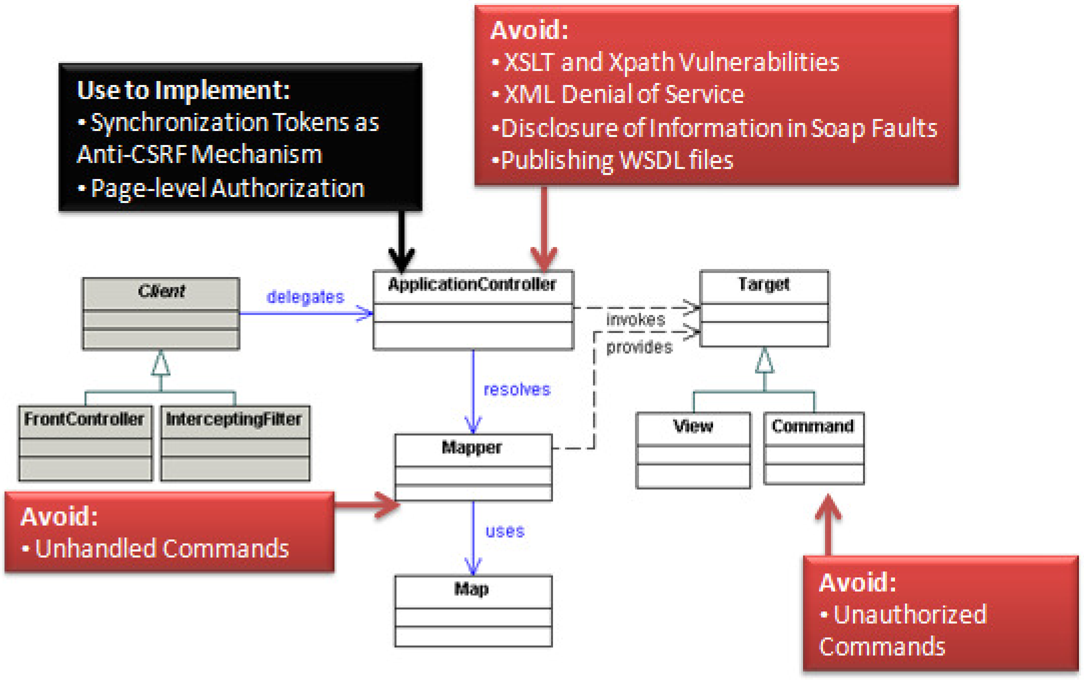
Flagging Tainted Variables

Conceptually, `Context` objects form the last layer where applications can differentiate between untrusted user-supplied data and trusted system-supplied data. For instance, a shopping cart `Context` object might contain a user-supplied shipping address and a database-supplied product name. Generally speaking, objects logically downstream from the `Context` object cannot distinguish user-supplied data from system-supplied data. If you encode data at sink points and you want to minimize performance impact by encoding only user-supplied data (as opposed to system generated data), then consider adding an "isTainted" flag for each variable in the `Context` class. Set "isTainted" to true if the variable is user supplied or derived from another user supplied value. Set "isTainted" to false if the variable is computer generated and can be trusted. Store the instance variable and "isTainted" booleans as key/value pairs in a collection with efficient lookups (such as a `WeakHashMap)`. Downstream in the application, simply check if a variable is tainted (originates from user-supplied input) prior to deciding to encode it at sink points. For instance, you might HTML, JavaScript, or Cascading Stylesheet (CSS) encode all tainted data that you print to stream in a Servlet while leaving untainted data as is.

Application Controller

While the *Front Controller* pattern stresses centralization of logic common to all incoming requests, the conditional logic related to mapping each request to a command and view set can be further separated. The `FrontController` performs all generic request processing and delegates the conditional logic to an `ApplicationController`. The `ApplicationController` can then decide, based on the request, which command should service the request and which view to dispatch. The `ApplicationController` can be extended to include new use cases through a map holding references to the Command and View for each transaction. *The Application Controller* pattern is central to the Apache Struts model view controller framework.

Diagram



Analysis

Avoid

Unauthorized Commands

In the Command Handler strategy, users supply a `Command` object which the `ApplicationController` subsequently handles by invoking an action. Developers who rely on client-side controls and page-level access control often forget to check if the user should actually be *allowed* to invoke a given `Command`.

Attackers take advantage of this vulnerability by simply modifying a parameter. A common example is a Create Read Update Delete (CRUD) transaction, such as http://siteurl/controller?command=viewUser&userName=jsmith. An attacker can simply modify "viewUser" to "deleteUser". Often developers assume that if clients cannot see a link to "deleteUser" on a web page then they will not be able to invoke the "deleteUser" command. We call this vulnerability *GUI-based Authorization* and it is a surprisingly common in web applications.

Ensure that clients are actually allowed to invoke the supplied command by performing an authorization check on the application server. Provide the `ApplicationController` sufficient data about the current user to perform the authorization check, such as roles and username. Consider using a `Context` object to store user data.

Unhandled Commands

Create a default response page for non-existent commands. Relying on application server defaults often leads to propagation of detailed error messages and sometimes even reflected XSS in the error message (e.g. "The resource <script>alert('xss')</script>.pdf could not be found")[viii].

XSLT and XPath Vulnerabilities

The Transform Handler strategy uses XML Stylesheet Language Transforms (XSLTs) to generate views. Avoid XSLT[ix] and related XPath[x] vulnerabilities by performing strict whitelist input validation or XML encoding on any user-supplied data used in view generation.

XML Denial of Service

If you use *Application Controller* with XML messages then remember that attackers may try Denial of Service (DOS) attacks on XML parsers and validators. Ensure either the web server, application server, or an *Intercepting Filter* performs a sanity check on the *size* of the XML message **prior** to XML parsing or validation to prevent DOS conditions.

Disclosure of Information in SOAP Faults

One of the most common information disclosure vulnerabilities in web services is when error messages disclose full stack trace information and/or other internal details. Stack traces are often embedded in SOAP faults by default. Turn this feature off and return generic error messages to clients.

Publishing WSDL Files

Web Services Description Language (WSDL) files provide details on how to access web services and are very useful to attackers. Many SOAP frameworks publish the WSDL by default (e.g. http://url/path?WSDL). Turn this feature off.

Use to Implement

Synchronization Token as Anti-CSRF Mechanism

Synchronizer tokens are random numbers designed to detect duplicate web page requests. Use cryptographically strong random synchronized tokens to help prevent anti-Cross Site Request Forgery (CSRF). Remember, however, that CSRF tokens can be defeated if an attacker can successfully launch a Cross Site Scripting (XSS) attack on the application to programmatically parse and read the synchronization token.

Page-Level Authorization

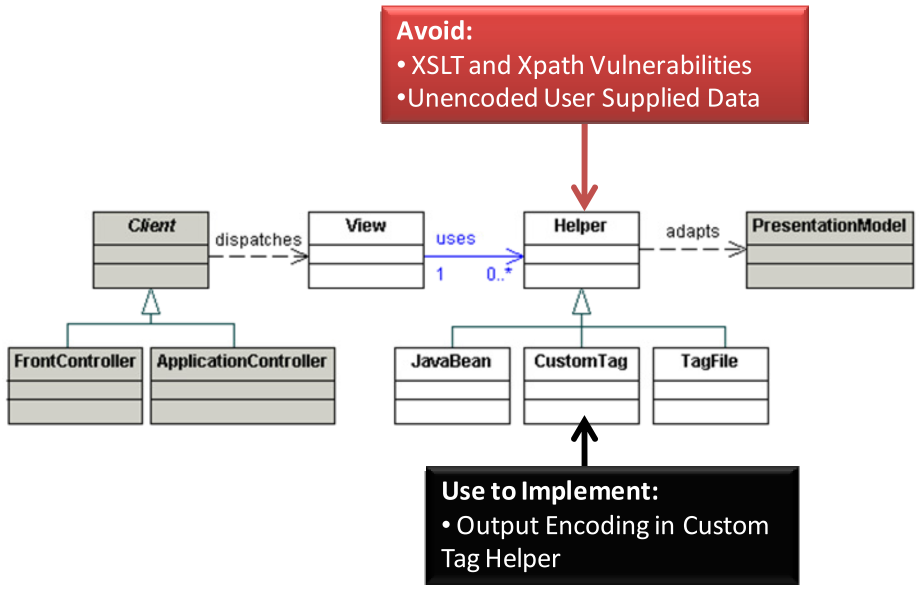
If not already done using an *Intercepting Filter*, use the *Application Controller* to examine client requests to ensure that only authorized users access a particular page. Centralizing authorization checks removes the burden of including explicit page-level authorization deeper in the application.

Remember that page-level authorization is only one component to a complete authorization scheme. Perform authorization at the command level if you use `Command` objects, the parameter level such as HTTP request parameters, and at the business logic level such as *Business Delegate* or *Session Façade*. Remember to propagate user access control information such as users' roles to other design layers. The OWASP Enterprise Security Application Programming Interface (ESAPI)[xi] uses `ThreadLocal`[xii] objects to maintain user authorization data throughout the life of a thread.

