Security Patterns Repository Version 1.0

Darrell M. Kienzle, Ph.D.

Matthew C. Elder, Ph.D.

David Tyree

James Edwards-Hewitt

Executive Summary

A *security pattern* is a well-understood solution to a recurring information security problem. They are patterns in the sense originally defined by Christopher Alexander (the basis for much of the later work in design patterns and pattern languages of programs), applied to the domain of information security. A security pattern encapsulates security expertise in the form of worked solutions to these recurring problems, presenting issues and trade-offs in the usage of the pattern. This document presents version 1.0 of our Security Patterns Repository.

The Security Patterns Repository Version 1.0 consists of 26 patterns and 3 mini-patterns. (A mini-pattern is a shorter, less formal discussion of security expertise in terms of just a problem and its solution.) To define the scope of the problems our patterns address, we focused on the domain of web application security. The patterns are divided between structural patterns and procedural patterns. Structural patterns are patterns that can be implemented in an application; they encompass design patterns (such as those presented by the Gang of Four), but can also apply at the architectural or implementation levels. Procedural patterns are patterns that can be used to improve the process for development of security-critical software; they often impact the organization or management of a development project. Following the presentation of security patterns in this document, we include a comprehensive bibliography collecting references from all the patterns with other relevant web application security and patterns material.

To supplement this patterns repository document, we have developed a Web application that is a functional repository for these Security Patterns. Our repository application enables viewing of patterns, submitting of feedback on the patterns, and editing of patterns for authorized users. We will include relevant example code within the repository document when this application is finalized.

The Security Patterns repository is available at **http://www.securitypatterns.com**. Feedback will be accepted on that Web site and updates will be posted there.

A. Security Patterns

There is a huge disconnect between security professionals and systems developers. Security professionals are primarily concerned with the security of a system, while developers are primarily concerned with building a system that works. While security is one of the non-functional goals with which developers must be concerned, it is but one of many. And while security professionals complain that developers don't take security seriously, developers are just as frustrated that security professionals don't understand that security is not their only concern.

Security patterns are proposed as a means of bridging this gap. Security patterns are intended to capture security expertise in the form of worked solutions to recurring problems. Security patterns are intended to be used and understood by developers who are not security professionals. While the emphasis is on security, these patterns capture the strengths and weaknesses of different approaches in order to allow developers to make informed trade-off decisions between security and other goals.

Above all, security patterns are meant to be constructive. Far too much of the available security expertise is presented in the form of laundry lists of what not to do. The poor developer who attempts to understand security is likely to be overwhelmed by these lists. Security patterns instead try to provide constructive assistance in the form of worked solutions and the guidance to apply them properly.

A.1. What Is a Security Pattern?

A security pattern is a well-understood solution to a recurring information security problem. They are patterns in the sense originally defined by Christopher Alexander, applied to the domain of information security. While some of these patterns will take the form of design patterns, *not all security patterns are design patterns*.

Because of the popularity of design patterns in the software engineering community, the natural inclination is to assume that anything going by the name "security patterns" should be described using a UML diagram and include sample source code. While it is true that many interesting security patterns can be presented this way, there are many other important patterns that do not fit within these constraints.

We make no attempt to categorize different classes of pattern formally. We have observed a few broad types, but we don't feel it important to rigidly enforce any formal identification. For informational purposes, we identify two broad categories:

- *Structural patterns*. These are patterns that can be implemented in the final product. They encompass design patterns, such as those used by the Gang of Four. They often include diagrams of structure and descriptions of interaction.

- *Procedural patterns*. These are patterns that can be used to improve the process for

development of security-critical software. They often impact the organization or management of a development project.

Our patterns adhere to the following security patterns template:

- Name (including aliases—"a.k.a"—parenthetically)
- Abstract
- Problem
- Solution
- Issues
- Trade-Offs
- Related Patterns
- References

There are numerous existing templates for design patterns, security patterns, and other patterns efforts. We have examined previous patterns templates and settled on the above structure specific to our security patterns.

Version 1.0 of the Security Patterns Repository, presented in this document, consists of 26 patterns and 3 mini-patterns. A mini-pattern is a shorter, less formal discussion of security expertise in terms of just a problem and its solution. There are 13 structural patterns and 3 structural mini-patterns. There are 13 procedural patterns.

It is important to note that there are a number of different efforts bearing the name "security patterns". Please see the section on *Related Work* in our Template and Tutorial document for a discussion of other approaches to (and other definitions of) security patterns.

A.2. Structural Patterns

The table below outlines the structural security patterns and mini-patterns presented in Section B.

Pattern Name	Abstract
Account Lockout	Passwords are the only approach to remote user authentication that has gained widespread user acceptance. However, password-guessing attacks have proven to be very successful at discovering poorly chosen, weak passwords. Worse, the Web environment lends itself to high-speed, anonymous guessing attacks. Account lockout protects customer accounts from automated password-

	guessing attacks, by implementing a limit on incorrect password attempts before further attempts are disallowed.
Authenticated Session	An authenticated session allows a Web user to access multiple access-restricted pages on a Web site without having to re-authenticate on every page request. Most Web application development environments provide basic session mechanisms. This pattern incorporates user authentication into the basic session model.
Client Data Storage	It is often desirable or even necessary for a Web application to rely on data stored on the client, using mechanisms such as cookies, hidden fields, or URL parameters. In all cases, the client cannot be trusted not to tamper with this data. The *Client Data Storage* pattern uses encryption to allow sensitive or otherwise security-critical data to be securely stored on the client.
Client Input Filters	Client input filters protect the application from data tampering performed on untrusted clients. Developers tend to assume that the components executing on the client system will behave as they were originally programmed. This pattern protects against subverted clients that might cause the application to behave in an unexpected and insecure fashion.
Directed Session	The *Directed Session* pattern ensures that users will not be able to skip around within a series of Web pages. The system will not expose multiple URLs but instead will maintain the current page on the server. By guaranteeing the order in which pages are visited, the developer can have confidence that users will not undermine or circumvent security checkpoints.
Hidden Implementation	The *Hidden Implementation* pattern limits an attacker's ability to discern the internal workings of an application—information that might later be used to compromise the application. It does not replace other defenses, but it supplements them by making an attacker's job more difficult.
Encrypted Storage	The *Encrypted Storage* pattern provides a second line of defense against the theft of data on system servers. Although server data is typically protected by a firewall and other server defenses, there are numerous publicized examples of hackers stealing databases containing sensitive user information. The *Encrypted Storage* pattern ensures that even if it is stolen, the most sensitive data will remain safe from prying eyes.
Minefield	The *Minefield* pattern will trick, detect, and block attackers during a break-in attempt. Attackers often know more than the

	developers about the security aspects of standard components. This pattern aggressively introduces variations that will counter this advantage and aid in detection of an attacker.
Network Address Blacklist	A network address blacklist is used to keep track of network addresses (IP addresses) that are the sources of hacking attempts and other mischief. Any requests originating from an address on the blacklist are simply ignored. Ideally, breaking attempts should be investigated and prosecuted, but there are simply too many such events to address them all. The *Network Address Blacklist* pattern represents a pragmatic alternative.
Partitioned Application	The *Partitioned Application* pattern splits a large, complex application into two or more simpler components. Any dangerous privilege is restricted to a single, small component. Each component has tractable security concerns that are more easily verified than in a monolithic application.
Password Authentication	Passwords are the only approach to remote user authentication that has gained widespread user acceptance. Any site that needs to reliably identify its users will almost certainly use passwords. The *Password Authentication* pattern protects against weak passwords, automated password-guessing attacks, and mishandling of passwords.
Password Propagation	Many Web applications rely on a single database account to store and manage all user data. If such an application is compromised, the attacker might have complete access to every user's data. The *Password Propagation* pattern provides an alternative by requiring that an individual user's authentication credentials be verified by the database before access is provided to that user's data.
Secure Assertion	The *Secure Assertion* pattern sprinkles application-specific sanity checks throughout the system. These take the form of *assertions* – a popular technique for checking programmer assumptions about the environment and proper program behavior. A secure assert maps conventional assertions to a system-wide intrusion detection system (IDS). This allows the IDS to detect and correlate application-level problems that often reveal attempts to misuse the system.
Server Sandbox	Many site defacements and major security breaches occur when a new vulnerability is discovered in the Web server software. Yet most Web servers run with far greater privileges than are necessary. The *Server Sandbox* pattern builds a wall around the Web server in order to contain the damage that could result from

	an undiscovered bug in the server software.
Trusted Proxy	A trusted proxy acts on behalf of the user to perform specific actions requiring more privileges than the user possesses. It provides a safe interface by constraining access to the protected resources, limiting the operations that can be performed, or limiting the user's view to a subset of the data.
Validated Transaction	The *Validated Transaction* pattern puts all of the security-relevant validation for a specific transaction into one page request. A developer can create any number of supporting pages without having to worry about attackers using them to circumvent security. And users can navigate freely among the pages, filling in different sections in whatever order they choose. The transaction itself will ensure the integrity of all information submitted.

A.3. Procedural Patterns

The table below outlines the procedural security patterns presented in Section C.

Pattern Name	Abstract
Build the Server from the Ground Up	Many Web compromises and defacements occur because of unnecessary and potentially vulnerable services present on the Web server. Default installations of many operating systems and applications are the source of many of these services. This pattern advocates building the server from the ground up: understanding the default installation of the operating system and applications, simplifying the configuration as much as possible, removing any unnecessary services, and investigating the vulnerable services that are a part of the Web server configuration.
Choose the Right Stuff	Many security problems can be avoided during system design if components, languages, and tools are selected with security in mind. This is not to say that security is the only criterion of concern – merely that it should not be ignored while making these decisions. This pattern provides guidance in selecting appropriate Commercial-Off-the-Shelf components and in deciding whether to use build custom components.
Document the Security	In order for developers to make consistent, intelligent development choices regarding security, they have to understand

Goals	the overall system goals and the business case behind them. If the security goals are not documented and disseminated, individual interpretation could lead to inconsistent policies and inappropriate mechanisms.
Document the Server Configuration	Web servers and application servers are extremely complex, and complexity is a major impediment to security. In order to help manage the complexity of Web server and application configurations, developers and administrators must document the initial configuration and all modifications to Web servers and applications.
Enroll by Validating Out of Band	When enrolling users for a Web site or service, sometimes it is necessary to validate identity using an out-of-band channel, such as postal mail, telephone, or even face-to-face authentication. The out-of-band channel can be used to establish a shared secret, which can then be used to establish identity during enrollment.
Enroll using Third-Party Validation	When enrolling users for a Web site or service, it is always easier to allow some other party to take on the difficult task of authenticating user identity. When a third-party service is available and sufficiently reliable, the Web application can offload this task on the third party. This approach is becoming more common as third-party services become available. The most common form of transaction authentication—credit card authentication—is a form of third-party validation.
Enroll with a Pre-Existing Shared Secret	When enrolling users for a Web site or service, sometimes it is sufficient to validate identity using a pre-existing shared secret, such as a social security number or birthday. The use of a pre-existing shared secret enables enrollment without prior communication specific to setting up an account.
Enroll without Validating	When enrolling users for a Web site or service, sometimes it is not necessary to validate the identity of the enrolling user. When there is no initial value involved in the Web site or service for which enrollment is occurring, validation is an unnecessary procedure and can be eliminated.
Log for Audit	Applications and components offer a variety of capabilities to log events that are of interest to administrators and other users. If used properly, these logs can help ensure user accountability and provide warning of possible security violations. The *Log for Audit* pattern ties logging to auditing, to ensure that logging is configured with audit in mind and that auditing is understood to be integral to effective logging.

Patch Proactively	During the lifetime of a software system, bugs and vulnerabilities are discovered in third-party software, and patches are provided to address those issues. Rather than waiting for the system to be compromised before applying patches ("patching *reactively*"), administrators of software systems should monitor for patches often and apply them *proactively*.
Red Team the Design	Red teams, which examine a system from the perspective of an attacker, are commonly used to assess the security of a finished system. However, the earlier in development that a problem is found, the easier it is to fix. The *Red Team the Design* pattern effects a security evaluation of the application at the stage when it is most possible to fix any problems identified.
Share Responsibility for Security	The *Share Responsibility for Security* pattern makes all developers building an application responsible for the security of the system. Security consists of more than just encryption, anti-virus software, and firewalls. Any element of a system can have security concerns, and system developers have to understand and address those concerns. Use of this pattern avoids the common problem of "the security guy" or security team being pitted against the rest of the development team.
Test on a Staging Server	Web site development requires extensive testing to enable availability, protect confidentiality, and ensure integrity. While unit testing can be done on development machines, system and integration testing should take place on machines as similar to the production servers as possible. The use of a staging server enables necessary testing while preventing the outages that often occur when developers and administrators experiment with the live production system on the fly.

B. Structural Patterns

The following structural patterns are presented in this section:

- Account Lockout
- Authenticated Session
- Client Data Storage
- Client Input Filters
- Directed Session (mini-pattern)
- Hidden Implementation (mini-pattern)
- Encrypted Storage
- Minefield
- Network Address Blacklist
- Partitioned Application
- Password Authentication
- Password Propagation
- Secure Assertion
- Server Sandbox
- Trusted Proxy
- Validated Transaction (mini-pattern)

Account Lockout

(a.k.a. Disabled Password)

Abstract

Passwords are the only approach to remote user authentication that has gained widespread user acceptance. However, password-guessing attacks have proven to be very successful at discovering poorly chosen, weak passwords. Worse, the Web environment lends itself to high-speed, anonymous guessing attacks. Account lockout protects customer accounts from automated password-guessing attacks, by implementing a limit on incorrect password attempts before further attempts are disallowed.

Problem

Many servers require that users be identified before using the system, either to ensure accountability or to protect user data between sessions of usage. At present, passwords are the only approach to user authentication that has gained widespread user acceptance. While passwords are extremely common, users tend to pick passwords that are easily guessed. It is easy to build (and trivial to download) password-guessing tools that are very effective at guessing poorly chosen passwords.

In the traditional, predominantly standalone computing environment, passwords might have been strong enough to protect an important system. Users who sit at a keyboard and guess passwords are exposed to detection and can only type so fast. Furthermore, many operating systems implement login delays that get successively longer with each incorrect password. Using these techniques, password-guessing attacks can be slowed to a crawl.

In the Web environment, however, an attacker can remain completely anonymous while guessing tens of thousands of passwords per hour. Even if the system implements delays, they can be circumvented by using numerous concurrent connections to make many guesses in parallel.

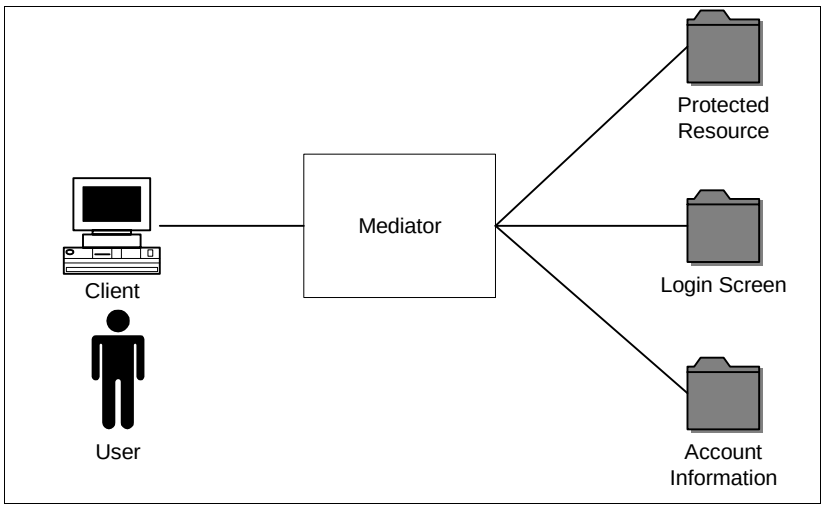
If a user account is compromised, the user will blame the Web site developer. Even if the compromise is due to a poor password choice by the user, the site operators and developers will lose customer goodwill and may even be considered negligent for not implementing best practice solutions.

Solution

The *Account Lockout* pattern protects accounts from password-guessing attacks. For every user account, the server maintains a count of incorrect password attempts. When a user successfully logs in, the count is cleared. When the user provides an incorrect password, the count is incremented. Once some predefined threshold of failed login attempts is reached, the account is locked.

When a user attempts to authenticate against a locked account, the system logs that event but never processes the request. It simply responds as if an incorrect password or username had been provided. The user is never aware of the lockout mechanism, and an automated guessing attack would progress, blithely unaware that all login attempts were actually being ignored.

The system also keeps track of the most recent login attempt. After some defined period of inactivity, the account is automatically unlocked. As an option, the administrator can also manually reset an account, should a customer request help.



Authentication Attempt

The following interactions occur on an authentication attempt:

- The client is provided with a transaction form or login screen requiring both a username and password
- The user provides the username and password, and submits the request for a protected resource
- The mediator checks the username, and if valid retrieves the account information. If the username is invalid, return a generic failed login message.
- The mediator checks if this user's account was locked out (number of successive failed logins exceeds the threshold) and not yet cleared (last failed login time + reset duration > current time). If locked out, skip to the increment step.
- The mediator checks the validity of the password.
- If the password is valid, reset the number of failed logins to 0, and execute the request against the protected resource. (End of successful interaction.)
- If the password was not valid, increment the number of failed login attempts against the account, and set the last failed login time to the current time.

- Return a generic failed login message.

Account Reset

The following interactions occur during an account reset:

- User contacts customer service and explains difficulty using the system
- Customer service representative optionally resets the password (see the *Password Authentication* pattern).
- Customer service representative resets the number of failed logins to 0.

Issues

Note that account lockout can only protect an individual account. If an attacker chooses a weak password (such as "password") and then randomly guesses account names, no single account will observe more than one incorrect attempt. And the attacker will eventually find an account (probably many) with that password. For this reason, the *Network Address Blacklist* pattern should also be implemented to protect valuable information.

The threshold for lockout cannot be too low. Many conventional systems (such as automated tellers) only allow three unsuccessful login attempts. This number is too low for all but the most critical systems. Web users typically have a number of different passwords and often will try several different candidates before getting the correct password. A limit of three incorrect attempts will inconvenience (even anger) legitimate users and will greatly increase the burden on customer service. A limit of five to ten attempts is far more reasonable.

Communication with the User

When a user authenticates successfully, it is a good idea to inform him/her of the number of failed login attempts since the last successful login. A user who mistyped his/her password will recognize that the invalid attempts were legitimate. But the user whose account is under attack will be alerted to the fact and may make the system administrators aware of the problem. For similar reasons, it is a good idea to inform the user of the last successful login – if the account has been compromised, the legitimate user may be made aware of that fact.

When a user fails to login correctly, a generic message should be provided. It should not provide any indication of whether the account name was invalid, the password was incorrect, or the account was locked out. A standard message could be the following: "The information entered was incorrect. Please try again. Note that passwords are case sensitive and the Caps Lock key should be turned off. If problems persist, please call customer service." Alternately, instead of directing the user to customer service, a message could say: "If problems persist, please wait and try back in an hour."

Note the importance of not revealing whether the account has been locked out. This ensures that an attacker cannot know whether a password-guessing attack is actually covering the entire

password space or whether successive guesses are simply being dropped on the floor. It also prevents the attacker from learning how many attempts are required in order to lock out an arbitrary account. Finally, if counts of login attempts are only maintained for valid accounts, informing the user that an account was locked out would reveal to an attacker that the account ID was a valid account.

Many sites opt to inform the user of the lockout condition. There are valid reasons for doing this. Most importantly, it avoids user frustration at being unable to login using the correct password. It also has the effect of reassuring the user that the site takes security seriously. However, an attacker will be able to learn how the account lockout mechanism works, and could misuse this information to effect a large-scale denial of service attack. Even more dangerous, the site would be forced to either (a) maintain a count of invalid logins for all account names, not just actual account names, or (b) risk revealing which accounts are valid, and which are not—since the system would only return a lockout condition for a valid account name. If you opt to inform the user of lockout conditions, it is imperative that an automatic reset mechanism not be implemented. An attacker would quickly determine the length of the delay and develop a patient, automated password guesser.

Account Reset Considerations

When a user account is locked out, it can be unlocked in one of two ways. A customer service representative can clear the lockout manually, or the lockout can clear automatically after a certain amount of time has elapsed. Manual unlocking impacts both management cost and usability. Automatic unlocking may allow a patient attacker to launch a slow password-guessing attack. However, if the attacker cannot distinguish between a lockout response and an incorrect guess, the automated guessing attack can be rendered ineffective.

When constructing an automatic reset mechanism, consider the rate at which guesses could be made. If he system allows 10 invalid attempts before locking the user out and then resets after 15 minutes, an attacker who is knowledgeable of the workings of the lockout mechanism could guess 960 passwords per day. That number is too large. Five guesses and an hour lockout drops that number to a more reasonable 120 attempts per day. In either case, the system should track lockout information and inform the system administrator of suspicious behavior. And if at all possible, the details of the account lockout mechanism should not be apparent to the end user.

Examples

Account lockout is common in conventional password and PIN systems. PINs are particularly susceptible to guessing attacks and are therefore usually protected with low limits on incorrect guesses. Passwords would seem to be harder to guess based on the increased number of possible passwords—for example, 8 characters of 96 printable ASCII codes gives approximately 7.2 x 10^{15} different possible passwords. However, studies have repeatedly shown that most users choose passwords from a tiny subset of actual words and slight variations. Therefore, account lockout is critical to protect against random guessing. FIPS 112 provides detailed instructions on using passwords in traditional systems [3].

All of the on-line banking systems with which we are familiar provide an account lockout mechanism. These generally have a low threshold of three incorrect attempts before the account is locked out and requires manual intervention by a customer service representative. One of the authors has the habit of forgetting which password was used for which account and has personally observed the lockout mechanisms on a number of such sites.

We have implemented the *Account Lockout* pattern in our Web repository application. The code will be made available on-line.

Trade-Offs

Accountability	Account lockout increases accountability by helping ensure that user accounts will not be compromised using a password-guessing attack. Furthermore, attempts by users to repudiate the transactions they initiate will be made more difficult by inclusion of best practice defenses such as account lockout.
Availability	Availability of individual accounts can be adversely affected by a low account lockout limit. An attacker could also misuse the account lockout mechanism to effect a large-scale denial-of-service attack on the site.
Confidentiality	Confidentiality of user data will be increased by any additional protection of the account.
Integrity	Integrity of individual accounts will be enhanced and this could indirectly improve the integrity of the site.
Manageability	Manageability will be somewhat adversely affected by the inclusion of an account lockout mechanism, as customer service calls will increase. A limit that is too restrictive will increase this burden significantly.
Usability	Usability will be somewhat adversely affected if the limit on attempts is too low, and more so if the system does not inform users of the lockout condition.
Performance	Performance is affected slightly by the need to track failed login attempts.
Cost	Development and quality assurance costs are increased by the need to construct the lockout mechanism. Management costs can be increased significantly.

Related Patterns

- *Authenticated Session* – a related pattern that can utilize an account lockout mechanism to protect user sessions.

- *Encrypted Storage* – a related pattern that can protect against compromise of the password store.

- *Enroll without Validating* – a related pattern that presents a procedure and circumstances for when initial authentication credentials are not required.

- *Enroll with a Pre-Existing Shared Secret* – a related pattern that presents one procedure for communicating the initial authentication credentials to a user.

- *Enroll by Validating Out of Band* – a related pattern that presents one procedure for communicating the initial authentication credentials to a user.

- *Enroll using Third-Party Validation* – a related pattern that presents one procedure for communicating the initial authentication credentials to a user.

- *Hidden Implementation* – a related pattern advocating that the user is not provided with information about the lockout scheme, or even that a lockout exists, in order to prevent it being misused against the system.

- *Network Address Blacklist* – a related pattern that provides a complementary protection mechanism to account lockout by locking out misbehaving client network addresses.

References

[1] Chun, M. "Authentication Mechanisms - Which is Best?" http://rr.sans.org/authentic/mechanisms.php, April 2001.

[2] Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology Security Evaluation Version 2.1*. http://www.commoncriteria.org/cc/cc.html, August 1999.

[3] National Computer Security Center. *DoD 5200.28-STD, Trusted Computer System Evaluation Criteria*. December 1985.

[4] National Institute of Standards and Technology (NIST) Information Technology Library (ITL). "Federal Information Processing Standards Publication 112: Password Usage". http://www.itl.nist.gov/fipspubs/fip112.htm, May 1985.

[5] Wheeler, D. "Secure Programming for Linux and Unix HOWTO – v2.965". http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html, May 2002.

Authenticated Session

(a.k.a. Server-Side Cookies, Single Sign-On)

Abstract

An authenticated session allows a Web user to access multiple access-restricted pages on a Web site without having to re-authenticate on every page request. Most Web application development environments provide basic session mechanisms. This pattern incorporates user authentication into the basic session model.

Problem

HTTP is a stateless, transaction-oriented protocol. Every page request is a separate atomic transaction with the Web server. But most interesting Web applications require some sort of session model, in which multiple user page requests are combined into an interactive experience. As a result, most Web application environments offer basic session semantics built atop the HTTP protocol. And the protocol itself has evolved to provide mechanisms—such as basic authentication and cookies—that allow session models to operate correctly across different Web browsers.

An obvious use for session semantics is to allow users to authenticate themselves once instead of every time they access a restricted page. However, great care must be taken when using session semantics in a trusted fashion. Most session mechanisms are perfectly adequate for tracking non-critical data and implementing innocuous transactions. In such cases, if an end user circumvents the session mechanism, no harm is caused. But it is easy to make mistakes when applying session mechanisms to situations where accountability, integrity, and privacy are critical.

Solution

An authenticated session keeps track of a user's authenticated identity through the duration of a Web session. It allows a Web user to access multiple protected pages on the Web site without having to re-authenticate him-/herself on every page request. It keeps track of the last page access time and causes the session to expire after a predetermined period of inactivity.

The server maintains the authenticated user identity and the time of the last requests as part of the client session data. Every protected page contains a standard header (executed on the server) that checks the authentication information associated with the session.

The first time the user requests a protected page, the server executes the authentication check and notes that no authenticated identity is present in the session information. At that point, the server records the original request and redirects the user to a login page.

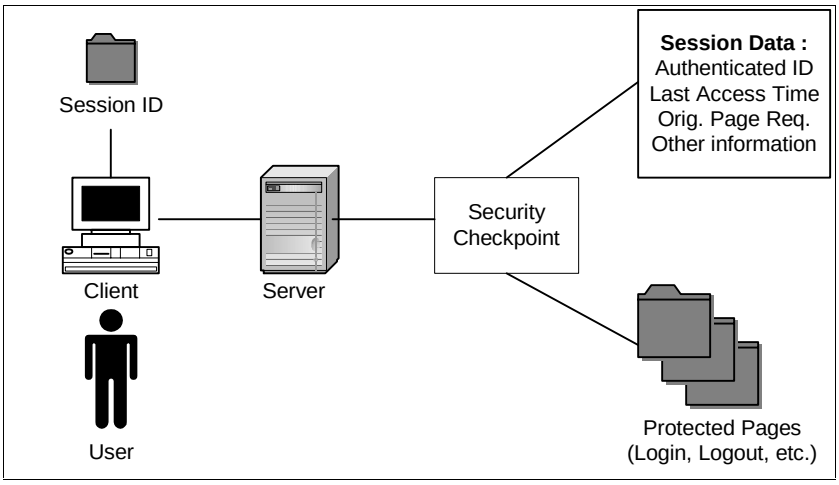
When the server receives and verifies a user's login credentials (typically a username and password), it redirects the user back to the originally requested page. On all subsequent page

requests, the server checks the authenticated identity without requiring that the user re-authenticate him or herself. If the authenticated user is allowed access to the page, the page is served. Dynamically created pages can use the authenticated identity in order to incorporate data that is sensitive to that user.

There are two ways in which an authenticated session can end. The user can explicitly invoke a logout page, which causes the cached credentials to be flushed. Alternately, if the user session remains inactive for some predetermined period of time, the session will time out and the first subsequent request will again require authentication. In order to effect this, each page request should check the current time against the recorded time of last page request, and update the time of last page request appropriately.

The *Authenticated Session* pattern caches the user's authenticated identity on the server. Because it is stored on the server, the application can be more confident that the user has not tampered with it. The only way that the authenticated identity session attribute can be set is if the user successfully authenticated to the authentication module. The *Authenticated Session* pattern relies on existing session mechanisms to associate the client with a particular session.

This pattern compartmentalizes the session security policy within a single component, so that changes to the policy (affecting usability, accountability, and performance) do not impact the client or other parts of the application.



Page request without valid authentication

The following interactions occur on a page request without valid authentication:

- The client requests a protected page from the server, passing the session identifier.
- The underlying session mechanism retrieves the session data and invokes the protected page object.
- The protected page invokes the authentication checkpoint.
- The authentication checkpoint determines that the authenticated identity field is empty or that

the last access time exceeds the session timeout window.

- The requested page URL and submitted data is stored in the session object as the *page originally requested*.

- The authentication checkpoint redirects the client to the login screen.

Login

The following interactions occur in the login procedure:

- The client requests the login page, submitting a username and password. The session identifier is passed as part of the request.

- The underlying session mechanism retrieves the session data and invokes the login page object.

- The login object validates the username and password.

- If unsuccessful, the unsuccessful login page ("try again") is returned to the user.

- If successful, the identity provided is stored in the authenticated identity field, and the current time in the last access time.

- The user is redirected to the page originally requested (stored in the session data).

Page request with valid authentication

The following interactions occur on a page request with valid authentication:

- The client requests a protected page from the server, passing the session identifier.

- The underlying session mechanism retrieves the session data and invokes the protected page object.

- The protected page invokes the authentication checkpoint.

- The authentication checkpoint validates the authenticated identity field in the session data and ensures that the last access time does not exceed the timeout period.

- The authentication checkpoint stores the current time as the last access time.

- The authentication checkpoint returns to the protected page object, which composes and delivers the requested page to the client.

Logout

The following interactions occur in the logout procedure:

- The client requests the logout page from the server, passing the session identifier.
- The underlying session mechanism retrieves the session data and invokes the logout object.
- The logout object clears the authenticated user id field in the session.
- The logout object returns a "logged out" notification page or redirects the browser to the home page.

Issues

The *Authenticated Session* pattern cannot be implemented using HTTP Basic Authentication, because Basic Authentication caches the user's password at the client and delivers it over the network on each page request. Basic Authentication also lacks the flexibility needed to implement session timeout and the *Account Lockout* pattern.

Some Web application environments permit session data to be stored on the client (as opposed to storing only a session identifier on the client). For example, iPlanet Enterprise Server's Server-Side JavaScript offers this feature. In general, client-side storage should not be used with this pattern. If the two absolutely must be combined, the *Client Data Storage* pattern must be used. One simply cannot trust the client to accurately report the authenticated user identity.

Policy Considerations

Sessions and authentication need not always be combined. For example, many e-commerce sites use sessions to track browsing habits and storing of items in a shopping cart. It is only the checkout and purchasing procedures that need to be authenticated.

It might be possible to make sessions optional. For example, e-Bay appears to store all state in the current URL and retain no session data on the server. When the user places a bid he/she is given the option of creating a session in order to avoid having to re-enter his/her password with each successive bid.

More dangerous transactions should not depend on session authentication and should require that the current password be explicitly provided as part of the transaction. (See the *Password Authentication* pattern.) If a session is somehow hijacked or a user walks away from the computer, this ensures that critical transactions cannot be triggered by somebody else. For example, authorization of an electronic funds transfer should always require authentication of the transaction itself. Password change requests also should always require that the old password be provided as part of the change request transaction.

Any page that contains sensitive data should include a header that instructs the browser not to cache the page. Depending on the level of sensitivity and perceived risk, at the end of a session involving sensitive data, the "logged out" page should either directly request the user's permission to exit the browser or warn the user that they must exit the browser soon to ensure that no one can see their sensitive data.

