Architectural Patterns for Enabling Application Security

Joseph W. Yoder

The Refactory, Inc.
University of Illinois

yoder@refactory.com

http://www.joeyoder.com

Presenter

- Joseph Yoder
  - e-mail: yoder@refactory.com
  - www: http://www.joeyoder.com
  - www.refactory.com

Security Collaborators

- ***Jeffrey Barcalow***
- Eduardo Fernandez
- Peter Sommerlad
- Quince Wilson
- Others…

Table of Contents

- Overview
- Motivation – a few examples
  - General Problem
  - General Solution
- Architectural Elements of Security
- A Description of the Patterns in Action
- Implementation Issues
- Putting it All Together
- Summary and Questions

What are we doing?

- Describing common Architectures for Application Security
- Going to describe them with patterns
- We will look at a set of Patterns that work together to solve issues raised while implementing application security

Security Problems

- Adding application security late in the development cycle can be a very difficult task.
- Sometimes it may even require large pieces of the system to be completely refactored or rewritten.

Context of the Style

- Systems are often developed without security in mind. This omission is primarily because the application programmer is focusing on trying to learn the domain rather than how to protect the system.
- The developer is building prototypes and learning what is needed to satisfy the needs of the users.
- In these cases, security is usually the last thing he or she needs or wants to worry about. When the time arrives to deploy these systems, it quickly becomes apparent that adding security is much harder than just adding a password protected login screen.

Forces – Security Patterns

- Application programmers usually start out focusing more on trying to learn the domain rather than worrying about how to protect the system.
- It is sometimes easier and better to show quick progress to the user, thus we get to get to get feedback and user validation (i.e. XP process).
- On the other hand, application security is an important issue and addressing it later phases of the development lifecycle can make it hard to retrofit into the architecture.

Security Issues

- In corporate environments where security is a priority, detailed security documents are written describing physical, operating system, network, and application security.
- These documents deal with issues such as user privileges, how secure passwords have to be and how often they might need to be changed, whether or not data needs to be encrypted, how secure the communication layer needs to be, etc.

Security Patterns Catalogue

Pattern Name	Intent
Single Access Point	Providing a security module and a way to log into the system.
Check Point	Organizing security checks and their repercussions.
Roles	Organizing users with similar security privileges.
Session	Localizing global information in a multi-user environment.
Full View With Errors	Provide a full view to users, showing exceptions when needed.
Limited View	Allowing users to only see what they have access to.
Secure Access Layer	Integrating application security with low level security.

Single Access Point

Also known as:

- Login Window
- One Way In
- Guard Door
- Validation Screen

Security is easier to guarantee and implement when there is only a single place to gain entry into the system

Single Access Point (context)

- Need to provide external access to a system, and want to protect it from misuse, damage, or unauthorized access.
- Define a single access point that grants or denies entrance to the system after checking the entity requiring access.
- The single access point is easy to implement, defines a clear entry point to the system and can be easily assessed for implementing the desired security policy.

Single Access Point

Problem:

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 to use 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.

Solution:

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

Single Access Point

- Defines clear entry point to the system.
- Provides external access to a system.
- Helps protect system from misuse or damage.
- Defines a single access point that grants or denies entrance to the system.
- Checks the entity requiring access.
- Used for setting up and implementing the desired security policy.

Single Access Point: CRC

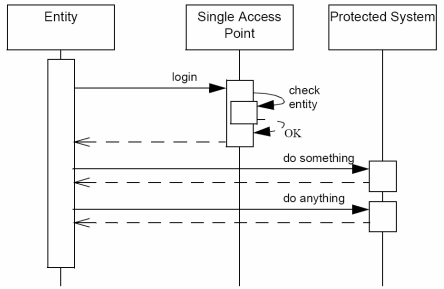
Component Protected System	***Collaborators*** Single Access Point, Entity
Responsibility • provides a service or interaction to entities • has a passive boundary protection only opened at the Single Access Point	

Component Boundary protection	***Collaborators*** Single Access Point, Protected System
Responsibility • provides a passive protection of the system hindering intruding entities to access the protected system from the outside	

Component Single Access Point	***Collaborators*** Entity, System
Responsibility • checks entity before it enters the system • denies unauthorized access to the protected system	

Component Entity	***Collaborators*** System, Single Access Point
Responsibility • requires service from protected system or interacts otherwise with it. •	

Single Access Point: Scenario



Single Access Point:
Implementation Issues

- Define your security policy for the system.
- Define a place for the *Single Access Point*.
- Implement the entry check at the *Single Access Point*.
- Implement the system initialization at the *Single Access Point*.
- Protect the boundary of your system.