VIEW HELPER

View processing occurs with each request and typically consists of two major activities: processing and preparation of the data required by the view, and creating a view based on a template using data prepared during the first activity. These two components, often termed as `Model` and `View`, can be separated by using the *View Helper* pattern. Using this pattern, a view only contains the formatting logic either using the Template-Based View strategy such as a JSP or Controller-Based View Strategy using Servlets and XSLT transformations. The `View` then makes use of `Helper` objects that both retrieve data from the `PresentationModel` and encapsulate processing logic to format data for the View. JavaBean Helper and Custom Tag Helper are two popular strategies which use different data types (JavaBeans and custom view tags respectively) for encapsulating the model components.

DIAGRAM



ANALYSIS

Avoid

XSLT and XPath Vulnerabilities

Some developers elect to use Xml Stylesheet Language Transforms (XSLTs) within their `Helper` objects. Avoid XSLT[xiii] and related XPath[xiv] vulnerabilities by performing strict whitelist input validation or XML encoding on any user-supplied data used in view generation.

Unencoded User Supplied Data

Many JEE developers use standard and third party tag libraries extensively. Libraries such as Java Server pages Tag Libraries (JSTL)[xv] and Java Server Faces (JSF)[xvi] simplify development by encapsulating view building logic and hiding difficult-to-maintain scriptlet code. Some tags automatically perform HTML encoding on common special characters. The "c:out" and "$(fn:escapeXml(variable))" Expression Language (EL) tags in JSTL automatically HTML encode potentially dangerous less-than, greater-than, ampersand, and apostrophe characters. Encoding a small subset of potentially malicious characters significantly reduces the risk of common Cross Site Scripting (XSS) attacks in web applications but may still leave other less-common malicious characters such as percentage signs unencoded. Many other tags do not perform any HTML encoding. Most do not perform any encoding for other sink types, such as JavaScript or Cascading Style Sheets (CSS).

Assume tag libraries do not perform output encoding unless you have specific evidence to the contrary. Wherever possible, wrap existing tag libraries with custom tags that perform output encoding. If you cannot use custom tags then manually encode user supplied data prior to embedding it in a tag.

Use to Implement

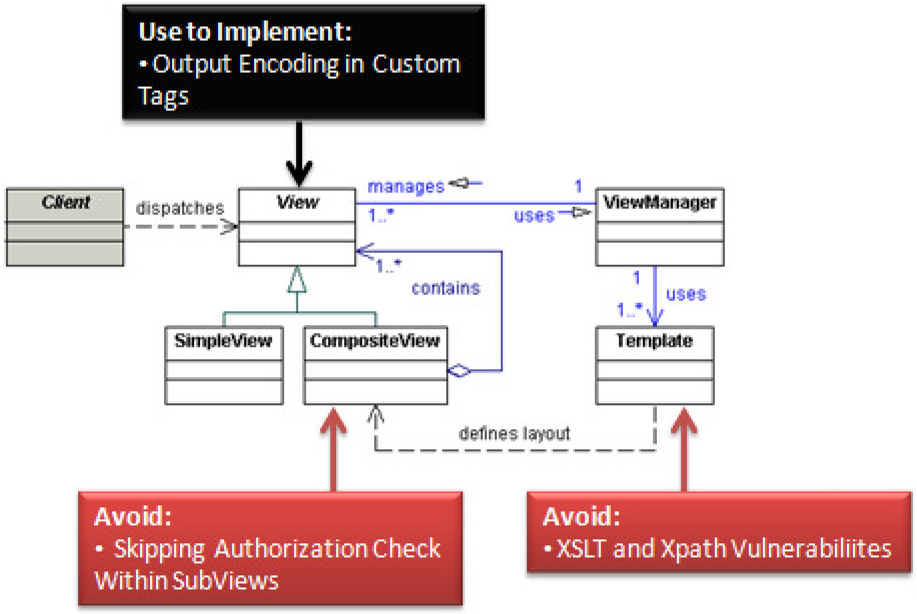
Output Encoding in Custom Tag Helper

The Custom Tag Helper strategy uses custom tags to create views. Wherever possible embed output encoding in custom tags to automatically protect against Cross Site Scripting (XSS) attacks.

Composite View

Applications often contain common `View` components, both from layout and content perspectives, across multiple `Views` in an application. These common components can be modularized by using the *Composite View* pattern which allows for a `View` to be built from modular, atomic components. Examples of modular components which can be reused are page headers, footers, and common tables. A `CompositeView` makes use of a `ViewManager` to assemble multiple views. A `CompositeView` can be assembled by JavaBeans, standard JSP tags such as <jsp:include>, or custom tags, using the JavaBean View Management, Standard Tag View Management, or Custom Tag View Management strategies respectively.

Diagram



Analysis

Avoid

XSLT and XPath Vulnerabilities

The Transformer View Management strategy uses XML Stylesheet Language Transforms (XSLTs). Avoid XSLT[xvii] and related XPath[xviii] vulnerabilities by performing strict whitelist input validation or XML encoding on any user-supplied data used in view generation.

Skipping Authorization Check Within SubViews

One of the most common web applications vulnerabilities is weak functional authorization. Developers sometimes use user role data to dynamically generate views. In the Core J2EE Pattern books the authors refer to these dynamic views as “Role-based content”. Role-based content prevents unauthorized users from *viewing* content but does not prevent unauthorized users from *invoking* Servlets and Java Server Pages (JSPs). For example, imagine a JEE accounting application with two JSPs – accounting_functions.jsp which is the main accounting page and gl.jsp representing the general ledger. In order to restrict access to the general ledger, developers include the following content in accounts.jsp:

```
<region: render template=’portal.jsp’>
     <region:put section=’general_ledger’ content=’gl.jsp’
      role=’accountant’ />
</region:render>
```

The code snippet above restricts the content in gl.jsp to users in the accountant role. Remember, however, that a user can simply access gl.jsp directly – a flaw that can be even more devastating if invoking gl.jsp with parameters causes a transaction to run such as posting a debit or credit.

Use to Implement

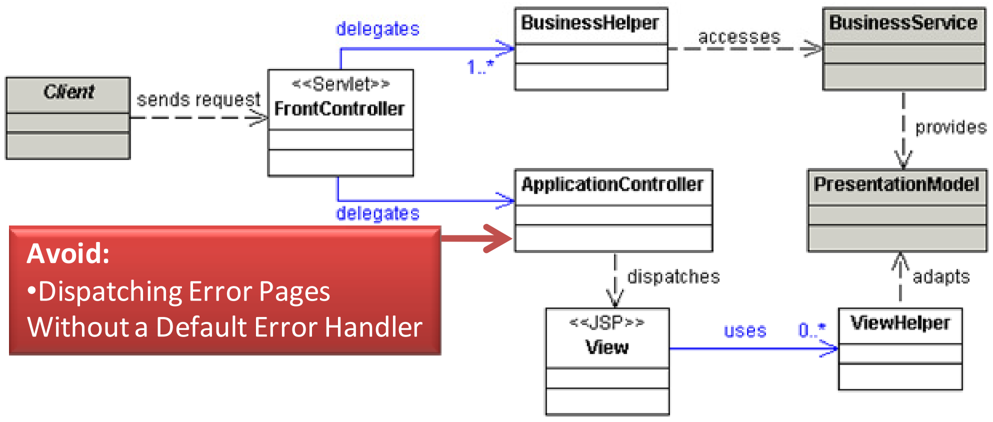
Output Encoding in Custom Tags

The Custom Tag View Management strategy uses custom tags to create views. Wherever possible embed output encoding in custom tags to automatically protect against Cross Site Scripting (XSS) attacks.

Service to Worker

Applications commonly dispatch a `View` based exclusively on the results of request processing. In this the *Service to Worker* pattern, the `Controller` (either `FrontController` or `ApplicationController`) performs business logic and, based on the result of that logic, creates a `View` and invokes its logic. *Service to Worker* is different from *Dispatcher View* because in the latter the `View` logic is invoked **before** the business logic is invoked.

