Designing Secure Web Application

Prof. Dr. M. Ameer Ali

Professor & Chairman

Department of Computer Science & Engineering

Bangladesh University of Business and Technology (BUBT)

Overview

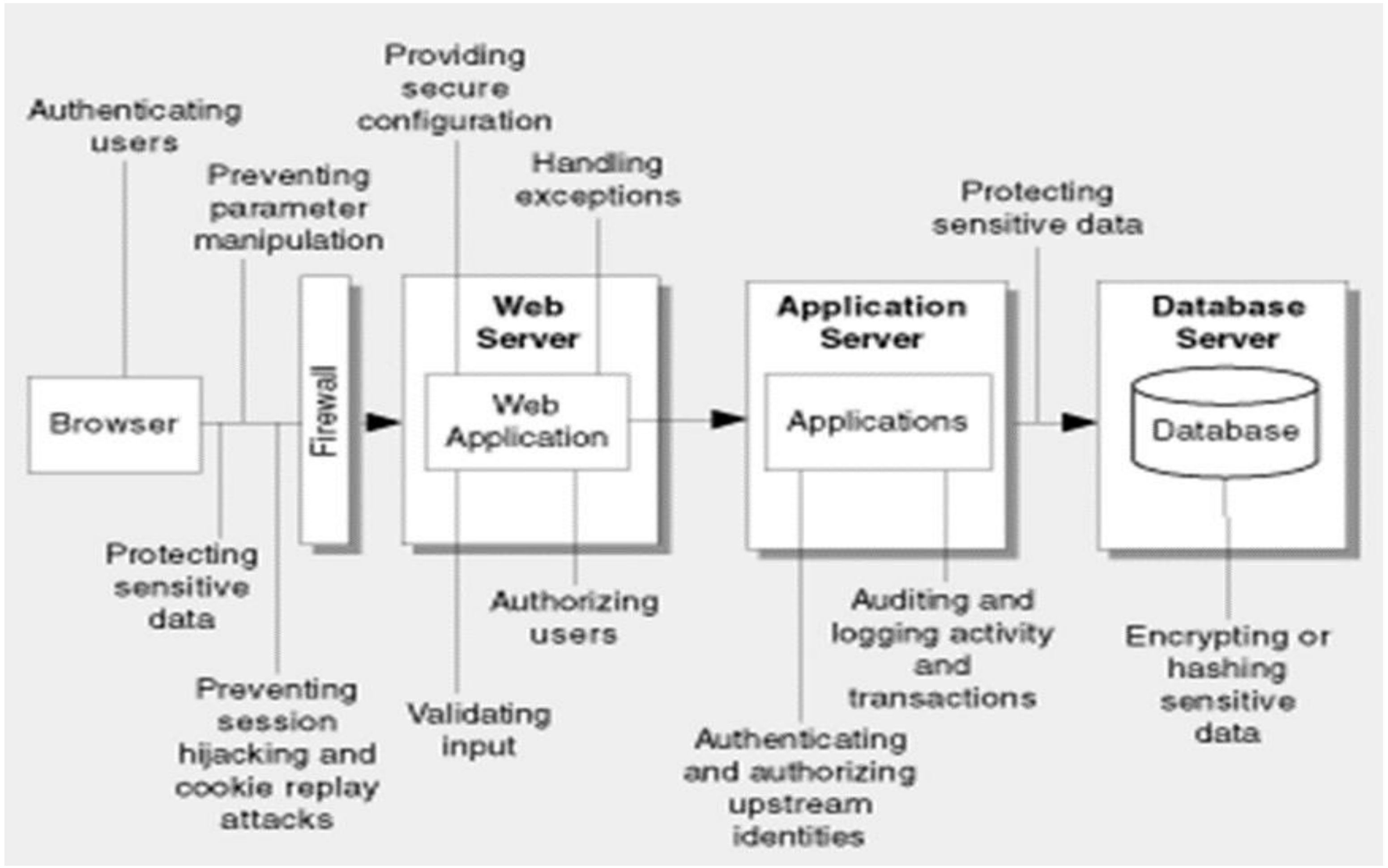
- Web applications present a complex set of security issues for architects, designers, and developers.
- The most secure and hack-resilient web applications are those that have been built from the ground up with security in mind (Applying sound architectural and design practices, incorporate deployment considerations and corporate security policies)
- Presenting a set of secure architecture and design guidelines organized by common application vulnerability category.

