

Authentication & Authorization

Authentication & Authorization Design Standards

Table of Contents

[2 Introduction 3](#_Toc424559874)

[2.1 Purpose 3](#_Toc424559875)

[3 Authentication 4](#_Toc424559876)

[3.1 User Accounts 4](#_Toc424559877)

[3.2 Email Addresses 4](#_Toc424559878)

[3.3 Passwords 4](#_Toc424559879)

[3.4 Logging In 5](#_Toc424559880)

[3.5 Session Management 6](#_Toc424559881)

[4 Authorization 7](#_Toc424559882)

[4.1 User Roles 7](#_Toc424559883)

[4.2 User Role Management 7](#_Toc424559884)

[5 Security 8](#_Toc424559885)

[5.1 System Accounts 8](#_Toc424559886)

[5.2 Error Messages 8](#_Toc424559887)

[5.3 Secure Communication 9](#_Toc424559888)

[5.4 General Validation 10](#_Toc424559889)

[5.5 SQL Injection 10](#_Toc424559890)

[5.6 Cross-Site Scripting (XSS) 10](#_Toc424559891)

[6 Logging 12](#_Toc424559892)

[6.1 Events to Log 12](#_Toc424559893)

[6.2 Log Retention 12](#_Toc424559894)

# Introduction

## Purpose

Security and Privacy are required considerations for any software development. This document provides the standard methods of ensuring both in software developed for AFS Technologies.

# Authentication

## User Accounts

Access to any application should be controlled via user accounts. Accounts should be created on a per-user basis. There may be times when a user has access to multiple accounts, but this should be kept to a minimum and multiple users should not have access to the same account. Only users who have authenticated themselves using a valid account should have access to the application. Anonymous users should only have access to the login page. Any anonymous user attempting to access any part of the application should be redirected to the login page.

The creation and management of user accounts should be controlled by trusted individuals. This may include internal system administrators or other users who have been explicitly granted this authority. Ensure that only this select group of users can create new user accounts. This keeps access to the application under control and helps protect against both internal and external attacks.

## Email Addresses

User accounts should be associated to an email address. This email address is a line of communication between the application and the user. In particular, password reset emails should go to the address tied to the account.

## Passwords

All accounts should use a username and password combination for authentication. Password strength should be enforced: an industry standard is that all passwords should contain a minimum of eight characters, containing at least one uppercase letter, one lowercase letter, and one number. This is the recommended minimum password complexity.

Enforcing password strength protects user accounts from being easily compromised by attackers.

Account security can be enhanced by requiring that users periodically change their passwords. Password security settings can also be set on a per-client basis. In applications that use this, it provides a way to further secure accounts to a client’s own security standards.

Passwords should be hashed or encrypted at the database level. The SHA-256 hashing algorithm is recommended for this.

When a user is given an auto-generated password (this occurs on the creation of a new account, or on the password reset of an existing account), the user should be forced to choose a new password upon logging in again.

When choosing a new password, the user must enter their existing password, and must enter their new password twice. Entering their existing passwords serves to re-authenticate the user, ensuring someone else has not gained access to their account.

Repeating the new password helps prevent the user from mistyping their password. Mistyping a password, while an annoyance for the user, also prevents a security risk: the more common it is that users must reset their password, the harder it is to catch attackers trying to compromise accounts.

The in-app password reset utility should be the only way passwords get reset by the user. If a user calls Support they should only be able to trigger the application’s password reset, which randomly generates a new password and emails that password to the user. This protects against social engineering attacks, where an attacker poses as a legitimate user who has lost their password. The attacker then contacts support, expecting Support to issue them a new password specified by the attacker or based on basic information (e.g. the user’s birthday). With this standard in place, only someone who has access to the user’s email account will have access to newly-generated passwords.

## Logging In

All applications should have a central login page. Anonymous users are redirected to this page. This forces all users to authenticate themselves before gaining access to the system.

