DotNet Security Cheat Sheet

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 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.

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
- ASP.NET Core & Entity Framework Core Security Announcements

.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 commands for all data access, without exception.
- Do not use SqlCommand with a string parameter made up of a concatenated SQL String.
- List allowable values coming from the user. Use enums, TryParse 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 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 is a very effective 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 over SQL authentication.
- Use Always Encrypted 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) 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.
  - In the .NET framework the strongest algorithm for password hashing is PBKDF2, implemented as System.Security.Cryptography.Rfc2898DeriveBytes.
  - In .NET Core the strongest algorithm for password hashing is PBKDF2, implemented as Microsoft.AspNetCore.Cryptography.KeyDerivation.Pbkdf2 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.
- Make sure your application or protocol can easily support a future change of cryptographic algorithms.
- Use Nuget to keep all of your packages up to date. Watch the updates on your development setup, and plan updates to your applications accordingly.

General

- Lock down the config file.
    - Remove all aspects of configuration that are not in use.
    - Encrypt sensitive parts of the `web.config` using `aspnet_regiis -pe` (command line help).
- For Click Once applications the .Net Framework should be upgraded to use version `4.6.2` to ensure `TLS 1.1/1.2` support.

ASP NET Web Forms Guidance

ASP.NET Web Forms is the original browser-based application development API for the .NET framework, and is still the most common enterprise platform for web application development.

- Always use HTTPS.
- Enable requireSSL on cookies and form elements and HttpOnly on cookies in the web.config.
- Implement customErrors.
- Make sure tracing is turned off.
- While viewstate isn't always appropriate for web development, using it can provide CSRF mitigation. To make the ViewState protect against CSRF attacks you need to set the ViewStateUserKey:

```
protected override OnInit(EventArgs e) {
    base.OnInit(e);
    ViewStateUserKey = Session.SessionID;
}
```

If you don't use Viewstate, then look to the default master page of the ASP.NET Web Forms default template for a manual anti-CSRF token using a double-submit cookie.

```
private const string AntiXsrfTokenKey = "__AntiXsrfToken";
private const string AntiXsrfUserNameKey = "__AntiXsrfUserName";
private string _antiXsrfTokenValue;
protected void Page_Init(object sender, EventArgs e)
{
    // The code below helps to protect against XSRF attacks
    var requestCookie = Request.Cookies[AntiXsrfTokenKey];
    Guid requestCookieGuidValue;
    if (requestCookie != null && Guid.TryParse(requestCookie.Value, out requestCo

    {
       // Use the Anti-XSRF token from the cookie
       _antiXsrfTokenValue = requestCookie.Value;
       Page.ViewStateUserKey = _antiXsrfTokenValue;
    }
    else
    {
       // Generate a new Anti-XSRF token and save to the cookie
       _antiXsrfTokenValue = Guid.NewGuid().ToString("N");
       Page.ViewStateUserKey = _antiXsrfTokenValue;
       var responseCookie = new HttpCookie(AntiXsrfTokenKey)
       {
          HttpOnly = true,
          Value = _antiXsrfTokenValue
       };
       if (FormsAuthentication.RequireSSL && Request.IsSecureConnection)
       {
          responseCookie.Secure = true;
       }
       Response.Cookies.Set(responseCookie);
    }
    Page.PreLoad += master_Page_PreLoad;
}
protected void master_Page_PreLoad(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
       // Set Anti-XSRF token
       ViewState[AntiXsrfTokenKey] = Page.ViewStateUserKey;
       ViewState[AntiXsrfUserNameKey] = Context.User.Identity.Name ?? String.Empt

    }
    else
    {
       // Validate the Anti-XSRF token

if ((string)ViewState[AntiXsrfTokenKey] != _antiXsrfTokenValue ||
          (string)ViewState[AntiXsrfUserNameKey] != (Context.User.Identity.Name ?

       {
          throw new InvalidOperationException("Validation of Anti-
XSRF token failed.");
       }
    }
}
```

