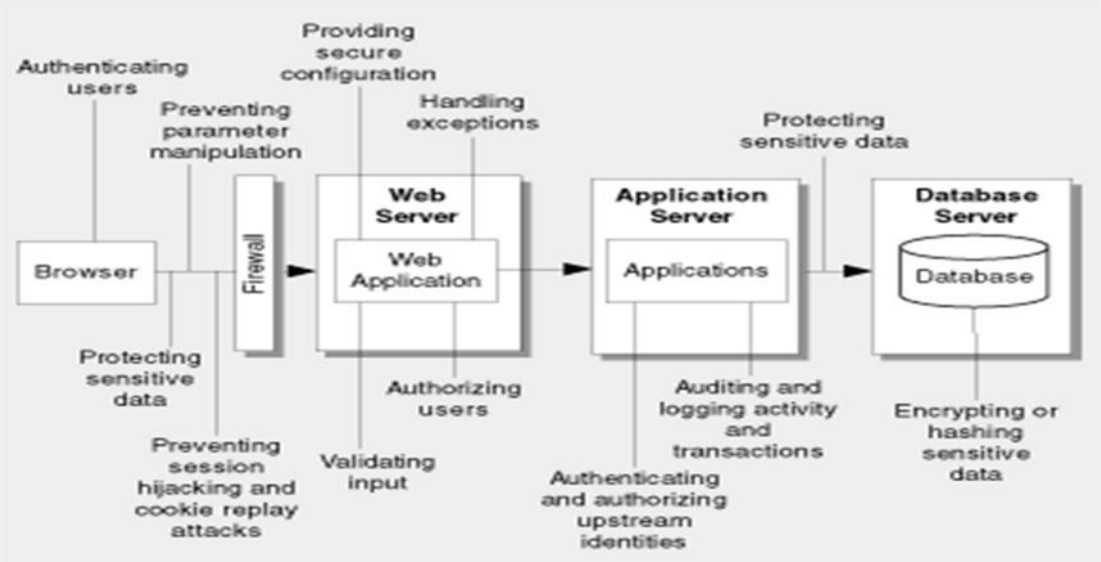
MODULE 6 SECURE ARCHITECTURE OF WEB SITES AND WEB DEPLOYMENT INFRASTRUCTURE

Overview

To build a secure Web application, you need an appropriate architecture and design. The cost and effort of retrofitting security after development are too high. An architecture and design review helps you validate the security-related design features of your application before you start the development phase. This allows you to identify and fix potential vulnerabilities before they can be exploited and before the fix requires a substantial reengineering effort.

Architecture and Design Review Process

The stateless nature of HTTP means that tracking per-user session state becomes the responsibility of the application. As a precursor to this, the application must be able to identify the user by using some form of authentication. Given that all subsequent authorization decisions are based on the user's identity, it is essential that the authentication process is secure and that the session handling mechanism used to track authenticated users is equally well protected. Designing secure authentication and session management mechanisms are just a couple of the issues facing Web application designers and developers. Other challenges occur because input and output data passes over public networks. Preventing parameter manipulation and the disclosure of sensitive data are other top issues.



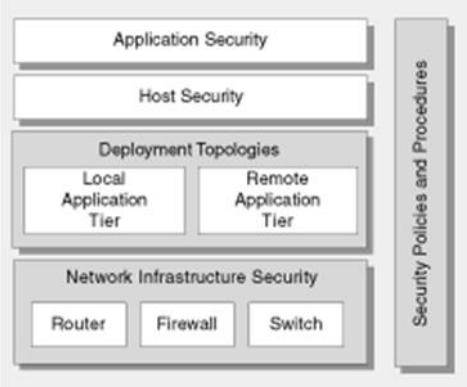
Web Application Vulnerabilities and Potential Problem Due to Bad Design

Vulnerability Category	Potential Problem Due to Bad Design
Input Validation	Attacks performed by embedding malicious strings in query strings, form fields, cookies, and HTTP headers. These include command execution, cross-site scripting (XSS), SQL injection, and buffer overflow attacks.
Authentication	Identity spoofing, password cracking, elevation of privileges, and unauthorized access.
Authorization	Access to confidential or restricted data, tampering, and execution of unauthorized operations.
Configuration Management	Unauthorized access to administration interfaces, ability to update configuration data, and unauthorized access to user accounts and account profiles.
Sensitive Data	Confidential information disclosure and data tampering.
Session Management	Capture of session identifiers resulting in session hijacking and identity spoofing.
Cryptography	Access to confidential data or account credentials, or both.
Parameter Manipulation	Path traversal attacks, command execution, and bypass of access control mechanisms among others, leading to information disclosure, elevation of privileges, and denial of service.
Exception Management	Denial of service and disclosure of sensitive system level details.
Auditing and Logging	Failure to spot the signs of intrusion, inability to prove a user's actions, and difficulties in problem diagnosis.

