```
[Authorize(Policy = "Over18")]
public class RegistrationController : Controller
{
    // Do Registration
```

Use multiple authentication schemes

Some apps may need to support multiple types of authentication. For example, your app might authenticate users from Azure Active Directory and from a users database. Another example is an app that authenticates users from both Active Directory Federation Services and Azure Active Directory B2C. In this case, the app should accept a JWT bearer token from several issuers.

Add all authentication schemes you'd like to accept. For example, the following code adds two JWT bearer authentication schemes with different issuers:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder(args);

// Authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://localhost:5000/identity/";
        })
        .AddJwtBearer("AzureAD", options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://login.microsoftonline.com/eb971100-
7f436/";
        });

// Authorization
builder.Services.AddAuthorization(options =>
{
    var defaultAuthorizationPolicyBuilder = new AuthorizationPolicyBuilder(
        JwtBearerDefaults.AuthenticationScheme,
        "AzureAD");
    defaultAuthorizationPolicyBuilder =
        defaultAuthorizationPolicyBuilder.RequireAuthenticatedUser();
    options.DefaultPolicy = defaultAuthorizationPolicyBuilder.Build();
});

builder.Services.AddAuthentication()
```

```
        .AddIdentityServerJwt();

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();
app.UseIdentityServer();
app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.MapFallbackToFile("index.html");

app.Run();
```

> **Note**
>
> Only one JWT bearer authentication is registered with the default authentication scheme `JwtBearerDefaults.AuthenticationScheme`. Additional authentication has to be registered with a unique authentication scheme.

Update the default authorization policy to accept both authentication schemes. For example:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder(args);

// Authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
```

```
        .AddJwtBearer(options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://localhost:5000/identity/";
        })
        .AddJwtBearer("AzureAD", options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://login.microsoftonline.com/eb971100-
7f436/";
        });

// Authorization
builder.Services.AddAuthorization(options =>
{
    var defaultAuthorizationPolicyBuilder = new AuthorizationPolicyBuilder(
        JwtBearerDefaults.AuthenticationScheme,
        "AzureAD");
    defaultAuthorizationPolicyBuilder =
        defaultAuthorizationPolicyBuilder.RequireAuthenticatedUser();
    options.DefaultPolicy = defaultAuthorizationPolicyBuilder.Build();
});

builder.Services.AddAuthentication()
        .AddIdentityServerJwt();

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();
app.UseIdentityServer();
app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.MapFallbackToFile("index.html");

app.Run();
```

As the default authorization policy is overridden, it's possible to use the `[Authorize]` attribute in controllers. The controller then accepts requests with JWT issued by the first or second issuer.

See this GitHub issue on using multiple authentication schemes.

The following example uses Azure Active Directory B2C and another Azure Active Directory tenant:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.IdentityModel.Tokens;
using Microsoft.Net.Http.Headers;
using System.IdentityModel.Tokens.Jwt;

var builder = WebApplication.CreateBuilder(args);

// Authentication
```

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme = "B2C_OR_AAD";
    options.DefaultChallengeScheme = "B2C_OR_AAD";
})
.AddJwtBearer("B2C", jwtOptions =>
{
    jwtOptions.MetadataAddress = "B2C-MetadataAddress";
    jwtOptions.Authority = "B2C-Authority";
    jwtOptions.Audience = "B2C-Audience";
})
.AddJwtBearer("AAD", jwtOptions =>
{
    jwtOptions.MetadataAddress = "AAD-MetadataAddress";
    jwtOptions.Authority = "AAD-Authority";
    jwtOptions.Audience = "AAD-Audience";
    jwtOptions.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateIssuerSigningKey = true,
        ValidAudiences =
builder.Configuration.GetSection("ValidAudiences").Get<string[]>(),
        ValidIssuers =
builder.Configuration.GetSection("ValidIssuers").Get<string[]>()
    };
})
.AddPolicyScheme("B2C_OR_AAD", "B2C_OR_AAD", options =>
{
    options.ForwardDefaultSelector = context =>
    {
        string authorization =
context.Request.Headers[HeaderNames.Authorization];
        if (!string.IsNullOrEmpty(authorization) &&
authorization.StartsWith("Bearer "))
        {
            var token = authorization.Substring("Bearer ".Length).Trim();
            var jwtHandler = new JwtSecurityTokenHandler();

            return (jwtHandler.CanReadToken(token) &&
jwtHandler.ReadJwtToken(token).Issuer.Equals("B2C-Authority"))
                ? "B2C" : "AAD";
        }
        return "AAD";
    };
});
```

```
builder.Services.AddAuthentication()
        .AddIdentityServerJwt();

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();
```

```
if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();
app.UseIdentityServer();
app.UseAuthorization();

app.MapDefaultControllerRoute().RequireAuthorization();
app.MapRazorPages().RequireAuthorization();

app.MapFallbackToFile("index.html");

app.Run();
```

In the preceding code, ForwardDefaultSelector is used to select a default scheme for the current request that authentication handlers should forward all authentication operations to by default. The default forwarding logic checks the most specific ForwardAuthenticate, ForwardChallenge, ForwardForbid, ForwardSignIn, and ForwardSignOut setting first, followed by checking the ForwardDefaultSelector, followed by ForwardDefault. The first non null result is used as the target scheme to forward to. For more information, see Policy schemes in ASP.NET Core.

ASP.NET Core Data Protection Overview

Article • 06/17/2024

ASP.NET Core provides a cryptographic API to protect data, including key management and rotation.

Web apps often need to store sensitive data. The Windows data protection API (DPAPI) isn't intended for use in web apps.

The ASP.NET Core data protection stack was designed to:

- Provide a built in solution for most Web scenarios.
- Address many of the deficiencies of the previous encryption system.
- Serve as the replacement for the `<machineKey>` element in ASP.NET 1.x - 4.x.

Problem statement

I need to persist trusted information for later retrieval, but I don't trust the persistence mechanism. In web terms, this might be written as *I need to round-trip trusted state via an untrusted client.*

Authenticity, integrity, and tamper-proofing is a requirement. The canonical example of this is an authentication cookie or bearer token. The server generates an ***I am Groot and have xyz permissions*** token and sends it to the client. The client presents that token back to the server, but the server needs some kind of assurance that the client hasn't forged the token.

Confidentiality is a requirement. Since the persisted state is trusted by the server, this state could contain information that shouldn't be disclosed to an untrusted client. For example:

- A file path.
- A permission.
- A handle or other indirect reference.
- Some server-specific data.

Isolation is a requirement. Since modern apps are componentized, individual components want to take advantage of this system without regard to other components in the system. For instance, consider a bearer token component using this stack. It should operate without any interference, for example, from an anti-CSRF mechanism also using the same stack.

Some common assumptions can narrow the scope of requirements:

- All services operating within the cryptosystem are equally trusted.
- The data doesn't need to be generated or consumed outside of the services under our direct control.
- Operations must be fast since each request to the web service might go through the cryptosystem one or more times. The speed requirement makes symmetric cryptography ideal. Asymmetric cryptography isn't used until it's required.

Design philosophy

ASP.NET Core data protection is an easy to use data protection stack. It's based on the following principles:

- Ease of configuration. The system strives for zero configuration. In situations where developers need to configure a specific aspect, such as the key repository, those specific configurations aren't difficult.
- Offer a basic consumer-facing API. The APIs are straight forward to use correctly and difficult to use incorrectly.
- Developers don't have to learn key management principles. The system handles algorithm selection and key lifetime on behalf of the developer. The developer doesn't have access to the raw key material.
- Keys are protected at rest as much as possible. The system figures out an appropriate default protection mechanism and applies it automatically.

The data protection APIs aren't primarily intended for indefinite persistence of confidential payloads. Other technologies, such as Windows CNG DPAPI and Azure Rights Management are more suited to the scenario of indefinite storage. They have correspondingly strong key management capabilities. That said, the ASP.NET Core data protection APIs can be used for long-term protection of confidential data.

Audience

The data protection system provides APIs that target three main audiences:

1. The consumer APIs target application and framework developers.

   I don't want to learn about how the stack operates or about how it's configured. I just want to perform some operation with high probability of using the APIs successfully.

2. The configuration APIs target app developers and system administrators.

I need to tell the data protection system that my environment requires non-default paths or settings.

3. The extensibility APIs target developers in charge of implementing custom policy. Usage of these APIs is limited to rare situations and developers with security experience.

   I need to replace an entire component within the system because I have truly unique behavioral requirements. I'm willing to learn uncommonly used parts of the API surface in order to build a plugin that fulfills my requirements.

Package layout

The data protection stack consists of five packages:

- Microsoft.AspNetCore.DataProtection.Abstractions contains:
  - IDataProtectionProvider and IDataProtector interfaces to create data protection services.
  - Useful extension methods for working with these types. for example, IDataProtector.Protect

  If the data protection system is instantiated elsewhere and you're consuming the API, reference `Microsoft.AspNetCore.DataProtection.Abstractions`.

- Microsoft.AspNetCore.DataProtection contains the core implementation of the data protection system, including:
  - Core cryptographic operations.
  - Key management.
  - Configuration and extensibility.

  To instantiate the data protection system, reference `Microsoft.AspNetCore.DataProtection`. You might need to reference the data protection system when:
  - Adding it to an IServiceCollection.
  - Modifying or extending its behavior.

- Microsoft.AspNetCore.DataProtection.Extensions contains additional APIs which developers might find useful but which don't belong in the core package. For instance, this package contains:
  - Factory methods to instantiate the data protection system to store keys at a location on the file system without dependency injection. See DataProtectionProvider.

    - Extension methods for limiting the lifetime of protected payloads. See ITimeLimitedDataProtector.

- Microsoft.AspNetCore.DataProtection.SystemWeb can be installed into an existing ASP.NET 4.x app to redirect its `<machineKey>` operations to use the new ASP.NET Core data protection stack. For more information, see Replace the ASP.NET machineKey in ASP.NET Core.

- Microsoft.AspNetCore.Cryptography.KeyDerivation provides an implementation of the PBKDF2 password hashing routine and can be used by systems that must handle user passwords securely. For more information, see Hash passwords in ASP.NET Core.