Session Protection

This pattern maintains session data (including the authenticated user identity) on the application server as part of a session object. In order to associate each client with a session object, the application server assigns each session object a unique, random identifier. This session identifier is then given to the client, and the client presents it on each subsequent page request.

The session identifier must be suitably random and hard to guess. Session identifiers are generally provided by the application server and cannot be modified by custom applications. Nevertheless, if the session identifier mechanism contains vulnerabilities, it could result in attackers being able to predict or guess the session identifiers assigned to active users. This would allow an attacker to hijack another user's session and gain access to sensitive data.

Developers and administrators should research the method used to generate session identifiers and check any known product vulnerabilities (at a site such as securityfocus.com). Examining the contents of cookies (or URLs) will give a basic indication of how long each session identifier is. Anything shorter than 20 hexadecimal characters ("hexits") is probably too short.

Administrators must also be aware of any session-guessing attacks underway. When a user submits an invalid session identifier, the application server should notify the application or the system administrator of that event. Large numbers of invalid session identifiers are a clear indicator that something is amiss (either the site is under attack or the server has just restarted). If a single network address is the source of many invalid session identifiers, it should be added to the network address blacklist (see the *Network Address Blacklist* pattern).

Session Encryption

If a site encrypts passwords, it should also encrypt any page that delivers an authenticated session identifier to the server. If the data on the site or the services offered on the site are sufficiently important, it should use encryption to protect passwords in transit (see the *Password Authentication* pattern for more details). If the session identifier can be used to gain access to password-protected pages, those pages should also be encrypted in order to protect the session identifier in transit.

Once the user has been authenticated, the session identifier will be communicated along with every page request. If those pages are not encrypted, the session identifier will be delivered in the clear. An attacker who can monitor network communications could use the session identifier to hijack the session. The attacker would not be able to see the user's sensitive data (such as a password or credit card number), but they would be able to use the session until the user logs out (longer if the user walks away from the keyboard). *Note that this is one reason why a Web site should never echo the most sensitive information back to the user*.

When using encrypted pages to protect session identifiers, it is important to understand how to prevent session identifiers from being inadvertently delivered to an unprotected page:

- When using cookies to store session identifiers, the cookie option that requires the cookie to only be delivered over an SSL-protected connection should be enabled. If the application server does not allow this option to be configured, the application should be divided into two

different servers: one for encrypted pages, one for other pages. The cookie identifier should explicitly name only the encrypted server.

- When using URL's to store session identifiers, the URL rewriting function that adds the session identifier to any links in the page should only be invoked when those links target other encrypted pages.

If all sessions on a server are authenticated, it might be possible to depend on the session mechanism to provide the timeout. Applications constructed using servlets and Java Server Pages even have access to per-application session timeout parameters. However, many session mechanisms are geared at general site usage and have fixed timeouts that are often too long for authenticated sessions.

Usability

The system should be tested with two sessions using the same user identity. Testing the system with two open windows in the same instance of the Web browser should also occur to ensure an appropriate user experience.

When an authenticated session times out, it should be possible to restore state to the page that was last accessed. The login page should not blindly overwrite all session data.

The length of the authenticated session timeout is a trade-off between usability and security. It should not be too short, because users often run multiple programs and work slowly. However, to protect users from unauthorized access if they walk away from their keyboard without logging out, the session timeout should not be more than 10-15 minutes.

The system must provide an explicit logout mechanism that clears the authenticated user identifier from the session. The logout button should be available from every page when the user's session contains an authenticated identity.

Possible Attacks

There are a number of possible attacks that can be perpetrated on authenticated sessions:

- Session continuation – if session data is not properly cleared from the client, it is possible for an attacker to revisit pages cached by the browser or even continue the authenticated session.

- Session hijacking – if the session identifier is observed traveling over the network, it is possible to jump into the session by using the identifier as a part of requests from another browser.

- Direct page access – if every page does not explicitly check authentication, it is possible to guess URLs (or use URLs stored in another user's browser) and gain direct access to the page.

- Manipulation of client data – if a session depends on data provided by the client (either hidden fields, encoded URLs, cookies, or referring pages), it is possible for an attacker to

manipulate that data to circumvent the server authentication model.

It is safe to use the referring page to ensure correct sequence within the application is observed, but one should not depend on the referring page to indicate whether a request has been authenticated.

Examples

Many significant Web banking and e-commerce applications rely on this pattern. Any site that enforces user authentication and does not store that information on the client uses something similar.

Trade-Offs

Accountability	This pattern increases accountability by providing a straightforward, secure approach to repeated authentication.
Availability	No effect.
Confidentiality	See Accountability.
Integrity	See Accountability.
Manageability	No effect.
Usability	The pattern increases usability by not requiring repeated logins. However, if the session timeout is too short or overwrites the session state, it will adversely affect usability.
Performance	There is little performance impact from storing authentication data on the server. However, storing session data on the server can increase server load and make load balancing difficult. If authentication data is stored on the client, significant overhead will be incurred in cryptographically validating the data against tampering.
Cost	Use of a standard authentication model for all protected pages can lower cost by reducing the quality assurance burden.

Related Patterns

- *Network Address Blacklist* – a related pattern that demonstrates a procedure for blocking a network address from further access attempts if a session identifier guessing attack is conducted.

- *Password Authentication* – a related pattern that presents the secure management of passwords, which are almost always used as the authentication mechanism for this pattern.

References

[1] Coggeshall, J. "Session Authentication". http://www.zend.com/zend/spotlight/sessionauth7may.php, May 2001.

[2] Cunningham, C. "Session Management and Authentication with PHPLIB". http://www.phpbuilder.com/columns/chad19990414.php3?page=2, April 1999.

[3] Kärkkäinen, S. "Session Management". *Unix Web Application Architectures*. http://webapparch.sourceforge.net/#23, October 2000.

Client Data Storage

(a.k.a. Cryptographic Storage, Tamperproof Cookie)

Abstract

It is often desirable or even necessary for a Web application to rely on data stored on the client, using mechanisms such as cookies, hidden fields, or URL parameters. In all cases, the client cannot be trusted not to tamper with this data. The *Client Data Storage* pattern uses encryption to allow sensitive or otherwise security-critical data to be securely stored on the client.

Problem

Many Web application designs depend on the ability to store data on the client. From a security perspective, it is almost always preferable to store data on the server. But there are considerations other than security that may require client-side storage to be used:

- Load balancing across multiple application servers can be complicated by the need to share session data across servers. It is much easier to store the session data on the client and have the client provide that data with each request. Making the application servers stateless with respect to any particular client allows large sites, such as e-Bay, to be far more responsive.

- Many e-commerce sites find considerable value in tracking users' browsing habits across visits. But such tracking can represent a significant amount of data that must be stored on the server. Furthermore, the server will generally include large amounts of stale data that may no longer be associated with any browser. It is far more economical to store this data on the client.

- Some designs (including versions of the *Password Propagation* pattern) require that the user's password be stored on the client in order to effect single sign-on. Amazon.com's one-click shopping is an excellent example of this.

- Storing sensitive data, such as passwords and credit card numbers, on the client instead of the server would prevent a compromise of the Web site from compromising the sensitive data associated with every client.

- In order to access data stored on the server, the client needs to be able to identify that data using a session identifier. The session identifier is itself sensitive.

Whatever the reason for storing sensitive data on the client, the problem arises that the client cannot be trusted:

- If attackers manually inspect the contents of cookies and Web pages, they may be able to glean information about the operation of the Web site that could later be used to compromise the site.

- If a security-conscious individual notes that sensitive data is stored in the clear, the results can be a public relations disaster.

- If a user's machine is stolen or otherwise compromised, an attacker would be able to gain access to any sensitive client-resident data.

- Cross-site scripting attacks allow a remote Web site to gain access to cookies created by another site. Any sensitive data in the cookie could be compromised remotely. Similarly, sensitive data in a URL could be extracted from the browser's referring page.

- If an attacker is able to modify authentication data on the client, the attacker could assume the identity of another user and compromise that user's account.

- If an attacker is able to modify sensitive application data, they could manipulate the application into behaving improperly.

Solution

The *Client Data Storage* pattern uses encryption techniques to protect data that this stored on the client. Using encryption ensures that sensitive data will not be inadvertently revealed. Using message authentication codes ensures that the data cannot be tampered with on the client.

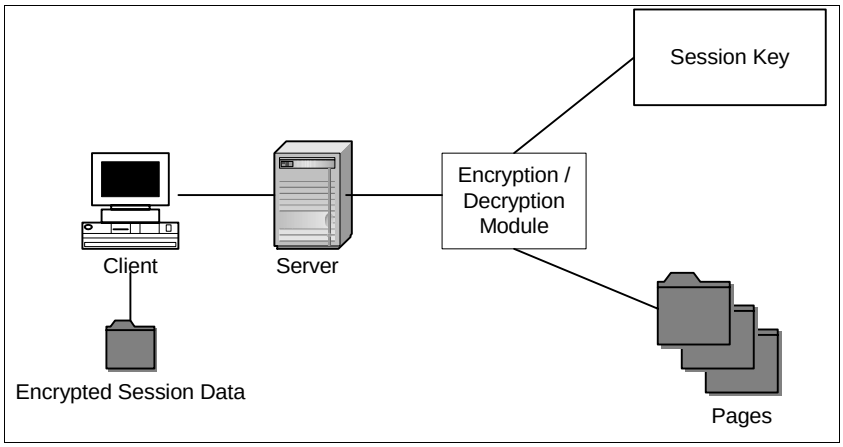
The *Client Data Storage* pattern employs these cryptographic techniques whenever sensitive data is to be delivered to the client. When this data is retrieved from the client it is decrypted and verified by a security checkpoint function. Any evidence of tampering is reported as a security-relevant event.

The *Client Data Storage* pattern uses a single symmetric key to encrypt and authenticate all data for all clients. When the technique is used to protect data within a single session, the key can be quite short, but should be rotated frequently. When the technique must protect long-lived client data across sessions, the key must be much stronger, as it will be very difficult to rotate keys without significant impact on either development cost or usability.

The most straightforward approach adds two functions on the application server: one to store sensitive data in the session object, and one to validate that data before it is used. When the user logs in, the authenticated username should be hashed using a session key stored on the server. The hash should be stored in the session data along with the username. When a protected page is requested, the server should recompute the hash of the username stored in the session data, and compare it to the hash stored in the session data. If the two match, the username can be trusted.

In order to protect against guessing attacks, the session key should be changed periodically. In general, changing the key once a day should be sufficient. In order to validate username hashes that span a change of keys, any failed hash comparison should be recomputed using the previous key as well. If the old key successfully validates the hash, the username should be accepted and a new hash created using the current key. In order to protect the key against brute force guessing attacks, repeated incorrect hashes submitted from a single network address should result in that

address being added to the network address blacklist. If the *Network Address Blacklist* pattern is not used, these events should be closely monitored by system administrators.



Issues

Many of the issues are identical to those of the *Encrypted Storage* pattern.

The following general principles should be followed:

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

Protection of the Key

If at all possible, the key should not be stored on the file system. There are COTS devices available that provide the system with a physical encryption card. These devices offer performance benefits and also guarantee that the key cannot be read off the device. The key is manually loaded into the card, the encryption takes place on the card, and the key cannot be extracted from the card. The only downside to this approach is the cost – both the cost of purchasing the hardware and the development and maintenance costs of programming around it.

A cheaper alternative to loading the key is to require that an administrator load the key at system start, either from removable media or using a strong passphrase. This reduces the risk of having the key on-line but does expose the key to a fair number of different people. This approach

might also sacrifice some availability because an operator must manually intervene whenever the system restarts.

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

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

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

Possible Attacks

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

- If the data is not sufficiently protected, an attacker may be able to decrypt the data via a brute force attack.
- An attacker that compromises the server may be able to gain access to the global encryption key.
- Cross-site scripting can be used to force the browser to reveal cookies that are intended for a different site.

Examples

Many application servers use cryptographic techniques to generate suitably random session identifiers that are stored on the client

Many large Web sites store encrypted cookies on the client's browser. They include Amazon.com, Buy.com, and e-Bay. Because they are encrypted, it is impossible to know precisely how they are being used.

Trade-Offs

Accountability	No effect.

Availability	Availability could be adversely affected if encryption keys are lost. Also, if a global key must be reentered by a human operator each time the system restarts, availability will suffer if an operator is not available to restart the server.
Confidentiality	This pattern increases confidentiality by ensuring that the data is secure, even if it has been captured in an encrypted form.
Integrity	This pattern helps to ensure the integrity of client data and processing that depends on that data. Any data tampering on the client can be detected.
Manageability	If system administrators must intervene to provide a global password on system restart, this will require around-the-clock administration.
Usability	No direct effect.
Performance	This pattern will have a negative impact on performance because of the processing burden of encrypting and decrypting every message.
Cost	Costs will be incurred from additional processing required to compensate for performance loss and the cost of adding encryption / decryption logic to the application.

Related Patterns

- *Client Input Filters* – a related pattern that verifies all data coming from the client.
- *Encrypted Storage* – a related pattern utilizing encryption to protect confidential data.

References

[1] Landrum, D. "Web Application and Databases Security". http://rr.sans.org/securitybasics/web_app.php, April 2001.

Client Input Filters

(a.k.a. Untrusted Client, Server-Side Validation, Sanity Checking)

Abstract

Client input filters protect the application from data tampering performed on untrusted clients. Developers tend to assume that the components executing on the client system will behave as they were originally programmed. This pattern protects against subverted clients that might cause the application to behave in an unexpected and insecure fashion.

Problem

Web applications are client-server applications, in which the client executes on untrusted hardware outside of the control of the Web application developer. Developers have a tendency to believe that the application will execute as programmed.

However, it is often trivial for attackers to tamper with Web clients, causing them to behave in an untrustworthy manner. For example, many sites depend on data validation performed by JavaScript functions executed on the client. It is easy to copy the page source, remove those checks, and execute the modified client code. Furthermore, the attacker can read the original code and learn what sanity checks the application applies to data.

Other sites rely on an intricate back-and-forth sequence of form submissions, in which data from earlier pages is stored in hidden fields on later pages. Again, it is trivial for the attacker to alter the contents of those fields before submitting the final page. In one famous case, a number of e-business sites used hidden fields to store catalog prices, so that a total selling price could be quickly computed on the client. Attackers were able to freely substitute prices reflecting extremely deep discounts.

A few sites even use client-side checks to enforce security measures, such as checking passwords, enforcing access control restrictions, or implementing account lockout after several failed password attempts. These measures cannot be trusted to execute properly, as attackers are free to ignore them.

A related problem is that many sites depend on anonymous users to submit data that cannot really be validated. For example, a site may ask users to provide their names and mailing addresses before a file can be downloaded or a physical catalog is mailed out. If a malicious user provides bogus data, the site might attempt to process the data. This can provide an avenue for resource consumption attacks and, in the case of the physical catalog mailing, can be quite expensive. Even if detected, a site flooded with spurious requests may cause valid transactions to be discarded along with the invalid ones.

Solution

All data provided by the client should be treated as suspect and filtered at the server:

- Calculated fields provided by the client should be ignored and recomputed at the server when the data is processed.

- Data validity checks performed on the client should be repeated on the server before the data is processed.

- Sensitive data that must be stored on the client should be kept in an encrypted, tamper-proof form. (See the *Client Data Storage* pattern.)

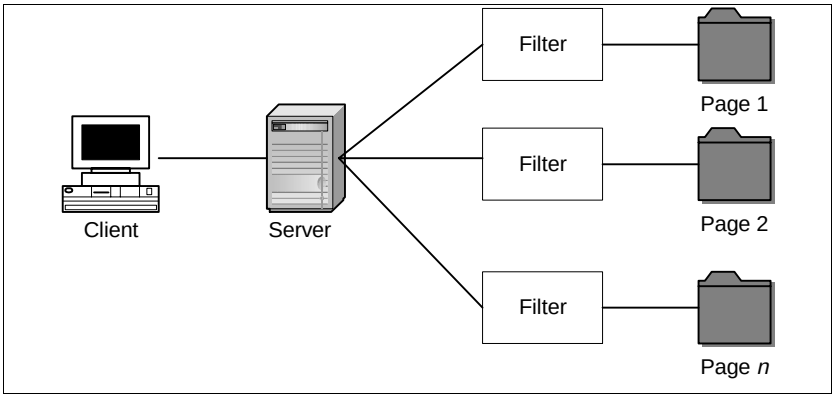
- In addition to field-level validity checks, the server should look for specific signs of bogus data submission and discard requests that are obviously questionable.

- Suspiciously long URLs and header fields should be dropped (and possibly logged).

- Text data that is submitted by the user should be filtered to eliminate scripting tags and other questionable content.

Sanity checks can be integrated into standard page objects, or they can be implemented in separate objects that use chaining (or servlet filtering) to intercept and modify the requests as they are dispatched to the page objects. The latter approach is less efficient, but is easier to get right. In either case, the client filters should always be invoked before any processing of the client-supplied data occurs.

Client filters should be able to modify requests before delivering them to the intended object. If the data cannot easily be fixed, the client filter should reject or simply drop the request. All filtering events should be reported to the central logging mechanism – although many will be benign, they might indicate a pattern of attempted misuse. If a filter detects an obvious attempt to sidestep the security of the system, the request should be blocked and the event reported.



Issues

If an authenticated user account is exhibiting signs of tampering with the client or client-resident data, one possibility is that the account has been compromised. The legitimate user should be contacted out-of-band, or the account should be disabled and the legitimate user will eventually contact customer service.

Many Web sites use hidden fields on the browser, or client-resident cookies, to maintain client state. This can be more efficient than storing that state on the server. However, it is trivial for users to inspect the contents of these fields. In particular, plaintext passwords should never be stored in cookies, URLs, or browser pages.

When testing a Web site, there are a number of techniques that can be used to see all the content that is usually invisible. Use "view source" to examine hidden fields, use "prompt for cookies" to inspect the contents of cookies, and install a packet sniffer to examine the contents of header fields. If using SSL, install a proxy so that the packet sniffer will be able to see decrypted packets between the browser and the proxy. All of these techniques help you understand how easy it is for attackers to look under the surface of an application.

Log all suspicious client behavior, including the originating network address. These logs may be critical for later prosecution if the client turns out to be an attacker. Furthermore, these logs should be integrated into the Intrusion Detection System in order to inform the system administrators that the site may be under attack.

Duplicating Client Side Checks

Web sites often use client-side JavaScript to validate form submissions (either that individual fields comply with formatting restrictions, or that specific fields have not been left empty). It is trivial to bypass these checks. Some user firewalls and browsers even filter out JavaScript, resulting in forms that work but are never validated. For these reasons, it is necessary to re-validate all user-supplied data on the server. The client-side validation is a useful efficiency: it provides greater response times and reduces the burden on the server. But it should never be depended on.

Sensitive data calculated on the browser should be validated on the server. Some e-commerce sites store unit pricing information on catalog Web pages, and then depend on the client browser to calculate total prices. This allows the Web page to be more responsive to user changes then one that must go back to the Web server for these calculations. However, it is critical that the results of that calculation be validated on the server.

When replicating client-side checks on the server, it is crucial that the computations be consistent. Continuing the previous example, if the server recalculates the total price and computes a value that is higher than that calculated by the client, users will be very, very upset. If there is any chance that the calculations could become inconsistent, it is better to take the performance hit and simply calculate everything on the server.

Sanity Checks

For any transaction request that can be initiated by an unauthenticated user, the Web application should invoke a sanity-checking module before processing the request. A sanity-checking module will conduct checks such as: do any of the fields not comply with field-level checks that should have been performed at the client? Is the request from an IP address that has made similar requests recently? Is the request largely similar to another recent request? Do any of the fields contain random garbage (e.g., "asdf" or other obvious sequences), or do multiple fields appear suspect? Do specific field contents raise red flags (e.g., e-mail address of

"president@whitehouse.gov") Using this approach, it should be possible to filter out a large proportion of obviously bogus submissions. Other questionable submissions can be flagged for later observation by a human operator.

Filtering Possible Attacks

There are products that allow data submitted to the Web server to be filtered according to predefined rules. For example, if any legitimate URL is no more than 256 characters, it would be appropriate to truncate any longer request (which may contain a buffer overflow attack). If binary files are never uploaded to the server, non-printable ASCII characters could be stripped from all requests (again, thwarting potential buffer overflows).

Many sites allow users to upload text to be posted on the site (e.g., product reviews, general message forums, auction postings, news stories). Users who know HTML enjoy adding tags to their text in order to add formatting or draw the reader's attention. But any site that echoes user-supplied HTML could be misused to attack other sites. Filter out anything but the most basic HTML tags (font formatting). If links to other sites are provided by users, be careful to filter out any scripting tags or escape sequences. There are libraries available to filter out *cross-site scripting attacks* – use one.

Possible Attacks

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

- Cross-site scripting attacks – this attack occurs when a Web server echoes user-provided JavaScript containing links to a hostile site. For example, many sites include user comments and reviews. If the site does not filter input, a client can be tricked into divulging sensitive cookie data to the cross-linked site.

- Resource consumption attack against the server – if the server is required to keep state for each client, this can lead to depletion of resources on the server if every new client consumes some finite resource on the server.

- Automated submissions – it is straightforward to develop an automated tool for making large numbers of form submissions to a Web site. There are even books explaining how to build "votebots" that can skew on-line polls [6]. The most dangerous of these are password-guessing attacks and enrollment secret-guessing attacks.

Examples

Operating systems do not trust the applications running on them. When applications attempt to invoke operating system services, the operating system checks that the parameters are all valid before executing the command. Operating systems also store critical data within protected memory, providing only opaque handles to the application [5].

As an efficiency, early versions of Windows did not perform parameter validation, and the results were a significant decrease in stability.

At the network level, modern e-commerce sites have generally learned to employ these techniques in order to avoid repeating the earlier mistakes of others. Amazon.com, for example, maintains shopping cart data and all pricing information on the server. The client contains only a session identifier. Session cookies are apparently used to track user browsing habits, but not anything critical to the shopping transaction. Permanent cookies are used to identify the user upon repeat visits, but do not authenticate the user. (One-click shopping, if enabled, is an exception to this.)

This pattern is commonly used in almost all Web applications to verify e-mail addresses; anything entered in an e-mail address field that is not valid will result in a warning.

On at least one Microsoft Web site that attempts to gather customer information, any attempt to enter an e-mail address ending in “microsoft.com”, “msn.com” or “msnbc.com” will result in the submission being rejected. This appears to be an instance of sanity checking.

Trade-Offs

Accountability	No effect.
Availability	If overly sensitive, this pattern can have an adverse effect on availability, preventing legitimate users from using the site.
Confidentiality	No effect.
Integrity	This pattern greatly enhances the integrity of the data processed by a Web site.
Manageability	The management burden could be increased if overly sensitive sanity checks result in a high number of false reports of attacks that must be investigated.
Usability	No effect.
Performance	This pattern will incur a small performance penalty, since it requires some time to perform checks. If data is stored in encrypted form on the client, encrypting and decrypting the data will also exact a performance hit.
Cost	This pattern has fixed implementation costs. However, if overly sensitive it could greatly increase the customer service burden on the site.

Related Patterns

- *Network Address Blacklist* – a related pattern the responds to suspicious client behavior; if

any client-side validation fields have been obviously disabled, this should be input to the network address blacklist.

References

[1] Dominy, R. "Focus on JavaScript: Email Field Validation". http://javascript.about.com/library/scripts/blemailvalidate.htm, 2002.

[2] INT Media Group. "Email Address Validation". http://javascript.internet.com/forms/check-email.html, 2002.

[3] Open Web Application Security Project (OWASP). "Input Filters". http://www.owasp.org/filters/index.shtml, May 2002.

[4] Peterson, S. and D. Fisher. "The next security threat: Web applications". http://zdnet.com.com/2100-11-503341.html?legacy=zdnn, January 2001.

[5] Silberschatz, A., J. Peterson, and P. Galvin. *Operating System Concepts Third Edition.* Addison-Wesley, 1991.

[6] van der Linden, P. *Just Java 2 – Fourth Edition*. Prentice Hall, 1999.

Directed Session

(Mini-Pattern)

Abstract

The *Directed Session* pattern ensures that users will not be able to skip around within a series of Web pages. The system will not expose multiple URLs but instead will maintain the current page on the server. By guaranteeing the order in which pages are visited, the developer can have confidence that users will not undermine or circumvent security checkpoints.

Problem

Web applications often have to collect a great deal of data from a user in order to complete a single transaction. E-commerce purchases, for example, require that a user select items to buy, provide contact information and a shipping address, select shipping options, and provide credit card information. Rather than simply present the user with a huge form with a bewildering array of options, most sites prefer to guide the user through the process, validating each piece of data as it is provided.

This approach can be vulnerable to attacks. An attacker can use known URLs to jump between the different pages, in attempt to bypass some of the data validation checks. For example, if the application developer is not extremely careful, it may be possible for the attacker to add items to the order after having paid.

Solution

The *Directed Session* pattern exposes a single URL to the end user. All pages on the server are accessed using that URL. Session data stored on the server is used to determine which page is served. When no session data is present, the user is given the initial home page. As the user navigates the system, the current selected page is maintained in the session data.

This approach ensures that users are not able to request specific URLs and thereby bypass data validation checks. When a transaction consists of several separate pages, the user cannot request the second page until the first page has been validated. The application designer has some lenience here. If the user goes back to cached pages and resubmits an earlier page, it may be acceptable to accept that page and notes that all subsequent pages must be resubmitted. Alternately, the application designer can enforce a strict sequence and require that the user navigate using “forward” and “back” commands on the page itself.

Note that many IIS .ASP pages use an approach like this to regulate user transactions. While not intended for security, it can have positive impact on security. It also has some shortcomings from a usability perspective, particularly where the back button on the browser is concerned.

Related Patterns

- *Authenticated Session* – a related pattern that can use this directed session mechanism to enforce a particular session interaction.

- *Client Input Filters* – a related pattern that can use this directed session mechanism to enforce that client input is accepted and validated in a particular order.

- *Validated Transaction* – a complementary pattern for ensuring that client input validation is performed on all user input.

References

None.

Encrypted Storage

(Cryptographic Storage)

Abstract

The *Encrypted Storage* pattern provides a second line of defense against the theft of data on system servers. Although server data is typically protected by a firewall and other server defenses, there are numerous publicized examples of hackers stealing databases containing sensitive user information. The *Encrypted Storage* pattern ensures that even if it is stolen, the most sensitive data will remain safe from prying eyes.

Problem

Web applications are often required to store a great deal of sensitive user information, such as credit card numbers, passwords, and social security numbers. Although every effort can be taken to defend the Web server, one can never be sure that some new vulnerability won't be discovered, leading to the compromise of the server. Hackers are known to specifically target this sort of information.

Historically, Web sites that have experienced the loss of sensitive customer data have found it very difficult to recover from the adverse publicity. While many sites have recovered from the shame of being defaced, the large-scale loss of credit card numbers is a catastrophic failure.

Ultimately, it is always preferable not to store sensitive data. However, sometimes it is not avoidable. For example, credit card transactions are often not a single event. If an item is back ordered or the user requires a refund, the site must be able to access the credit card number that was used. Similarly, many government and financial sites rely on the social security number as the primary identifier for American users. These sites need a better approach to protecting this data.

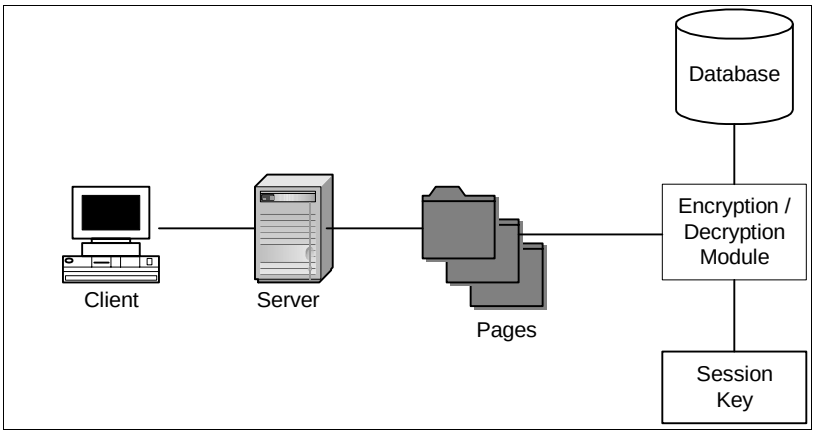
Solution

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

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

The key should be changed occasionally. This requires that the system be able to decrypt data using the old key and re-encrypt it using the new. Because of the complexity of encrypting and decrypting data on the fly, this should be performed with the database off-line during a period of

downtime. If downtime is not possible, a large key should be selected with the expectation that it will not be changed.



Server startup:

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

Receipt of sensitive data:

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

Use of sensitive data:

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

Key refreshing:

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

Issues

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

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

The following general principles should be followed:

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

Protection of the Key

If at all possible, the key should not be stored on the file system. There are COTS devices available that provide the system with a physical encryption card. These devices offer performance benefits, and also guarantee that the key cannot be read off the device. The key is manually loaded into the card, the encryption takes place on the card, and the key cannot be extracted from the card. The only downside to this approach is the cost – both the cost of purchasing the hardware and the development and maintenance costs of programming around it.

A cheaper alternative to loading the key is to require that an administrator load the key at system start, either from removable media or using a strong passphrase. This reduces the risk of having the key on-line, but does expose the key to a fair number of different people. This approach may sacrifice some availability because an operator must manually intervene whenever the system restarts.

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

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

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

Variation: One Key Per User

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

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

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

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

Possible Attacks

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

- Search of virtual memory – if sensitive data is paged out of RAM into a disk-based virtual memory file, it may be possible for an attacker to search the pagefile for obvious data

patterns (such as numeric strings that are recognized as credit card numbers).

- Key capture – an attacker will attempt to gain access to the key used to encrypt the data
- Dictionary attack – when encryption keys are generated from passwords, password-guessing attacks are generally much less difficult than exhaustive search of all possible keys.
- Race condition attack – an attacker may be able to capture the data before it has been encrypted.

Examples

The UNIX password file hashes each user's password and stores only the hashed form.

Several Web sites with which we are familiar use encryption to protect the most sensitive data that must be stored on the server. All use variations on this pattern.

Trade-Offs

Accountability	No effect.
Availability	Availability could be adversely affected if encryption keys are lost. If a global key must be reentered by a human operator each time the system restarts, availability can suffer in those circumstances as well.
Confidentiality	This pattern increases confidentiality by ensuring that the data cannot be decrypted, even if it has been captured.
Integrity	No direct effect.
Manageability	If system administrators must intervene to provide a global password on system restart, this will require around-the-clock administration. If individual passwords are used to encrypt data, administration will be complicated when users lose their passwords, or if other access to user data is required.
Usability	If individual keys are used, usability will suffer when a user loses his/her password and consequently loses all encrypted data.
Performance	This pattern will have a slight impact on performance due to the encryption algorithms and extra storage logic required.
Cost	Costs will be incurred from additional processing power required, additional management overhead, and the cost of adding encryption and decryption logic to the application.

Related Patterns

- *Client Input Filters* – a related pattern that verifies all data coming from the client.

References

[1] Landrum, D. "Web Application and Databases Security". http://rr.sans.org/securitybasics/web_app.php, April 2001.

Hidden Implementation

(Mini-Pattern)

Abstract

The *Hidden Implementation* pattern limits an attacker's ability to discern the internal workings of an application—information that might later be used to compromise the application. It does not replace other defenses, but it supplements them by making an attacker's job more difficult.

Problem

Before an attacker can wage a successful attack on a system, he or she must understand how the system operates. Current systems tend to provide the attacker with a great deal of information about both the standard components and the custom elements. As a result, it can be very easy for a potential attacker to assess whether an attack is likely to be successful—or at least go undetected—before having to do anything that might expose him or her to risk.