Single Access Point:
Positive Consequences

- *Single Access Point* provides a place to properly setup security within the application.
- Control flow can be simpler since everything must go through a single point of responsibility.
- A single place to check for vulnerabilities.
- Inner structure of system is simpler, because repeated authorization checks are avoided.
- No redundant authorization checks. The entity is trusted once it passes through the access point.

Single Access Point:
Negative Consequences

- Application cannot have multiple entry points to make entering an application more flexible.
- Can be difficult to incorporate different types of entry (different info) into single point of access.
- In a complex system several *Single Access Points* of subsystems might be required.
- *Single Access Point* may make the system cumbersome to use or completely unusable.

Single Access Point:
Related Patterns

- Validates the user's login information through a *Check Point* and uses that information to initialize the user's *Roles* and *Security Session.*
- A *Singleton* [GHJV 95] could be used for the login class and a *Singleton* could also be used to keep track of the *Sessions.*

Single Access Point: Known Uses

- *Many- M*ost Operating Systems such as MacOS, Microsoft Windows or UNIX, require a user to log into the system
- Firewalls and Protection Proxies use Single Access Point.
- Most application login screens are a *Single Access Point* as they are the only way to startup and run the application.

Check Point

Also known as:

- Access Verification
- Authentication and Authorization
- Holding off hackers
- Validation and Penalization
- Make the Punishment Fit the Crime

It is important to only allow authorized users to perform their allowed actions. Penalize and block inappropriate actions.

Check Point (context)

- Application security allows occasional mistakes while doing its best to keep a hacker out.
- A developer could design many checks to determine if a user is trying to break into the system or is just making common mistakes.
- Checks could become complicated and could be spread out throughout the application, making it difficult to manage and maintain.
- *Check Point* addresses this problem by organizing these checks.

Check Point

Problem:

An application needs to be secure from break-in attempts, and appropriate actions should be taken when such attempts occur. Different organizations have different security policies and there needs to be a way to incorporate these policies into the system independently from the design of the application.

Forces:

- Having a way to authenticate users and provide validation on what they can do is important.
- Users make mistakes and should not be punished too harshly for mistakes.
- If too many mistakes are made, some type of action needs to be taken.
- Different actions need to be taken depending on the severity of the mistake.
- When error-checking code spread over an application, it is difficult to debug and maintain.

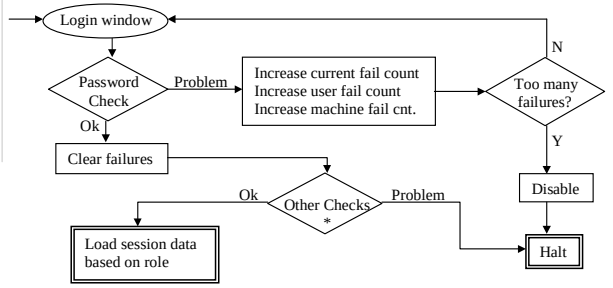
Solution:

Create an object that encapsulates the algorithm for the company's security policy.

Check Point

- Defines clear entry point to the system
- Provide external access to a system
- Need to protect system from misuse or damage
- Define a single access point that grants or denies entrance to the system
- First checks the entity requiring access
- Used for setting up and implementing the desired security policy

Check Point: Example



* These other checks could include: Is the machine legal? Is the machine disabled? Is user's account disabled? Does user have valid role? Has the user's password expired? These other checks are related to the companies security policy.

Check Point:
Implementation Issues

- Pattern is separated into two conceptual parts: Authentication and Authorization.
  - Authentication verifies who and where a user is.
  - Authorization involves checking the privileges of an authenticated user.
- *Check Point* utilizes the user's *Role* to provide the authorization to the system.
- *Check Point* can be an Achilles' Heel of an application's security, so every branch in the logic must be carefully checked.

Check Point:
Positive Consequences

- *Check Point* is a critical location where security must be absolutely enforced. *Check Point* localizes the security model that needs to be certified.
- *Check Point* can be a complex algorithm. While this complexity may be unavoidable, it is isolated in one location, making the security algorithm easier to change.

Check Point:
Negative Consequences

- Some security checks might not be able to be done at startup, so *Check Point* must have a secondary interface for parts of the application which need those checks.
- Some information needed for further security checks must be kept until needed. This information could include username, password, and *Roles*, (store in a *Session)*.

