Design should also factor in protection mechanisms of the log, itself; and maintaining the chain of custody of the logs will ensure that the logs are admissible in court. Validating the integrity of the logs can be accomplished by hashing the before and after images of the logs and checking their hash values. Auditing in conjunction with other security controls such as authentication can provide non-repudiation. It is preferable to design the software to automatically log the authenticated principal and system timestamp, and not let it be user-defined, to avoid potential integrity issues. For example, using the Request. ServerVariables[LOGON_USER] in an IIS web application or the T-SQL in-built getDate() system function in SQL Server is preferred over passing a user-defined principal name or timestamp.

We have learned about how to design software incorporating core security elements of confidentiality, integrity, availability, authentication, authorization, and accountability. In the next section, we will learn about architecting software with secure design principles.

Architecting Software with Secure Design Principles

Some of the common, insecure design issues observed in software are the following:

- improper implementation of least privilege,
- software fails insecurely,
- authentication mechanisms are easily bypassed,
- security through obscurity,
- improper error handling, and
- weak input validation.

In the following section we will look at some of the design principles pertinent to architecting secure software. The following principles were introduced and defined in the Secure Software Concepts chapter. It is revisited here as a refresher and discussed in more depth with examples.

Least Privilege

Although the principle of least privilege is more applicable to administering a system, where the number of users with access to critical functionality and controls is restricted, least privilege can be implemented within software design. When software is said to be operating with least privilege, it means that only the necessary and minimum level of access rights (privileges) has been given

explicitly to it for a minimum amount of time in order for it to complete its operation. The main objective of least privilege is containment of the damage that can result from a security breach that occurs accidentally or intentionally. Some of the examples of least privilege include the military security rule of "need-to-know" clearance level classification, modular programming, and non-administrative accounts.

The military security rule of need-to-know limits the disclosure of sensitive information to only those who have been authorized to receive such information, thereby aiding in confidentiality assurance. Those who have been authorized can be determined from the clearance level classifications they hold, such as Top Secret, Secret, Sensitive but Unclassified, etc. Best practice also suggests that it is preferable to have many administrators with limited access to security resources instead of one user with "super user" rights.

Modular programming is a software design technique in which the entire program is broken down into smaller sub-units or modules. Each module is discrete with unitary functionality and is said to be therefore *cohesive*, meaning each module is designed to perform one and only one logical operation. The degree of how cohesive a module is indicates the strength at which various responsibilities of a software module are related. The discreetness of the module increases its maintainability and the ease of determining and fixing software defects. Since each unit of code (class, method, etc.) has a single purpose and the operations that can performed by the code is limited to only that which it is designed to do, modular programming is also referred to as the Single Responsibility Principle of software engineering. For example, the function, CalcDiscount(), should have the single responsibility to calculate the discount for a product while the CalcSH() function should be exclusively used to calculate shipping and handling rates. When code is not designed modularly, not only does it increase the attack surface, but it also makes the code difficult to read and troubleshoot. If there is a requirement to restrict the calculation of discounts to a sales manager, not separating this functionality into its own function, such as CalcDiscount(), can lead potentially to a non-sales manager's running code that is privileged to a sales manager. An aspect related to cohesion is *coupling*. Coupling is a reflection of the degree of dependencies between modules; i.e., how dependent one module is to another. The more dependent one module is to another, the higher its degree of coupling, and "loosely coupled modules" is the condition where the interconnections among modules are not rigid or hardcoded.

Good software engineering practices ensure that the software modules are *highly cohesive* and *loosely coupled* at the same time. This means that the dependencies between modules will be weak (loosely coupled) and each module will be responsible to perform a discrete function (highly cohesive).

Modular programming thereby helps to implement least privilege, in addition to making the code more readable, reusable, maintainable, and easy to troubleshoot.

The use of accounts with non-administrative abilities also helps implement least privilege. Instead of using the "sa" or "sysadmin" account to access and execute database commands, using a "datareader" or "datawriter" account is an example of least privilege implementation.