There are numerous specific occasions in which implementation details are revealed, including the following:

- Most Web servers greet the client with a detailed description of the Web server software, even going so far as to indicate the precise patch level.
- Many Web application systems use telltale file extensions (.asp, .jsp, .cfm) that indicate what sort of dynamic content generation is taking place.
- Web authoring environments often default to a standard directory layout that reveals the specific product used for generation.
- Application error messages can give detailed information about the software that failed and the nature of the failure.
- Field names in Web forms and hidden variables leak application variable names.

A savvy attacker can use all of this information to increase the effectiveness of an attack.

Solution

The *Hidden Implementation* pattern forces examination of all communication with the client for anything that might provide information about the internal workings of the system. If the system has already been designed, then consider implementing changes or installing filters that sanitize the data received by the client. Otherwise, the system should be designed in such a fashion that would make it impossible or difficult for an attacker to learn how the system is implemented.

One must examine the system to determine where information might be gathered and the potential impact this could have on the system.

- Understand what pieces of information are sensitive; consider how it could be modified or abused.
- Do not expose anything that is not absolutely necessary.
- Go back and look at the system from an attacker's point of view...what could the attacker learn? (See *Red Team the Design*.)
- Examine the information the user sees on several levels -- what does it reveal about the technology used to implement the system, the location of files, the content of the system data, etc., and consider how feasible it is to hide each of these.

All errors should be mapped to generic "unavailable" screens and the specific details logged to an internal debugging log.

Remove debugging capabilities, and comments in generated code.

Hidden Implementation presents some challenges:

- It is difficult to imagine all the negative uses of information.
- It may be difficult to hide all the important aspects of an application.
- The platform or tools used to build the system may not allow you to hide some implementation clues.
- It may require significantly more design and development time hide pertinent information.

Related Patterns

- *Account Lockout* – a related pattern; authentication failure screens can be made to reveal no information about the security policy. For example, a failed login screen can give the same generic message, regardless of whether the user ID was incorrect, the password was incorrect, or the maximum number of failed attempts was exceeded.
- *Authenticated Session* – a related pattern; sessions can be used to prevent leaking important information to clients, instead of telling the client what its username is you can instead assign the client a session identifier that maps to the information which is stored locally.
- *Minefield* – a related pattern; the application of implementation hiding principles cause the potential hacker to work much harder to understand the workings of the system. By making the reconnaissance more difficult and time-consuming, the attacker is more likely to be set off the well-tuned or customized intrusion detection system of a minefield.

- *Red Team the Design* – a related pattern in which red teams can examine the system from an attacker's point of view to look for parts of the design that reveal implementation details.

References

None.

Minefield

(a.k.a. Booby-Trap, Tripwire)

Abstract

The *Minefield* pattern will trick, detect, and block attackers during a break-in attempt. Attackers often know more than the developers about the security aspects of standard components. This pattern aggressively introduces variations that will counter this advantage and aid in detection of an attacker.

Problem

Any server that is accessible from the Internet will be attacked. No matter how much effort you expend in protecting it, the possibility exists that an attacker will penetrate the system. Once inside, an attacker who is familiar with the operation of a system will be at a huge advantage over one who has to perform lengthy, detectable reconnaissance activities.

Using standard Commercial-off-the-Shelf (COTS) software is an economic necessity. But attackers have access to the same COTS software, and thus are likely to know the operation of a system as well as its developers and administrator. Being intimately familiar with the internals of a system allows an attacker to operate with confidence, carefully avoiding standard detection mechanisms and covering his/her tracks afterwards. If a site is too cookie-cutter, attackers will even be able to use existing scripted tools to automate this process.

Solution

The *Minefield* pattern involves making slight customizations to a system. These modifications can have a number of different effects, depending on the attacker's skill and risk aversion:

- They might break compatibility with existing scripted attack tools.
- They might alert the operator to the presence of an intruder without the intruder's knowledge.
- They might cause the attacker to become uncomfortable enough with the prospect of being caught to cease attacking.

There is no cookie-cutter approach to developing minefields, as that would defeat the purpose. But there are a few basic approaches, including the following:

- Rename common, exclusively administrative commands on the server and replace them with instrumented versions that alert the administrator to an intruder before executing the requested command.
- Alter the file system structure.

- Introduce controlled variation using tools such as the Deception Toolkit.

- Add application-specific boobytraps that will recognize tampering with the site and prevent the application from starting.

Taken together, these techniques provide higher assurance that an attacker will not be able to attack the application server without being detected.

Issues

Customization

The single most significant security customization is to remove (or not install) the hundreds of tools, sample files, and potentially dangerous utilities that are part of any default installation, but not necessary for the server to execute. (See the *Build the Server From the Ground Up* pattern for more details.)

Simply moving files from the default locations can be enough to break many automated attack scripts. For example, many tools that probe for Web vulnerabilities are hard-coded to request files in the cgi-bin directory. Creating an alternative directory and placing cgi-bin files in that directory will cause many tools to fail. While something this simple will not deter a skilled adversary, it can cause unskilled attackers to leave a system alone.

Another useful technique is to change the disk location of the Web server's configuration and document files. If the apache configuration file is always located in /etc/httpd/config, a scripted attack could cause the file to be overwritten with a less secure version. Likewise, if the root Web page is stored at /var/www/html/index.html, it will be easy for a scripted attack to deface the site by replacing that page. Simply moving these files will thwart many tool-based attacks.

A more aggressive strategy is to use customization to make the server inhospitable to an attacker who actually gains access to the system. An excellent example of this described by Marcus Ranum is to rename all of the standard system utilities ("ls", "cd", etc.) to bizarre, non-standard names, and replace them with boobytrapped versions that will shut the system down. If legitimate administrators need access, they can load a secret alias file that maps all the names back to their standard value. An attacker who attempts to do anything on the system will cause it to be shut down almost immediately.

As noted in Security Assertion, it is also possible to develop application-specific tripwire-like functionality that shuts the system down if application data fails internal consistency checks.

Any customizations should be fully documented as part of the *Document the Server Configuration* pattern.

Detection

Custom boobytraps are most effective when carefully monitored for signs of an intrusion. Given enough time, a good attacker will learn to avoid the various mines and boobytraps. But initially,

the attacker will stumble across some of these devices. It is at this point that discovery of the attacker's presence is most possible.

In the example of the renamed cgi-bin directory, the system administrator should be aware of any attempts to access files in the original cgi-bin directory. Because no valid URL from the site will reference that directory, this is an obvious sign of mischief. Such activities should immediately result in an entry in a network address blacklist. If not blocked, they should be closely monitored for further evidence of attempted misuse.

In the example of the relocated apache configuration and html files, the original file locations should be carefully monitored. If these files change, the system administrator will know that a potentially devastating vulnerability exists on the Web server. At that point, monitoring should be increased in order to understand how these files were altered and to close the security vulnerabilities before they are exploited further.

Likewise, in the example of the boobytrapped system, if a boobytrap is triggered, the system administrator should perform a complete forensic analysis of the server in order to understand the extent of the damage.

In all of these cases, the existence of customizations allows for early discovery of security problems, but only if appropriate monitoring is present. A conventional intrusion detection system will not uncover these problems, unless it is customized appropriately.

Manageability

The biggest weakness of this approach is the adverse affect that it can have on system manageability. An unwary system administrator could easily set off the various boobytraps.

The extent of this problem will depend on the degree to which the server configuration is controlled. When the *Build the Server from the Ground Up* and *Use a Staging Server* patterns are properly employed, there should be no on-the-fly system administration performed on the production server. In this case, the customizations have no real affect on manageability. However, when the server must be actively administered, this pattern will be less appropriate.

Possible Attacks

Standard attack tools will cause a commercial intrusion detection systems to light up like a Christmas tree. When the system administrator has no recourse but to ignore these attacks, they represent a huge drain of personnel resources.

Distributed denial-of-service attacks against these systems can also limit the availability of the system.

Examples

There are products that implement some of the concepts in this pattern, such as Tripwire, Tripwire for Web Pages [2], and the Deception Toolkit [1].

Trade-Offs

Accountability	No direct effect.
Availability	If implemented in a fail-secure manner, this will adversely affect availability, by causing the system to halt when an intruder is detected.
Confidentiality	No direct effect.
Integrity	This pattern improves system integrity by increasing confidence that scripted attacks will fail and that human attackers will be detected before they can cause damage.
Manageability	This pattern will increase the management overhead of the application because it will require that log auditing be performed along with other security tasks. If the various sensors are too finely tuned, a significant administration burden will result. Customizations that are meant to trip up an intruder or break compatibility with attack tools can also trip up administrators and break compatibility with management and development tools.
Usability	Usability will suffer if sensors are overly sensitive, causing users to be denied access for too many legitimate mistakes.
Performance	No direct effect.
Cost	Development and documentation of custom configuration and monitoring mechanism is an additional one-time cost.

Related Patterns

- *Build the Server from the Ground Up* – a related pattern that provides more details about the trade-offs between developing custom components and using standard parts.

- *Network Address Blacklist* – a related pattern that discusses a mechanism for blocking network access to remote systems that have triggered numerous alerts.

References

[1] Cohen, F. and Associates. "The Deception Toolkit Home Page and Mailing List". http://all.net/dtk/dtk.html, 1998.

[2] Gillespie, G. "Saving face: Get Tripwire for Web Pages to protect your site against vandalism". http://www.computeruser.com/articles/daily/7,3,1,0620,01.html, June 2001.

[3] Ranum, M. “Intrusion Detection and Network Forensics”. USENIX '99, Monterey, CA, June 1999.

[4] U.S. Department of Energy Computer Incident Advisory Capability (CIAC). “J-042: Web Security”. http://ciac.llnl.gov/ciac/bulletins/j-042.shtml, May 1999.

Network Address Blacklist

(a.k.a. Client Filtering, IP Lockout)

Abstract

A network address blacklist is used to keep track of network addresses (IP addresses) that are the sources of hacking attempts and other mischief. Any requests originating from an address on the blacklist are simply ignored. Ideally, breaking attempts should be investigated and prosecuted, but there are simply too many such events to address them all. The *Network Address Blacklist* pattern represents a pragmatic alternative.

Problem

In a Web-based environment, where anonymous access is possible, there can be little to prevent an attacker from brazenly attacking a system. Even if their actions are detected, many attackers will proceed with impunity, knowing that their victims have little or no recourse. Unless the site is a commercial bank or a government agency, the FBI is too busy to investigate hacking attempts. When deterrence fails, there must be some other means of defending a site against anonymous attackers. It is not sufficient to simply rely on the strength of static defenses. Attackers that probe a site and find no resistance are likely to become more brazen.

Locking out individual accounts that appear to be under attack is an important defensive measure, but not sufficient to defend the site. Many hackers focus on network attacks and canned scripts that know nothing about individual accounts. Furthermore, a remote attacker could exploit the lockout mechanism, deliberately causing a large number of accounts to be locked out. Alternately, an attacker could choose a single weak password and then conduct an "account guessing attack" until an account using that password is discovered. The account lockout mechanism would never detect such an attack.

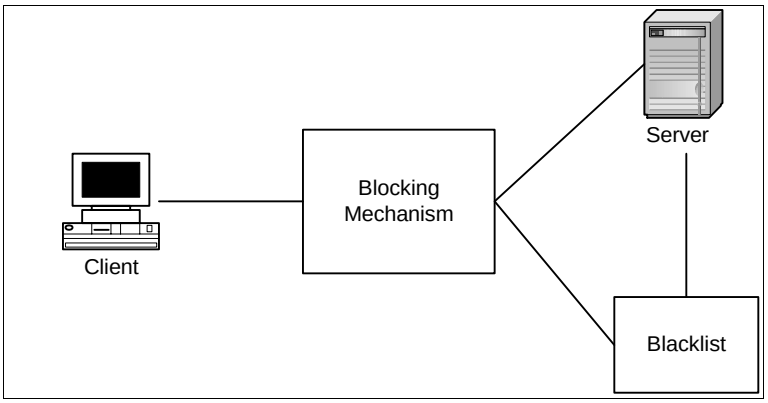
Solution

A network address blacklist mechanism maintains a list of network addresses that have exhibited inappropriate behavior. Once a given network address has exhibited enough such behavior to have obviously malicious intent, the network address will be temporarily added to a blacklist. When a request is received from a blacklisted address, it will simply be dropped on the floor.

All Web accesses are based upon TCP, which uses a handshake protocol to ensure that incoming IP addresses are not spoofed. While simple network-level denial-of-service attacks can be spoofed, more sophisticated HTTP-based malfeasance requires that the attacker be able to receive responses at the originating address. As a result, most attackers will have only have a handful of different IP addresses at their disposal. Blocking requests from a questionable IP address can effectively throttle many automated attacks.

The *Network Address Blacklist* pattern contains the folowing major elements:

- A client (with a network address) that is up to no good.
- The server, which is responsible for detecting and reporting misuse of the system.
- A blocking mechanism that is capable of intercepting client requests before they reach the server and dropping requests from blacklisted addresses.
- A blacklist that controls the blocking mechanism and responds to server requests to blacklist specific addresses.
- (Optionally) a human administrator that can manually remove addresses from the blacklist.



It is important to recognize that in many systems, the application developer will not have access to a fully automated blacklisting mechanism. In those instances, it is perfectly acceptable to allow a human administrator to act as the blacklist. The server will alert the administrator to possible misuse, and the administrator can choose to take manual measures, such as inserting a firewall or Web server access control rule that blocks that address. This is actually the most common form of network address blacklist – many application servers share either a single host, or are collocated behind a single firewall. In these instances, it would be inappropriate to allow a single application to block an address from accessing all the collocated applications.

In normal circumstances, a client that is not on the blacklist will make a benign request of the server. The blocking mechanism will let the request pass, and the server will respond with normal functionality.

When a non-blacklisted client makes a suspicious request, the server will keep a record of the network address and the nature of the request. If the request is sufficiently egregious (or the last in a sequence of requests that exceeds some predefined threshold of tolerance) the server will request that the address be blacklisted. The blacklist will configure the blocking mechanism to deny further requests from that address.

When a blacklisted client makes a request of any sort, the blocking mechanism will simply drop the request on the floor. Optionally, it may log the request for administrator audit.

After a period of inactivity from a blacklisted address, the blacklist mechanism will remove the address from the blacklist and configure the blocking mechanisms to allow further requests from

that address. Alternately, this can occur because of manual administrator intervention, possibly at a user's request.

Issues

Where to Block

If at all possible, the blocking mechanism should be implemented at the network level, either on a separate firewall or using host-based network filtering. This ensures that automated attack tools will be completely blunted. It eliminates the ability of the attacker to consume resources on the Web server. And keeping this component separate makes it easier to validate its proper operation.

There are circumstances where it may be either desirable or necessary to implement the blocking mechanism within the application itself. In such a case, the application would be structured roughly similar to the figure above. When a request is received, it is first checked against the blacklist before being dispatched to the appropriate page. This approach has the advantage of being entirely self-contained, not requiring integration with other components. It has several disadvantages: it adds complexity to the application, it allows blacklisted addresses to continue to consume server resources, and it cannot prevent attacks that exploit vulnerabilities to bypass the application.

When implementing a filtering mechanism it is generally a good idea to continue to log requests by blacklisted addresses. These logs may be necessary if an attacker must be prosecuted, or succeeds in compromising security. However, it is important that the logging mechanism not place a significant burden on the system. If logging events slows the system, stops the system when the logs fill up, or presents a significant auditing burden, an attacker can continue to cause mischief even after being blacklisted.

Administration of the Mechanism

As noted above, many implementations of this pattern provide a relatively static blocking mechanism, and rely on human intervention to make changes to the addresses being blocked. This has the advantage of simplicity. More importantly, it maintains a human in the loop – a fully automated system raises the danger of the mechanism failing, or being attacked, in a manner that denies service to large numbers of legitimate users. If the system will be manned around the clock, an approach that keeps the operator in the loop is preferable. However, if it will need to be unattended for a significant amount of time, a fully automated response should be considered.

When you block a network address, you run the risk of blocking access to an entire organization behind a firewall providing network address translation. This should not dissuade you from using this approach. If legitimate users within the organization complain about being unable to reach the site, they may be in a position to track down the misbehaving party, and even punish that individual.

If the blacklist mechanism is designed to eventually time out and remove addresses from the blacklist, this timeout period should be measured in days or even weeks. However, it will have to be fine-tuned in order to ensure that the list doesn't grow so large that it adversely affects performance. As noted below, the lockout mechanism should be fairly conservative – there should be no way that an honest mistake could result in lockout. This being the case, there is no need from a usability perspective to keep the time out period low.

It may be necessary to provide an interface to manage the blacklist, so that an administrator can remove addresses manually. However, this is not an interface that customer service representatives will need to be able to use. Customers will generally not even know their IP address and customer service representatives will not understand the security implications of de-listing an address that has been detected trying to actively hack the system.

Causes for Blacklisting

Some of the sensors that can be used to determine misuse by a network address are:

- When a single network address generates a number of requests for non-existent pages, they may be performing an automated search of the system.

- If a site does not use cgi-bin scripts, any attempt to request a cgi-bin script will not have come from a legitimate user clicking on a link. Many such requests indicate that an off-the-shelf Web vulnerability scan is being performed.

- If multiple invalid login attempts are received from the same network address, it may be a user who is attempting to guess usernames and passwords. Likewise, if a reserved name such as "guest", "root", or "administrator" is used, the client is likely up to no good. However, it is possible that a legitimate user has simply forgotten their username. In this case, the threshold for invalid attempts should be set quite high.

- If hidden fields or client cookies show clear evidence of tampering, the IP address should be blacklisted. However, this should be tested with a wide variety of Web browsers in order to avoid a large number of mistaken alerts.

- If the server receives requests containing invalid data that should have been filtered by client side validity checks. This is often an indicator that the user is deliberately bypassing those checks in hope that the server will act on the incorrect data. (See the *Client Input Filters* pattern.) It is important to recognize that in some cases—such as when JavaScript is disabled on the client or blocked by the client's firewall—the client side checks may be disabled through no fault of the user. If the site must be accessible to users who will not accept JavaScript, this should not be considered as evidence of attempted misuse of the site.

Possible Attacks

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

- Denial-of-Service Attack – an attacker may constantly hit a site after he/she has been blocked, causing that site to eat up resources to verify their address, or the attacker may

purposely block addresses so that legitimate clients sharing the same address cannot access the site. This pattern might be ineffective against large-scale, distributed denial-of-service attacks where many individual network addresses are used. Similarly, attackers may make it appear that the attack is coming from critical infrastructure such as a router, database, or firewall, causing you to block these servers, rendering the site in-operable.

- Resource consumption attack – attackers may grow the blacklist to a large size so that it is inefficient to examine the blacklist, causing a general degradation in performance.

Examples

Manual implementations of this pattern are very common. Best practice in Web security is to implement some sort of intrusion detection, even if it consists only of auditing the usage logs. Administrators that observe repeated misuse originating from certain IP addresses generally contact the ISP in question. If this fails to stop the behaviors in question, the administrator generally has the capability to block further access via a firewall rule.

Commercial intrusion detection systems often incorporate some sort of blocking mechanism. ISS RealSecure, for example, implements "firecell rules," which allow known attacks to be blocked in real time. This capability can be integrated with certain commercial firewalls, to allow blocking rules to be inserted into the firewall in real time. It should be noted that these are good at detecting network-level attacks, and even attacks on Web server vulnerabilities. But they will not detect application level misuse out-of-the-box. Instead, they offer APIs that allow application developers to report suspicious activities that are specific to the application.

We know of Web applications that track invalid login attempts by IP address. Once a certain number of invalid login attempts have been observed originating from a specific IP addresses, further login attempts are simply ignored. This ensures that the account lockout mechanism cannot be misused to effect denial of service conditions on more than a handful of accounts.

A similar approach has been used to combat unsolicited junk e-mail. The Maps approach maintains a large list of blacklisted IP addresses. As IP addresses are observed as the origination point for SPAM, they are added to the blacklist. The blacklist is made available to servers across the Internet, which discard mail from any blacklisted address. For more details, see http://mail-abuse.org.

We are currently developing an example of a fully automated mechanism that lets Java servlets insert NetFilter ("iptables") rules on the system hosting the application. The source code will be made available on-line upon completion.

Trade-Offs

Accountability	The network address blacklist is necessary when there is no other recourse, and by definition no accountability. In some sense, denying further access is a form of punishment and could be considered a measure of accountability.

Availability	A network address blacklist can have a negative effect on availability. Specifically, the network address blacklist offers an interface whereby specific network addresses can be blocked from further access: if this interface is misused or implemented incorrectly, it could cause legitimate users to be denied access. Also, all users whose requests originate from behind the same firewall may appear as if coming from the same network address. It is conceivable that one user could cause an entire enterprise to become locked out of the Web site.
Confidentiality	When implemented effectively, a network address blacklist enhances site confidentiality and integrity. This occurs because it will dissuade or prevent attempts to misuse the Web application (since many forms of misuse are intended to compromise those security characteristics). Even if ineffective, it will not adversely affect these properties.
Integrity	See Confidentiality.
Manageability	A network address blacklist can have a positive effect on the administrative burden of the Web site. By eliminating many nuisance attacks, the system administrators can be freed to investigate more significant security concerns. However, if legitimate users are finding themselves locked out on a routine basis, the management burden will be raised, particularly if administrators have to intervene manually to clear an address from the blacklist.
Usability	If the server is tuned too sensitively, it may cause legitimate behaviors to be reported as suspicious. This could have a very negative effect on usability. But if the server is fairly conservative, legitimate users should never see any impact on usability. Those who attempt to misuse the system will see a significant negative effect on their usability of the system.
Performance	A network address blacklist has mixed effects on performance. On the one hand, misbehaving users who might otherwise be consuming considerable resources are prevented from gaining access to the site. On the other hand, the lockout mechanism could itself incur significant performance penalty on all users. If a network address blacklist is implemented in a way that accesses a database for every Web request, the performance will suffer. If the lockout depends on an in-memory blacklist, it is important that the size of the list be bounded and that the list does not itself become a bottleneck.
Cost	Manual administrator lockout adds little additional expense. If automated lockout can be implemented using existing APIs, it can be developed quite cheaply. If novel interfaces are required, it will be

	quite expensive to develop. In either case, quality assurance will be non-trivial.

Related Patterns

- *Account Lockout* – a related pattern that supplements, and should be coordinated with, the network address blacklist.
- *Client Input Filters* – a related pattern that provides a source of information about suspicious activities.
- *Log for Audit* – a related pattern.
- *Minefield* – a related pattern that provides input to the network address blacklist mechanism.

References

[1] Brown University Computing and Information Services. "IP Address Filtering". *Web Authorization/Authentication*. http://www.brown.edu/Facilities/CIS/ATGTest/Infrastructure/Web_Access_Control/GoalsOptions-ver2.html#anchor136476, October 1997.

[2] Stein, L. and J. Stewart. "The World Wide Web Security FAQ – Version 3.1.2". http://www.w3.org/Security/Faq, February 2002.

Partitioned Application

(a.k.a. Security Kernel, Insulation)

Abstract

The *Partitioned Application* pattern splits a large, complex application into two or more simpler components. Any dangerous privilege is restricted to a single, small component. Each component has tractable security concerns that are more easily verified than in a monolithic application.

Problem

Complex applications often require dangerous privileges in order to perform some of their tasks. These privileges are often only needed by a small part of the application, but the existence of these privileges makes the entire application dangerous. If any part of the application can be compromised, the dangerous privileges could be misused. As a result, it is very difficult to have confidence that these privileges won't be abused – the entire application must be free of exploitable flaws.

There are many examples of this. The IIS Web server requires administrative privileges in order to execute. It is a large, complex application in which mistakes are frequently found. Because it executes with administrative privilege, every one of those mistakes could potentially lead to remote compromise of the entire system. Similarly, the UNIX sendmail program is a large, complex program that runs with administrator privilege in order to be able to write e-mail into any user's incoming mail files. For years, errors were found in sendmail that would allow remote attackers to gain complete control over the system.

Even when this approach does not lead to compromises, it causes the quality assurance burden to be significantly increased. Developers and quality assurance engineers are never really sure that some minor change won't have some unanticipated effect on security. As a result, routine maintenance activities are often inhibited by the need to perform full-blown security assessments of even the smallest change.

Solution

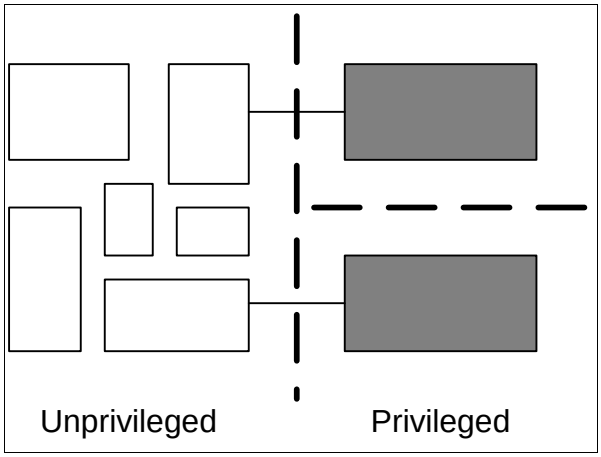
The *Partitioned Application* pattern divides a complex application into smaller components that minimize and isolate the elements that require dangerous privileges. Existing protection mechanisms are used to assign privileges to each component and prevent the various components from interacting with one another except through the published interfaces. The privileged components each provide some single privileged service to the rest of the application.

Each privileged component publishes an interface that the remainder of the program can use to perform privileged actions. For example, the postfix application (a more secure alternative to sendmail) contains a "maildrop" component that is responsible only for writing mail to user accounts. The maildrop component has administrative privileges, but it is a small component

that thoroughly checks all inputs it receives and writes only valid data to valid mail user's mail queues. Because the maildrop component is so simple, it has been extensively analyzed for possible security problems. The remainder of the application can be updated as necessary without concern that the administrative privileges might somehow be misused.

There is no one-size-fits-all approach to developing a partitioned application. Much like the problem of assigning functionality to specific objects, the problem of developing an appropriate application partition is a craft. The models presented here are general suggestions; specific examples provide far more concrete details.

By isolating the privileged elements into one or more relatively small components, significant security issues can be restricted to just those components. The privileged components can be subjected to rigorous quality assurance, and the functionality of the non-privileged components can be updated with less worry about the security impact.



Issues

Depending on the partition, it may be important that one component be able to have confidence that it is talking to a specific component and not an impostor. This can generally be ensured using queues in which only a specific object is able to read from the queue. If each object has such a queue, mutual authentication can take place by using simple handshakes delivered through the queues. The appropriate queue should be read from a controlled configuration file, not accepted as provided by the caller.

A partitioned application can impact performance, as inter-process communication is introduced and synchronization is required. Benchmark and understand any performance penalties before committing to a partitioned design.

A partitioned application introduces logs and other intermediate message queues that can fill up; appropriate responses to jammed queues should be specified, implemented, and tested.

Installation / Configuration

This pattern also encompasses the installation and configuration scripts and files used to assign privileges and allow components to locate one another at run-time. Using operating system privileges, this could include the creation of user accounts, the establishment of privileges, and the deployment of scripts that ensure that all the components will start correctly and be able to find one another.

Installation of a partitioned application should address the following:

- The components need to be registered with the underlying protection mechanisms. Typically, this means creating multiple user accounts for each of the major components.

- The components will each have a configuration file that identifies each of the other components and interfaces. These names should be absolute, not relative. Specifically, never rely on the PATH or other environment variables to locate a component – hardcode the entire pathname into the configuration file.

- Create any resources needed by the privileged components. Changes the access rights on all the resources and configuration files so that they can only be accessed by the owning account.

- Configure the components so that they will run under the appropriate identities.

- Install a startup script that permits each of the privileged components to be brought up in the appropriate order.

A partitioned application will often require that components are initialized in a particular sequence. Determine the correct startup sequence and ensure that it is enforced. Depending on the design, it may be necessary to stop the entire application if any one component fails to start.

Possible Attacks

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

- Direct invocation of an internal interface – early versions of postfix contained an internal interface that if manipulated directly would allow privileged commands to be invoked.

- Race condition problems – if an attacker can deliberately slow some component, it may be possible to cause inconsistencies that never appeared in testing. Partitioned applications are susceptible to race condition bugs. There is an entire class of security vulnerability called “time of check, time of use” (TOCTOU) errors.

- Man-in-the-middle attack – numerous mistakes have been made in which applications invoke other components by depending on an environment variable (or the path) to locate the appropriate file. When this approach is used, it is possible to intercept the invocation request and cause the invoking application to start an untrusted process. From this it is possible to conduct a man-in-the-middle attack.

- C++ language protections – when partitioned applications are built using the C++ language protection mechanisms, it is possible to bypass these protections and access private functions directly. A number of advanced C++ programming texts provide guidance for directly manipulating C++ virtual function tables.

Examples

There are numerous examples of the *Partitioned Application* pattern. Interestingly, examples can be found at different abstraction levels. This section presents examples at the code, system, and network levels.

In addition, we will eventually include the code from our password management object from our Web repository application as an example of a partitioned application. The Java object for password management will be called by less trusted servlets and will be the only object able to access the application password file.

Code Level

At the code level, many object-oriented programs are constructed as partitioned applications. Most applications written in an object-oriented language use the Private keyword to hide implementation details of an object. These features are useful in protecting against unintentional errors. For example, a dangerous function call can be wrapped using an object that checks consistency before invoking the function. *These features can often be circumvented by a savvy attacker though.* In C++, for example, the application as a whole must have the privileges to execute the dangerous instruction, and there is nothing preventing some other part of the code from invoking the function.

Java, on the other hand, protects objects from one another and allows specific privileges to be granted on a per-object basis [2]. Thus, Java can be used to develop partitioned applications if object access modifiers are used appropriately. This makes Java less susceptible to the attacks that C++ programs are.

System Level

Kernel-based operating systems are the archetypal system-level example of partitioned applications. Operating systems must be able to execute privileged instructions and directly manipulate the hardware resources of the machine. But operating systems also contain millions of lines of code that perform no such privileged work. Operating system design has embraced the notion of a kernel, in which only a very small subset of the system runs privileged code [5]. Functions such as direct manipulation of the system memory tables and swapping of running processes are performed within the kernel. Other functions, such as managing the file system, run as unprivileged code but are able to make carefully controlled requests of the kernel.

As a counterexample of this, Windows NT v.4 moved its entire graphics and user interface subsystems into kernel mode code, where any bug is a potential breach of system security. The rationale given was that performance was increased, and any bug in that code would eventually lead to a system crash in either case [4].

Another set of applications that are partitioned as described in this pattern are replacements for sendmail. Sendmail is an extremely complex UNIX mail processing system that has been the home of innumerable security-relevant bugs. As a result, several replacements to sendmail have been developed, most notably qmail [3] and postfix [1]. Both of these programs are instances of the *Partitioned Application* pattern. The "maildrop" elements of the program, which need access to any user's e-mail queue, are carefully restricted to very small, separate executables that have been subjected to intense scrutiny.

Network Level

By definition, most distributed network applications are partitioned applications. While they are not usually partitioned for security reasons, security does impact their design. Most client-server applications place all of the low-risk user interface activities on the client and ensure that the server will only execute authenticated, validated transactions.

Web front ends to banking applications are very common examples of network-level partitioned applications. Many Web systems isolate the application that performs electronic funds transfers or credit-card transactions on a separate system. Access to this system is through a minimal interface, often implemented using an internal firewall. While the rest of the banking application must still be careful to ensure that only valid transactions are initiated, the partitioned model guarantees that audit rules and certain safeguards cannot be circumvented. For example, charges for negative values can be discarded, a shipping address can be validated against the billing address, and the credit card number can be prevented from ever being disclosed to a client.