Additional resources

- Get started with the Data Protection APIs in ASP.NET Core
- Host ASP.NET Core in a web farm

Get started with the Data Protection APIs in ASP.NET Core

Article • 06/17/2024

Basically, protecting data consists of the following steps:

1. Create a data protector from a data protection provider.
2. Call the `Protect` method with the data you want to protect.
3. Call the `Unprotect` method with the data you want to turn back into plain text.

Most frameworks and app models, such as ASP.NET Core or SignalR, already configure the data protection system and add it to a service container that is accessed via dependency injection. The following sample demonstrates:

- Configuring a service container for dependency injection and registering the data protection stack.
- Receiving the data protection provider via DI.
- Creating a protector.
- Protecting then unprotecting data.

C#

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
```

```
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload =
_protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZZhlAlTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
 */
```

When you create a protector you must provide one or more Purpose Strings. A purpose string provides isolation between consumers. For example, a protector created with a purpose string of "green" wouldn't be able to unprotect data provided by a protector with a purpose of "purple".

> **Tip**
>
> Instances of `IDataProtectionProvider` and `IDataProtector` are thread-safe for multiple callers. It's intended that once a component gets a reference to an `IDataProtector` via a call to `CreateProtector`, it will use that reference for multiple calls to `Protect` and `Unprotect`.
>
> A call to `Unprotect` will throw CryptographicException if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch CryptographicException instead of swallowing all exceptions.

Use AddOptions to configure custom repository

Consider the following code which uses a service provider because the implementation of `IXmlRepository` has a dependency on a singleton service:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    // ...

    var sp = services.BuildServiceProvider();
    services.AddDataProtection()
      .AddKeyManagementOptions(o => o.XmlRepository =
sp.GetService<IXmlRepository>());
}
```

The preceding code logs the following warning:

Calling 'BuildServiceProvider' from application code results in an additional copy of singleton services being created. Consider alternatives such as dependency injecting services as parameters to 'Configure'.

The following code provides the `IXmlRepository` implementation without having to build the service provider and therefore making additional copies of singleton services:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<DataProtectionDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    // Register XmlRepository for data protection.
    services.AddOptions<KeyManagementOptions>()
    .Configure<IServiceScopeFactory>((options, factory) =>
    {
        options.XmlRepository = new CustomXmlRepository(factory);
    });

    services.AddRazorPages();
}
```

The preceding code removes the call to `GetService` and hides `IConfigureOptions<T>`.

The following code shows the custom XML repository:

C#

```
using CustomXMLrepo.Data;
using Microsoft.AspNetCore.DataProtection.Repositories;
using Microsoft.Extensions.DependencyInjection;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

public class CustomXmlRepository : IXmlRepository
{
    private readonly IServiceScopeFactory factory;

    public CustomXmlRepository(IServiceScopeFactory factory)
    {
        this.factory = factory;
    }

    public IReadOnlyCollection<XElement> GetAllElements()
    {
        using (var scope = factory.CreateScope())
        {
            var context =
scope.ServiceProvider.GetRequiredService<DataProtectionDbContext>();
            var keys = context.XmlKeys.ToList()
                .Select(x => XElement.Parse(x.Xml))
                .ToList();
            return keys;
        }
    }

    public void StoreElement(XElement element, string friendlyName)
    {
        var key = new XmlKey
        {
            Xml = element.ToString(SaveOptions.DisableFormatting)
        };

        using (var scope = factory.CreateScope())
        {
            var context =
scope.ServiceProvider.GetRequiredService<DataProtectionDbContext>();
            context.XmlKeys.Add(key);
            context.SaveChanges();
        }
    }
}
```

The following code shows the XmlKey class:

C#

```
public class XmlKey
{
    public Guid Id { get; set; }
    public string Xml { get; set; }

    public XmlKey()
    {
        this.Id = Guid.NewGuid();
    }
}
```

Consumer APIs overview for ASP.NET Core

Article • 06/17/2024

The `IDataProtectionProvider` and `IDataProtector` interfaces are the basic interfaces through which consumers use the data protection system. They're located in the Microsoft.AspNetCore.DataProtection.Abstractions package.

IDataProtectionProvider

The provider interface represents the root of the data protection system. It cannot directly be used to protect or unprotect data. Instead, the consumer must get a reference to an `IDataProtector` by calling `IDataProtectionProvider.CreateProtector(purpose)`, where purpose is a string that describes the intended consumer use case. See Purpose Strings for much more information on the intent of this parameter and how to choose an appropriate value.

IDataProtector

The protector interface is returned by a call to `CreateProtector`, and it's this interface which consumers can use to perform protect and unprotect operations.

To protect a piece of data, pass the data to the `Protect` method. The basic interface defines a method which converts byte[] -> byte[], but there's also an overload (provided as an extension method) which converts string -> string. The security offered by the two methods is identical; the developer should choose whichever overload is most convenient for their use case. Irrespective of the overload chosen, the value returned by the Protect method is now protected (enciphered and tamper-proofed), and the application can send it to an untrusted client.

To unprotect a previously-protected piece of data, pass the protected data to the `Unprotect` method. (There are byte[]-based and string-based overloads for developer convenience.) If the protected payload was generated by an earlier call to `Protect` on this same `IDataProtector`, the `Unprotect` method will return the original unprotected payload. If the protected payload has been tampered with or was produced by a different `IDataProtector`, the `Unprotect` method will throw CryptographicException.

The concept of same vs. different `IDataProtector` ties back to the concept of purpose. If two `IDataProtector` instances were generated from the same root `IDataProtectionProvider` but via different purpose strings in the call to `IDataProtectionProvider.CreateProtector`, then they're considered different protectors, and one won't be able to unprotect payloads generated by the other.

Consuming these interfaces

For a DI-aware component, the intended usage is that the component takes an `IDataProtectionProvider` parameter in its constructor and that the DI system automatically provides this service when the component is instantiated.

> **Note**
>
> Some applications (such as console applications or ASP.NET 4.x applications) might not be DI-aware so cannot use the mechanism described here. For these scenarios consult the Non DI Aware Scenarios document for more information on getting an instance of an `IDataProtection` provider without going through DI.

The following sample demonstrates three concepts:

1. Add the data protection system to the service container,
2. Using DI to receive an instance of an `IDataProtectionProvider`, and
3. Creating an `IDataProtector` from an `IDataProtectionProvider` and using it to protect and unprotect data.

Console app

C#

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
```

```
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload =
_protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZZhlAlTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
 */
```

Web app

Call AddDataProtection(IServiceCollection, Action<DataProtectionOptions>) in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);
```

```
// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddDataProtection();

var app = builder.Build();
```

The following highlighted code shows how to use IDataProtector in a controller:

C#

```
public class HomeController : Controller
{
    private readonly IDataProtector _dataProtector;

    public HomeController(IDataProtectionProvider dataProtectionProvider)
    {
        _dataProtector =
dataProtectionProvider.CreateProtector("HomeControllerPurpose");
    }

    // ...

    public IActionResult Privacy()
    {
        // The original data to protect
        string originalData = "original data";

        // Protect the data (encrypt)
        string protectedData = _dataProtector.Protect(originalData);
        Console.WriteLine($"Protected Data: {protectedData}");

        // Unprotect the data (decrypt)
        string unprotectedData = _dataProtector.Unprotect(protectedData);
        Console.WriteLine($"Unprotected Data: {unprotectedData}");

        return View();
    }

    // ...
```

The package `Microsoft.AspNetCore.DataProtection.Abstractions` contains an extension method GetDataProtector as a developer convenience. It encapsulates as a single operation both retrieving an IDataProtectionProvider from the service provider and calling IDataProtectionProvider.CreateProtector. The following sample demonstrates its usage:

C#

