|  |  |
| --- | --- |
| A picture of a winding road and trees  Application sECURITY  Through development | Abstract  Secure your applications from the beginning  Hazem Waddah  Application Security |

Application Security

Through development

# Content

Contents

[Content 2](#_Toc121248569)

[Prefix 7](#_Toc121248570)

[Chapter 1: dotNet Microsoft Application Security 8](#_Toc121248571)

[Introduction 8](#_Toc121248572)

[Updating the Framework 8](#_Toc121248573)

[Always utilize the latest version 8](#_Toc121248574)

[Adjust .NET Framework security on a zone-by-zone basis 9](#_Toc121248575)

[Adjust the level of trust in a .NET Framework assembly 10](#_Toc121248576)

[Restore policy levels that have been customized 10](#_Toc121248577)

[Evaluate the permissions that are granted to an assembly 10](#_Toc121248578)

[Audit the security of .NET-connected applications 10](#_Toc121248579)

[Configure a .NET-connected application and SQL Server to use an alternate port number for network communications 11](#_Toc121248580)

[Lock down an ASP.NET web application or web service 11](#_Toc121248581)

[Configure NTFS file permissions to increase security of ASP.NET applications 12](#_Toc121248582)

[Configure SQL Server security for applications that are built on the .NET Framework 12](#_Toc121248583)

[Configure URLScan to increase protection of ASP.NET web applications 13](#_Toc121248584)

[Require authentication for ASP.NET web applications 13](#_Toc121248585)

[Restrict specific users from gaining access to specified web resources 13](#_Toc121248586)

[Limit the web services protocols that a server permits 13](#_Toc121248587)

[Don't permit browser access to .NET-connected web services 14](#_Toc121248588)

[Use ASP.NET to protect file types 14](#_Toc121248589)

[Upgrade Framework 14](#_Toc121248590)

[Chapter 2: OWASP ASVS 15](#_Toc121248591)

[Introduction 15](#_Toc121248592)

[The .NET Framework 15](#_Toc121248593)

[Updating the Framework 15](#_Toc121248594)

[.NET Framework Guidance 15](#_Toc121248595)

[Data Access 16](#_Toc121248596)

[Encryption 16](#_Toc121248597)

[HTTP validation and encoding 17](#_Toc121248598)

[Forms authentication 17](#_Toc121248599)

[ASP NET MVC Guidance 18](#_Toc121248600)

[A1 Injection 18](#_Toc121248601)

[SQL Injection 18](#_Toc121248602)

[OS Injection 19](#_Toc121248603)

[LDAP injection 21](#_Toc121248604)

[A2 Broken Authentication 21](#_Toc121248605)

[A3 Sensitive Data Exposure 22](#_Toc121248606)

[A4 XML External Entities (XXE) 23](#_Toc121248607)

[A5 Broken Access Control 24](#_Toc121248608)

[Weak Account management 24](#_Toc121248609)

[Missing function-level access control 25](#_Toc121248610)

[Insecure Direct object references 25](#_Toc121248611)

[A6 Security Misconfiguration 26](#_Toc121248612)

[Debug and Stack Trace 26](#_Toc121248613)

[Cross-site request forgery 26](#_Toc121248614)

[A7 Cross-Site Scripting (XSS) 29](#_Toc121248615)

[A8 Insecure Deserialization 30](#_Toc121248616)

[A9 Using Components with Known Vulnerabilities 31](#_Toc121248617)

[A10 Insufficient Logging & Monitoring 31](#_Toc121248618)

[Logging 31](#_Toc121248619)

[Monitoring 33](#_Toc121248620)

[OWASP 2013 33](#_Toc121248621)

[A10 Unvalidated redirects and forwards 33](#_Toc121248622)

[XAML Guidance 34](#_Toc121248623)

[Windows Forms Guidance 34](#_Toc121248624)

[WCF Guidance 35](#_Toc121248625)

[Chapter 3: REST API Security Standard OWASP ASVS 35](#_Toc121248626)

[Introduction 35](#_Toc121248627)

[HTTPS 36](#_Toc121248628)

[Access Control 36](#_Toc121248629)

[JWT 36](#_Toc121248630)

[API Keys 38](#_Toc121248631)

[Restrict HTTP methods 38](#_Toc121248632)

[Input validation 38](#_Toc121248633)

[Validate content types 39](#_Toc121248634)

[Validate request content types 39](#_Toc121248635)

[Send safe response content types 39](#_Toc121248636)

[Management endpoints 40](#_Toc121248637)

[Error handling 40](#_Toc121248638)

[Audit logs 40](#_Toc121248639)

[Security Headers 40](#_Toc121248640)

[CORS 42](#_Toc121248641)

[Sensitive information in HTTP requests 42](#_Toc121248642)

[HTTP Return Code 43](#_Toc121248643)

[Chapter 4: SQL Injection Prevention 45](#_Toc121248644)

[Introduction 45](#_Toc121248645)

[Primary Defenses 47](#_Toc121248646)

[Defense Option 1: Prepared Statements (with Parameterized Queries) 47](#_Toc121248647)

[Defense Option 2: Stored Procedures 48](#_Toc121248648)

[Defense Option 3: Allow-list Input Validation 50](#_Toc121248649)

[Defense Option 4: Escaping All User-Supplied Input 51](#_Toc121248650)

[Additional Defenses 56](#_Toc121248651)

[Least Privilege 57](#_Toc121248652)

[Multiple DB Users 57](#_Toc121248653)

[Views 58](#_Toc121248654)

[Allow-list Input Validation 58](#_Toc121248655)

[Related Articles 58](#_Toc121248656)

[Chapter 5: SonarQube Application Security 60](#_Toc121248657)

[Use parameterized queries, prepared statements, or stored procedures and bind variables to SQL query parameters. 60](#_Toc121248658)

[Never use hard-coded passwords 60](#_Toc121248659)

[Always use strong encryption 60](#_Toc121248660)

[Use Parameterized Query 60](#_Toc121248661)

[Always use HTTPS not HTTP 61](#_Toc121248662)

[Chapter 6: Top software vulnerabilities globally 61](#_Toc121248663)

[CWE: Top 25 Most Dangerous Software Weaknesses CWE 2022 61](#_Toc121248664)

[Types of Weaknesses 61](#_Toc121248665)

[Base 61](#_Toc121248666)

[Class 61](#_Toc121248667)

[Composite 62](#_Toc121248668)

[Variant 62](#_Toc121248669)

[Out-of-bounds Write - (787)-B 62](#_Toc121248670)

[Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') - (79)-B 62](#_Toc121248671)

[Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') - (89)-B 62](#_Toc121248672)

[Improper Input Validation - (20)-C 63](#_Toc121248673)

[Out-of-bounds Read - (125)-B 63](#_Toc121248674)

[Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') - (78)-B 63](#_Toc121248675)

[Use After Free - (416)-V 63](#_Toc121248676)

[Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') - (22)-B 64](#_Toc121248677)

[Cross-Site Request Forgery (CSRF) - (352)-C 64](#_Toc121248678)

[Unrestricted Upload of File with Dangerous Type - (434)-B 64](#_Toc121248679)

[NULL Pointer Dereference - (476)-B 64](#_Toc121248680)

[Deserialization of Untrusted Data - (502)-B 65](#_Toc121248681)

[Integer Overflow or Wraparound - (190)-B 65](#_Toc121248682)

[Improper Authentication - (287)-C 65](#_Toc121248683)

[Use of Hard-coded Credentials - (798)-B 65](#_Toc121248684)

[Missing Authorization - (862)-C 66](#_Toc121248685)

[Improper Neutralization of Special Elements used in a Command ('Command Injection') - (77)-C 66](#_Toc121248686)

[Missing Authentication for Critical Function - (306)-B 66](#_Toc121248687)

[Improper Restriction of Operations within the Bounds of a Memory Buffer - (119)-C 66](#_Toc121248688)

[Incorrect Default Permissions - (276)-B 67](#_Toc121248689)

[Server-Side Request Forgery (SSRF) - (918)-B 67](#_Toc121248690)

[Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition') - (362)-C 67](#_Toc121248691)

[Uncontrolled Resource Consumption - (400)-C 68](#_Toc121248692)

[Improper Restriction of XML External Entity Reference - (611)-B 68](#_Toc121248693)

[Improper Control of Generation of Code ('Code Injection') - (94)-B 68](#_Toc121248694)

[SANS: Top 25 Errors 2022 69](#_Toc121248695)

[Chapter 7: Mobile Application Security 71](#_Toc121248696)

[Enforce Strong Authentication 72](#_Toc121248697)

[Encrypt Mobile Communications 72](#_Toc121248698)

[Patch App and Operating System Vulnerabilities 72](#_Toc121248699)

[Protect Against Device Theft 73](#_Toc121248700)

[Scan Mobile Apps for Malware 73](#_Toc121248701)

[Perform VAPT 73](#_Toc121248702)

[Protect app data on your device 73](#_Toc121248703)

[Secure the Platform 74](#_Toc121248704)

[Prevent Data Leaks 74](#_Toc121248705)

[For ironclad data leak prevention of confidential data: 74](#_Toc121248706)

[Optimise Data Caching‌‌ 74](#_Toc121248707)

[Isolate Application Information‌‌ 74](#_Toc121248708)

[JSON Web Token Cheat Sheet for Java 75](#_Toc121248709)

[Token Structure 75](#_Toc121248710)

[Objective 76](#_Toc121248711)

[Consideration about Using JWT 76](#_Toc121248712)

[Reference 77](#_Toc121248713)

[Index 79](#_Toc121248714)

# Prefix

This document is intended for developers, scrum masters, software engineers and security experts who are responsible for ensuring their applications are secure from the beginning.

Security is crucial in this day and age; hence it is vital to embed security into the process of writing code. Some security measures can be considered after security testing. On the other hand, other measures can be considered when starting to develop software.

This document explains what to give attention to when developing software to alleviate security issues and vulnerabilities.

In this document, you will find standards for software secure development from different vendors like Microsoft, OWASP top 10 and for different languages .Net, SQL, Android and IOS.

In addition to this, it contains top 25 vulnerabilities and software error that impose security threats to code from various platforms. i. e. CWE, SANS.

# Chapter 1: dotNet Microsoft Application Security

## Introduction

This section contains best practice for .Net applications.

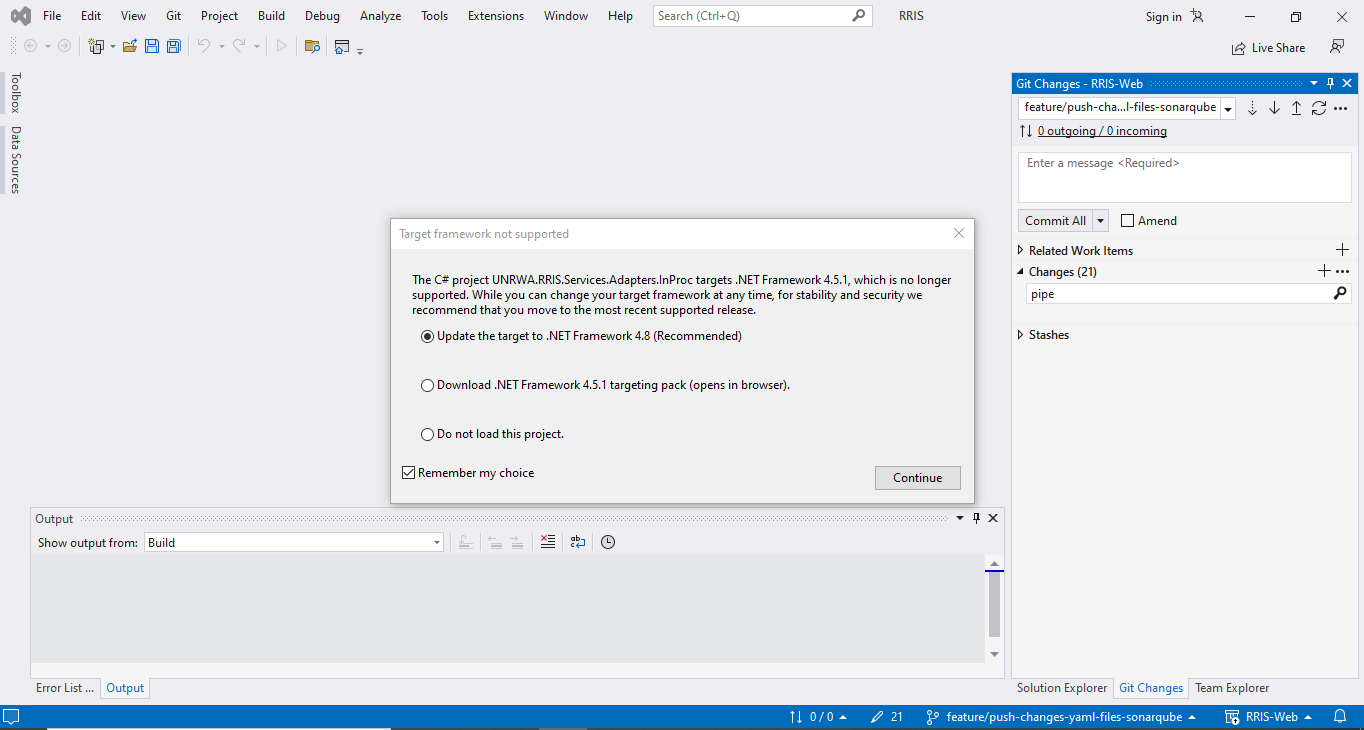
## Updating the Framework

The .NET Framework is kept up-to-date by Microsoft with the Windows Update service. Developers do not normally need to run separate updates to the Framework. Windows Update can be accessed at Windows Update or from the Windows Update program on a Windows computer.

Individual frameworks can be kept up to date using NuGet. As Visual Studio prompts for updates, build it into your lifecycle.