Deployment Considerations –

During the application design phase, you should review your corporate security policies and procedures together with the infrastructure your application is to be deployed on. Frequently, the target environment is rigid, and your application design must reflect the restrictions. Sometimes design tradeoffs are required, for example, because of protocol or port restrictions, or specific deployment topologies. Identify constraints early in the design phase to avoid surprises later and involve members of the network and infrastructure teams to help with this process.

The below figure shows you what are all the different aspects for the deployment considerations



1. Security Policies and Procedures
   Security policy determines what your applications are allowed to do and what the users of the application are permitted to do. More importantly, they define restrictions to determine what applications and users are not allowed to do. Identify and work within the framework defined by your corporate security policy while designing your applications to make sure you do not breach policy that might prevent the application being deployed.
2. Network Infrastructure Components
   The provided network understanding is very crucial, and you need to align the security requirements for the targeted system which are in terms of filtering rules, port restriction, and supported protocols and so on.
   Identify how firewalls and firewall policies are likely to affect your application's design and deployment. There may be firewalls to separate the Internet-facing applications from the internal network. There may be additional firewalls in front of the database. These can affect your possible communication ports and, therefore, authentication options from the Web server to remote application and database servers. For example, Windows authentication requires additional ports.
   At the design stage, consider what protocols, ports, and services are allowed to access internal resources from the Web servers in the perimeter network. Also identify the

protocols and ports that the application design requires and analyze the potential threats that occur from opening new ports or using new protocols.

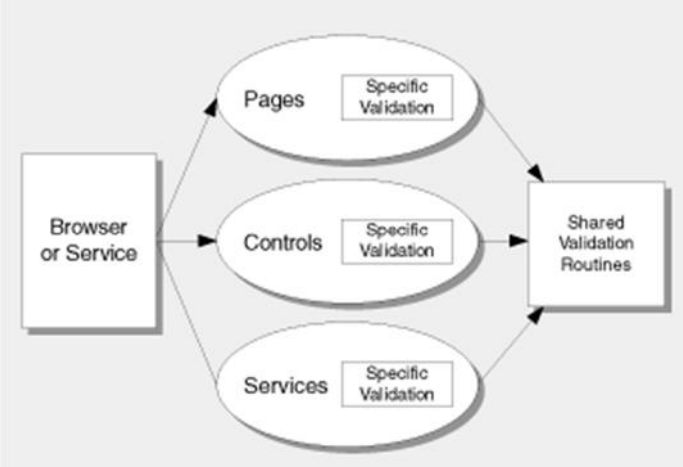
3. Deployment Topologies
   Your application's deployment topology and whether you have a remote application tier is a key consideration that must be incorporated in your design. If you have a remote application tier, you need to consider how to secure the network between servers to address the network eavesdropping threat and to provide privacy and integrity for sensitive data.
   Also consider identity flow and identify the accounts that will be used for network authentication when your application connects to remote servers. A common approach is to use a least privileged process account and create a duplicate (mirrored) account on the remote server with the same password. Alternatively, you might use a domain process account, which provides easier administration but is more problematic to secure because of the difficulty of limiting the account's use throughout the network. An intervening firewall or separate domains without trust relationships often makes the local account approach the only viable option.

Input Validation –

Input validation is a challenging issue and the primary burden of a solution falls on application developers. However, proper input validation is one of your strongest measures of defense against today's application attacks. Proper input validation is an effective countermeasure that can help prevent XSS, SQL injection, buffer overflows, and other input attacks.

The following practices improve your Web application's input validation:

- **Assume all input is malicious**.
  Input validation starts with a fundamental supposition that all input is malicious until proven otherwise. Whether input comes from a service, a file share, a user, or a database, validate your input if the source is outside your trust boundary.
- **Centralize your approach**.
  Make your input validation strategy a core element of your application design. Consider a centralized approach to validation, for example, by using common validation and filtering code in shared libraries. This ensures that validation rules are applied consistently. It also reduces development effort and helps with future maintenance.
  In many cases, individual fields require specific validation, for example, with specifically developed regular expressions. However, you can frequently factor out common routines to validate regularly used fields such as e-mail addresses, titles, names, postal addresses including ZIP or postal codes, and so on. The below figure shows the centralized approach to the input validation:

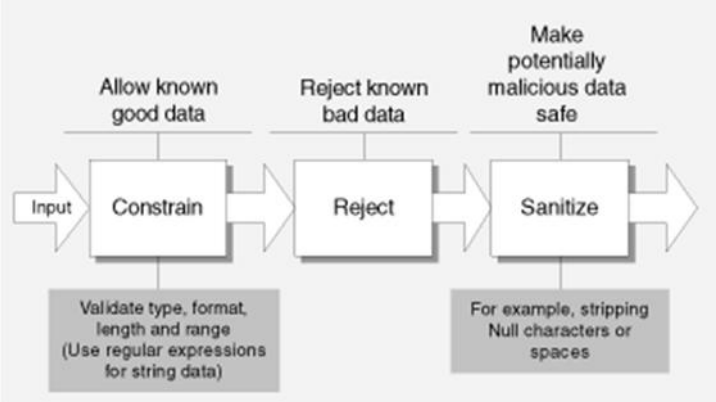


- **Do not rely on client-side validation**.
  Server-side code should perform its own validation. What if an attacker bypasses your client, or shuts off your client-side script routines, for example, by disabling JavaScript? Use client-side validation to help reduce the number of round trips to the server but do not rely on it for security. This is an example of defense in depth.
- **Be careful with canonicalization issues**.
  Data in canonical form is in its most standard or simplest form. Canonicalization is the process of converting data to its canonical form. File paths and URLs are particularly prone to canonicalization issues and many well-known exploits are a direct result of canonicalization bugs. For example, consider the following string that contains a file and path in its canonical form.
  c:\temp\somefile.dat
  The following strings could also represent the same file.
  somefile.dat
  c:\temp\subdir\..\somefile.dat
  c:\ temp\ somefile.dat
  ..\somefile.dat
  c%3A%5Ctemp%5Csubdir%5C%2E%2E%5Csomefile.dat
  In the last example, characters have been specified in hexadecimal form:
    - %3A is the colon character.
    - %5C is the backslash character.
    - %2E is the dot character.

  You should generally try to avoid designing applications that accept input file names from the user to avoid canonicalization issues. Consider alternative designs instead. For example, let the application determine the file name for the user.
  If you do need to accept input file names, make sure they are strictly formed before making security decisions such as granting or denying access to the specified file.

- **Constrain, reject, and sanitize your input**

The preferred approach to validating input is to constrain what you allow from the beginning. It is much easier to validate data for known valid types, patterns, and ranges than it is to validate data by looking for known bad characters. When you design your application, you know what your application expects. The range of valid data is generally a more finite set than potentially malicious input. However, for defense in depth you may also want to reject known bad input and then sanitize the input.



Authentication –

Authentication is the process of determining caller identity. There are three aspects to consider:

- Identify where authentication is required in your application. It is generally required whenever a trust boundary is crossed. Trust boundaries usually include assemblies, processes, and hosts.
- Validate who the caller is. Users typically authenticate themselves with user names and passwords.
- Identify the user on subsequent requests. This requires some form of authentication token.

Many Web applications use a password mechanism to authenticate users, where the user supplies a user name and password in an HTML form. The issues and questions to consider here include:

- **Are user names and passwords sent in plaintext over an insecure channel?** If so, an attacker can eavesdrop with network monitoring software to capture the credentials. The countermeasure here is to secure the communication channel by using Secure Socket Layer (SSL).
- **How are the credentials stored?** If you are storing user names and passwords in plaintext, either in files or in a database, you are inviting trouble. What if your application directory is improperly configured and an attacker browses to the file and downloads its contents or

adds a new privileged logon account? What if a disgruntled administrator takes your database of user names and passwords?

- **How are the credentials verified?** There is no need to store user passwords if the sole purpose is to verify that the user knows the password value. Instead, you can store a verifier in the form of a hash value and re-compute the hash using the user-supplied value during the logon process. To mitigate the threat of dictionary attacks against the credential store, use strong passwords and combine a randomly generated salt value with the password hash.
- **How is the authenticated user identified after the initial logon?** Some form of authentication ticket, for example an authentication cookie, is required. How is the cookie secured? If it is sent across an insecure channel, an attacker can capture the cookie and use it to access the application. A stolen authentication cookie is a stolen logon.

The following practices improve your Web application's authentication:

- **Separate public and restricted areas**.

  A public area of your site can be accessed by any user anonymously. Restricted areas can be accessed only by specific individuals and the users must authenticate with the site. Consider a typical retail Web site. You can browse the product catalog anonymously. When you add items to a shopping cart, the application identifies you with a session identifier. Finally, when you place an order, you perform a secure transaction. This requires you to log in to authenticate your transaction over SSL.