European Union

Architecture and Design Issues for Web Applications

- Web applications present designers and developers with many challenges:
    - The stateless nature of HTTP (i.e. tracking per-user session state becomes the responsibility of the application);
    - Application must be able to identify the user by using some form of authentication;
    - Ensure authentication process is secure;
    - Session handling mechanism for track authenticated users is equally well protected;
    - Preventing parameter manipulation.
    - Preventing disclosure of sensitive data.

Top issues need to address with secure design practices


Authenticating users
Preventing parameter manipulation
Providing secure configuration
Handling exceptions
Protecting sensitive data
Browser
Firewall
Web Server
Web Application
Application Server
Applications
Database Server
Database
Protecting sensitive data
Preventing session hijacking and cookie replay attacks
Validating input
Authorizing users
Authenticating and authorizing upstream identities
Auditing and logging activity and transactions
Encrypting or hashing sensitive data

European Union

Web Application Vulnerabilities due to Bad design

1. Input Validation
2. Authentication
3. Authorization
4. Configuration Management
5. Sensitive Data
6. Session Management
7. Cryptography
8. Parameter Manipulation
9. Exception Management
10. Auditing and Logging

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.

1.0 Input Validation

- Practices improve Web application's input validation:
  - 1.01 Assume all input is malicious.
  - 1.02 Centralize approach.
  - 1.03 Do not rely on client-side validation.
  - 1.04 Be careful with canonicalization issues.
  - 1.05 Constrain, reject, and sanitize your input.

1.01 Assume all inputs are malicious

- Validate all type of input (i.e. it comes from a service, a file share, a user, or a database) if the source is outside of the trusted boundary;

- **For example**

  - How do we know that malicious commands are not present if the application call an external Web service that returns strings?

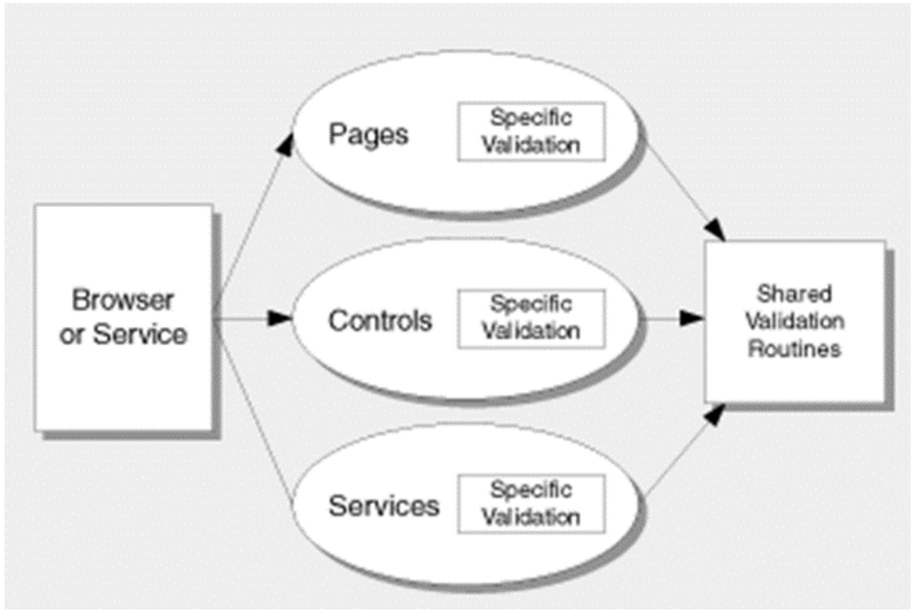
  - When you read data, how do you know whether it is safe if several applications write to a shared database?

1.02 Centralize your approach

- Make the input validation strategy a core element of the application design.
- Consider a centralized approach to validation, for example, by using common validation and filtering code in shared libraries.

<u>This ensures that</u>

- Validation rules are applied consistently.
- It reduces development effort.
- It helps with future maintenance



1.03 Do not rely on client-side validation.

- Server-side code should perform its own validation.
- Attackers may bypass the client or shuts off the 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.