In .NET Web Applications, this should make use of Forms Authentication.

The login page should use SSL. At no point should plaintext login credentials be transferred between the client’s machine and the application server. Accessing the login page via unprotected HTTP should redirect to HTTPS.

All login attempts should be logged. After repeated unsuccessful login attempts, a user account should be locked. Unlocking an account requires either resetting the user’s password, manual unlocking from an administrator, or both. It’s standard to allow two incorrect login attempts, locking the user after a third incorrect login attempt.

## Session Management

The application should make use of Cookies and Sessions, where appropriate, to manage the authenticated state of the current user. The user account should be stored in Session. A Cookie should be used to identify the user’s Session. This Session is stored in memory on the application server and is managed by IIS.

Cookies should simply tie a user to a server-managed Session. A Cookie should not contain the username or password (including encrypted or hashed passwords) of the user.

Sessions should time out after a period of time. Generally this is after one hour of inactivity. Once a Session has timed out, the user will be redirected to the login page and forced to re-authenticate in order to continue using the system.

When the user is logged out of the system (voluntarily or forcibly as a result of a timeout), their associated Cookie and Session data is cleared.

# Authorization

## User Roles

User Roles should be created in such a way that they can be applied many-to-many in regards to User Accounts. That is, multiple Users can share the same Role, and one User can have multiple Roles.

These Roles function to grant or restrict a user’s access to various parts of the application.

Each distinct area of the application should have an associated View and Edit role. Edit roles allow a user to both view and edit, while View roles provide view-only access.

If a user attempts to perform an action they are not privileged to perform, they should be shown an error message. Ideally, in web applications, this error message should be returned to the user with “401 Unauthorized” status code.

A user should not be granted more privilege than needed. Their user roles should be customized to limit their access and abilities in the system to whatever that individual user needs, and nothing more.

## User Role Management

A specific User Role should grant the ability to add or remove roles from other users. This role is highly privileged: while all users should be given the minimum permissions necessary, this role in particular is important to limit. The more users that are allowed to manage application permissions, the less site security, account integrity, and data integrity can be ensured.

# Security

## System Accounts

In order to reduce vulnerability to attackers, system accounts should be tightly controlled.

* Access to server Windows accounts, particularly Administrator accounts, should be tightly controlled. As with other privilege types, no user should have administrative access who does not require this for their job duties.
* The application should connect to any databases using an account with customized privileges. The database user used by the application should not have any more privileges than is required to execute the kinds of queries the application requires. For example, the application should not connect using a database user with *DROP* privileges if the application is never expected to execute a *DROP* query.
* All accounts, including Windows and Database accounts, should conform to password requirements equal to or stricter than the password requirements for the application itself.

## Error Messages

Error messages must be customized to not reveal more information than necessary.

A good error message will:

* Alert the user than an error has occurred.
* Use non-technical terms to describe the error to the user, if possible.
* Direct the user on how they can contact Support with help on resolving the error.

A good error message will not contain any debug or technical information. This data should be logged server-side and should not be a part of any HTTP Response, even if the data is not displayed to the user.

To conform to these standards, custom-written error pages should be used in .NET web applications. The default errors screens should not be used, even if debug info is hidden. The application should never reveal debug information, or server information (including the server operating system version or the .NET Framework version). This information is useful to attackers who may be looking exploit specific software versions. Furthermore, error pages that reveal or imply database architecture are also dangerous and must be avoided.

## Secure Communication

Network communication between the application and an external entity should be encrypted where possible. This includes:

* Using SSL for HTTP connections to the application.
* Using SFTP for FTP connections to the application.
* Using a VPN or other means of encryption for communication between application servers.

For any communication protocol used by the application, check whether there is an encrypted equivalent. If there is no equivalent, consider whether another encrypted protocol could be substituted.

No data should leave internal network of the application in an unencrypted state.