## Always utilize the latest version

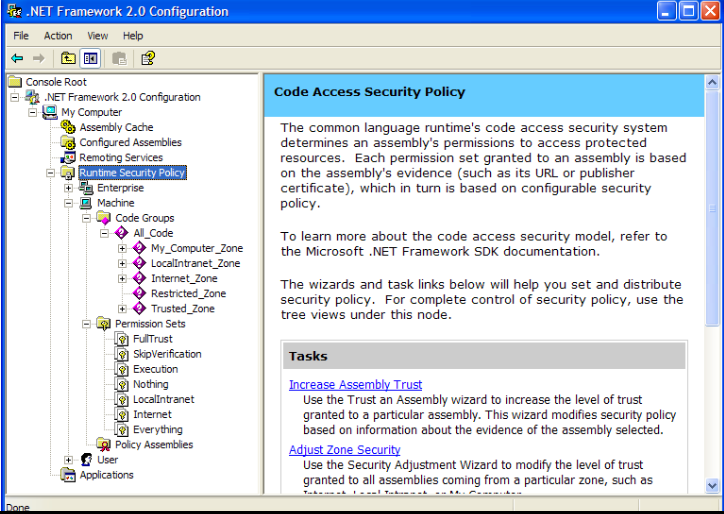
The current version of .Net framework is 4.5.1, which shall be upgraded to .Net framework version 4.8 since 4.5.1 reached EOS.



## Adjust .NET Framework security on a zone-by-zone basis

The .NET Framework assigns trust levels to managed assemblies. These assignments are based, in part, on the zone where the assembly runs. The standard zones are My Computer, Local Intranet, Internet, Trusted Sites, and Untrusted Sites. You may have to increase or reduce the trust level that is associated with one of these zones. The .NET Framework includes tools for adjusting these settings.

* To access this tool “.Net framework configuration tool”, go to Run= mscorcfg.msc
* Or you can find it in this path: %systemroot%\Microsoft.Net\Framework\



## Adjust the level of trust in a .NET Framework assembly

The .NET Framework includes many ways to determine the level of trust that you should grant to an assembly. However, you can make exceptions to the rules to enable a specific assembly to receive a higher level of trust than it would typically receive based on the evidence provided to the common language runtime. The .NET Framework provides a wizard tool specifically for this purpose.

These tools are automatically installed with Visual Studio. For more information about changing

Trust level, go to this link:

<https://learn.microsoft.com/en-us/previous-versions/aspnet/wyts434y(v=vs.100)>

## Restore policy levels that have been customized

As an Administrator, you have complete control over the access that you grant to assemblies that run at the various trust levels. If you customize trust levels, you may experience problems when you run an application that typically runs under a standard trust level. However, you can quickly restore policy levels to their default settings.

## Evaluate the permissions that are granted to an assembly

When you have enterprise, machine, and user security configuration policies, and customizable trust levels, it can be difficult to assess the permissions that have been granted to a managed assembly. The .NET Framework Configuration tool includes a simple method to evaluate these permissions.

## Audit the security of .NET-connected applications

During upgrades, testing, and troubleshooting, the configuration of production systems may change in unintentional ways. For example, an administrator might grant administrative credentials to a user while determining whether an error is related to access rights. If that administrator forgets to revoke those elevated credentials after completing the troubleshooting process, the integrity of the system is compromised.

Because system security can be degraded over time by this type of action, it's a good idea to do regular audits. To do it, document key aspects of a pristine system to create a baseline measure. Compare these settings against the baseline over time to determine if any problems have developed that might significantly reduce the level of vulnerability.

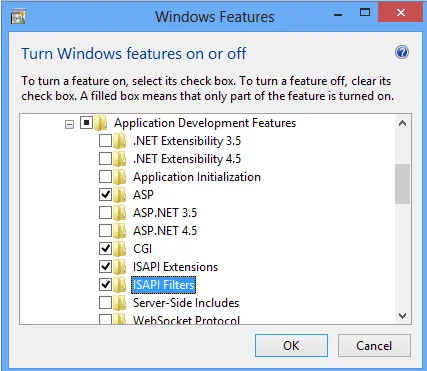
## Configure a .NET-connected application and SQL Server to use an alternate port number for network communications

Many automated tools identify available services and vulnerabilities by querying well-known port numbers. These tools include both legitimate security assessment tools and tools that malicious users might use.

One way to reduce the exposure to these types of tools is to change the port number that the applications use. You can apply this method to .NET-connected applications that rely on a back-end SQL Server database. This method works if both the server and the client are correctly configured.

## Lock down an ASP.NET web application or web service

There are many ways to increase the security of ASP.NET web applications and web services. For example, you can use packet filtering, firewalls, restrictive file permissions, the URL Scan Internet Server Application Programming Interface (ISAPI) filter, and carefully controlled SQL Server privileges. It's a good idea to review these different methods to provide security-in-depth for ASP.NET applications.



## Configure NTFS file permissions to increase security of ASP.NET applications

New Technology File System (NTFS) file permissions continue to be an important layer of security for web applications. ASP.NET applications include many more file types than previous web application environments. The files that anonymous user accounts must have access to isn't obvious.

## Configure SQL Server security for applications that are built on the .NET Framework

By default, SQL Server doesn't grant users the ability to query or update databases. This rule also applies to ASP.NET applications and the ASPNET user account. To enable ASP.NET applications to gain access to data that is stored in a SQL Server database, the database administrator must grant rights to the ASPNET account.

For additional information about how to configure SQL Server to allow queries and updates from ASP.NET applications, visit [Configure permissions on database objects](https://learn.microsoft.com/en-us/sql/t-sql/lesson-2-configuring-permissions-on-database-objects?view=sql-server-ver15&preserve-view=true).

## Configure URLScan to increase protection of ASP.NET web applications

When you install URLScan on an Internet Information Services 5.0 (IIS 5.0) server, it's configured to allow ASP 3.0 applications to run. However, when you install the .NET Framework, the URLScan configuration isn't updated to include the new ASP.NET file types. If you want the added security of the URLScan ISAPI filter for your ASP.NET applications, adjust the URLScan configuration.

## Require authentication for ASP.NET web applications

Many ASP.NET applications don't allow anonymous access. An ASP.NET application that requires authentication can use one of the following three methods: Forms authentication, .NET Passport authentication, and Windows authentication. Each authentication method requires a different configuration technique.

## Restrict specific users from gaining access to specified web resources

ASP.NET includes Forms authentication. It's a unique way to authenticate users without creating Windows accounts. ASP.NET also includes the ability to grant or deny these users' access to different web resources.

For more information about how to control access to web resources on a per-user basis, visit [How To Restrict Specific Users from Gaining Access to Specified web Resources](https://support.microsoft.com/help/815151).

## Limit the web services protocols that a server permits

By default, ASP.NET supports three ways for web services clients to issue requests to web services: SOAP, HTTP GET, and HTTP PUT. However, most applications require only one of these three methods. It's a good idea to reduce the attack surface by disabling any unused protocols.

## Don't permit browser access to .NET-connected web services

ASP.NET web services provide a browser-friendly interface to make it easier for developers to create web services clients. This friendly interface permits anyone who can reach the web service to view the complete details of the methods that are available and any required parameters. This access is useful for public web services that include only publicly available methods. However, it may reduce the security of private web services.

For additional information about how to control access to web resources on a per-user basis, visit [How To Restrict Specific Users from Gaining Access to Specified web Resources](https://support.microsoft.com/help/815151).

## Use ASP.NET to protect file types

The structure of ASP.NET applications causes many private files to be stored with files that end-users request. ASP.NET protects these files by intercepting requests for the files and returning an error. You can extend this type of protection to any file type by using configuration settings. If your application includes unusual file types that should remain private, you can use ASP.NET file protection to protect those files.

## Upgrade Framework

It is crucial to regularly update .Net framework. Updates provide security patches, bug fixes, and more features. Therefore, it is extremely recommended to update .Net framework whenever an update is released.

# Chapter 2: OWASP ASVS

## Introduction

This page intends to provide quick basic .NET security tips for developers.

## The .NET Framework

The .NET Framework is Microsoft's principal platform for enterprise development. It is the supporting API for ASP.NET, Windows Desktop applications, Windows Communication Foundation services, SharePoint, Visual Studio Tools for Office and other technologies.

## Updating the Framework

The .NET Framework is kept up-to-date by Microsoft with the Windows Update service. Developers do not normally need to run separate updates to the Framework. Windows Update can be accessed at [Windows Update](http://windowsupdate.microsoft.com/) or from the Windows Update program on a Windows computer.

Individual frameworks can be kept up to date using [NuGet](https://docs.microsoft.com/en-us/nuget/). As Visual Studio prompts for updates, build it into your lifecycle.

Remember that third-party libraries have to be updated separately and not all of them use NuGet. ELMAH for instance, requires a separate update effort.

Security Announcements

Receive security notifications by selecting the "Watch" button at the following repositories:

* [.NET Core Security Announcements](https://github.com/dotnet/announcements/issues?q=is%3Aopen+is%3Aissue+label%3ASecurity)
* [ASP.NET Core & Entity Framework Core Security Announcements](https://github.com/aspnet/Announcements/issues?q=is%3Aopen+is%3Aissue+label%3ASecurity)

## .NET Framework Guidance

The .NET Framework is the set of APIs that support an advanced type system, data, graphics, network, file handling and most of the rest of what is needed to write enterprise apps in the Microsoft ecosystem. It is a nearly ubiquitous library that is strongly named and versioned at the assembly level.

## Data Access

* Use [Parameterized SQL](https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand.prepare?view=netframework-4.7.2) commands for all data access, without exception.
* Do not use [SqlCommand](https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand) with a string parameter made up of a [concatenated SQL String](https://docs.microsoft.com/en-gb/visualstudio/code-quality/ca2100-review-sql-queries-for-security-vulnerabilities?view=vs-2017).
* List allowable values coming from the user. Use enums, [TryParse](https://docs.microsoft.com/en-us/dotnet/api/system.int32.tryparse#System_Int32_TryParse_System_String_System_Int32__) or lookup values to assure that the data coming from the user is as expected.
  + Enums are still vulnerable to unexpected values because .NET only validates a successful cast to the underlying data type, integer by default. [Enum.IsDefined](https://docs.microsoft.com/en-us/dotnet/api/system.enum.isdefined) can validate whether the input value is valid within the list of defined constants.
* Apply the principle of least privilege when setting up the Database User in your database of choice. The database user should only be able to access items that make sense for the use case.
* Use of the [Entity Framework](https://docs.microsoft.com/en-us/ef/) is a very effective [SQL injection](https://owasp.org/www-community/attacks/SQL_Injection) prevention mechanism. Remember that building your own ad hoc queries in Entity Framework is just as susceptible to SQLi as a plain SQL query.
* When using SQL Server, prefer [integrated authentication](https://docs.microsoft.com/en-us/sql/connect/odbc/linux-mac/using-integrated-authentication?view=sql-server-2017) over [SQL authentication](https://docs.microsoft.com/en-us/sql/relational-databases/security/choose-an-authentication-mode?view=sql-server-2017#connecting-through-sql-server-authentication).
* Use [Always Encrypted](https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-database-engine) where possible for sensitive data (SQL Server 2016 and SQL Azure),

## Encryption

* Never, ever write your own encryption.
* Use the [Windows Data Protection API (DPAPI)](https://docs.microsoft.com/en-us/dotnet/standard/security/how-to-use-data-protection) for secure local storage of sensitive data.
* Use a strong hash algorithm.
  + In .NET (both Framework and Core) the strongest hashing algorithm for general hashing requirements is [System.Security.Cryptography.SHA512](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.sha512).
  + In the .NET framework the strongest algorithm for password hashing is PBKDF2, implemented as [System.Security.Cryptography.Rfc2898DeriveBytes](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rfc2898derivebytes).
  + In .NET Core the strongest algorithm for password hashing is PBKDF2, implemented as [Microsoft.AspNetCore.Cryptography.KeyDerivation.Pbkdf2](https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/consumer-apis/password-hashing) which has several significant advantages over Rfc2898DeriveBytes.
  + When using a hashing function to hash non-unique inputs such as passwords, use a salt value added to the original value before hashing.
  + To activate DPAPI, you need to use ProtectData Class as described in the following link:

<https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.protecteddata?view=dotnet-plat-ext-7.0>

## HTTP validation and encoding

* Do not disable [validateRequest](https://www.asp.net/whitepapers/request-validation) in the web.config or the page setup. This value enables limited XSS protection in ASP.NET and should be left intact as it provides partial prevention of Cross Site Scripting. Complete request validation is recommended in addition to the built-in protections.
* To enable/disable validateRequest, you can follow the following link:

<https://learn.microsoft.com/en-us/aspnet/whitepapers/request-validation>

## Forms authentication

* Use cookies for persistence when possible. Cookieless auth will default to [UseDeviceProfile](https://docs.microsoft.com/en-us/dotnet/api/system.web.httpcookiemode?view=netframework-4.7.2).
* Don't trust the URI of the request for persistence of the session or authorization. It can be easily faked.
* Reduce the forms authentication timeout from the default of *20 minutes* to the shortest period appropriate for your application. If [slidingExpiration](https://docs.microsoft.com/en-us/dotnet/api/system.web.security.formsauthentication.slidingexpiration?view=netframework-4.7.2) is used this timeout resets after each request, so active users won't be affected.
* If HTTPS is not used, [slidingExpiration](https://docs.microsoft.com/en-us/dotnet/api/system.web.security.formsauthentication.slidingexpiration?view=netframework-4.7.2) should be disabled. Consider disabling [slidingExpiration](https://docs.microsoft.com/en-us/dotnet/api/system.web.security.formsauthentication.slidingexpiration?view=netframework-4.7.2) even with HTTPS.
* Always implement proper access controls.
  + Compare user provided username with User.Identity.Name.
  + Check roles against User.Identity.IsInRole.
* Use the [ASP.NET Membership provider and role provider](https://docs.microsoft.com/en-us/dotnet/framework/wcf/samples/membership-and-role-provider), but review the password storage. The default storage hashes the password with a single iteration of SHA-1 which is rather weak. The ASP.NET MVC4 template uses [ASP.NET Identity](https://www.asp.net/identity/overview/getting-started/introduction-to-aspnet-identity) instead of ASP.NET Membership, and ASP.NET Identity uses PBKDF2 by default which is better. Review the OWASP [Password Storage Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html) for more information.
* Explicitly authorize resource requests.
* Leverage role based authorization using User.Identity.IsInRole.