1.04 Be careful with canonicalization issues

- Canonicalization is the process of converting data to its canonical form.
- Data in canonical form is in its most standard or simplest 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

** 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.

1.04 Be careful with canonicalization issues (Cont’d)

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
```

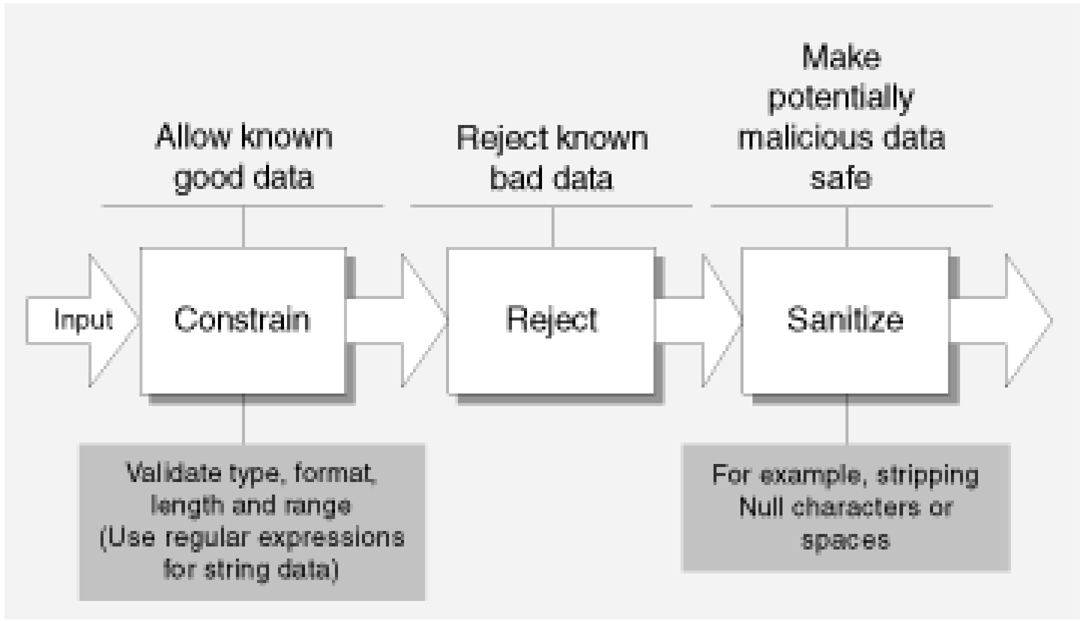
We should generally try to avoid designing applications that accept input file names from the user to avoid canonicalization issues.

Consider alternative designs instead. For example, let the application determine the file name for the user.

If we 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.

1.05 Constrain, reject, and sanitize your input

- The preferred approach to validating input is to **constrain** what we 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 we design the web application, we know what our application expects. The range of valid data is generally a more finite set than potentially malicious input. However, for defense in depth we may also want to reject known bad input and then sanitize the input.



Allow known good data
Reject known bad data
Make potentially malicious data safe
Input
Constrain
Reject
Sanitize
Validate type, format, length and range (Use regular expressions for string data)
For example, stripping Null characters or spaces

Fig: Recommended strategy

2.0 Authentication

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

1. Identify where authentication is required in the application. It is generally required whenever a trust boundary (includes assemblies, processes, and hosts) is crossed.
2. Validate the caller (Users typically authenticate themselves with user names and passwords).
3. Identify the user on subsequent requests. This requires some form of authentication token.

Practices improve the Web application's authentication:

2.01 Separate public and restricted areas.
2.02 Use account lockout policies for end-user accounts.
2.03 Support password expiration periods.
2.04 Be able to disable accounts.
2.05 Do not store passwords in user stores.
2.06 Require strong passwords.
2.07 Do not send passwords over the wire in plaintext.
2.08 Protect authentication cookies.

2.01 Separate public and restricted areas

- A public area of the 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.
- **Example:** 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 the site into public and restricted access areas,
  - 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 the site to limit the use of SSL to the areas that require authenticated access.

2.02 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 we 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.

2.03 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.

2.04 Be able to disable accounts

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

2.05 Do not store passwords in user stores

- If we 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.

2.06 Require strong passwords

- Do not make it easy for attackers to crack passwords.
- General practice is to require a minimum of eight characters and a mixture of uppercase and lowercase characters, numbers, and special characters.
- Whether we are using the platform to enforce these for us, or we are developing our 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.