Diagram



Analysis

Avoid

Dispatching Error Pages Without a Default Error Handler

In *Service to Worker*, an `ApplicationController` manages the flow of web pages. When an application encounters an error, the `ApplicationController` typically determines which error page to dispatch. The Core J2EE Patterns book uses the following line of code to determine which error page to forward the user to:

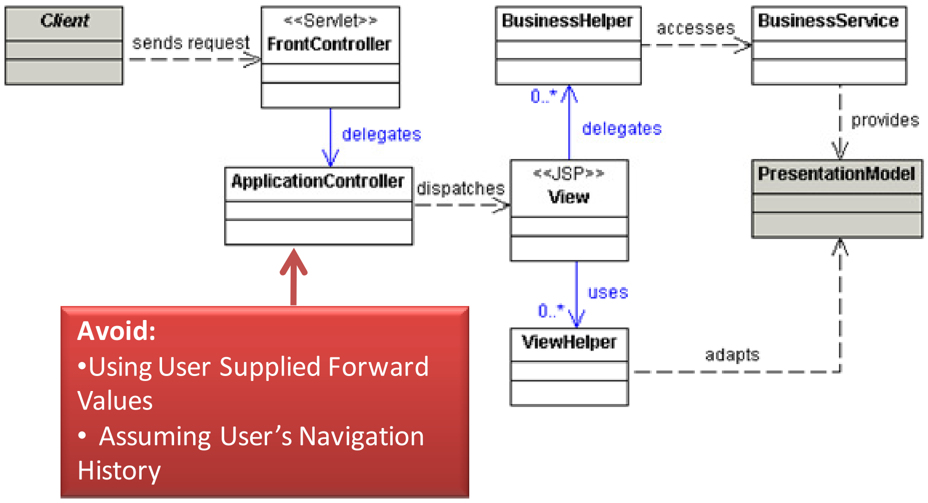
```
next= ApplicationResources.getInstance().getErrorPage(e);
```

Many developers use the approach of mapping exceptions to particular error pages, but do not account for a default error page. As applications evolve, the job of creating a friendly error page for every exception type becomes increasingly difficult. In many applications the `ApplicationController` will simply dump a stack trace or even redisplay the client's request back to them. Avoid both scenarios by providing a default generic error page. In Java EE you can implement default error handling through web.xml configuration.

Dispatcher View

In instances where there is a limited amount or no business processing required prior to generating a response, control can be passed to the `View` directly from the `ApplicationController`. The `ApplicationController` performs general control logic and is only responsible for mapping the request to a corresponding `View`. Dispatcher View pattern is typically used in situations where the response is entirely static or the response is generated from data which has been generated from a previous request. Using this pattern, view management is often so trivial (i.e. mapping of an alias to a resource) that no application-level logic is required and it is handled by the container.

Diagram



Analysis

Avoid

Using User Supplied Forward Values

The `ApplicationController` determines which view to dispatch to a user. In the Core J2EE Patterns book, the authors provide the following sample of code to determine the next page to dispatch to a user:

```
next = request.getParameter(“nextview”);
```

Depending on how you forward users to the next page, an attacker may be able to:

- Gain unauthorized access to pages if the forwarding mechanism bypasses authorization checks
- Forward unsuspecting users to a malicious site. Attackers can take advantage of the forwarding feature to craft more convincing phishing emails with links such as http://siteurl/page?nextview=http://malicioussite/attack.html. Since the URL originates from a trusted domain – "http://siteurl" in the example – it is less likely to be caught by phishing filters or careful users
- Perform Cross Site Scripting (XSS) in browsers that accept JavaScript URLs. For example, an attacker might send a phishing email with the following URL: http://siteurl/page?nextview=javascript:do_something_malicious
- Perform HTTP Response Splitting if the application uses an HTTP redirect to forward the user to the next page. The user supplies the literal value of the destination URL. Because the application server uses that URL in the header of the HTTP redirect response, a malicious user can inject carriage return and line feeds into the response using hex encoded %0A and %0D

Do not rely on literal user-supplied values to determine the next page. Instead, use a logical mapping of numbers to actual pages. For example, in http://siteurl/page?nextview=1 the "1" maps to "edituser.jsp" on the server.

Assuming User's Navigation History

The Core J2EE Patterns book discusses an example where the *Dispatcher View* pattern may be used. In the example, the application places an intermediate model in a temporary store and a later request uses that model to generate a response. When writing code for the second view, some developers assume that the user has already navigated to the first view thereby instantiating and populating the intermediate model. An attacker may take advantage of this assumption by directly navigating to the second view before the first. Similarly, if the first view automatically dispatches the second view (through an HTTP or JavaScript redirect), an attacker may drop the second request, resulting in an unexpected state. Always user server-side checks such as session variables to verify that the user has followed the intended navigation path.

Business Tier Patterns

Middle and Integration Tier Security

The majority of Open Web Application Security Project (OWASP) Top 10 vulnerabilities occur in the presentation tier. While attacks do exploit vulnerabilities in the business and enterprise integration tiers, the attacks typically *originate* from presentation tier web pages. For example, a SQL injection payload is usually delivered as part of an HTTP request to a web page, but the exploit itself usually occurs in the integration tier.

We examine the business and integration tier patterns for security from two perspectives:

1. Attacks originating from the presentation tier, such as a Cross-Site Scripting (XSS) attack sent by a malicious web client
2. Attacks originating from the business and integration tiers, such as an unauthorized web service request to an integration tier device

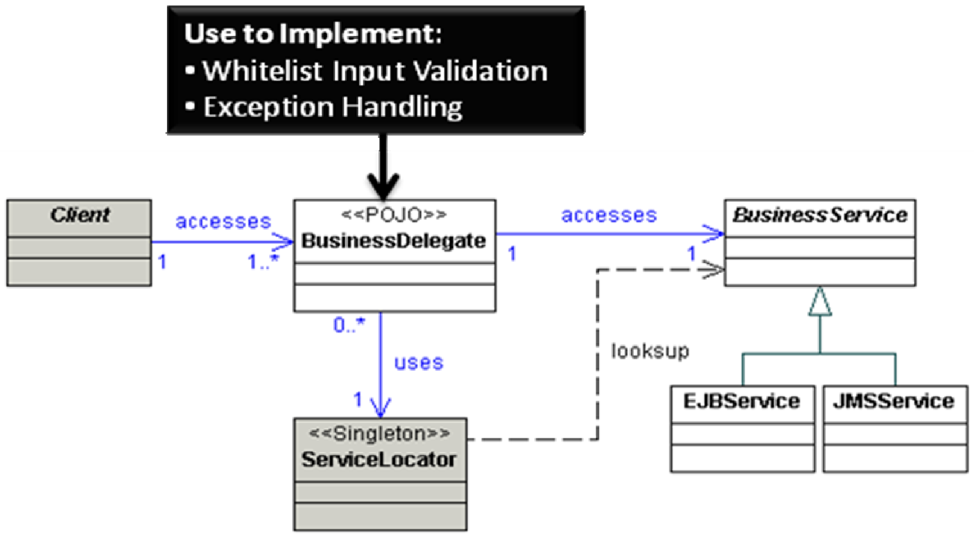
The second perspective is important enough to warrant special attention. Many organizations ignore attacks originating from within the internal network. Unfortunately, the notion that organizations can completely trust insiders is flawed[xix].In application security, developers sometimes argue that the process of launching an attack in the business or integration tiers is complicated and therefore less likely. As organizations increasingly adopt Service Oriented Architectures (SOA) in business and integration tiers with standards like Simple Object Access Protocol (SOAP) and Representation State Transfer (REST), security tools targeting these protocols are becoming readily available[xx] [xxi] [xxii].

Even if you have not yet experienced an insider security incident, remember that insider attacks are on the rise and the systems you architect today may remain in production for years or even decades to come. Create secure applications by building-in protection from insider threats.

Business Delegate

The *Business Delegate* serves as an abstraction of the business service classes of the business tier from the client tier. Implementing a business delegate effectively reduces the coupling between the client tier and business tier, and allows for greater flexibility and maintainability of the application code. The most significant benefit of this pattern is the capability to hide potentially sensitive implementation details of the business services from the calling client tier. Furthermore, a business delegate can effectively handle business tier exceptions (such as `java.rmi.Remote` exceptions) and translate them into more meaningful, application-level exceptions to be forwarded to the client.

Diagram



Analysis

Use to Implement

Whitelist input validation

The DelegateProxyStrategy uses `BusinessDelegate` objects as simple proxies to underlying services. Each `BusinessDelegate` is business context specific and is therefore a good place to implement whitelist security input validation. Remember, however, that `BusinessDelegate` validation only applies to input originating from the presentation tier. You need to duplicate input validation functionality for all other channels that access the same business tier, such as web services.