## ASP NET MVC Guidance

ASP.NET MVC (Model–View–Controller) is a contemporary web application framework that uses more standardized HTTP communication than the Web Forms postback model.

The OWASP Top 10 2017 lists the most prevalent and dangerous threats to web security in the world today and is reviewed every 3 years.

This section is based on this. Your approach to securing your web application should be to start at the top threat A1 below and work down, this will ensure that any time spent on security will be spent most effectively spent and cover the top threats first and lesser threats afterwards. After covering the top 10 it is generally advisable to assess for other threats or get a professionally completed Penetration Test.

## A1 Injection

### SQL Injection

DO: Using an object relational mapper (ORM) or stored procedures is the most effective way of countering the SQL Injection vulnerability.

DO: Use parameterized queries where a direct sql query must be used. More Information can be found [here](https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html).

e.g. In entity frameworks:

var sql = @"Update [User] SET FirstName = @FirstName WHERE Id = @Id";

context.Database.ExecuteSqlCommand(

sql,

new SqlParameter("@FirstName", firstname),

new SqlParameter("@Id", id));

DO NOT: Concatenate strings anywhere in your code and execute them against your database (Known as dynamic sql).

NB: You can still accidentally do this with ORMs or Stored procedures so check everywhere.

e.g

string strQry = "SELECT \* FROM Users WHERE UserName='" + txtUser.Text + "' AND Password='"

+ txtPassword.Text + "'";

EXEC strQry // SQL Injection vulnerability!

DO: Practice Least Privilege - Connect to the database using an account with a minimum set of permissions required to do it's job i.e. not the sa account

### OS Injection

General guidance about OS Injection can be found on this [cheat sheet](https://cheatsheetseries.owasp.org/cheatsheets/OS_Command_Injection_Defense_Cheat_Sheet.html).

DO: Use [System.Diagnostics.Process.Start](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.start?view=netframework-4.7.2) to call underlying OS functions.

e.g

var process = new System.Diagnostics.Process();

var startInfo = new System.Diagnostics.ProcessStartInfo();

startInfo.FileName = "validatedCommand";

startInfo.Arguments = "validatedArg1 validatedArg2 validatedArg3";

process.StartInfo = startInfo;

process.Start();

DO NOT: Assume that this mechanism will protect against malicious input designed to break out of one argument and then tamper with another argument to the process. This will still be possible.

DO: Use allow-list validation on all user supplied input wherever possible. Input validation prevents improperly formed data from entering an information system. For more information please see the [Input Validation Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html).

e.g Validating user input using [IPAddress.TryParse Method](https://docs.microsoft.com/en-us/dotnet/api/system.net.ipaddress.tryparse?view=netframework-4.8)

//User input

string ipAddress = "127.0.0.1";

//check to make sure an ip address was provided

if (!string.IsNullOrEmpty(ipAddress))

{

// Create an instance of IPAddress for the specified address string (in

// dotted-quad, or colon-hexadecimal notation).

if (IPAddress.TryParse(ipAddress, out var address))

{

// Display the address in standard notation.

return address.ToString();

}

else

{

//ipAddress is not of type IPAddress

...

}

...

}

DO: Try to only accept characters which are simple alphanumeric.

DO NOT: Assume you can sanitize special characters without actually removing them. Various combinations of \, ' and @ may have an unexpected impact on sanitization attempts.

DO NOT: Rely on methods without a security guarantee.

e.g. .NET Core 2.2 and greater and .NET 5 and greater support [ProcessStartInfo.ArgumentList](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.argumentlist) which performs some character escaping but it is not clear if this is guaranteed to be secure.

DO: Look at alternatives to passing raw untrusted arguments via command-line parameters such as encoding using Base64 (which would safely encode any special characters as well) and then decode the parameters in the receiving application.

### LDAP injection

Almost any characters can be used in Distinguished Names. However, some must be escaped with the backslash \ escape character. A table showing which characters that should be escaped for Active Directory can be found at the in the [LDAP Injection Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/LDAP_Injection_Prevention_Cheat_Sheet.html#introduction).

NB: The space character must be escaped only if it is the leading or trailing character in a component name, such as a Common Name. Embedded spaces should not be escaped.

More information can be found [here](https://cheatsheetseries.owasp.org/cheatsheets/LDAP_Injection_Prevention_Cheat_Sheet.html#introduction).

## A2 Broken Authentication

DO: Use [ASP.net Core Identity](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-2.2&). ASP.net Core Identity framework is well configured by default, where it uses secure password hashes and an individual salt. Identity uses the PBKDF2 hashing function for passwords, and they generate a random salt per user.

DO: Set secure password policy

e.g ASP.net Core Identity

//startup.cs

services.Configure<IdentityOptions>(options =>

{

// Password settings

options.Password.RequireDigit = true;

options.Password.RequiredLength = 8;

options.Password.RequireNonAlphanumeric = true;

options.Password.RequireUppercase = true;

options.Password.RequireLowercase = true;

options.Password.RequiredUniqueChars = 6;

options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);

options.Lockout.MaxFailedAccessAttempts = 3;

options.SignIn.RequireConfirmedEmail = true;

options.User.RequireUniqueEmail = true;

});

DO: Set a cookie policy

e.g

//startup.cs

services.ConfigureApplicationCookie(options =>

{

options.Cookie.HttpOnly = true;

options.Cookie.Expiration = TimeSpan.FromHours(1)

options.SlidingExpiration = true;

});

## A3 Sensitive Data Exposure

DO NOT: [Store encrypted passwords](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#do-not-limit-the-character-set-and-set-long-max-lengths-for-credentials).

DO: Use a strong hash to store password credentials. For hash refer to [this section](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#guidance).

DO: Enforce passwords with a minimum complexity that will survive a dictionary attack i.e. longer passwords that use the full character set (numbers, symbols and letters) to increase the entropy.

DO: Use a strong encryption routine such as AES-512 where personally identifiable data needs to be restored to it's original format. Protect encryption keys more than any other asset, please find [more information of storing encryption keys at rest](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#guidance). Apply the following test: Would you be happy leaving the data on a spreadsheet on a bus for everyone to read. Assume the attacker can get direct access to your database and protect it accordingly. More information can be found [here](https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html).

DO: Use TLS 1.2 for your entire site. Get a free certificate [LetsEncrypt.org](https://letsencrypt.org/).

DO NOT: [Allow SSL, this is now obsolete](https://github.com/ssllabs/research/wiki/SSL-and-TLS-Deployment-Best-Practices).

DO: Have a strong TLS policy (see [SSL Best Practices](https://www.ssllabs.com/projects/best-practices/index.html)), use TLS 1.2 wherever possible. Then check the configuration using [SSL Test](https://www.ssllabs.com/ssltest/) or [TestSSL](https://testssl.sh/).

DO: Ensure headers are not disclosing information about your application. See [HttpHeaders.cs](https://github.com/johnstaveley/SecurityEssentials/blob/master/SecurityEssentials/Core/HttpHeaders.cs) , [Dionach StripHeaders](https://github.com/Dionach/StripHeaders/), disable via web.config or [startup.cs](https://medium.com/bugbountywriteup/security-headers-1c770105940b):

More information on Transport Layer Protection can be found [here](https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html). e.g Web.config

<system.web>

<httpRuntime enableVersionHeader="false"/>

</system.web>

<system.webServer>

<security>

<requestFiltering removeServerHeader="true" />

</security>

<httpProtocol>

<customHeaders>

<add name="X-Content-Type-Options" value="nosniff" />

<add name="X-Frame-Options" value="DENY" />

<add name="X-Permitted-Cross-Domain-Policies" value="master-only"/>

<add name="X-XSS-Protection" value="0"/>

<remove name="X-Powered-By"/>

</customHeaders>

</httpProtocol>

</system.webServer>

e.g Startup.cs

app.UseHsts(hsts => hsts.MaxAge(365).IncludeSubdomains());

app.UseXContentTypeOptions();

app.UseReferrerPolicy(opts => opts.NoReferrer());

app.UseXXssProtection(options => options.FilterDisabled());

app.UseXfo(options => options.Deny());

app.UseCsp(opts => opts

.BlockAllMixedContent()

.StyleSources(s => s.Self())

.StyleSources(s => s.UnsafeInline())

.FontSources(s => s.Self())

.FormActions(s => s.Self())

.FrameAncestors(s => s.Self())

.ImageSources(s => s.Self())

.ScriptSources(s => s.Self())

);

For more information about headers can be found [here](https://owasp.org/www-project-secure-headers/).

## A4 XML External Entities (XXE)

XXE attacks occur when an XML parse does not properly process user input that contains external entity declaration in the doctype of an XML payload.

[This article](https://docs.microsoft.com/en-us/dotnet/standard/data/xml/xml-processing-options) discusses the most common XML Processing Options for .NET.

Please refer to the [XXE cheat sheet](https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html#net) for more detailed information on preventing XXE and other XML Denial of Service attacks.

## A5 Broken Access Control

### Weak Account management

Ensure cookies are sent via httpOnly:

CookieHttpOnly = true,

Reduce the time period a session can be stolen in by reducing session timeout and removing sliding expiration:

ExpireTimeSpan = TimeSpan.FromMinutes(60),

SlidingExpiration = false

See [here](https://github.com/johnstaveley/SecurityEssentials/blob/master/SecurityEssentials/App_Start/Startup.Auth.cs) for full startup code snippet

Ensure cookie is sent over HTTPS in the production environment. This should be enforced in the config transforms:

<httpCookies requireSSL="true" xdt:Transform="SetAttributes(requireSSL)"/>

<authentication>

<forms requireSSL="true" xdt:Transform="SetAttributes(requireSSL)"/>

</authentication>

Protect LogOn, Registration and password reset methods against brute force attacks by throttling requests (see code below), consider also using ReCaptcha.

[HttpPost]

[AllowAnonymous]

[ValidateAntiForgeryToken]

[AllowXRequestsEveryXSecondsAttribute(Name = "LogOn",

Message = "You have performed this action more than {x} times in the last {n} seconds.",

Requests = 3, Seconds = 60)]

public async Task<ActionResult> LogOn(LogOnViewModel model, string returnUrl)

DO NOT: Roll your own authentication or session management, use the one provided by .Net

DO NOT: Tell someone if the account exists on LogOn, Registration or Password reset. Say something like 'Either the username or password was incorrect', or 'If this account exists then a reset token will be sent to the registered email address'. This protects against account enumeration.

The feedback to the user should be identical whether or not the account exists, both in terms of content and behavior: e.g. if the response takes 50% longer when the account is real then membership information can be guessed and tested.

### Missing function-level access control

DO: Authorize users on all externally facing endpoints. The .NET framework has many ways to authorize a user, use them at method level:

[Authorize(Roles = "Admin")]

[HttpGet]

public ActionResult Index(int page = 1)

or better yet, at controller level:

[Authorize]

public class UserController

You can also check roles in code using identity features in .net: System.Web.Security.Roles.IsUserInRole(userName, roleName)

You can find more information [here](https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html#introduction) on Access Control and [here](https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Testing_Automation_Cheat_Sheet.html) for Authorization.

### Insecure Direct object references

When you have a resource (object) which can be accessed by a reference (in the sample below this is the id) then you need to ensure that the user is intended to be there

// Insecure

public ActionResult Edit(int id)

{

var user = \_context.Users.FirstOrDefault(e => e.Id == id);

return View("Details", new UserViewModel(user);

}

// Secure

public ActionResult Edit(int id)

{

var user = \_context.Users.FirstOrDefault(e => e.Id == id);

// Establish user has right to edit the details

if (user.Id != \_userIdentity.GetUserId())

{

HandleErrorInfo error = new HandleErrorInfo(

new Exception("INFO: You do not have permission to edit these details"));

return View("Error", error);

}

return View("Edit", new UserViewModel(user);

}

More information can be found [here](https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html) for Insecure Direct Object Reference.

## A6 Security Misconfiguration

### Debug and Stack Trace

Ensure debug and trace are off in production. This can be enforced using web.config transforms:

<compilation xdt:Transform="RemoveAttributes(debug)" />

<trace enabled="false" xdt:Transform="Replace"/>

DO NOT: Use default passwords

DO: (When using TLS) Redirect a request made over Http to https:

e.g Global.asax.cs

protected void Application\_BeginRequest()

{

#if !DEBUG

// SECURE: Ensure any request is returned over SSL/TLS in production

if (!Request.IsLocal && !Context.Request.IsSecureConnection) {

var redirect = Context.Request.Url.ToString()

.ToLower(CultureInfo.CurrentCulture)

.Replace("http:", "https:");

Response.Redirect(redirect);

}

#endif

}

e.g Startup.cs in the Configure()

app.UseHttpsRedirection();

### Cross-site request forgery

DO NOT: Send sensitive data without validating Anti-Forgery-Tokens ([.NET](https://docs.microsoft.com/en-us/aspnet/web-api/overview/security/preventing-cross-site-request-forgery-csrf-attacks) / [.NET Core](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-3.0#aspnet-core-antiforgery-configuration)).

DO: Send the anti-forgery token with every POST/PUT request:

#### USING .NET FRAMEWORK

using (Html.BeginForm("LogOff", "Account", FormMethod.Post, new { id = "logoutForm",

@class = "pull-right" }))

{

@Html.AntiForgeryToken()

<ul class="nav nav-pills">

<li role="presentation">

Logged on as @User.Identity.Name

</li>

<li role="presentation">

<a href="javascript:document.getElementById('logoutForm').submit()">Log off</a>

</li>

</ul>

}

Then validate it at the method or preferably the controller level:

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult LogOff()

Make sure the tokens are removed completely for invalidation on logout.