2.07 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.

2.08 Protect authentication cookies

- A stolen authentication cookie is a stolen logon.
- Protect authentication tickets using encryption and secure communication channels.
- 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.

3.0 Authorization

Authorization determines

- What the authenticated identity can do and the resources that can be accessed.

The following practices improve the Web application's authorization:

3.01 Use multiple gatekeepers.

3.02 Restrict user access to system-level resources.

3.03 Consider authorization granularity.

European Union

3.01 Use multiple gatekeepers.

- 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.
- NTFS permissions allow us to specify per user access control lists.
- ASP.NET provides URL authorization and File authorization together with principal permission demands.

By combining these gatekeepers we can develop an effective authorization strategy.

3.02 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.

3.03 Consider authorization granularity

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

- **Impersonation Model:** 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.
- **Trusted subsystem Model:** Authorization performed in the application's logical middle tier using roles, which group together users who share the same privileges in the application.

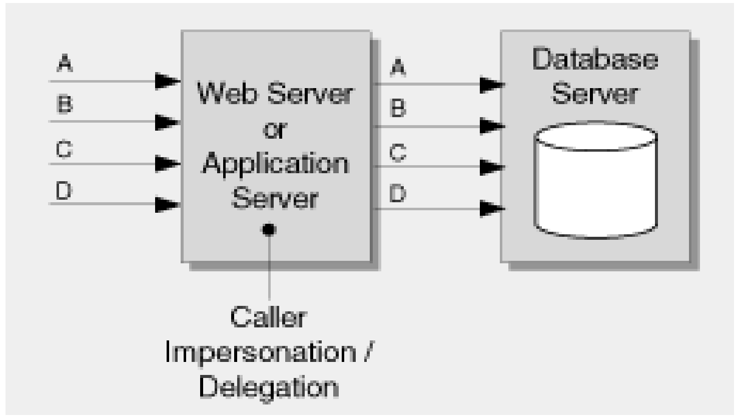

A
B
C
D
Web Server or Application Server
A
B
C
D
Database Server
Caller Impersonation / Delegation

Fig: Impersonation Model

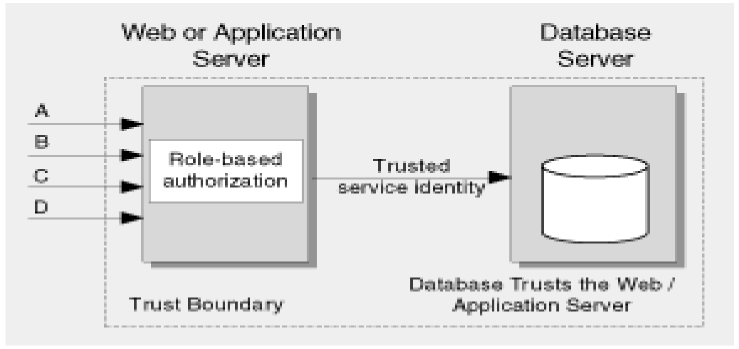
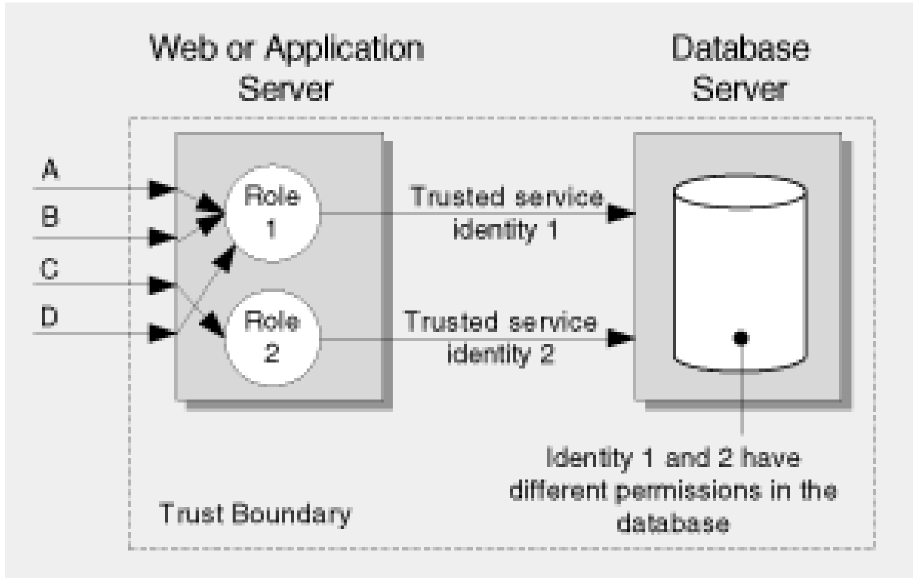


Fig: Trusted subsystem Model

3.03 Consider authorization granularity (Cont'd)

- **Hybrid Model:** This is really a hybrid of the two models described earlier. Callers are mapped to roles in the application's logical middle tier, and access to classes and methods is restricted based on role membership. Downstream resource access is performed using a restricted set of identities determined by the current caller's role membership.



Web or Application Server
Database Server
A
B
C
D
Role 1
Role 2
Trusted service identity 1
Trusted service identity 2
Identity 1 and 2 have different permissions in the database
Trust Boundary

Fig: Hybrid Model

4.0 Configuration Management

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

4.01 Secure the administration interfaces.

4.02 Secure the configuration store.

4.03 Maintain separate administration privileges.

4.04 Use least privileged process and service accounts.

4.01 Secure the administration interfaces

- Configuration management functionality should accessible only to authorized operators and administrators.
- Enforce strong authentication over the administration interfaces, for example, by using certificates.
- Limit or avoid the use of remote administration if possible.
- Require administrators to log on locally.
- If we 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.
- Consider limiting remote administration to computers on the internal network by using IPSec policies, to further reduce risk.

4.02 Secure the configuration store

- Avoid using configuration files in the application's web space.
- Using Windows ACLs or database permissions that will secure access to the configuration store.
- Avoid storing plaintext secrets such as database connection strings or account credentials.