Trade-Offs

Accountability	No direct effect.
Availability	In some cases, this pattern can increase availability. For example, the sendmail replacements can continue to receive mail even if an internal processing component has died. However, the interfaces between components might also introduce new avenues for resource consumption attacks that result in a denial of service. In addition, the increased processing overhead can also affect availability adversely under heavy load conditions.
Confidentiality	This pattern can increase confidentiality by simplifying the security-critical aspects of the design and containing the damage that could be caused by many bugs.
Integrity	See Confidentiality.
Manageability	It is generally harder to deploy a partitioned application appropriately than its monolithic cousin. Each of the interfaces between the components must be understood by the administrator in order to debug problems or avoid dangerous misconfiguration. It can be very difficult

	to ensure that elements are all started in the correct sequence. While installation scripts can help ease this burden, they cannot eliminate it entirely. The maintenance of a partitioned application can be easier though, if the components and interfaces are well designed. Monolithic applications can be exceedingly difficult to maintain and enhance; a partitioned application could be easier to understand due to its modular structure.
Usability	No direct effect.
Performance	The primary drawback of this pattern is its impact on performance. Each of the restricted interfaces introduces overhead and processing delays. For example, the message queues used by sendmail alternatives introduce significant processing overhead.
Cost	A partitioned application can have both negative and positive affects on development costs. Structuring an application around security requirements can have an adverse impact on other development priorities. Complex installation scripts and documentation requirements introduce further costs. But enforcing a discipline of application decomposition can permit more focused testing and greatly improve quality assurance.

Related Patterns

- *Log for Audit* – a related pattern alerting of the need to maintain consistent logs across components.
- *Password Propagation* – a related pattern stating that, where possible, trust in any single trusted proxy should be minimized.
- *Server Sandbox* – a related pattern advising that sandboxes be built around individual components of a partitioned application, such as servers that should not be trusted.
- *Trusted Proxy* – a related pattern where components protecting critical resources are viewed as trusted proxies.

References

[1] Blum, R. *Postfix*. Sams Publishing, 2001.

[2] Gong, L. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[3] Nelson, R. "The qmail home page". http://www.qmail.org/top.html, June 2002.

[4] Solomon, D. *Inside Windows NT Second Edition*. Microsoft Press, 1998.

[5] Tanenbaum, A. *Operating Systems: Design and Implementation.* Prentice-Hall, 1987.

Password Authentication

(a.k.a. Client Login)

Abstract

Passwords are the only approach to remote user authentication that has gained widespread user acceptance. Any site that needs to reliably identify its users will almost certainly use passwords. The *Password Authentication* pattern protects against weak passwords, automated password-guessing attacks, and mishandling of passwords.

Problem

Many servers require that users be identified before using the system. Sometimes, this is for the benefit of the user, protecting the user's data from access by others. And sometimes it is for the benefit of the system, ensuring accountability should the user misbehave or attempt to repudiate transactions performed in his or her name. In either case, a password—a secret shared between the user and the system—is the most practical form of authenticating a user's identity. Other authentication mechanisms are possible, but only passwords have gained widespread adoption.

Because passwords are so familiar, many system architects have developed their own password authentication schemes. These schemes are often subtly flawed, failing to take into account the dangers specific to remote Internet authentication. If improperly implemented, a password scheme can fail in many different manners:

- It can allow users to access accounts to which they should have no access.
- It can leave cause users to lose faith in the security of the system.
- It can fail to provide accountability for disputed transactions.
- It can adversely affect usability of the site, frustrating users and losing business.
- It can cause a tremendous drain on system administrators or customer service.

Many development platforms and development environments provide password authentication mechanisms. Often, the benefits of using a canned system outweigh the disadvantages. But even when such a system is used, it is important to understand the possible ramifications.

Solution

The *Password Authentication* pattern identifies specific Web pages, documents, and transactions as requiring user authentication. Any time a protected resource is requested, the server will require that the user provide an identity, as well as proof that the claimed identity is valid. If the user supplies the appropriate credentials, the requested page will be delivered. If not, the user

will be directed to a login screen at which time he/she will be prompted to provide his/her username and password.

The *Password Authentication* pattern stores each password in encrypted form ("hashed") on the server. Each password is optionally hashed using additional randomly selected data ("salt") that is unique to the account. When the user provides their password, the hash is recomputed and checked against the stored copy. By not retaining copies of the unencrypted passwords, the Web site ensures that a compromise of site security will not result in a large-scale loss of passwords.

The *Password Authentication* pattern also encompasses the policy for the creation, storage, and alteration of passwords, as well as the recovery of lost passwords.

This pattern focuses on the simplest form of password authentication: the authenticated transaction. In an authenticated transaction, the user supplies his or her username and password on the same form as other transaction data. There is no notion of "logging in" – each transaction is validated individually. If the user provides an invalid username or password, he or she is returned to the transaction screen with an error message.

An important variation is the *Authenticated Session* pattern, in which the user provides his/her username and password once, then the appropriate credentials are cached and automatically reused for further requests. See the *Authenticated Session* pattern for additional details that are specific to that approach.

While the authenticated session is arguably more useful, there are many situations where session semantics are not necessarily desirable. When authenticating high-value transactions, it is sometimes appropriate to require the individual transaction to be authenticated. Specific examples include:

- When submitting a bid on e-Bay, the system asks that the username and password be supplied along with the bid amount.

- When initiating a funds transfer request at a banking site, the system generally asks for specific approval for that transaction, even if the user has already logged into a valid session.

- When attempting to change a password, most systems require that the user provide the old password, even if already logged into the system.

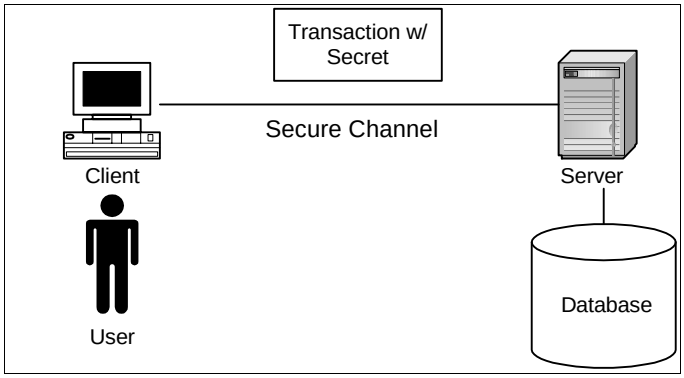
To protect the password in transit over the network, SSL or other application-level encryption mechanisms should be employed. This is critical for any high-value transactions. Low-value transactions, such as retrieving personal Web-based e-mail

To protect the account from automated password-guessing attacks, see the *Account Lockout* pattern.

Password authentication only provides a user's identity. In order to establish policies about which users should have rights to which resources, see Access Control.

A password-authenticated transaction consists of the following steps:

- The client delivers a request to the server. The request includes the user's identity and password.
- The mediation component on the server receives the request
- The mediation component determines that the request is for a protected resource
- The mediation component checks the account name and retrieves the password hash and optionally any password encoding data ("salt") specific to that account.
- The mediation component hashes the password (using any salt) and compares the result to the stored password hash
- If the result is a mismatch, the server redirects the user back to the transaction screen with a message indicating that the username / password combination was in error.
- If the result is a match, the server delivers the request and associated transaction data to the protected resource for processing.



Initial password selection. Depending on the enrollment pattern selected, the user may need to enter an initial password. In this case, the system will provide the client with an enrollment form that includes a text input box for the password. As a safeguard against typing errors, the page should require that the user enter the password twice. When submitted the copies of the password are checked against each other, and a new account created. The salt, if any, is selected and stored with the account data. And the password is hashed and the hash stored in the account. If password recovery options are used, the appropriate questions and answers should be included in the enrollment form and stored with the account information.

Password change. This is actually just another password-protected transaction, in which the transaction is the selection of a new password. Again, the user should provide two copies of the new password. And even if a session mechanism is in place, the user should be forced to provide the old password as part of the change request.

Forced password change. Depending on the enrollment pattern selected, it may be necessary to provide the user with an initial password that should be changed on first usage. In such situations, the user account can be marked as requiring a password change. Before any

authenticated transactions can take place, the user must first perform a password change transaction, as described above.

Manual password reset. Users have a habit of forgetting their passwords. When a user contacts customer/user support, the administrators much have processes for authenticating the user, and mechanisms for either resetting the account to a standard password, or entering a randomly-selected password that can be communicated to the user out of band (e.g. over the telephone). In either case, the account should be marked to force a password change on first usage.

Password recovery. Some systems provide an alternative authentication mechanism. Typically, when the user first enrolls, he/she must also provide a list of questions and answers that should be private to the individual. Sometimes these are selected from a list (e.g. "what's your favorite sports team", "what was the name of your first boy-/girlfriend", "what is your hometown"). In any case, if the user forgets his or her password, these questions can be used to implement an automated password recovery scheme where a form prompts the user for each answer and allows a new password to be chosen if each of the questions is answered correctly. Alternately, this sort of information can also help administrators manually authenticate users who call in requesting a password reset.

Issues

For issues relating to lockout from password-protected resources, see the *Account Lockout* pattern.

Protecting password in transmission

For high valued transactions, passwords should always be communicated securely. Any page that requests a user password should be encrypted using SSL. Furthermore, users should be informed to always look for the lock/key indicating a secure channel. It is generally poor practice to deliver a login page that is unencrypted with a submit button pointing to a SSL protected page, because the user's browser will not provide the user with visual feedback that the submission will be protected. Any site that does not encrypt all password submissions will have trouble claiming accountability or defending itself against claims that it allowed user data to be divulged/compromised.

Nevertheless, it is not an ideal world and SSL encryption can be very expensive to implement. If the data being protected is not that valuable, it can be appropriate to use passwords without encryption. For example, on-line fantasy baseball systems that offers no cash prizes, the personalized HBO schedule, and even some ISP Web-based e-mail systems all use passwords without SSL. But any system that collects sensitive data such as social security numbers, credit card numbers, or bank account information should always use SSL to encrypt passwords in transit.

There is an alternative approach to protecting passwords in transmission that does not rely on an encrypted session. This alternative is to hash the password on the client, and transmit only the hashed password. To prevent the hash from being captured and reused, the server should challenge the client to with specific information that should be included as part of the hashing

operation. The http protocol actually includes this capability as part of the “digest authentication” mechanism, but it failed to gain popularity because many early Web browsers did not support it. Proprietary clients, such as Java applets, can use this approach to communicate passwords securely between the client and the server. It should also be recognized that this approach will protect the password, but cannot prevent a network eavesdropper from capturing the hash and using that to tamper with the transaction in question.

Protecting passwords in storage

Passwords should not be stored in plaintext. Instead, when a user first enters a password, it should be hashed with a one-way function and the hash stored in its place. On subsequent authentication attempts, the hash should be recomputed and compared with the stored value. If the user's password must be stored in the client browser (e.g., within a cookie), it should be encrypted using a session key that will expire after some reasonable amount of time.

Hashed passwords should never be encoded as part of a URL. If a user bookmarks the page, any subsequent user of the system will be able to circumvent authentication. If URLs must be used, one possible implementation strategy is to encrypt the hashed password using a short-lived session key (which is global to all users of the system) before being stored in the URL.

Choosing passwords

Best practice is to allow users to select their own passwords. Some systems provide the user with a randomly generated password in order to protect against poorly chosen passwords. However, this approach generally leads to a significant number of lost passwords, resulting in greater management costs. It also has been found to have adverse effects on usability. In addition, users tend to write these passwords down and leave them near the computer.

Users should be allowed to select passwords from the entire printable ASCII character set (codes 32-127). Reducing the character set to only letters and numbers provides no savings and drastically reduces the number of possible password combinations. Users should be allowed to select reasonably long passwords (12-15 characters) if they choose.

Users generally pick very weak passwords. This should be mitigated by (a) explaining to the user the importance of picking a strong password (b) providing guidance on what makes a password strong (c) forcing minimum length and variety on chosen passwords. Some system administrators even go so far as to run automated cracking tools against their own users' passwords. Unfortunately, the processing cost of running these tools is prohibitive in a Web environment.

Generally, it is not good practice to require Web users to change their passwords. Many Web sites are not visited regularly, and enforcing frequent password expiration would only increase the management burden and decrease usability without appreciably increasing security.

User Education

Users must be educated about the importance of protecting their passwords. They must not divulge passwords to others. They should not allow the browser to cache their password,

particularly if logged in from a public system. They should never reveal their password to anyone: best practice is that customer service representatives will never ask a user for their password. And they should always check that their communication is encrypted (and be aware of the URL and alert to any browser security warnings) before entering their password.

When the user logs in successfully, it is a good idea to inform the user of the number of failed login attempts since the last successful login. A user who mistyped his/her password will recognize that the invalid attempts were legitimate. But the user whose account is under attack will be alerted to the fact and may make the system administrators aware of the problem.

Session-Related Material

Do not rely on the HTTP basic authentication model for protection of a Web application. Basic authentication provides no protection against password-guessing attacks. Furthermore, it allows users to cache their passwords for the sake of convenience. This undermines the protection and the accountability offered by passwords.

Possible Attacks

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

- Network sniffing – when SSL is not used, it is easy to capture passwords by sniffing the network. For Web-based applications, the greatest risk comes from those on the same local area network as a specific targeted user, or a compromised system at the Web site itself. Protect against this attack by using SSL for any page where a password is entered.

- Keyboard sniffing – users can be fooled into running malicious code, such as viruses. A keyboard sniffer captures all user keystrokes (including passwords) directly from the keyboard and makes them available to the writer of the sniffer.

- Improper caching of passwords – if a password is stored in a permanent cookie, encoded in URLs, or stored by the browser as a convenience to the user, it is possible for somebody else to authenticate themselves as that user by merely gaining access to the computer. Even a non-permanent cookie is a danger if the user does not close the browser after logging out of the site.

- Password guessing – it is often possible to guess a user's password merely by trying some of the most common passwords (e.g., “password” or “secret”). Alternately, if something is known about the user, it can be easy to guess his/her password. For example, a child's or spouse's name is a very common choice.

- Sticky notes – many users write their passwords on sticky notes and attach them to their monitors. Looking around the user's desktop for login instructions is a very common attack.

- Social engineering against the user – attackers have been known to call users and pretend to be customer service in need of the users' passwords.

- Social engineering against customer service – conversely, attackers often contact customer

service pretending to be a user who has lost his/her password.

- Compromise of password store – if a site is compromised, passwords that are stored in the clear can be stolen in bulk. This incurs a massive management burden by forcing the site to establish a new password with every user. And because users typically use the same passwords at a number of sites, it can result in huge inconvenience to the whole user community. (See the *Encrypted Storage* pattern to address this attack.)

- Trojan horse login screens – attackers have been known to mail phony URLs to users, instructing them to login to that URL as a part of a site upgrade. The Trojan horse site then collects account names and passwords for the real site.

Examples

Systems often use passwords to authenticate interactive user sessions. This pattern is focused more on distributed applications. The Common Criteria [2] and FIPS 112 [3] provide detailed instructions on using passwords in conventional (non-distributed) systems.

Every major Web site that authenticates users uses passwords. The vast majority of sites protecting high-value items (e.g. banking) use passwords in a manner very similar to this pattern.

Trade-Offs

Accountability	Password authentication permits individual users to be held accountable for their actions. However, passwords must be carefully protected in order to maintain accountability. In addition, users should not be able to claim ignorance—they must be informed of appropriate password handling.
Availability	If an overly complex password scheme is used (e.g. exactly 12 characters, three of which must be punctuation marks), users will likely find themselves forgetting their passwords on a regular basis, making the site unavailable to them.
Confidentiality	Password authentication allows user data to be protected from unauthorized disclosure.
Integrity	Password authentication allows user data to be protected from unauthorized modification.
Manageability	The manageability impact of using passwords is significantly lower than most other authentication technologies. However, the impact on manageability will depend on choices made in the implementation of the pattern. If customer service intervention is not required for either lockout or lost passwords, the management burden is very low. If either require manual intervention, manageability will be more

	difficult depending on the complexity of the passwords and the lockout threshold.
Usability	Passwords are a nuisance that users are willing to live with. Forcing frequent re-authentication does have an adverse effect on usability. Enforcing long, complex passwords also impacts usability. Allowing the browser to cache passwords would seem to have a positive effect, but it encourages users to not remember their passwords and there are many circumstances where they may need the password when away from the browser. The impact on usability of using passwords is significantly lower than most other authentication technologies.
Performance	Using SSL to protect passwords in transit will have a significant impact on performance. Even with hardware acceleration, a server using SSL can handle far fewer connections than a standard HTTP server.
Cost	While the cost of using passwords is significantly lower than most other authentication technologies, there can be significant costs associated with passwords, such as the cost of SSL servers. Using customer service to deal with lost passwords and locked out accounts can be expensive. Also, the quality assurance burden associated with protecting all sensitive pages can be significant.

Related Patterns

- *Authenticated Session* – a related pattern in which passwords are used to authenticate user interaction with session semantics (as opposed to a single transaction in this pattern).

- *Enroll without Validating* – a related pattern that presents a procedure and circumstances for when initial authentication credentials are not required.

- *Enroll with a Pre-Existing Shared Secret* – a related pattern that presents one procedure for communicating the initial authentication credentials to a user.

- *Enroll by Validating Out of Band* – a related pattern that presents one procedure for communicating the initial authentication credentials to a user.

- *Enroll using Third-Party Validation* – a related pattern that presents one procedure for communicating the initial authentication credentials to a user.

- *Network Address Blacklist* – a related pattern describing a protection mechanism from misbehaving clients that perpetrate password-guessing attacks on multiple accounts.

- *Encrypted Storage* – a related pattern that can protect against compromise of the password

store.

References

[1] Chun, M. "Authentication Mechanisms - Which is Best?" http://rr.sans.org/authentic/mechanisms.php, April 2001.

[2] Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology Security Evaluation Version 2.1*. http://www.commoncriteria.org/cc/cc.html, August 1999.

[3] National Computer Security Center. *DoD 5200.28-STD, Trusted Computer System Evaluation Criteria*. December 1985.

[4] National Institute of Standards and Technology (NIST) Information Technology Library (ITL). "Federal Information Processing Standards Publication 112: Password Usage". http://www.itl.nist.gov/fipspubs/fip112.htm, May 1985.

[5] Wheeler, D. "Secure Programming for Linux and Unix HOWTO – v2.965". http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html, May 2002.

Password Propagation

Abstract

Many Web applications rely on a single database account to store and manage all user data. If such an application is compromised, the attacker might have complete access to every user's data. The *Password Propagation* pattern provides an alternative by requiring that an individual user's authentication credentials be verified by the database before access is provided to that user's data.

Problem

Web applications generally require a user to provide a password in order to access his/her account data. However, it is often impractical to create a separate user account on the database for each Web user. Many database packages do not support a very large number of user accounts. And there is no standard SQL interface for managing user accounts and privileges. As a result, most Web applications do not rely on the database to maintain individual user accounts.

Most Web applications rely on a single database account to store all user account information. The application server provides a hard-coded password to the database in order to gain full access to all application data. The application server is responsible for authenticating the user and restricting access to only the data associated with that user. This makes the application server a trusted proxy: if it is compromised, every single user's data is placed at risk.

A specific example of this is the Netscape electronic commerce server, ECXpert. ECXpert relies on an Oracle database to store individual customer data. Every customer has an individual username and password, which are stored in the accounts table in the database. But all customer data (including the accounts table) is stored under a single Oracle username ("ECX"). If the ECX account is compromised, an attacker will have full access to all user data.

This is a very common problem. A few years ago, it was discovered that adding a space to the end of a ColdFusion URL would cause the ColdFusion server to reveal the page source instead of executing the page on the server. Attackers who used this technique were delighted to discover that many applications stored the database account name and password within the ColdFusion source code. As a result, a great number of systems were compromised.

Solution

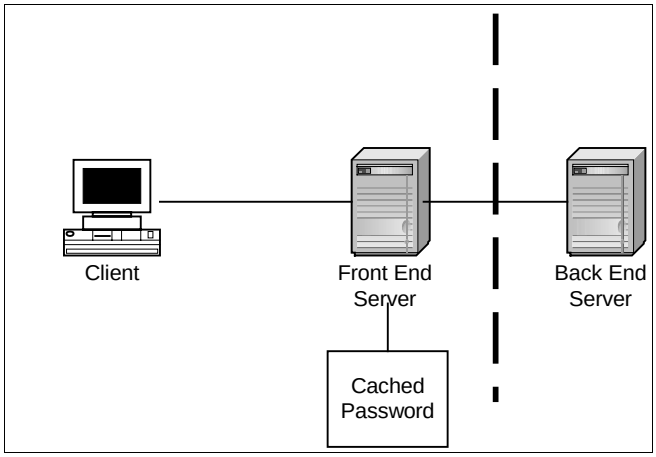
The *Password Propagation* pattern splits the application into a front end and a back end. The front end is responsible for interacting with the user and presenting data in a formatted manner. The back end performs transaction processing and manages individual user account data. The front end has no direct access to user account data – it must go through the back end.

When the front end authenticates a user, it caches that user's password as part of the session data stored on the server. When the front end initiates a transaction or accesses user account data, it presents the password to the back end. The back end validates the user's password before

allowing any access to the user's data or performing any transactions on the user's behalf. When a user logs out or his/her session expires, the password is flushed from server memory.

If the Web server front end is compromised, the attacker will not have the privileges to access or alter individual user account data. Any requests of the back end will require that the appropriate password be provided as part of the request. This ensures that the front-end cannot be tricked into accessing other user's data because the back-end will reject requests that do not include the appropriate password. If the front end is completely compromised, only the passwords of currently logged-in users are at risk.

The *Password Propagation* pattern is a specific instance of the *Partitioned Application* pattern. As such, the back end still represents a trusted proxy. If the back end is compromised, all user data can be accessed, as before. However, a well-constructed back end will offer a constrained interface that is easier to protect than an entire Web server.



There are many ways in which this pattern can be implemented. Some of the more straightforward are:

- When creating a Web front end to a legacy system, the legacy system may already enforce password authentication of all transaction requests. In this case, the legacy system can be used as the back end with very little modification.

- If the database system supports individual user authentication, the back end can be constructed using the database's native authentication and access controls. In this case, the front end would still use a single global account to manage system-wide data, but user-specific data would be stored user that user's account name and protected by that user's password.

- If the database system supports it, stored procedures can be used to implement the back-end processing on the database. The front end is given access to global system data, and the right to invoke stored procedures on the back end. The stored procedures then require that the user's password be provided as part of any transaction that manipulates that user's data.

Issues

It is possible to implement the *Password Propagation* pattern using either interactive user sessions, or individually authenticated transactions. Some differences are:

- Individually Authenticated Transactions. This is the most straightforward case. Users must provide their password as part of any significant transaction. The password is not cached, but is delivered to the back-end as part of the transaction request.

- Interactive Sessions. In order to support sessions, the user's password must be cached, either on the server or on the client. In addition, the back end must offer a login transaction whose sole purpose is to authenticate the user.

User login

- The user attempts to access a protected page on the Web server

- The Web server responds with an authentication challenge (login screen)

- The user provides the authentication credentials

- The application server invokes the login function on the database

- The database login function checks whether the authentication credentials are correct

- If the login is successful, the server caches the authenticated identity and the user's credentials in the session data

User transactions that are not individually authenticated

- The user completes one or more Web forms containing transaction data and submits a transaction request

- The application server retrieves the user's identity/credentials from the session data and submits them to the database with any requests to retrieve or change information or invoke a stored procedure.

- The database validates the user authentication information before delivering or altering that user's account data or invoking a stored procedure on that user's behalf.

User transactions that are individually authenticated

- The user completes one or more Web forms containing transaction data and submits a transaction request. The submission must include the user's authentication credentials (e.g. password)

- The application server retrieves the user's identity from the session data and submits it along with the form data to the database with any requests to retrieve or change information or

invoke a stored procedure.

- The database validates the user authentication information before delivering or altering that user's account data or invoking a stored procedure on that user's behalf.
- The application server does not cache the password presented by the user unless the user has specifically selected an option to "remember my password".

Password changes

- The user completes a Web form containing the old password and the new password.
- The application server retrieves the user id from the session data.
- The application server submits the user identity and the password values submitted in the form to the database.
- The database checks that the old password matches before accepting the new password.
- If it does not match, the application server flushes the session data and logs the user out
- If it does match, the application server caches the new password as the user's authentication credentials

Examples

This pattern often appears when legacy applications are Web-enabled. The legacy transaction server typically already requires that a user's password be provided with each transaction. For example, a Web based interface to an e-mail server. The user logs in. The application server stores the password and attempts to retrieve the user's inbox summary from the database. The database transaction checks the password and returns the information if valid.

This pattern also appears in Web front ends to existing banking systems. Most banking systems already use PINs to authenticate users. But PINs are not strong enough to withstand automated guessing attacks. Therefore, a Web enabled banking application will use a password to authenticate the user to the Web application, but require that the user provide a PIN in order to authenticate the user to the legacy banking system. If the Web application is compromised, the attacker will still need to mount a PIN guessing attack against the back-end system, which will generally be detected or cause the account to be locked out.

Trade-Offs

Accountability	No direct effect.
Availability	No direct effect.

Confidentiality	Confidentiality is improved because a compromised Web server will not have permission to access arbitrary user account data.
Integrity	Integrity is improved because a compromised Web server will not have permission to alter arbitrary user account data.
Manageability	Manageability will be impacted adversely because of the additional complexity. If database access control features are used, they will impose a management burden on the administrators.
Usability	No effect.
Performance	Depending on the implementation, this pattern may exact a small performance penalty.
Cost	Development costs will be increased due to the additional complexity introduced in the interface between the application server and the database. Additional maintenance costs might be incurred by the reliance on proprietary database features (either access controls or stored procedures). If a switch of database systems is necessitated, the costs of making that switch could be very high.

Related Patterns

- *Account Lockout* – a related pattern describing a protection mechanism against password-guessing attacks.
- *Authenticated Session* – a related pattern describing a session mechanism that uses passwords for authentication.

References

None.

Secure Assertion

(a.k.a. Application Logging, Application-Level Tripwire, Sanity Checks, Custom Intrusion Detection)

Abstract

The *Secure Assertion* pattern sprinkles application-specific sanity checks throughout the system. These take the form of *assertions* – a popular technique for checking programmer assumptions about the environment and proper program behavior. A secure assert maps conventional assertions to a system-wide intrusion detection system (IDS). This allows the IDS to detect and correlate application-level problems that often reveal attempts to misuse the system.

Problem

Any server that is accessible from the Internet will be attacked. No matter how much effort you expend in protecting it, the possibility exists that an attacker will penetrate the system. The most difficult attacks to detect are those that exploit vulnerabilities in the application itself. These attacks are generally not visible to intrusion detection systems because they are unique to the application and not widely known attack on standard COTS component.

Application-level attacks cannot be defended at the system level. The firewall, the operating systems, the intrusion detection system, and even the application server may view all such traffic as legitimate application requests. For example, if a banking application fails to adequately validate electronic funds transfer requests, an attacker might use the system to cause a victim's bank account to become overdrawn. To the system, these requests look to be legitimate. In this case, the application provides the attacker with a user-friendly graphical user interface for remotely causing mischief.

An attacker that discovers an application-level problem may go completely undetected by the intrusion detection system and the system administrators monitoring the event logs. As noted above, all of the system-level protection mechanisms will view the attacker's requests as valid user transactions. Unless the application author has taken care to log application-level events, the administrators will have no visibility into the problems within the application itself.

Solution

The *Secure Assertion* pattern transparently re-links all the application-level sanity checks (assert statements) with a mechanism that integrates into the system-wide intrusion detection or event reporting system. Any failed assertions encountered by the application are automatically reported to the system-wide monitoring console as a possible security-relevant event.

In languages that include exception handling mechanisms, the *Secure Assertion* pattern replaces the Exception base class with a custom alternative that reports any generated exceptions to the system-wide event reporting system.

The *Secure Assertion* pattern also offers developers an interface for reporting detected problems that are discovered and recovered from. For example, a function that scans user input and replaces illegal and dangerous characters should report any such replacements via the provided reporting interface.

The *Secure Assertion* pattern transparently reports events that developers already detect and recover from. It requires no security-specific coding to be useful. However, it provides developers who are security-aware a set of interfaces for communicating the state of the application to systems administrators at run-time. The developer best understands what events are extraordinary. This pattern provides a mechanism for sharing that understanding with the systems administrators at run-time.

Issues

The *Secure Assertion* pattern provides developers with a reporting framework that allows system administrators to be aware of potentially security-relevant events occurring within the application. In order for the pattern to provide value, the developers must use these mechanisms.

Each application will have its own specific appropriate checks. Some useful generic approaches are:

- Whenever client-side form validation is employed, double-check the client form validation on the server. If any mismatches occur, this may be evidence that an attacker is tampering with the forms.

- After any complex calculation that involves client input, perform sanity checks on the result. For example, many Web servers compute which local files to serve based on the client's URL. Before serving the page, it can't hurt to make sure that the file in question isn't in a critical system directory.

- Objects within an application should check that function arguments comply with stated restrictions. For example, before transferring funds, it can't hurt to ensure that the amount to be transferred isn't negative.

These checks are relatively inexpensive to perform, pay for themselves during system debugging, and could alert a system administrator before the system is remotely exploited.

Intrusion Detection Systems

Many sites use standard intrusion detection system (IDS) packages to detect attacks on the system. While the use of an IDS is a valuable security practice, it is important to realize that the value of the IDS is greatly enhanced by site-specific customization. Commercial intrusion detection systems use comparison against a library of known attack signatures to detect attacks on COTS products. They are generally incapable of detecting attacks on the Web application itself. Any attempt to misuse the application itself, such as disabling client-side form validation, password guessing, or hidden fields, or performing a resource consumption attack, will go undetected by even the best IDS.

If a commercial IDS is not employed, deliver these assertions via the system's error logging facilities (syslog or NT event logs). These logs should be reviewed routinely. In addition, other security-relevant events, such as failed logins, should be reported via the same mechanism.

The knowledge of the application should be used in tuning any commercial intrusion detection system. For example, many standard intrusion detection systems are capable of alerting on specific URL patterns. If the application does not use cgi-bin scripts or .asp files, it is a good idea to tune the IDS to alert on any requests for these types of resources, since they are obviously not legitimate client requests.

The efficacy of an IDS depends on the fact that it is unknown to the attacker. Don't let the attacker know, through messages or any other means, about the details of the IDS system. If any warning messages are communicated to the attacker, they should be generic in nature.

Integrity Checking

Integrity checks alert the system administrator to the fact that the application has been tampered with. Depending on site policy, they may prevent the Web server from starting up if it appears that the site may have been penetrated.

As with Intrusion Detection Systems, there are commercial integrity checking packages, most notably Tripwire. However, as with IDS, custom integrity checks can greatly increase the efficiency of a commercial package.

For example, an application can be designed to maintain a tripwire-like checksums file for the key application components. Those checksums are stored in some obscure, application-unique location. Whenever the application starts, it checks that the files in question have not been tampered with. When a legitimate system upgrade it installed, the administrator will know to update the checksum file. But any attempts by an attacker to alter the system will result in the administrator being notified.

Monitoring Concerns

The best logging system in the world will serve no useful purpose if it is not actively monitored. (See the *Log for Audit* pattern for advice on auditing issues that will affect the type and amount of log data being collected.)

A little customization can go a long way. Often just a few application-aware checks are necessary. Don't overwhelm the system administrators responsible for monitoring the logs. Overly sensitive sensors that constantly fire have a tendency to inure the reader and could cause legitimate attacks to go unnoticed.

The *Account Lockout* and *Network Address Blacklist* patterns should be used by the system administrator monitoring these events. If the application is clearly under distress, an IP blocking rule should be inserted until the problems can be further investigated.

This pattern does not encompass any form of automated response. It is possible to add such a response, although developing the intelligence to respond appropriately to arbitrary text messages is extremely challenging.

