9.3 Check Point

Once you have secured a system using SINGLE ACCESS POINT (279), a means of identification and authentication (I&A) and response to unauthorized break-in attempts is required for securing the system. CHECK POINT (287) makes such an effective I&A and access control mechanism easy to deploy and evolve.

Also Known As

Policy Definition Point (PDP), Policy Enforcement Point (PEP), Access Verification, Holding off hackers, Validation and Penalization, Make the Punishment fit the Crime, Validation Screen, Pluggable Authentication.

Example

The mayor of our medieval town that established SINGLE ACCESS POINT (279) with their gate and guard is concerned about their protection during times when different threats come close to the town. For example, the merchants would like to have the gate freely open during daytime, to let traders in and out easily. However, they are concerned about burglars sneaking into their warehouses during night time.

Context

You have a system protected from unauthorized access in general, for example by applying SINGLE ACCESS POINT (279). Nevertheless, you want authorized clients be able to enter your system.

Problem

Whenever you introduce a security measure, you often do not know in advance about all its weaknesses. Also, you only learn how it influences usability if you deploy it to actual users.

A protected system needs to be secure from break-in attempts, and appropriate actions should be taken when such attempts occur. On the other hand, authorized clients should still be able to enter the protected system, and should not be impeded too much when they (in the case of a human) make a mistake when providing their credentials.

In addition, you want to consider the change of requirements for identification and authorization (I&A) that might occur over time, either because you need to address new threats, or because you learn from its use. One example for handling that situation is the development of a protected system in which a developer will use a dummy I&A implementation to test the system, without the hassle of logging into the system for every test. Later on, the deployed system needs to be protected by a log-in mechanism that authenticates and authorizes its users.

How can you provide an architecture that allows you to effectively protect system access while still being able to tune I&A to evolving needs without impact to the system you protect?

The solution to this problem must resolve the following forces:

- Having a way to authenticate users and provide validation about what they can do is important.
- Human users make mistakes and should not be punished too harshly for them. However, too many consecutive mistakes at authentication by a user can indicate an attack to the system and should be dealt with.
- Different actions need to be taken depending on the severity of the mistake and current context.
- Spreading checks throughout your protected system increase complexity and make it hard to change. It would be helpful to have a single place to which to refer for authentication and authorization of users.

- You might learn better ways and techniques for I&A after you have deployed an initial system, you might have to change your system after you recognized that it is vulnerable to specific attacks, or you might have to modify the protection because your risks have changed.
- Security-providing code is critical and requires thorough validation through reviews and tests. The smaller such components, the easier are these validations. Reuse of well-proven security components minimizes expensive validations.

Solution

Apply the STRATEGY design pattern [GoF95] to vary the checking behavior at the SINGLE ACCESS POINT (279). CHECK POINT (287) defines the interface to be supported by concrete implementations to provide the I&A service to the SINGLE ACCESS POINT (279). A separate configuration (mechanism) defines which concrete implementation of the CHECK POINT (287) interface to use.

The check point interface might provide further security-related functionality in addition to performing I&A. For example, it might define hooks for creating a SECURITY SESSION (297), checking access rights for a user or session by other system components, logging security-related information, or detecting attack patterns when unauthenticated access attempts occur.