/// <summary>

/// SECURE: Remove any remaining cookies including Anti-CSRF cookie

/// </summary>

public void RemoveAntiForgeryCookie(Controller controller)

{

string[] allCookies = controller.Request.Cookies.AllKeys;

foreach (string cookie in allCookies)

{

if (controller.Response.Cookies[cookie] != null &&

cookie == "\_\_RequestVerificationToken")

{

controller.Response.Cookies[cookie].Expires = DateTime.Now.AddDays(-1);

}

}

}

#### USING .NET CORE 2.0 OR LATER

Starting with .NET Core 2.0 it is possible to [automatically generate and verify the antiforgery token](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-3.0#aspnet-core-antiforgery-configuration).

If you are using [tag-helpers](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro), which is the default for most web project templates, then all forms will automatically send the anti-forgery token. You can check if tag-helpers are enabled by checking if your main \_ViewImports.cshtml file contains:

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

IHtmlHelper.BeginForm also sends anti-forgery-tokens automatically.

Unless you are using tag-helpers or IHtmlHelper.BeginForm, you must use the requisite helper on forms as seen here:

<form action="RelevantAction" >

@Html.AntiForgeryToken()

</form>

To automatically validate all requests other than GET, HEAD, OPTIONS and TRACE you need to add a global action filter with the [AutoValidateAntiforgeryToken](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.autovalidateantiforgerytokenattribute?view=aspnetcore-2.2) attribute inside your Startup.cs as mentioned in the following [article](https://andrewlock.net/automatically-validating-anti-forgery-tokens-in-asp-net-core-with-the-autovalidateantiforgerytokenattribute/):

services.AddMvc(options =>

{

options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute());

});

If you need to disable the attribute validation for a specific method on a controller you can add the [IgnoreAntiforgeryToken](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.ignoreantiforgerytokenattribute?view=aspnetcore-2.2) attribute to the controller method (for MVC controllers) or parent class (for Razor pages):

[IgnoreAntiforgeryToken]

[HttpDelete]

public IActionResult Delete()

[IgnoreAntiforgeryToken]

public class UnsafeModel : PageModel

If you need to also validate the token on GET, HEAD, OPTIONS or TRACE - requests you can add the [ValidateAntiforgeryToken](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.validateantiforgerytokenattribute?view=aspnetcore-2.2) attribute to the controller method (for MVC controllers) or parent class (for Razor pages):

[HttpGet]

[ValidateAntiforgeryToken]

public IActionResult DoSomethingDangerous()

[HttpGet]

[ValidateAntiforgeryToken]

public class SafeModel : PageModel

In case you can't use a global action filter, add the [AutoValidateAntiforgeryToken](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.autovalidateantiforgerytokenattribute?view=aspnetcore-2.2) attribute to your controller classes or razor page models:

[AutoValidateAntiforgeryToken]

public class UserController

[AutoValidateAntiforgeryToken]

public class SafeModel : PageModel

#### USING .NET CORE 2.0 OR .NET FRAMEWORK WITH AJAX

You will need to attach the anti-forgery token to AJAX requests.

If you are using jQuery in an ASP.NET Core MVC view this can be achieved using this snippet:

@inject  Microsoft.AspNetCore.Antiforgery.IAntiforgery antiforgeryProvider

$.ajax(

{

type: "POST",

url: '@Url.Action("Action", "Controller")',

contentType: "application/x-www-form-urlencoded; charset=utf-8",

data: {

id: id,

'\_\_RequestVerificationToken': '@antiforgeryProvider.GetAndStoreTokens(this.Context).RequestToken'

}

})

If you are using the .NET Framework, you can find some code snippets [here](https://docs.microsoft.com/en-us/aspnet/web-api/overview/security/preventing-cross-site-request-forgery-csrf-attacks#anti-csrf-and-ajax).

More information can be found [here](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html) for Cross-Site Request Forgery.

## A7 Cross-Site Scripting (XSS)

DO NOT: Trust any data the user sends you, prefer allow lists (always safe) over block lists

You get encoding of all HTML content with MVC3, to properly encode all content whether HTML, javascript, CSS, LDAP etc use the Microsoft AntiXSS library:

Install-Package AntiXSS

Then set in config:

<system.web>

<httpRuntime targetFramework="4.5"

enableVersionHeader="false"

encoderType="Microsoft.Security.Application.AntiXssEncoder, AntiXssLibrary"

maxRequestLength="4096" />

DO NOT: Use the [AllowHTML] attribute or helper class @Html.Raw unless you really know that the content you are writing to the browser is safe and has been escaped properly.

DO: Enable a [Content Security Policy](https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html#context), this will prevent your pages from accessing assets it should not be able to access (e.g. a malicious script):

<system.webServer>

<httpProtocol>

<customHeaders>

<add name="Content-Security-Policy"

value="default-src 'none'; style-src 'self'; img-src 'self';

font-src 'self'; script-src 'self'" />

More information can be found [here](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html) for Cross-Site Scripting.

## A8 Insecure Deserialization

Information about Insecure Deserialization can be found on this [cheat sheet](https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html#net-csharp).

DO NOT: Accept Serialized Objects from Untrusted Sources

DO: Validate User Input Malicious users are able to use objects like cookies to insert malicious information to change user roles. In some cases, hackers are able to elevate their privileges to administrator rights by using a pre-existing or cached password hash from a previous session.

DO: Prevent Deserialization of Domain Objects

DO: Run the Deserialization Code with Limited Access Permissions If a deserialized hostile object tries to initiate a system processes or access a resource within the server or the host's OS, it will be denied access and a permission flag will be raised so that a system administrator is made aware of any anomalous activity on the server.

More information can be found here: [Deserialization Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html#net-csharp)

## A9 Using Components with Known Vulnerabilities

DO: Keep the .Net framework updated with the latest patches

DO: Keep your [NuGet](https://docs.microsoft.com/en-us/nuget/) packages up to date, many will contain their own vulnerabilities.

DO: Run the [OWASP Dependency Checker](https://cheatsheetseries.owasp.org/cheatsheets/Vulnerable_Dependency_Management_Cheat_Sheet.html) against your application as part of your build process and act on any high level vulnerabilities.

## A10 Insufficient Logging & Monitoring

DO: Ensure all login, access control failures and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts.

DO: Establish effective monitoring and alerting so suspicious activities are detected and responded to in a timely fashion.

DO NOT: Log generic error messages such as: csharp Log.Error("Error was thrown"); rather log the stack trace, error message and user ID who caused the error.

DO NOT: Log sensitive data such as user's passwords.

### Logging

What Logs to Collect and more information about Logging can be found on this [cheat sheet](https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html).

.NET Core come with a LoggerFactory, which is in Microsoft.Extensions.Logging. More information about ILogger can be found [here](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.logging.ilogger).

How to log all errors from the Startup.cs, so that anytime an error is thrown it will be logged.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

\_isDevelopment = true;

app.UseDeveloperExceptionPage();

}

//Log all errors in the application

app.UseExceptionHandler(errorApp =>

{

errorApp.Run(async context =>

{

var errorFeature = context.Features.Get<IExceptionHandlerFeature>();

var exception = errorFeature.Error;

Log.Error(String.Format("Stacktrace of error: {0}",exception.StackTrace.ToString()));

});

});

app.UseAuthentication();

app.UseMvc();

}

}

e.g Injecting into the class constructor, which makes writing unit test simpler. It is recommended if instances of the class will be created using dependency injection (e.g. MVC controllers). The below example shows logging of all unsuccessful log in attempts.

public class AccountsController : Controller

{

private ILogger \_Logger;

public AccountsController( ILogger logger)

{

\_Logger = logger;

}

[HttpPost]

[AllowAnonymous]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Login(LoginViewModel model)

{

if (ModelState.IsValid)

{

var result = await \_signInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, lockoutOnFailure: false);

if (result.Succeeded)

{

//Log all successful log in attempts

Log.Information(String.Format("User: {0}, Successfully Logged in", model.Email));

//Code for successful login

}

else

{

//Log all incorrect log in attempts

Log.Information(String.Format("User: {0}, Incorrect Password", model.Email));

}

}

...

}

Logging levels for ILogger are listed below, in order of high to low importance:

### Monitoring

Monitoring allow us to validate the performance and health of a running system through key performance indicators.

In .NET a great option to add monitoring capabilities is [Application Insights](https://docs.microsoft.com/en-us/azure/azure-monitor/app/asp-net-core).

More information about Logging and Monitoring can be found [here](https://github.com/microsoft/code-with-engineering-playbook/tree/master/observability).

### OWASP 2013

Below is vulnerability not discussed in OWASP 2017

## A10 Unvalidated redirects and forwards

A protection against this was introduced in Mvc 3 template. Here is the code:

public async Task<ActionResult> LogOn(LogOnViewModel model, string returnUrl)

{

if (ModelState.IsValid)

{

var logonResult = await \_userManager.TryLogOnAsync(model.UserName, model.Password);

if (logonResult.Success)

{

await \_userManager.LogOnAsync(logonResult.UserName, model.RememberMe);

return RedirectToLocal(returnUrl);

...

private ActionResult RedirectToLocal(string returnUrl)

{

if (Url.IsLocalUrl(returnUrl))

{

return Redirect(returnUrl);

}

else

{

return RedirectToAction("Landing", "Account");

}

}

Other advice:

* Protect against Clickjacking and man in the middle attack from capturing an initial Non-TLS request, set the X-Frame-Options and Strict-Transport-Security (HSTS) headers. Full details [here](https://github.com/johnstaveley/SecurityEssentials/blob/master/SecurityEssentials/Core/HttpHeaders.cs)
* Protect against a man in the middle attack for a user who has never been to your site before. Register for [HSTS preload](https://hstspreload.org/)
* Maintain security testing and analysis on Web API services. They are hidden inside MEV sites, and are public parts of a site that will be found by an attacker. All of the MVC guidance and much of the WCF guidance applies to the Web API.
* [Unvalidated Redirects and Forwards Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html).

More information:

For more information on all of the above and code samples incorporated into a sample MVC5 application with an enhanced security baseline go to [Security Essentials Baseline project](https://github.com/johnstaveley/SecurityEssentials/)

### XAML Guidance

* Work within the constraints of Internet Zone security for your application.
* Use ClickOnce deployment. For enhanced permissions, use permission elevation at runtime or trusted application deployment at install time.

### Windows Forms Guidance

* Use partial trust when possible. Partially trusted Windows applications reduce the attack surface of an application. Manage a list of what permissions your app must use, and what it may use, and then make the request for those permissions declaratively at runtime.
* Use ClickOnce deployment. For enhanced permissions, use permission elevation at runtime or trusted application deployment at install time.

### WCF Guidance

* Keep in mind that the only safe way to pass a request in RESTful services is via HTTP POST, with TLS enabled. GETs are visible in the querystring, and a lack of TLS means the body can be intercepted.
* Avoid [BasicHttpBinding](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.basichttpbinding?view=netframework-4.7.2). It has no default security configuration. Use [WSHttpBinding](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.wshttpbinding?view=netframework-4.7.2) instead.
* Use at least two security modes for your binding. Message security includes security provisions in the headers. Transport security means use of SSL. [TransportWithMessageCredential](https://docs.microsoft.com/en-us/dotnet/framework/wcf/samples/ws-transport-with-message-credential) combines the two.
* Test your WCF implementation with a fuzzer like the [ZAP](https://www.zaproxy.org/).

# Chapter 3: REST API Security Standard OWASP ASVS

## Introduction

[REST](https://en.wikipedia.org/wiki/Representational_state_transfer) (or REpresentational State Transfer) is an architectural style first described in [Roy Fielding](https://en.wikipedia.org/wiki/Roy_Fielding)'s Ph.D. dissertation on [Architectural Styles and the Design of Network-based Software Architectures](https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm).

It evolved as Fielding wrote the HTTP/1.1 and URI specs and has been proven to be well-suited for developing distributed hypermedia applications. While REST is more widely applicable, it is most commonly used within the context of communicating with services via HTTP.

The key abstraction of information in REST is a resource. A REST API resource is identified by a URI, usually a HTTP URL. REST components use connectors to perform actions on a resource by using a representation to capture the current or intended state of the resource and transferring that representation.

The primary connector types are client and server, secondary connectors include cache, resolver and tunnel.

REST APIs are stateless. Stateful APIs do not adhere to the REST architectural style. State in the REST acronym refers to the state of the resource which the API accesses, not the state of a session within which the API is called. While there may be good reasons for building a stateful API, it is important to realize that managing sessions is complex and difficult to do securely.

Stateful services are out of scope of this Cheat Sheet: Passing state from client to backend, while making the service technically stateless, is an anti-pattern that should also be avoided as it is prone to replay and impersonation attacks.

In order to implement flows with REST APIs, resources are typically created, read, updated and deleted. For example, an ecommerce site may offer methods to create an empty shopping cart, to add items to the cart and to check out the cart. Each of these REST calls is stateless and the endpoint should check whether the caller is authorized to perform the requested operation.

Another key feature of REST applications is the use of standard HTTP verbs and error codes in the pursuit or removing unnecessary variation among different services.

Another key feature of REST applications is the use of [HATEOAS or Hypermedia As The Engine of Application State](https://en.wikipedia.org/wiki/HATEOAS). This provides REST applications a self-documenting nature making it easier for developers to interact with a REST service without prior knowledge.

## HTTPS

Secure REST services must only provide HTTPS endpoints. This protects authentication credentials in transit, for example passwords, API keys or JSON Web Tokens. It also allows clients to authenticate the service and guarantees integrity of the transmitted data.

See the [Transport Layer Protection Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html) for additional information.

Consider the use of mutually authenticated client-side certificates to provide additional protection for highly privileged web services.

