Online security management
White paper
December 2007

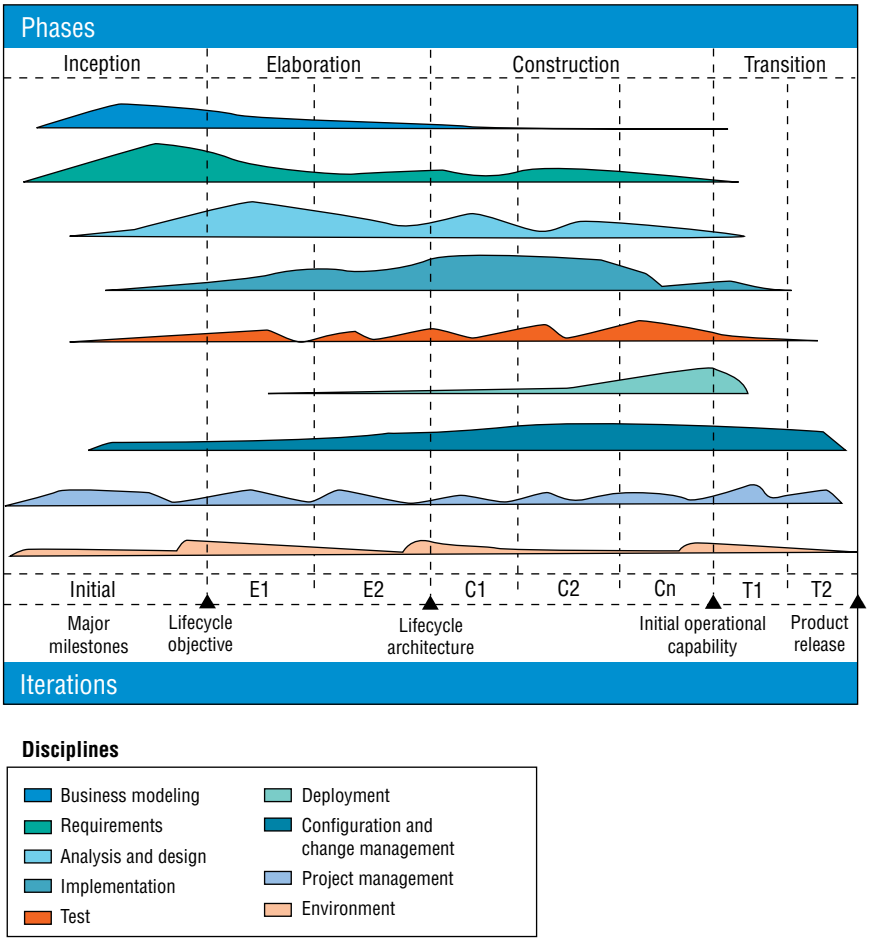
Rational. software

A layered approach to delivering security-rich Web applications.

Contents

The Web application security challenge

As businesses grow increasingly dependent upon Web applications to provide services to customers, employees and partners, these complex applications become more difficult to secure. As their code typically resides on a combination of Web servers, application servers, databases and back-end systems, potential breaches lurk in every layer.



Phases, disciplines and milestones in the IBM Rational Unified Process.

Although traditional security solutions protect Internet infrastructure layers, they do not guard against HTTP and HTML attacks. Many organizations that conduct security testing still deploy applications that allow attackers to manipulate their logic and wreak havoc on their business. To mitigate this risk, development and delivery teams must address Web application security throughout the lifecycle, addressing the many layers detailed in this paper.

Understanding the Web application lifecycle

This paper refers to lifecycle phases in the IBM Rational® Unified Process, or IBM RUP®, a widely used iterative process framework based on industry best practices (see figure). Below is a brief description of activities within each phase, which may require two or more iterations to complete.

Inception: Establish a business case, scope and operational vision, and create an initial use-case model (of how users will interact with the system), project plan, risk assessment and project description, including core requirements, security requirements (including clarification of security compliance and policies), constraints, features and prototype candidate architectures.

Elaboration: Refine the vision, baseline the architecture by addressing architecturally significant scenarios, and detail the use-case model. Create and test one or more prototypes to mitigate technical risks.