Examples

A number of application servers (including BEA WebLogic, Netscape Application Server, and Apache Tomcat) provide developers with a logging function that will append alerts to a system log. The Extended Log Format defines specific classes of log events, including security events. These logging mechanisms can be integrated directly into Intrusion Detection Systems such as ISS RealSecure. These systems allow logged events to be securely forwarded to a remote monitoring console.

Trade-Offs

Accountability	This pattern might improve accountability indirectly by preventing exploits that would circumvent authentication.
Availability	This pattern could impact availability adversely if local policy dictates a fail-secure approach, in which case detected violations will cause the system to become unavailable.
Confidentiality	See Accountability.
Integrity	This pattern improves integrity by removing opportunities for compromise of the application and helps ensure that exploitation of remaining weaknesses will not go undetected.
Manageability	This pattern will increase the management overhead of the application because it will require that log auditing be performed along with other security tasks. If the various sensors are too finely tuned, a significant administration burden will result.
Usability	No effect.
Performance	Properly implemented secure assertions should not significantly impact performance.
Cost	Assertions require that developers spend additional time and effort inserting checks. However, they usually pay for themselves in reduced debugging effort.

Related Patterns

- *Choose the Right Stuff* – a related pattern that provides more details about the trade-offs

between developing custom components and using standard parts.

- *Client Input Filters* – a related pattern that describes filters to detect or fix problems with client input; these filters are an important class of application Tripwire and must report those events.

- *Network Address Blacklist* – a related pattern that discusses a mechanism for blocking network access to remote systems that have triggered numerous alerts.

References

[1] Cohen, F. and Associates. "The Deception Toolkit Home Page and Mailing List". http://all.net/dtk/dtk.html, 1998.

[2] Ranum, M. "Intrusion Detection and Network Forensics". USENIX '99, Monterey, CA, June 1999.

Server Sandbox

(a.k.a. Privilege Drop, Untrusted Server, Constrained Execution Environment, Unprivileged/Restricted User Account, Run as Nobody, Demilitarized Zone)

Abstract

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

Problem

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

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

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

For instance, Web servers running on the UNIX operating system must be started with administrative privileges in order to listen on port 80 – the standard HTTP port – which is a privileged port. Likewise, the Microsoft IIS default installation executes the Web server using the privileged SYSTEM user. If a Web server running with administrative privileges is compromised, an attacker will have complete access to the entire system. This is widely considered the single greatest threat to Web site security [1].

Solution

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

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

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

- An application can switch the user account under which it is executing at run-time. For example, a UNIX application can switch from running with administrator privileges to a specific server account or even the *nobody* account.

- An application can inform the operating system that it wishes to drop certain privileges dynamically. This is common in capability-based systems, where the operating system dynamically maintains a list of application capabilities. In Linux, an application can ask the operating system to make entire APIs invisible for the remainder of the lifetime of that process.

- An application can instruct the operating system to no longer accept any changes that it requests. For example, once a Linux system has fully booted, it can instruct the operating system to no longer allow kernel modules to be dynamically loaded, even by the administrative account.

- An application can be executed within a virtualized file system. The UNIX chroot option allows the application to think it can see the actual file system, when in fact it only sees a small branch set aside for that application. Any changes to the system files it sees will not affect the actual system files.

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

The *Server Sandbox* pattern partitions the privileges required by the server between those needed at server startup and those needed during normal operation. For example, UNIX systems require administrative privileges to create a server listening on port 80, the standard HTTP port. However, the server should not possess administrative privileges at run-time. A server sandbox allows dangerous privileges to be used to create the server but then revoked before the server is exposed to client input.

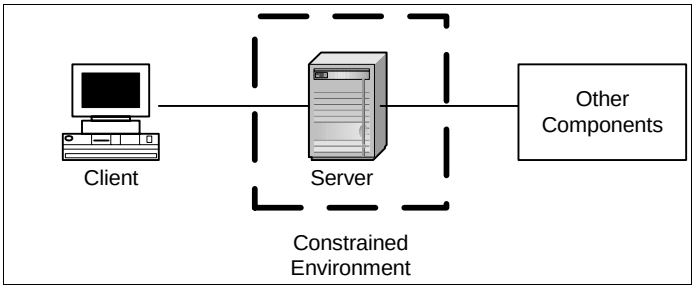
While the most common implementation of the *Server Sandbox* pattern relies on a restricted user account, other (additional) implementations are possible, including:

- Creating a virtual file system and restricting the server so that it cannot see files outside of

this space (chroot).

- Putting wrappers around dangerous components that limit the application's ability to access resources and operating system APIs

- Using operating system network filtering to prevent the server from initiating connections to other machines



Issues

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

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

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

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

Possible Attacks

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

- Buffer overflow attacks – buffer overflow attacks on the server are the most common approach to remote compromise of the server. The sandbox is intended to contain the damage of such an attack.

- Privilege escalation – if an attacker is able to compromise a Web server, even one running as nobody, they will be able to execute code on the system. Attackers typically attempt to break out of the sandbox by exploiting vulnerabilities in other privileged applications, such as sendmail. If a vulnerable, privileged application is accessible to the restricted user account, a privilege escalation attack is possible.

- Breaking out of the sandbox – if the sandbox mechanism contains bugs, an attacker may be able to exploit them to break out of the sandbox. If the attacker can somehow gain root privilege, many sandbox features (such as chroot) are reversible.

- Snooping – if an attacker is able to exploit a server vulnerability and gain a toehold on the system, they may have enough privilege to monitor further server operations. They could capture passwords or other sensitive data. If the server has privileges to access a back-end database, the attacker will have those same privileges.

- Application level exploits – even if the server is perfectly sandboxed, it may still suffer from application-level vulnerabilities. The remote attacker may not have to compromise the server in order to misuse its services.

Examples

At the code level, Java provides the most widely known implementation of a sandbox. It prevents the user from using features and functions that are outside of the Java security policy [2], [3].

At the system level, the canonical example of this pattern is the Apache Web server, which by default runs as user nobody. Although root privileges are required to start the server on port 80, the server drops into the nobody account after initialization.

The nobody account is able to read (but not write) all of the public html files on the server. But a well-configured server will disallow the nobody account from executing any commands or reading any other files.

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

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

Trade-Offs

Accountability	No direct effect.
Availability	No direct effect.
Confidentiality	No direct effect.
Integrity	This pattern will greatly enhance integrity by preventing component vulnerabilities from causing the entire server to be compromised.
Manageability	This pattern will affect the manageability of the software in question because constrained execution environments often incur overhead to setup and maintain.
Usability	No effect.
Performance	This pattern will often have a negative effect on performance, but this will depend on the specific techniques used. Using chroot or unprivileged user accounts do not affect performance. Other techniques that impose additional runtime validity checks will incur a performance penalty.
Cost	This pattern will increase development costs somewhat. This can be minimized if the application is developed with the constraints already in place. Retrofitting an existing application is much more difficult.

Related Patterns

- *Minefield* – a related pattern; any atypical behavior by the restricted user account should be a source of intrusion information.

- *Partitioned Application* – a related pattern; a complex application may require some dangerous privileges throughout its execution time. In that case, the application should be partitioned so that only a minimal component has the dangerous privileges. The other

component(s) should run using restricted user account(s). Components that communicate directly with clients should have the bare minimum privileges. Components with dangerous privileges should be buffered from client requests by other components.

References

[1] Stein, L. and J. Stewart. "The World Wide Web Security FAQ – Version 3.1.2". http://www.w3.org/Security/Faq, February 2002.

[2] Sun Microsystems. "Secure Computing with Java: Now and the Future". http://java.sun.com/marketing/collateral/security.html, 1998.

[3] Venners, B. "Java's security architecture: An overview of the JVM's security model and a look at its built-in safety features". http://www.javaworld.com/javaworld/jw-08-1997/jw-08-hood.html, August 1997.

Trusted Proxy

(a.k.a. Rights Amplifier, Limited View, Restricted Channel,
Integrity Preserving Function, Safe Protocol)

Abstract

A trusted proxy acts on behalf of the user to perform specific actions requiring more privileges than the user possesses. It provides a safe interface by constraining access to the protected resources, limiting the operations that can be performed, or limiting the user's view to a subset of the data.

Problem

It is often necessary to expose components that are not adequately protected to an audience of untrusted users. These components are identifiable by one or more of the following characteristics:

- The component offers protection mechanisms at too great a level of granularity. In order to accomplish necessary tasks, the user must be granted privileges that could be misused. For example, in UNIX, the right to append data to a file brings with it the right to overwrite that file.

- The component is designed for a "safe" environment, and does not offer protection mechanisms. Many products are designed for a LAN environment and cannot be safely exposed to the Internet.

- The component is very complex, and contains many features that could be misused, or many bugs that could be exploited. Many commercial general-purpose operating systems, utilities, and applications fall into this category.

- The component is subject to frequent change, and it is not possible to adequately assess the security of every change. Many major Web sites are continually revised. There simply isn't time to perform a full security assessment of every change.

Exposing these components to anonymous access in an untrusted environment can be dangerous. Many components can be misused to negatively effect accountability, availability, integrity and confidentiality. Worse, some vulnerabilities can result in total system compromise.

Solution

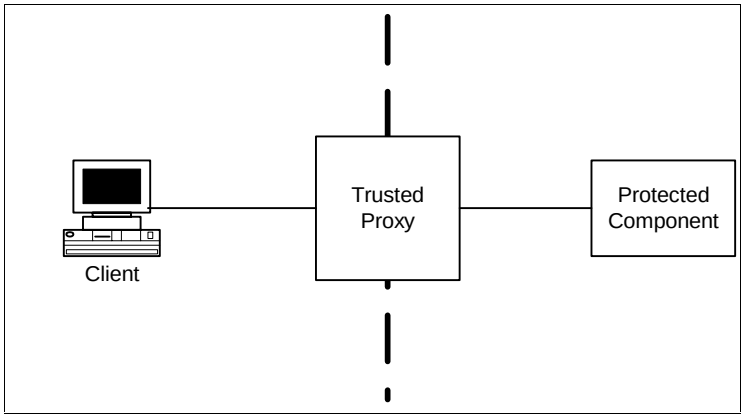
A trusted proxy acts as a buffer between inadequately protected components and an audience of untrusted users. It intercepts and filters all communication between the users and the component(s) in question. By preventing direct access, it can compensate for weaknesses in the protected component(s) and ensure that the appropriate policy is consistently enforced.

Trusted proxies allow custom, finely tuned security policies to be developed and enforced. The *Trusted Proxy* pattern can be used to add protection to components that do not offer any protection mechanisms. Or it can augment the protection mechanisms of existing components to enforce more restrictive policies.

Trusted proxies also allow integrity constraints to be enforced. For example, many systems require transactional models where multiple data files must be kept synchronized. For example, accounting systems require integrity constraints on the contents of multiple files. Inventory systems allow changes to the inventory, but record the identity of the individual making the change. And database systems require that any change to the database obey the integrity rules defined for that database.

When building a trusted proxy to protect an existing component, there are two basic approaches to filtering the data. You can either search the data for known bad characteristics, or you can create the new request to be delivered to the target from scratch and only copy certain parts from the original client-provided request. Scanning the original data is generally faster and allows new features to be added without breaking the proxy. But rebuilding the data is generally more likely to defuse new attacks. For example, when new vulnerabilities in the TCP/IP stack are found, proxies that simply send the packets provide little defense. Proxies that rebuild every packet usually prevent the malformed packets from being delivered to the target.

Proxies generally do not keep up with the latest and greatest features in the components that they are protecting. Whether the proxies are developed in house or commercially available, the developers of the proxy need additional time to update and perform quality assurance on the proxy. As a result, the proxy can be seen as an impediment to achieving other non-security goals. In an environment where such changes are likely to be required, a scanning approach to filtering is generally preferred.



Issues

Types of Proxies

Trusted proxies can be either transparent or opaque. A transparent proxy is one that requires no modification to the original system. It can be inserted between existing components without breaking compatibility. Often, the components in question will not even be aware of the

existence of the proxy. Transparent proxies are useful for filtering out known attacks, or providing a limiting view of the protected resources. Many network-level filters act as transparent proxies, allowing benign traffic to flow through undisturbed, but intercepting traffic that is known to be dangerous.

An opaque proxy is one that offers a different interface from that offered by the protected resources. An opaque proxy requires that clients program directly to the proxy – if inserted between existing components, it will generally require modification to the client. Opaque proxies sometimes augment the existing interfaces with additional security features, such as authentication. Opaque proxies often define higher-level functions and then map these to the low-level functions offered by the protected resources.

For the sake of efficiency, it may be possible to have the target respond directly to the client instead of proxying the response as well as the request. When the proxy is used to enforce a conventional access control model, the security generally is applied only to the request and it is safe to have the response be unproxied.

If possible, proxies should be designed to be layered. A composite solution of simple, layered proxies is generally easier to maintain and secure than a single blob that enforces multiple policies and goals. Many proxies are developed for non-security purposes. In particular, load-balancing and fail-over mechanisms often require the ability to redirect requests from the original target to a stand-in. Make sure to test these facilities to understand how they interact with the proposed security proxies.

Defending the Proxy

Realize that the proxy itself will be the target of any attacks launched at the original. If the proxy is more vulnerable than the original, the result is a net minus. In particular, never use variable-sized buffers of unbounded string manipulation functions, as these are often compromised using buffer-overflow attacks. If possible, use an existing product or technology instead of developing one from scratch. Of course, any mechanism that defends an unprotected component is by definition stronger than the original.

Proxies should be developed with Implementation Hiding in mind. For example, when malformed or invalid data is encountered, the proxy should not provide a possible attacker with specific information that could aid in identifying or circumventing the proxy. The error should be logged, so that administrators can debug the problem, but the client should only be given a generic failure response.

A trusted proxy should filter all input from untrusted clients, as the components being protected may not perform such filtering. See the *Client Input Filters* pattern for specific strategies.

Designing Around the Proxy

The trusted proxy must be non-circumventable. Attackers generally prefer to attack the weakest part of a system. Often, that means looking for a way to circumvent the proxy rather than attacking the proxy itself. Don't fall into the trap of imagining that the attacker will focus all his/her efforts against the proxy. (See the *Red Team the Design* pattern.)

Do not give the proxy any privileges that it does not absolutely require. A vulnerable proxy running with administrative privileges may be a bigger security flaw than the components it was intended to defend. If at all possible, use the operating system or language access control mechanisms to restrict the privileges of the proxy to the minimum possible.

A proxy cannot fully eliminate the security implications on the systems that it defends. If an attacker figures out a way to misuse legitimate services, the proxy may simply forward the seemingly legitimate request through to the target. (See the *Share Responsibility for Security* pattern.)

Provide logging interfaces that allow invalid attempts to be monitored and stored. Many attackers will systematically probe the components in a system. A system administrator who is alerted to attacks on the proxy may be better able to distinguish attacks on other components. (See the *Log for Audit* and *Network Address Blacklist* patterns.)

Examples

At the code level, any object that contains private data is a form of trusted proxy. It ensures that the integrity constraints within the object are enforced by disallowing direct access to that data. It exports integrity preserving functions so that users who are not trusted to directly manipulate the data can be allowed to access the data via certain controlled interfaces. However, most programming languages (including C++), cannot guarantee that other components will not bypass these protections. Java is a notable exception: in most cases, objects will not be able to circumvent the language's protection mechanisms.

J2EE Enterprise JavaBeans (EJBs) can provide access to a database with only the specific limited functionality required by the application.

At the system level, the UNIX Mail delivery program has the privilege to write into any user's files. The mail delivery program is trusted to only append incoming mail to the appropriate mailbox file, and not touch any other files.

All database programs allow users to modify files in a constrained manner but generally disallow direct access to those files. Oracle, for example, does not give users permissions to open database files directly. They can invoke Oracle to open the file on their behalf, subject to Oracle's access control rules. Many databases also allow restricted views of tables to be established. A restricted view permits each user to see only that subset of a table that he or she has privilege to.

At the network level, a firewall is a classic trusted proxy. It provides a constrained interface to systems that are not trusted to be directly accessible from the Internet. Many firewalls also proxy outgoing connections, in order to prevent untrusted users from bringing dangerous materials into the protected network.

A Web/ftp server that accesses local files and serves them to remote users who have no privileges on the local system(s) in another example of a trusted proxy.

Many Web applications provide users with restricted access to a private database. Users are only allowed to see their own account data (and often only a subset of that). For example, Amazon.com will let you view your order history, but it will not show you the profile data that it has compiled about your activities at that site.

Trade-Offs

Accountability	Trusted proxies are often used to enhance accountability by enforcing authenticated access to protected resources and adding logging rules to those resources.
Availability	A trusted proxy adds another possible point of failure. If either the proxy or the proxy's target fail, the system will be unavailable. A proxy that is significantly less reliable than the protected resource, or vulnerable to overloading attacks, will drastically reduce the overall availability.
Confidentiality	This pattern can be used to protect important data or subsystems.
Integrity	Trusted proxies have a huge impact on integrity: they prevent misuse of important subsystems.
Manageability	Configuration of the trusted proxy can be extremely difficult, especially if the individual configuring the trusted proxy does not fully understand the types of data and the consequences of data flowing through the trusted proxy.
Usability	Trusted proxies generally do not affect usability, particularly if they are transparent to the user. However, firewalls are an example where a trusted proxy has been observed to reduce usability dramatically. Many users find that their on-line activities are thwarted by an overly restrictive firewall policy. One unfortunate result is that end users often attempt to circumvent firewalls when the firewall prevents them from conducting their desired activities.
Performance	A trusted proxy represents an additional level of indirection between the user and the protected resource and as such has a negative impact on performance. This impact can be quite severe and should be investigated using prototypes early in the development cycle.
Cost	A trusted proxy is a textbook example of information hiding, in the classic software engineering sense. A well-designed trusted proxy can greatly reduce inter-object coupling. However, if the object interfaces are not chosen well, the result can be a maintenance headache. The traditional object-oriented approach to maintenance (re-factoring of

	the object design) can run into problems if organizational security policies are defined in terms of those object interfaces.

Related Patterns

- *Red Team the Design* – a related pattern.
- *Share Responsibility for Security* – a related pattern.
- *Log for Audit* – a related pattern.
- *Network Address Blacklist* – a related pattern.

References

[1] Balestracci, S. "PC Week Hack of 1999". http://www.sans.org/infosecFAQ/threats/PC_week.htm, February 2001.

[2] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] Zwicky, E., S. Cooper, and D. Chapman. *Building Internet Firewalls 2nd Edition*. O'Reilly & Associates, 2000.

Validated Transaction

(Mini-Pattern)

Abstract

The *Validated Transaction* pattern puts all of the security-relevant validation for a specific transaction into one page request. A developer can create any number of supporting pages without having to worry about attackers using them to circumvent security. And users can navigate freely among the pages, filling in different sections in whatever order they choose. The transaction itself will ensure the integrity of all information submitted.

Problem

Web applications often have to collect a great deal of data from a user in order to complete a single transaction. E-commerce purchases, for example, require that a user to select items for purchase, provide contact information and a shipping address, choose shipping options, and submit credit card information. Rather than simply present the user with a huge form with a bewildering array of options, most sites prefer to guide the user through the process, validating each piece of data as it is provided.

This approach can be vulnerable to attacks. An attacker can use known URLs to jump between the different pages, in attempt to bypass some of the data validation checks. For example, if the application developer is not extremely careful, it may be possible for the attacker to add items to the order after having paid.

Solution

The *Validated Transaction* pattern solves this problem by establishing a single point at which the transaction is committed. At that point, any data validation checks are duplicated in order to ensure that data tampering will be discovered. Furthermore, data consistency checks are also implemented at that point.

This pattern can be used to retrofit an application that was not designed with security in mind. Alternately, it allows developers of the application to make changes to the application, safe in the knowledge that the *Validated Transaction* pattern offers a safety net against security problems.

Related Patterns

- *Client Input Filters* – a related pattern that can use this validated transaction mechanism to enforce that client input is validated.

- *Directed Session* – a complementary pattern that exposes a single URL to the user and enforces form validation by storing the actual URL in session data.

References

None.

C. Procedural Patterns

The following procedural patterns are presented in this section:

- Build the Server from the Ground Up
- Choose the Right Stuff
- Document the Security Goals
- Document the Server Configuration
- Enroll by Validating Out of Band
- Enroll using Third-party Validation
- Enroll with a Pre-Existing Shared Secret
- Enroll without Validating
- Log for Audit
- Patch Proactively
- Red Team the Design
- Share Responsibility for Security
- Test on a Staging Server

Build the Server from the Ground Up

(a.k.a. Know Thyself and Configure Appropriately, Keep It Simple Stupid)

Abstract

Many Web compromises and defacements occur because of unnecessary and potentially vulnerable services present on the Web server. Default installations of operating systems and applications are the source of many of these services. This pattern advocates building the server from the ground up: understanding the default installation of the operating system and applications, simplifying the configuration as much as possible, removing any unnecessary services, and investigating the vulnerable services that are a part of the Web server configuration.

Problem

Web servers, their underlying operating systems, and application server environments are extremely complex. Most developers (and even many administrators) generally do not understand this complexity—they depend on pre-existing installation programs to configure the system correctly. When things don't work, developers and administrators tweak and patch until the problems appear to be resolved.

While this approach might be appropriate for administration of desktop systems, it is not at all suitable for Internet-accessible servers. Operating system installations tend to include large numbers of unnecessary and dangerous tools and utilities. Server installations contain a vast number of extraneous sample files, tools, and a wealth of different programming options, all enabled. Powerful development tools left on a Web server provide a sophisticated attacker with tools that can be turned against the server and even the organization. Installations of Web tools often include poorly designed sample scripts that reveal information about the server and might be vulnerable to attack.

These extraneous and unnecessary services and tools are the source of many Web site security breaches. There are many examples of security vulnerabilities resulting from insecure default installations:

- The Microsoft IIS Web server installs with a large number of programming options and APIs enabled. Most developers use a tiny fraction of these. However, hackers and worm authors have taken advantage of the default configuration, finding innovative ways to exploit these generally unused capabilities. Recent examples include Nimda and Code Red.

- Many application environments and operating systems have provided guest user accounts and/or default administrative accounts with well-known passwords. Hackers often exploit these accounts during intrusions. For example, Oracle creates a “system” account with the password “manager” and a “dba” account with the password “change_on_install.” Microsoft SQL Server creates an administrator account named “sa” with a blank password.

- Default file permissions for many operating systems are geared to general purpose computing

environments in which users are expected to collaborate. User accounts are given a great deal of access to system data and other users' data. This is not appropriate for a server, where and attacker may compromise a single user account and thereby gain access to a great deal of additional data.

Furthermore, due to the complexity of these configurations, system administrators often do not understand enough about the workings of their system to even realize when an attacker has penetrated and installed backdoor software.

Solution

Instead of relying on default installation programs, the system administrator should attempt to build the server by copying individual files and packages. This will ensure that only the absolutely necessary components are installed on the server. It will also ensure that the administrator knows the function of every component installed on the server and will be able to detect tampering with the server.

When a component cannot be installed without an installation program, the administrator should always choose a custom installation option when it is available. The administrator should install only the bare minimum functionality. If the bare minimum installation does not offer the needed services, the administrator should install additional components one at a time, until the absolute minimum installation is uncovered.

When default installation programs absolutely cannot be avoided, the system administrator should perform installation experiments using two machines. The installer should be run on the first machine, and files manually copied to the second. Files that look extraneous should initially be omitted and only added if the system requires them in order to run. On systems such as Windows NT, which use a global registry, the administrator should track changes to the registry and manually copy the entries to the second machine.

In some circumstances, copy protection schemes may require that an installation program execute on the actual system that will be used in production. In these cases, all changes made by the installer should be tracked. The system administrator can then experiment with the removal of extraneous files and configuration settings.

Under no circumstances should standard examples be installed on a production system. Under no circumstances should development tools or system source code be installed on a production system.

Issues

There is an obvious trade-off between using a default server configuration and configuring a server oneself. The trade-off involves the level of security provided by the default configuration versus the level of security resulting from configuring complex software from scratch. Some systems provide very reasonable and secure default configurations, and the level of effort required to derive a more secure custom configuration might be very high. Other systems,

however, have well-documented insecure default configurations, and building those servers from the ground up is absolutely necessary.

When vendor installation programs and scripts must be used, it is important to understand the complete impact of the installation process. Unfortunately, vendor installation programs usually alter many parts of the system (user accounts, startup scripts, system registry, device drivers, etc.) and give little indication of what was done or why. The use of these scripts is often unavoidable, but system developers should take the time to assess the end results. For example, on a UNIX system you can use "ls –alr" or other file system checking tools to determine the files changed by an automated installation process. On Microsoft systems, before and after snapshots of the registry can be used to track changes.

When installation scripts are unavoidable, general hardening procedures should be followed. There are a number of guides available providing both general guidance [1] and specific details about locking down particular systems [3].

Programs on UNIX operating systems that run with SUID privileges especially should be left off Web servers except when absolutely necessary. Because these programs have superuser privileges, when they are misused or compromised the consequences are especially disastrous [2]. Many SUID programs can be subverted to perform different actions than originally intended; their presence on Web servers should be minimized as much as possible. Similarly, SUID shell scripts should *never* be left on Web servers [2].

Examples

System Level

The Red Hat Linux installation program represents a good example of the difference between an automatic installation and a manual installation. It offers a default "server installation" which proceeds to install over a gigabyte of services. However it is possible to choose packages individually. Using this model, one can select only those Redhat Packages (RPMs) that are absolutely necessary. This produces a much simpler configuration.

The Red Hat installation program also includes the ability to select (and even customize) a firewall rule set. It provides a graphical tool that allows the administrator to select which ports on the system should be left open and which protocols should be allowed. This is a very useful tool in that it automates much of the security configuration process, but offers enough flexibility to be useful for non-standard systems.

It is also possible to build a Linux server entirely from scratch. The Linux from Scratch Project (www.linuxfromscratch.org) gives step by step instructions for building a Linux system without installation scripts. Part of the rationale is:

Another advantage of a custom built Linux system is added security. You will compile the entire system from source, thus allowing you to audit everything, if you wish to do so, and apply all the security patches you want or need to apply. You don't have to wait for somebody else to provide a new binary package that fixes a security hole. Besides, you have no guarantee that the new

package actually fixes the problem (adequately). You never truly know whether a security hole is fixed or not unless you do it yourself.

Trade-Offs

Accountability	No direct effect.
Availability	This pattern facilitates restoration or reconstruction of a Web site in the event of site compromise, minimizing downtime.
Confidentiality	No direct effect.
Integrity	This pattern enhances integrity by helping prevent site compromise.
Manageability	This pattern significantly enhances manageability by reducing the complexity of the servers, and providing the administrator with a better understanding of the remaining complexity.
Usability	No direct effect.
Performance	This pattern may improve performance if it results in the elimination of extraneous services that would otherwise consume resources.
Cost	This pattern will incur significant additional cost during system development and deployment.

Related Patterns

- *Document the Server Configuration* – a related procedure for documenting the configuration of Web and application servers, enabling reproduction of the server configuration(s).

- *Test on a Staging Server* – a related procedure for development and testing of Web applications that explores the impact of changes on a separate installation before committing them to the final system.

- *Patch Proactively* – a related procedure for administering servers.

References

[1] Allen, J. *The CERT Guide to System and Network Security Practices*. Addison-Wesley, 2001.

[2] Garfinkel, S. and G. Spafford. *Practical UNIX & Internet Security.* O'Reilly & Associates, 1996.

[3] Naval Information Systems Security Office. *Secure Windows NT Installation and Configuration Guide*. Windows NT for Navy IT-21 Version 1.5, December 2000.

Choose the Right Stuff

(a.k.a. Select Components for Security, Using Standard or Custom Parts)

Abstract

Many security problems can be avoided during system design if components, languages, and tools are selected with security in mind. This is not to say that security is the only criterion of concern – merely that it should not be ignored while making these decisions. This pattern provides guidance in selecting appropriate Commercial-Off-the-Shelf components and in deciding whether to use build custom components.

Problem

Modern systems are built using a collection of products, tools, language libraries, and custom code. These components can all impact system security. Some components lack the functions needed to support a secure application. Others have the necessary features but are crippled by recurring bugs. Some products are completely proprietary, ensuring that both defenders and attackers don't fully understand the product, and are dependent on the vendor to address security issues that come up. And custom components, while appealing, often suffer from the same problems that plague commercial components.

If the major functional components of a system are selected without a consideration of security issues, it is highly likely that the system will suffer from significant security problems. The primary criteria in selecting tools and languages will generally not be security. However, it is a mistake to ignore security completely. Even the best products have known weaknesses that developers must work around. Many components that are perfectly adequate for LAN usage are inappropriate for an untrusted environment such as the Internet. And there are other components and tools with such miserable track records for security that they should simply be avoided.

Solution

During system design, carefully evaluate the security all candidate components, languages, and tools. In a fully rational process, this would involve choosing multiple candidate products and performing a rigorous security and functionality analysis of each. In a more pragmatic process, the collection of appropriate components may be predetermined by other factors. Whatever the case, investigate the security implications of each component before development starts.

When investigating Commercial-Off-the-Shelf (COTS) products, it is often the case that no one product meets all the functional and non-functional requirements of the system. It is appealing to think that a custom-built solution can meet all of the needs. However, before developing a custom component, make absolutely sure that an alternative is not available. In comparing alternatives, do not assume that the custom solution will be a panacea—it is entirely likely that the custom approach will have as many security flaws as any commercial alternative.

Issues

The software engineering literature provides a great deal of guidance in selecting components, languages, and tools that minimize development risk and meet the needs of the project. This pattern provides some additional security-specific guidance that should supplement other understanding.

Past History of Vulnerabilities

When choosing any COTS component, the most significant security question to investigate is how frequently major security vulnerabilities have been discovered in the product. Web sites such as securityfocus.com provide historical information about security advisories, organized by product. If a product has a track record of major vulnerabilities, it is safe to assume that record will continue.

When investigating past vulnerabilities, it is instructive to examine the vendor's response. Did the vendor act proactively, issuing a response at the same time the vulnerability was released, or did they have to be prodded into action by the security community? How quickly did they respond? Did they issue patches, or did they simply offer workarounds? Were major elements of functionality disabled as part of any of these workarounds?

It is also important to consider any application in context. A highly popular application is going to be attacked more often than an unpopular one. And a feature-rich, complex application is going to have more security problems than a much simpler alternative. If you do not need the additional features, then the simpler alternative is usually a better choice. And if the more obscure application is less vulnerable on account of its lack of popularity, that works in its favor.

When looking into past vulnerabilities, pay close attention to the workarounds suggested by vendors. On many occasions, a vendor has simply required that a feature be disabled if the system is to be secure. Understand which features may have to be disabled, and do not build an application to rely on those features.

Security Features

Some vendors pay greater attention to security than others. The simplest discriminator is to look at the evaluated products list maintained by the National Institute of Standards and Technology (NIST) and the National Security Agency (NSA). If a product has been successfully evaluated, it means that the vendor has gone through a rigorous process of adding security features, documenting those features, and convincing an independent laboratory that the product behaves as described.

Unfortunately, too few products have been evaluated, and many that have are one or two versions behind the vendor's currently shipping products. In these cases, there are other things to look for that will help assess the vendor's commitment to security.

- Read the product documentation. How many security features are described? How well are they explained? Does the documentation include guidance on securely configuring the

product?

- There are many mailing lists, Web sites, and product support forums that discuss security. Search the archives for information about security features for the product in question

- When looking at tools and languages, how rich are the security configuration options? How extensive are the logging capabilities? Do they allow you to define security policies that meet your needs?

- Has anyone published any experiences describing a system largely similar to the one you envision?

Safe Programming Environments

The wrong language or tool could be a huge detriment to the security of the system. Some languages and tools are better suited for some tasks than others, and some are easier to program securely than others. For instance, C is very efficient, system-level language that is good for programming many lower-level applications. However, if C is used incorrectly without understanding the security implications, it is easy to produce code with latent vulnerabilities. Java, on the other hand, is not as efficient, but it is in many ways easier to understand and provides a much more robust security model.