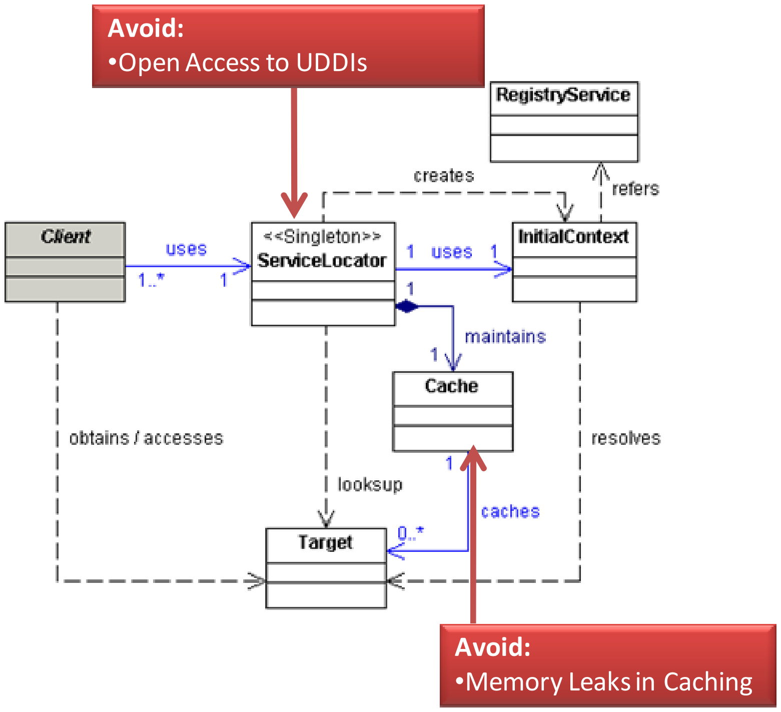
Exception Handling

A property of sound exception management is the practice of throwing an exception that is meaningful to the target tier. For example, a `JMSException` caught by a business tier object should not be propagated to the client tier. Instead, a custom, application-level, exception should be sent to the client tier. `BusinessDelegate` can effectively perform this translation, intercepting service-level exceptions from the business tier and throwing application-level exceptions to the client tier. This practice helps protect implementation-level details of the business services from calling clients. Note that exceptions can occur within and across many tiers, and restricting exception handling logic to the `BusinessDelegate` alone is insufficient.

Service Locator

JEE application modules from the client tier often need to access business or integration tier services, such as JMS components, EJB components, or data sources. These components are typically housed in a central registry. In order to communicate with this registry, the client uses the Java Naming and Directory Interface (JNDI) API to obtain an `InitialContext` object containing the desired object name. Unfortunately, this process can be repeated across several clients, increasing complexity and decreasing performance. The *Service Locator* pattern implements a `ServiceLocator` singleton class that encapsulates API lookups and lookup complexities, and provides an easy-to-use interface for the client tier. This pattern helps promote reuse of resource-intensive lookups, and decouples the client tier from the underlying lookup implementation details.

Diagram



Analysis

Avoid

Memory Leaks in Caching

Developers typically use caching to improve performance. If you are not careful about implementing caching, however, you can actually degrade performance over time.

Many developers use collections like HashMaps to maintain references to cached objects. Suppose you maintain a cache using a HashMap where the keys are Strings describing services in the `ServiceLocator` and the values are `Target` objects. After a period of inactivity you want to remove the Target reference to free memory. Simply running a method like `close()` on the Target object will not actually make the object eligible for garbage collection because the cache HashMap still maintains a pointer to the Target object. You must remember to remove references from the HashMap otherwise you will experience memory degradation overtime and possibly suffer from Denial of Service (DoS) conditions.

Use a WeakHashMap or another collection of WeakReferences or SoftReferences rather than traditional collections to maintain caches. A WeakHashMap maintains WeakReferences[xxiii] to key and value objects, meaning the cache will allow key and value objects to be eligible for garbage collection. Using WeakReferences allows the garbage collector to automatically remove objects from the cache when necessary, as long as the application does not maintain any strong references (i.e. regular pointers) to the object.

If you cannot use WeakReferences or SoftReferences then ensure you regularly remove objects from the cache.

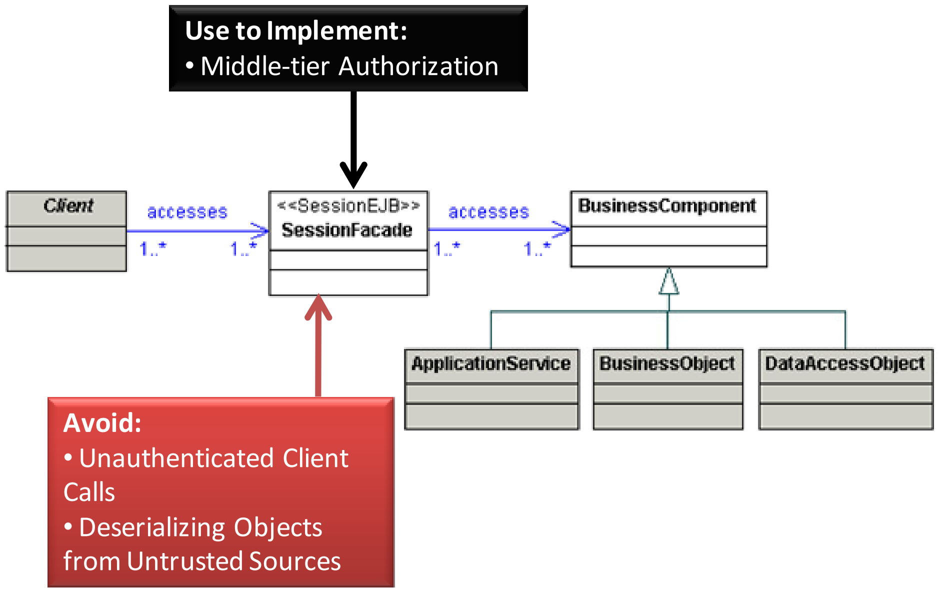
Open Access to UDDI

Many developers opt to use web services instead of more traditional middle tier technologies like Enterprise Java Beans (EJBs). In Service Oriented Architecture (SOAs), Universal Description, Discovery and Integration (UDDI), registries often play the role of Service Locator. UDDIs are extremely valuable to attackers since they house Web Service Definition Language (WSDL) files that provide blueprints of how and where to access web services. If you use UDDI, ensure that only authorized users can access registries. Consider using mutual certificate authentication with each UDDI request. Also ensure that an authenticated user is actually authorized to access specific parts of the registry.

SESSION FAÇADE

Business components in JEE applications often interact with many different services, each of which may involve complex processes. Exposing these components directly to the client tier is undesirable, as this leads to a tight coupling between the client and business tiers, and can lead to code duplication. The *Session Façade* pattern is used to encapsulate the services of the business tier and acts as an interface to the client. Developers typically implement a `SessionFacade` class as a session bean and expose only the interfaces that clients require, hiding the complex interdependencies between `BusinessObjects`. Very little or no business logic should be placed within a `SessionFacade`.

DIAGRAM



ANALYSIS

Avoid

Unauthenticated Client Calls

Allowing unauthenticated access to Enterprise Java Beans (EJBs), web services, or other middle tier components leaves applications susceptible to insider attacks. In some cases, developers configure application servers to allow open access to the underlying DataSources such as databases. Build system-to-system authentication into the Session Façade itself or use container-managed mechanisms such as mutual certificate authentication. Maybe also mention that DataSources can be made available by app servers for anyone on the network to retrieve.

Deserializing Objects from Untrusted Sources

Developers often use Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP) to communicate between the presentation and business tiers. RMI uses Java serialization and deserialization to transfer objects over the network. Treat deserialization as a dangerous function because some Java Virtual Machines are vulnerable to Denial Of Service (DOS) conditions when attackers transmit specially crafted serialized objects[xxiv]. Remember to treat client or third party-supplied serialized objects as untrusted input; malicious users can modify the serialized version of an object to inject malicious data or even supply objects running malicious code[xxv]

Decrease the risk of a DOS attack by authenticating requests *prior* to deserializing data. Also upgrade and patch Java Virtual Machines just like other critical software components such as operating systems.

Use to Implement

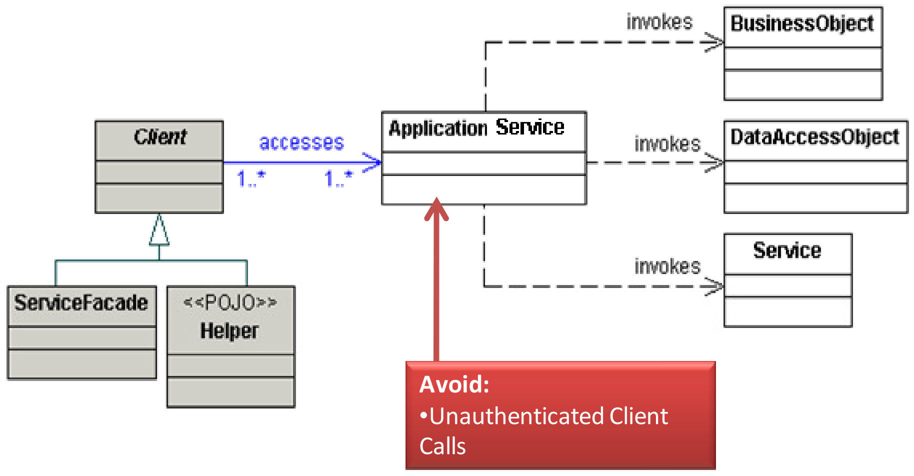
Middle Tier Authorization at Session Facade

Session Façades are entry points to the middle tier, meaning they are ideal for handling middle-tier authorization. Ensure clients have sufficient privileges to access any function exposed through *Session Façades*.

