.NET Security Cheat Sheet

From OWASP

Last revision (mm/dd/yy): **09/20/2017**

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 seperate 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 (http://nuget.codeplex.com/wikipage?title=Getting%20Started&referringTitle=Home). 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.

.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 strong named and versioned at the assembly level.

Data Access

- Use Parameterized SQL (http://msdn.microsoft.com/en-us/library/ms175528(v=sql.105).aspx) commands for all data access, without exception.
- Do not use SqlCommand (http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.aspx) with a string parameter made up of a concatenated SQL String (http://msdn.microsoft.com/en-us/library/ms182310.aspx).
- Whitelist allowable values coming from the user. Use enums, TryParse (http://msdn.microsoft.com/en-us/library/f02979c7.aspx) 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://msdn.microsoft.com/en-us/library/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 (http://msdn.microsoft.com/en-us/data/ef.aspx) is a very effective SQL injection (http://msdn.microsoft.com/en-us/library/ms161953(v=sql.105).aspx) prevention mechanism. Remember that building your own *ad hoc* queries in EF is just as susceptible to SQLi as a plain SQL query.
- When using SQL Server, prefer integrated authentication over SQL authentication.
- Use Always Encrypted (https://msdn.microsoft.com/en-us/library/mt163865.aspx) 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) (http://msdn.microsoft.com/en-us/library/ms995355.aspx) 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 (http://msdn.microsoft.com/en-us/library/system.security.cryptography.sha512.aspx).
  - In the .NET framework the strongest algorithm for password hashing is PBKDF2, implemented as System.Security.Cryptography.Rfc2898DeriveBytes (http://msdn.microsoft.com/en-us/library/system.security.cryptography.rfc2898derivebytes(v=vs.110).aspx).
  - 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.
- 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

- 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 (http://support.microsoft.com/kb/324069).
- Enable requireSSL (http://msdn.microsoft.com/en-us/library/system.web.configuration.httpcookiessection.requiressl.aspx) on cookies and form elements and HttpOnly (http://msdn.microsoft.com/en-us/library/system.web.configuration.httpcookiessection.httponlycookies.aspx) on cookies in the web.config.
- Implement customErrors (http://msdn.microsoft.com/en-us/library/h0hfz6fc(v=VS.71).aspx).
- Make sure tracing (http://www.iis.net/configreference/system.webserver/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 (http://msdn.microsoft.com/en-us/library/ms972969.aspx#securitybarriers_topic2):

```
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 requestCookieGuidValue))
    {
       // 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.Empty;
    }
    else
    {
       // Validate the Anti-XSRF token
       if ((string)ViewState[AntiXsrfTokenKey] != _antiXsrfTokenValue ||
          (string)ViewState[AntiXsrfUserNameKey] != (Context.User.Identity.Name ?? String.Empty))
       {
          throw new InvalidOperationException("Validation of Anti-XSRF token failed.");
       }
    }
}
```

- Consider HSTS (http://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security) in IIS.
  - In the Connections pane, go to the site, application, or directory for which you want to set a custom HTTP header.
  - In the Home pane, double-click HTTP Response Headers.
  - In the HTTP Response Headers pane, click Add... in the Actions pane.

- In the Add Custom HTTP Response Header dialog box, set the name and value for your custom header, and then click OK.
- 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="1; mode=block"/>
        <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 (http://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.
- 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.
- Whitelist allowable values anytime user input is accepted.
- Validate the URI format using Uri.IsWellFormedUriString (http://msdn.microsoft.com/en-us/library/system.uri.iswellformeduristring.aspx).

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 (http://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 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 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 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.

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: Practise 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

- **A2 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)
```

Find here (https://github.com/johnstaveley/SecurityEssentials/blob/master/SecurityEssentials/Core/Attributes/ThrottleAttribute.cs) the code to prevent throttling

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 behaviour: e.g. if the response takes 50% longer when the account is real then membership information can be guessed and tested.

- **A3 Cross Site Scripting**

DO NOT: Trust any data the user sends you, prefer white lists (always safe) over black 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'" />
            ...
```

- **A4 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);
    }
```

- **A5 Security Misconfiguration**

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: In 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
```

}

- **A6 Sensitive data exposure**

DO NOT: Store encrypted passwords.