  By partitioning your site into public and restricted access areas, you can apply separate authentication and authorization rules across the site and limit the use of SSL. To avoid the unnecessary performance overhead associated with SSL, design your site to limit the use of SSL to the areas that require authenticated access.

- **Use account lockout policies for end-user accounts**.

  Disable end-user accounts or write events to a log after a set number of failed logon attempts. If you are using Windows authentication, such as NTLM or the Kerberos protocol, these policies can be configured and applied automatically by the operating system. With Forms authentication, these policies are the responsibility of the application and must be incorporated into the application design.

  Be careful that account lockout policies cannot be abused in denial of service attacks. For example, well known default service accounts such as IUSR_MACHINENAME should be replaced by custom account names to prevent an attacker who obtains the Internet Information Services (IIS) Web server name from locking out this critical account.

- **Support password expiration periods**.

Passwords should not be static and should be changed as part of routine password maintenance through password expiration periods. Consider providing this type of facility during application design.

- **Be able to disable accounts**.

  If the system is compromised, being able to deliberately invalidate credentials or disable accounts can prevent additional attacks.

- **Do not store passwords in user stores**.

  If you must verify passwords, it is not necessary to actually store the passwords. Instead, store a one way hash value and then re-compute the hash using the user-supplied passwords. To mitigate the threat of dictionary attacks against the user store, use strong passwords and incorporate a random salt value with the password.

- **Require strong passwords**.

  Do not make it easy for attackers to crack passwords. There are many guidelines available, but a general practice is to require a minimum of eight characters and a mixture of uppercase and lowercase characters, numbers, and special characters. Whether you are using the platform to enforce these for you, or you are developing your own validation, this step is necessary to counter brute-force attacks where an attacker tries to crack a password through systematic trial and error. Use regular expressions to help with strong password validation.

- **Do not send passwords over the wire in plaintext**.

  Plaintext passwords sent over a network are vulnerable to eavesdropping. To address this threat, secure the communication channel, for example, by using SSL to encrypt the traffic.

- **Protect authentication cookies**.

  A stolen authentication cookie is a stolen logon. Protect authentication tickets using encryption and secure communication channels. Also limit the time interval in which an authentication ticket remains valid, to counter the spoofing threat that can result from replay attacks, where an attacker captures the cookie and uses it to gain illicit access to your site. Reducing the cookie timeout does not prevent replay attacks but it does limit the amount of time the attacker has to access the site using the stolen cookie.

Authorization

Authorization determines what the authenticated identity can do and the resources that can be accessed. Improper or weak authorization leads to information disclosure and data tampering. Defense in depth is the key security principle to apply to your application's authorization strategy.

The following practices improve your Web application's authorization:

- **Use Multiple Gatekeepers**

  On the server side, you can use IP Security Protocol (IPSec) policies to provide host restrictions to restrict server-to-server communication. For example, an IPSec policy might restrict any host apart from a nominated Web server from connecting to a database server. IIS provides Web permissions and Internet Protocol/ Domain Name System (IP/DNS) restrictions. IIS Web permissions apply to all resources requested over HTTP regardless of the user. They do not provide protection if an attacker manages to log on to the server. For this, NTFS permissions allow you to specify per user access control lists. Finally, ASP.NET provides URL authorization and File authorization together with principal permission demands. By combining these gatekeepers you can develop an effective authorization strategy.

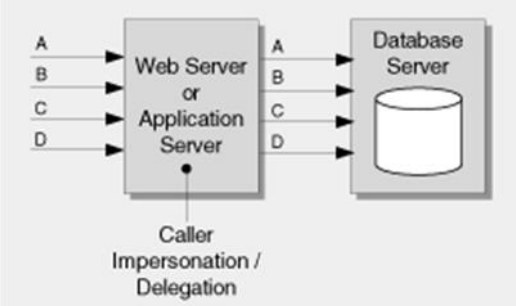
- **Restrict User Access to System Level Resources**

  System level resources include files, folders, registry keys, Active Directory objects, database objects, event logs, and so on. Use Windows Access Control Lists (ACLs) to restrict which users can access what resources and the types of operations that they can perform. Pay particular attention to anonymous Internet user accounts; lock these down with ACLs on resources that explicitly deny access to anonymous users.

- **Consider Authorization Granularity**

  There are three common authorization models, each with varying degrees of granularity and scalability.