```
using System;
using Microsoft.AspNetCore.DataProtection;
```

```
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // get an IDataProtector from the IServiceProvider
        var protector = services.GetDataProtector("Contoso.Example.v2");
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
    }
}
```

Tip

Instances of `IDataProtectionProvider` and `IDataProtector` are thread-safe for multiple callers. It's intended that once a component gets a reference to an `IDataProtector` via a call to `CreateProtector`, it will use that reference for multiple calls to `Protect` and `Unprotect`. A call to `Unprotect` will throw CryptographicException if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch CryptographicException instead of swallowing all exceptions.

Purpose strings in ASP.NET Core

Article • 06/17/2024

Components which consume `IDataProtectionProvider` must pass a unique *purposes* parameter to the `CreateProtector` method. The purposes *parameter* is inherent to the security of the data protection system, as it provides isolation between cryptographic consumers, even if the root cryptographic keys are the same.

When a consumer specifies a purpose, the purpose string is used along with the root cryptographic keys to derive cryptographic subkeys unique to that consumer. This isolates the consumer from all other cryptographic consumers in the application: no other component can read its payloads, and it cannot read any other component's payloads. This isolation also renders infeasible entire categories of attack against the component.

In the diagram above, `IDataProtector` instances A and B **cannot** read each other's payloads, only their own.

The purpose string doesn't have to be secret. It should simply be unique in the sense that no other well-behaved component will ever provide the same purpose string.

> **Tip**
>
> Using the namespace and type name of the component consuming the data protection APIs is a good rule of thumb, as in practice this information will never conflict.
>
> A Contoso-authored component which is responsible for minting bearer tokens might use Contoso.Security.BearerToken as its purpose string. Or - even better - it might use Contoso.Security.BearerToken.v1 as its purpose string. Appending the version number allows a future version to use Contoso.Security.BearerToken.v2 as

its purpose, and the different versions would be completely isolated from one another as far as payloads go.

Since the purposes parameter to `CreateProtector` is a string array, the above could've been instead specified as `[ "Contoso.Security.BearerToken", "v1" ]`. This allows establishing a hierarchy of purposes and opens up the possibility of multi-tenancy scenarios with the data protection system.

> ⚠ **Warning**
>
> Components shouldn't allow untrusted user input to be the sole source of input for the purposes chain.
>
> For example, consider a component Contoso.Messaging.SecureMessage which is responsible for storing secure messages. If the secure messaging component were to call `CreateProtector([ username ])`, then a malicious user might create an account with username "Contoso.Security.BearerToken" in an attempt to get the component to call `CreateProtector([ "Contoso.Security.BearerToken" ])`, thus inadvertently causing the secure messaging system to mint payloads that could be perceived as authentication tokens.
>
> A better purposes chain for the messaging component would be `CreateProtector([ "Contoso.Messaging.SecureMessage", $"User: {username}" ])`, which provides proper isolation.

The isolation provided by and behaviors of `IDataProtectionProvider`, `IDataProtector`, and purposes are as follows:

- For a given `IDataProtectionProvider` object, the `CreateProtector` method will create an `IDataProtector` object uniquely tied to both the `IDataProtectionProvider` object which created it and the purposes parameter which was passed into the method.
- The purpose parameter must not be null. (If purposes is specified as an array, this means that the array must not be of zero length and all elements of the array must be non-null.) An empty string purpose is technically allowed but is discouraged.
- Two purposes arguments are equivalent if and only if they contain the same strings (using an ordinal comparer) in the same order. A single purpose argument is equivalent to the corresponding single-element purposes array.

- Two `IDataProtector` objects are equivalent if and only if they're created from equivalent `IDataProtectionProvider` objects with equivalent purposes parameters.
- For a given `IDataProtector` object, a call to `Unprotect(protectedData)` will return the original `unprotectedData` if and only if `protectedData := Protect(unprotectedData)` for an equivalent `IDataProtector` object.

> **Note**
>
> We're not considering the case where some component intentionally chooses a purpose string which is known to conflict with another component. Such a component would essentially be considered malicious, and this system isn't intended to provide security guarantees in the event that malicious code is already running inside of the worker process.

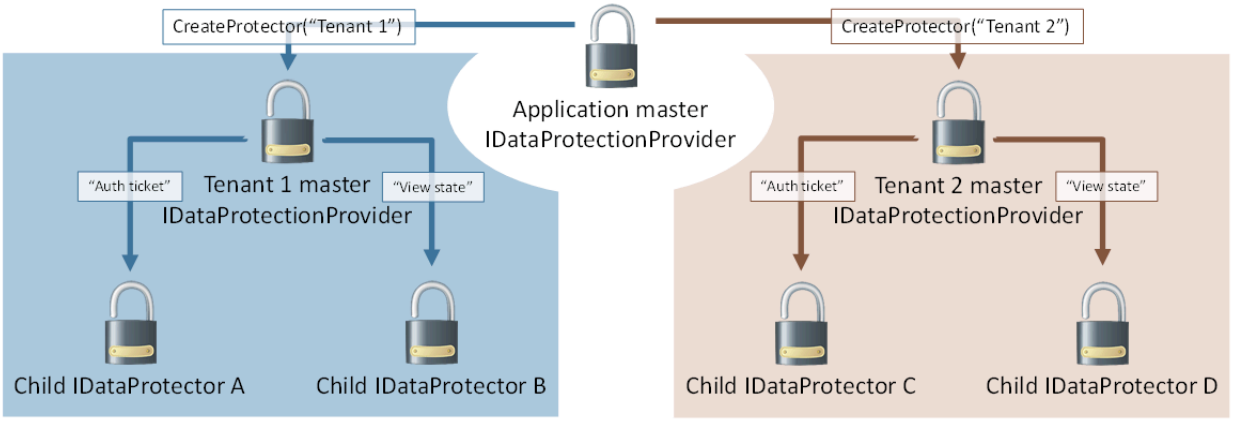
Purpose hierarchy and multi-tenancy in ASP.NET Core

Article • 06/03/2022

Since an `IDataProtector` is also implicitly an `IDataProtectionProvider`, purposes can be chained together. In this sense, `provider.CreateProtector([ "purpose1", "purpose2" ])` is equivalent to `provider.CreateProtector("purpose1").CreateProtector("purpose2")`.

This allows for some interesting hierarchical relationships through the data protection system. In the earlier example of Contoso.Messaging.SecureMessage, the SecureMessage component can call `provider.CreateProtector("Contoso.Messaging.SecureMessage")` once up-front and cache the result into a private `_myProvider` field. Future protectors can then be created via calls to `_myProvider.CreateProtector("User: username")`, and these protectors would be used for securing the individual messages.

This can also be flipped. Consider a single logical application which hosts multiple tenants (a CMS seems reasonable), and each tenant can be configured with its own authentication and state management system. The umbrella application has a single master provider, and it calls `provider.CreateProtector("Tenant 1")` and `provider.CreateProtector("Tenant 2")` to give each tenant its own isolated slice of the data protection system. The tenants could then derive their own individual protectors based on their own needs, but no matter how hard they try they cannot create protectors which collide with any other tenant in the system. Graphically, this is represented as below.



> ⚠ **Warning**
>
> This assumes the umbrella application controls which APIs are available to individual tenants and that tenants cannot execute arbitrary code on the server. If a

tenant can execute arbitrary code, they could perform private reflection to break the isolation guarantees, or they could just read the master keying material directly and derive whatever subkeys they desire.

The data protection system actually uses a sort of multi-tenancy in its default out-of-the-box configuration. By default master keying material is stored in the worker process account's user profile folder (or the registry, for IIS application pool identities). But it's actually fairly common to use a single account to run multiple applications, and thus all these applications would end up sharing the master keying material. To solve this, the data protection system automatically inserts a unique-per-application identifier as the first element in the overall purpose chain. This implicit purpose serves to isolate individual applications from one another by effectively treating each application as a unique tenant within the system, and the protector creation process looks identical to the image above.

Hash passwords in ASP.NET Core

Article • 07/19/2022

This article shows how to call the KeyDerivation.Pbkdf2 method which allows hashing a password using the PBKDF2 algorithm.

> **⚠ Warning**
>
> The `KeyDerivation.Pbkdf2` API is a low-level cryptographic primitive and is intended to be used to integrate apps into an existing protocol or cryptographic system. `KeyDerivation.Pbkdf2` should not be used in new apps which support password based login and need to store hashed passwords in a datastore. New apps should use PasswordHasher. For more information on `PasswordHasher`, see Exploring the ASP.NET Core Identity PasswordHasher.

The data protection code base includes a NuGet package Microsoft.AspNetCore.Cryptography.KeyDerivation which contains cryptographic key derivation functions. This package is a standalone component and has no dependencies on the rest of the data protection system. It can be used independently. The source exists alongside the data protection code base as a convenience.

> **⚠ Warning**
>
> The following code shows how to use `KeyDerivation.Pbkdf2` to generate a shared secret key. It should not be used to hash a password for storage in a datastore.

```
using Microsoft.AspNetCore.Cryptography.KeyDerivation;
using System.Security.Cryptography;

Console.Write("Enter a password: ");
string? password = Console.ReadLine();

// Generate a 128-bit salt using a sequence of
// cryptographically strong random bytes.
byte[] salt = RandomNumberGenerator.GetBytes(128 / 8); // divide by 8 to
convert bits to bytes
Console.WriteLine($"Salt: {Convert.ToBase64String(salt)}");

// derive a 256-bit subkey (use HMACSHA256 with 100,000 iterations)
string hashed = Convert.ToBase64String(KeyDerivation.Pbkdf2(
    password: password!,
    salt: salt,
    prf: KeyDerivationPrf.HMACSHA256,
```

```
    iterationCount: 100000,
    numBytesRequested: 256 / 8));

Console.WriteLine($"Hashed: {hashed}");

/*
 * SAMPLE OUTPUT
 *
 * Enter a password: Xtw9NMgx
 * Salt: CGYzqeN4plZekNC88Umm1Q==
 * Hashed: Gt9Yc4AiIvmsC1QQbe2RZsCIqvoYlst2xbz0Fs8aHnw=
 */
```

See the source code for ASP.NET Core Identity's `PasswordHasher` type for a real-world use case.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Limit the lifetime of protected payloads in ASP.NET Core

Article • 06/03/2022

There are scenarios where the application developer wants to create a protected payload that expires after a set period of time. For instance, the protected payload might represent a password reset token that should only be valid for one hour. It's certainly possible for the developer to create their own payload format that contains an embedded expiration date, and advanced developers may wish to do this anyway, but for the majority of developers managing these expirations can grow tedious.

To make this easier for our developer audience, the package Microsoft.AspNetCore.DataProtection.Extensions contains utility APIs for creating payloads that automatically expire after a set period of time. These APIs implement the `ITimeLimitedDataProtector` interface.

API usage

The `ITimeLimitedDataProtector` interface is the core interface for protecting and unprotecting time-limited / self-expiring payloads. To create an instance of an `ITimeLimitedDataProtector`, you'll first need an instance of a regular IDataProtector constructed with a specific purpose. Once the `IDataProtector` instance is available, call the `IDataProtector.ToTimeLimitedDataProtector` extension method to get back a protector with built-in expiration capabilities.

`ITimeLimitedDataProtector` exposes the following API surface and extension methods:

- CreateProtector(string purpose) : ITimeLimitedDataProtector - This API is similar to the existing `IDataProtectionProvider.CreateProtector` in that it can be used to create purpose chains from a root time-limited protector.
- Protect(byte[] plaintext, DateTimeOffset expiration) : byte[]
- Protect(byte[] plaintext, TimeSpan lifetime) : byte[]
- Protect(byte[] plaintext) : byte[]
- Protect(string plaintext, DateTimeOffset expiration) : string
- Protect(string plaintext, TimeSpan lifetime) : string

- Protect(string plaintext) : string

In addition to the core `Protect` methods which take only the plaintext, there are new overloads which allow specifying the payload's expiration date. The expiration date can be specified as an absolute date (via a `DateTimeOffset`) or as a relative time (from the current system time, via a `TimeSpan`). If an overload which doesn't take an expiration is called, the payload is assumed never to expire.

- Unprotect(byte[] protectedData, out DateTimeOffset expiration) : byte[]
- Unprotect(byte[] protectedData) : byte[]
- Unprotect(string protectedData, out DateTimeOffset expiration) : string
- Unprotect(string protectedData) : string

The `Unprotect` methods return the original unprotected data. If the payload hasn't yet expired, the absolute expiration is returned as an optional out parameter along with the original unprotected data. If the payload is expired, all overloads of the Unprotect method will throw CryptographicException.

> ⚠ **Warning**
>
> It's not advised to use these APIs to protect payloads which require long-term or indefinite persistence. "Can I afford for the protected payloads to be permanently unrecoverable after a month?" can serve as a good rule of thumb; if the answer is no then developers should consider alternative APIs.

The sample below uses the non-DI code paths for instantiating the data protection system. To run this sample, ensure that you have first added a reference to the Microsoft.AspNetCore.DataProtection.Extensions package.

C#