DO: Use a strong hash to store password credentials. Use PBKDF2, BCrypt or SCrypt with at least 8000 iterations and a strong key.

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. Do not encrypt passwords. Protect encryption keys more than any other asset. 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.

DO: Use TLS 1.2 for your entire site. Get a free certificate from StartSSL.com (https://www.startssl.com/) or LetsEncrypt.org (https://letsencrypt.org/).

DO NOT: Allow SSL, this is now obsolete

DO: Have a strong TLS policy (see SSL Best Practises (http://www.ssllabs.com/projects/best-practises/)), use TLS 1.2 wherever possible. Then check the configuration using SSL Test (https://www.ssllabs.com/ssltest/)

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/) or disable via 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="1; mode=block"/>
      <remove name="X-Powered-By"/>
    </customHeaders>
  </httpProtocol>
```

- **A7 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)

- **A8 Cross site request forgery**

DO: Send the anti-forgery token with every Post/Put request:

```
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);
            }
        }
    }
```

NB: You will need to attach the anti-forgery token to Ajax requests.

- **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. [OWASP Dependency Checker (https://www.owasp.org/index.php/OWASP_Dependency_Check)]

- **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.

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 (http://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 run time.
- 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 Zed Attack Proxy.

Authors and Primary Editors

Bill Sempf - bill.sempf(at)owasp.org
Troy Hunt - troyhunt(at)hotmail.com
Jeremy Long - jeremy.long(at)owasp.org

Contributors

Shane Murnion John Staveley Steve Bamelis Xander Sherry

Other Cheatsheets

V · T · E (https://www.owasp.org/index.php?title=.NET_Security_Cheat_Sheet&action=edit)	Cheat Sheets [Collapse]
Developer / Builder	3rd Party Javascript Management · Access Control · AJAX Security Cheat Sheet · Authentication (ES) · Bean Validation Cheat Sheet · Choosing and Using Security Questions · Clickjacking Defense · Credential Stuffing Prevention Cheat Sheet · Cross-Site Request Forgery (CSRF) Prevention · Cryptographic Storage · C-Based Toolchain Hardening · Deserialization · DOM based XSS Prevention · Forgot Password · HTML5 Security · HTTP Strict Transport Security · Injection Prevention Cheat Sheet · Injection Prevention Cheat Sheet in Java · JSON Web Token (JWT) Cheat Sheet for Java · Input Validation · Insecure Direct Object Reference Prevention · JAAS · Key Management · LDAP Injection Prevention · Logging · Mass Assignment Cheat Sheet · **.NET Security** · OS Command Injection Defense Cheat Sheet · OWASP Top Ten · Password Storage · Pinning · Query Parameterization · REST Security · Ruby on Rails · Session Management · SAML Security · SQL Injection Prevention · Transaction Authorization · Transport Layer Protection · Unvalidated Redirects and Forwards · User Privacy Protection · Web Service Security · XSS (Cross Site Scripting) Prevention · XML External Entity (XXE) Prevention Cheat Sheet
Assessment / Breaker	Attack Surface Analysis · REST Assessment · Web Application Security Testing · XML Security Cheat Sheet · XSS Filter Evasion
Mobile	Android Testing · IOS Developer · Mobile Jailbreaking
OpSec / Defender	Virtual Patching · Vulnerability Disclosure
Draft and Beta	Application Security Architecture · Business Logic Security · Content Security Policy · Denial of Service Cheat Sheet · Grails Secure Code Review · IOS Application Security Testing · PHP Security · Regular Expression Security Cheatsheet · Secure Coding · Secure SDLC · Threat Modeling
All Pages In This Category	

Retrieved from "https://www.owasp.org/index.php?title=.NET_Security_Cheat_Sheet&oldid=233542"

Categories: Cheatsheets | OWASP .NET Project