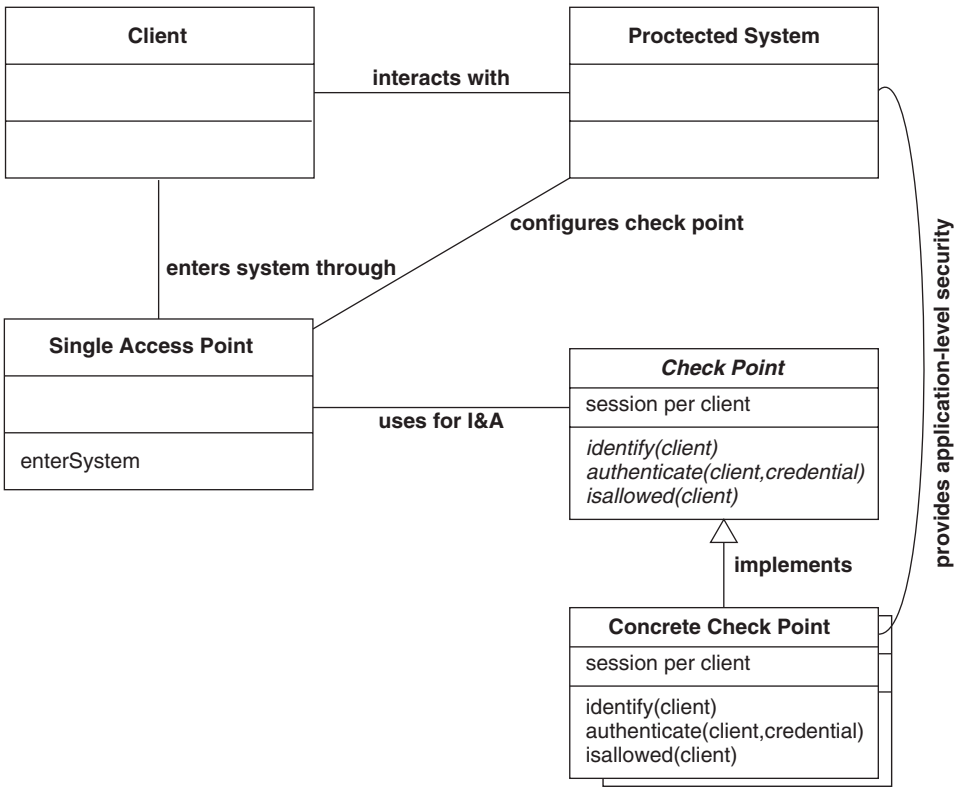
By changing the configuration and thus the concrete CHECK POINT (287) implementation, the behavior at the SINGLE ACCESS POINT (279) changes. For example, Linux provides pluggable authentication modules (PAM) allowing the source of user identities and passwords to be changed [LinuxPAM]. PAM defines a module interface and a configuration mechanism in `/etc/pam.d` that allows system administrators to adapt the authentication mechanism easily by exchanging the corresponding modules.

The check point effectively encapsulates the security policy to be applied. This allows the development of systems to be independent of a concrete security policy, which might not be available during development. It also allows for easier later adaptation of the security policy of a system whenever external pressure or better knowledge require it.

If not all security decisions can be made at the time of passing the single access point, CHECK POINT (287) should supply an interface to be used later on by the system's applications. This interface can be used to determine application-specific access rights that might rely on values of application variables not within the scope of the initial access control at the single access point. For example, a bank's application might allow posting of transactions up to $10,000 for all internal users and up to $1,000,000 for managers, and require a director to acknowledge higher-valued transactions. Hard-wiring such decisions within the applications would hinder the evolution of the security policy.