Check Point:
Related Patterns

- The *Check Point* algorithm can be a *Strategy*.
- *Single Access Point* is used to insure that *Check Point* gets initialized correctly and that no security checks are skipped.
- *Roles* are used for *Check Point's* security checks and initialized by *Check Point*.
- *Check Point* configures a *Session* and stores the necessary security information.
- *Check Point* uses the *Secure Access Layer* to interface with external security systems.

Check Point:
Known Uses

- Most Operating Systems have *Check Points* for accessing files and running programs (used with ACLs).
- The login process for an ftp server uses *Check Point*. Depending on the server's configuration files, anonymous logins may or may not be allowed. For anonymous logins, a valid email is sometimes required.
- Most applications use *Check Point* as users' access different parts of the system.

Roles

Also known as:

- Actors
- Groups
- Projects
- Profiles
- Jobs
- User Types

Organizing groups of users by actions they are allowed to perform.

Roles (context)

- Security can be complicated in multi-user applications with different privileges.
- Users have different areas of the application that they can see, can change, and "own."
- When the number of users is large, the security permissions for users often fall into several categories such as a user's job titles, experience, or division.
- An administrator needs an easier way to manage permissions and the security profiles for a large number of users.

Roles

Problem:

Users have different security profiles, and some profiles are similar. If the user base is large enough or the security profiles are complex enough, then managing user-privilege relationships can become difficult.

Forces:

- With a large number of users it is hard to customize security for each person.
- Groups of users usually share similar security profiles.
- A user may need to have an individual security profile.
- Security profiles may overlap.
- A user's security profile may change over time.

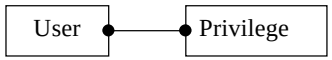
Solution:

Create one or more role objects that define the permissions and access rights that groups of users have.

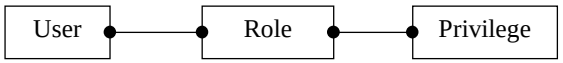
Roles

- This pattern introduces a level of indirection (called the *Role*).
- This level of indirection splits the user-privilege relationship into user-role and role-privilege relationships.
- While these two new relationships are still M-to-N, selecting appropriate *Roles* can reduce the total number of relationships.
- The benefit is that privileges can usually be grouped together into common categories.

Roles: Example



User-Privilege Relationship



User-Role-Privilege Relationship

Roles:
Implementation Issues

- Introducing *Roles* creates two new relationships that must be managed: user-role and role-privilege.
  - These new relationships can help make managing security easier. When the privileges of a job title change, that role-privilege relationship can be edited directly.
- Sometimes, a subset of the original user-privilege relationship must also be maintained to allow each user to have private privileges.
  - The easiest way to do this is to give each user an independent role, which happens to be the same as their username. *Roles* should only be used when the extra level of indirection provides a conceptual or manageability advantage over the direct user-privilege relationship.

Roles:
Positive Consequences

- Instead of managing user-privilege relationships, the administrator will manage the user-role and role-privilege relationships.
- *Roles* can be a convenient organizational technique for administrators.
- *Roles* are a good way to group together common privileges.
- Administrative tasks can be simplified by using *Roles*. For example, all new employees could be allowed to view and edit a training database, but only view the real database. A "training" *Role* could be created for these permissions. Then, any new employee account will only have to be given a training *Role* instead of a potentially large set of permission options.

Roles:
Negative Consequences

- *Roles* add an extra layer of complexity for developers.
- Even if *Roles* are used, each user will need a private *Role* to maintain private privileges and preferences.

Roles:
Related Patterns

- *Dealing with Roles* [Fowler 97] and the *Role Object* Pattern the provides discusses roles with specific implementation details.
- *Check Point* used validation *Roles* with the proper permissions.
- The *Role* information can be stored in a *Session* object for access whenever needed.
- *Roles* could be used to determine the scope of a *Limited View* or a *Full View with Errors*.
- When an application should behave differently depending on a user's job, the user's *Roles* could be a *Strategy* for the application.

Roles:
Known Uses

- UNIX uses three classifications for secure access to files and directories. The middle classification is "group," which is an example of *Roles*. The user-role relationship is stored in /etc/group and is sorted by *Roles*. The file system stores the role-privilege relationship and uses ACLs.
- Windows NT allows for descriptions of groups for allocating privileges for users in a similar way.
- Some web servers use .htaccess and .htgroups files which define groups of users (*Roles*) that can access certain areas of a web site.
- Oracle has a *Roles* feature for defining security privileges. User-role and role-privilege relationships are stored in tables.