Form Validation

Form validation is crucial in enforcing authorization controls. This is a subtopic on the general topic of application security, in the sense that any security vulnerabilities may be exploited by an attacker to bypass application controls or elevate account privileges.

## General Validation

All form input must be validated server-side. Never rely on client-side (e.g. JavaScript) validation, as this can be easily bypassed by attackers. Fields should be validated at a minimum for datatype, minimum value (or minimum length), and maximum value (or maximum length).

This not only helps ensure data integrity, but can also protect against certain attacks such as buffer overflows. Preventing bad data from being stored in the database helps minimize the number of errors users will experience, and in the case of attackers, helps minimize the number of errors an attacker is able to force.

## SQL Injection

SQL Injections are attacks on the application’s database, and have the potential to be devastating. At its worst, imagine that a SQL Injection can give an attacker the full ability to read or write anything in the application’s databases. There are a few ways to help prevent these kinds of attacks:

* Make use of Stored Procedures in SQL Server, use named parameters, and minimize use of dynamically-generated SQL.
* All user input must be validated. If there is ever a point where the application generates a SQL query dynamically, all parameters must be cleaned. This includes, at a minimum, escaping all special characters (such as apostrophes and percent signs).
* Minimize the privileges granted to the database user used by the application. Do not grant the database user any commands it does not need, nor access to any databases/tables it does not need. This does not prevent SQL Injection, but can greatly minimize the damage done by a successful attack.

## Cross-Site Scripting (XSS)

XSS is two separate attacks with the same name. In Passive XSS, the user submits HTML or other malicious code as form input. This form input is then stored in the database and displayed for other users. For example, say the application has a page listing every username. If an attacker specified their username as “<strong>username</strong>”, when any other user views the username list, that HTML will be rendered and the username will appear bolded.

This can be used in a number of malicious ways. These are attacks, not necessarily on the application (like SQL Injection), but on its users. It can be used to attempt to gain control over another user’s account, or to serve malware to every user in the system.

The way to prevent this is to cleanse all form input, either before it gets into the database or before it gets displayed back to a user. The two main ways of achieving this are:

* Strip all HTML from form input (e.g. using Regex), or…
* Replace all HTML Special Characters with their associated HTML Entities (e.g. replace all instances of “<” and “>” with “&lt;” and “&gt;”.

There are often methods in server-side languages for doing either of these automatically. Always use the built-in methods where possible, as opposed to writing a custom method for the same task.

# Logging

## Events to Log

Logging is essential to building a secure and private application. There are a number of different actions that should be logged:

* User logins should be logged in order to recognize attacks on accounts and lock accounts being attacked.
* The database should be set up in a way that any data modifications can be tied back to the specific user that made them. This includes logging the time of the modification, the user who made the modification, and what exactly was modified.
* All application errors should be logged. While Windows provides some basic logging for this purpose, it does not always log sufficient information. Errors should be logged to a database table, recording information such as the user who encountered the error, the error message itself, and any other information a developer may find useful when attempting to reproduce the error. Appropriate parties (developers, support analysts, etc.) should either monitor these errors or be directly notified of them.  
    
  This is not only useful from a user experience standpoint (as it allows developers to fix application errors), but it also helps monitor attacks on the system. Hacking attempts can often cause errors (sometimes intentionally, as an attacker may be probing the system for what causes and error and what doesn’t). Logging these errors and monitoring these logs allows unusual or outright malicious behavior to be recognized.

## Log Retention

Logs should be retained for a number of days as specified by contract, but keeping them longer is recommended.

Different logs should have different priorities, in order to balance space usage with usefulness. Data change logs, ideally, should rarely be purged. These logs often contain useful information, tracing the history of the application’s data.

From a user-experience and developer standpoint, keeping error logs from the past few releases lets developers identify new errors introduced by a release. This allows developers researching a specific error to look back over previous releases to identify when the error was introduced. That said, code is written, rewritten, and removed with every application release, so certain error logs can become irrelevant over time.