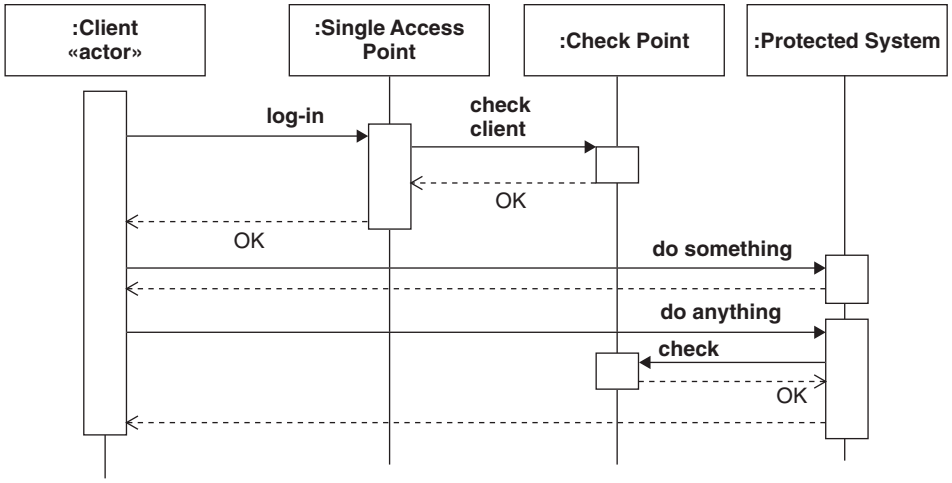
Structure

The following structure shows the elements of CHECK POINT (287):



Dynamics

The scenario shows how a SINGLE ACCESS POINT (279) employs a CHECK POINT (287) implementation to identify, authenticate and authorize a client. The potential creation of a SECURITY SESSION (297) for a client successfully logged in is not shown.



Implementation

To implement CHECK POINT (287), several tasks need to be done:

1. *Define or re-use an interface to be used by your* CHECK POINT (287) *components*. If implemented in an object-oriented language, this can be an abstract class or an interface. Other languages or implementation techniques might require different means appropriate for the chosen technology. For example, Linux PAM uses a object module with pre-defined function entry points in a table for different operations supported by PAM (authentication, access control, session management, password management).

   This CHECK POINT (287) interface corresponds to the abstract strategy in STRATEGY [GoF95]. The interface will provide hooks for I&A, authorization, handling unsuccessful attempts.

2. *Implement the entry check at the single access point*. A single access point ensures that CHECK POINT (287) is initialized and used correctly, and cannot be bypassed by intruders. SINGLE ACCESS POINT (279) usually calls CHECK POINT (287), providing a client's identification and the authentication information they provided. On successful authentication CHECK POINT (287) establishes SECURITY SESSION (297) for the client. If ROLE-BASED ACCESS CONTROL (249) is used, the SECURITY SESSION (297) gets initialized with the client's valid roles.

3. *Provide a configuration mechanism to select a concrete* CHECK POINT (287) *implementation*. To make it easy to adjust a system to use a different security policy, and thus a different concrete CHECK POINT (287) implementation, provide a means to configure it. This configuration mechanism must be protected as well, since changing the configuration effectively changes the security policy. Some implementations provide simple configuration files to be maintained by a privileged user.

   If different concrete CHECK POINT (287) implementations implement different parts of the CHECK POINT (287) interface, it can be handy to be able to combine these concrete CHECK POINT (287)s in a CHAIN OF RESPONSIBILITY [GoF95]. For example, if local Unix users should be authenticated by a system as well as users stored in a corporate LDAP directory, one can implement two concrete CHECK POINT (287)s, one accessing the `/etc/passwd` file and one accessing an LDAP directory. By configuring them to be applied one after the other and allowing access if either succeeds allows local users to log in as well as those in the directory. You can also change the configuration so that both checkpoints must be passed successfully. That way, only users that are registered both locally and in the corporate LDAP directory can pass. Such a change of the policy is possible without changing the code of any of the concrete check points—however, this is at the expense of increased configuration complexity.