## Access Control

Non-public REST services must perform access control at each API endpoint. Web services in monolithic applications implement this by means of user authentication, authorization logic and session management. This has several drawbacks for modern architectures which compose multiple microservices following the RESTful style.

* in order to minimize latency and reduce coupling between services, the access control decision should be taken locally by REST endpoints
* user authentication should be centralised in a Identity Provider (IdP), which issues access tokens

## JWT

There seems to be a convergence towards using [JSON Web Tokens](https://tools.ietf.org/html/rfc7519) (JWT) as the format for security tokens. JWTs are JSON data structures containing a set of claims that can be used for access control decisions. A cryptographic signature or message authentication code (MAC) can be used to protect the integrity of the JWT.

* Ensure JWTs are integrity protected by either a signature or a MAC. Do not allow the unsecured JWTs: {"alg":"none"}.
  + See [here](https://tools.ietf.org/html/rfc7519#section-6.1)
* In general, signatures should be preferred over MACs for integrity protection of JWTs.

If MACs are used for integrity protection, every service that is able to validate JWTs can also create new JWTs using the same key. This means that all services using the same key have to mutually trust each other. Another consequence of this is that a compromise of any service also compromises all other services sharing the same key. See [here](https://tools.ietf.org/html/rfc7515#section-10.5) for additional information.

The relying party or token consumer validates a JWT by verifying its integrity and claims contained.

* A relying party must verify the integrity of the JWT based on its own configuration or hard-coded logic. It must not rely on the information of the JWT header to select the verification algorithm. See [here](https://www.chosenplaintext.ca/2015/03/31/jwt-algorithm-confusion.html) and [here](https://www.youtube.com/watch?v=bW5pS4e_MX8%3E)

Some claims have been standardized and should be present in JWT used for access controls. At least the following of the standard claims should be verified:

* iss or issuer - is this a trusted issuer? Is it the expected owner of the signing key?
* aud or audience - is the relying party in the target audience for this JWT?
* exp or expiration time - is the current time before the end of the validity period of this token?
* nbf or not before time - is the current time after the start of the validity period of this token?

As JWTs contain details of the authenticated entity (user etc.) a disconnect can occur between the JWT and the current state of the users session, for example, if the session is terminated earlier than the expiration time due to an explicit logout or an idle timeout. When an explicit session termination event occurs, a digest or hash of any associated JWTs should be submitted to a block list on the API which will invalidate that JWT for any requests until the expiration of the token. See the [JSON\_Web\_Token\_for\_Java\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html#token-explicit-revocation-by-the-user) for further details.

## API Keys

Public REST services without access control run the risk of being farmed leading to excessive bills for bandwidth or compute cycles. API keys can be used to mitigate this risk. They are also often used by organisation to monetize APIs; instead of blocking high-frequency calls, clients are given access in accordance to a purchased access plan.

API keys can reduce the impact of denial-of-service attacks. However, when they are issued to third-party clients, they are relatively easy to compromise.

* Require API keys for every request to the protected endpoint.
* Return 429 Too Many Requests HTTP response code if requests are coming in too quickly.
* Revoke the API key if the client violates the usage agreement.
* Do not rely exclusively on API keys to protect sensitive, critical or high-value resources.

## Restrict HTTP methods

* Apply an allow list of permitted HTTP Methods e.g. GET, POST, PUT.
* Reject all requests not matching the allow list with HTTP response code 405 Method not allowed.
* Make sure the caller is authorised to use the incoming HTTP method on the resource collection, action, and record

In Java EE in particular, this can be difficult to implement properly. See [Bypassing Web Authentication and Authorization with HTTP Verb Tampering](https://cheatsheetseries.owasp.org/assets/REST_Security_Cheat_Sheet_Bypassing_VBAAC_with_HTTP_Verb_Tampering.pdf) for an explanation of this common misconfiguration.

## Input validation

* Do not trust input parameters/objects.
* Validate input: length / range / format and type.
* Achieve an implicit input validation by using strong types like numbers, booleans, dates, times or fixed data ranges in API parameters.
* Constrain string inputs with regexps.
* Reject unexpected/illegal content.
* Make use of validation/sanitation libraries or frameworks in your specific language.
* Define an appropriate request size limit and reject requests exceeding the limit with HTTP response status 413 Request Entity Too Large.
* Consider logging input validation failures. Assume that someone who is performing hundreds of failed input validations per second is up to no good.
* Have a look at input validation cheat sheet for comprehensive explanation.
* Use a secure parser for parsing the incoming messages. If you are using XML, make sure to use a parser that is not vulnerable to [XXE](https://owasp.org/www-community/vulnerabilities/XML_External_Entity_%28XXE%29_Processing) and similar attacks.

## Validate content types

A REST request or response body should match the intended content type in the header. Otherwise, this could cause misinterpretation at the consumer/producer side and lead to code injection/execution.

* Document all supported content types in your API.

### Validate request content types

* Reject requests containing unexpected or missing content type headers with HTTP response status 406 Unacceptable or 415 Unsupported Media Type.
* For XML content types ensure appropriate XML parser hardening, see the [XXE cheat sheet](https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html).
* Avoid accidentally exposing unintended content types by explicitly defining content types e.g. [Jersey](https://jersey.github.io/) (Java) @consumes("application/json"); @produces("application/json"). This avoids [XXE-attack](https://owasp.org/www-community/vulnerabilities/XML_External_Entity_%28XXE%29_Processing) vectors for example.

### Send safe response content types

It is common for REST services to allow multiple response types (e.g. application/xml or application/json, and the client specifies the preferred order of response types by the Accept header in the request.

* Do NOT simply copy the Accept header to the Content-type header of the response.
* Reject the request (ideally with a 406 Not Acceptable response) if the Accept header does not specifically contain one of the allowable types.

Services including script code (e.g. JavaScript) in their responses must be especially careful to defend against header injection attack.

* Ensure sending intended content type headers in your response matching your body content e.g. application/json and not application/javascript.

## Management endpoints

* Avoid exposing management endpoints via Internet.
* If management endpoints must be accessible via the Internet, make sure that users must use a strong authentication mechanism, e.g. multi-factor.
* Expose management endpoints via different HTTP ports or hosts preferably on a different NIC and restricted subnet.
* Restrict access to these endpoints by firewall rules or use of access control lists.

## Error handling

* Respond with generic error messages - avoid revealing details of the failure unnecessarily.
* Do not pass technical details (e.g. call stacks or other internal hints) to the client.

## Audit logs

* Write audit logs before and after security related events.
* Consider logging token validation errors in order to detect attacks.
* Take care of log injection attacks by sanitizing log data beforehand.

## Security Headers

There are a number of [security related headers](https://owasp.org/www-project-secure-headers/) that can be returned in the HTTP responses to instruct browsers to act in specific ways. However, some of these headers are intended to be used with HTML responses, and as such may provide little or no security benefits on an API that does not return HTML.

The following headers should be included in all API responses:

| Header | Rationale |
| --- | --- |
| Cache-Control: no-store | Prevent sensitive information from being cached. |
| Content-Security-Policy: frame-ancestors 'none' | To protect against [drag-and-drop](https://www.w3.org/Security/wiki/Clickjacking_Threats#Drag_and_drop_attacks) style clickjacking attacks. |
| Content-Type | To specify the content type of the response. This should be application/json for JSON responses. |
| Strict-Transport-Security | To require connections over HTTPS and to protect against spoofed certificates. |
| X-Content-Type-Options: nosniff | To prevent browsers from performing MIME sniffing, and inappropriately interpreting responses as HTML. |
| X-Frame-Options: DENY | To protect against drag-and-drop style clickjacking attacks. |

The headers below are only intended to provide additional security when responses are rendered as HTML. As such, if the API will never return HTML in responses, then these headers may not be necessary. However, if there is any uncertainty about the function of the headers, or the types of information that the API returns (or may return in future), then it is recommended to include them as part of a defence-in-depth approach.

| Header | Rationale |
| --- | --- |
| Content-Security-Policy: default-src 'none' | The majority of CSP functionality only affects pages rendered as HTML. |
| Feature-Policy: 'none' | Feature policies only affect pages rendered as HTML. |
| Referrer-Policy: no-referrer | Non-HTML responses should not trigger additional requests. |

## CORS

Cross-Origin Resource Sharing (CORS) is a W3C standard to flexibly specify what cross-domain requests are permitted. By delivering appropriate CORS Headers your REST API signals to the browser which domains, AKA origins, are allowed to make JavaScript calls to the REST service.

* Disable CORS headers if cross-domain calls are not supported/expected.
* Be as specific as possible and as general as necessary when setting the origins of cross-domain calls.

## Sensitive information in HTTP requests

RESTful web services should be careful to prevent leaking credentials. Passwords, security tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them intrinsically valuable.

* In POST/PUT requests sensitive data should be transferred in the request body or request headers.
* In GET requests sensitive data should be transferred in an HTTP Header.

OK:

https://example.com/resourceCollection/[ID]/action

https://twitter.com/vanderaj/lists

NOT OK:

https://example.com/controller/123/action?apiKey=a53f435643de32 because API Key is into the URL.

## HTTP Return Code

HTTP defines [status code](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes). When designing REST API, don't just use 200 for success or 404 for error. Always use the semantically appropriate status code for the response.

Here is a non-exhaustive selection of security related REST API status codes. Use it to ensure you return the correct code.

| Code | Message | Description |
| --- | --- | --- |
| 200 | OK | Response to a successful REST API action. The HTTP method can be GET, POST, PUT, PATCH or DELETE. |
| 201 | Created | The request has been fulfilled and resource created. A URI for the created resource is returned in the Location header. |
| 202 | Accepted | The request has been accepted for processing, but processing is not yet complete. |
| 301 | Moved Permanently | Permanent redirection. |
| 304 | Not Modified | Caching related response that returned when the client has the same copy of the resource as the server. |
| 307 | Temporary Redirect | Temporary redirection of resource. |
| 400 | Bad Request | The request is malformed, such as message body format error. |
| 401 | Unauthorized | Wrong or no authentication ID/password provided. |
| 403 | Forbidden | It's used when the authentication succeeded but authenticated user doesn't have permission to the request resource. |
| 404 | Not Found | When a non-existent resource is requested. |
| 405 | Method Not Acceptable | The error for an unexpected HTTP method. For example, the REST API is expecting HTTP GET, but HTTP PUT is used. |
| 406 | Unacceptable | The client presented a content type in the Accept header which is not supported by the server API. |
| 413 | Payload too large | Use it to signal that the request size exceeded the given limit e.g. regarding file uploads. |
| 415 | Unsupported Media Type | The requested content type is not supported by the REST service. |
| 429 | Too Many Requests | The error is used when there may be DOS attack detected or the request is rejected due to rate limiting. |
| 500 | Internal Server Error | An unexpected condition prevented the server from fulfilling the request. Be aware that the response should not reveal internal information that helps an attacker, e.g. detailed error messages or stack traces. |
| 501 | Not Implemented | The REST service does not implement the requested operation yet. |
| 503 | Service Unavailable | The REST service is temporarily unable to process the request. Used to inform the client it should retry at a later time. |

# Chapter 4: SQL Injection Prevention

## Introduction

This article is focused on providing clear, simple, actionable guidance for preventing SQL Injection flaws in your applications. [SQL Injection](https://owasp.org/www-community/attacks/SQL_Injection) attacks are unfortunately very common, and this is due to two factors:

1. the significant prevalence of SQL Injection vulnerabilities, and
2. the attractiveness of the target (i.e., the database typically contains all the interesting/critical data for your application).

SQL Injection flaws are introduced when software developers create dynamic database queries constructed with string concatenation which includes user supplied input. To avoid SQL injection flaws is simple. Developers need to either: a) stop writing dynamic queries with string concatenation; and/or b) prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

This article provides a set of simple techniques for preventing SQL Injection vulnerabilities by avoiding these two problems. These techniques can be used with practically any kind of programming language with any type of database. There are other types of databases, like XML databases, which can have similar problems (e.g., XPath and XQuery injection) and these techniques can be used to protect them as well.

Primary Defenses:

* Option 1: Use of Prepared Statements (with Parameterized Queries)
* Option 2: Use of Properly Constructed Stored Procedures
* Option 3: Allow-list Input Validation
* Option 4: Escaping All User Supplied Input

Additional Defenses:

* Also: Enforcing Least Privilege
* Also: Performing Allow-list Input Validation as a Secondary Defense

Unsafe Example:

SQL injection flaws typically look like this:

The following (Java) example is UNSAFE, and would allow an attacker to inject code into the query that would be executed by the database. The unvalidated "customerName" parameter that is simply appended to the query allows an attacker to inject any SQL code they want. Unfortunately, this method for accessing databases is all too common.

String query = "SELECT account\_balance FROM user\_data WHERE user\_name = "

+ request.getParameter("customerName");

try {

Statement statement = connection.createStatement( ... );

ResultSet results = statement.executeQuery( query );

}

...

## Primary Defenses

### Defense Option 1: Prepared Statements (with Parameterized Queries)

The use of prepared statements with variable binding (aka parameterized queries) is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.

Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker. In the safe example below, if an attacker were to enter the userID of tom' or '1'='1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'='1.

Language specific recommendations:

* Java EE – use PreparedStatement() with bind variables
* .NET – use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
* PHP – use PDO with strongly typed parameterized queries (using bindParam())
* Hibernate - use createQuery() with bind variables (called named parameters in Hibernate)
* SQLite - use sqlite3\_prepare() to create a [statement object](https://www.sqlite.org/c3ref/stmt.html)

In rare circumstances, prepared statements can harm performance. When confronted with this situation, it is best to either a) strongly validate all data or b) escape all user supplied input using an escaping routine specific to your database vendor as described below, rather than using a prepared statement.

Safe Java Prepared Statement Example:

The following code example uses a PreparedStatement, Java's implementation of a parameterized query, to execute the same database query.