Application Service

While the *Session Façade* pattern allows for the decoupling of the client tier from the business tier and encapsulates the services of the business tier, `SessionFacade` objects should not encapsulate business logic. An application may contain multiple `BusinessObjects` (see the next pattern) that perform distinct services, but placing the business logic here introduces coupling between `BusinessObjects`. The *Application Service* pattern provides a uniform service layer for the client. An `ApplicationService` class acts as a central location to implement business logic that encapsulates a specific business service of the application and reduces coupling between `BusinessObjects`. The `ApplicationService` object can implement common logic acting on different `BusinessObjects`, implement use case-specific business logic, invoke business methods in a `BusinessObject`, or methods in other `ApplicationServices`. An `ApplicationService` is commonly implemented as a plain-old Java object (POJO).

Diagram



Analysis

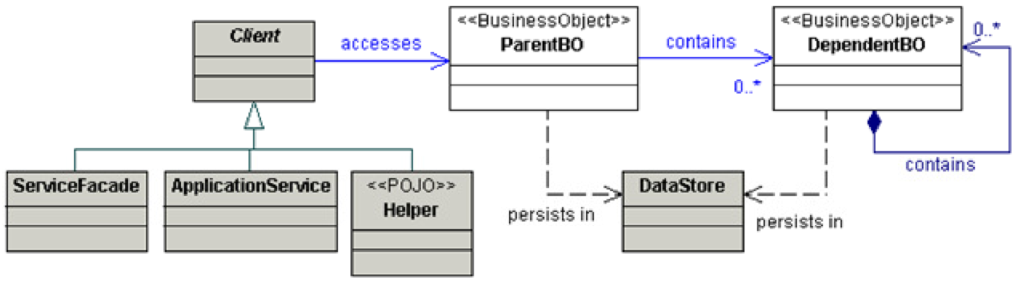
Unauthenticated Client Calls

Allowing unauthenticated access to Enterprise Java Beans (EJBs), web services, or other middle tier components leaves applications susceptible to insider attacks. Build system-to-system authentication into the *Application Service* itself or use container-managed mechanisms such as mutual certificate authentication.

Business Object

Ideally, a multi-tiered application is decoupled into presentation, business and data tiers. Of the three, the business tier often results in duplicated code and tight coupling with data logic. The *Business Object* pattern allows for the separation of the business state and behavior from the rest of the application, and the centralization of both. `BusinessObjects` encapsulate the core business data, and implement the desired business behavior, all while separating persistence logic from the business logic. Using this pattern, the client interacts directly with a `BusinessObject` that will delegate behavior to one or more business entities, and manage its own persistence logic using a desired persistence strategy, such as POJOs or EJB entity beans.

Diagram



Analysis

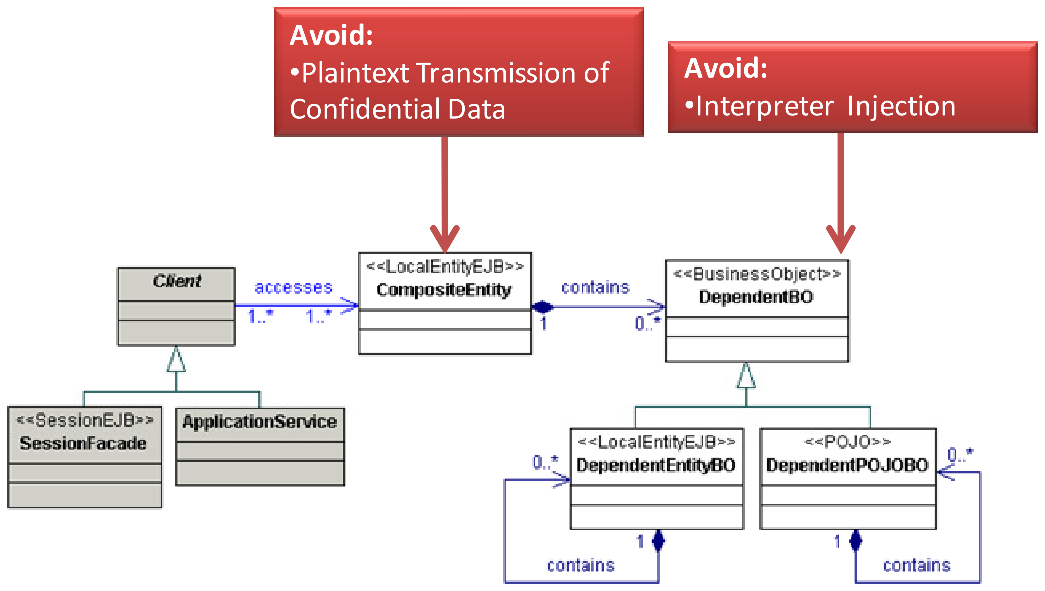
General Notes

In most cases, *Business Objects* do not implement common security features such as authentication and authorization, data validation and encoding, or session management. Ideally, a *Business Object* should simply provide accessor methods and offer basic business logic functions. You still need to develop with standard secure coding practices such as error handling and logging[xxvi].

COMPOSITE ENTITY

Business Objects separate business logic and business data. While Enterprise Java Beans (EJB) entity beans are one way to implement *Business Objects*, they come with complexity and network overhead. The *Composite Entity* pattern uses both local entity beans and Plain Old Java Objects (POJOs) to implement persistent *Business Objects*. The *Composite Entity* can aggregate a single `BusinessObject` and its related dependant `BusinessObjects` into coarse-grained entity beans. Using this pattern allows developers to leverage the EJB architecture, such as container-managed transactions, security, and persistence.

DIAGRAM



ANALYSIS

General Notes

Entity Bean Alternatives

With the rise of third party persistence frameworks such as Hibernate and IBATIS, fewer developers use entity beans. In general, application developers can take security advice for *Composite Entity* and apply it to persistence frameworks other than Entity Beans.

Avoid

Interpreter Injection

If you use bean managed persistence (BMP) with entity beans, then protect against injection attacks by using secure prepared statements, stored procedures, or appropriate encoding for non-database persistence mechanisms such as XML encoding for creating XML documents.

In most cases persistence frameworks like Hibernate automatically protect against SQL injection. Remember, however, that many persistence frameworks allow you to create custom queries through special functions like Hibernate Query Language (HQL). If you dynamically concatenate strings to create custom queries, then your application may be vulnerable to injection despite the fact that you use a persistence framework[xxvii].

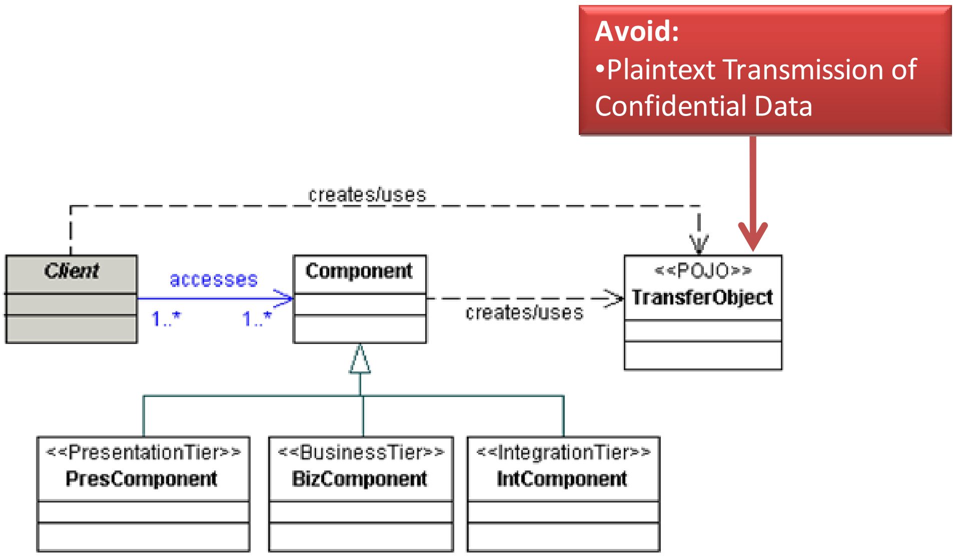
Plaintext Transmission of Confidential Data

The Composite Transfer Object strategy creates a `TransferObject` to send to a remote client. Many organizations do not use encrypted communication channels within the internal network, so a malicious insider attacker might sniff confidential data within a `TransferObject` during transmission. Either do not serialize confidential variables or use an encrypted communication channel like Secure Socket Layer (SSL) during transmission.

TRANSFER OBJECT

Integration patterns such as *Session Façade* and *Business Object* often need to return data to the client. A client, however, may potentially call several getter methods on these objects, and when these patterns are implemented as remote enterprise beans, a significant network overhead is incurred with each call. The *Transfer Object* pattern is designed to optimize the transfer of data to the client tier. A `TransferObject` encapsulates all data elements within a single structure and returns this structure to the calling client. The *Transfer Object* pattern serves to reduce network traffic, transfer bulk data with far fewer remote calls, and promote code reuse.