```
using System;
using System.IO;
using System.Threading;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // create a protector for my application

        var provider = DataProtectionProvider.Create(new
```

```
DirectoryInfo(@"c:\myapp-keys\"));
        var baseProtector =
provider.CreateProtector("Contoso.TimeLimitedSample");

        // convert the normal protector into a time-limited protector
        var timeLimitedProtector =
baseProtector.ToTimeLimitedDataProtector();

        // get some input and protect it for five seconds
        Console.Write("Enter input: ");
        string input = Console.ReadLine();
        string protectedData = timeLimitedProtector.Protect(input, lifetime:
TimeSpan.FromSeconds(5));
        Console.WriteLine($"Protected data: {protectedData}");

        // unprotect it to demonstrate that round-tripping works properly
        string roundtripped = timeLimitedProtector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped data: {roundtripped}");

        // wait 6 seconds and perform another unprotect, demonstrating that
the payload self-expires
        Console.WriteLine("Waiting 6 seconds...");
        Thread.Sleep(6000);
        timeLimitedProtector.Unprotect(protectedData);
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello!
 * Protected data: CfDJ8Hu5z0zwxn...nLk7Ok
 * Round-tripped data: Hello!
 * Waiting 6 seconds...
 * <<throws CryptographicException with message 'The payload expired at
...'>>

 */
```

Unprotect payloads whose keys have been revoked in ASP.NET Core

Article • 09/27/2024

The ASP.NET Core data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like Windows CNG DPAPI and Azure Rights Management are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there's nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data. Keys are never removed from the key ring, so `IDataProtector.Unprotect` can always recover existing payloads as long as the keys are available and valid.

However, an issue arises when the developer tries to unprotect data that has been protected with a revoked key, as `IDataProtector.Unprotect` will throw an exception in this case. This might be fine for short-lived or transient payloads (like authentication tokens), as these kinds of payloads can easily be recreated by the system, and at worst the site visitor might be required to log in again. But for persisted payloads, having `Unprotect` throw could lead to unacceptable data loss.

IPersistedDataProtector

To support the scenario of allowing payloads to be unprotected even in the face of revoked keys, the data protection system contains an `IPersistedDataProtector` type. To get an instance of `IPersistedDataProtector`, simply get an instance of `IDataProtector` in the normal fashion and try casting the `IDataProtector` to `IPersistedDataProtector`.

> **Note**
>
> Not all `IDataProtector` instances can be cast to `IPersistedDataProtector`. Developers should use the C# as operator or similar to avoid runtime exceptions caused by invalid casts, and they should be prepared to handle the failure case appropriately.

`IPersistedDataProtector` exposes the following API surface:

```
C#
```

```
DangerousUnprotect(byte[] protectedData, bool ignoreRevocationErrors,
     out bool requiresMigration, out bool wasRevoked) : byte[]
```

This API takes the protected payload (as a byte array) and returns the unprotected payload. There's no string-based overload. The two out parameters are as follows.

- `requiresMigration`: Is set to `true` if the key used to protect this payload is no longer the active default key. For example, the key used to protect this payload is old and a key rolling operation has since taken place. The caller may wish to consider reprotecting the payload depending on their business needs.
- `wasRevoked`: Is set to `true` if the key used to protect this payload was revoked.

> ⚠ **Warning**
>
> Exercise extreme caution when passing `ignoreRevocationErrors: true` to the `DangerousUnprotect` method. If after calling this method the `wasRevoked` value is true, then the key used to protect this payload was revoked, and the payload's authenticity should be treated as suspect. In this case, only continue operating on the unprotected payload if you have some separate assurance that it's authentic, for example that it's coming from a secure database rather than being sent by an untrusted web client.

C#

```
using System;
using System.IO;
using System.Text;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // get a protector and perform a protect operation
        var protector =
services.GetDataProtector("Sample.DangerousUnprotect");
```

```
        Console.Write("Input: ");
        byte[] input = Encoding.UTF8.GetBytes(Console.ReadLine());
        var protectedData = protector.Protect(input);
        Console.WriteLine($"Protected payload:
{Convert.ToBase64String(protectedData)}");

        // demonstrate that the payload round-trips properly
        var roundTripped = protector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped payload:
{Encoding.UTF8.GetString(roundTripped)}");

        // get a reference to the key manager and revoke all keys in the key
ring
        var keyManager = services.GetService<IKeyManager>();
        Console.WriteLine("Revoking all keys in the key ring...");
        keyManager.RevokeAllKeys(DateTimeOffset.Now, "Sample revocation.");

        // try calling Protect - this should throw
        Console.WriteLine("Calling Unprotect...");
        try
        {
            var unprotectedPayload = protector.Unprotect(protectedData);
            Console.WriteLine($"Unprotected payload:
{Encoding.UTF8.GetString(unprotectedPayload)}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
        }

        // try calling DangerousUnprotect
        Console.WriteLine("Calling DangerousUnprotect...");
        try
        {
            IPersistedDataProtector persistedProtector = protector as
IPersistedDataProtector;
            if (persistedProtector == null)
            {
                throw new Exception("Can't call DangerousUnprotect.");
            }

            bool requiresMigration, wasRevoked;
            var unprotectedPayload = persistedProtector.DangerousUnprotect(
                protectedData: protectedData,
                ignoreRevocationErrors: true,
                requiresMigration: out requiresMigration,
                wasRevoked: out wasRevoked);
            Console.WriteLine($"Unprotected payload:
{Encoding.UTF8.GetString(unprotectedPayload)}");
            Console.WriteLine($"Requires migration = {requiresMigration},
was revoked = {wasRevoked}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
```

```
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Input: Hello!
 * Protected payload: CfDJ8LHIzUCX1ZVBn2BZ...
 * Round-tripped payload: Hello!
 * Revoking all keys in the key ring...
 * Calling Unprotect...
 * CryptographicException: The key {...} has been revoked.
 * Calling DangerousUnprotect...
 * Unprotected payload: Hello!
 * Requires migration = True, was revoked = True
 */
```

Data Protection configuration in ASP.NET Core

Article • 06/03/2022

Visit these topics to learn about Data Protection configuration in ASP.NET Core:

- Configure ASP.NET Core Data Protection
  An overview on configuring ASP.NET Core Data Protection.
- Data Protection key management and lifetime
  Information on Data Protection key management and lifetime.
- Data Protection machine-wide policy support
  Details on setting a default machine-wide policy for all apps that use Data Protection.
- Non-DI aware scenarios for Data Protection in ASP.NET Core
  How to use the DataProtectionProvider concrete type to use Data Protection without going through DI-specific code paths.

Configure ASP.NET Core Data Protection

Article • 11/04/2024

When the Data Protection system is initialized, it applies default settings based on the operational environment. These settings are appropriate for apps running on a single machine. However, there are cases where a developer may want to change the default settings:

- The app is spread across multiple machines.
- For compliance reasons.

For these scenarios, the Data Protection system offers a rich configuration API.

> **⚠ Warning**
>
> Similar to configuration files, the data protection key ring should be protected using appropriate permissions. You can choose to encrypt keys at rest, but this doesn't prevent cyberattackers from creating new keys. Consequently, your app's security is impacted. The storage location configured with Data Protection should have its access limited to the app itself, similar to the way you would protect configuration files. For example, if you choose to store your key ring on disk, use file system permissions. Ensure only the identity under which your web app runs has read, write, and create access to that directory. If you use Azure Blob Storage, only the web app should have the ability to read, write, or create new entries in the blob store, etc.
>
> The extension method AddDataProtection returns an IDataProtectionBuilder. `IDataProtectionBuilder` exposes extension methods that you can chain together to configure Data Protection options.

> **ⓘ Note**
>
> This article was written for an app that runs within a docker container. In a docker container the app always has the same path and, therefore, the same application discriminator. Apps that need to run in multiple environments (for example local and deployed), must set the default application discriminator for the environment. Running an app in multiple environments is beyond the scope of this article.

The following NuGet packages are required for the Data Protection extensions used in this article:

- Azure.Extensions.AspNetCore.DataProtection.Blobs
- Azure.Extensions.AspNetCore.DataProtection.Keys

ProtectKeysWithAzureKeyVault

Sign in to Azure using the CLI, for example:

Azure CLI

```
az login
```

To manage keys with Azure Key Vault, configure the system with ProtectKeysWithAzureKeyVault in `Program.cs`. `blobUriWithSasToken` is the full URI where the key file should be stored. The URI must contain the SAS token as a query string parameter:

C#

```
builder.Services.AddDataProtection()
    .PersistKeysToAzureBlobStorage(new Uri("<blobUriWithSasToken>"))
    .ProtectKeysWithAzureKeyVault(new Uri("<keyIdentifier>"), new
DefaultAzureCredential());
```

For an app to communicate and authorize itself with KeyVault, the Azure.Identity package must be added.

Set the key ring storage location (for example, PersistKeysToAzureBlobStorage). The location must be set because calling `ProtectKeysWithAzureKeyVault` implements an IXmlEncryptor that disables automatic data protection settings, including the key ring storage location. The preceding example uses Azure Blob Storage to persist the key ring. For more information, see Key storage providers: Azure Storage. You can also persist the key ring locally with PersistKeysToFileSystem.

The `keyIdentifier` is the key vault key identifier used for key encryption. For example, a key created in key vault named `dataprotection` in the `contosokeyvault` has the key identifier `https://contosokeyvault.vault.azure.net/keys/dataprotection/`. Provide the app with **Get**, **Unwrap Key** and **Wrap Key** permissions to the key vault.

`ProtectKeysWithAzureKeyVault` overloads:

- ProtectKeysWithAzureKeyVault(IDataProtectionBuilder, Uri, TokenCredential) permits the use of a keyIdentifier Uri and a tokenCredential to enable the data protection system to use the key vault.

- ProtectKeysWithAzureKeyVault(IDataProtectionBuilder, String, IKeyEncryptionKeyResolver) permits the use of a keyIdentifier string and IKeyEncryptionKeyResolver to enable the data protection system to use the key vault.

If the app uses the older Azure packages (Microsoft.AspNetCore.DataProtection.AzureStorage and Microsoft.AspNetCore.DataProtection.AzureKeyVault), we recommend ***removing*** these references and upgrading to the Azure.Extensions.AspNetCore.DataProtection.Blobs and Azure.Extensions.AspNetCore.DataProtection.Keys. These packages are where new updates are provided, and address some key security and stability issues with the older packages.

C#