In general, you want to use the safest environment that offers adequate performance, scalability, and reliability. Perform some rudimentary prototypes using a safe language such as Java in order to assess the performance. Recognize that Java will never be as fast as C, but it may well be fast enough. It may be cheaper to use several servers than to recover from a major security breach.

If you must use a language that stresses performance over security, there are many references that provide low-level details about writing code that is less likely to have major security vulnerabilities [7]. Every developer should be familiar with this material.

Other Considerations

Some other security-relevant considerations are:

- Is the programming model an industry standard or will the system be locked into a single vendor solution? If the vendor's commitment to security wanes, will it be possible to replace the component with a more secure alternative?

- What are your developers familiar with? Developers are less likely to make mistakes with languages and tools that they are familiar with. Forcing them into a new programming environment will involve a learning curve that could introduce security problems.

- Is source code available to the system builders? To potential attackers? There is open debate about whether open source is less secure or more secure than closed source COTS components. But nobody contests the fact that open source systems are less dependent on a single vendor's response to a security vulnerability.

Examples

At the code level, the C language is very fast, but requires skilled developers who understand the security implications of their programs. It is very easy to make a mistake that will allow an attacker to gain the same level of privilege as the application. Here are just a few of the major mistakes commonly made with C: incorrect usage of string functions (e.g., using “strcpy” when “strncpy” should be used), poor signal handling, incorrect pointer arithmetic, and incorrect function usage.

A number of different sources recommend the use of Java for Web applications. While Java is not as fast as C, it provides significantly more security out of the box. When an error occurs there is less chance for an attacker to gain any privileges because Java is a type-safe language, and runs within a constrained security model that protects the rest of the system. Many common mistakes in C, such as those leading to buffer overflow vulnerabilities, cannot be made in Java. However, it is important to understand that logical errors can still be made in Java. For instance, if the Web application does not properly check client-supplied data, even a Java program can be tricked into behaving inappropriately.

At the system level, the choice of server operating system is extremely important. In general, servers should not use the very latest desktop operating system. Because these systems are constantly updated to include support for the latest and greatest hardware, the bugs have generally not been worked out. A recent example is the FBI’s warning about the use of Universal Plug and Play on Windows XP. This is a feature that has been added to Windows in order to support hardware that is not yet available. The feature is enabled by default and allows remote compromise of any system on which it is running.

A security consultant friend may have put it best when he explained, “I always put the servers on UNIX because people are scared of the UNIX box and will leave it alone. If I put them on Windows, somebody always installs Word on the box.”

At the network level, many banks and financial institutions have opted to use the Hewlett-Packard Virtual Vault as their Web server. The Virtual Vault has a reputation of being the world’s most secure Web server [5]. The Virtual Vault is built atop a military-grade multi-level security system. While management of the server is considerably more difficult than some other platforms, the vendor the vendor has established service level agreements for providing fixes to identified vulnerabilities. Furthermore, because the product uses non-standard hardware and the vendor tracks all sales of the software, hackers generally do not have access to a system on which to experiment. Finally, HP has offered a one million dollar reward to anyone who manages to compromise a properly configured Virtual Vault.

Trade-Offs

Accountability	No direct effect.
Availability	By helping minimize security flaws, this pattern will help minimize downtime.

Confidentiality	No direct effect.
Integrity	This pattern helps ensure the integrity of the system.
Manageability	This pattern will help identify and avoid potential manageability problems that might otherwise have been undiscovered.
Usability	No direct effect.
Performance	Choosing the most secure products will generally have an adverse effect on performance.
Cost	This pattern will have an impact on cost. Researching product capabilities will introduce delays into the development cycle. And sometimes the products selected will have higher costs than those that might otherwise have been chosen.

Related Patterns

- *Document Security Goals* – a related pattern that establishes the security policy, which guides and informs the process of choosing the right stuff.

- *Patch Proactively* – a related pattern that is applicable when standard or COTS components are selected; this pattern establishes the process of checking regularly and often for vulnerabilities and patches applicable to standard parts.

References

[1] Desal, G., J. Fenner, J. Patel, and M. Schenecker. "Web Application Servers are Here To Stay". http://www.informationweek.com/726/app.htm, March 1999.

[2] Dyck, T. "Four scripting languages speed development". http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2646052,00.html, November 2000.

[3] Kamath, M. "Choosing a scripting language for ASP". http://www.kamath.com/columns/my3cents/mtc002_scripting.asp, October 1999.

[4] Pountain, D. and J. Montgomery. "Web Components: Components and the Web are a match made in developer heaven". http://www.byte.com/art/9708/sec5/art1.htm, August 1997.

[5] Stein, L. and J. Stewart. "The World Wide Web Security FAQ – Version 3.1.2". http://www.w3.org/Security/Faq, February 2002.

[6] Strom, D. "More on ActiveX Versus Java Security: Are you safe?" http://www.webdeveloper.com/security/security_java_activex.html, 1999

[7] Viega, J. and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.

Document the Security Goals

Abstract

In order for developers to make consistent, intelligent development choices regarding security, they have to understand the overall system goals and the business case behind them. If the security goals are not documented and disseminated, individual interpretation could lead to inconsistent policies and inappropriate mechanisms.

Problem

There are a variety of different processes used for software development. An important early activity in most development processes involves requirements gathering and definition. Generally, the core functional requirements are defined explicitly early in the development effort. Non-functional requirements, including those regarding security, are often left vague, or not considered at all. There are a variety of problems that can arise when security goals are not documented and disseminated properly.

When security requirements are not documented and/or disseminated to the development team, developers of individual parts have little chance of making consistent security decisions across the system and throughout the system lifecycle. When security requirements are left vague, they might target the wrong threats or even build measures of inappropriate strength. In either case, the system's overall security suffers because the system is only as strong as its weakest link [8]. Overly weak components will compromise system security. Overly strong components will not enhance security because attackers will not target these defenses; at the same time, these overly strong components might penalize other requirements (such as usability or performance) and are certainly not cost effective.

Perhaps the worst possible circumstance is that developers might not address security at all throughout the development process. It is often stated that security must be built into a system from the start; it cannot be added effectively, or cost effectively, after the fact [4]. If developers are not aware of security goals and then ignore security, the process of retrofitting security mechanisms into an already developed system will most likely be expensive, painful, and ultimately unsuccessful.

Without a clear understanding of the security goals of the organization and application, it is impossible for developers to build the right thing. If too little, too much, or even the wrong type of security is built into the system, the result can be an expensive rewrite.

Solution

Document the security goals early in the project, as part of requirements gathering and definition.

Regardless of the particular development process employed on a project, documentation of security requirements should be addressed early on. While it is often not possible to document all the security goals in the beginning stages of development, an attempt to document the greatest

risks to security is important. In a spiral development model, each iteration of development should identify and address the most significant remaining security risks [1].

Ideally, documentation of the security goals will not result in an overly large document. This documentation must be readable by its target audience: the developers of the system.

At a high level, a security requirements document should do the following:

- Identify classes of users and the actions each class is permitted to perform
- Specify the relative importance of confidentiality, integrity, availability, and management costs for major elements of functionality.
- State any legal or regulatory constraints (such as accessibility, privacy, and cryptographic export restrictions).
- Include a risk analysis that documents what sort of losses are tolerable and what sort are not.

Once the security goals are documented, they must be disseminated to the entire development team. Developers cannot make localized trade-off decisions that reflect the relative goals of the project unless they understand what those goals are. It is crucial that everyone—not just "security people"—understand these goals. This will ensure that conflicts between security and other goals are identified and resolved early in the development lifecycle.

As the project progresses, the security goals (and all other functional and non-functional requirements) will be refined as details emerge and conflicts are ironed out. The documentation of the security goals should be maintained as these inevitable changes occur.

Issues

Documentation of security goals involves understanding security in the context of the entire system.

- In developing the security goals, it is instructive to identify different classes of stakeholder (e.g. end users, administrators, etc.) Each group of stakeholders should have input into the process. This input can clarify overall objectives and assist in trade-off analysis between security and other important goals.
- Avoid the temptation to require inappropriate degrees of security. The goals document should be supported by rationale that demonstrates why security measures in excess of best industry practice are necessary, and cost-effective.
- Explicitly describe the operational requirements (and associated costs) that security measures will require. For example, using the *Account Lockout* pattern to protect against password guessing might require an operator to be available to reset locked accounts.
- Consider your security policy and procedures when picking your goals. A secure account

management system does little good if the policies and procedures behind its use are never enforced.

Determining security goals requires an understanding of business issues, including legal and regulatory issues.

- User-provided data may have some liability associated with its misuse. For instance, credit card numbers need to be protected according to the credit card company's requirements and automated use of bank account information is subject to NACHA regulations [3]. Government sites are restricted in their use of social security numbers. And sites are forbidden to collect most information about minors. The security goals document should reflect these requirements.

- If your site offers services, the services could be misused in a manner that introduces liability. File storage could result in trafficking of illegal software or child pornography. Public forms could result in libel charges. Anonymous e-mail accounts could be used for illegal activities. Understand the relevant legal and regulatory constraints and document them.

- Understand that you can be subpoenaed for anything you log. The security goals document should explain to developers the hidden costs are associated with any additional data that they log.

- Recognize that often the best defense against liability claims is adherence to best practice. Be aware of standard industry practices, and cite them whenever possible in the security goals document.

- While it is important not to goldplate the security of an Internet portal, it is important to recognize that Internet access makes this interface far more vulnerable and subject to attack then other, weaker, interfaces. It is true that even the strongest Web site can be undermined by a dumpster diving attack. But the Web site can be attacked from halfway around the world, while the dumpster is protected from all but the most determined attackers by physical distance.

Examples

There are many software processes designed to document system requirements, including security goals. Specific examples of their usage are sometimes difficult to cite because these experiences are not often documented.

One published example of a security goals document exists for the Grex, a public-access computer conferencing system [9]. Rationale for the various policy decisions is explained in this document.

Another example of a high-level security goals document can be found for K-Meleon, a derivative of the Mozilla Web browser [2]. This document is very short and really represents only a first attempt at a security goals document. More details must be incorporated in order for

users of the document to be able to assess the trade-offs between specific security goals and other requirements.

Examples of even more high-level security goals can be found for the San Diego Supercomputer Center [5], a scalable and secure e-commerce hub [7], and Open Financial Exchange security [6]. Again, the few slides in each of these presentations regarding security policy must be expanded into a full document with more specific requirements.

Trade-Offs

Accountability	This pattern will help ensure that appropriate trade-offs are made between accountability goals and other requirements.
Availability	See Accountability.
Confidentiality	See Accountability.
Integrity	See Accountability.
Manageability	This pattern will often enhance manageability, because appropriate security measures designed in from the start are usually easier to manage than solutions added at a later date.
Usability	See Accountability.
Performance	See Accountability.
Cost	Usage of this pattern will probably increase cost in the short run, but should reduce overall cost in the long run. By designing the system with security goals in mind, developers are betting that the cost now is significantly less than the cost of security problems that could be experienced later.

Related Patterns

- *Share Responsibility for Security* – a related pattern that distributes security concerns amongst the entire development team; the security goals documented in this pattern must be communicated to all team members.

References

[1] Abrams, M. “Security Engineering in an Evolutionary Acquisition Environment”. New Security Paradigms Workshop 1998, Charlottesville, VA, September 1998.

[2] Mutch, A. "Mozilla Security Goals", http://tln.lib.mi.us/~amutch/pro/mozilla/secgoals.htm, April 2002.

[3] NACHA Internet Council. *Understanding Internet-Initiated ACH Debits*. http://internetcouncil.nacha.org, 2002.

[4] Open Web Application Security Project (OWASP). "A Guide to Building Secure Web Applications and Web Services – Draft 0.2". http://www.owasp.org, May 2002.

[5] Perrine, T. "NPACI/SDSC Security Activities", http://www.edcenter.sdsu.edu/training/workshop99/june29_ppt/tep1999/sld008.htm, July 1999.

[6] Rodriguez, C. "Electronic Bill Presentment and Payment (EBPP) and Open Financial Exchange (OFX) Security", http://cs1.cs.nyu.edu/rodr7076/ebpp/sld013.htm, December 1998.

[7] Ryan, S. "A Scalable and Secure E-Commerce Hub for Electronics Recycling", http://www.ses.imse.iastate.edu/Presentation.htm, 2000.

[8] Viega, J. and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.

[9] Wolter, J. "Grex Staff Notes: Security Goals". http://www.cyberspace.org/staffnote/goals.html, March 1998.

Document the Server Configuration

Abstract

Web servers and application servers are extremely complex, and complexity is a major impediment to security. In order to help manage the complexity of Web server and application configurations, developers and administrators must document the initial configuration and all modifications to Web servers and applications.

Problem

Web servers, their underlying operating systems, and Web applications have complicated and dynamic configurations.

The complexity arises from the multitude of options available within the general-purpose operating systems on which Web and application servers run. The wide range of functional and non-functional requirements that Web applications must satisfy also complicates the configuration. In addition, the many possible interactions, compatibility concerns, and interoperability issues between various operating systems and applications complicate configuration control.

The dynamism arises from user feedback, evolving requirements, changing threats and vulnerabilities, and updates in underlying system software. Web applications must continuously solicit user feedback and respond to usability issues. This input, in addition to normal, planned enhancements to functionality, contributes to the evolution of system and application requirements. All these changes can impact the server configuration. In addition, the threats to Web servers and associated vulnerabilities are always changing, and this requires frequent updates to server configurations.

Without a process for managing the complexity and dynamism of server configurations, developers and administrators can quickly lose intellectual control and cease to understand the systems they must maintain. Perhaps most importantly, in the event of catastrophic failure or security breach, it might be impossible to reconstruct the precise configuration of Web and application servers because administrators cannot remember the exact series of changes and upgrades over time.

Solution

Developers and administrators should create and maintain a Server Configuration Document with the following characteristics:

- It provides step-by-step instructions for recreating the server from scratch.
- It includes only the minimum functionality needed to execute the application.
- It is subjected to rigorous, independent quality assurance on staging servers before being

rolled out on production servers.

- It explains the *why* behind the server configuration.
- It is maintained under a version control system.

When creating a Server Configuration Document, developers and administrators must first understand the configuration options selected for deployed systems, whether using a default configuration or custom settings. This understanding of the configuration should be captured in the Server Configuration Document, so that later maintainers of the Web and application servers can understand the rationale behind configuration decisions.

As the Web and application servers are maintained and their configurations evolve, all changes should be recorded, in the same order that they should be applied, in the Server Configuration Document. Specific version numbers for each piece of software and patch should also be recorded. As upgrades to system software are made, the version numbers in the Server Configuration Document should be updated.

As part of disaster recovery and contingency planning, the Server Configuration Document can be tested periodically on staging servers to ensure that it faithfully reflects and reproduces the current production server environment.

Issues

There are issues associated with including rationale information in the Server Configuration Document. Developers and administrators do not enjoy writing documentation in general. Writing anything beyond the basic description of configuration steps could be a challenge, even when the value of the rationale information is explained. Similarly, some administrators will prefer to not read long documents, so the rationale information must be structured in such a way that it is not distracting to those who do not require it.

Another issue with documenting the server configuration involves the trade-off between experimentation and documentation effort. Often times, especially early in the development effort, the server configuration is unstable as developers and administrators experiment in search of a stable configuration. It might seem like a waste of time to document all of the server configuration experiments before the initial configuration is discovered. Documenting all experimental configurations can be useful in managing the process though. In addition, documenting throughout the experimentation process avoids the situation in which a stable configuration is finally discovered, but the steps that took place cannot be remembered and consequently reproduced.

In addition to documenting the server configuration, it might be prudent to maintain a repository of system software and patches applied to the Web server. If the Web server is reliant on specific versions of software that might not be maintained or might become unavailable, saving versions of specific software in a repository could be the difference between success and failure in reconstructing a Web site from scratch.

Examples

We have first-hand knowledge of at least two substantial federal Web systems that employ this pattern in practice: the complete configuration of each system is fully documented. As a part of quality assurance for the documentation, administrators are challenged to rebuild the system using only that documentation. The resulting systems are then subjected to thorough usability and security testing. Any changes to the system are performed on staging server and subjected to quality assurance before being deployed to the production systems.

We utilized this pattern in the development of our Web repository application, producing the resultant example Server Configuration Document.

Trade-Offs

Accountability	No effect.
Availability	This pattern increases availability by providing developers and administrators with a document describing Web and application servers' baseline configurations. This will facilitate restoration or reconstruction of a Web site in the event of site compromise, thus minimizing downtime.
Confidentiality	No effect.
Integrity	This pattern contributes to integrity by providing a baseline by which integrity can be accurately assessed in the event of site compromise.
Manageability	This pattern enhances manageability in a variety of important ways. The process of documenting helps manage the complexity of the server configurations. It also provides insurance against the loss of the only employee(s) with the knowledge required to keep the Web site running. In the most extreme situations where the entire Web site must be rebuilt, this pattern enables manageability of the entire recovery process.
Usability	No effect.
Performance	No effect.
Cost	The process of documenting server configurations will incur some additional up-front cost in terms of administrator time and effort. The long-term maintenance cost, especially in circumstances requiring Web site recovery, should be reduced significantly though

Related Patterns

- *Build the Server from the Ground Up* – a related procedure for understanding and configuring Web and application servers properly; the process of configuring servers described in this related pattern is orthogonal to the documentation process described in this pattern.

- *Test on a Staging Server* – a related procedure for development and testing of Web applications that explores the impact of changes on a separate installation before committing them to the production system; this related pattern describes how quality assurance of the documentation produced in this pattern should occur.

- *Patch Proactively* – a related procedure for administering servers; any patches applied to the system as recommended in this related pattern should be documented continuously as part of the Server Configuration Document in this pattern.

References

[1] Allen, J. *The CERT Guide to System and Network Security Practices*. Addison-Wesley, 2001.

Enroll by Validating Out of Band

(a.k.a. Round-Trip Authentication)

Abstract

When enrolling users for a Web site or service, sometimes it is necessary to validate identity using an out-of-band channel, such as postal mail, telephone, or even face-to-face authentication. The out-of-band channel can be used to establish a shared secret, which can then be used to establish identity during enrollment.

Problem

Web sites are accessible to a vast audience of potential users. While some sites will simply offer data for anonymous retrieval, many will find greater value in establishing relationships with regular users. Some Web sites are even designed primarily to service an existing customer base, offering a convenient interface that supplements more traditional customer service.

Enrollment is the problem of establishing long-lived user accounts. The primary purpose of this (and other) enrollment patterns is to establish authentication credentials (usually a password) so that the Web site can reliably authenticate the user on return visits. Once the authentication credentials are established, the site will be able to maintain sensitive user data and offer services that require the approval of the customer.

Sites that offer transactions of high value must be very careful to validate a user's identity at enrollment. These sites cannot depend on authentication over the network because there are too many ways in which an attacker might be able to impersonate some other person. Furthermore, there is currently no accepted universal authentication model.

Solution

Out-of-band validation falls back on conventional methods of authenticating a user's identity. Under this pattern, enrollment is a two-step process. First, the user initiates the enrollment process via the Web site. Any necessary data is provided via a Web form. This ensures that the user will be able to enter the data directly, reducing costs and improving accuracy.

Once the data is collected, the account is created and marked as *enrollment pending*. During this time, the user may be able to access some low-value services on the site. But any high-value transactions will be disallowed until the enrollment is completed. Typically, this will mean read-only access to the customer's data. But if the data is sensitive it may preclude any access to account-specific data.

Enrollment is completed using an out-of-band process. There are many different ways in which this can take place. Some examples are:

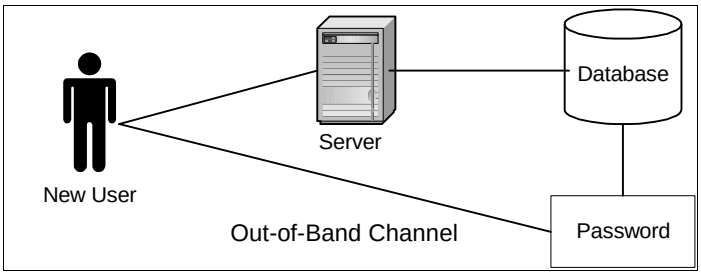
The user may be asked to present him- or herself at a local office (e.g. a bank branch) in order to validate his/her identity and receive an initial password. The user must then login and change the password in order to complete enrollment.

The system may mail out an initial password to the user's address on record. This is not an address that the user enters at enrollment, but an existing address that the system already associates with the user's claimed identity. The user must then login and change the password in order to complete enrollment.

The user may be asked to telephone a customer service representative and be able to authenticate him- or herself using some pre-existing shared secrets (e.g. a PIN or social security number and mother's maiden name).

If enrollment is not completed in a relatively short time period (2-3 days plus time for mail delivery when required) the account should be disabled or locked out (see Account Lockout).

One slight variation to this pattern eliminates the initial data collection step. Instead of waiting for users to enroll, some systems actually enroll their users a priori and then mail an initial password to each new user. This is common in systems where an existing user base is to be migrated from conventional (e.g. telephone) customer service systems to a new Web-based system.



Issues

The weakest link in many enrollment systems is the human customer service representative. First-line customer service representatives should follow scripted instructions when authenticating users. They should have little or no discretion in interpreting user input. They should be terse and businesslike. Many attackers have persuaded customer service representatives to divulge information or accept incorrect responses.

The most secure out-of-band communication is initiated by the system requesting enrollment, not the user. If the user initiates contact, the details of the event should be recorded. Face-to-face authentication can be videotaped. Caller ID can be used to record telephone numbers. Even the network address associated with the original request should be recorded as part of the enrollment record.

When the system initiates out-of-band contact, it should always be directed to an address that is already on record or received from a reputable third party. If the contact uses an address that is

provided as part of the enrollment, that represents a validation of the address, not the user's identity. See E-mail Validation for a discussion of that approach.

Enrollment requests should be carefully monitored. In particular, administrators should look for patterns that might indicate potential misuse. If a number of enrollment requests originate from the same address or contain largely similar data, they should be flagged for manual review before enrollment can be completed.

Web Banking Systems

This pattern is particularly applicable to banking systems, and should be considered standard practice in the banking industry. In particular, best practice is to require users to "opt in" for Internet access. No account should be made Internet accessible without explicit approval of the user. An out-of-band enrollment solution would have the user contact their bank and (a) explicitly authorize Internet access and (b) retrieve or choose an initial password.

Most banks already share secrets with their users, however those secrets are often 4-digit PINS, which cannot be safely exposed to the Internet. It is simply too easy to exhaustively search the 10,000 possible combinations using automated attack scripts. However, PINs are perfectly acceptable for automated telephone-based systems. A telephone based system can use a PIN to authenticate a user and then either provide an Internet password directly, or cause a password to be mailed to the user's address of record.

Variation: Out of Band Notification

A variation to this pattern uses an out-of-band notification rather than out-of-band password delivery. Under this approach, a customer that wishes to sign up for Web-based account service will be asked to provide weak authentication data (e.g. an existing account number) and will be allowed immediate system access.

At this point, a postcard or letter is mailed to the address of record for that customer. The purpose of this mailing is to alert the user to the fact that an account was registered in his/her name. If the user did not initiate this enrollment he/she is instructed to contact customer service (at which time the account would be disabled). This approach allows the site to be immediately available, but also ensures that misuse will be detected. It also provides legitimate users with a strong sense of security.

This approach is appropriate for relatively low-risk activities. A banking system should not use it. But it would be appropriate for on-line interfaces to conventional billing systems (e.g. a cable TV company). However, even the cable TV company should disallow requests to cancel service from the Web site.

Variation: E-mail Validation

(***Editor's note***: This may warrant an entire pattern to itself. It is a variant of out-of-band validation, but it is applicable to a completely different problem. Most importantly, it is possibly the most popular enrollment mechanism on the Internet.)

Many Web systems use e-mail to perform out of band validation of an e-mail address. When a user enrolls, he/she is asked to provide an e-mail address. Before the system will establish the account, the user is e-mailed a secret (typically a URL that contains a lengthy unique address or parameter). The user must enter that secret to complete the enrollment process.

E-mail validation does not authenticate the user's identity. It merely ensures that the e-mail provided to the system is one that can be received by the user. There are a number of uses for this approach:

- Many sites wish to be able to contact users. The e-mail customer list is a valuable asset, and the first step in developing long-lived customer relationships.

- Verisign class 1 certificates, which are used to encrypt e-mail, guarantee only that the holder of the certificate is able to receive e-mail at the address in the certificate. E-mail validation proves that the enrolling user can receive e-mail at that address.

- Many sites offer mailing list services. Before they will add a user's address to the mailing list, as a courtesy they check that the user owning the e-mail address really wanted to be subscribed to the list. This protects the mailing list server from misuse, in which an attacker signs up other accounts to receive mail it does not want.

Possible Attacks

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

- Eavesdropping – attackers will try to capture information on the out-of-band channel. This may thwart many attackers because it can be very difficult to eaves drop on certain out of band channels.

- Social engineering – since this policy may require human interaction it may be the subject of social engineering attacks. The attacker may try to impersonate the user or manipulate personnel at an organization.

- Race condition – attackers may either have the secret or be attempting to enroll with the captured secret before the user has a chance to.

Examples

There are many examples of this pattern in practice:

- Certificate authorities. Verisign requires that applicants for Class 2 and 3 certificates present themselves in person. Requests for corporate certificates require that an officer of the company present documents validating the claimed corporate affiliation.

- Local system administrators. University campuses and corporations that offer intranet services often require users to present themselves in person.

- Web banking. As noted above, most banks require a telephone communication before they will allow any account access via an Internet portal. They also typically use out of band notification to the address of record.

- Utilities. Many utilities have developed on-line bill payment systems. They typically use out-of-band notification to alert users when Web access to their accounts have been enabled.

- The Internal Revenue Service. In past years, the IRS made mass-mailings of e-file PINs to taxpayers that have not previously filed electronically.

- Lost passwords. Many sites use out-of-band delivery mechanisms to respond to lost password requests. Once a user has provided a mailing address, mailing a new password to that address is a safe way to ensure that the same user receives the new password.

Trade-Offs

Accountability	This pattern provides a great deal of accountability by ensuring with a high degree of confidence that actual individuals can be associated with on-line identities.
Availability	This pattern negatively impacts availability to new users. Until the enrollment information is received on the out-of-band channel, the system cannot be fully utilized.
Confidentiality	This pattern helps ensure that attackers will not be able to enroll for Web access to existing customer accounts.
Integrity	This pattern helps ensure that attackers will not be able to enroll for Web access to existing customer accounts.
Manageability	This pattern complicates manageability because the out-of-band channel must be incorporated into the management of the system.
Usability	This pattern lessens usability because users cannot enroll prior to receiving their information.
Performance	N/A
Cost	This pattern increases cost because an out-of-band channel must be used to communicate enrollment information; this out-of-band channel can have costs associated with its usage (e.g., postage) or with its procedural implementation (e.g., customer service staff).

Related Patterns

- *Enroll using Third-party Validation* – an alternative pattern describing another enrollment procedure.
- *Enroll with a Pre-Existing Shared Secret* – an alternative pattern describing another enrollment procedure.
- *Enroll without Validating* – an alternative pattern describing another enrollment procedure.
- *Password Authentication* – a related pattern that relies on an enrollment pattern for establishment of a user password.

References

None.

Enroll using Third-Party Validation

Abstract

When enrolling users for a Web site or service, it is always easier to allow some other party to take on the difficult task of authenticating user identity. When a third-party service is available and sufficiently reliable, the Web application can offload this task on the third party. This approach is becoming more common as third-party services become available. The most common form of transaction authentication—credit card authentication—is a form of third-party validation.

Problem

Web sites are accessible to a vast audience of potential users. While some sites will simply offer data for anonymous retrieval, many will find greater value in establishing relationships with regular users. Some Web sites are even designed primarily to service an existing customer base, offering a convenient interface that supplements more traditional customer service.

Enrollment is the problem of establishing long-lived user accounts. The primary purpose of this (and other) enrollment patterns is to establish authentication credentials (usually a password) so that the Web site can reliably authenticate the user on return visits. Once the authentication credentials are established, the site will be able to maintain sensitive user data and offer services that require the approval of the customer.

For some Web sites, it may be practical to outsource the problem of validating the identities of end users, or other critical data provided by those users.

Solution

Third-party validation off-loads the burden of establishing identity to a third party. When a user wishes to enroll with the site, they are directed to the third party service. Once they have enrolled with that service, they can authenticate themselves to the site. If the site wishes to perform additional enrollment tasks, those can be performed the first time the user authenticates himself/herself to the site.

There are many different mechanisms for performing third-party validation. Some significant approaches are:

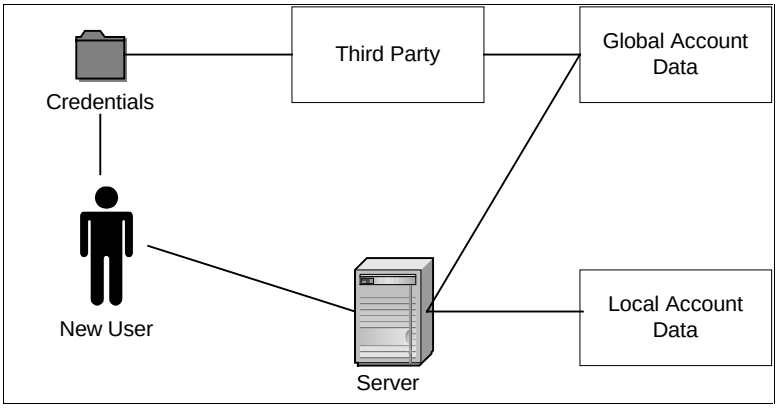
- The site offers a login screen that submits login requests directly to the third party. On successful login, the third party returns a ticket to the client. The ticket is encoded with secrets shared between the site and the third-party service. The site validates the ticket and allows the user access without having to communicate with the third party. However, if the client has never enrolled with the site, the site may have to request additional identity information from the third party.

- The site collects authentication information directly from the user and submits it to the third

party for validation. On successful login, the third party provides the client with any needed enrollment information.

- The third party provides the client with signed credentials. The client presents these credentials to the site and the site can validate them against the public key of the third party. The site should check the credentials against a list of revoked credentials, distributed periodically by the third party. If the client has never enrolled with the site, the site may have to request additional identity information from the third party.

Note that this pattern has many applications beyond enrollment. It can be used to authenticate one-time transactions as well. This allows an anonymous site to offer services to the general public without the overhead of maintaining accounts for every user who has ever used the site.



Issues

When implementing third-party validation, it is crucial that you follow the instructions and guidance provided by the third party. The third part will have experience in the use of their service and will have learned many of the more subtle problems through experience. Don't relearn the lessons they already know. More significantly, failure to adhere to the third party's processes may absolve the third party of any liability should a failure occur.

Before depending on a third party, you must understand the liability they assume, *if any*. Credit card companies generally absolve the site of risk if certain basic, well-documented procedures are followed. Certificate Authorities on the other hand generally disclaim any sort of responsibility for any damages caused by mistakes on their part. Because the case law regarding certificate authentication is virtually non-existent, relying on a third-party digital ID is very risky.

Consider how trustworthy the third party is, how easily are they subverted? Historically, have there been incidents reported that indicate flawed procedures? Trust in the identity of the client is only as strong as the trust in the third-party authentication.

Ideally, the third party should do the actual validation of authentication information. This prevents an application from needing to be the middleman, lessening the chance of a mistake or bug.

Issues affecting the third-party system will also affect your system. For instance if the third-party system is unavailable your system may also be unavailable.

Possible Attacks

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

- Vulnerabilities in the third-party system will also affect yours.
- If the third party requires on-line validation, denial-of-service attacks launched at the third party will effectively deny service to all sites depending on that service for access decisions.