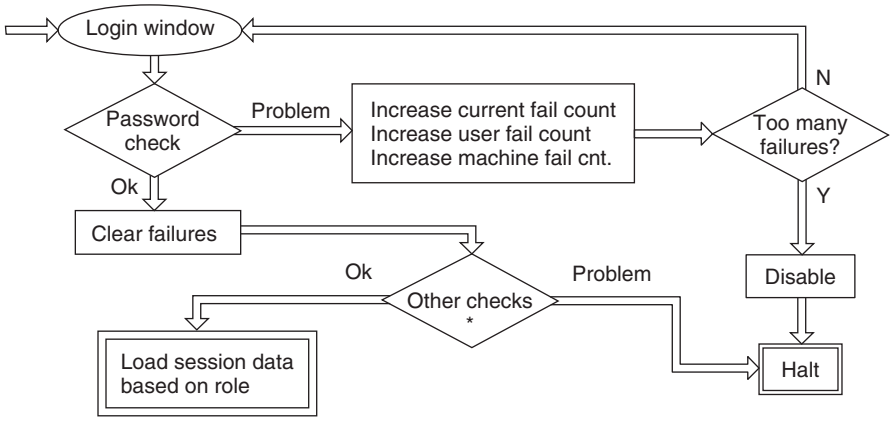
4. *Implement required concrete* CHECK POINT (287). At least one concrete CHECK POINT (287) implementation is needed. More than one makes it useful for different use scenarios, or their combination by configuration in the system. For example, you can apply the NULL OBJECT pattern [Woolf96] to implement a CHECK POINT (287) that is always successful, allowing easier testing during development. A regular concrete CHECK POINT (287) will definitely authenticate clients accessing a system. Usually it stores the client's identification in a SECURITY SESSION (297) object. If ROLE-BASED ACCESS CONTROL (249) is used, the concrete CHECK POINT (287) initializes the session object with the corresponding role set of the client.

5. *Dealing with client errors by the check point*. Depending on the security violation or error, different types of failure actions may be taken. Failure actions can be broken down by level of severity. These types of failures and actions are contingent upon the security policy you are implementing. For example, the simplest action is to return a warning or error message to the user. If the error is non-critical, the security algorithm could treat it as a warning and continue. A second level of failure could force the user to start over. The next level of severity could force an abort of the log-in process or quit the program. The highest level of severity could lock out a machine or user name. In this case, an administrator might have to reset the user name and/or machine access. Unfortunately this could cause problems when a legitimate user tries to log in later, so if the violation is not extremely critical, the user name and

machine-disabled flags could be time-stamped and automatically re-enabled after an hour or so. All security failures could also be logged.

Sometimes, the level of severity of a security violation depends on how many times the violation is repeated. A user who types a password incorrectly once or twice should not be punished too harshly. Three or four consecutive failures could indicate that a hacker is trying to guess a password. To handle this situation, STRATEGY can include counters to keep track of the frequency of security violations and parameterize the algorithm.

The following diagram shows an example for such an algorithm [YB97].



* These other checks could include: Is the machine legal? Is the machne 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.

6. *Provide application level API to check point*. If some security checks cannot be performed in the check point as the user enters the system, they must be deferred until later. For these cases, the check point could have a secondary interface for application components to use, or a separate authorization component will be required.

   Because a consistent security concept is difficult to achieve in an application, it may be desirable to try to make a reusable security module for use in several applications. That goal is difficult when security requirements vary between the applications. All application teams will need to be involved to ensure the framework will satisfy each application's requirements.

   One approach to reusing security code is to create a library of pluggable security components and a framework for incorporating these components into

applications. However, the algorithm for putting together components will almost always be overridden, making the framework difficult to generalize. Another approach is to use the configuration mechanism explained above to allow small parts of the security algorithm to be combined.

Example Resolved

The growing medieval town does not yet know the best security policy to use at the gate that is its single access point, but they learn that a single one is insufficient for all their needs at different times.

The mayor decides to provide a more flexible policy at the city gate. During day time, the gate remains open, while the day-time guard observes the traffic and picks out suspicious-looking people and interrogates them to find out their objectives. During night time the gate remains strictly closed and observed by well-armed night guards. Only town dwellers are let in or out during night time.

The single gate with a newly-built watch house attached to it allows the town leaders to provide a more flexible security policy at their gate, and to change it if harder times require better protection, or prosperous times require easier access.

Known Uses

There are numerous systems and applications that implement CHECK POINT (287).

PAM [LinuxPAM] implements CHECK POINT (287). It allows different modules to implement different user authentication strategies. In addition, it allows different applications to be configured using different modules. Once a new technology for user authentication becomes available, for example storing user information in a new kind of database, a new corresponding PAM module allows this technology to be used immediately by all PAM-aware applications.