- Secure the secret items using encryption.
- Restrict access to the registry key, file, or table that contains the encrypted data.

4.03 Maintain separate administration privileges

- If the functionality supported by the features of the 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.

4.04 Use least privileged process and service accounts

- An important aspect of the application's configuration is the
  - Process accounts used to run the Web server process; and
  - Service accounts used to access downstream resources and systems.
- Make sure accounts are set up as least privileged.
- **Example:** 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.

5.0 Sensitive Data

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

5.01 Retrieve sensitive data on demand.

5.02 Encrypt the data or secure the communication channel.

5.03 Do not store sensitive data in persistent cookies.

5.04 Do not pass sensitive data using the HTTP-GET protocol.

5.01 Retrieve sensitive data on demand

- The preferred approach is to retrieve sensitive data on demand when it is needed instead of 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**: Retrieve the secret when the application loads and 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

5.02 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.

5.03 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.

5.04 Do not pass sensitive data using the HTTP-GET protocol

- 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.

6.0 Session Management

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

- 6.01 Use SSL to protect session authentication cookies.
- 6.02 Encrypt the contents of the authentication cookies.
- 6.03 Limit session lifetime.
- 6.04 Protect session state from unauthorized access.

6.01 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.

6.02 Encrypt the contents of the authentication cookies

- Encrypt the cookie contents even if you are using SSL.
- SSL 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.

6.03 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.

6.04 Protect session state from unauthorized access

- Consider how session state is to be stored. For optimum performance, we can store session state in the Web application's process address space. However, this approach has limited scalability and implications in Web form 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.
- We 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.

7.0 Cryptography

The following practices improve your Web application's security when you use cryptography:

7.01 Do not develop your own cryptography.

7.02 Keep unencrypted data close to the algorithm.

7.03 Use the correct algorithm and correct key size.

7.04 Secure your encryption keys.

7.01 Do not develop your own cryptography

- Cryptographic algorithms and routines are extremely difficult to develop successfully.
- We 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.

7.02 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.

European Union

7.03 Use the correct algorithm and correct key size

- Choose the right algorithm for the right job
- 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~~2~~56 bit keys (16~~3~~2 bytes)
- RSA 384~~1~~6,384 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.

European Union

7.04 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**: By using DPAPI, 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.