Construction: Develop detailed designs for specific components and their interactions with other applications, continuously tracking against original requirements. Generate code and test components for performance, reliability and security, continuously tracking and resolving issues. Integrate the tested components into a first release.

Transition: Deploy the application, train users and conduct beta testing to verify security and performance and to validate the application against requirements. Continuously monitor for performance, reliability and security as the application undergoes changes.

When marketplace pressures motivate organizations to push Web applications through these phases without adequate security testing, serious vulnerabilities can place the business at risk.

Software and systems development and delivery teams need to think defensively. Instead of focusing exclusively on making things easy for users, assume that some users who visit your Web site will try to manipulate your applications. One defensive approach is to test often, using automated testing and security tools like those in the IBM Rational Software Delivery Platform, to help ensure coverage and detect issues that can slip through the cracks with manual testing. In addition, IBM Rational security experts suggest a set of guidelines for building security into every layer of a Web application, which is further discussed in this paper.

Security from the eagle's view

First, a view from the top. Here is one rule to guide your thinking about Web application security overall: Never trust data that comes from a user, and never make assumptions about the limits of users' technologies. Assume that anything a user can theoretically manipulate will be manipulated in reality by one or more users. Moreover, just because a user is supposedly employing a specific technology, do not assume that it will constrain his or her actions. For example, even if a browser does not show hidden fields in a page's HTML code, assume that some users will still be able to find and manipulate those fields before sending pages back to your server.

Security at the single transaction layer

Single transactions are the basic building blocks of a Web application. Providing security measures for these transactions gives developers a safer way to create more complex entities. This section explains how to do this.

Standardize encoding

The first step in providing security for transaction processing is encoding: putting each transaction into a standard format that leaves no room for ambiguity. You need a single encoding scheme, with a single representation, for each request—preferably one that is common to all requests within the same application—to apply standardized security measures.

Protect parameter values

Hackers often change the value of parameters that a Web application sends to the server, so check all input for the maximum number of characters. Setting limits on an HTML page or using a scripting language to verify input by the client are not reliable security measures. Hackers can remove client-side tests by changing the page on their browsers or creating requests outside the browser. Even when dealing with constrained input from pull-down menus or hidden fields, it is not safe to rely on assumptions about length. Insert validation checks for parameters after the standardized encoding function has completed to avoid value changes that might result from the encoding.

Filter meta-characters such as <, >, " and & from your parameters, as hackers use them to encode attacks. If you must use them, specify which characters are allowed, and eliminate potentially dangerous sequences. Avoid free-format input, and wherever possible, make users choose specific values from a list instead. During the elaboration phase, define correct input; during the construction phase, enforce adherence to it by adding as many attributes and constraints as you can, such as maximum size and valid characters for the field, using the `CHARSET` HTML attribute.

Use obscurity for security

An effective way to protect transaction information is to keep it on your system's back end, away from the client and opportunities for easy access and manipulation. Using the HTTP `POST` method instead of the `GET` method can also help; it relays parameters within the body of a request instead of exposing them within a URL. However, this will not deter advanced hackers who typically have tools for seeing and manipulating `POST` parameters.

Use a closely related technique to improve obscurity: remove parameters from links. For those you must leave, either encrypt and sign them, or pass them as hidden parameters whose names and values you can encrypt, or sign them for protection from manipulation. Meta-information on Web pages, such as comments sent to the client, can provide clues to hackers. In addition to stripping comments from Web pages in the production environment, delete any client-side code or HTML code that includes comments.

Control dynamic pages

Dynamic sites that can place client input onto a Web page can provide users with an experience tailored to that specific profile and session. However, this can create security risks. One way to combat these security risks is to avoid using values that you receive directly from the client. Suppose you want to create a welcome message with the user's name. Be aware that the client might submit a JavaScript code within this parameter that will cause cross-site scripting; when the script runs in the user's browser, it may either copy or alter data passed between the client and the site. Instead, help ensure that the client input is script free by removing dangerous meta-characters after the standardized encoding function completes.