```
builder.Services.AddDataProtection()
    // This blob must already exist before the application is run
    .PersistKeysToAzureBlobStorage("<storageAccountConnectionString", "
<containerName>", "<blobName>")
    // Removing this line below for an initial run will ensure the file is
created correctly
    .ProtectKeysWithAzureKeyVault(new Uri("<keyIdentifier>"), new
DefaultAzureCredential());
```

PersistKeysToFileSystem

To store keys on a UNC share instead of at the *%LOCALAPPDATA%* default location, configure the system with PersistKeysToFileSystem:

C#

```
builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new
DirectoryInfo(@"\\server\share\directory\"));
```

> ⚠ **Warning**
>
> If you change the key persistence location, the system no longer automatically encrypts keys at rest, since it doesn't know whether DPAPI is an appropriate encryption mechanism.

PersistKeysToDbContext

To store keys in a database using EntityFramework, configure the system with the Microsoft.AspNetCore.DataProtection.EntityFrameworkCore package:

```
C#

builder.Services.AddDataProtection()
    .PersistKeysToDbContext<SampleDbContext>();
```

The preceding code stores the keys in the configured database. The database context being used must implement `IDataProtectionKeyContext`. `IDataProtectionKeyContext` exposes the property `DataProtectionKeys`

```
C#

public DbSet<DataProtectionKey> DataProtectionKeys { get; set; } = null!;
```

This property represents the table in which the keys are stored. Create the table manually or with `DbContext` Migrations. For more information, see DataProtectionKey.

ProtectKeysWith*

You can configure the system to protect keys at rest by calling any of the ProtectKeysWith* configuration APIs. Consider the example below, which stores keys on a UNC share and encrypts those keys at rest with a specific X.509 certificate:

```
C#

builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new
DirectoryInfo(@"\\server\share\directory\"))

.ProtectKeysWithCertificate(builder.Configuration["CertificateThumbprint"]);
```

You can provide an X509Certificate2 to ProtectKeysWithCertificate, such as a certificate loaded from a file:

```
C#

builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new
DirectoryInfo(@"\\server\share\directory\"))
    .ProtectKeysWithCertificate(
        new X509Certificate2("certificate.pfx",
builder.Configuration["CertificatePassword"]));
```

See Key Encryption At Rest for more examples and discussion on the built-in key encryption mechanisms.

UnprotectKeysWithAnyCertificate

You can rotate certificates and decrypt keys at rest using an array of X509Certificate2 certificates with UnprotectKeysWithAnyCertificate:

C#

```
builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new
DirectoryInfo(@"\\server\share\directory\"))
    .ProtectKeysWithCertificate(
        new X509Certificate2("certificate.pfx",
builder.Configuration["CertificatePassword"]))
    .UnprotectKeysWithAnyCertificate(
        new X509Certificate2("certificate_1.pfx",
builder.Configuration["CertificatePassword_1"]),
        new X509Certificate2("certificate_2.pfx",
builder.Configuration["CertificatePassword_2"]));
```

SetDefaultKeyLifetime

To configure the system to use a key lifetime of 14 days instead of the default 90 days, use SetDefaultKeyLifetime:

C#

```
builder.Services.AddDataProtection()
    .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
```

SetApplicationName

By default, the Data Protection system isolates apps from one another based on their content root paths, even if they share the same physical key repository. This isolation prevents the apps from understanding each other's protected payloads.

To share protected payloads among apps:

- Configure SetApplicationName in each app with the same value.
- Use the same version of the Data Protection API stack across the apps. Perform **either** of the following in the apps' project files:

- Reference the same shared framework version via the Microsoft.AspNetCore.App metapackage.
- Reference the same Data Protection package version.

```
C#

builder.Services.AddDataProtection()
    .SetApplicationName("<sharedApplicationName>");
```

SetApplicationName internally sets DataProtectionOptions.ApplicationDiscriminator. For troubleshooting purposes, the value assigned to the discriminator by the framework can be logged with the following code placed after the WebApplication is built in `Program.cs`:

```
C#

var discriminator =
app.Services.GetRequiredService<IOptions<DataProtectionOptions>>()
    .Value.ApplicationDiscriminator;
app.Logger.LogInformation("ApplicationDiscriminator:
{ApplicationDiscriminator}", discriminator);
```

For more information on how the discriminator is used, see the following sections later in this article:

- Per-application isolation
- Data Protection and app isolation

> ⚠ **Warning**
>
> In .NET 6, WebApplicationBuilder normalizes the content root path to end with a DirectorySeparatorChar. For example, on Windows the content root path ends in `\` and on Linux `/`. Other hosts don't normalize the path. Most apps migrating from HostBuilder or WebHostBuilder won't share the same app name because they won't have the terminating `DirectorySeparatorChar`. To work around this issue, remove the directory separator character and set the app name manually, as shown in the following code:
>
> ```
> C#
>
> using Microsoft.AspNetCore.DataProtection;
> using System.Reflection;
>
> var builder = WebApplication.CreateBuilder(args);
> var trimmedContentRootPath =
> ```

```
builder.Environment.ContentRootPath.TrimEnd(Path.DirectorySeparatorChar
);
builder.Services.AddDataProtection()
 .SetApplicationName(trimmedContentRootPath);
var app = builder.Build();

app.MapGet("/", () => Assembly.GetEntryAssembly()!.GetName().Name);

app.Run();
```

DisableAutomaticKeyGeneration

You may have a scenario where you don't want an app to automatically roll keys (create new keys) as they approach expiration. One example of this scenario might be apps set up in a primary/secondary relationship, where only the primary app is responsible for key management concerns and secondary apps simply have a read-only view of the key ring. The secondary apps can be configured to treat the key ring as read-only by configuring the system with DisableAutomaticKeyGeneration:

C#

```
builder.Services.AddDataProtection()
    .DisableAutomaticKeyGeneration();
```

Per-application isolation

When the Data Protection system is provided by an ASP.NET Core host, it automatically isolates apps from one another, even if those apps are running under the same worker process account and are using the same master keying material. This is similar to the IsolateApps modifier from System.Web's `<machineKey>` element.

The isolation mechanism works by considering each app on the local machine as a unique tenant, thus the IDataProtector rooted for any given app automatically includes the app ID as a discriminator (ApplicationDiscriminator). The app's unique ID is the app's physical path:

- For apps hosted in IIS, the unique ID is the IIS physical path of the app. If an app is deployed in a web farm environment, this value is stable assuming that the IIS environments are configured similarly across all machines in the web farm.
- For self-hosted apps running on the Kestrel server, the unique ID is the physical path to the app on disk.

The unique identifier is designed to survive resets—both of the individual app and of the machine itself.

This isolation mechanism assumes that the apps aren't malicious. A malicious app can always impact any other app running under the same worker process account. In a shared hosting environment where apps are mutually untrusted, the hosting provider should take steps to ensure OS-level isolation between apps, including separating the apps' underlying key repositories.

If the Data Protection system isn't provided by an ASP.NET Core host (for example, if you instantiate it via the `DataProtectionProvider` concrete type) app isolation is disabled by default. When app isolation is disabled, all apps backed by the same keying material can share payloads as long as they provide the appropriate purposes. To provide app isolation in this environment, call the SetApplicationName method on the configuration object and provide a unique name for each app.

Data Protection and app isolation

Consider the following points for app isolation:

- When multiple apps are pointed at the same key repository, the intention is that the apps share the same master key material. Data Protection is developed with the assumption that all apps sharing a key ring can access all items in that key ring. The application unique identifier is used to isolate application specific keys derived from the key ring provided keys. It doesn't expect item level permissions, such as those provided by Azure KeyVault to be used to enforce extra isolation. Attempting item level permissions generates application errors. If you don't want to rely on the built-in application isolation, separate key store locations should be used and not shared between applications.

- The application discriminator (ApplicationDiscriminator) is used to allow different apps to share the same master key material but to keep their cryptographic payloads distinct from one another. For the apps to be able to read each other's cryptographic payloads, they must have the same application discriminator, which can be set by calling SetApplicationName.

- If an app is compromised (for example, by an RCE attack), all master key material accessible to that app must also be considered compromised, regardless of its protection-at-rest state. This implies that if two apps are pointed at the same repository, even if they use different app discriminators, a compromise of one is functionally equivalent to a compromise of both.

This "functionally equivalent to a compromise of both" clause holds even if the two apps use different mechanisms for key protection at rest. Typically, this isn't an expected configuration. The protection-at-rest mechanism is intended to provide protection in the event a cyberattacker gains read access to the repository. A cyberattacker who gains write access to the repository (perhaps because they attained code execution permission within an app) can insert malicious keys into storage. The Data Protection system intentionally doesn't provide protection against a cyberattacker who gains write access to the key repository.

- If apps need to remain truly isolated from one another, they should use different key repositories. This naturally falls out of the definition of "isolated". Apps are ***not*** isolated if they all have Read and Write access to each other's data stores.

Changing algorithms with UseCryptographicAlgorithms

The Data Protection stack allows you to change the default algorithm used by newly generated keys. The simplest way to do this is to call UseCryptographicAlgorithms from the configuration callback:

C#