Roles:
More Known Uses

- In the GemStone OODB data is stored in a segment. GemStone treats segments analogously to the way UNIX treats files. Users in GemStone can have one or more groups (*Roles*), and each segment has read and write privileges defined for all users, for a set of groups, and for the owner. Since a segment can have a set of groups, it is a little more powerful than UNIX with respect to groups.
- The PLoP '98 registration program [Yoder & Manolescu 98] has two *Roles*: attendee and administrator.
- Java's Principal object can be to store *Roles* which are just strings. The Access API for Reuters SSL Developers Kit - Java Edition has an Attribute class which is analogous to Java's Principal.

Security Session

Also known as:

- User's Environment
- Namespace
- Threaded-based Singleton
- Localized Globals

Secure applications need to keep track of information used throughout the application such as username, roles, and their respective privileges.

Session (context)

- Secure applications need to keep track of information used throughout the application such as username, roles, and their respective privileges.
- Usually the *Singleton* pattern [GHJV 95] is used to store global information in a static or class variable.
- *Singleton* can be difficult to use when an application is multi-threaded, multi-user, or distributed.
- Each thread or each distributed process can be viewed as an independent application, each needing its own private *Singleton*.
- When the applications share a common global address space, the single global *Singleton* cannot be shared.
- Thread safe "*Singletons*".

Session

Problem:

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

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.

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.

Session

- Verification of a users identity and access rights for every system function can be tedious. To not annoy users by re-requesting authentication information over and over systems establish a Security Session for keeping track of who is using them and what the corresponding access rights are.
- Instead of passing all information around, only a unique reference to the session object is passed and all queries regarding a user's security properties are delegated to the attached session object.

Session: CRC

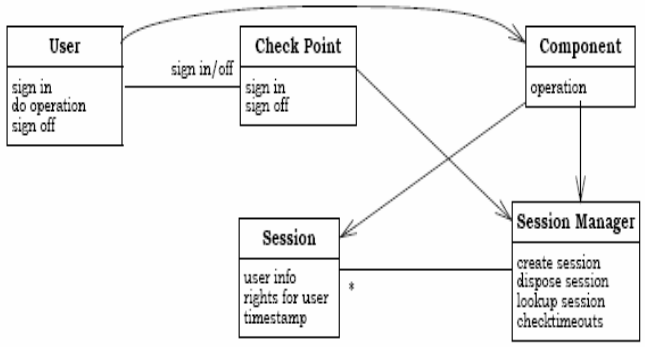
Component Session	*Collaborators* Check Point, Session Manager
Responsibility • keeps user related (security) data • optionally times last access by user	

Component Session Manager	*Collaborators* Check Point, Session
Responsibility • keeps track of instantiated sessions • maps (external) session identifiers to session objects • collects obsolete sessions	

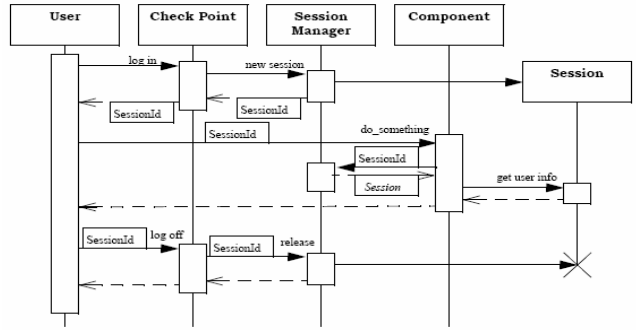
Component Check Point	*Collaborators* User (not shown), Session Manager
Responsibility • creates new sessions when user logs in • populates initial session data for user, e.g., credentials • retires session if user signs off	

Component System Component	*Collaborators* Session
Responsibility • retrieves user info from session object • checks user credentials via info stored in session • stores additional user data into session object for later use	

Session: Structure



Session: Dynamics



Session:
Positive Consequences

- 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.

Session:
Negative Consequences

- While an object may not need a *Session*, it may later create an object that needs the *Session*.
- Potential proliferation of *Session* instance variables.
- Adding *Session* late in the development process can be difficult. The authors have experience retrofitting *Session* and can attest that this can very tedious when *Singletons* are spread among several classes.
- 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.

Session:
Related Patterns

- *Session* is an alternative to a *Singleton* in a multi-threaded, multi-user, or distributed environment.
- *Single Access Point* validates a user through *Check Point*. It gets a *Session* in return if the user validation is acceptable.
- A *Session* is a convenient place to implement the State pattern when the state is needed throughout the application.
- A *Session* can keep track of the users *Role* and possibly cache Limited View data.