Protect HTTP headers

Like all other information from the client, HTTP headers are easy to manipulate and should not be used to provide security. For example, you can add any URL to the `REFERER` header, which denotes the page leading to a transaction. However, you can use this header to disqualify requests from outside the site, thereby forcing hackers to make manual changes to get around it. Whenever you use an HTTP header, it must be signed—and preferably encrypted—just as you would with cookies.

Apply standards

Although Web protocols and standards have well-defined specifications, most Web servers and Web applications allow deviations from them. Using nonstandard protocols can lead to ambiguity and ill-defined responses. Instead, it is good practice to use standard HTML and HTTP to help prevent components from misinterpreting information and misbehaving.

The following links will direct you to specifications for the most common standards and protocols:

- *HTTP/1.0: http://www.ietf.org/rfc/rfc1945.txt*
- *HTTP/1.1: http://www.ietf.org/rfc/rfc2616.txt*
- *HTML 3.2: http://www.w3.org/TR/REC-html32*
- *HTML 4.0: http://www.w3.org/TR/html4*
- *HTML 4.01: http://www.w3.org/TR/html401*

Security at the session layer

Multiple requests are organized into sessions tied to a logical entity representing a single user. To secure a session, you must first secure all its component transactions. Since Web-based protocols and standards such as HTTP and HTML are context free, the first task is to create an application-level context mechanism. Typically, sessions are initiated through an authentication process that identifies the user via a simple user name/password mechanism. The user is assigned a token that identifies him or her to the application and provides a context within which to evaluate interactions with the server. In most instances, once a session is created, all further actions are legal and do not require additional authentication.

Safeguard your authentication process

Additional defensive measures might include a delay in the authentication mechanism to slow enumeration, and a threshold for multiple failed authentication attempts on an account, including consecutive attempts and cumulative failures.

As the session creation process described above is highly vulnerable to security attacks, it is desirable to pass authentication information (user name/password) over a secure medium, such as Secure Sockets Layer (SSL), to prevent it from falling into the wrong hands. Make sure the application is free of default user names and passwords for demos and administrators, which are easy prey for attackers. And use a strong password scheme to prevent hackers from easily enumerating short or obvious passwords.

Finally, use a secondary authentication mechanism for access to crucial transactions. For example, require one set of credentials to enter an online bank and a second internal set to transfer money from an account.

Use strong session IDs

Session IDs should be cryptographically strong, signed and time stamped to protect illegal authentication bypasses. Use well-known algorithms instead of inventing new ones that may be prone to design mistakes. A strong session ID for all public areas helps slow hackers and represents a formidable obstacle to both automatic and manual attacks. Following login, ensure that all private areas of the site are associated with the session IDs. Switching to a weaker ID or counting on Internet Protocol (IP)/SSL information to maintain the user's identity places the whole session at risk.

Attach a session ID to the user whenever the application is accessed, even before the authentication process is complete. Following successful authentication, you can assign a second ID or associate the public ID with the login credentials via the back end. Never allow the client to change the ID, as this may open security vulnerabilities.

Historically, session IDs have been passed as cookies that are sent to the client and submitted back to the server with every request. However, some applications now add the ID to the URL as either a parameter or part of the path. This is not a secure practice because the ID becomes part of the `REFERER` header, which is exposed when the user accesses a different site.

Automatically terminate sessions

Automatic termination is an important aspect of session maintenance, as unattended sessions invite hackers to steal the legitimate user's identity. It is good practice to automatically terminate a session under any of the following conditions:

- *Inactive session. The session has not been active over a reasonable time, which varies among sites and applications. The typical limit is 15 to 30 minutes.*
- *Long session. The user has exceeded the maximum time allowed for a session and must either reauthenticate or start a new session. The maximum can vary according to application type, but it is typically several hours.*
- *Security error. A session should terminate automatically if any security error is found in the application.*

Enforce and protect sequencing

By default, Web-based protocols have no flow, so transactions have no inherent sequencing. However, many Web applications have implicit constraints that require sequencing. For example, applying for an account might require filling out multiple forms in a certain order that is crucial to protecting parts of the application from hackers. During the elaboration phase, it is important to define all possible flows for the site and assign every single Web page to one of them. The server should maintain that assigned flow, and, since it might be sent to the user as a cookie or a field, it should be encrypted and signed to prevent hijacking.