```
builder.Services.AddDataProtection()
    .UseCryptographicAlgorithms(new AuthenticatedEncryptorConfiguration
    {
        EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
        ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
    });
```

The default EncryptionAlgorithm is AES-256-CBC, and the default ValidationAlgorithm is HMACSHA256. The default policy can be set by a system administrator via a machine-wide policy, but an explicit call to `UseCryptographicAlgorithms` overrides the default policy.

Calling `UseCryptographicAlgorithms` allows you to specify the desired algorithm from a predefined built-in list. You don't need to worry about the implementation of the algorithm. In the scenario above, the Data Protection system attempts to use the CNG implementation of AES if running on Windows. Otherwise, it falls back to the managed System.Security.Cryptography.Aes class.

You can manually specify an implementation via a call to UseCustomCryptographicAlgorithms.

> **Tip**
>
> Changing algorithms doesn't affect existing keys in the key ring. It only affects newly-generated keys.

Specifying custom managed algorithms

To specify custom managed algorithms, create a ManagedAuthenticatedEncryptorConfiguration instance that points to the implementation types:

C#

```
builder.Services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(new
ManagedAuthenticatedEncryptorConfiguration
    {
        // A type that subclasses SymmetricAlgorithm
        EncryptionAlgorithmType = typeof(Aes),

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // A type that subclasses KeyedHashAlgorithm
        ValidationAlgorithmType = typeof(HMACSHA256)
    });
```

Generally the *Type properties must point to concrete, instantiable (via a public parameterless ctor) implementations of SymmetricAlgorithm and KeyedHashAlgorithm, though the system special-cases some values like `typeof(Aes)` for convenience.

> **Note**
>
> The SymmetricAlgorithm must have a key length of ≥ 128 bits and a block size of ≥ 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The KeyedHashAlgorithm must have a digest size of >= 128 bits, and it must support keys of length equal to the hash algorithm's digest length. The KeyedHashAlgorithm isn't strictly required to be HMAC.

Specifying custom Windows CNG algorithms

To specify a custom Windows CNG algorithm using CBC-mode encryption with HMAC validation, create a CngCbcAuthenticatedEncryptorConfiguration instance that contains the algorithmic information:

C#

```
builder.Services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(new
CngCbcAuthenticatedEncryptorConfiguration
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // Passed to BCryptOpenAlgorithmProvider
        HashAlgorithm = "SHA256",
        HashAlgorithmProvider = null
    });
```

> **Note**
>
> The symmetric block cipher algorithm must have a key length of >= 128 bits, a block size of >= 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The hash algorithm must have a digest size of >= 128 bits and must support being opened with the BCRYPT_ALG_HANDLE_HMAC_FLAG flag. The *Provider properties can be set to null to use the default provider for the specified algorithm. For more information, see the BCryptOpenAlgorithmProvider documentation.

To specify a custom Windows CNG algorithm using Galois/Counter Mode encryption with validation, create a CngGcmAuthenticatedEncryptorConfiguration instance that contains the algorithmic information:

C#

```
builder.Services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(new
CngGcmAuthenticatedEncryptorConfiguration
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
```

```
        EncryptionAlgorithmKeySize = 256
    });
```

> **Note**
>
> The symmetric block cipher algorithm must have a key length of >= 128 bits, a block size of exactly 128 bits, and it must support GCM encryption. You can set the EncryptionAlgorithmProvider property to null to use the default provider for the specified algorithm. For more information, see the BCryptOpenAlgorithmProvider documentation.

Specifying other custom algorithms

Though not exposed as a first-class API, the Data Protection system is extensible enough to allow specifying almost any kind of algorithm. For example, it's possible to keep all keys contained within a Hardware Security Module (HSM) and to provide a custom implementation of the core encryption and decryption routines. For more information, see IAuthenticatedEncryptor in Core cryptography extensibility.

Persisting keys when hosting in a Docker container

When hosting in a Docker container, keys should be maintained in either:

- A folder that's a Docker volume that persists beyond the container's lifetime, such as a shared volume or a host-mounted volume.
- An external provider, such as Azure Blob Storage (shown in the ProtectKeysWithAzureKeyVault section) or Redis.

Persisting keys with Redis

Only Redis versions supporting Redis Data Persistence should be used to store keys. Azure Blob storage is persistent and can be used to store keys. For more information, see this GitHub issue.

Logging DataProtection

Enable `Information` level logging of DataProtection to help diagnosis problem. The following `appsettings.json` file enables information logging of the DataProtection API:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.DataProtection": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

For more information on logging, see Logging in .NET Core and ASP.NET Core.

Additional resources

- Non-DI aware scenarios for Data Protection in ASP.NET Core
- Data Protection machine-wide policy support in ASP.NET Core
- Host ASP.NET Core in a web farm
- Key storage providers in ASP.NET Core

Data Protection key management and lifetime in ASP.NET Core

Article • 08/21/2024

By Rick Anderson

Key management

The app attempts to detect its operational environment and handle key configuration on its own.

1. If the app is hosted in Azure Apps, keys are persisted to the *%HOME%\ASP.NET\DataProtection-Keys* folder. This folder is backed by network storage and is synchronized across all machines hosting the app.

   - Keys aren't protected at rest.
   - The *DataProtection-Keys* folder supplies the key ring to all instances of an app in a single deployment slot.
   - Separate deployment slots, such as Staging and Production, don't share a key ring. When you swap between deployment slots, for example swapping Staging to Production or using A/B testing, any app using Data Protection won't be able to decrypt stored data using the key ring inside the previous slot. This leads to users being logged out of an app that uses the standard ASP.NET Core cookie authentication, as it uses Data Protection to protect its cookies. If you desire slot-independent key rings, use an external key ring provider, such as Azure Blob Storage, Azure Key Vault, a SQL store, or Redis cache.

2. If the user profile is available, keys are persisted to the *%LOCALAPPDATA%\ASP.NET\DataProtection-Keys* folder. If the operating system is Windows, the keys are encrypted at rest using DPAPI.

   The app pool's setProfileEnvironment attribute must also be enabled. The default value of `setProfileEnvironment` is `true`. In some scenarios (for example, Windows OS), `setProfileEnvironment` is set to `false`. If keys aren't stored in the user profile directory as expected:
   a. Navigate to the *%windir%/system32/inetsrv/config* folder.
   b. Open the *applicationHost.config* file.
   c. Locate the `<system.applicationHost><applicationPools><applicationPoolDefaults><processModel>` element.

   d. Confirm that the `setProfileEnvironment` attribute isn't present, which defaults the value to `true`, or explicitly set the attribute's value to `true`.

3. If the app is hosted in IIS, keys are persisted to the HKLM registry in a special registry key that's ACLed only to the worker process account. Keys are encrypted at rest using DPAPI.

4. If none of these conditions match, keys aren't persisted outside of the current process. When the process shuts down, all generated keys are lost.

The developer is always in full control and can override how and where keys are stored. The first three options above should provide good defaults for most apps similar to how the ASP.NET **<machineKey>** auto-generation routines worked in the past. The final, fallback option is the only scenario that requires the developer to specify configuration upfront if they want key persistence, but this fallback only occurs in rare situations.

When hosting in a Docker container, keys should be persisted in a folder that's a Docker volume (a shared volume or a host-mounted volume that persists beyond the container's lifetime) or in an external provider, such as Azure Key Vault or Redis. An external provider is also useful in web farm scenarios if apps can't access a shared network volume (see PersistKeysToFileSystem for more information).

> ⚠ **Warning**
>
> If the developer overrides the rules outlined above and points the Data Protection system at a specific key repository, automatic encryption of keys at rest is disabled. At-rest protection can be re-enabled via **configuration**.

Key lifetime

Keys have a 90-day lifetime by default. When a key expires, the app automatically generates a new key and sets the new key as the active key. As long as retired keys remain on the system, your app can decrypt any data protected with them. See key management for more information.

Default algorithms

The default payload protection algorithm used is AES-256-CBC for confidentiality and HMACSHA256 for authenticity. A 512-bit master key, changed every 90 days, is used to derive the two sub-keys used for these algorithms on a per-payload basis. See subkey derivation for more information.

Delete keys

Deleting a key makes its protected data permanently inaccessible. To mitigate that risk, we recommend not deleting keys. The resulting accumulation of keys generally has minimal impact because they're small. In exceptional cases, such as extremely long-running services, keys can be deleted. Only delete keys:

- That are old (no longer in use).
- When you can accept the risk of data loss in exchange for storage savings.

We recommend not deleting data protection keys.

C#

```
using Microsoft.AspNetCore.DataProtection.KeyManagement;

var services = new ServiceCollection();
services.AddDataProtection();

var serviceProvider = services.BuildServiceProvider();

var keyManager = serviceProvider.GetService<IKeyManager>();

if (keyManager is IDeletableKeyManager deletableKeyManager)
{
    var utcNow = DateTimeOffset.UtcNow;
    var yearAgo = utcNow.AddYears(-1);