8.0 Parameter Manipulation

- The following practices help secure your Web application's parameter manipulation:
  - 8.01 Encrypt sensitive cookie state.
  - 8.02 Make sure that users do not bypass your checks.
  - 8.03 Validate all values sent from the client.
  - 8.04 Do not trust HTTP header information.

8.01 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.

8.02 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.

8.03 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 we 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.

European Union

8.04 Do not trust HTTP header information.

- HTTP headers are sent at the start of HTTP requests and HTTP responses.
- 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.

9.0 Exception Management

- The following practices help secure your Web application's exception management:
  - 9.01 Do not leak information to the client.
  - 9.02 Log detailed error messages.
  - 9.03 Catch exceptions.

9.01 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).
- Return generic error messages to the client.

9.02 Log detailed error messages.

- Send detailed error messages to the error log.
- Send minimal information to the consumer of the 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 we do not log passwords or other sensitive data.

9.03 Catch exceptions

- Use structured exception handling and catch exception conditions.
- Doing so avoids leaving the 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 the application.
- Give special consideration to what occurs at the application boundary.

10.0 Auditing and Logging

- The following practices improve your Web application's security:

  10.1 Audit and log access across application tiers.

  10.2 Consider identity flow.

  10.3 Log key events.

  10.4 Secure log files.

  10.5 Back up and analyze log files regularly.

10.1 Audit and log access across application tiers

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

European Union

10.2 Consider identity flow

- Consider how the application will flow caller identity across multiple application tiers. We have two basic choices.
  - We can flow the caller's identity at the operating system level using the Kerberos protocol delegation.
    - This allows us 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.
  - We can flow the caller's identity at the application level and use trusted identities to access back-end resources. With this approach,
    - We have to trust the middle tier and there is a potential repudiation risk. We 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.

10.3 Log key events

- The types of events that should be logged include successful and failed logon attempts, modification of data, 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.

10.4 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.

10.5 Back up and analyze log files regularly

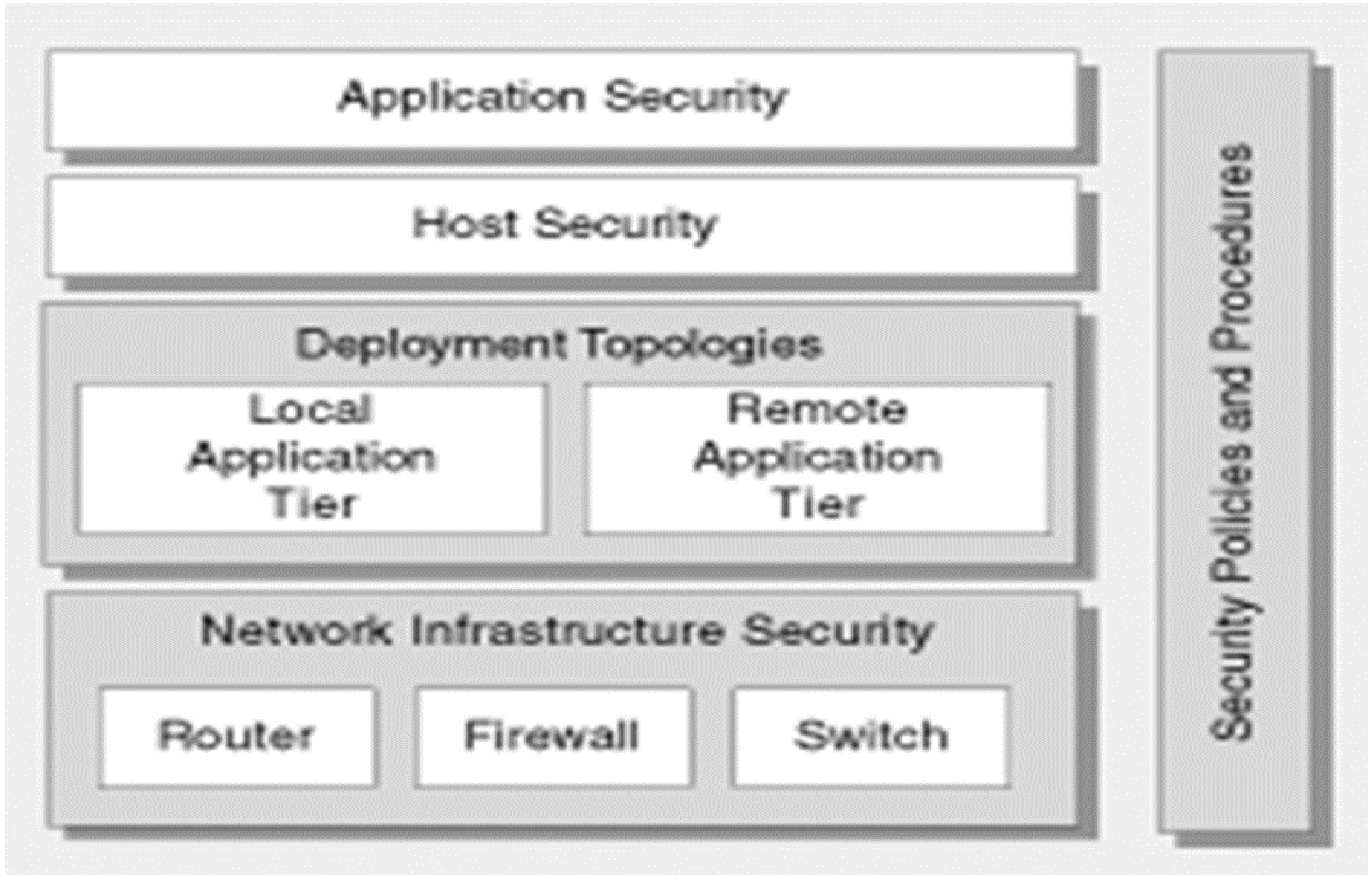
- There's no point in logging activity if the log files are never analyzed.
- Log files should be removed from production servers on a regular basis.
- The frequency of removal is dependent upon the application's level of activity.
- Design of application should consider the way that log files will be 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.

Deployment Considerations

During the application design phase,

- Review the corporate security policies and procedures with the infrastructure where application is to be deployed on.