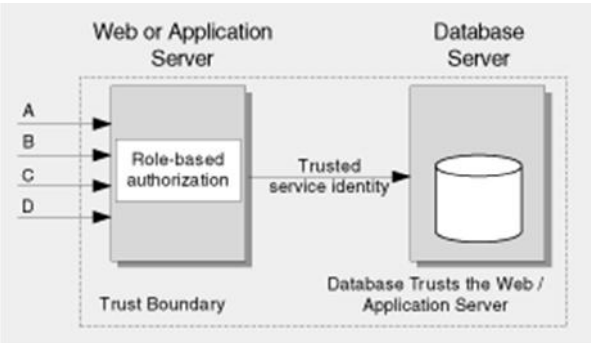
  The most granular approach relies on impersonation. Resource access occurs using the security context of the caller. Windows ACLs on the secured resources (typically files or tables, or both) determine whether the caller is allowed to access the resource. If your application provides access primarily to user specific resources, this approach may be valid. It has the added advantage that operating system level auditing can be performed across the tiers of your application, because the original caller's security context flows at the operating system level and is used for resource access. However, the approach suffers from poor application scalability because effective connection pooling for database access is not possible. As a result, this approach is most frequently found in limited scale intranet-based applications. The impersonation model is shown in Figure



The least granular but most scalable approach uses the application's process identity for resource access. This approach supports database connection pooling but it means that the permissions granted to the application's identity in the database are common, irrespective of the identity of the original caller. The primary authorization is performed in the application's logical middle tier using roles, which group together users who share the same privileges in the application. Access to classes and methods is restricted based on the role membership of the caller. To support the retrieval of per user data, a common approach is to include an identity column in the database tables and use query parameters to restrict the retrieved data. For example, you may pass the original caller's identity to the database at the application (not operating system) level through stored procedure parameters, and write queries similar to the following:

SELECT field1, field2, field3 FROM Table1 WHERE {some search criteria} AND UserName = @originalCallerUserName

This model is referred to as the trusted subsystem or sometimes as the trusted server model. It is shown in Figure



Configuration Management

Carefully consider your Web application's configuration management functionality. Most applications require interfaces that allow content developers, operators, and administrators to configure the application and manage items such as Web page content, user accounts, user profile information, and database connection strings. If remote administration is supported, how are the administration interfaces secured? The consequences of a security breach to an

administration interface can be severe, because the attacker frequently ends up running with administrator privileges and has direct access to the entire site.

The following practices improve the security of your Web application's configuration management:

- **Secure Your Administration Interfaces**

  It is important that configuration management functionality is accessible only by authorized operators and administrators. A key part is to enforce strong authentication over your administration interfaces, for example, by using certificates.

  If possible, limit or avoid the use of remote administration and require administrators to log on locally. If you need to support remote administration, use encrypted channels, for example, with SSL or VPN technology, because of the sensitive nature of the data passed over administrative interfaces. Also consider limiting remote administration to computers on the internal network by using IPSec policies, to further reduce risk.

- **Secure Your Configuration Stores**

  Text-based configuration files, the registry, and databases are common options for storing application configuration data. If possible, avoid using configuration files in the application's Web space to prevent possible server configuration vulnerabilities resulting in the download of configuration files. Whatever approach you use, secure access to the configuration store, for example, by using Windows ACLs or database permissions. Also avoid storing plaintext secrets such as database connection strings or account credentials. Secure these items using encryption and then restrict access to the registry key, file, or table that contains the encrypted data.

- **Separate Administration Privileges**

  If the functionality supported by the features of your application's configuration management varies based on the role of the administrator, consider authorizing each role separately by using role-based authorization. For example, the person responsible for updating a site's static content should not necessarily be allowed to change a customer's credit limit.

- **Use Least Privileged Process and Service Accounts**

  An important aspect of your application's configuration is the process accounts used to run the Web server process and the service accounts used to access downstream resources and systems. Make sure these accounts are set up as least privileged. If an attacker manages to take control of a process, the process identity should have very restricted access to the file system and other system resources to limit the damage that can be done.

- **Sensitive Data**

Applications that deal with private user information such as credit card numbers, addresses, medical records, and so on should take special steps to make sure that the data remains private and unaltered. In addition, secrets used by the application's implementation, such as passwords and database connection strings, must be secured. The security of sensitive data is an issue while the data is stored in persistent storage and while it is passed across the network.

- **Secrets**

Secrets include passwords, database connection strings, and credit card numbers. The following practices improve the security of your Web application's handling of secrets:

- **Do Not Store Secrets if You Can Avoid It**

Storing secrets in software in a completely secure fashion is not possible. An administrator, who has physical access to the server, can access the data. For example, it is not necessary to store a secret when all you need to do is verify whether a user knows the secret. In this case, you can store a hash value that represents the secret and compute the hash using the user-supplied value to verify whether the user knows the secret.

- **Do Not Store Secrets in Code**

Do not hard code secrets in code. Even if the source code is not exposed on the Web server, it is possible to extract string constants from compiled executable files. A configuration vulnerability may allow an attacker to retrieve the executable.