Security at the application layer

Web application architecture plays a major role in overall security. The simpler it is, the greater your chances for achieving good security.

Separate public and private areas

During the elaboration phase, determine which areas search engines and crawlers can access without initiating a session. Best practice is to separate these areas into individual directories or onto separate servers to avoid mixing public data with private applications. Remove cross-dependencies within the application and reduce linkages between applications. Be sure to map security considerations for those that remain.

Protect entry points

You cannot secure an application without knowing where external users might have access to it. Entry points fall into the following four main categories:

- ***User access points.*** *These are root points of entry. Increasing their number increases the design, coding and testing effort you will need to secure the application.*
- ***Search and index agent access.*** *Such agents do not maintain a session, and any page they access is an entry point. During the elaboration phase, define pages that should be indexed and regarded as public and sessionless; assign them to different areas of the site hierarchy. Use the* `robots.txt` *file to limit access for search agents or robots and prevent indexing of a confidential section.*
- ***Bookmark access.*** *Bookmarks should be public, sessionless and accessible entry points. Allow for these in your design. Typically, users bookmark session-based pages following authentication; these bookmarks should redirect to a legal entry point instead of to the requested object.*
- ***Secure entry points.*** *These typically allow business partners to access private areas of the site. Instead of treating them as simple entry points, identify them early in the elaboration phase, limit their number and sign them to minimize the risk that hackers will use them to bypass authentication mechanisms. Pass data for these entry points over SSL to prevent interception.*

In general, keep entry points to a minimum so that you can easily review them on an ongoing basis. Disable any that are unnecessary or that create high security risks.

Use encryption

Encryption is a key aspect of securing a Web application. It is possible to encrypt specific parts of a transaction, and it is necessary to encrypt complete transactions using SSL. However, the application architecture should not create dependencies on the encryption itself, because dependencies allow hackers to insert an SSL proxy between the client and the Web site. Links should be relative and should not contain the `https://` prefix. Never use IP addresses in URLs; instead, use host names to allow flexibility and the addition of secure applications (or, better yet, to keep the links relative).

Applications that include proxies and SSL accelerators can change server IP addresses. SSL protects the transport layer but is not specific to the application, so in addition to encrypting the data stream, you should encrypt specific fields in HTML forms and also mangle links (URLs) within pages to obscure the application's content and structure from probing. All fields and links used by the server should be signed—and preferably encrypted—or compared with values stored in the back end to prevent manipulation.

Limit caching

When a Web site allows proxy servers to cache information, it actually transfers part of its logic to these external servers. These servers can speed delivery of multimedia information to users, but they also pose security hazards. To protect against page changes and the loss of control over application flow, never put content pages (i.e., anything that is not multimedia) on external, unprotected cache servers.

Also, use the no-cache setting for private information. This will prevent records from remaining on the user's computer, cached for future use, after the Web page is served—exposing private data to potential manipulation.

Securing the Web application environment

No Web application can be truly secure unless the elements within its operating environment are secure as well.

Control your production server

The server running the production version of an application must always be separate from other internal servers (preferably in a demilitarized zone[DMZ], as discussed later in this paper). The production server should not run any other software that might disrupt the Web application, and it should never be used to develop code. This prevents the possibility of exposing temporary or old files by mistake that were saved for development or test purposes. To ensure that only the necessary parts of the application reside in the production environment, clean it thoroughly before every new release, and then copy the new application into the environment from an internal computer.

To prevent management application violations, do not allow the production site to be administered from outside your organization. In fact, it is best not to use remote administration even from within your organization. To be safe, administer your Web applications locally on a production computer.

Create a DMZ

A DMZ is a crucial component of an organization's periphery and network defense system, and it plays a vital part in creating a safe environment for the Web application. The DMZ separates external-facing machines from internal machines, enabling you to separate Web applications from one another, and to offer the same application to internal and external users. The application's front end can reside within the DMZ while the back end resides within the internal part of the network. External users can access the application through the DMZ, and internal users can go through an intranet or other internal application.