Examples

There are many examples of this pattern in practice:

- ***Microsoft Passport/Sun Liberty:*** Microsoft has popularized the idea of third-party validation through their passport service. Initially, only Microsoft sites such as hotmail and msn use passport – they offer no other authentication options. Other sites, such as e-Bay have made passport authentication an option. We know of no non-Microsoft site that depends exclusively on passport for authentication. The Liberty Alliance, led by Sun, will eventually offer a similar service.
- ***Credit Card Validation:*** Many Web sites allows users to browse or shop without enrollment or authentication; however, when the user attempts to purchase an item, the Web site will require that the user provide a credit card number, a name, and a billing address. Before the transaction is processed, these items are submitted to an electronic validation service. The service provides a response that indicates the degree of validation. The site can decide what level of validation is acceptable, depending on the risk associated with the transaction.
- ***Bank Account Validation:*** Any site that offers electronic funds transfers must follow NACHA's regulations governing the use of the Automated Clearinghouse Network [1]. Before initiating any electronic funds transfers, the site must issue a prenote, which validates the name and address provided by the user against the account number. The Automated Clearinghouse Network acts as a proxy – each bank receives and responds to queries associated with its accounts. Because the prenote process can take several days, the enrollment must be divided into two steps, as with Out of Band Validation.
- ***Certificate Authorities:*** Certificate authorities such as Thawte and Verisign issue end user certificates that identify a user by name to an SSL-protected server. The Thawte certificates depend on a Web of trust, where individuals authenticate other individuals that have already been authenticated to Thawte. Verisign class 2 and 3 certificates use face-to-face authentication to ensure that the certificate accurately identifies the user. Note that Verisign class 1 certificates contain the claimed name of the user, but the certificate only vouches for the e-mail address, not the name.
- ***National ID services:*** At present, there are no public key infrastructures that can reliably

authenticate the general public. The US Post Office has long talked about creating a public key infrastructure, but proposed designs have never been implemented. At present, only certain government and military user bases have access to PKI services. If a site caters to those communities, it may be able to take advantage of one of those services.

Trade-Offs

Accountability	The effect this pattern has on accountability depends on whether the third-party mechanism is stronger than the alternative.
Availability	This pattern can reduce the availability of the system if the third party suffers from availability problems and the mechanism requires on-line access.
Confidentiality	The effect this pattern has on confidentiality depends on whether the third-party mechanism is stronger than the alternative.
Integrity	The effect this pattern has on integrity depends on whether the third-party mechanism is stronger than the alternative.
Manageability	This pattern could either enhance or reduce manageability, depending on the interface with the third party.
Usability	This pattern often increases usability because the third-party validation is often implemented such that users do not have enter authentication credentials multiple times.
Performance	This pattern can impact the performance of the system if the third party is slow to respond to validation requests.
Cost	This pattern could reduce costs because enrollment and authentication procedures do not have to be implemented in house. However, the costs of certificates are not insubstantial. If the site depends on a third party, the possibility exists that the third party may charge (additional) fees for the service without warning.

Related Patterns

- *Enroll by Validating Out of Band* – an alternative pattern describing another enrollment procedure.
- *Enroll with a Pre-Existing Shared Secret* – an alternative pattern describing another enrollment procedure.
- *Enroll without Validating* – an alternative pattern describing another enrollment procedure.

- *Password Authentication* – a related pattern that relies on an enrollment pattern for establishment of a user password.

References

[1] NACHA Internet Council. *Understanding Internet-Initiated ACH Debits*. http://internetcouncil.nacha.org, 2002.

Enroll with a Pre-Existing Shared Secret

Abstract

When enrolling users for a Web site or service, sometimes it is sufficient to validate identity using a pre-existing shared secret, such as a social security number or birthday. The use of a pre-existing shared secret enables enrollment without prior communication specific to setting up an account.

Problem

Web sites are accessible to a vast audience of potential users. While some sites will simply offer data for anonymous retrieval, many will find greater value in establishing relationships with regular users. Some Web sites are even designed primarily to service an existing customer base, offering a convenient interface that supplements more traditional customer service.

Enrollment is the problem of establishing long-lived user accounts. The primary purpose of this (and other) enrollment patterns is to establish authentication credentials (usually a password) so that the Web site can reliably authenticate the user on return visits. Once the authentication credentials are established, the site will be able to maintain sensitive user data and offer services that require the approval of the customer.

Some Web sites are developed to serve an existing community. The Web site administrators know personalized information about the prospective user base. In addition, the site may not be able to justify incurring the costs associated with out-of-band authentication of each individual user.

Solution

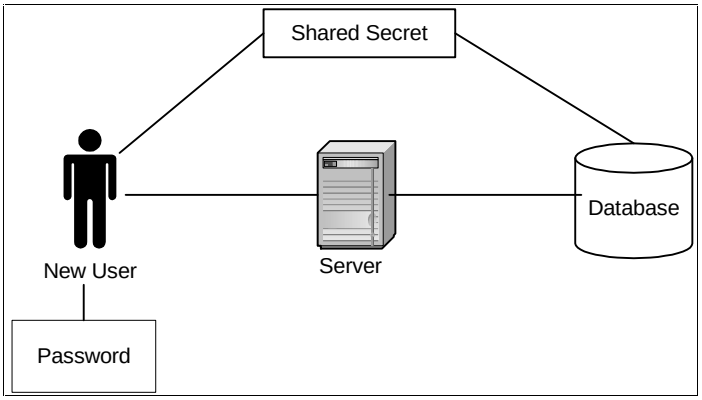
Enrollment Using a Shared Secret takes advantage of pre-existing knowledge of eligible users. The site database is pre-loaded with enrollment information specific to each user. The enrollment form requests that users provide that data. The user base is then informed of the site via some form of broadcast medium. Users who enroll at the site are required to provide personal information that the site can validated, but that most other people would not know.

For example, a university course registration system can use student names, social security numbers, and birthdays to enroll new users. The University can publish instructions for all students that don't have to be individually tailored to the student. On enrolling, the system can validate the student's identity using these pieces of data, already known to the university.

After users successfully authenticate themselves using the pre-arranged shared secret, they will be redirected to an initial password selection screen. At this point, the user will be prompted to supply two identical copies of their chosen password, in order to complete the enrollment process. Once enrollment is complete, the account it marked as enrolled.

After a relatively small number of incorrect guesses, the account should be locked and require manual (out-of-band) authentication. This provides protection against social engineering attacks, but still ensures that only a fraction of the users will incur the costs of out-of-band authentication.

If an attacker successfully enrolls a user account, the legitimate user will not be able to enroll later. This places the burden of discovery on the shoulders of the user, who presumably has an interest in protecting his/her account. Alternatively, if an attempt is made to enroll an already enrolled account, the system administrator could be alerted and the account locked.



Issues

This pattern relies heavily on the fact that most users will want to enroll. Even a weak shared secret can be appropriate, if all users will attempt to enroll in short order. If a secret is guessed, the user whose account was enrolled will be unable to enroll again, and will bring this to the attention of the system administrators. But if large numbers of users never bother to enroll, their potential accounts will be vulnerable to an imposter who is able to figure out the shared secrets.

Make sure to educate users about how to validate that the server they are speaking to really is the correct server and not an impostor. If the value of the data warrants encryption in transit, the enrollment forms should be similarly protected. Consider sending out https based URLs (and call attention to the ‘s’).

This approach can be combined with out-of-band authentication. For example, a secret personalized URL can be e-mailed to users. The users are then instructed to validate their account using a user-specific shared secret.

Some systems actually use the shared secret as an initial password. This allows the login screen to double as an enrollment screen. If this approach is used, the password must be flagged as expired, forcing the user to change the password before other access is possible. This is critical to the pattern because it ensures that an attacker cannot compromise the account without causing changes that would alert the legitimate user when he/she attempts to first login.

Lock out enrollment of a user account after a relatively small number of failed attempts. Instead of directing the user to customer service, it is more secure (and possibly less expensive) to have customer service contact the user through their known address.

Selecting Secrets

Carefully consider the strength of the pre-existing shared secret. Many pre-existing secrets are not completely secure, because the information can be discovered using other means. For example, when the secret is a birthday or social security number, an attacker might be able to use open sources to collect this information. Instead of using a single piece of information consider using multiple pieces and validating them all to authenticate the user.

Shared secrets may have to be more tolerant of errors than passwords. Words that are prone to alternative spelling, capitalization, etc. can cause problems.

Be extremely specific when describing the shared secret. Instructions that seem obvious to you ("the amount of your third estimated tax payment from last year") can be very confusing to users ("original or amended?", "calendar year or fiscal year?")

Possible Attacks

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

- Social engineering – many corporations outsource employee benefits servicing to managed Web sites, providing those sites with a database of employee data from which to derive shared secrets (e.g. "hire date"). If an attacker determines what sort of information is used, they could easily gather the information using social engineering techniques. Users may be less hesitant to divulge their hire date than they would a password.

- Eavesdropping – attackers will try to discover the shared secret by eaves dropping on connections.

- Man-in-the-middle attack – if an attacker can trick a user or redirect their browser to an alternative site, they can challenge legitimate users for the shared secrets. They can then use those shared secrets to authenticate themselves as the user. Alternatively, they may simple forward the enrollment request and observe the password selected by the user.

- Brute-force account and shared secret guessing – attackers develop automated guessing tools that will try many different common shared secrets and accounts.

Examples

There are many examples of this pattern in practice:

- ***Internal Revenue Service:*** The IRS is charged with increasing the number of taxpayers using electronic interfaces to existing tax services. They have developed a variety of systems that provide service for both businesses and individual taxpayers. Most of these systems use pre-arranged shared secrets. For example, when initially enrolling for electronic submission of tax forms, individuals are asked to supply detailed information from past years' tax returns.

- ***University Course Student Information Systems:*** Many universities use this type of system.

For example, when a student first logs on to the Integrated Student Information System at the University of Virginia, the student's user name is his/her social security number and the password is his/her birthday.

- ***www.benefitaccess.com:*** This site is used to administer benefits for employees, it allows employees to enroll by using a pre-existing shared secret. Each employer selects the appropriate shared secrets.

Trade-Offs

Accountability	This pattern offers limited accountability, depending on the strength of the shared secrets. When social security numbers or birthdays are used, the result is not a particularly insecure enrollment process. As a result, users can disclaim actions performed on their account.
Availability	This pattern helps make full functionality of the Web site available to users immediately (see Usability).
Confidentiality	This pattern can effect the confidentiality of the Web-site if the pre-existing shared secret is chosen poorly; allowing illegitimate users to login to the site.
Integrity	Same as confidentiality
Manageability	This pattern can simplify management because it eliminates manual authentication of users. However, if the instructions are unclear or the secret too impractical, it could create a significant management burden.
Usability	This approach to enrollment has positive impact on usability because it allows users to immediately access the Web site without waiting for out-of-band authentication. However, if the shared secret is too obtuse (e.g., the processing number from your last tax return) it can be a hassle for users to discover.
Performance	N/A
Cost	This pattern is significantly cheaper than out-of-band authentication, because most user accounts will be enrolled without requiring manual intervention.

Related Patterns

- *Enroll by Validating Out of Band* – an alternative pattern describing another enrollment procedure.

- *Enroll using Third-party Validation* – an alternative pattern describing another enrollment procedure.
- *Enroll without Validating* – an alternative pattern describing another enrollment procedure.
- *Password Authentication* – a related pattern that relies on an enrollment pattern for establishment of a user password.

References

None.

Enroll without Validating

(a.k.a. Bootstrap Trust, Grow Credibility)

Abstract

When enrolling users for a Web site or service, sometimes it is not necessary to validate the identity of the enrolling user. When there is no initial value involved in the Web site or service for which enrollment is occurring, validation is an unnecessary procedure and can be eliminated.

Problem

Web sites are accessible to a vast audience of potential users. While some sites will simply offer data for anonymous retrieval, many will find greater value in establishing relationships with regular users. Some Web sites are even designed primarily to service an existing customer base, offering a convenient interface that supplements more traditional customer service.

Enrollment is the problem of establishing long-lived user accounts. The primary purpose of this (and other) enrollment patterns is to establish authentication credentials (usually a password) so that the Web site can reliably authenticate the user on return visits. Once the authentication credentials are established, the site will be able to maintain sensitive user data and offer services that require the approval of the customer.

There are some Web sites and services in which there is no need to associate accounts with actual individuals. It is only necessary to ensure that on subsequent visits, the account is only accessible to the user who created the account. In these cases, the account has no value prior to enrollment – any value is created over time by the user.

For example, there are a number of sites that offer free Web-based e-mail accounts. These sites have no need to initially validate a user's identity. Once the account is in use, however, the site must protect it from others who may wish to eavesdrop on or impersonate the original account holder.

Solution

In situations where there are no initial items of value to protect and no need to associate accounts with specific individuals, the system can allow users to enroll without validating their identities. When a user wishes to create an account, the site allows the user to select a username. The site then creates initial authentication credentials. For password-based authentication, the user typically supplies both their chosen username and initial password on the same form. These credentials now represent a shared secret that can be used to authenticate the user on subsequent visits.

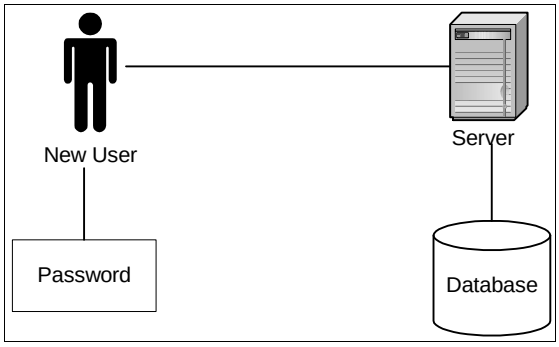
At the time of enrollment, the site should prompt the user to supply other identifying information (shared secrets) that can be used to later re-authenticate the user in the event that the credentials

are lost. This is necessary because the site cannot rely on traditional user validation – it doesn't reliably know the identity of the user.

If the site uses encryption to protect password submissions, it should also use encryption to protect the initial password submission screen. The question of when encryption is warranted is addressed in Password Authentication.

The password enrollment screen should contain the user's chosen account name and two copies of the initial password. On the server, the username should be checked and the enrollment rejected if the name is already in use. The two passwords should be checked and accepted only if they are identical.

When authentication technologies other than passwords are used, the situation is only slightly different. Either the client or the server generates the keys that will be used by the client. The client stores the keys and the server stores whatever data is needed to authenticate the keys.



Issues

Anonymous Users

This pattern allows anonymous users to enroll with a site and be authenticated again later. Since users are anonymous there is often little value in a new account, instead value grows over time. For instance a new e-mail account is worth less to a user than an e-mail account that has been established for some length of time. Since accounts are anonymous it is possible for a single user to register many accounts with out being able to verify that the users are unique. This means that one user could be responsible for a large number of accounts.

There may be legal ramifications if you provide anonymous users with e-mail, open forums, file storage, or other services. Users may slander, defile, issue death threats, conspire to break the law, traffic in child pornography, or use a server to distribute stolen intellectual property. Research the relevant case law before providing the unwashed masses of the Internet any anonymous services.

Consider tracking logins/logouts and source IP address of the user if it may later be needed. But be careful about how much information you collect from the user. Anything you collect can be something you are later subpoenaed for. Log only that which you need to protect yourself from litigation.

Although the user may be asked to provide other identifying information (name, address, phone, etc.) it is important to realize that these entries are not being validated and are really no more than a convenience. The site should be careful not to give other users any indications that these values have been authenticated. For example, e-bay refers to its users only by their account names, not their given names. The latter were provided by the users and never authenticated. However, see Client Input Filters for information about sanity checking the data provided.

Account Management

Be wary of the possibility of some sort of automated enrollment attack. If patterns of enrollments (or even multiple enrollments from a single IP appear), it may be necessary to alert the system administrators or rely on a Network Address Blacklist.

If accounts have no value to anyone other than the user, it is natural for many accounts to be created and then lie dormant. In order to constrain usage, the system should include a policy on how long accounts will be allowed to remain dormant before they are expired and their resources reclaimed. However even after an account expires, it may be unwise to allow new users to choose an account name that was recently in service.

It is generally necessary to make extra provisions against the loss of a password. When the password is initially submitted, it should be submitted in duplicate. Current best practice is to present users with a series of questions and collect personalized answers. In the event that the password is lost, these questions should help customer service manually re-authenticate the user and reset the account password. See Password Authentication.

This pattern is often combined with round-trip validation using e-mail. E-mail is a good way to build relations with customers, and it is simply good form to ensure that an address actually belongs to the user in question before deluging it with commercial messages. Do not fall into the trap of thinking that a unique e-mail address identifies a unique individual. There are many ways that a single user can create an arbitrarily large number of unique e-mail addresses.

Possible Attacks

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

- Resource consumption attacks – an attacker may create many accounts and consume an extreme amount of resources

- Abuse of services – since there is virtually no way to verify the identity of the user, there is no accountability associated with the user so abuse of services is more likely.

- Site spoofing (man-in-the-middle) – a malicious site can use references and redirections to copy present all the content from a legitimate site. To the unwary user, the site would be virtually indistinguishable from the original. To the original site, the malicious site would appear to be a normal user. But the malicious site can ensure that critical user interactions (such as picking a password) go through its servers where they can be monitored, recorded, or tampered with.

Examples

There are many examples of this pattern in practice:

- ***Yahoo mail, Hotmail, IRC, etc.:*** Free Web services, such as e-mail and chat rooms, do not require validation during enrollment because there is nothing of value involved initially. For example, when signing up for an free e-mail account with Hotmail, a new user creates a user name and password during enrollment, but the user name (the identity of the new user) does not need to be validated because the new user name has no value. The user enrolling that user name will build value through the usage of the account.

- ***Usenet, Open Source Community, etc.:*** Certain communities of users do not require validation of identity during enrollment because new users will build an identity through interaction over time. For example, users posting to Usenet groups build a reputation through repeated postings; initially, no validation of identity is performed on the user name or PGP key attached to Usenet postings, but the community as a whole associates an identity with a particular user name or PGP key over time.

Trade-Offs

Accountability	This pattern sacrifices accountability for reduced cost because a user is allowed to enroll with any identity he or she chooses. Depending upon the type of services depends upon the accountability that the user will have. For instance, sites like Ebay where users build a reputation require that the user have some accountability, where as a site that offers little value requires little more accountability.
Availability	This pattern could compromise availability because if a Web site or service allows users to enroll without any validation, a denial-of-service attack could be performed against the enrollment mechanism.
Confidentiality	This pattern should not affect confidentiality because it should only be applied in circumstances where there is no data to protect, or where the effected data is only valuable to the user.
Integrity	This pattern should not affect integrity because it should only be applied in circumstances where there is no data to protect, or where the effected data is only valuable to the user.
Manageability	This pattern should facilitate manageability because no enrollment mechanism must be designed and maintained.
Usability	This pattern should enhance usability because no extraneous registration steps are imposed on the user.

Performance	This pattern should improve performance because no enrollment procedures are implemented.
Cost	This pattern could reduce cost because no enrollment steps are implemented. If the users require customer support the cost could be high because users have no reason to be strict about remembering their authentication information. In addition, the costs of maintaining an unbounded number of accounts could get very high.

Related Patterns

- *Client Input Filters* – a related pattern for validating client data, which is especially important when enrolling users without validating authentication credentials.
- *Enroll by Validating Out of Band* – an alternative pattern describing another enrollment procedure.
- *Enroll using Third-party Validation* – an alternative pattern describing another enrollment procedure.
- *Enroll with a Pre-Existing Shared Secret* – an alternative pattern describing another enrollment procedure.
- *Password Authentication* – a related pattern that relies on an enrollment pattern for establishment of a user password.
- *Network Address Blacklist* – a related pattern describing a protection mechanism to be used in cases of particular client misbehavior.

References

None.

Log for Audit

(a.k.a. Audit the Logs, Unified Logging)

Abstract

Applications and components offer a variety of capabilities to log events that are of interest to administrators and other users. If used properly, these logs can help ensure user accountability and provide warning of possible security violations. The *Log for Audit* pattern ties logging to auditing, to ensure that logging is configured with audit in mind and that auditing is understood to be integral to effective logging.

Problem

As events occur during the life cycle of a system, some events are of particular interest to administrators and other users. Recording specific information about events that have occurred creates records that allow the system to be debugged, monitored for security events, and measured for performance. The term *logging* means the actual act of writing information about events to some type of permanent storage. The term *auditing* means actually examining this information and ensuring that all is as expected.

There are a number of important problems that logging and auditing solve. They provide evidence of accountability, reliability, performance, and security. Logs provide accountability by enabling verification of an event's occurrence and any users associated with that event. Reliability comes from the logging of errors. Performance analysis can occur when performance data is logged. There are many relevant types of event that must be logged to address security.

However, all of these benefits are only achieved if the logs are audited and appropriate information is logged to make the logs useful. If too much unnecessary information is logged, effective auditing will be more difficult; if essential information is not logged, it will be impossible.

Solution

Every major component is responsible for logging events that it considers noteworthy. Some of these will be tagged as security-relevant events, others will not. Each system will typically deliver these events in some non-standard format to permanent storage, using one or more predefined log files.

Whenever possible, these log files should be collected centrally and handled consistently. When applications allow the log format to be modified, these features should be used to make logs more consistent. All logs should be directed to a single disk partition, so that disk space can be managed and resource consumption attacks cannot be used to deny disk space to other critical components. A log rotating program monitors disk usage to ensure that the log partition has adequate space – periodically, old logs are moved to off-line storage in order to free disk space.

The most critical component of the Log for Audit pattern is the human that audits the logs. A regular schedule should be established for examining the logs. The auditor's concern is to uncover any interesting or unusual events. The data that is collected cannot be so voluminous that the auditor will not be able to spot the unusual events. Should the data become too large, more frequent audits should be scheduled.

Responsibility for auditing the logs should be distributed to those parties in the best position to recognize suspicious events, rather than having system administrators audit all logs by themselves. For example, database administrators should examine database logs while Web application administrators can examine application-specific logs.

If possible, log information should be propagated to a single location and stored in a unified log record at the back end. If this is not possible, a unique identifier should be generated for each request and that identifier should be a part of every log event. Unified log files will enable better diagnosis of related events, such as a single intrusion that affects multiple applications.

Issues

One of the most important issues regarding logging and auditing is what and how much information to log. Minimally, log data should include the time of the event, the source, and the application or object that generated the event. The logs should contain enough information so that someone reading the logs can locate the specific cause of the logged event in a timely fashion.

Applications should be configured for different logging streams (security, debugging, performance) so that administrators can configure where and how much of each type of log to store.

Consider using the default or system supplied logging system. Administrators will have tools to work with this system and will know where to look for the logs.

Plan how to react if logs fill before the log rotation interval. Do you stop the system or drop older/less important logs? This involves a trade-off between availability and accountability.

The storage of logs must be done in a secure fashion: there must be high assurance that the logs have not been modified. Consider controls against misuse. Admissibility as evidence requires that you be able to demonstrate that the logs could not have been manufactured. Sending logs to a separate system represents greater assurance that they cannot be tampered with. However the channel to the logging server must be suitably restricted (e.g. a serial line or one way flow).

It is important to recognize that there are two distinct classes of remote event that might be logged: those that can be associated with a specific user account, and those that can only be associated with a network address. In the former case, an accountability relationship exists – you known who the user is and presumably you have some business relationship with them. In the latter case, the user is truly anonymous, and you may have no recourse, no matter how badly they misbehave.

Understand the accountability relationship and log only what you need. Logging too many events will cause administrators and users to scrutinize the logs less carefully. Understand that you can be subpoenaed for anything you log. Speak to a lawyer about the rules of evidence for collecting and handling logs.

Unified Logging

A Web site will contain numerous components, all of which generate their own log files. The Web server, application server, application, firewall, and database all log data. It can be extremely difficult to correlate these logs. Where multiple logs are unavoidable, make sure that technical or procedural measures are in place to ensure that clocks do not drift significantly. If systems in different parts of the country will need to be correlated, set their clocks to a single unified time.

Though passing through log information to a single log makes auditing easier and more effective, it may significantly impact performance. If simple errant requests place a load on the back end, a denial-of-service attack is much easier to effect. Firewalls will also have logs that need to be correlated. However, the firewall generally is not permitted to alter or amend the incoming requests.

Possible Attacks

The logging system itself can be the target of misuse by an attacker that is trying to prevent detection. If logs are not protected from tampering, an attacker might attempt to modify the logs after the fact to cover his/her tracks. If tampering is not possible, an attacker might generate a large number of events around the time of an attack to obfuscate important indications of the actual attack. Similarly, an attacker might perpetrate a resource consumption attack by filling the log storage system with meaningless events so that more important events do not get logged later. Finally, a patient adversary might create a pattern of false alarms in order to condition the auditors to ignore a subsequent attack.

Examples

At the system level, UNIX offers a variety of standard log files, all of which are directed to the /var partition. The UNIX cron program is predefined to perform daily and weekly housekeeping tasks, including the rotation of log files. Old logs are compressed in order to save space and renamed with a unique suffix. UNIX also offers a syslog facility, which allows individual programs to generate events that will be inserted into the UNIX system log file. The syslog feature can also be redirected to another machine on the network, allowing a single centralized log file to be created using system log events from a number of different machines.

At the network level, Micromuse produces a unified logging system called NetCool. NetCool offers a number of different “probes” that can be used to collect data from a variety of sources, including conventional text log files, UNIX syslog streams, PIX firewall events, and Oracle table insertions. Micromuse’s list of case studies all represent examples of this pattern.

Trade-Offs

Accountability	Suitable logging and auditing can have a very positive effect on accountability. Poor controls can weaken accountability.
Availability	If full logs cause a denial of service, this can have a negative impact on availability.
Confidentiality	None.
Integrity	Suitable logging can help recover from failures and prevent system misuse.
Manageability	Controls on logs can hamper system administrators' freedom to operate.
Usability	Excessively detailed logging can impact performance adversely, but normal-case logging generally is negligible.
Performance	No effect.
Cost	This pattern could affect costs either positively or negatively. Human auditing of logs is very expensive, so if this pattern results in more logging and auditing than was previously being done, then costs will be significantly higher. The overall point of this pattern though is to log only what will be audited, so the end result of this pattern should be more cost-effective logging and auditing. The short-term costs of auditing can be recouped by more effective use of logs in the long term.

Related Patterns

- *Document Security Goals* – a related pattern that ensures the security goals are understood by all parties during system development. Some of the security goals that must be documented address the planned audit procedures and use of logs for accountability.

- *Share Responsibility for Security* – a related pattern that advocates distributing responsibility for security amongst the entire project team; similarly, this pattern recommends distributing auditing duties to those parties with a vested interest in that which is being protected, rather than having just system administrators audit all logs.

References

[1] Allen, J. *The CERT Guide to System and Network Security Practices*. Addison-Wesley, 2001.

Patch Proactively

Abstract

During the lifetime of a software system, bugs and vulnerabilities are discovered in third-party software, and patches are provided to address those issues. Rather than waiting for the system to be compromised before applying patches ("patching *reactively*"), administrators of software systems should monitor for patches often and apply them *proactively*.

Problem

All software systems contain bugs and vulnerabilities. The longer a software system is deployed and the more it is in use, the higher the probability that those bugs and vulnerabilities will be discovered. Patches are released to fix the bugs and vulnerabilities, and these patches must be applied in order to protect the system from exploits and compromises. During the window of time between when a vulnerability is announced to the public and when its corresponding patch is applied, the system is wide open to attack.

Unfortunately, many system administrators wait until a system is attacked and compromised before checking the lists of known vulnerabilities and applying patches. Often times, overburdened administrators choose to attend to daily tasks that arise, forcing them to patch *reactively*: waiting until after their systems are attacked and compromised to find and apply the appropriate patches.

The SANS (System Administration Network Society) Institute cites "failing to update systems when security holes are found" as one of *The Ten Worst Security Mistakes Information Technology People Make* [5]. Similarly, "failing to install security patches" is rated as one of *The Five Worst Security Mistakes End Users Make* [5]. Clearly both problems relate to system administrators maintaining Web systems and applications.

Solution

Administrators of software systems must minimize the window of time between announcement of a vulnerability and application of its patch by patching *proactively*. That is, administrators must monitor for vulnerability and patch announcements often and on a regular basis, patching their systems before they are compromised by the latest vulnerabilities.

There are automated mechanisms for patch distribution and application that facilitate patching proactively. These can be used when appropriate; however, many production Web sites require significant testing whenever changes are made, and automated mechanisms must be integrated with quality assurance procedures.

Patching proactively is a straightforward process:

- The *Administrator* monitors the *Vulnerability Database(s)* for announcements regarding the *System(s)* of concern.

- When a vulnerability is discovered and a software *Patch* provided, the *Administrator* obtains the *Patch* from the appropriate vendor *Patch Database*.

- The *Administrator* tests the *Patch* on a *Staging Server* to ensure that the *Patch* performs correctly with the existing configuration on the *System to Be Patched*.

- The *Administrator* backs up existing configuration and data files on the *System to Be Patched.*

- The *Administrator* applies the *Patch* to the *System to Be Patched*.

Issues

The patch *must* come from a trusted source. It is important to download patches from vendor Web sites rather than from any other random location on the Internet. It is conceivable that an attacker could subvert the patch and provide a rogue patch instead, though this attack would require a great deal of sophistication. (An example of this occurred with TCP wrappers, causing many administrators to download and install a Trojan horse patch.) It is also conceivable that an attacker could corrupt a patch. Whenever possible, one should obtain a checksum for the patch and verify that it has not been tampered with. Unless digitally signed, the checksum could be tampered with as well, so it is best to obtain the checksum from a different source than the patch itself.

Patches might break existing functionality. It is important to be able to back out of a patch installation in the case that the patch is incompatible with existing, required functionality. This trade-off between running with or without the most current patch must be considered carefully, however: while running without current patches is a short-term solution for incompatibility problems, there is an increased chance of compromise when known vulnerabilities are not addressed.

Some patches might overwrite or erase existing configuration data. For this reason, it is important to backup essential data, including configuration data, before applying a patch.

Furthermore, it is important to keep a log of what patches are applied, when, and by whom. If the server needs to be rebuilt, a log of its last known status is available. (See the *Document the Server Configuration* pattern.)

Examples

There are a number of third-party mailing lists and Web sites that will advise of the latest security vulnerabilities. Vulnerabilities are listed by product, their effects are explained, and workarounds or vendor patches are offered. Some sites offer alerting services that send e-mail notifications when vulnerabilities are discovered in specific products enrollees express interest in. CERT (www.cert.org), Security Focus (www.securityfocus.com) and Bugtraq are three primary resources.

Many server platforms also have e-mail lists for security alerts, separate from other e-mail sent to users.

Trade-Offs

Accountability	This pattern improves accountability indirectly because a proactively patched system is less likely to be compromised and subverted.
Availability	See Accountability.
Confidentiality	See Accountability.
Integrity	See Accountability.
Manageability	This pattern complicates manageability because additional effort must be expended to keep up with vulnerabilities and patches as they are announced and released. Furthermore, there is no guarantee that some patch will not break compatibility with other applications and features. Subjecting each new patch to rigorous quality assurance is a very labor-intensive activity.
Usability	No effect.
Performance	No effect.
Cost	This pattern increases the up-front cost of maintaining a system, because resources must be devoted to monitoring vulnerability and patch announcements for the systems being administered. However, this up-front investment could cost less in the long term because the patched system will be less vulnerable to attacks, which are inherently expensive to detect and recover from. In addition, it makes costs more predictable, because an ongoing program of testing and applying patches is a steady cost, whereas recovery from an attack is a high cost that can occur without warning.

Related Patterns

- *Document the Server Configuration* – a related pattern that advocates understanding and documenting the system configuration; this would include the patches that are applied as a result of this pattern.

References

[1] Allen, J. *The CERT Guide to System and Network Security Practices*. Addison-Wesley, 2001.

[2] Anonymous. “Patches, We need those steeking patches to fix my steeking software”. http://bofh.ucs.ualberta.ca/patches.html, unknown date.