- **Do Not Store Database Connections, Passwords, or Keys in Plaintext**

Avoid storing secrets such as database connection strings, passwords, and keys in plaintext. Use encryption and store encrypted strings.

- **Avoid Storing Secrets in the LSA**

Avoid the LSA because your application requires administration privileges to access it. This violates the core security principle of running with least privilege. Also, the LSA can store secrets in only a restricted number of slots. A better approach is to use DPAPI, available on Microsoft Windows® 2000 and later operating systems.

- **Use DPAPI for Encrypting Secrets**

To store secrets such as database connection strings or service account credentials, use DPAPI. The main advantage to using DPAPI is that the platform system manages the encryption/decryption key and it is not an issue for the application. The key is either tied to a Windows user account or to a specific computer, depending on flags passed to the DPAPI functions.

DPAPI is best suited for encrypting information that can be manually recreated when the master keys are lost, for example, because a damaged server requires an operating system re-install. Data that cannot be recovered because you do not know the plaintext value, for example, customer credit card details, require an alternate approach that uses traditional symmetric key-based cryptography such as the use of triple-DES.

- **Sensitive per User Data**

Sensitive data such as logon credentials and application level data such as credit card numbers, bank account numbers, and so on, must be protected. Privacy through encryption and integrity through message authentication codes (MAC) are the key elements.

The following practices improve your Web application's security of sensitive per user data:

- **Retrieve sensitive data on demand**.
- **Encrypt the data or secure the communication channel**.
- **Do not store sensitive data in persistent cookies**.
- **Do not pass sensitive data using the HTTP-GET protocol**.
- **Retrieve Sensitive Data on Demand**

The preferred approach is to retrieve sensitive data on demand when it is needed instead of persisting or caching it in memory. For example, retrieve the encrypted secret when it is needed, decrypt it, use it, and then clear the memory (variable) used to hold the plaintext secret. If performance becomes an issue, consider the following options:

- **Cache the encrypted secret**.
- **Cache the plaintext secret**.
- **Cache the Encrypted Secret**

Retrieve the secret when the application loads and then cache the encrypted secret in memory, decrypting it when the application uses it. Clear the plaintext copy when it is no longer needed. This approach avoids accessing the data store on a per request basis.

- **Cache the Plaintext Secret**

Avoid the overhead of decrypting the secret multiple times and store a plaintext copy of the secret in memory. This is the least secure approach but offers the optimum performance. Benchmark the other approaches before guessing that the additional performance gain is worth the added security risk.

- **Encrypt the Data or Secure the Communication Channel**

  If you are sending sensitive data over the network to the client, encrypt the data or secure the channel. A common practice is to use SSL between the client and Web server. Between servers, an increasingly common approach is to use IPSec. For securing sensitive data that flows through several intermediaries, for example, Web service Simple Object Access Protocol (SOAP) messages, use message level encryption.

- **Do Not Store Sensitive Data in Persistent Cookies**

  Avoid storing sensitive data in persistent cookies. If you store plaintext data, the end user is able to see and modify the data. If you encrypt the data, key management can be a problem. For example, if the key used to encrypt the data in the cookie has expired and been recycled, the new key cannot decrypt the persistent cookie passed by the browser from the client.

- **Do Not Pass Sensitive Data Using the HTTP-GET Protocol**

  You should avoid storing sensitive data using the HTTP-GET protocol because the protocol uses query strings to pass data. Sensitive data cannot be secured using query strings and query strings are often logged by the server.

Session Management

Web applications are built on the stateless HTTP protocol, so session management is an application-level responsibility. Session security is critical to the overall security of an application.

The following practices improve the security of your Web application's session management:

- **Use SSL to protect session authentication cookies**.
- **Encrypt the contents of the authentication cookies**.
- **Limit session lifetime**.
- **Protect session state from unauthorized access**.
- **Use SSL to Protect Session Authentication Cookies**

  Do not pass authentication cookies over HTTP connections. Set the secure cookie property within authentication cookies, which instructs browsers to send cookies back to the server only over HTTPS connections. For more information, see Chapter 10, "Building Secure ASP.NET Web Pages and Controls."

- **Encrypt the Contents of the Authentication Cookies**

Encrypt the cookie contents even if you are using SSL. This prevents an attacker viewing or modifying the cookie if he manages to steal it through an XSS attack. In this event, the attacker could still use the cookie to access your application, but only while the cookie remains valid.

- **Limit Session Lifetime**

Reduce the lifetime of sessions to mitigate the risk of session hijacking and replay attacks. The shorter the session, the less time an attacker has to capture a session cookie and use it to access your application.