DIAGRAM



ANALYSIS

Avoid

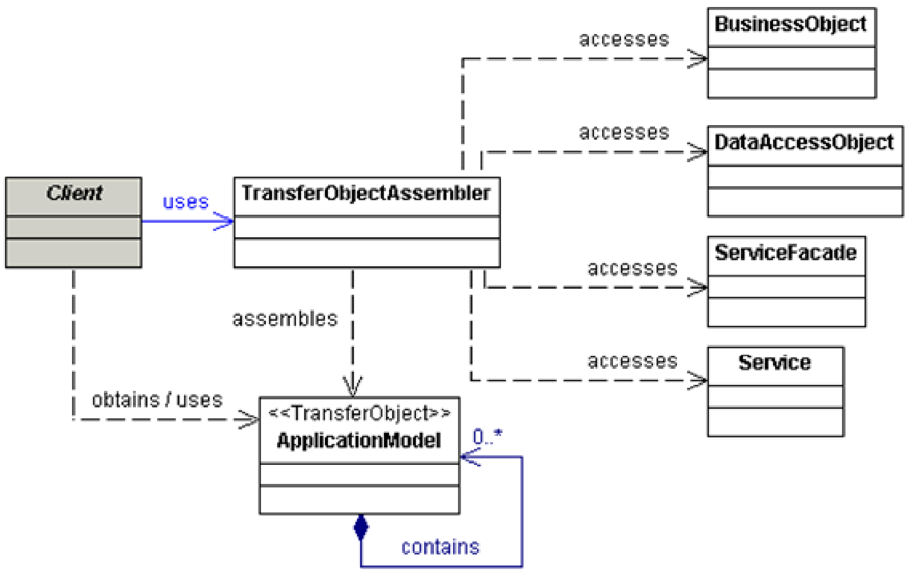
Plaintext Transmission of Confidential Data

Many organizations do not use encrypted communication channels within the internal network, so a malicious insider attacker might sniff confidential data within a `TransferObject` during transmission. Either do not serialize confidential variables or use an encrypted communication channel like Secure Socket Layer (SSL) during transmission.

TRANSFER OBJECT ASSEMBLER

Clients often need to access business data from potentially many different `BusinessObjects`, `TransferObjects`, or `ApplicationServices` to perform processing. Direct access to these different components by the client can lead to tight coupling between tiers and code duplication. The *Transfer Object Assembler* pattern provides an efficient way to collect multiple `Transfer Objects` across different business components and return the information to the client. Think of *Transfer Object Assembler* as a factory to create `Transfer Objects`. The client invokes the `TransferObjectAssembler`, which retrieves and processes different `TransferObjects` from the business tier, constructs a composite `TransferObject` known as the `ApplicationModel`, and returns the `ApplicationModel` to the client. Note that the client uses the `ApplicationModel` for read-only purposes, and does not modify any data contained within it.

DIAGRAM



ANALYSIS

Unauthenticated Client Calls

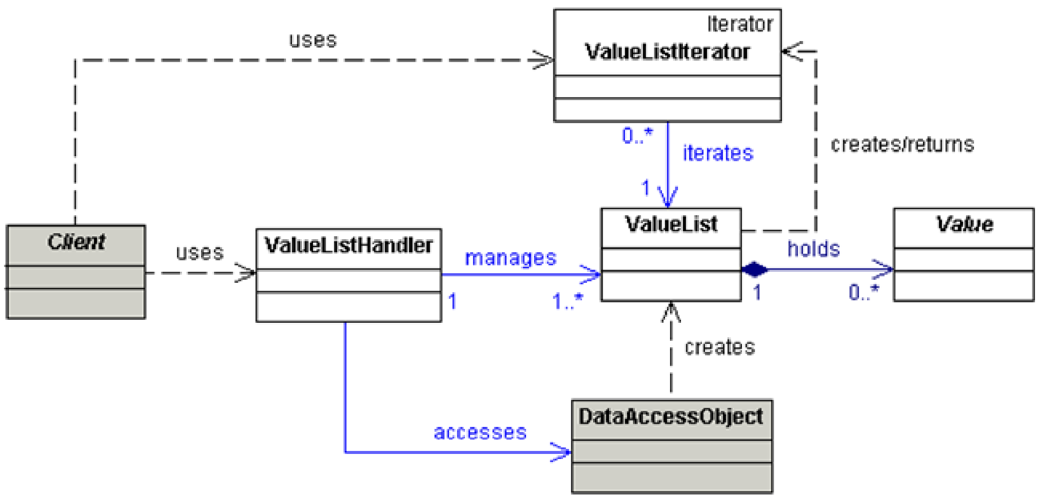
Allowing unauthenticated access to `TransferObjectAssemblers` or other middle tier components leaves applications susceptible to insider attacks. If you expose your

`TransferObjectAssemblers` directly to clients, then build in system-to-system authentication or use container-managed mechanisms such as mutual certificate authentication.

Value List Handler

Searching is a common operation in JEE applications. A search typically is initiated by the client tier and execution by the business tier. More concretely, this can involve several invocations of entity bean finder methods or `DataAccessObjects` by the client, particularly if the result set of the search is large. This introduces network overhead. The *Value List Handler* pattern affords efficient searching and result set caching, which can allow the client to traverse and select desired objects from the result set. A `ValueListHandler` object executes the search and obtains the results in a `ValueList`. The `ValueList` is normally created and manipulated by the `ValueListHandler` through a `DataAccessObject`. The `ValueList` is returned to the client, and the client can iterate through the list contents using a `ValueListIterator`.

Diagram



Analysis

General Notes

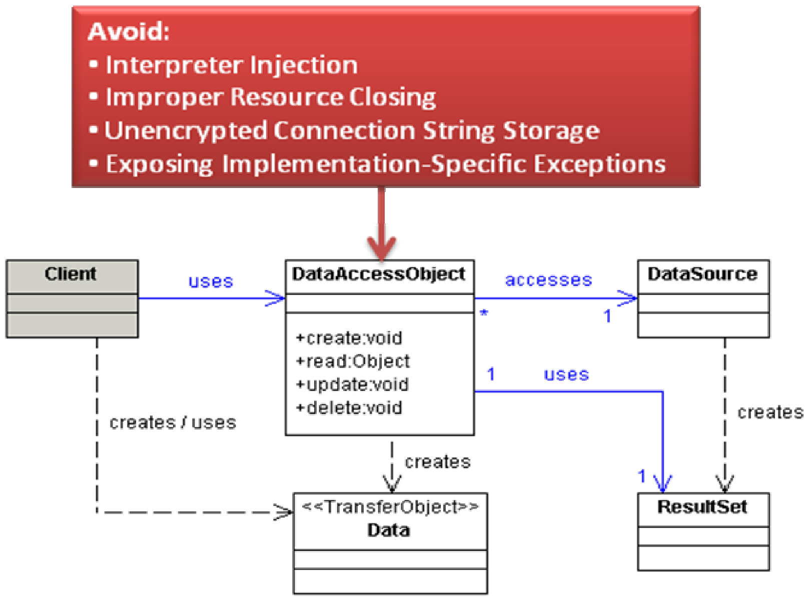
In most cases, *Value List Handlers* do not implement common security features such as authentication and authorization, data validation and encoding, or session management. You still need to develop with standard secure coding practices such as error handling and logging[xxviii].

INTEGRATION TIER PATTERNS

DATA ACCESS OBJECT

Most if not all JEE applications require access to enterprise or business data from a persistent data store. Data, however, can reside in many different kinds of repositories, from relational databases to mainframe or legacy systems. Mixing application logic with persistence logic introduces tight coupling and creates a maintenance nightmare for an application. The *Data Access Object* (DAO) pattern allows for the encapsulation of all access to the persistent data store and manages connections to the data tier, while hiding the implementation details of the persistence API from the client. A `DataAccessObject` implements create, find, update, and delete operations.

DIAGRAM



ANALYSIS

Avoid

Interpreter Injection

DataAccessObjects (*DAOs*) interact with interpreters such as SQL-enabled databases or Lightweight Directory Access Protocol (LDAP) enabled repositories. *DAOs* are therefore susceptible to injection-style attacks such as SQL injection. Avoid SQL attacks by using properly constructed prepared statements or stored procedures. For other interpreters, such as LDAP or XML data stores, use appropriate encoding to escape potentially malicious characters such as LDAP encoding or XML encoding. Remember to use a whitelist approach for encoding rather than a blacklist approach.

You may not find encoding methods for some interpreters, leaving your code highly susceptible to interpreter injection. In these cases use strict whitelist input validation.

Improper Resource Closing

Developers often forget to properly close resources such as database connections. Always close resources in a finally block, check for null pointers before closing on an object, and catch and handle any possible exception that may occur during resource closing *within* the finally block[xxix].

Unencrypted Connection String Storage

In order to communicate with a database or other backend server, most applications use a connection string that includes a user name and password. Developers often store this connection string un-encrypted in server configuration files such as server.xml. Malicious insiders and external attackers who exploit path traversal vulnerabilities[xxx] may be able to use a plaintext connection string to gain unauthorized access to the database.