The Apache Web server implements CHECK POINT (287) with CHAIN OF RESPONSIBILITY within its modular extension mechanism. Extension modules get the chance to validate each HTTP request according to a configured and implicit activation sequence. Modules might reject a request (that is, its URL), modify it, or allow it for further processing.

The log-in process for an FTP server uses CHECK POINT (287). Depending on the server's configuration files, anonymous log-ins may or may not be allowed. For anonymous log-ins, a valid e-mail is sometimes required. This is similar for Telnet. Linux versions of these applications rely on PAM today.

Xauth uses a cookie to provide a CHECK POINT (287) that X-Windows applications can use for securely communicating between clients and servers.

A Swiss bank uses CHECK POINT (287) based on a CORBA interface throughout all their application systems. In addition to variation of access control by different implementations of the interface, they also allow variation by changing a corporate configuration of user roles, organizational structure, and access rights.

Consequences

The following benefits may be expected from applying this pattern:

- *Concentrate implementation of a security policy.* All aspects of a security policy are implemented in a single place and are thus easily accessible for assessment.
- *Flexibility in security policy.* The common interface to be used for CHECK POINT (287) allows for easy exchange of a concrete implementation if required.
- *Easier testing and development.* Applying a null CHECK POINT (287) allows more efficient testing and development without the need to provide correct user credentials for every run.
- *Independent testing of security policy implementation.* CHECK POINT (287) implementations can be tested independently of their surrounding system, allowing testing of this component more thoroughly than would be economic for the integrated system.
- *Reuse of security components.* Applying CHAIN OF RESPONSIBILITY by configuration of concrete CHECK POINT (287) implementations allows for reuse of these components in different contexts or combinations, effectively providing different security policies with a single code base.

The following potential liabilities may arise from applying this pattern:

- *Criticality.* Concrete CHECK POINT (287) implementations also localize critical sections. Vulnerabilities contained within concrete CHECK POINT (287)s can severely undermine security. Thus concrete CHECK POINT (287) implementations must be validated thoroughly.
- *Algorithm complexity.* Dealing with invalid access attempts and detecting malicious users can require complex algorithms. While this complexity is unavoidable, CHECK POINT (287) at least concentrates it in a single defined location.
- *State complexity.* Some security checks cannot be done at start-up. CHECK POINT (287) must have a secondary interface for parts of applications that require such checks. Usually the necessary information is already collected at log-in of a client and stored in its SECURITY SESSION (297) for reuse by these later checks.
- *Interface complexity.* Designing a good and future-proof check point interface for applications can be challenging. Enforcing its use in the complex application landscape of a corporation can take years.
- *Configuration complexity.* In addition to the implementations of concrete CHECK POINT (287)s and its user applications, the specific configuration also

needs to be considered when assessing security. If CHAIN OF RESPONSIBILITY is applied, such as with Linux PAM, understanding the implications of such chaining of concrete CHECK POINT (287) implementations is no longer trivial.

See Also

CHECK POINT (287) uses STRATEGY [GoF95] for gaining flexibility in application security.

CHECK POINT (287) implementations can employ CHAIN OF RESPONSIBILITY [GoF95] to delegate decisions among several concrete CHECK POINT (287) implementations. PAM allows chaining of its modules in this way based on its configuration.

SINGLE ACCESS POINT (279) is used to ensure that CHECK POINT (287) gets initialized correctly and that none of the security checks are skipped.

CHECK POINT (287) usually configures a SECURITY SESSION (297) and stores the necessary security information in it.

ROLE-BASED ACCESS CONTROL (249) is often used to implement CHECK POINT (287)'s security checks. CHECK POINT (287) sets or evaluates a user's roles stored in its SECURITY SESSION (297).

For development purposes, or in domains in which security is not a requirement, NULL OBJECT can be used for implementing a concrete CHECK POINT (287) that permits everything.