Separation of Duties

When design compartmentalizes software functionality into two or more conditions, all of which need to be satisfied before an operation can be completed, it is referred to as *separation of duties*. The use of split keys for cryptographic functionality is an example of separation of duties in software. Keys are needed for encryption and decryption operations. Instead of storing a key in a single location, splitting a key and storing the parts in different locations, with one part in the system's registry and the other in a configuration file, provides more security. Software design should factor in the locations to store keys, as well as the mechanisms to protect them.

Another example of separation of duties in software development is related to the roles that people play during its development and the environment in which the software is deployed. The programmer should not be allowed to review his own code nor should a programmer have access to deploy code to the production environment. We will cover in more detail the separation of duties based on the environment in the configuration section of the Software Deployment, Operations, Maintenance, and Disposal chapter.

When architected correctly, separation of duties reduces the extent of damage that can be caused by one person or resource. When implemented in conjunction with auditing, it can also discourage insider fraud, as it will require collusion between parties to conduct fraud.

Defense in Depth

Layering security controls and risk mitigation safeguards into software design incorporates the principle of *defense in depth*. This is also referred to as *layered*

defense. The reasons behind this principle are two-fold, the first of which is that the breach of a single vulnerability in the software does not result in complete or total compromise. In other words, defense in depth is akin to not putting all the eggs in one basket. Secondarily, incorporating the defense of depth in software can be used as a deterrent for the curious and non-determined attackers when they are confronted with one defensive measure over another.

Some examples of defense in depth measures are listed below.

- Use of input validation along with prepared statements or stored procedures, disallowing dynamic query constructions using user input to defend against injection attacks.
- Disallowing active scripting in conjunction with output encoding and input- or request-validation to defend against Cross-Site Scripting (XSS).
- The use of security zones, which separates the different levels of access according to the zone that the software or person is authorized to access.

Fail Secure

Fail secure is the security principle that ensures that the software *reliably* functions when attacked and is rapidly *recoverable* into a normal business and secure state in the event of design or implementation failure. It aims at maintaining the *resiliency* (confidentiality, integrity, and availability) of software by defaulting to a secure state. Fail secure is primarily an availability design consideration, although it provides confidentiality and integrity protection as well. It supports the design and default aspects of the SD3 initiative, which implies that the software or system is secure by design, secure by default, and secure by deployment. In the context of software security, "fail secure" can be used interchangeably with "fail safe" which is commonly observed in physical security.

Some examples of fail secure design in software include the following:

- The user is denied access by default and the account is locked out after the maximum number (clipping level) of access attempts is tried.
- Not designing the software to ignore the error and resume next operation. The On Error Resume Next functionality in scripting languages such as VBScript as depicted in *Figure 3.12*.
- Errors and exceptions are explicitly handled and the error messages are non-verbose in nature. This ensures that system exception information, along with the stack trace, is not bubbled up to the

```
Option Explicit
Dim objNetwork, strDrive, strRemotePath

strDrive = "J:"
strRemotePath = "\\FinServer\Software"

On Error Resume Next

Set objNetwork = CreateObject("WScript.Network")
objNetwork.MapNetworkDrive strDrive, strRemotePath

Wscript.Quit
```

Figure 3.12 – **On Error Resume Next**

client in raw form, which an attacker can use to determine the internal makeup of the software and launch attacks accordingly to circumvent the security protection mechanisms or take advantage of vulnerabilities in the software. Secure software design will take into account the logging of the error content into a support database and the bubbling up of only a reference value (such as error ID) to the user with instructions to contact the support team for additional support.

Economy of Mechanisms

In the Secure Software Concepts domain, we noted that one of the challenges to the implementation of security is the tradeoff that happens between the usability of the software and the security features that need to be designed and built in. With the noble intention of increasing the usability of software developers often design and code in more functionality, than is necessary. This additional functionality is commonly referred to as "bells-and-whistles". A good indicator of which features in the software are unneeded "bells-and-whistles" is reviewing the requirements traceability matrix (RTM) that is generated during the requirements gathering phase of the software development project. Bells-and-whistles features will never be part of the RTM. While such added functionality may increase user experience and usability of the software, it increases the attack surface and is contrary to the *economy of mechanisms,* secure design principle, which states that the more complex the design of the software, the more likely there is of vulnerabilities. Simpler design implies easy-to-understand programs, decreased attack surface, and fewer weak links. With a decreased attack surface, there is less opportunity for failure and when failures do occur, the time needed to understand and fix the issues is less, as well. Additional benefits of economy of

mechanisms include ease of understanding program logic and data flows and fewer inconsistencies. Economy of mechanism in layman's terms is also referred to as the KISS (Keep It Simple Stupid) principle and in some instances as the principle of *unnecessary complexity*. Modular programming not only supports the principle of least privilege but also supports the principle of economy of mechanisms.