Encrypt database and other server credentials during storage. Unfortunately, any method you use to encrypt the connection string has weaknesses. For instance, WebLogic provides an encrypt utility to encrypt any clear text property within a configuration file[xxxi], but the utility relies on an encryption key stored on the application server. Other methods, such as Jasypt's password based encryption mechanisms require either manual intervention or remote server communication during startup to implement securely[xxxii]. Nevertheless, storing a password plaintext is the least secure option so always use some form of encryption.

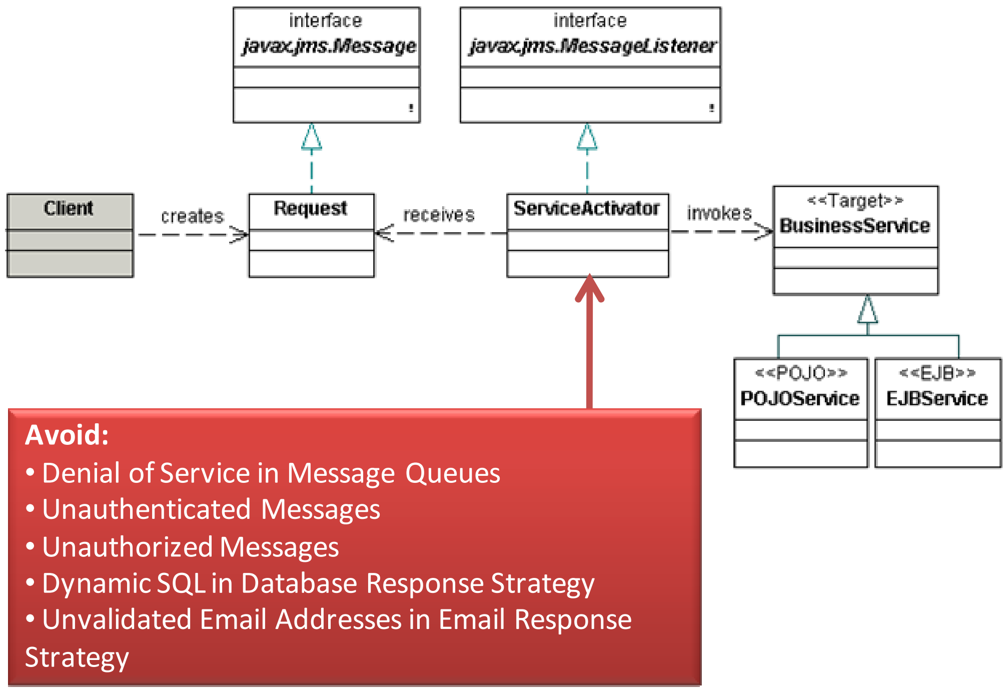
Exposing Implementation-specific Exceptions

A `DataAccessObject` makes a direct connection to the persistence layer and executes queries against it, returning data to the client. Such access requires correct handling and translation of exceptions that stem from the data tier. Ensure that exceptions that are thrown from implementation-specific packages, such as `java.sql.*` or `javax.sql.*` are handled and logged appropriately in the `DataAccessObject` and are not simply propagated to the client. One popular strategy is to log the exception and throw a custom-made exception to the client, such as a `DAOException`.

Service Activator

In JEE applications, certain business services can involve complex processes that may consume considerable time and resources. In such cases, developers often prefer to invoke these services asynchronously. The *Service Activator* pattern allows for the reception of asynchronous requests from a client and invokes multiple business services. A `ServiceActivator` class is typically implemented as a JMS listener that receives and parses messages from the client, identifies and invokes the correct business service to process the request, and informs the client of the outcome.

Diagram


interface
javax.jms.Message
interface
javax.jms.MessageListener
Client
creates
Request
receives
ServiceActivator
invokes
<<Target>>
BusinessService
<<POJO>>
POJOService
<<EJB>>
EJBService
Avoid:
• Denial of Service in Message Queues
• Unauthenticated Messages
• Unauthorized Messages
• Dynamic SQL in Database Response Strategy
• Unvalidated Email Addresses in Email Response Strategy

Analysis

Avoid

Denial of Service in Message Queues

Attackers may fill up message queues to launch a Denial of Service (DOS) attacks. Perform a sanity check on the size of a message prior to accepting it into the message queue.

Unauthenticated Messages

Allowing unauthenticated access to `BusinessServices` or other middle tier components leaves applications susceptible to insider attacks. Build system-to-system authentication into the `ServiceActivator` itself or use container-managed mechanisms such as mutual certificate authentication.

Another approach is to add authentication credentials into each message, similar to WS-Security authentication in web services[xxxiii]. Remember, however, that processor-intensive authentication functions may themselves leave applications vulnerable to DOS attacks.

Unauthorized Messages

`BusinessServices` often expose critical middle tier or backend functions. Include functional authorization checks to protect `BusinessServices` from authenticated user privilege escalation.

Developers sometimes skip middle tier authorization, relying instead on functional authorization checks in the presentation tier (e.g. application controller checks). A presentation-tier-only authorization approach might work with strong infrastructure-level access controls. Remember, however, that you must duplicate authorization checks on all channels that access the business service. Wherever possible use the `ServiceActivator` to enforce consistent access control for all messages.

Dynamic SQL in Database Response Strategy

The Database Response strategy stores the message response in a database that a client subsequently polls. As with other database interaction patterns, remember to use properly configure Prepared Statements or stored procedures to avoid SQL injection.

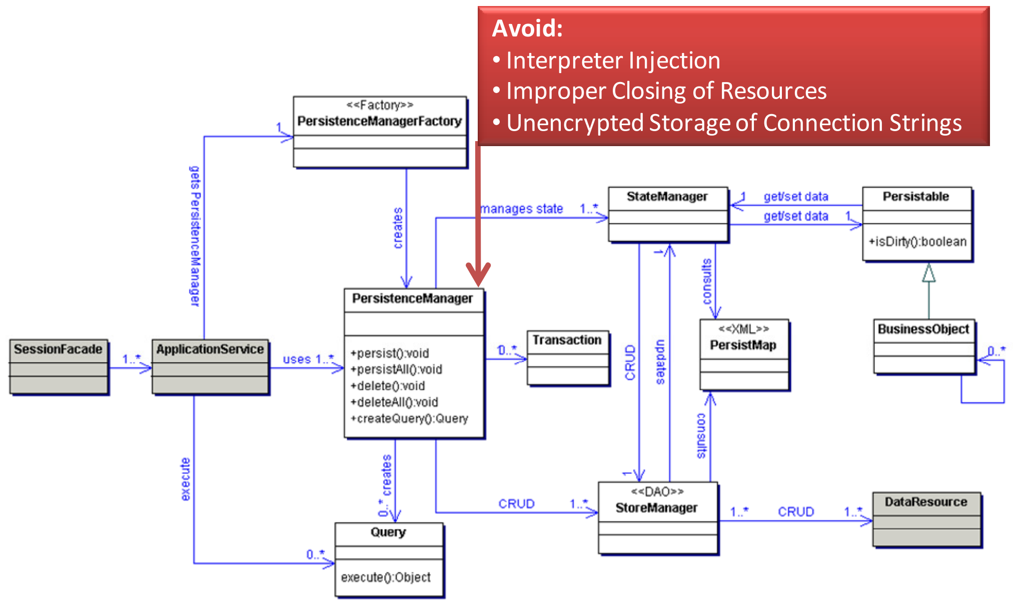
Unvalidated Email Addresses in Email Response Strategy

The Email Response strategy sends a message response though email. Malicious users who can influence the response may provide multiple email addresses and use the server as a gateway to send unauthorized spam messages. Use whitelist validation to ensure that users supply only one email address[xxxiv].

DOMAIN STORE

Patterns such as *Data Access Object* emphasize the value of separating persistence details `from BusinessObjects`. Some applications must also separate persistence details from the application object model. The *Domain Store* pattern allows for this separation, unlike container-managed persistence and bean-managed persistence strategies, which tie persistence code with the object model. The *Domain Store* pattern can be implemented using either a custom persistence framework or a persistence product, such as Java Data Objects.

DIAGRAM



ANALYSIS

Avoid

Interpreter Injection

Domain Stores interact with interpreters such as SQL-enabled databases or Lightweight Directory Access Protocol (LDAP) enabled repositories. DAOs are therefore susceptible to injection-style attacks such as SQL injection. Avoid SQL attacks by using properly constructed prepared statements or stored procedures. For other interpreters, such as LDAP or XML data stores, use appropriate encoding to escape potentially malicious characters such as LDAP encoding or XML encoding. Remember to use a whitelist approach for encoding rather than a blacklist approach.

You may not find encoding methods for some interpreters, leaving your code highly susceptible to interpreter injection. In these cases use strict whitelist input validation.