Session:
More Related Patterns

- *Abstract Session* [Pryce 97] is a related pattern. While *Abstract Session* concentrates on network session, this pattern concentrates on where data is stored. In a networking environment, both patterns are typically seen and a could be implemented together.
- The *Thread Specific Storage Pattern* [Schmidt, Pryce, & Harrison 97] allows multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead.
- *Double-Checked Locking* [Schmidt & Harrison 97] is the *Singleton* replacement when dealing with multitasking or parallelism.

Session:
Known Uses

- For VisualWorks, GemBuilder for GemStone and the ObjectLens framework for Oracle have GbsSession and OracleSession classes, respectively. Each stores information such as the transaction state and the database connection. These Sessions are then referenced by any object within the same database context.
- The PLoP '98 registration program [Yoder & Manolescu 98] has a Session object that keeps track of the user's global information.
- Most databases use a Session for keeping track of user information.
- VisualWave uses a Session for its httpd service, which keeps track of any web requests made to it.
- UNIX ftp and telnet services use a Session for tracking requests and restricting user actions.
- The open sourced implementation of SSL (secure sockets layer) openSSL uses a session id, to avoid re-negotiating certificates and encryption algorithm and session key for connections re-established between the same client and server.

Full View With Errors

Also known as:

- Full Access With Errors
- Full Access with Exceptions
- Reveal All and Handle Exceptions
- Notified View

Provides a view of the maximal functionality of the system and gives the user an error when not entitled to use a presented function.

Full View With Errors (context)

- Graphical applications often provide many ways to view data.
- Users can dynamically choose which view on which data they want.
- When an application has multiple views, the developer needs to be concerned with which operations are legal given the current state of the application and the privileges of the user.

Full View With Errors

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 permission denied messages and illegal operation errors.

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.

Full View With Errors

- Defines clear entry point to the system.
- Provides external access to a system.
- Helps protect system from misuse or damage.
- Defines a single access point that grants or denies entrance to the system.
- Checks the entity requiring access.
- Used for setting up and implementing the desired security policy.

Full View With Errors: CRC (1)

Component User	*Collaborators* Interface
Responsibility • uses a system via its interface	

Component Access Rights	*Collaborators*
Responsibility • passive representation of a user's access rights typically associated at a Check Point.	

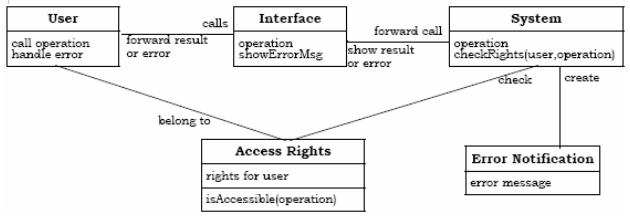
Full View With Errors: CRC (2)

Component Interface	*Collaborators* User System Error Notification
Responsibility • presents all of system's functionality to user and calls it on activation • presents error notification in case of insufficent access rights • presents system's regular results	

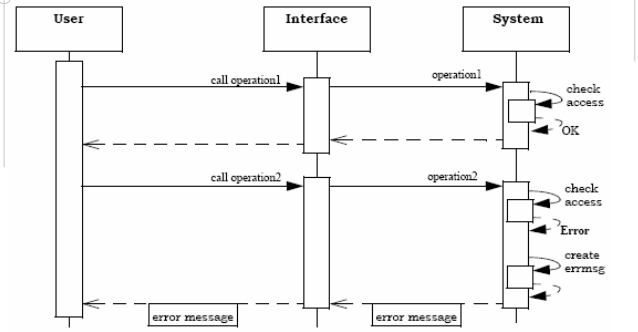
Component System	*Collaborators* Access Rights Interface
Responsibility • checks access rights of user • triggers Interface to raise error to user, when insufficient rights • perform function if permitted	

Component Error Notification	*Collaborators*
Responsibility • notifies user of inaccesible system function • can provide a means of remedy to the error situation	

Full View With Errors: Structure



Full View With Errors: Scenario



Full View With Errors: Implementation Issues

- To implement *Full Access with Errors* several tasks need to be done:
  - User log in - refer to *CheckPoint* and *Security Session*, obtain access rights, Interface design, etc.
- *Implement association of access rights with user.* The patterns *CheckPoint* and *Security Session* are typical means to provide a user log in and attach his access rights.

Full View With Errors: More Implementation Issues

- *Design the interface representing the full set of the system's functionality.* visual hints for users (requires registration), grouping, etc.
- Provide access to user's access rights check for system's functions. *Check Point* again.
- Define and implement the trust relationship within the system by the *Session* concept.
- Some more…

Full View With Errors: Positive Consequences

- A system can be effectively secured, since before each individual operation is executed a user's permissions for this operation are checked.
- All possible functions are visible to a user, providing not only a consistent interface but also demonstrate all available features, even when the user is not (yet) privileged to use them.
- It is easy to change access rights and groups for such a system without influencing the concrete implementation of the system or its interface.