Taken into account, the following considerations support the designing of software with the economy of mechanisms principle in mind:

- *Unnecessary functionality or unneeded security mechanisms should be avoided.* Since patching and configuration of newer software versions has been known to security features that were disabled in previous versions, it is advisable to not even design unnecessary features, instead of designing them and leaving the features in a disabled state.
- *Strive for simplicity.* Keeping the security mechanisms simple ensures that the implementation is not partial, which could result in compatibility issues. It is also important to model the data to be simple so that the data validation code and routines are not overly complex or incomplete. Supporting complex, regular expressions for data validation can result in algorithmic complexity weaknesses as stated in the Common Weakness Enumeration publication 407 (CWE-407).
- *Strive for operational ease of use.* Single Sign On (SSO) is a good example that illustrates the simplification of user authentication so that the software is operationally easy to use.

Complete Mediation

In the early days of web application programming, it was observed that a change in the value of a QueryString parameter would display the result that was tied to the new value without any additional validation. For example, if Aidan is logged in, and the Uniform Resource Locator (URL) in the browser address bar shows the name value pair, user=aidan, changing the value "aidan" to "reuben" would display reuben's information without validating that the logged-on user is indeed Reuben. If Aidan changes the parameter value to user=reuben, he can view Reuben's information, potentially leading to attacks on confidentiality, wherein Reuben's sensitive and personal information is disclosed to Aidan.

While this is not as prevalent today as it used to be, similar design issues are still evident in software. Not checking access rights each time a subject requests access to objects violates the principle of complete mediation. *Complete mediation*

is a security principle that states that access requests need to be mediated each time, every time, so that authority is not circumvented in subsequent requests. It enforces a system-wide view of access control. Remembering the results of the authority check, as is done when the authentication credentials are cached, can increase performance; however, the principle of complete mediation requires that results of an authority check be examined skeptically and systematically updated upon change. Caching can therefore lead to an increased security risk of authentication bypass, session hijacking and replay attacks, and Man-in-the-Middle (MITM) attacks. Therefore, designing the software to rely solely on client-side, cookie-based caching of authentication credentials for access should be avoided, if possible.

Complete mediation not only protects against authentication threats and confidentiality threats, but is useful in addressing the integrity aspects of software, as well. Not allowing browser post-backs without validation of access rights, or checking that a transaction is currently in a state of processing, can protect against the duplication of data, avoiding data integrity issues. Merely informing the user to not click more than once, as depicted in *Figure 3.13*, is not foolproof and so design should include the disabling of user controls once a transaction is initiated until the transaction is completed.

The complete mediation design principle also addresses the failure to protect alternate path vulnerability. To properly implement complete mediation in software, it is advisable during the design phase of the SDLC to identify all

Figure 3.13 – Weak design of complete mediation

possible code paths that access privileged and sensitive resources. Once these privileged code paths are identified, then the design must force these code paths to use a single interface that performs access control checks before performing the requested operation. Centralizing input validation by using a single input validation layer with a single, input filtration checklist for all externally controlled inputs is an example of such design. Alternatively, using an external input validation framework that validates all inputs before they are processed by the code may be considered when designing the software.

Complete mediation also augments the protection against the *weakest link*. Software is only as strong as its weakest component (code, service, interface, or user). It is also important to recognize that any protection that technical safeguards provide can be rendered futile if people fall prey to social engineering attacks or are not aware of how to use the software. The catch 22 is that *people* who are the first line of defense in software security can also become the weakest link, if they are not made aware, trained, and educated in software security.