- **Protect Session State from Unauthorized Access**

Consider how session state is to be stored. For optimum performance, you can store session state in the Web application's process address space. However, this approach has limited scalability and implications in Web farm scenarios, where requests from the same user cannot be guaranteed to be handled by the same server. In this scenario, an out-of-process state store on a dedicated state server or a persistent state store in a shared database is required. ASP.NET supports all three options.

You should secure the network link from the Web application to state store using IPSec or SSL to mitigate the risk of eavesdropping. Also consider how the Web application is to be authenticated by the state store. Use Windows authentication where possible to avoid passing plaintext authentication credentials across the network and to benefit from secure Windows account policies.

Cryptography

Cryptography in its fundamental form provides the following:

- **Privacy** (Confidentiality). This service keeps a secret confidential.
- **Non-Repudiation** (Authenticity). This service makes sure a user cannot deny sending a particular message.
- **Tamperproofing** (Integrity). This service prevents data from being altered.
- **Authentication**. This service confirms the identity of the sender of a message.

Web applications frequently use cryptography to secure data in persistent stores or as it is transmitted across networks. The following practices improve your Web application's security when you use cryptography:

- **Do not develop your own cryptography**.
- **Keep unencrypted data close to the algorithm**.

- **Use the correct algorithm and correct key size**.
- **Secure your encryption keys**.
- **Do Not Develop Your Own Cryptography**

Cryptographic algorithms and routines are notoriously difficult to develop successfully. As a result, you should use the tried and tested cryptographic services provided by the platform. This includes the .NET Framework and the underlying operating system. Do not develop custom implementations because these frequently result in weak protection.

- **Keep Unencrypted Data Close to the Algorithm**

When passing plaintext to an algorithm, do not obtain the data until you are ready to use it, and store it in as few variables as possible.

- **Use the Correct Algorithm and Correct Key Size**

It is important to make sure you choose the right algorithm for the right job and to make sure you use a key size that provides a sufficient degree of security. Larger key sizes generally increase security. The following list summarizes the major algorithms together with the key sizes that each uses:

- Data Encryption Standard (DES) 64-bit key (8 bytes)
- TripleDES 128-bit key or 192-bit key (16 or 24 bytes)
- Rijndael 128–256 bit keys (16–32 bytes)
- RSA 384–16,384 bit keys (48–2,048 bytes)

For large data encryption, use the TripleDES symmetric encryption algorithm. For slower and stronger encryption of large data, use Rijndael. To encrypt data that is to be stored for short periods of time, you can consider using a faster but weaker algorithm such as DES. For digital signatures, use Rivest, Shamir, and Adleman (RSA) or Digital Signature Algorithm (DSA). For hashing, use the Secure Hash Algorithm (SHA)1.0. For keyed hashes, use the Hash-based Message Authentication Code (HMAC) SHA1.0.

- **Secure Your Encryption Keys**

An encryption key is a secret number used as input to the encryption and decryption processes. For encrypted data to remain secure, the key must be protected. If an attacker compromises the decryption key, your encrypted data is no longer secure.

The following practices help secure your encryption keys:

- **Use DPAPI to avoid key management**.
- **Cycle your keys periodically**.
- **Use DPAPI to Avoid Key Management**

As mentioned previously, one of the major advantages of using DPAPI is that the key management issue is handled by the operating system. The key that DPAPI uses is derived from the password that is associated with the process account that calls the DPAPI functions. Use DPAPI to pass the burden of key management to the operating system.

- **Cycle Your Keys Periodically**

Generally, a static secret is more likely to be discovered over time. Questions to keep in mind are: Did you write it down somewhere? Did Bob, the administrator with the secrets, change positions in your company or leave the company? Do not overuse keys.

- **Parameter Manipulation**

With parameter manipulation attacks, the attacker modifies the data sent between the client and Web application. This may be data sent using query strings, form fields, cookies, or in HTTP headers. The following practices help secure your Web application's parameter manipulation:

- **Encrypt sensitive cookie state**.
- **Make sure that users do not bypass your checks**.
- **Validate all values sent from the client**.
- **Do not trust HTTP header information**.

Encrypt Sensitive Cookie State

Cookies may contain sensitive data such as session identifiers or data that is used as part of the server-side authorization process. To protect this type of data from unauthorized manipulation, use cryptography to encrypt the contents of the cookie.

- **Make Sure that Users Do Not Bypass Your Checks**

Make sure that users do not bypass your checks by manipulating parameters. URL parameters can be manipulated by end users through the browser address text box. For example, the URL http://www.<*YourSite*>/<*YourApp*>/sessionId=10 has a value of 10 that can be changed to some random number to receive different output. Make sure that you check this in server-side code, not in client-side JavaScript, which can be disabled in the browser.