Full View With Errors: More Positive Consequences

- Retro-fitting this pattern into an existing system is straight forward: Write an interface that will handle all possible functions and whenever a problem happens with an operation simply abort the operation and display an error message.
- Documentation and training material for an application can be consistent for each type of user.
- Full Access with Errors fits well in situations, where users can upgrade their privileges for a otherwise unavailable operation on the fly, e.g. by confirming a dialogue, without breaking their flow of work.
- For web applications applying the pattern allows stable URLs and links to a download area, even in the case a user must register first. A pre-registered user will be able to directly download using the same URL.

Full View With Errors: Negative Consequences

- Users may get confused with a constant barrage of error dialogs.
- Operation validation can be more difficult when users can perform any operation.
- Users will get frustrated when they see options that they cannot perform.

Full View With Errors: Related Patterns

- *Limited View* is a competitor to this pattern. If limiting the view completely is not possible, this pattern can fill in the holes.
- *Checks* [Cunningham 95] describes many details on implementing GUI's and where to put the error notifications.
- *Roles* will be used for the error notification or validating what the user can and can not do.

Full View With Errors:
Known Uses

- Under the UNIX shell you can activate almost any program on any file in the file system. However, when your access rights are insufficient for accessing files, you get a message saying permission denied. Only the dedicate super user "root" is not protected from careless calling programs or overwriting files, it gets access to everything overriding all access rights set.
- Oracle's SQLPlus interactive database access language allows you to execute any syntactically valid SQL statement and displays an appropriate error message if illegal access is attempted.
- 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.

Limited Access

Also known as:

- Limited View
- Blinders
- Child Proofing
- Invisible Road Blocks
- Hiding the cookie jars
- Early Authorization

Limited Access guides a developer to just presenting the currently available functions to a user and hiding anything that they lacks permission to perform.

Limited Access (context)

- Graphical applications often provide many ways to view data.
- Users can dynamically choose which view on which data they want.
- When an application has multiple views, the developer needs to 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 limiting the view to what the user has access to, conditional code can be eliminated.

Limited Access

Problem:

Users should not be allowed to perform illegal operations.

Forces:

- Users may be confused when some options are either not present or disabled.
- Users should not be able to see operations they cannot 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 permission denied messages and illegal operation errors.
- User validation can be easier when you limit the user to see only what they can access.
- If options pop in and out depending upon *Roles*, the user may get confused on what is available.

Solution:

Only let the users see what they have access to; only give them selections and menus to options that their current access-privileges permit.

Limited Access

- You are designing the (user-) interface of a system where access restrictions such as user authorization to parts of the interface apply.
- While most applications of this pattern are within the domain of graphical user interfaces (GUI), it can also apply to other interface types as well.

Limited Access: CRC

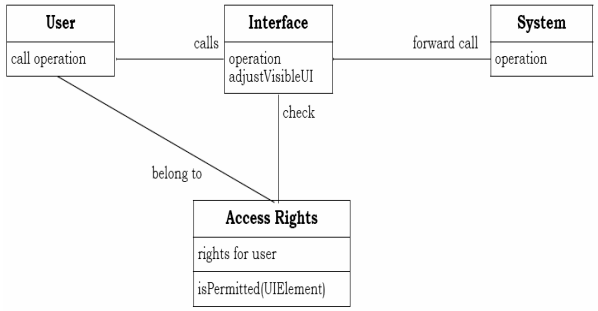
Component User	*Collaborators* Interface
Responsibility • uses a system via its interface	

Component Access Rights	*Collaborators*
Responsibility • passive representation of a user's access rights typically associated at a Check Point.	

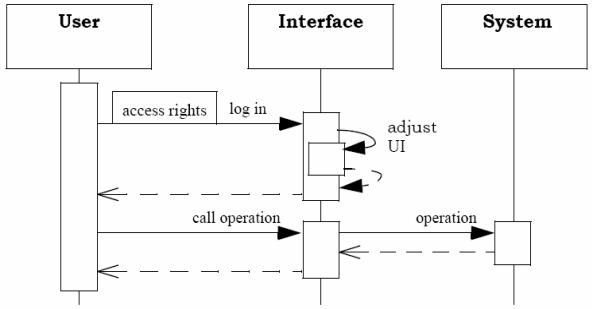
Component Interface	*Collaborators* User System Access Rights
Responsibility • presents accessible system's functionality to user • creates or defines presentation according to user's access rights • performs called function	

Component System	*Collaborators* Interface
Responsibility • perform function on behalf of interface and user respectively.	