Open Design

Dr. Auguste Kerckhoff, who is attributed with giving us the cryptographic *Kerckhoff's principle,* states that all information about the crypto system is public knowledge except the key, and the security of the crypto system against cryptanalysis attacks is dependent on the secrecy of the key. An outcome of the Kerckhoff's principle is the open design principle, which states that the implementation of security safeguards should be independent of the design, itself, so that review of the design does not compromise the protection the safeguards offer. This is particularly applicable in cryptography where the protection mechanisms are decoupled from the keys that are used for cryptographic operations, and algorithms used for encryption and decryption are open and available to anyone for review.

The inverse of the open design principle is *security through obscurity,* which means that the software employs protection mechanisms whose strength is dependent on the obscurity of the design, so much so that the understanding of the inner workings of the protection mechanisms is all that is necessary to defeat the protection mechanisms. A classic example of security through obscurity, which must be avoided if possible, is the hard coding and storing of sensitive information, such as cryptographic keys, or connection strings information with username and passwords inline code, or executables. Reverse engineering, binary analysis of executables, and runtime analysis of protocols can reveal these secrets. Review of the Diebold voting machines code revealed that passwords

were embedded in the source code, cryptographic implementation was incorrect, the design allowed voters to vote an unlimited number of times without being detected, privileges could be escalated, and insiders could change a voter's ballot choice, all of which could have been avoided if the design was open for review by others. Another example of security through obscurity is the use of hidden form fields in web applications, which affords little, if any protection against disclosure, as they can be processed using a modified client.

Software design should therefore take into account the need to leave the design open but keep the implementation of the protection mechanisms independent of the design. Additionally, while security through obscurity may increase the work factor needed by an attacker and provide some degree of defense in depth, it should not be the sole and primary security mechanism in the software. Leveraging publicly vetted, proven, tested industry standards, instead of custom developing one's own protection mechanism, is recommended. For example, encryption algorithms, such as the Advanced Encryption Standard (AES) and Triple Data Encryption Standard (3DES), are publicly vetted and have undergone elaborate security analysis, testing, and review by the information security community. The inner workings of these algorithms are open to any reviewer, and public review can throw light on any potential weaknesses. The key that is used in the implementation of these proven algorithms is what should be kept secret.

Some of the fundamental aspects of the open design principle are as follows:

- The security of your software should not be dependent on the *secrecy of the design*.
- Security through obscurity should be avoided.
- The design of protection mechanisms should be open for scrutiny by members of the community, as it is better for an ally to find a security vulnerability or flaw than it is for an attacker.

Least Common Mechanisms

Least common mechanisms is the security principle by which mechanisms common to more than one user or process are designed not to be shared. Since shared mechanisms, especially those involving shared variables, represent a potential information path, mechanisms that are common to more than one user and depended on by all users are to be minimized. Design should compartmentalize or isolate the code (functions) by user roles, since this increases the security of the software by limiting the exposure. For example, instead of having one

function or library that is shared between members with supervisor and non-supervisor roles, it is recommended to have two distinct functions, each serving its respective role.

Psychological Acceptability

One of the primary challenges in getting users to adopt security is that they feel that security is usually very complex. With a rise in attacks on passwords, many companies resolved to implement strong password rules, such as the need to have mixed-case, alpha-numeric passwords which are to be of a particular length. Additionally these complex passwords are often required to be periodically changed. While this reduced the likelihood of brute-forcing or guessing passwords, it was observed that the users had difficulty remembering complex passwords. Therefore, they nullified the effect that the strong password rules brought by jotting down their passwords and sticking them under their desks and, in some cases, even on their computer screens. This is an example of security protection mechanisms that were not psychologically acceptable and, hence, not effective.

Psychological acceptability is the security principle that states that security mechanisms should be designed to maximize usage, adoption, and automatic application.

A fundamental aspect of designing software with the psychological acceptability principle is that the security protection mechanisms:

- are easy to use,
- do not affect accessibility, and
- are transparent to the user.