    if (!deletableKeyManager.DeleteKeys(key => key.ExpirationDate <
yearAgo))
    {
        Console.WriteLine("Failed to delete keys.");
    }
    else
    {
        Console.WriteLine("Old keys deleted successfully.");
    }
}
else
{
    Console.WriteLine("Key manager does not support deletion.");
}
```

Additional resources

- Key management extensibility in ASP.NET Core
- Host ASP.NET Core in a web farm

Data Protection machine-wide policy support in ASP.NET Core

Article • 06/17/2024

By Rick Anderson

When running on Windows, the Data Protection system has limited support for setting a default machine-wide policy for all apps that consume ASP.NET Core Data Protection. The general idea is that an administrator might wish to change a default setting, such as the algorithms used or key lifetime, without the need to manually update every app on the machine.

> ⚠ **Warning**
>
> The system administrator can set default policy, but they can't enforce it. The app developer can always override any value with one of their own choosing. The default policy only affects apps where the developer hasn't specified an explicit value for a setting.

Setting default policy

To set default policy, an administrator can set known values in the system registry under the following registry key:

HKLM\SOFTWARE\Microsoft\DotNetPackages\Microsoft.AspNetCore.DataProtection

If you're on a 64-bit operating system and want to affect the behavior of 32-bit apps, remember to configure the Wow6432Node equivalent of the above key.

The supported values are shown below.

Expand table

Value	Type	Description
EncryptionType	string	Specifies which algorithms should be used for data protection. The value must be CNG-CBC, CNG-GCM, or Managed and is described in more detail below.
DefaultKeyLifetime	DWORD	Specifies the lifetime for newly-generated keys. The value is specified in days and must be >= 7.

Value	Type	Description
KeyEscrowSinks	string	Specifies the types that are used for key escrow. The value is a semicolon-delimited list of key escrow sinks, where each element in the list is the assembly-qualified name of a type that implements IKeyEscrowSink.

Encryption types

If EncryptionType is CNG-CBC, the system is configured to use a CBC-mode symmetric block cipher for confidentiality and HMAC for authenticity with services provided by Windows CNG (see Specifying custom Windows CNG algorithms for more details). The following additional values are supported, each of which corresponds to a property on the CngCbcAuthenticatedEncryptionSettings type.

Expand table

Value	Type	Description
EncryptionAlgorithm	string	The name of a symmetric block cipher algorithm understood by CNG. This algorithm is opened in CBC mode.
EncryptionAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm EncryptionAlgorithm.
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric block cipher algorithm.
HashAlgorithm	string	The name of a hash algorithm understood by CNG. This algorithm is opened in HMAC mode.
HashAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm HashAlgorithm.

If EncryptionType is CNG-GCM, the system is configured to use a Galois/Counter Mode symmetric block cipher for confidentiality and authenticity with services provided by Windows CNG (see Specifying custom Windows CNG algorithms for more details). The following additional values are supported, each of which corresponds to a property on the CngGcmAuthenticatedEncryptionSettings type.

Expand table

Value	Type	Description
EncryptionAlgorithm	string	The name of a symmetric block cipher algorithm understood by CNG. This algorithm is opened in Galois/Counter Mode.
EncryptionAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm EncryptionAlgorithm.
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric block cipher algorithm.

If EncryptionType is Managed, the system is configured to use a managed SymmetricAlgorithm for confidentiality and KeyedHashAlgorithm for authenticity (see Specifying custom managed algorithms for more details). The following additional values are supported, each of which corresponds to a property on the ManagedAuthenticatedEncryptionSettings type.

Expand table

Value	Type	Description
EncryptionAlgorithmType	string	The assembly-qualified name of a type that implements SymmetricAlgorithm.
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric encryption algorithm.
ValidationAlgorithmType	string	The assembly-qualified name of a type that implements KeyedHashAlgorithm.

If EncryptionType has any other value other than null or empty, the Data Protection system throws an exception at startup.

> ⚠ **Warning**
>
> When configuring a default policy setting that involves type names (EncryptionAlgorithmType, ValidationAlgorithmType, KeyEscrowSinks), the types must be available to the app. This means that for apps running on Desktop CLR, the assemblies that contain these types should be present in the Global Assembly Cache (GAC). For ASP.NET Core apps running on .NET Core, the packages that contain these types should be installed.

Non-DI aware scenarios for Data Protection in ASP.NET Core

Article • 06/17/2024

By Rick Anderson

The ASP.NET Core Data Protection system is normally added to a service container and consumed by dependent components via dependency injection (DI). However, there are cases where this isn't feasible or desired, especially when importing the system into an existing app.

To support these scenarios, the Microsoft.AspNetCore.DataProtection.Extensions package provides a concrete type, DataProtectionProvider, which offers a simple way to use Data Protection without relying on DI. The `DataProtectionProvider` type implements IDataProtectionProvider. Constructing `DataProtectionProvider` only requires providing a DirectoryInfo instance to indicate where the provider's cryptographic keys should be stored, as seen in the following code sample:

C#

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder));

        var protector = dataProtectionProvider.CreateProtector("Program.No-
DI");
        Console.Write("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
```

```
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8FWbAn6...ch3hAPm1NJA
 * Unprotect returned: Hello world!
 *
 * Press any key...
*/
```

By default, the `DataProtectionProvider` concrete type doesn't encrypt raw key material before persisting it to the file system. This is to support scenarios where the developer points to a network share and the Data Protection system can't automatically deduce an appropriate at-rest key encryption mechanism.

Additionally, the `DataProtectionProvider` concrete type doesn't isolate apps by default. All apps using the same key directory can share payloads as long as their purpose parameters match.

The DataProtectionProvider constructor accepts an optional configuration callback that can be used to adjust the behaviors of the system. The sample below demonstrates restoring isolation with an explicit call to SetApplicationName. The sample also demonstrates configuring the system to automatically encrypt persisted keys using Windows DPAPI. If the directory points to a UNC share, you may wish to distribute a shared certificate across all relevant machines and to configure the system to use certificate-based encryption with a call to ProtectKeysWithCertificate.

C#

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
```

```
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder),
            configuration =>
            {
                configuration.SetApplicationName("my app name");
                configuration.ProtectKeysWithDpapi();
            });

        var protector = dataProtectionProvider.CreateProtector("Program.No-
DI");
        Console.Write("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}
```

Tip

Instances of the `DataProtectionProvider` concrete type are expensive to create. If an app maintains multiple instances of this type and if they're all using the same key storage directory, app performance might degrade. If you use the `DataProtectionProvider` type, we recommend that you create this type once and reuse it as much as possible. The `DataProtectionProvider` type and all IDataProtector instances created from it are thread-safe for multiple callers.

ASP.NET Core Data Protection extensibility APIs

Article • 06/03/2022

- Core cryptography extensibility
- Key management extensibility
- Miscellaneous APIs

Core cryptography extensibility in ASP.NET Core

Article • 09/27/2024

> ⚠ **Warning**
>
> Types that implement any of the following interfaces should be thread-safe for multiple callers.

IAuthenticatedEncryptor

The **IAuthenticatedEncryptor** interface is the basic building block of the cryptographic subsystem. There's generally one IAuthenticatedEncryptor per key, and the IAuthenticatedEncryptor instance wraps all cryptographic key material and algorithmic information necessary to perform cryptographic operations.

As its name suggests, the type is responsible for providing authenticated encryption and decryption services. It exposes the following two APIs.

- `Decrypt(ArraySegment<byte> ciphertext, ArraySegment<byte> additionalAuthenticatedData) : byte[]`
- `Encrypt(ArraySegment<byte> plaintext, ArraySegment<byte> additionalAuthenticatedData) : byte[]`

The Encrypt method returns a blob that includes the enciphered plaintext and an authentication tag. The authentication tag must encompass the additional authenticated data (AAD), though the AAD itself need not be recoverable from the final payload. The Decrypt method validates the authentication tag and returns the deciphered payload. All failures (except ArgumentNullException and similar) should be homogenized to CryptographicException.

> ⓘ **Note**
>
> The IAuthenticatedEncryptor instance itself doesn't actually need to contain the key material. For example, the implementation could delegate to an HSM for all operations.

How to create an IAuthenticatedEncryptor

ASP.NET Core 2.x

The **IAuthenticatedEncryptorFactory** interface represents a type that knows how to create an IAuthenticatedEncryptor instance. Its API is as follows.

- CreateEncryptorInstance(IKey key) : IAuthenticatedEncryptor

For any given IKey instance, any authenticated encryptors created by its CreateEncryptorInstance method should be considered equivalent, as in the below code sample.

C#

```
// we have an IAuthenticatedEncryptorFactory instance and an IKey
instance
IAuthenticatedEncryptorFactory factory = ...;
IKey key = ...;

// get an encryptor instance and perform an authenticated encryption
operation
ArraySegment<byte> plaintext = new ArraySegment<byte>
(Encoding.UTF8.GetBytes("plaintext"));
ArraySegment<byte> aad = new ArraySegment<byte>
(Encoding.UTF8.GetBytes("AAD"));
var encryptor1 = factory.CreateEncryptorInstance(key);
byte[] ciphertext = encryptor1.Encrypt(plaintext, aad);

// get another encryptor instance and perform an authenticated
decryption operation
var encryptor2 = factory.CreateEncryptorInstance(key);
byte[] roundTripped = encryptor2.Decrypt(new ArraySegment<byte>
(ciphertext), aad);