You can also achieve a more secure application configuration by separating the DMZ into two parts—one with public application areas and multimedia files on separate servers, and the other with private application areas that access the back-end system. This would enable you to contain damage to the public part of the site and prevent it from spreading to more critical private areas.

Securing third-party tools

Yet another level of security concern is the third-party software included in typical Web applications. Ranging from freeware to packaged applications created by large vendors, third-party tools might include Web servers, application servers and e-commerce packages.

Security for your configuration

Many vendors supply guidelines for creating a maximum-security installation, which you should follow closely. If possible, use separate servers to segregate third-party software from the rest of the application, thereby minimizing risk to your own code and private data.

Remove demos and sample applications, which hackers know how to abuse. Clean these from the production server—or do not install them in the first place. Check for other default accounts and change default passwords to prevent attackers from using publicly available user and password lists.

Stay on top of vulnerabilities and patches

Use one of the many Web sites and mailing lists that announce vulnerabilities and available patches to stay on top of third-party tool security issues. The time between public discovery of vulnerability and the hour that you can actually obtain and deploy a patch in your production environment may be dangerously long. During this time, it is crucial to monitor your site diligently for signs of intrusion or other forms of attack.

Although you may be tempted to try off-brand patches during your wait, it is usually best to hold out for a carefully tested remedy. Once the patch becomes available, be certain to apply it to all affected product installations, including new deployments.

Summary guidelines for building security-rich Web applications

We have covered a lot of territory in this paper. To help you remember which security measures you can apply at various layers of concern, print out and post the summary guidelines below.

Eagle's view

- *Never trust data that comes from a user, and never make assumptions about the limits of users' technologies.*
- *Remember that it is always easier to secure simple logic than complex logic.*

Transaction layer

- *Translate all incoming requests into a standard encoding scheme.*
- *Verify the maximum number of characters for all fields, and verify that input does not contain dangerous characters.*
- *Avoid free-format input; use as many constraints as possible.*
- *Never use values supplied directly from the client to create dynamic pages.*
- *Obscure your transaction by using the* `POST` *method instead of* `GET`.
- *Ensure that parameters passed through the client are encrypted and signed.*
- *Remove meta-information from information sent to the client.*
- *Use only standard protocols in the application.*

Session layer

- *Encrypt all authentication information and use strong password schemes to prevent hacking via enumeration.*
- *Require secondary authentication for critical parts of the application.*
- *Delete default accounts from the application.*
- *Use defensive measures to counter authentication attacks, such as a delay in the authentication mechanism to slow enumeration and a threshold for multiple failed authentication attempts.*
- *Ensure that all private areas of the application are associated with a session ID that the client or client code cannot change.*
- *Terminate sessions automatically if they become inactive or overly long, or if a security error is detected.*
- *Define all possible application flows during the elaboration phase, and ensure that each page is associated with a flow.*

Application layer

- *During the elaboration phase, identify application entry points and areas that will not be associated with session information.*
- *Minimize cross-dependencies within applications and links between applications.*
- *Encrypt the information stream to all private areas of the application; do not create links that are dependent on the stream encryption.*
- *Encrypt and sign critical information (in addition to stream encryption).*
- *Use the no-cache setting for private information.*

Application environment layer

- *Never develop code on the production server; use an internal server, and then copy the code to the production server.*
- *Minimize the code you keep on the production server, and distribute components to other servers.*
- *Never put content pages on an external cache server.*
- *Use a DMZ to separate the back and front ends of your application.*

Third-party tools layer

- *Use vendor guidelines for a maximum-security installation.*
- *Remove all demos, samples and default accounts; change default passwords.*
- *Stay up to date on vulnerabilities and patches for all tools.*
- *Wait for vendor-approved patches, and install them on all affected deployments as soon as they become available.*

For more information

To learn more about creating security-rich Web applications using IBM Rational automated lifecycle security tools such as IBM Rational AppScan, please contact your IBM representative or IBM Business Partner, or visit:

ibm.com/software/rational/offerings/testing/webapplicationsecurity

IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Produced in the United States of America
12-07

RAW14007-USEN-00