Users should not be additionally burdened as a result of security and the protection mechanisms must not make the resource more difficult to access than if the security mechanisms were not present. Accessibility and usability should not be impeded by security mechanisms, because otherwise, users will elect to turn off or circumvent the mechanisms, thereby neutralizing or nullifying any protection that is designed.

Examples of incorporating the psychological acceptability principle in software include designing the software to notify the user through explicit error messages and callouts as depicted in *Figure 3.14*, message box displays and help dialogs, and intuitive user interfaces.

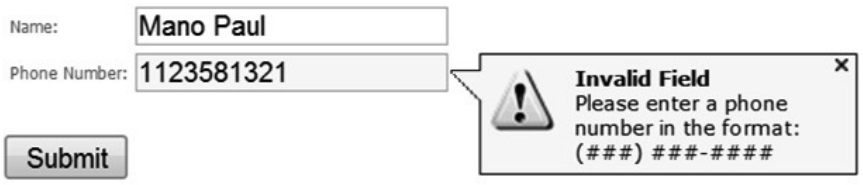

Figure 3.14 – **Callouts**

Weakest Link

You may have heard of the saying, "*A chain is only as strong as its weakest links.*" This security principle states that the resiliency of your software against hacker attempts will depend heavily on the protection of its weakest components, be it the code, service or an interface. A breakdown in the weakest link will result in a security breach.

Another approach to this security related concept is that "*A chain is only as weak as its strongest links.*" Irrespective of the approach one takes, what is important to note is that when designing software, careful attention must be given so that there are no exploitable components.

A related concept is "Single Point of Failure". Software must be architected so that there is no single source of complete compromise. While this is similar to the Weakest Link principle, the distinguishing difference between the two is that the weakest link need not necessarily be as a result of a single point of failure but could be as a result of various weak sources. Usually in software security, the weakest link is a superset of several single points of failures. When software is designed with defense in depth, threats arising from weakest links and single points of failure are mitigated.

Leveraging Existing Components

Service Oriented Architecture (SOA) is prevalent in today's computing environment and one of the primary aspects for its popularity is the ability it provides for communication between heterogeneous environments and platforms. Such communication is possible because the service oriented architecture protocols are understandable by disparate platforms, and business functionality is abstracted and exposed for consumption as contract-based, application programming interfaces (APIs). For example, instead of each financial institution writing its own currency conversion routine, it can invoke a common, currency conversion, service contract. This is the fundamental premise of the leveraging existing components design principle. Leveraging existing components is the security principle that promotes the reusability of existing components.

A common observance in security code reviews is that developers try to write their own cryptographic algorithms instead of using validated and proven cryptographic standards such as AES. These custom implementations of cryptographic functionality are also determined often to be the weakest link. Leveraging proven and validated cryptographic algorithms and functions is recommended.

Designing the software to scale using tier architecture is advisable from a security standpoint, since the software functionality can be broken down into presentation, business, and data access tiers. The use of a single, data access layer (DAL) to mediate all access to the backend data stores not only supports the principle of leveraging existing components but also allows for scaling to support various clients or if the database technology changes. Enterprise application blocks are recommended over custom developing shared libraries and controls that attempt to provide the same functionality as the enterprise application blocks.

Reusing tested and proven, existing libraries and common components has the following security benefits:. First, the attack surface is not increased, and second, no newer vulnerabilities are introduced. An ancillary benefit of leveraging existing components is increased productivity because leveraging existing components can significantly reduce development time.

Balancing Secure Design Principles

It is important to recognize that it may not be possible to design for each of these security principles in totality within your software, and tradeoff decisions about the extent to which these principles can be designed may be necessary. For example, while SSO can heighten user experience and increase psychological acceptability, it contradicts the principle of complete mediation and so a business decision is necessary to determine the extent to which SSO is designed into the software or to determine that it is not even an option to consider. SSO design considerations should also take into account the need to ensure that there is no single point of failure and that appropriate, defense in depth mitigation measures are undertaken. Additionally, implementing complete mediation by checking access rights and privileges, each time and every time, can have a serious impact on the performance of the software. So this design aspect needs to be carefully considered and factored in, along with other defense in depth strategies, to mitigate vulnerability while not decreasing user experience or psychological acceptability. The principle of least common mechanism may