Limited Access: Structure



Limited Access: Scenario



Limited Access: Implementation Issues

- A *Limited View* configures which selection choices are possible for the user based upon the current set of roles. This ensures that the user only selects data they are allowed to see.
- A *Limited View* takes the current *Session* with the user's *Roles*, applies the current state of the application, and dynamically builds a GUI that limits the view based upon these attributes.
- *Null Objects* [Woolf 97] can be used for values the user does not have permission to view.

Limited Access: More Implementation Issues

- When a user starts the system a mechanism authenticates her and associates a *Security Session*, typically within a *Checkpoint* architecture.
- The *Security Session* object caches the current privileges of the user that can be used by the GUI implementation to decide what functions and data are permissible and should be presented to the user.
- The interface of the system checks the access rights before presenting itself to the user. Only that functionality that is available to the user is rendered…hiding menus, buttons, etc.

Limited Access: Positive Consequences

- By only allowing the user to see and edit what he or she can access, the developer doesn't have to worry about verifying each operation after a user selects it.
- Security checks can be simplified by performing all of them up front.
- Users will not get frustrated with error dialogs popping up all the time telling them what they can not do. Users will also not get frustrated by constantly seeing options they do not have access to.

Limited Access: Negative Consequences

- Users can become frustrated when options appear and disappear on the screen. For example, if when viewing one set of data, the editing button is there and when viewing another set of data, it disappears, the user may wonder if something is wrong with the application or why the data isn't available.
- Training materials for an application must be customized for each set of users because menu operations will disappear and reappear and GUIs will change based on the *Limited View*.
- Retrofitting a *Limited View* into an existing system can be difficult because the data for the *Limited View*, as well as the code for selecting it, could be spread throughout the system.

Limited Access:
Related Patterns

- *Full View With Errors* is a competitor to this pattern. If limiting the view completely is not possible, error messages can fill in the holes.
- A *Session* may have a *Limited View* of data that it distributes throughout the application.
- *Roles* are sometimes used to configure a *Limited View*.
- *State* or *Strategy* can be used to implement a *Limited View*.

Limited Access:
More Related Patterns

- *Composites* and *Builders* can be used to create GUIs for a *Limited View*.
- *Null Objects* can be used in places where a view has been limited.
- *Metadata* and *Adaptive Object-Models* can be used to configure what parts of a view need to be limited.
- *Checks* [Cunningham 95] describes many details on implementing GUI's and where to put the error notifications.

Limited Access:
Known Uses

- Firewalls provide *Limited Views* on data by filtering network data and filtering services available.
- Web servers provide a Limited View by only allowing users to view directories in the root web directory and in users' public_html directories.
- Most operating systems provide hidden files and directories, which are forms of *Limited Views*.
- Microsoft's Windows NT provides *Limited Views* based upon a user's role. Users only see files that they have permission to see, and they get customized menus based upon those roles.
- The PLoP '98 registration program provides *Limited Views* for the administrator and for those registering for PLoP. People registering for PLoP get a *Limited View* to view and edit only their information.

Full Access vs. Limited View

- *Limited View* can simplify security checks by performing them at login and by dynamically building the appropriate views. Application logic will have a simpler design because it does not have to handle security exceptions throughout the code.
- *Limited View* can be more user friendly since users do not get barraged with error dialogs. However, the application code that presents views and options to the users becomes more difficult to implement, as the system must dynamically build these views based upon the privileges of the user.
- *Full View With Errors* is easier to implement. It is as simple as performing a security check when an operation is executed and opening an error dialog or printing to standard error when a violation occurs. It also can be used when a security check cannot be performed up front because it is time-consuming or all necessary information is not yet known.

Full Access vs. Limited View

- Full Access is best for when an application has a small number of security checks to perform on user operations,
- *Full View With Errors* is also better when most options are the same independent of the user's roles.
- If security exception handling is spread throughout the application and there are many variations of security privileges depending upon the users *Role*, *Limited View* should be used.
- *Limited View* and *Full View With Errors* are competing patterns for individual security checks, they can both be used by an application to provide a "*Limited View With Minimal Errors*."

Secure Access Layer

Also known as:

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

Limited Access guides a developer to just presenting the currently available functions to a user and hiding anything that they lacks permission to perform.