- **Validate All Values Sent from the Client**

  Restrict the fields that the user can enter and modify and validate all values coming from the client. If you have predefined values in your form fields, users can change them and post them back to receive different results. Permit only known good values wherever possible. For example, if the input field is for a state, only inputs matching a state postal code should be permitted.

- **Do Not Trust HTTP Header Information**

  HTTP headers are sent at the start of HTTP requests and HTTP responses. Your Web application should make sure it does not base any security decision on information in the HTTP headers because it is easy for an attacker to manipulate the header. For example, the **referer** field in the header contains the URL of the Web page from where the request originated. Do not make any security decisions based on the value of the referer field, for example, to check whether the request originated from a page generated by the Web application, because the field is easily falsified.

- **Exception Management**

  Secure exception handling can help prevent certain application-level denial of service attacks and it can also be used to prevent valuable system-level information useful to attackers from being returned to the client. For example, without proper exception handling, information such as database schema details, operating system versions, stack traces, file names and path information, SQL query strings and other information of value to an attacker can be returned to the client.

A good approach is to design a centralized exception management and logging solution and consider providing hooks into your exception management system to support instrumentation and centralized monitoring to help system administrators.

The following practices help secure your Web application's exception management:

- **Do not leak information to the client**.
- **Log detailed error messages**.
- **Catch exceptions**.
- **Do Not Leak Information to the Client**

In the event of a failure, do not expose information that could lead to information disclosure. For example, do not expose stack trace details that include function names and line numbers in the case of debug builds (which should not be used on production servers). Instead, return generic error messages to the client.

- **Log Detailed Error Messages**

Send detailed error messages to the error log. Send minimal information to the consumer of your service or application, such as a generic error message and custom error log ID that can subsequently be mapped to detailed message in the event logs. Make sure that you do not log passwords or other sensitive data.

- **Catch Exceptions**

Use structured exception handling and catch exception conditions. Doing so avoids leaving your application in an inconsistent state that may lead to information disclosure. It also helps protect your application from denial of service attacks. Decide how to propagate exceptions internally in your application and give special consideration to what occurs at the application boundary.

- **Auditing and Logging**

You should audit and log activity across the tiers of your application. Using logs, you can detect suspicious-looking activity. This frequently provides early indications of a full-blown attack and the logs help address the repudiation threat where users deny their actions. Log files may be required in legal proceedings to prove the wrongdoing of individuals. Generally, auditing is considered most authoritative if the audits are generated at the precise time of resource access and by the same routines that access the resource.

The following practices improve your Web application's security:

- **Audit and log access across application tiers**.
- **Consider identity flow**.
- **Log key events**.
- **Secure log files**.
- **Back up and analyze log files regularly**.

Audit and Log Access Across Application Tiers

Audit and log access across the tiers of your application for non-repudiation. Use a combination of application-level logging and platform auditing features, such as Windows, IIS, and SQL Server auditing.

- **Consider Identity Flow**

Consider how your application will flow caller identity across multiple application tiers. You have two basic choices. You can flow the caller's identity at the operating system level using

the Kerberos protocol delegation. This allows you to use operating system level auditing. The drawback with this approach is that it affects scalability because it means there can be no effective database connection pooling at the middle tier. Alternatively, you can flow the caller's identity at the application level and use trusted identities to access back-end resources. With this approach, you have to trust the middle tier and there is a potential repudiation risk. You should generate audit trails in the middle tier that can be correlated with back-end audit trails. For this, you must make sure that the server clocks are synchronized, although Microsoft Windows 2000 and Active Directory do this for you.

- **Log Key Events**

The types of events that should be logged include successful and failed logon attempts, modification of data, and retrieval of data, network communications, and administrative functions such as the enabling or disabling of logging. Logs should include the time of the event, the location of the event including the machine name, the identity of the current user, the identity of the process initiating the event, and a detailed description of the event.

- **Secure Log Files**

Secure log files using Windows ACLs and restrict access to the log files. This makes it more difficult for attackers to tamper with log files to cover their tracks. Minimize the number of individuals who can manipulate the log files. Authorize access only to highly trusted accounts such as administrators.

- **Back Up and Analyze Log Files Regularly**

There's no point in logging activity if the log files are never analyzed. Log files should be removed from production servers on a regular basis. The frequency of removal is dependent upon your application's level of activity. Your design should consider the way that log files will be retrieved and moved to offline servers for analysis. Any additional protocols and ports opened on the Web server for this purpose must be securely locked down.