J.D. Meier's Blog

Software Engineering, Project Management, and Effectiveness

Security Principles

JD Meier 7 Apr 2008 12:03 PM | 5

If you know the underlying principles for security, you can be more effective in your security design. While working on Improving Web Application Security: Threats and Countermeasures, my team focused on creating a durable set of security principles. The challenge was to make the principles more useful. It's one thing to know the principles, but another to turn it into action.

Turning Insights Into Action

To make the principles more useful, we organized them using our Security Frame. Our Security Frame is a set of actionable, relevant categories that shape your key engineering and deployment decisions. With the Security Frame we could quickly find principles related to authentication, or authorization or input validation ... etc.

Once we had these principles and this organizing frame, we could then evaluate technologies against it to find effective, principle-based techniques. For example, when we analyzed doing input and data validation in ASP.NET, we focused on finding the best ways to constrain, reject, and sanitize input. For constraining input, we focused on checking for length, range, format and type. Using these strategies both shortened our learning curve and improved our results.

Core Security Principles

We started with a firm foundation of core security principles. These influenced the rest of our security design principles. Here's the core security principles we started with:

- **Adopt the principle of least privilege** – Processes that run script or execute code should run under a least privileged account to limit the potential damage that can be done if the process is compromised
- **Use defense in depth**. Place check points within each of the layers and subsystems within your application. The check points are the gatekeepers that ensure that only authenticated and authorized users are able to access the next downstream layer.
- **Don't trust user input.** Applications should thoroughly validate all user input before performing operations with that input. The validation may include filtering out special characters.
- **Use secure defaults**. If your application demands features that force you to reduce or change default security settings, test the effects and understand the implications before making the change
- **Don't rely on security by obscurity**. Trying to hide secrets by using misleading variable names or storing them in odd file locations does not provide security. In a game of hide-and-seek, it's better to use platform features or proven techniques for securing your data.
- **Check at the gate**. Checking the client at the gate refers to authorizing the user at the first point of authentication (for example, within the Web application on the Web server), and determining which resources and operations (potentially provided by downstream services) the

user should be allowed to access.

- **Assume external systems are insecure**. If you don't own it, don't assume security is taken care of for you.
- **Reduce Surface Area** Avoid exposing information that is not required. By doing so, you are potentially opening doors that can lead to additional vulnerabilities. Also, handle errors gracefully; don't expose any more information than is required when returning an error message to the end user.
- **Fail to a secure mode**. your application fails, make sure it does not leave sensitive data unprotected. Also, do not provide too much detail in error messages; meaning don't include details that could help an attacker exploit a vulnerability in your application. Write detailed error information to the Windows event log.
- **Security is a concern across all of your application layers and tiers**. Remember you are only as secure as your weakest link.
- **If you don't use it, disable it**. You can remove potential points of attack by disabling modules and components that your application does not require. For example, if your application doesn't use output caching, then you should disable the ASP.NET output cache module. If a future security vulnerability is found in the module, your application is not threatened.

Frame for Organizing Security Design Principles

Rather than a laundry list of security principles, you can use the Security Frame as a way to organize and share security principles:

- Auditing and Logging
- Authentication
- Authorization
- Configuration Management
- Cryptography
- Exception Management
- Input / Data Validation
- Sensitive Data
- Session Management

Auditing and Logging

Here's our security design principles for auditing and logging:

- **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**. You have two basic choices. You can flow the caller's identity at the operating system level or you can flow the caller's identity at the application level and use trusted identities to access back-end resources.
- **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
- **Protect log files.** Protect log files using access control lists 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.

Authentication

Here's our security design principles for 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. By partitioning your site into public and restricted access areas, you can apply separate authentication and authorization rules across the site.
- **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. 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.
- **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

Here's our security design principles for authorization:

- **Use multiple gatekeepers.** By combining multiple gatekeepers across layers and tiers, 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

access control lists 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 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: (1.) the impersonation model providing per end user authorization granularity, (2.) the trusted subsystem model uses the application's process identity for resource access, and (3.) the hybrid model uses multiple trusted service identities for downstream resource access. The most granular approach relies on impersonation. The impersonation model provides per end user authorization granularity.

Configuration Management

Here's our security design principles for configuration management:

- **Protect 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.
- **Protect your configuration store**. 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 access control lists 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.
- **Maintain 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.

Cryptography

Here's our security design principles for cryptography:

- **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.
- **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.

- **Protect 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. Avoid key management when you can, and when you do need to store encryption keys, cycle your keys periodically.

Exception Management

Here's our security design principles for exception management:

- **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.

Input / Data Validation

Here's our security design principles for input and data 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.
- **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. You should generally try to avoid designing applications that accept input file names from the user to avoid canonicalization issues.

- **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.
- **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 **referrer** 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 referrer field, for example, to check whether the request originated from a page generated by the Web application, because the field is easily falsified.

Sensitive Data

Here's our security design principles for sensitive data:

- **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 Local Security Authority (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.

- **Use Data Protection API (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.
- **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 caching along with potential security implications.
- **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

Here's our security design principles for session management:

- **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.
- **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. 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.

Using the Security Design Principles

This is simply a baseline set of principles so that you don't have to start from scratch. You can build on this set and tailor for your specific context. I find that while having a set of principles helps, that you can't stop there. To share the knowledge and help others use the information, it's important to encapsulate the principles in patterns as well as show concrete examples and create precise, actionable guidelines for developers. Personally, I've found Wikis to be the most effective way to share and manage the information.

Additional Resources

- patterns & practices Improving Web Application Security (MSDN)
- patterns & practices Security Engineering Explained (MSDN)
- Security Design Principles (Guidance Share)

My Related Posts

- Security Frame
- How To Use Guidance Explorer to Do a Security Code Inspection
- Guidance Share Sweep

Comments

» Security Principles Credit Card on Credit Speak: Find Info, News and More on Credit Card
7 Apr 2008 1:10 PM

PingBack from http://creditcard.creditspeak.com/2008/04/07/security-principles/

Eber Irigoyen
7 Apr 2008 2:31 PM

...and when you are done with all that, make it user-friendly J

Dave Ockwell-Jenner
8 Apr 2008 9:19 AM

This is some great foundational work! I couldn't agree more completely about the need to base everything upon solid principles. This takes me back to my days studying physics, where everything was proved through from first-principles... I think that lesson (as you have suggested) is a perfect fit for approaching security.

Jim Manico
8 Apr 2008 4:10 PM

Why no mention of HttpOnly cookies in your session handling section? Otherwise, this is a fantastic resource.

JD Meier
8 Apr 2008 10:12 PM

@Eber - I agree. Great design is effetive trade-offs/balance among user experience, business, and technological requirements.

@Dave - I'm a fan of principles. They simplify my life. I also like to know "how things work." That too has simplified my life.

@Jim - Good catch!