[3] Microsoft Corporation. “Microsoft Strategic Technology Protection Program”. www.microsoft.com/security/mstpp.asp, March 2002.

[4] Rosato, R. “Best Practices for Applying Service Packs, Hotfixes and Security Patches”. http://www.microsoft.com/technet/security/bestprac/bpsp.asp, 2002.

[5] SANS Institute. “Mistakes People Make that Lead to Security Breaches”. http://www.sans.org/mistakes.htm, October 2001.

[6] Scambray, J. “Ask Us About...Security, August 2000”. http://www.microsoft.com/technet/columns/security/askus/au072400.asp, August 2000.

[7] Stein, L. *Web Security: A Step-by-Step Reference Guide*. Addison-Wesley, 1998.

Red Team the Design

(a.k.a. Use a Red Team to Attack the Design,
Assess Security Throughout the Development Life Cycle)

Abstract

Red teams, which examine a system from the perspective of an attacker, are commonly used to assess the security of a finished system. However, the earlier in development that a problem is found, the easier it is to fix. The *Red Team the Design* pattern effects a security evaluation of the application at the stage when it is most possible to fix any problems identified.

Problem

Current best practice demands that a system be "red teamed" before deployment. Typically, this means that the completed system is subjected to intense scrutiny by security experts, either internal to the organization or independent. However, leaving security validation to the end of the development process can result in the discovery of significant issues at a point where they are very expensive to resolve.

Solution

Instead of waiting until the system is completed, perform security red teaming of intermediate work products. In addition to uncovering mistakes, the red teaming process ensures that security issues will not be simply put off until late in the development process.

In a conventional waterfall approach to software development, this pattern translates to red teaming the design before it is implemented. Problems uncovered could then be addressed at a much lower cost than would be incurred if the same problems had been discovered after the implementation had been completed. Conducting a security review of the design ensures that designers do not ignore security at the design level.

In an iterative software development process, the security red teaming should take place against the earliest prototype. If the prototype does not adequately address the security requirements, those weaknesses can be addressed in the next iteration. Again, conducting security reviews of the earliest prototypes ensures that security is not simply put off indefinitely.

Ideally, the review should be conducted by independent security experts who were not involved with the development or design of the system. If this is not possible, the review team should at least consist of knowledgeable developers who are not part of the design team.

Issues

There are methodologies available for red teaming a system. The IDART methodology from Sandia National Labs provides a great deal of specific guidance on locating potential weak spots. A less heavyweight process is the Attack Trees approach developed at NSA.

Selecting a Red Team

Many people passing themselves off as security red teams have histories of criminal behavior. While there is a certain cachet to hiring “real hackers” this practice should be avoided. These individuals have demonstrated that they have questionable ethics and are not above behaving criminally. They should not be trusted with complete details of the security elements of your system. You can have no confidence that they will not turn around and sell those details to some other interested party.

Red teaming should be performed by people who are not invested in the system design. As is the case with code debugging and design inspections, the developer of a system will be blind to certain problems that appear obvious to external reviewers. The developer approaches the system with certain assumptions in mind—an effective red teamer will not have his/her thinking constrained by those same assumptions.

The red team is often well positioned to suggest adjustments to the design to deal with identified problems. However, be aware that this changes the role of the red team. They are no longer a disinterested party, but are now invested in the new design. As a result, it may be necessary to enlist additional eyes to look at the amended system. The need for strict independence is part of the rationale for not introducing a red team until the system is complete. However, an argument can be made that having the most security knowledgeable people contributing to the design is more likely to result in a secure system than arbitrarily restricting their involvement until that point where they can no longer contribute to the success, but only note the failures.

Areas of Investigation for Red Teams

If possible, skilled security practitioners should be employed for red teaming. If such individuals (or budget for hiring them) are not available, the following hints will help non-security experts uncover many potential problems.

The most important considerations are what specific targets must be protected, what the threats to those targets are, and what the consequences of a successful attack are. For example, how important is Web site defacement, disclosure of customer data, destruction of customer data, or extended system downtime? What would be the consequences of numerous spurious orders or special orders for products that can't be sold to anyone else? It is important to remember denial-of-service and other resource consumption attacks. There are many net denizens who enjoy crippling a site for no other reason than they are able to.

A primary source of errors is the automation of existing manual processes without understanding the environmental differences that may make those processes vulnerable. Many Web sites automate processes that were adequate when performed by humans, or conducted using paper forms or even telephone access. Because of the speed, inherent parallelism, and anonymity of the Internet, these processes may not be adequately secure.

It is common practice to provide the red team with full details of the implementation. This helps ensure that they will be able to locate problems that may not be apparent from outside the system. However, recognize that real attackers will not have this advantage. It is permissible to have some residual risk, but every effort should be made to prevent attackers from knowing what

those risks are. As an example, an attacker who is fully knowledgeable about the lockout mechanism could misuse it to disable customer accounts. But attackers who aren't aware of that mechanism would not be able to tell that their subsequent requests were being ignored by the system.

Understand environmental assets. Often people forget that other things affect how their system will perform, for instance the system hardware, the network leading to the system, etc. These things can also be subjected to attack. Consider what resources could affect the system. Consider how an outage in any component of the system can be used by an attacker. Consider how an attacker might be able to effect an outage in a specific component.

Break the system down level by level looking for potential points of attack. For instance consider attacks at the network, server, system, database, page, object and form level. Research common vulnerabilities of existing applications that are similar to the one being built. Many application designers introduce similar bugs into their applications when designing similar functionality; understanding these can help locate vulnerabilities in the current design. Similarly, research the security problems of existing components and technologies that are used in the system and consider how those problems may impact the overall system.

Remember that just as the goal of quality assurance is to find bugs and not to "prove the system works," the goal of red teaming is to find any security issues, not to prove that the system is secure. No system is completely secure; success is defined as finding the remaining security problems and assessing the remaining risks. Keeping this in mind can help make red teaming more effective, especially when it is done internally by members of the same design team.

Examples

Code reviews are a common approach to identifying security problems. The TCSEC [2] and Common Criteria both rely on independent code and design reviews by external evaluation labs. While those evaluations are theoretically restricted to assessment of finished products, in practice the independent review team often is involved for several iterations of problem identification and resolution.

The red teams from Sandia National Labs have stated publicly that they prefer to be brought in during the design phase so that appropriate security measures can be built into the systems they are evaluating.

On one system with which we are familiar, the developers of a Web interface to a legacy system assumed that it was safe to develop a straightforward automation of existing processes. Unfortunately, the existing processes were flawed and vulnerable to a mass enrollment attack. The existing processes had not been attacked because enrollment used mail-in paper forms, which made a mass enrollment expensive and likely to be detected. Over the Internet, however, the attack would have been trivial to mount and would have permitted a massive corruption of the accounts on the legacy system. Fortunately, we were brought in to examine the design. We identified the problem and suggested a relatively inexpensive workaround that mitigated the risk. However, had the red teaming not occurred until the system was complete, it would have required major code modifications at a much higher cost.

Trade-Offs

Accountability	Using a red team (at any point in the lifecycle) should help identify problems that might undermine accountability.
Availability	See Accountability.
Confidentiality	See Accountability.
Integrity	See Accountability.
Manageability	No direct effect.
Usability	No direct effect.
Performance	No direct effect.
Cost	Red teaming the design introduces additional development costs. If the team locates problems that would otherwise have gone undetected until the system was complete (or worse in production), the costs will be likely be recouped.

Related Patterns

- *Document Security Goals* – a related pattern that establishes the security policy, which the red team could use to focus attacks.

References

[1] Salter, C., S. Saydjari, B. Schneier, and J. Wallner. "Toward a Secure System Engineering Methodology". New Security Paradigms Workshop 1998, September 1998.

[2] National Computer Security Center. *DoD 5200.28-STD, Trusted Computer System Evaluation Criteria*. December 1985.

[3] Sandia National Laboratories. "Information Design Assurance Red Team (IDART) Home Page." http://www.sandia.gov/idart, August 2000.

[4] Schneier, B. "Attack Trees". Dr. Dobb's Journal, December 1999.

Share Responsibility for Security

(a.k.a. Non-Separation of Duty)

Abstract

The *Share Responsibility for Security* pattern makes all developers building an application responsible for the security of the system. Security consists of more than just encryption, anti-virus software, and firewalls. Any element of a system can have security concerns, and system developers have to understand and address those concerns. Use of this pattern avoids the common problem of "the security guy" or security team being pitted against the rest of the development team.

Problem

When security is addressed on a project, the development effort often designates a separate person or team as responsible for the security of the system: "the security guy" or the security team. Under these circumstances, the persons responsible for security do not always have the power to effect any actual change. They can make recommendations, but they are only responsible for the development of a few critical components, if anything at all. Developers of the system implement their components without having to take ownership of the security implications; many times this results in security being sacrificed at the expense of other functional and non-functional requirements for which developers are held accountable.

In addition, the creation of a specialized security guy or group can create an unnecessarily adversarial relationship between a development team and security team. When responsibility for security requirements and all other functional and non-functional requirements is divided between the security team and development team, both groups are left incapable of properly conducting trade-off analysis of requirements. The development team will make decisions independent of the security ramifications because they are not responsible for those requirements. The security team, on the other hand, might over-engineer the security of the system because they are not taking into account the other functional and non-functional requirements that must be balanced. (The latter occurred in one development effort where a separate security team over-engineered a secure network without understanding the full business impact. Lack of a clear process for management decision making resulted in long project delays and considerable wasted effort.)

Solution

The *Share Responsibility for Security* pattern distributes responsibility for the security of a system amongst all developers. No single person or team should be solely responsible for security.

One reason for distributing security responsibility is to balance security appropriately with other functional and non-functional requirements. All developers must understand the security

implications of their components in order to make appropriate trade-off decisions between security and other requirements.

Another reason to share responsibility for security is to encourage the development of secure systems from the start, rather than trying to bolt security as an add-on to the system after the fact. By making the entire development team—architects, designers, and developers—consider possible threats and vulnerabilities, security is addressed throughout the software life cycle. Many potential problems can be addressed earlier in the process, when changes are easier to make and problems less expensive to fix [3].

Part of the rationale for this pattern is that security is typically only as strong as the weakest link [5]. When developers of some system components are more concerned about security than others, the effort expended addressing security concerns in some parts of the system is often wasted. Vulnerabilities arising in less protected components many times result in system compromise. Therefore, all developers must identify the security ramifications of their code components and address their security requirements to ensure an appropriate and similar level of security throughout the system.

Finally, this pattern improves security simply by involving more people in analysis and discussion of the problem. Security often benefits just from having more eyes examining a system for vulnerabilities. Different people will bring different perspectives and make different assumptions about a system, and the process of sharing responsibility amongst a larger number of people for vulnerability analysis and threat mitigation should improve the overall security of the system.

This pattern does *not* recommend against the inclusion of security experts in a development effort, however. It is critical that certain team members have a deep understanding of relevant threats, vulnerabilities, and the solution technologies for addressing them. Security experts should be utilized as a resource by the entire development team though, and not dumped upon to solve the entire security problem.

Issues

In all but the most routine development efforts, the development of some functionality will precede the inclusion of security mechanisms. It is impossible to understand the security implications until important functional requirements are understood. But it is critical that security mechanisms be included as soon as possible.

For example, in an iterative process, it might be necessary to omit security from the first iteration, but it should absolutely be included in the second iteration. Leaving the security mechanisms until the last phase always introduces problems. One very common, specific example of this is developing an application and then attempting to deploy it behind a firewall, only to discover that critical services will not work with the firewall.

Operators, administrators, and end users must also share responsibility for security. System development should recognize this and provide the necessary education, mechanisms, and

safeguards to allow these groups to actively and effectively participate in maintaining security. (See the *Password Authentication* pattern for specific examples of this.)

Maintenance programmers are another group that must be considered as contributing to overall system security. Developers must keep the needs of future maintenance programmers in mind as they develop security-critical elements. Developers should provide meaningful comments about the security ramifications of specific code and the assumptions on which it depends.

Finally, there is a delicate balance between robust code and inefficient code. If every developer checks the same assumptions and filters the same inputs, the result will be code that is exceptionally robust but very inefficient. Simple, inexpensive checks can be repeated wherever desired, but more expensive checks should be performed once and documented as part of the overall design.

Examples

Microsoft recently announced a company-wide initiative (indicated in a memo from Bill Gates) that makes security the responsibility of everyone [2]. They have since subjected every developer in the company to a month long security training program. They have publicly acknowledged that their previous security tiger team approach was not sufficient.

The security policy for the University of Michigan references the shared responsibility for security [4].

Sharing responsibility for security must also occur at the organizational level when corporations form collaborative partnerships [1].

Trade-Offs

Accountability	This pattern can improve accountability, because it enables developers, designers, and others to understand better and improve the overall security of the application.
Availability	See Accountability.
Confidentiality	See Accountability.
Integrity	See Accountability.
Manageability	This pattern will have an effect on manageability, either positive or negative, because developers, designers, and others will explicitly assess the trade-offs between security and manageability.
Usability	See Manageability.

Performance	See Manageability.
Cost	This pattern will increase short-term costs because developers must be trained in security in order to share in the responsibility. However, it can also reduce development time and cost by eliminating excess security designed by security teams that don't understand the business decision-making process. Also, the short-term investment in security training ideally reduces long-term costs by producing a more secure system.

Related Patterns

- *Document Security Goals* – a related pattern that establishes the security policy; documenting security goals is a precursor activity to this pattern, or one that should be conducted in parallel. All those involved must understand the documented security goals of the system.

- *Red Team the Design* – a related pattern that advocates red team procedures throughout the development cycle; red teaming can be a shared responsibility, as it will foster a better understanding of the security and possibly improve the red teaming effort.

References

[1] Cale, D. and T. McGinnis. "Partners Share Responsibility for Marketplace Security". http://www.informationweek.com/813/prmarketplace.htm, November 2000.

[2] Gates, B. "Trustworthy Computing". Internal Microsoft memo (available at http://www.informationweek.com/story/IWK20020118S0093), January 2002.

[3] Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1992.

[4] University of Michigan-Dearborn Information Technology Services. "Network Security Policies and Standards". http://www.its.umd.umich.edu/policies/security.html, 2000

[5] Viega, J. and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.

Test on a Staging Server

Abstract

Web site development requires extensive testing to enable availability, protect confidentiality, and ensure integrity. While unit testing can be done on development machines, system and integration testing should take place on machines as similar to the production servers as possible. The use of a staging server enables necessary testing while preventing the outages that often occur when developers and administrators experiment with the live production system on the fly.

Problem

Web servers, their underlying operating systems, and Web applications are extremely complex. The development process for a Web site requires extensive testing in order to ensure that evolving Web applications achieve both security requirements and non-security requirements. Similarly, the maintenance process for a Web site requires extensive testing to ensure that Web applications continue to execute as desired even as the operating system and other services are patched or upgraded.

As developers implement and upgrade Web site functionality, they typically unit test on their development machines. System and integration testing cannot be done effectively on those same machines though, because the production environment must be replicated as much as possible to uncover and address all issues related to that context. However, when developers use the production servers for any amount of testing, the system could become unavailable, usability suffers, and vulnerabilities are introduced by the use of untested, possibly unstable code.

As administrators maintain a Web site, they run into similar testing issues with the production servers. System administrators must apply relevant patches to address vulnerabilities and upgrade system services in support of the Web application. The introduction of new software patches and versions sometimes breaks compatibility with existing application and system software. If system administrators are experimenting on the actual production machines, this can cause the same problems as developers using those machines for testing.

Solution

The use of a staging server enables developers and administrators to test Web site changes in a production-like environment without compromising the security of the actual production Web site servers.

The *Test on a Staging Server* pattern recommends a development, testing, and deployment procedure as follows:

- Development of a Web site occurs on the Development Machines.
- Testing of the Web site occurs on the Staging Servers.

- The initial version of the Web application is deployed on the Production Servers.
- The developers and/or administrators configure all Web site machines (e.g., Web Server, Application Server, Database Server) appropriately and consistently across the Development Machines, Staging Servers, and Production Servers.
- As development of the Web site continues on the Development Machines, testing takes place on the Staging Servers prior to update deployments on the Production Servers.

Issues

A reliable mechanism must be utilized for transferring the tested configuration from the staging servers to the production machines. Manually copying over only the files that have changed might work for small sites or very incremental updates, but that solution strategy does not scale to production sites and is not amenable to configuration control. Most production Web site building packages contain facilities for replicating or transferring entire Web site updates to machines.

Examples

[HELP]

Trade-Offs

Accountability	This pattern enhances accountability indirectly by helping prevent site compromises that could occur if testing were performed on production machines.
Availability	This pattern increases availability by helping to prevent the outages that often occur when administrators attempt to tweak the production system on the fly.
Confidentiality	See Accountability.
Integrity	See Accountability.
Manageability	This pattern enhances manageability by separating testing and production machines, thus ensuring that small changes need not be made directly to the production machines without previous testing.
Usability	This pattern improves usability by ensuring that testing is not performed on the production machines with which users interact.
Performance	No effect.

Cost	This pattern will incur additional cost due to the duplicate hardware and software required for the staging servers, though additional licenses for staging servers will generally be cheaper than licenses for production server software.

Related Patterns

- *Build the Server from the Ground Up* – a related procedure for understanding and configuring Web and application servers properly; the configurations developed in this related pattern should be tested on the staging servers of this pattern.

- *Patch Proactively* – a related procedure for administering servers; any patches applied to the system as recommended in this related pattern should be tested on the staging servers of this pattern before being applied to the production servers.

References

[1] Allen, J. *The CERT Guide to System and Network Security Practices*. Addison-Wesley, 2001.

D. Bibliography

This bibliography collects the references from all of the security patterns and supplements those references with additional informational material. The numbers in this bibliography do not map to the reference numbers used in each pattern: each pattern contains its own independent reference numbers.

[1] Abrams, M. "Security Engineering in an Evolutionary Acquisition Environment". New Security Paradigms Workshop 1998, Charlottesville, VA, September 1998.

[2] Advosys Consulting, Inc. "Writing Secure Web Applications". http://advosys.ca/papers/web-security.html, February 2002.

[3] AG Communication Systems. "Pattern Template". http://www.agcs.com/supportv2/techpapers/patterns/template.htm, 2001.

[4] Allen, J. *The CERT Guide to System and Network Security Practices*. Addison-Wesley, 2001.

[5] Anonymous. "Patches, We need those steeking patches to fix my steeking software". http://bofh.ucs.ualberta.ca/patches.html, unknown date.

[6] Balestracci, S. "PC Week Hack of 1999". http://www.sans.org/infosecFAQ/threats/PC_week.htm, February 2001.

[7] Bishop, M. "How Attackers Break Programs, and How to Write Programs Securely". 8th USENIX Security Symposium, Washington, D.C., 1999.

[8] Blum, R. *Postfix*. Sams Publishing, 2001.

[9] Bobbitt, M. "Web Security: Bulletproof". Information Security Magazine, http://www.infosecuritymag.com/2002/may/bulletproof.shtml, May 2002.

[10] Bobbitt, M. and A. Briney. "Different Approaches, Same Goal". Information Security Magazine, http://www.infosecuritymag.com/2002/may/samegoal.shtml, May 2002.

[11] Braga, A., C. Rubira, and R. Dahab. "Tropyc: A Pattern Language for Cryptographic Software". Pattern Languages of Programs 1998, Monticello, IL, 1998.

[12] Brown, F., J. DiVietri, G. Villegas, and E. Fernandez. "The Authenticator Pattern". Pattern Languages of Programs 1999, Monticello, IL, 1999.

[13] Brown, W., R Malveau, H. McCormick, and T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

[14] Brown University Computing and Information Services. "IP Address Filtering". *Web Authorization/Authentication.*

http://www.brown.edu/Facilities/CIS/ATGTest/Infrastructure/Web_Access_Control/Goals Options-ver2.html#anchor136476, October 1997.

[15] Cale, D. and T. McGinnis. "Partners Share Responsibility for Marketplace Security". http://www.informationweek.com/813/prmarketplace.htm, November 2000.

[16] Chun, M. "Authentication Mechanisms - Which is Best?" http://rr.sans.org/authentic/mechanisms.php, April 2001.

[17] Coggeshall, J. "Session Authentication". http://www.zend.com/zend/spotlight/sessionauth7may.php, May 2001.

[18] Cohen, F. and Associates. "The Deception Toolkit Home Page and Mailing List". http://all.net/dtk/dtk.html, 1998.

[19] Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology Security Evaluation Version 2.1*. http://www.commoncriteria.org/cc/cc.html, August 1999.

[20] Cooper, R. "IIS Security: 10 Steps to Better IIS Security". Information Security Magazine, http://www.infosecuritymag.com/articles/september01/features_IIS_security.shtml, August 2001.

[21] Cunningham, C. "Session Management and Authentication with PHPLIB". http://www.phpbuilder.com/columns/chad19990414.php3?page=2, April 1999.

[22] Desal, G., J. Fenner, J. Patel, and M. Schenecker. "Web Application Servers are Here To Stay". http://www.informationweek.com/726/app.htm, March 1999.

[23] Dominy, R. "Focus on JavaScript: Email Field Validation". http://javascript.about.com/library/scripts/blemailvalidate.htm, 2002.

[24] Dyck, T. "Four scripting languages speed development". http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2646052,00.html, November 2000.

[25] Fernandez, E. "Metadata and Authorization Patterns". Technical report TR-CSE-00-16, Computer Science and Engineering Department, Florida Atlantic University, 2000.

[26] Fernandez, E. and R. Pan. "A Pattern Language for Security Models". Pattern Languages of Programs 2001, Monticello, IL, 2001.

[27] Gabriel, R. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1997.

[28] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[29] Galvin, P. "The Unix Secure Programming FAQ". http://www.petergalvin.org/solsecprogfaq.html, September 1998.

[30] Garfinkel, S. and G. Spafford. *Practical UNIX and Internet Security*. O'Reilly & Associates, 1996.

[31] Gates, B. "Trustworthy Computing". Internal Microsoft memo (available at http://www.informationweek.com/story/IWK20020118S0093), January 2002.

[32] Ghosh, A. *E-commerce Security: Weak Links, Best Defenses*. Addison-Wesley, 1998.

[33] Gillespie, G. "Saving face: Get Tripwire for Web Pages to protect your site against vandalism". http://www.computeruser.com/articles/daily/7,3,1,0620,01.html, June 2001.

[34] Gong, L. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[35] INT Media Group. "Email Address Validation". http://javascript.internet.com/forms/check-email.html, 2002.

[36] Jaquith, A. "Learning from Wall Street: Risk Management for Applications". http://www.sbq.com/sbq/app_security/, Q2-2002.

[37] Hays, V., M. Loutrel, and E. Fernandez. "The Object Filter and Access Control Framework". Pattern Languages of Programs 2000, Monticello, IL, 2000.

[38] Hillside Group. "Writing Patterns and Pattern Languages". http://www.hillside.net/patterns/writing/writingpatterns.htm, 2001.

[39] Kamath, M. "Choosing a scripting language for ASP". http://www.kamath.com/columns/my3cents/mtc002_scripting.asp, October 1999.

[40] Kärkkäinen, S. "Session Management". *Unix Web Application Architectures*. http://webapparch.sourceforge.net/#23, October 2000.

[41] Kienzle, D., M. Elder, D. Tyree, and J. Edwards-Hewitt. "Security Patterns Template and Tutorial". http://www.securitypatterns.com, February 2002.

[42] King, C. "Best Practices How-Tos: Building a Secure Site". http://dcb.sun.com/practices/howtos/securesite.jsp, March 2001.

[43] Landrum, D. "Web Application and Databases Security". http://rr.sans.org/securitybasics/web_app.php, April 2001.

[44] Larson, E. "Best Practices How-Tos: How to Secure Services by Running in a Chrooted Environment". http://dcb.sun.com/practices/howtos/chrooted.jsp, November 2001.

[45] McGraw, G. and J. Viega. "Securing Software: Practice Safe Software Coding". Information Security Magazine,

http://www.infosecuritymag.com/articles/september01/features_securing.shtml, August 2001.

[46] Mendelsohn, J. "Building a Web Site: A Developer's Guide". Sun Microsystems, 2001.

[47] Microsoft Consulting Services. "Manage Security of Your Windows IIS Web Services". http://www.microsoft.com/technet/security/bestprac/MCSWebBP.asp, 2002.

[48] Mutch, A. "Mozilla Security Goals", http://tln.lib.mi.us/~amutch/pro/mozilla/secgoals.htm, April 2002.

[49] Naidu, K. "Web Application Checklist". SANS Institute SCORE (Security Consensus Operational Readiness Evaluation), http://www.sans.org/SCORE/checklists/WebApplicationChecklist.doc, 2001.

[50] NACHA Internet Council. *Understanding Internet-Initiated ACH Debits*. http://internetcouncil.nacha.org, 2002.

[51] National Computer Security Center. *DoD 5200.28-STD, Trusted Computer System Evaluation Criteria*. December 1985.

[52] National Institute of Standards and Technology (NIST) Computer Security Resource Center (CSRC). "Frequently Asked Questions (FAQ) on Information Technology Security Issues, Federal Agency Security Practices". http://csrc.nist.gov/fasp/FAQ.html, October 2001.

[53] National Institute of Standards and Technology (NIST) Information Technology Library (ITL). "Federal Information Processing Standards Publication 112: Password Usage". http://www.itl.nist.gov/fipspubs/fip112.htm, May 1985.

[54] Naval Information Systems Security Office. *Secure Windows NT Installation and Configuration Guide*. Windows NT for Navy IT-21 Version 1.5, December 2000.

[55] Nelson, R. "The qmail home page". http://www.qmail.org/top.html, June 2002.

[56] Oaks, S. *Java Security 2nd Edition*. O'Reilly & Associates, 2001.

[57] O'Dell, M. "Application Security Reporting". http://www.sbq.com/sbq/app_security/, Q2-2002.

[58] Open Web Application Security Project (OWASP). "A Guide to Building Secure Web Applications and Web Services – Draft 0.2". http://www.owasp.org, May 2002.

[59] Open Web Application Security Project (OWASP). "Input Filters". http://www.owasp.org/filters/index.shtml, May 2002.

[60] Perrine, T. "NPACI/SDSC Security Activities", http://www.edcenter.sdsu.edu/training/workshop99/june29_ppt/tep1999/sld008.htm, July 1999.

[61] Peteanu, R. "Best Practices for Secure Development – v4.03", http://members.rogers.com/razvan.peteanu/best_prac_for_sec_dev4.pdf, October 2001.

[62] Peteanu, R. "Best Practices for Secure Web Development", http://softwaredev.earthweb.com/security/print/0,,10527_640861,00.html, February 2001.

[63] Peteanu, R. "Best Practices for Secure Web Development: Technical Details", http://softwaredev.earthweb.com/security/print/0,,10527_640891,00.html, February 2001.

[64] Peteanu, R. "Design Patterns in Security". http://members.rogers.com/razvan.peteanu/designpatterns20010611.html, June 2001.

[65] Peterson, S. and D. Fisher. "The next security threat: Web applications". http://zdnet.com.com/2100-11-503341.html?legacy=zdnn, January 2001.

[66] Pountain, D. and J. Montgomery. "Web Components: Components and the Web are a match made in developer heaven". http://www.byte.com/art/9708/sec5/art1.htm, August 1997.

[67] Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1992.

[68] Ranum, M. "Intrusion Detection and Network Forensics". USENIX '99, Monterey, CA, June 1999.

[69] Rodriguez, C. "Electronic Bill Presentment and Payment (EBPP) and Open Financial Exchange (OFX) Security", http://cs1.cs.nyu.edu/rodr7076/ebpp/sld013.htm, December 1998.

[70] Romanosky, S. "Security Design Patterns". http://online.securityfocus.com/guest/9793, January 2002.

[71] Romanosky, S. "Security Design Patterns Part 1 – v1.4". http://www.romanosky.net/papers/securityDesignPatterns.html, November 2001.

[72] Rosato, R. "Best Practices for Applying Service Packs, Hotfixes and Security Patches", http://www.microsoft.com/technet/security/bestprac/bpsp.asp, 2002.

[73] Rubin, A. *White-Hat Security Arsenal: Tackling the Threats*. Addison-Wesley, 2001.

[74] Ryan, S. "A Scalable and Secure E-Commerce Hub for Electronics Recycling", http://www.ses.imse.iastate.edu/Presentation.htm, 2000.

[75] Salter, C., S. Saydjari, B. Schneier, and J. Wallner. "Toward a Secure System Engineering Methodology". New Security Paradigms Workshop 1998, Charlottesville, VA, September 1998.

[76] Sandia National Laboratories. "Information Design Assurance Red Team (IDART) Home Page." http://www.sandia.gov/idart, August 2000.

[77] SANS Institute. "Mistakes People Make that Lead to Security Breaches". http://www.sans.org/mistakes.htm, October 2001.

[78] SANS Institute. "The Twenty Most Critical Internet Security Vulnerabilities – Version 2.504". http://www.sans.org/top20.htm, May 2002.

[79] Scambray, J. "Ask Us About...Security, August 2000". http://www.microsoft.com/technet/columns/security/askus/au072400.asp, August 2000.

[80] Schneier, B. "Attack Trees". Dr. Dobb's Journal, December 1999.

[81] Schumacher, M. and U. Roedig. "Security Engineering with Patterns". Pattern Languages of Programs 2001, Monticello, IL, 2001.

[82] Security Focus SECPROG Mailing List. "Secure Programming – v1.00". http://online.securityfocus.com/popups/forums/secprog/secure-programming.shtml, 2002.

[83] Solomon, D. *Inside Windows NT Second Edition*. Microsoft Press, 1998.

[84] Stein, L. *Web Security: A Step-by-Step Reference Guide*. Addison-Wesley, 1998.

[85] Stein, L. and J. Stewart. "The World Wide Web Security FAQ – Version 3.1.2". http://www.w3.org/Security/Faq, February 2002.

[86] Strom, D. "More on ActiveX Versus Java Security: Are you safe?" http://www.webdeveloper.com/security/security_java_activex.html, 1999.

[87] Sun Microsystems. "Security Code Guidelines". http://java.sun.com/security/seccodeguide.html, February 2000.

[88] Sun Microsystems. "Secure Computing with Java: Now and the Future". http://java.sun.com/marketing/collateral/security.html, 1998.

[89] Tanenbaum, A. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.

[90] University of Michigan-Dearborn Information Technology Services. "Network Security Policies and Standards". http://www.its.umd.umich.edu/policies/security.html, 2000.

[91] U.S. Department of Energy Computer Incident Advisory Capability (CIAC). "J-042: Web Security". http://ciac.llnl.gov/ciac/bulletins/j-042.shtml, May 1999.

[92] van der Linden, P. *Just Java 2 – Fourth Edition*. Prentice Hall, 1999.

[93] Venners, B. "Java's security architecture: An overview of the JVM's security model and a look at its built-in safety features". http://www.javaworld.com/javaworld/jw-08-1997/jw-08-hood.html, August 1997.

[94] Viega, J. and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.

[95] Vlissides, J. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.

[96] Walsh, L. "Case Study: Standard Expectations". Information Security Magazine, http://www.infosecuritymag.com/2002/mar/standardexpectations.shtml, March 2002.

[97] Walsh, L. "Security Standards: Standard Practice". Information Security Magazine, http://www.infosecuritymag.com/2002/mar/iso17799.shtml, March 2002.

[98] Wheeler, D. "Secure Programming for Linux and Unix HOWTO – v2.965". http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html, May 2002.

[99] Wolter, J. "Grex Staff Notes: Security Goals". http://www.cyberspace.org/staffnote/goals.html, March 1998.

[100] Yoder, J. and J. Barcalow. "Architectural Patterns for Enabling Application Security". Pattern Languages of Programs 1997, Monticello, IL, 1998.

[101] Zwicky, E., S. Cooper, and D. Chapman. *Building Internet Firewalls 2nd Edition*. O'Reilly & Associates, 2000.