- Since the target environment is rigid, web 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 to avoid surprises later and involve members of the network and infrastructure teams to help with this process.

Deployment aspects that require design time consideration



Security Policies and Procedures

- Determines what the applications are allowed to do and what the users of the application are permitted to do.
- Define restrictions to determine what applications and users are not allowed to do.
- Identify and work within the framework defined by the corporate security policy while designing the applications to make sure you do not breach policy that might prevent the application being deployed.

European Union

Network Infrastructure Components

- Understand the network structure provided by the target environment;
- Understand the baseline security requirements of the network in terms of filtering rules, port restrictions, supported protocols, and so on;
- Identify how firewalls and firewall policies are likely to affect the application's design and deployment. There may be
    - Firewalls to separate the Internet-facing applications from the internal network.
    - Additional firewalls in front of the database.
  - These can affect the possible communication ports and, therefore, authentication options from the Web server to remote application and database servers.
- Consider what protocols, ports, and services are allowed to access internal resources from the Web servers in the perimeter network.
- 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.
- Communicate and record any assumptions made about network and application layer security and which component will handle what.
- Pay attention to the security defenses that your application relies upon the network to provide.
- Consider the implications of a change in network configuration.

Deployment Topologies

- Understand web application's deployment topology
- Figure out whether remote application tier is a key consideration that must be incorporated in the design.
- If there have a remote application tier,
    - How to secure the network between servers to address the network
        - Eavesdropping (spying) threat
        - Ensure privacy and
        - Ensure integrity for sensitive data.
- Consider identity flow and the accounts used for network authentication when the application connects to remote servers.

Intranet, Extranet, and Internet

Present design challenges:

Questions that we should consider include:

- How will we flow caller identity through multiple application tiers to back- end resources?
- Where will we perform authentication?
- Can we trust authentication at the front end and then use a trusted connection to access back-end resources?
- In extranet scenarios, we also must consider whether we trust partner accounts.

European Union

Summary

- Security should permeate every stage of the product development life cycle and it should be a focal point of application design.
- Pay particular attention to the design of a solid authentication and authorization strategy.
- Also remember that the majority of application level attacks rely on maliciously formed input data and poor application input validation.
- The guidance presented in this chapter should help you with these and other challenging aspects of designing and building secure applications.