Properly Close Resources

Developers often forget to properly close resources such as database connections. Always close resources in a finally block, check for null pointers before closing on an object, and catch and handle any possible exception that may occur during resource closing *within* the finally block[xxxv].

Connection String Storage

In order to communicate with a database or other backend server, most applications use a connection string that includes a user name and password. Most developers store this connection string un-encrypted in server configuration files such as server.xml. Malicious insiders and external attackers who exploit path traversal vulnerabilities[xxxvi] may be able to use a plaintext connection string to gain unauthorized access to the database.

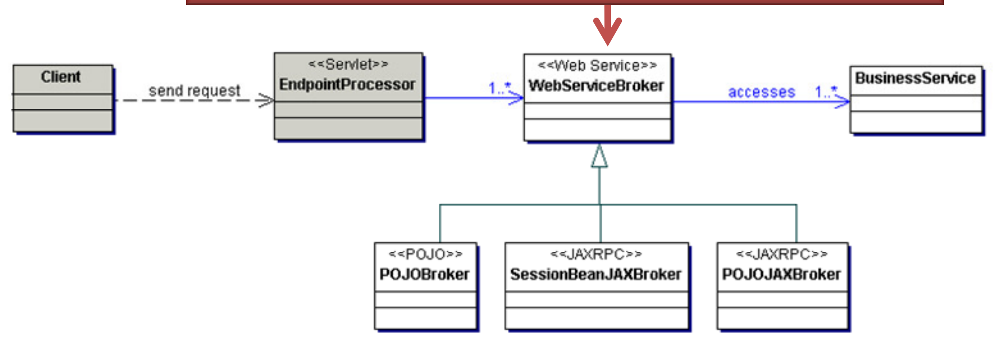
Encrypt database and other server credentials during storage. Unfortunately, any method you use to encrypt the connection string has weaknesses. For instance, WebLogic provides an encrypt utility to encrypt any clear text property within a configuration file[xxxvii], but the utility relies on an encryption key stored on the application server. Other methods, such as Jasypt's password based encryption mechanisms require either manual intervention or remote server communication during startup to implement securely[xxxviii]. Nevertheless, storing a password plaintext is the least secure option so always use some form of encryption.

Web Service Broker

A web service is a popular way of exposing business services to other applications. The integration of different systems, however, can typically involve incompatibilities and complexities. Furthermore, it is desirable to limit the set of services that are exposed by an application as web services. The *Web Service Broker* pattern uses XML and web protocols to selectively expose and broker the services of an application. A `WebServiceBroker` coordinates the interaction among services, collects responses and performs transactions. It is typically exposed using a WSDL. The `EndpointProcessor` class is the entry point into the web service, and processes the client request. The `EndpointProcessor` then invokes the web service through the `WebServiceBroker`, which brokers to one or more services.

Diagram

Avoid:
- XML Denial of Service
- Disclosure of Information in SOAP Faults
- Publishing WSDL Files
- Using DTDs
- Unvalidated Input on Web Services
- Unauthenticated and Unauthorized Web Service Requests



Analysis

Avoid

XML Denial of Service

Attackers may try Denial of Service (DOS) attacks on XML parsers and validators. Ensure either the web server, application server or an *Intercepting Filter* performs a sanity check on the *size* of the XML message **prior** to XML parsing or validation to prevent DOS conditions.

Disclosure of Information in SOAP Faults

One of the most common information disclosure vulnerabilities in web services is when error messages disclose full stack trace information and/or other internal details. Stack traces are often embedded in SOAP faults by default. Turn this feature off and return generic error messages to clients.

Publishing WSDL Files

Web Services Description Language (WSDL) files provide details on how to access web services and are very useful to attackers. Many SOAP frameworks publish the WSDL by default (e.g. http://url/path?WSDL). Turn this feature off.

Using DTDs

Document Type Definition (DTD) validators may be susceptible to a variety of attacks such as entity reference attacks. If possible, use XML Schema Definition (XSD) validators instead. If a DTD validator is required, ensure that the prologue of the DTD is not supplied by the message sender. Also ensure that external entity references are disabled unless absolutely necessary.

Unvalidated Input on Web Services

Web services often expose critical Enterprise Information Systems (EIS) that are vulnerable to interpreter injection attacks. Protect EIS systems at the web service level with strict input validation on all client-supplied parameters. XML encode untrusted data prior to its inclusion in a web service / XML response.

Unauthenticated and Unauthorized Web Service Requests

Like the other middle and EIS tier components, developers often employ weaker or altogether ignore authentication and authorization controls on web service requests – making web services an ideal target for attackers. Authenticate and authorize every web service request.

REFERENCES

[i] Data Validation, OWASP, http://www.owasp.org/index.php/Data_Validation#Reject_known_bad
[ii] Spring Security, http://static.springframework.org/spring-security/site/
[iii] OWASP ESAPI Project, OWASP, http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API
[iv] Thread Local object, Java API, http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ThreadLocal.html
[v] Struts XSS Vulnerability, Struts Wiki, http://wiki.apache.org/struts/StrutsXssVulnerability
[vi] Two Security Vulnerabilities in the Spring Framework's MVC, Ryan Berg and Dinis Cruz, http://www.ouncelabs.com/pdf/Ounce_SpringFramework_Vulnerabilities.pdf
[vii] Apache Commons Validator, Apache, http://commons.apache.org/validator/
[viii] Struts XSS Vulnerability, Struts Wiki, http://wiki.apache.org/struts/StrutsXssVulnerability
[ix] Security Risks with XSLT, http://msdn.microsoft.com/en-us/library/ms950792.aspx
[x] XPath Injection, Web Application Security Consortium Threat Classification, http://www.webappsec.org/projects/threat/classes/xpath_injection.shtml
[xi] OWASP ESAPI Project, OWASP, http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API
[xii] Thread Local object, Java API, http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ThreadLocal.html
[xiii] Security Risks with XSLT, http://msdn.microsoft.com/en-us/library/ms950792.aspx
[xiv] XPath Injection, Web Application Security Consortium Threat Classification, http://www.webappsec.org/projects/threat/classes/xpath_injection.shtml
[xv] JSTL Technology, Sun Microsystems, http://java.sun.com/products/jsp/jstl/
[xvi] Java Server Faces Technology, Sun Microsystems, http://java.sun.com/javaee/javaserverfaces/
[xvii] Security Risks with XSLT, http://msdn.microsoft.com/en-us/library/ms950792.aspx
[xviii] XPath Injection, Web Application Security Consortium Threat Classification, http://www.webappsec.org/projects/threat/classes/xpath_injection.shtml
[xix] The "Big Picture" of Insider IT Sabotage Across U.S. Critical Infrastructures, Moore, Cappelli, and Trezicak, Software Engineering Institute
[xx] Web Scarab, OWASP, http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
[xxi] WSFuzzer, OWASP, http://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project
[xxii] wsScanner, Blueinfy, http://blueinfy.com/tools.html
[xxiii] WeakReference, Java API, http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ref/WeakReference.html
[xxiv]Sun Java JRE Deserialization Denial of Service Vulnerability, Secuina Advisories, http://secunia.com/advisories/17478/
[xxv] Effective Java, Second Edition, Bloch, Joshua, Sun, http://java.sun.com/docs/books/effective/
[xxvi] Error Handling, Auditing, and Logging, OWASP, http://www.owasp.org/index.php/Error_Handling%2C_Auditing_and_Logging
[xxvii] Hibernate SQL Injection, OWASP, http://www.owasp.org/index.php/Hibernate-Guidelines#SQL_Injection

[xxviii] Error Handling, Auditing, and Logging, OWASP, http://www.owasp.org/index.php/Error_Handling%2C_Auditing_and_Logging
[xxix] Java Secure Coding Standards, CERT, https://www.securecoding.cert.org/confluence/display/java/FIO34-J.+Ensure+all+resources+are+properly+closed+when+they+are+no+longer+needed
[xxx] Path Traversal Attacks, OWASP, http://www.owasp.org/index.php/Path_Traversal
[xxxi] Using the WebLogic Server Java Utilitlies, WebLogic Server Command Reference, http://e-docs.bea.com/wls/docs81/admin_ref/utils17.html
[xxxii] Web PBE Configuration, Jasypt, http://www.jasypt.org/webconfiguration.html
[xxxiii] WS Security Specification, OASIS, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

[xxxiv] Internet Message Format, IETF, http://tools.ietf.org/html/rfc2822#section-3.4.1
[xxxv] Java Secure Coding Standards, CERT, https://www.securecoding.cert.org/confluence/display/java/FIO34-J.+Ensure+all+resources+are+properly+closed+when+they+are+no+longer+needed
[xxxvi] Path Traversal Attacks, OWASP, http://www.owasp.org/index.php/Path_Traversal
[xxxvii] Using the WebLogic Server Java Utilitlies, WebLogic Server Command Reference, http://e-docs.bea.com/wls/docs81/admin_ref/utils17.html
[xxxviii] Web PBE Configuration, Jasypt, http://www.jasypt.org/webconfiguration.html