Secure Access Layer (context)

- 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 communication with other systems happens, then the code could become very convoluted.
- An application that is built on an insecure foundation will be insecure. In other words, it doesn't do any good to lock your door when you leave the key on the front porch.

Secure Access Layer

Problem:

An application will be insecure if it is not properly integrated with the security of the external systems it uses.

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.

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.

Secure Access 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.
- This layer will have a protocol for accessing secure data in other systems and will send the required information needed by the external systems.
- The crux of this pattern is to separate external protocols so they can be more easily secured.
- The architecture for different *Secure Access Layers* could vary greatly.

"Building on top of SSL is an example"

Secure Access Layer: Implementation Issues

- Create a *Secure Access Layer* with a standard set of protocols for communicating with the outside world.
- Communication in and out of the application will pass through the protocols provided by this layer.
- This pattern assumes a convenient abstraction is possible.
- *Secure Access Layer* provides a location for a more general abstraction.

Secure Access Layer: Positive Consequences

- A Secure Access Layer can help isolate where an application communicates with external security systems. Isolating secure access points makes it easier to integrate new security components and upgrade existing ones.
- 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.

Secure Access Layer: Negative Consequences

- 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.

Secure Access Layer:
Related Patterns

- *Secure Access Layer* is part of a layered architecture. *Layers* [BMRSS 96] discusses the details of building layered architectures.
- *Layered Architecture for Information Systems* [Fowlers 97-1] discusses implementation details that can be applied when developing layered systems.
- *Check Point* in conjunction with *Roles* can be used in the *Secure Access Layer* to ensure only authorized access.
- Session information might be used to pass global information to a Secure Access Layer.

Secure Access Layer:
Known Uses

- Secure Shell [SSH] includes secure protocols for communicating in X11 sessions and can use RSA encryption through TCP/IP connections.
- Netscape Server's Secure Socket Layer (SSL) 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 [OMG] 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.

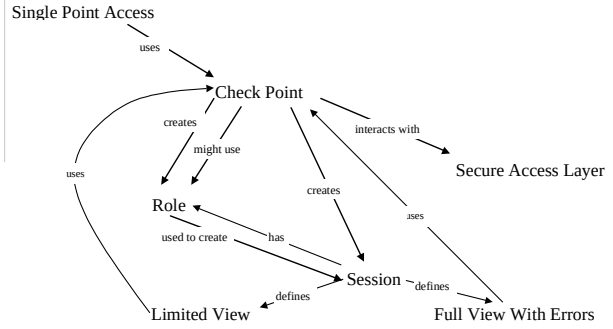
Secure Access Layer:
More Known Uses

- The Caterpillar/NCSA Financial Model Framework [Yoder] uses a *Secure Access Layer* provided by the LensSession in VisualWorks Smalltalk.
- The PLoP '98 registration program [Yoder & Manolescu 98] goes through a *Secure Layer* for access to the system.
- The Access API used by the Reuters SSL Developers Kit – Java Edition uses a DACSPrincipal object to interact with the Data Access Control System (DACS), an entitlements system for controlling access to Reuters' market data.

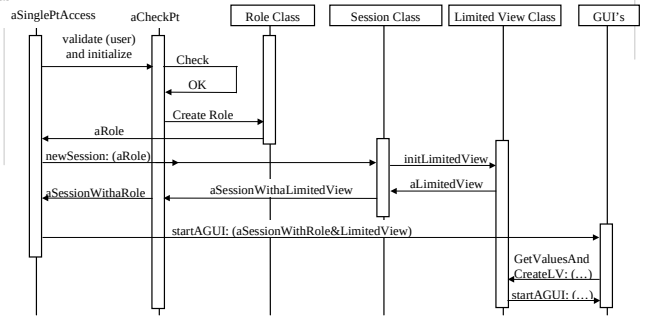
Problems with Retrofitting

- *Secure Access Layer*, *Session*, and *Limited View*, can be very difficult to retrofit into a system that was developed without security in mind.
- If a *Single Access Point* is created up front, it is fairly straightforward to add *Check Point* later.
- Since *Roles* are used to define a *Session* and are set up during *Check Point*, additional *Roles* can easily be added later.
- A dummy *Check Point* can start out as just a placeholder and the details of the corporate security policy can be added later. Also, if all outside requests are forwarded through some form of a *Secure Access Layer*, it will be easy to enhance and abstract the *Secure Access Layer* at a later point.

Pattern Interaction Diagram



Class Collaboration Diagram



Where to Find More Information

- http://www.joeyoder.com/papers/patterns
- http://hillside.net
- http://www.refactory.com
- http://www.joeyoder.com
- http://securitypatterns.org

That's All