// This should REALLY be validated too

String custname = request.getParameter("customerName");

// Perform input validation to detect attacks

String query = "SELECT account\_balance FROM user\_data WHERE user\_name = ? ";

PreparedStatement pstmt = connection.prepareStatement( query );

pstmt.setString( 1, custname);

ResultSet results = pstmt.executeQuery( );

Safe C# .NET Prepared Statement Example:

With .NET, it's even more straightforward. The creation and execution of the query doesn't change. All you have to do is simply pass the parameters to the query using the Parameters.Add() call as shown here.

String query = "SELECT account\_balance FROM user\_data WHERE user\_name = ?";

try {

OleDbCommand command = new OleDbCommand(query, connection);

command.Parameters.Add(new OleDbParameter("customerName", CustomerName Name.Text));

OleDbDataReader reader = command.ExecuteReader();

// …

} catch (OleDbException se) {

// error handling

}

We have shown examples in Java and .NET but practically all other languages, including Cold Fusion, and Classic ASP, support parameterized query interfaces. Even SQL abstraction layers, like the [Hibernate Query Language](http://hibernate.org/) (HQL) have the same type of injection problems (which we call [HQL Injection](https://cwe.mitre.org/data/definitions/564.html)). HQL supports parameterized queries as well, so we can avoid this problem:

Hibernate Query Language (HQL) Prepared Statement (Named Parameters) Examples:

//First is an unsafe HQL Statement

Query unsafeHQLQuery = session.createQuery("from Inventory where productID='"+userSuppliedParameter+"'");

//Here is a safe version of the same query using named parameters

Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid");

safeHQLQuery.setParameter("productid", userSuppliedParameter);

For examples of parameterized queries in other languages, including Ruby, PHP, Cold Fusion, and Perl, see the [Query Parameterization Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html) or this [site](http://bobby-tables.com/).

Developers tend to like the Prepared Statement approach because all the SQL code stays within the application. This makes your application relatively database independent.

### Defense Option 2: Stored Procedures

Stored procedures are not always safe from SQL injection. However, certain standard stored procedure programming constructs have the same effect as the use of parameterized queries when implemented safely which is the norm for most stored procedure languages.

They require the developer to just build SQL statements with parameters which are automatically parameterized unless the developer does something largely out of the norm. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application. Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you.

Note: 'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation. Developers do not usually generate dynamic SQL inside stored procedures. However, it can be done, but should be avoided. If it can't be avoided, the stored procedure must use input validation or proper escaping as described in this article to make sure that all user supplied input to the stored procedure can't be used to inject SQL code into the dynamically generated query. Auditors should always look for uses of sp\_execute, execute or exec within SQL Server stored procedures. Similar audit guidelines are necessary for similar functions for other vendors.

There are also several cases where stored procedures can increase risk. For example, on MS SQL server, you have 3 main default roles: db\_datareader, db\_datawriter and db\_owner. Before stored procedures came into use, DBA's would give db\_datareader or db\_datawriter rights to the webservice's user, depending on the requirements. However, stored procedures require execute rights, a role that is not available by default. Some setups where the user management has been centralized, but is limited to those 3 roles, cause all web apps to run under db\_owner rights so stored procedures can work. Naturally, that means that if a server is breached the attacker has full rights to the database, where previously they might only have had read-access.

Safe Java Stored Procedure Example:

The following code example uses a CallableStatement, Java's implementation of the stored procedure interface, to execute the same database query. The sp\_getAccountBalance stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

// This should REALLY be validated

String custname = request.getParameter("customerName");

try {

CallableStatement cs = connection.prepareCall("{call sp\_getAccountBalance(?)}");

cs.setString(1, custname);

ResultSet results = cs.executeQuery();

// … result set handling

} catch (SQLException se) {

// … logging and error handling

}

Safe VB .NET Stored Procedure Example:

The following code example uses a SqlCommand, .NET's implementation of the stored procedure interface, to execute the same database query. The sp\_getAccountBalance stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

Try

Dim command As SqlCommand = new SqlCommand("sp\_getAccountBalance", connection)

command.CommandType = CommandType.StoredProcedure

command.Parameters.Add(new SqlParameter("@CustomerName", CustomerName.Text))

Dim reader As SqlDataReader = command.ExecuteReader()

'...

Catch se As SqlException

'error handling

End Try

### Defense Option 3: Allow-list Input Validation

Various parts of SQL queries aren't legal locations for the use of bind variables, such as the names of tables or columns, and the sort order indicator (ASC or DESC). In such situations, input validation or query redesign is the most appropriate defense. For the names of tables or columns, ideally those values come from the code, and not from user parameters.

But if user parameter values are used for targeting different table names and column names, then the parameter values should be mapped to the legal/expected table or column names to make sure unvalidated user input doesn't end up in the query. Please note, this is a symptom of poor design and a full rewrite should be considered if time allows.

Here is an example of table name validation.

String tableName;

switch(PARAM):

case "Value1": tableName = "fooTable";

break;

case "Value2": tableName = "barTable";

break;

...

default      : throw new InputValidationException("unexpected value provided"

+ " for table name");

The tableName can then be directly appended to the SQL query since it is now known to be one of the legal and expected values for a table name in this query. Keep in mind that generic table validation functions can lead to data loss as table names are used in queries where they are not expected.

For something simple like a sort order, it would be best if the user supplied input is converted to a boolean, and then that boolean is used to select the safe value to append to the query. This is a very standard need in dynamic query creation.

For example:

public String someMethod(boolean sortOrder) {

String SQLquery = "some SQL ... order by Salary " + (sortOrder ? "ASC" : "DESC");`

...

Any time user input can be converted to a non-String, like a date, numeric, boolean, enumerated type, etc. before it is appended to a query, or used to select a value to append to the query, this ensures it is safe to do so.

Input validation is also recommended as a secondary defense in ALL cases, even when using bind variables as is discussed later in this article. More techniques on how to implement strong input validation is described in the [Input Validation Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html).

### Defense Option 4: Escaping All User-Supplied Input

This technique should only be used as a last resort, when none of the above are feasible. Input validation is probably a better choice as this methodology is frail compared to other defenses and we cannot guarantee it will prevent all SQL Injection in all situations.

This technique is to escape user input before putting it in a query. It is very database specific in its implementation. It's usually only recommended to retrofit legacy code when implementing input validation isn't cost effective. Applications built from scratch, or applications requiring low risk tolerance should be built or re-written using parameterized queries, stored procedures, or some kind of Object Relational Mapper (ORM) that builds your queries for you.

This technique works like this. Each DBMS supports one or more character escaping schemes specific to certain kinds of queries. If you then escape all user supplied input using the proper escaping scheme for the database you are using, the DBMS will not confuse that input with SQL code written by the developer, thus avoiding any possible SQL injection vulnerabilities.

The OWASP Enterprise Security API (ESAPI) is a free, open source, web application security control library that makes it easier for programmers to write lower-risk applications. The ESAPI libraries are designed to make it easier for programmers to retrofit security into existing applications. The ESAPI libraries also serve as a solid foundation for new development:

* Full details on [ESAPI are available here on OWASP](https://owasp.org/www-project-enterprise-security-api/).
* The javadoc for [ESAPI 2.x (Legacy) is available](https://www.javadoc.io/doc/org.owasp.esapi/esapi/2.1.0). This code was migrated to GitHub in November 2014.
* [The legacy ESAPI for Java at GitHub](https://github.com/ESAPI/esapi-java-legacy) helps understand existing use of it when Javadoc seems insufficient.
* [An attempt at another ESAPI for Java GitHub](https://github.com/ESAPI/esapi-java) has other approaches and no tests or concrete codecs.

To find the javadoc specifically for the database encoders, click on the Codec class on the left hand side. There are lots of Codecs implemented. The two Database specific codecs are OracleCodec, and MySQLCodec.

Just click on their names in the All Known Implementing Classes: at the top of the Interface Codec page.

At this time, ESAPI currently has database encoders for:

* Oracle
* MySQL (Both ANSI and native modes are supported)

Database encoders are forthcoming for:

* SQL Server
* PostgreSQL

If your database encoder is missing, please let us know.

#### Database Specific Escaping Details

If you want to build your own escaping routines, here are the escaping details for each of the databases that we have developed ESAPI Encoders for:

* Oracle
* SQL Server
* DB2

##### Oracle Escaping

This information is based on the [Oracle Escape character information](http://www.orafaq.com/wiki/SQL_FAQ#How_does_one_escape_special_characters_when_writing_SQL_queries.3F).

###### Escaping Dynamic Queries

To use an ESAPI database codec is pretty simple. An Oracle example looks something like:

ESAPI.encoder().encodeForSQL( new OracleCodec(), queryparam );

So, if you had an existing Dynamic query being generated in your code that was going to Oracle that looked like this:

String query = "SELECT user\_id FROM user\_data WHERE user\_name = '"

+ req.getParameter("userID")

+ "' and user\_password = '" + req.getParameter("pwd") +"'";

try {

Statement statement = connection.createStatement( … );

ResultSet results = statement.executeQuery( query );

}

You would rewrite the first line to look like this:

Codec ORACLE\_CODEC = new OracleCodec();

String query = "SELECT user\_id FROM user\_data WHERE user\_name = '"

+ ESAPI.encoder().encodeForSQL( ORACLE\_CODEC, req.getParameter("userID"))

+ "' and user\_password = '"

+ ESAPI.encoder().encodeForSQL( ORACLE\_CODEC, req.getParameter("pwd")) +"'";

And it would now be safe from SQL injection, regardless of the input supplied.

For maximum code readability, you could also construct your own OracleEncoder:

Encoder oe = new OracleEncoder();

String query = "SELECT user\_id FROM user\_data WHERE user\_name = '"

+ oe.encode( req.getParameter("userID")) + "' and user\_password = '"

+ oe.encode( req.getParameter("pwd")) +"'";

With this type of solution, you would need only to wrap each user-supplied parameter being passed into an ESAPI.encoder().encodeForOracle( ) call or whatever you named the call and you would be done.

###### Turn off character replacement

Use SET DEFINE OFF or SET SCAN OFF to ensure that automatic character replacement is turned off. If this character replacement is turned on, the & character will be treated like a SQLPlus variable prefix that could allow an attacker to retrieve private data.

See [here](https://docs.oracle.com/cd/B19306_01/server.102/b14357/ch12040.htm#i2698854) and [here](https://stackoverflow.com/a/410490) for more information

###### Escaping Wildcard characters in Like Clauses

The LIKE keyword allows for text scanning searches. In Oracle, the underscore \_ character matches only one character, while the ampersand % is used to match zero or more occurrences of any characters. These characters must be escaped in LIKE clause criteria.

For example:

SELECT name FROM emp WHERE id LIKE '%/\_%' ESCAPE '/';

SELECT name FROM emp WHERE id LIKE '%\%%' ESCAPE '\';

###### Oracle 10g escaping

An alternative for Oracle 10g and later is to place { and } around the string to escape the entire string. However, you have to be careful that there isn't a } character already in the string. You must search for these and if there is one, then you must replace it with }}. Otherwise that character will end the escaping early, and may introduce a vulnerability.

##### MySQL Escaping

MySQL supports two escaping modes:

1. ANSI\_QUOTES SQL mode, and a mode with this off, which we call
2. MySQL mode.

ANSI SQL mode: Simply encode all ' (single tick) characters with '' (two single ticks)

MySQL mode, do the following:

NUL (0x00) --> \0  [This is a zero, not the letter O]

BS  (0x08) --> \b

TAB (0x09) --> \t

LF  (0x0a) --> \n

CR  (0x0d) --> \r

SUB (0x1a) --> \Z

"   (0x22) --> \"

%   (0x25) --> \%

'   (0x27) --> \'

\   (0x5c) --> \\

\_   (0x5f) --> \\_

all other non-alphanumeric characters with ASCII values

less than 256 --> \c where 'c' is the original non-alphanumeric character.

This information is based on the [MySQL Escape character information](https://dev.mysql.com/doc/refman/5.7/en/string-literals.html).

##### SQL Server Escaping

We have not implemented the SQL Server escaping routine yet, but the following has good pointers and links to articles describing how to prevent SQL injection attacks on SQL server, see [here](https://aka.ms/sql-injection).

##### DB2 Escaping

This information is based on [DB2 WebQuery special characters](https://www.ibm.com/support/pages/web-query-special-characters) as well as some information from [Oracle's JDBC DB2 driver](https://docs.oracle.com/cd/E12840_01/wls/docs103/jdbc_drivers/sqlescape.html).

Information in regards to differences between several [DB2 Universal drivers](https://www.ibm.com/support/pages/db2-jdbc-driver-versions-and-downloads).

#### Hex-encoding all input

A somewhat special case of escaping is the process of hex-encode the entire string received from the user (this can be seen as escaping every character). The web application should hex-encode the user input before including it in the SQL statement. The SQL statement should take into account this fact, and accordingly compare the data.

For example, if we have to look up a record matching a sessionID, and the user transmitted the string abc123 as the session ID, the select statement would be:

SELECT ... FROM session WHERE hex\_encode(sessionID) = '616263313233'

hex\_encode should be replaced by the particular facility for the database being used. The string 606162313233 is the hex encoded version of the string received from the user (it is the sequence of hex values of the ASCII/UTF-8 codes of the user data).

If an attacker were to transmit a string containing a single-quote character followed by their attempt to inject SQL code, the constructed SQL statement will only look like:

... WHERE hex\_encode ( ... ) = '2720 ... '

27 being the ASCII code (in hex) of the single-quote, which is simply hex-encoded like any other character in the string. The resulting SQL can only contain numeric digits and letters a to f, and never any special character that could enable an SQL injection.

#### Escaping SQLi in PHP

Use prepared statements and parameterized queries. These are SQL statements that are sent to and parsed by the database server separately from any parameters. This way it is impossible for an attacker to inject malicious SQL.