- Consider HSTS in IIS. See here for the procedure.
- This is a recommended `web.config` setup that handles HSTS among other things.

```
<?xml version="1.0" encoding="UTF-8"?>
 <configuration>
   <system.web>
     <httpRuntime enableVersionHeader="false"/>
   </system.web>
   <system.webServer>
     <security>
       <requestFiltering removeServerHeader="true" />
     </security>
     <staticContent>
       <clientCache cacheControlCustom="public"
            cacheControlMode="UseMaxAge"
            cacheControlMaxAge="1.00:00:00"
            setEtag="true" />
```

```
      </staticContent>
      <httpProtocol>
        <customHeaders>
          <add name="Content-Security-Policy"
             value="default-src 'none'; style-src 'self'; img-src 'self'; font-
src 'self'" />
          <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>
      <rewrite>
        <rules>
          <rule name="Redirect to https">
            <match url="(.*)"/>
            <conditions>
              <add input="{HTTPS}" pattern="Off"/>
              <add input="{REQUEST_METHOD}" pattern="^get$|^head$" />
            </conditions>
            <action type="Redirect" url="https://{HTTP_HOST}/{R:1}"
redirectType="Permanent"/>
          </rule>
        </rules>
        <outboundRules>
          <rule name="Add HSTS Header" enabled="true">
            <match serverVariable="RESPONSE_Strict_Transport_Security"
pattern=".*" />
            <conditions>
              <add input="{HTTPS}" pattern="on" ignoreCase="true" />
            </conditions>
            <action type="Rewrite" value="max-age=15768000" />
          </rule>
        </outboundRules>
      </rewrite>
    </system.webServer>
  </configuration>
```

- Remove the version header.

```
<httpRuntime enableVersionHeader="false" />
```

- Also remove the Server header.

```
HttpContext.Current.Response.Headers.Remove("Server");
```

HTTP validation and encoding

- Do not disable validateRequest 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.

- The 4.5 version of the .NET Frameworks includes the AntiXssEncoder library, which has a comprehensive input encoding library for the prevention of XSS. Use it.
- List allowable values anytime user input is accepted.
- Validate the URI format using Uri.IsWellFormedUriString.

Forms authentication

- Use cookies for persistence when possible. `Cookieless` auth will default to UseDeviceProfile.
- 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 is used this timeout resets after each request, so active users won't be affected.
- If HTTPS is not used, slidingExpiration should be disabled. Consider disabling slidingExpiration 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, 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 instead of ASP.NET Membership, and ASP.NET Identity uses PBKDF2 by default which is better. Review the OWASP Password Storage Cheat Sheet 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.

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 Pa

                + 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

Information about OS Injection can be found on this cheat sheet.

DO: Use System.Diagnostics.Process.Start to call underlying OS functions.

e.g

```
System.Diagnostics.Process process = new System.Diagnostics.Process();
System.Diagnostics.ProcessStartInfo startInfo = new
System.Diagnostics.ProcessStartInfo();
startInfo.FileName = "validatedCommand";
startInfo.Arguments = "validatedArg1 validatedArg2 validatedArg3";
process.StartInfo = startInfo;
process.Start();
```

DO: Use allow-list validation on all user supplied input. Input validation prevents improperly formed data from entering an information system. For more information please see the Input Validation Cheat Sheet.

e.g Validating user input using IPAddress.TryParse Method

```
//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
  ...
 }
    ...
}
```

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.

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.

A2 Broken Authentication

DO: Use ASP.net Core Identity. 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.

DO: Use a strong hash to store password credentials. For hash refer to this section.

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. 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.

DO: Use TLS 1.2 for your entire site. Get a free certificate LetsEncrypt.org.

DO NOT: Allow SSL, this is now obsolete.

DO: Have a strong TLS policy (see SSL Best Practices), use TLS 1.2 wherever possible. Then check the configuration using SSL Test or TestSSL.

DO: Ensure headers are not disclosing information about your application. See HttpHeaders.cs , Dionach StripHeaders, disable via `web.config` or startup.cs:

More information on Transport Layer Protection can be found here. 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.

A4 XML External Entities (XXE)

Please refer to the XXE cheat sheet so more detailed information, which can be found here.

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.

Below are the three most common XML Processing Options for .NET.

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 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 on Access Control and here 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 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 / .NET Core).

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.

If you are using tag-helpers, 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 attribute inside your `Startup.cs` as mentioned in the following article:

```
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 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 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 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.

More information can be found here 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, 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 for Cross-Site Scripting.

A8 Insecure Deserialization

Information about Insecure Deserialization can be found on this cheat sheet.

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

A9 Using Components with Known Vulnerabilities

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

DO: Keep your NuGet packages up to date, many will contain their own vulnerabilities.

DO: Run the OWASP Dependency Checker 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.

.NET Core come with a LoggerFactory, which is in Microsoft.Extensions.Logging. More information about ILogger can be found here.

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.

More information about Logging and Monitoring can be found here.

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.Passworc

        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
- Protect against a man in the middle attack for a user who has never been to your site before. Register for HSTS preload
- 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.

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

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. It has no default security configuration. Use WSHttpBinding 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 combines the two.
- Test your WCF implementation with a fuzzer like the ZAP.