You basically have two options to achieve this:

1. Using [PDO](https://www.php.net/manual/en/book.pdo.php) (for any supported database driver):

$stmt = $pdo->prepare('SELECT \* FROM employees WHERE name = :name');

$stmt->execute(array('name' => $name));

foreach ($stmt as $row) {

// do something with $row

}

1. Using [MySQLi](https://www.php.net/manual/en/book.mysqli.php) (for MySQL):

$stmt = $dbConnection->prepare('SELECT \* FROM employees WHERE name = ?');

$stmt->bind\_param('s', $name);

$stmt->execute();

$result = $stmt->get\_result();

while ($row = $result->fetch\_assoc()) {

// do something with $row

}

PDO is the universal option. If you're connecting to a database other than MySQL, you can refer to a driver-specific second option (e.g. pg\_prepare() and pg\_execute() for PostgreSQL).

## Additional Defenses

Beyond adopting one of the four primary defenses, we also recommend adopting all of these additional defenses in order to provide defense in depth. These additional defenses are:

* Least Privilege
* Allow-list Input Validation

### Least Privilege

To minimize the potential damage of a successful SQL injection attack, you should minimize the privileges assigned to every database account in your environment. Do not assign DBA or admin type access rights to your application accounts. We understand that this is easy, and everything just 'works' when you do it this way, but it is very dangerous.

Start from the ground up to determine what access rights your application accounts require, rather than trying to figure out what access rights you need to take away. Make sure that accounts that only need read access are only granted read access to the tables they need access to.

If an account only needs access to portions of a table, consider creating a view that limits access to that portion of the data and assigning the account access to the view instead, rather than the underlying table. Rarely, if ever, grant create or delete access to database accounts.

If you adopt a policy where you use stored procedures everywhere, and don't allow application accounts to directly execute their own queries, then restrict those accounts to only be able to execute the stored procedures they need. Don't grant them any rights directly to the tables in the database.

SQL injection is not the only threat to your database data. Attackers can simply change the parameter values from one of the legal values they are presented with, to a value that is unauthorized for them, but the application itself might be authorized to access. As such, minimizing the privileges granted to your application will reduce the likelihood of such unauthorized access attempts, even when an attacker is not trying to use SQL injection as part of their exploit.

While you are at it, you should minimize the privileges of the operating system account that the DBMS runs under. Don't run your DBMS as root or system! Most DBMSs run out of the box with a very powerful system account. For example, MySQL runs as system on Windows by default! Change the DBMS's OS account to something more appropriate, with restricted privileges.

### Multiple DB Users

The designer of web applications should not only avoid using the same owner/admin account in the web applications to connect to the database. Different DB users could be used for different web applications.

In general, each separate web application that requires access to the database could have a designated database user account that the web-app will use to connect to the DB. That way, the designer of the application can have good granularity in the access control, thus reducing the privileges as much as possible. Each DB user will then have select access to what it needs only, and write-access as needed.

As an example, a login page requires read access to the username and password fields of a table, but no write access of any form (no insert, update, or delete). However, the sign-up page certainly requires insert privilege to that table; this restriction can only be enforced if these web apps use different DB users to connect to the database.

### Views

You can use SQL views to further increase the granularity of access by limiting the read access to specific fields of a table or joins of tables. It could potentially have additional benefits: for example, suppose that the system is required (perhaps due to some specific legal requirements) to store the passwords of the users, instead of salted-hashed passwords.

The designer could use views to compensate for this limitation; revoke all access to the table (from all DB users except the owner/admin) and create a view that outputs the hash of the password field and not the field itself. Any SQL injection attack that succeeds in stealing DB information will be restricted to stealing the hash of the passwords (could even be a keyed hash), since no DB user for any of the web applications has access to the table itself.

### Allow-list Input Validation

In addition to being a primary defense when nothing else is possible (e.g., when a bind variable isn't legal), input validation can also be a secondary defense used to detect unauthorized input before it is passed to the SQL query. For more information please see the [Input Validation Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html). Proceed with caution here. Validated data is not necessarily safe to insert into SQL queries via string building.

## Related Articles

SQL Injection Attack Cheat Sheets:

The following articles describe how to exploit different kinds of SQL Injection Vulnerabilities on various platforms that this article was created to help you avoid:

* [SQL Injection Cheat Sheet](https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/)
* Bypassing WAF's with SQLi - [SQL Injection Bypassing WAF](https://owasp.org/www-community/attacks/SQL_Injection_Bypassing_WAF)

Description of SQL Injection Vulnerabilities:

* OWASP article on [SQL Injection](https://owasp.org/www-community/attacks/SQL_Injection) Vulnerabilities
* OWASP article on [Blind\_SQL\_Injection](https://owasp.org/www-community/attacks/Blind_SQL_Injection) Vulnerabilities

How to Avoid SQL Injection Vulnerabilities:

* [OWASP Developers Guide](https://github.com/OWASP/DevGuide) article on how to avoid SQL injection vulnerabilities
* OWASP Cheat Sheet that provides [numerous language specific examples of parameterized queries using both Prepared Statements and Stored Procedures](https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html)
* [The Bobby Tables site (inspired by the XKCD webcomic) has numerous examples in different languages of parameterized Prepared Statements and Stored Procedures](http://bobby-tables.com/)

How to Review Code for SQL Injection Vulnerabilities:

* [OWASP Code Review Guide](https://wiki.owasp.org/index.php/Category:OWASP_Code_Review_Project) article on how to [Review Code for SQL Injection](https://wiki.owasp.org/index.php/Reviewing_Code_for_SQL_Injection) Vulnerabilities

How to Test for SQL Injection Vulnerabilities:

* [OWASP Testing Guide](https://owasp.org/www-project-web-security-testing-guide) article on how to [Test for SQL Injection](https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection.html) Vulnerabilities

# Chapter 5: SonarQube Application Security

## Use parameterized queries, prepared statements, or stored procedures and bind variables to SQL query parameters.

An example from UNRWA Code. This is a compliant solution that shows how to write down a compliant code.

public void Foo(DbContext context, string query, string param)

{

context.Database.ExecuteSqlCommand("SELECT \* FROM mytable WHERE mycol=@p0", param); // Compliant, it's a parametrized safe query

}

## Never use hard-coded passwords

Always review hard-coded messages and remove them from code. Passwords, keys, secrets, …etc shall never be put inside code. They should be referred to by a url in a key vault.

var domain = ConfigurationManager.AppSettings["DMSImpersonationPWD"];

//  sharepointServiceProxy.Credentials = new System.Net.NetworkCredential($"{domain}\\{userName}", password );

sharepointServiceProxy.Credentials = new System.Net.NetworkCredential($"UNRWA\\RRIS-SPAdmin", "P@ssw0rd");

## Always use strong encryption

* Only use random number generators which are [recommended by OWASP](https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html#secure-random-number-generation) or any other trusted organization.
* Use the generated random values only once.
* You should not expose the generated random value. If you have to store it, make sure that the database or file is secure.

## Use Parameterized Query

SQL Injection is best prevented through the use of parameterized queries. The following chart demonstrates, with real-world code samples, how to build parameterized queries in most of the common web languages. The purpose of these code samples is to demonstrate to the web developer how to avoid SQL Injection when building database queries within a web application.

## Always use HTTPS not HTTP

Usage of encrypted protocols adds another security layer to alleviate against attacks such as sniffing and eavesdropping. Even a free self-signed certificate is better than using plaintext protocols such as HTTP.

<https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html>

# Chapter 6: Top software vulnerabilities globally

## CWE: Top 25 Most Dangerous Software Weaknesses CWE 2022

### Types of Weaknesses

Base: a weakness that is still mostly independent of a resource or technology, but with sufficient details to provide specific methods for detection and prevention. Base level weaknesses typically describe issues in terms of 2 or 3 of the following dimensions: behavior, property, technology, language, and resource.

Class: a weakness that is described in a very abstract fashion, typically independent of any specific language or technology. More specific than a Pillar Weakness, but more general than a Base Weakness. Class level weaknesses typically describe issues in terms of 1 or 2 of the following dimensions: behavior, property, and resource.

Composite: a Compound Element that consists of two or more distinct weaknesses, in which all weaknesses must be present at the same time in order for a potential vulnerability to arise. Removing any of the weaknesses eliminates or sharply reduces the risk. One weakness, X, can be "broken down" into component weaknesses Y and Z. There can be cases in which one weakness might not be essential to a composite, but changes the nature of the composite when it becomes a vulnerability.

Variant: a weakness that is linked to a certain type of product, typically involving a specific language or technology. More specific than a Base weakness. Variant level weaknesses typically describe issues in terms of 3 to 5 of the following dimensions: behavior, property, technology, language, and resource.

### [Out-of-bounds Write - (787)](https://cwe.mitre.org/data/definitions/787.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 787 (Out-of-bounds Write)

The software writes data past the end, or before the beginning, of the intended buffer.Memory Corruption

### [Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') - (79)](https://cwe.mitre.org/data/definitions/79.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 79 (Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'))

The software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.XSSHTML InjectionCSS

### [Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') - (89)](https://cwe.mitre.org/data/definitions/89.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 89 (Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'))

The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

### [Improper Input Validation - (20)](https://cwe.mitre.org/data/definitions/20.html)-C

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 20 (Improper Input Validation)

The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly.

### [Out-of-bounds Read - (125)](https://cwe.mitre.org/data/definitions/125.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 125 (Out-of-bounds Read)

The software reads data past the end, or before the beginning, of the intended buffer.

### [Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') - (78)](https://cwe.mitre.org/data/definitions/78.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 78 (Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'))

The software constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.Shell injectionShell metacharacters

### [Use After Free - (416)](https://cwe.mitre.org/data/definitions/416.html)-V

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 416 (Use After Free)

Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.Dangling pointerUse-After-Free

### [Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') - (22)](https://cwe.mitre.org/data/definitions/22.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 22 (Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'))

The software uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.Directory traversalPath traversal

### [Cross-Site Request Forgery (CSRF) - (352)](https://cwe.mitre.org/data/definitions/352.html)-C

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 352 (Cross-Site Request Forgery (CSRF))

The web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.Session RidingCross Site Reference ForgeryXSRF

### [Unrestricted Upload of File with Dangerous Type - (434)](https://cwe.mitre.org/data/definitions/434.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 434 (Unrestricted Upload of File with Dangerous Type)

The software allows the attacker to upload or transfer files of dangerous types that can be automatically processed within the product's environment.Unrestricted File Upload

### [NULL Pointer Dereference - (476)](https://cwe.mitre.org/data/definitions/476.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 476 (NULL Pointer Dereference)

A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.NPDnull derefnil pointer dereference

### [Deserialization of Untrusted Data - (502)](https://cwe.mitre.org/data/definitions/502.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 502 (Deserialization of Untrusted Data)

The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid.Marshaling, UnmarshalingPickling, UnpicklingPHP Object Injection

### [Integer Overflow or Wraparound - (190)](https://cwe.mitre.org/data/definitions/190.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 190 (Integer Overflow or Wraparound)

The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control.

### [Improper Authentication - (287)](https://cwe.mitre.org/data/definitions/287.html)-C

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 287 (Improper Authentication)

When an actor claims to have a given identity, the software does not prove or insufficiently proves that the claim is correct.authentificationAuthNAuthC

### [Use of Hard-coded Credentials - (798)](https://cwe.mitre.org/data/definitions/798.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 798 (Use of Hard-coded Credentials)

The software contains hard-coded credentials, such as a password or cryptographic key, which it uses for its own inbound authentication, outbound communication to external components, or encryption of internal data.

### [Missing Authorization - (862)](https://cwe.mitre.org/data/definitions/862.html)-C

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 862 (Missing Authorization)

The software does not perform an authorization check when an actor attempts to access a resource or perform an action.AuthZ

### [Improper Neutralization of Special Elements used in a Command ('Command Injection') - (77)](https://cwe.mitre.org/data/definitions/77.html)-C

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 77 (Improper Neutralization of Special Elements used in a Command ('Command Injection'))

The software constructs all or part of a command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended command when it is sent to a downstream component.

### [Missing Authentication for Critical Function - (306)](https://cwe.mitre.org/data/definitions/306.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 306 (Missing Authentication for Critical Function)

The product does not perform any authentication for functionality that requires a provable user identity or consumes a significant amount of resources.

### [Improper Restriction of Operations within the Bounds of a Memory Buffer - (119)](https://cwe.mitre.org/data/definitions/119.html)-C

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 119 (Improper Restriction of Operations within the Bounds of a Memory Buffer)

The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.Buffer Overflowbuffer overrunmemory safety

### [Incorrect Default Permissions - (276)](https://cwe.mitre.org/data/definitions/276.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 276 (Incorrect Default Permissions)

During installation, installed file permissions are set to allow anyone to modify those files.

### [Server-Side Request Forgery (SSRF) - (918)](https://cwe.mitre.org/data/definitions/918.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 918 (Server-Side Request Forgery (SSRF))

The web server receives a URL or similar request from an upstream component and retrieves the contents of this URL, but it does not sufficiently ensure that the request is being sent to the expected destination.XSPA

### [Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition') - (362)](https://cwe.mitre.org/data/definitions/362.html)-C

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 362 (Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'))

The program contains a code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently.

### [Uncontrolled Resource Consumption - (400)](https://cwe.mitre.org/data/definitions/400.html)-C

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 400 (Uncontrolled Resource Consumption)

The software does not properly control the allocation and maintenance of a limited resource, thereby enabling an actor to influence the amount of resources consumed, eventually leading to the exhaustion of available resources.Resource Exhaustion

### [Improper Restriction of XML External Entity Reference - (611)](https://cwe.mitre.org/data/definitions/611.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 611 (Improper Restriction of XML External Entity Reference)

The software processes an XML document that can contain XML entities with URIs that resolve to documents outside of the intended sphere of control, causing the product to embed incorrect documents into its output.XXE

### [Improper Control of Generation of Code ('Code Injection') - (94)](https://cwe.mitre.org/data/definitions/94.html)-B

1387 (Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses) > 94 (Improper Control of Generation of Code ('Code Injection'))

The software constructs all or part of a code segment using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the syntax or behavior of the intended code segment.

## SANS: Top 25 Errors 2022

| ID | Name |
| --- | --- |
| [CWE-787](https://cwe.mitre.org/data/definitions/787.html) | Out-of-bounds Write |
| [CWE-79](https://cwe.mitre.org/data/definitions/79.html) | Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') |
| [CWE-89](https://cwe.mitre.org/data/definitions/89.html) | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') |
| [CWE-20](https://cwe.mitre.org/data/definitions/20.html) | Improper Input Validation |
| [CWE-125](https://cwe.mitre.org/data/definitions/125.html) | Out-of-bounds Read |
| [CWE-78](https://cwe.mitre.org/data/definitions/78.html) | Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') |
| [CWE-416](https://cwe.mitre.org/data/definitions/416.html) | Use After Free |
| [CWE-22](https://cwe.mitre.org/data/definitions/22.html) | Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') |
| [CWE-352](https://cwe.mitre.org/data/definitions/352.html) | Cross-Site Request Forgery (CSRF) |
| [CWE-434](https://cwe.mitre.org/data/definitions/434.html) | Unrestricted Upload of File with Dangerous Type |
| [CWE-476](https://cwe.mitre.org/data/definitions/476.html) | NULL Pointer Dereference |
| [CWE-502](https://cwe.mitre.org/data/definitions/502.html) | Deserialization of Untrusted Data |
| [CWE-190](https://cwe.mitre.org/data/definitions/190.html) | Integer Overflow or Wraparound |
| [CWE-287](https://cwe.mitre.org/data/definitions/287.html) | Improper Authentication |
| [CWE-798](https://cwe.mitre.org/data/definitions/798.html) | Use of Hard-coded Credentials |
| [CWE-862](https://cwe.mitre.org/data/definitions/862.html) | Missing Authorization |
| [CWE-77](https://cwe.mitre.org/data/definitions/77.html) | Improper Neutralization of Special Elements used in a Command ('Command Injection') |
| [CWE-306](https://cwe.mitre.org/data/definitions/306.html) | Missing Authentication for Critical Function |
| [CWE-119](https://cwe.mitre.org/data/definitions/119.html) | Improper Restriction of Operations within the Bounds of a Memory Buffer |
| [CWE-276](https://cwe.mitre.org/data/definitions/276.html) | Incorrect Default Permissions |
| [CWE-918](https://cwe.mitre.org/data/definitions/918.html) | Server-Side Request Forgery (SSRF) |
| [CWE-362](https://cwe.mitre.org/data/definitions/362.html) | Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition') |
| [CWE-400](https://cwe.mitre.org/data/definitions/400.html) | Uncontrolled Resource Consumption |
| [CWE-611](https://cwe.mitre.org/data/definitions/611.html) | Improper Restriction of XML External Entity Reference |
| [CWE-94](https://cwe.mitre.org/data/definitions/94.html) | Improper Control of Generation of Code ('Code Injection') |

# Chapter 7: Mobile Application Security

Security has always been a major concern for businesses. And this concern is even greater when it comes to mobile apps.

Today every business has a mobile app to connect more easily with their customers. And if that business does not take proper security protections it can put their brand at risk.

Mobile devices span multiple operating systems and, given the distributed nature of components, mobile app security often experiences problems.

I hope your business is properly secured and you are just looking for a mobile app security checklist for the future. If that's the case, good for you – being a business owner means you must take care of mobile app security.

But according to a [survey](https://www.pixelcrayons.com/blog/mobile-app-stats/?utm_source=freecodecamp&utm_medium=mobile%2Bapp%2Bdevelopment_sk&utm_campaign=website), more than 75% of mobile applications will fail basic security tests.

Many employees download apps from app stores and use mobile applications that can access enterprise assets or perform business functions. And unfortunately, these applications have little or no security assurances. They are exposed to attacks and violations of enterprise security policies all the time.

I know that nobody wants to be a part of this failure. That is why you need to follow a proper mobile app security checklist.

## Enforce Strong Authentication

To prevent unauthorised access and password guessing attacks, you should implement multi-factor authentication. The three main factors for authentication are

* something that a user knows, such as a password or PIN
* something the user has, such as a mobile device
* or something the user is, such as a fingerprint.

Combining password-based authentication with a client certificate, device ID, or one-time password significantly reduces the risk of unauthorised access. You can also implement time-of-day and location-based restrictions to prevent fraud.

## Encrypt Mobile Communications

With threats like snooping and man-in-the-middle attacks over WiFi and cellular networks, IT should make sure that all communications between mobile apps and app servers are encrypted.

Strong encryption that leverages 4096-bit SSL keys and session-based key exchanges can prevent even the most determined hackers from decrypting communications.

Besides encrypting traffic, IT should confirm that data at rest—the sensitive data stored on users' phones—is also encrypted. For ultra-sensitive data, IT might want to prevent data from ever being downloaded to the end user device at all.

## Patch App and Operating System Vulnerabilities

Recent Android and iOS vulnerabilities such as [Stagefright](https://play.google.com/store/apps/details?id=com.zimperium.stagefrightdetector&hl=en_IN) and [XcodeGhost](https://www.macrumors.com/2015/09/20/xcodeghost-chinese-malware-faq/) have exposed mobile users to attack.

In addition to mobile OS flaws, IT must contend with a never-ending succession of app updates and fixes.

To protect mobile users from attack, IT should check mobile devices and ensure that the latest patches and updates have been applied.

## Protect Against Device Theft

Every year, millions of mobile devices are lost or stolen. To ensure sensitive data does not end up in the wrong hands, IT should provide a way to remotely wipe sensitive data Or—better yet—make sure data is never stored on mobile devices in the first place.

For employee-owned devices, IT should lock or wipe corporate information while leaving personal apps and files intact. When the device is found or replaced, IT should be able to quickly restore users’ apps and data.

## Scan Mobile Apps for Malware

Eliminate malware and adware by testing apps for malicious behaviour. Malware can be detected using virtual sandboxing or signature-based scanning tools. For mobile workspace or virtual mobile solutions, perform malware scans on the server.

## Perform VAPT

Penetration Testing and Vulnerabilities assessment is an important step in securing the mobile applications. There are many automated tools to apply VAPT on mobile applications:

* 1. Appie

For Android

* 1. Mobsf

Android/IOS

Integration with CI/CD

* 1. Android Studio

## Protect app data on your device

Make sure developers are not storing any sensitive data on their devices. If you must store data on device for some reason, first make sure it's encrypted/protected. And then only store it in files, data stores, and databases.

If you use the latest encryption technologies, you can get a higher level of security.

## Secure the Platform

Your platform should be properly secured and controlled. This process consists of detecting [jailbroken phones](https://www.scribd.com/document/226019655/IOS-Application-Security-Part-24-Jailbreak-Detection-and-Evasion) and preventing access to other services when needed.

## Prevent Data Leaks

To avoid data leaks while still allowing users to install personal apps on their mobile devices, IT must separate business apps from personal apps.

Creating secure mobile workspaces helps prevent malware from accessing corporate apps and stops users from copying, saving, or distributing sensitive data.

### For ironclad data leak prevention of confidential data:

* Control clipboard access to prevent copy and paste functions
* Block screen captures
* Prevent users from downloading confidential files to their phone or saving files on file sharing sites or connected devices or drives.
* Watermark sensitive files with users’ usernames and timestamps

## Optimise Data Caching‌‌

Did you know that mobile devices usually store cached data in order to enhance an app's performance? This is a major cause of security issues because those apps and devices become more vulnerable and it is relatively easy for attackers to breach and decrypt the cached data. This often results stolen user data.

You can require a password to access the application in case the nature of your data is extremely sensitive. This will help reduce vulnerabilities associated with cached data. ‌‌

After that, set up an automatic process that wipes cached data whenever the device gets restarted. This helps reduce the cache and mitigate security concerns.

## Isolate Application Information‌‌

You need to separate all information accessed through a mobile device from a user’s data. And this process of isolating information requires a few levels of protection around enterprise-deployed apps. This way corporate data will be separated from the employee’s private data as well as the consumer-facing application. ‌‌

This process of isolating data should increase your customers' satisfaction and productivity, all while making sure they're compliant with your security rules.

Using a container-based model can help you out in this case. Security is often more strict and won't compromise at any level of transmission. This ultimately helps eliminate the risk of corporate data loss. ‌‌‌‌‌‌‌‌

## JSON Web Token Cheat Sheet for Java

JSON Web Token is used to carry information related to the identity and characteristics (claims) of a client. This information is signed by the server in order for it to detect whether it was tampered with after sending it to the client. This will prevent an attacker from changing the identity or any characteristics (for example, changing the role from simple user to admin or change the client login).

### Token Structure

Token structure example taken from [JWT.IO](https://jwt.io/#debugger):

[Base64(HEADER)].[Base64(PAYLOAD)].[Base64(SIGNATURE)]

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.

TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

Chunk 1: Header

{

"alg": "HS256",

"typ": "JWT"

}

Chunk 2: Payload

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

Chunk 3: Signature

HMACSHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload), KEY )

## Objective

This cheatsheet provides tips to prevent common security issues when using JSON Web Tokens (JWT) with Java.

The tips presented in this article are part of a Java project that was created to show the correct way to handle creation and validation of JSON Web Tokens.

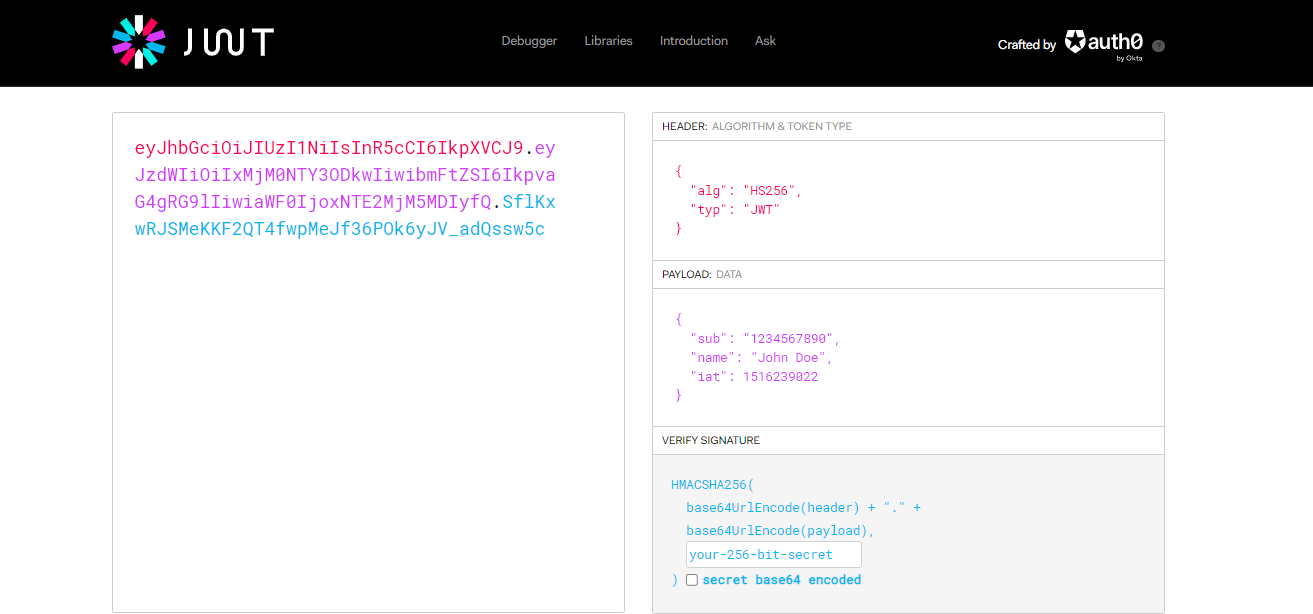
You can find the Java project [here](https://github.com/righettod/poc-jwt), it uses the official [JWT library](https://jwt.io/#libraries).

In the rest of the article, the term token refers to the JSON Web Tokens (JWT).

## Consideration about Using JWT

Even if a JWT token is "easy" to use and allow to expose services (mostly REST style) in a stateless way, it's not the solution that fits for all applications because it comes with some caveats, like for example the question of the storage of the token (tackled in this cheatsheet) and others...

If your application does not need to be fully stateless, you can consider using traditional session system provided by all web frameworks and follow the advice from the dedicated [session management cheat sheet](https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html). However, for stateless applications, when well implemented, it's a good candidate.



# Reference

* .Net:

1. <https://learn.microsoft.com/en-us/troubleshoot/developer/dotnet/framework/general/secure-applications>

* OWASP:

1. <https://cheatsheetseries.owasp.org/cheatsheets/DotNet_Security_Cheat_Sheet.html>
2. <https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html>
3. <https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html>

* TOP 25 Vulnerabilities:

1. <https://www.sans.org/top25-software-errors/>
2. <https://cwe.mitre.org/data/definitions/1387.html>
3. <https://github.com/OWASP/owasp-masvs/releases/latest/download/OWASP_MASVS-v1.4.2-en.pdf>

* SQL Injection:

1. <https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html>

* Mobile Application:

1. <https://www.freecodecamp.org/news/how-to-secure-mobile-apps/>
2. <https://koreascience.kr/article/JAKO202034465346164.pdf>
3. <https://zero2infynite.com/mobile-application-vulnerability-assessment-and-penetration-testing-services/>
4. <https://developer.android.com/studio/test>
5. <https://www.youtube.com/watch?v=lOY8oWQxG-E&ab_channel=TheCodeSwitch>

# Index

.Net Microsoft Application Security

Introduction, 8

Application Security Manual Instruction, 1

Contents, 2

CWE 2022, 62

OWASP ASVS, 13

Prefix, 7

Reference, 78

SonarQube Application Security, 60

SQL Injection Prevention, 44