// the 'roundTripped' and 'plaintext' buffers should be equivalent
```

IAuthenticatedEncryptorDescriptor (ASP.NET Core 2.x only)

ASP.NET Core 2.x

The **IAuthenticatedEncryptorDescriptor** interface represents a type that knows how to export itself to XML. Its API is as follows.

- ExportToXml() : XmlSerializedDescriptorInfo

XML Serialization

The primary difference between IAuthenticatedEncryptor and IAuthenticatedEncryptorDescriptor is that the descriptor knows how to create the encryptor and supply it with valid arguments. Consider an IAuthenticatedEncryptor whose implementation relies on SymmetricAlgorithm and KeyedHashAlgorithm. The encryptor's job is to consume these types, but it doesn't necessarily know where these types came from, so it can't really write out a proper description of how to recreate itself if the application restarts. The descriptor acts as a higher level on top of this. Since the descriptor knows how to create the encryptor instance (for example it knows how to create the required algorithms), it can serialize that knowledge in XML form so that the encryptor instance can be recreated after an application reset.

The descriptor can be serialized via its ExportToXml routine. This routine returns an XmlSerializedDescriptorInfo which contains two properties: the XElement representation of the descriptor and the Type which represents an IAuthenticatedEncryptorDescriptorDeserializer which can be used to resurrect this descriptor given the corresponding XElement.

The serialized descriptor may contain sensitive information such as cryptographic key material. The data protection system has built-in support for encrypting information before it's persisted to storage. To take advantage of this, the descriptor should mark the element which contains sensitive information with the attribute name "requiresEncryption" (xmlns `"<http://schemas.asp.net/2015/03/dataProtection>"`), value "true".

> **Tip**
>
> There's a helper API for setting this attribute. Call the extension method XElement.MarkAsRequiresEncryption() located in namespace Microsoft.AspNetCore.DataProtection.AuthenticatedEncryption.ConfigurationModel .

There can also be cases where the serialized descriptor doesn't contain sensitive information. Consider again the case of a cryptographic key stored in an HSM. The descriptor cannot write out the key material when serializing itself since the HSM won't expose the material in plaintext form. Instead, the descriptor might write out the key-

wrapped version of the key (if the HSM allows export in this fashion) or the HSM's own unique identifier for the key.

IAuthenticatedEncryptorDescriptorDeserializer

The **IAuthenticatedEncryptorDescriptorDeserializer** interface represents a type that knows how to deserialize an IAuthenticatedEncryptorDescriptor instance from an XElement. It exposes a single method:

- ImportFromXml(XElement element) : IAuthenticatedEncryptorDescriptor

The ImportFromXml method takes the XElement that was returned by IAuthenticatedEncryptorDescriptor.ExportToXml and creates an equivalent of the original IAuthenticatedEncryptorDescriptor.

Types which implement IAuthenticatedEncryptorDescriptorDeserializer should have one of the following two public constructors:

- .ctor(IServiceProvider)
- .ctor()

> ⓘ **Note**
>
> The IServiceProvider passed to the constructor may be null.

The top-level factory

ASP.NET Core 2.x

The **AlgorithmConfiguration** class represents a type which knows how to create IAuthenticatedEncryptorDescriptor instances. It exposes a single API.

- CreateNewDescriptor() : IAuthenticatedEncryptorDescriptor

Think of AlgorithmConfiguration as the top-level factory. The configuration serves as a template. It wraps algorithmic information (for example this configuration produces descriptors with an AES-128-GCM master key), but it's not yet associated with a specific key.

When CreateNewDescriptor is called, fresh key material is created solely for this call, and a new IAuthenticatedEncryptorDescriptor is produced which wraps this key

material and the algorithmic information required to consume the material. The key material could be created in software (and held in memory), it could be created and held within an HSM, and so on. The crucial point is that any two calls to CreateNewDescriptor should never create equivalent IAuthenticatedEncryptorDescriptor instances.

The AlgorithmConfiguration type serves as the entry point for key creation routines such as automatic key rolling. To change the implementation for all future keys, set the AuthenticatedEncryptorConfiguration property in KeyManagementOptions.

Key management extensibility in ASP.NET Core

Article • 02/06/2024

Read the key management section before reading this section, as it explains some of the fundamental concepts behind these APIs.

Warning: Types that implement any of the following interfaces should be thread-safe for multiple callers.

Key

The `IKey` interface is the basic representation of a key in cryptosystem. The term key is used here in the abstract sense, not in the literal sense of "cryptographic key material". A key has the following properties:

- Activation, creation, and expiration dates
- Revocation status
- Key identifier (a GUID)

Additionally, `IKey` exposes a `CreateEncryptor` method which can be used to create an IAuthenticatedEncryptor instance tied to this key.

> **Note**
>
> There's no API to retrieve the raw cryptographic material from an `IKey` instance.

IKeyManager

The `IKeyManager` interface represents an object responsible for general key storage, retrieval, and manipulation. It exposes three high-level operations:

- Create a new key and persist it to storage.
- Get all keys from storage.
- Revoke one or more keys and persist the revocation information to storage.

> ⚠ **Warning**
>
> Writing an `IKeyManager` is a very advanced task, and the majority of developers shouldn't attempt it. Instead, most developers should take advantage of the facilities offered by the XmlKeyManager class.

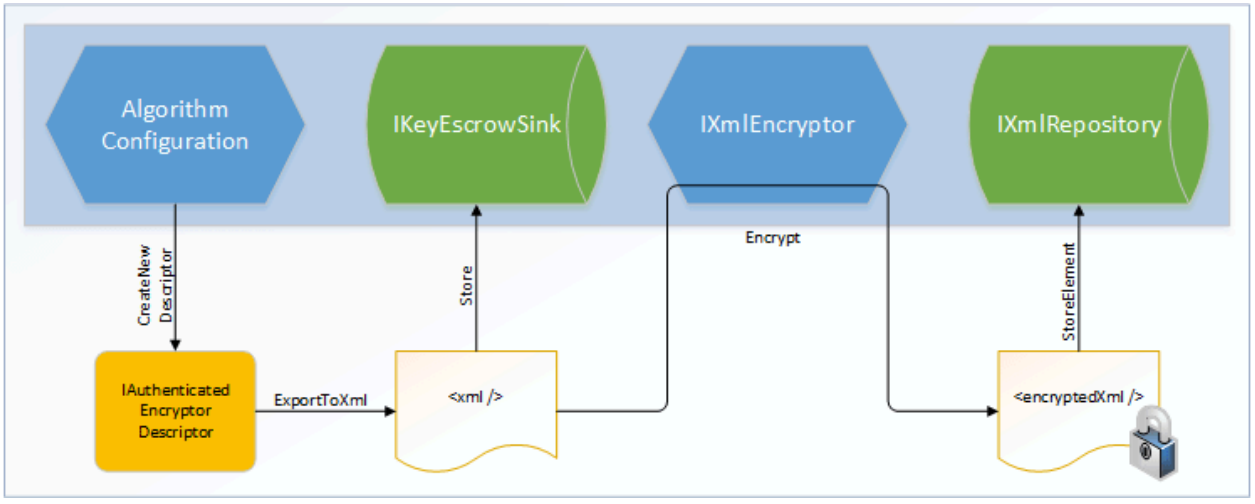
XmlKeyManager

The `XmlKeyManager` type is the in-box concrete implementation of `IKeyManager`. It provides several useful facilities, including key escrow and encryption of keys at rest. Keys in this system are represented as XML elements (specifically, XElement).

`XmlKeyManager` depends on several other components in the course of fulfilling its tasks:

- `AlgorithmConfiguration`, which dictates the algorithms used by new keys.
- `IXmlRepository`, which controls where keys are persisted in storage.
- `IXmlEncryptor` [optional], which allows encrypting keys at rest.
- `IKeyEscrowSink` [optional], which provides key escrow services.

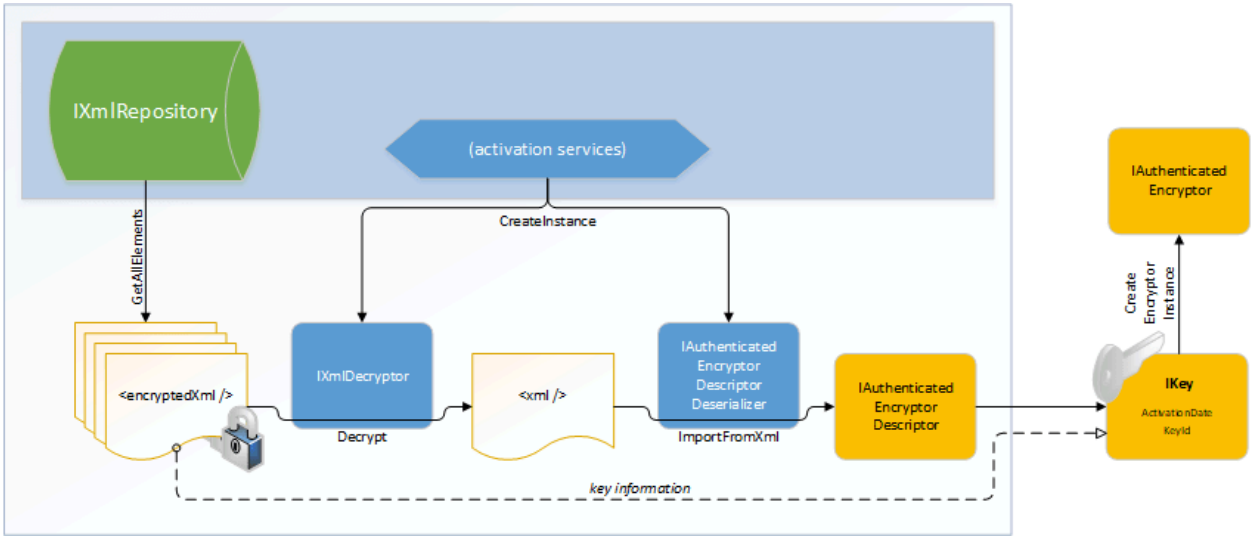
Below are high-level diagrams which indicate how these components are wired together within `XmlKeyManager`.



Key Creation / CreateNewKey

In the implementation of `CreateNewKey`, the `AlgorithmConfiguration` component is used to create a unique `IAuthenticatedEncryptorDescriptor`, which is then serialized as XML. If a key escrow sink is present, the raw (unencrypted) XML is provided to the sink for long-term storage. The unencrypted XML is then run through an `IXmlEncryptor` (if

required) to generate the encrypted XML document. This encrypted document is persisted to long-term storage via the `IXmlRepository`. (If no `IXmlEncryptor` is configured, the unencrypted document is persisted in the `IXmlRepository`.)



Key Retrieval / GetAllKeys

In the implementation of `GetAllKeys`, the XML documents representing keys and revocations are read from the underlying `IXmlRepository`. If these documents are encrypted, the system will automatically decrypt them. `XmlKeyManager` creates the appropriate `IAuthenticatedEncryptorDescriptorDeserializer` instances to deserialize the documents back into `IAuthenticatedEncryptorDescriptor` instances, which are then wrapped in individual `IKey` instances. This collection of `IKey` instances is returned to the caller.

Further information on the particular XML elements can be found in the key storage format document.

IXmlRepository

The `IXmlRepository` interface represents a type that can persist XML to and retrieve XML from a backing store. It exposes two APIs:

- `GetAllElements` : `IReadOnlyCollection<XElement>`
- `StoreElement(XElement element, string friendlyName)`

Implementations of `IXmlRepository` don't need to parse the XML passing through them. They should treat the XML documents as opaque and let higher layers worry about generating and parsing the documents.

There are four built-in concrete types which implement `IXmlRepository`:

- FileSystemXmlRepository
- RegistryXmlRepository
- AzureStorage.AzureBlobXmlRepository
- RedisXmlRepository

See the key storage providers document for more information.

Registering a custom `IXmlRepository` is appropriate when using a different backing store (for example, Azure Table Storage).

To change the default repository application-wide, register a custom `IXmlRepository` instance:

```
C#

services.Configure<KeyManagementOptions>(options => options.XmlRepository =
new MyCustomXmlRepository());
```

IXmlEncryptor

The `IXmlEncryptor` interface represents a type that can encrypt a plaintext XML element. It exposes a single API:

- Encrypt(XElement plaintextElement) : EncryptedXmlInfo

If a serialized `IAuthenticatedEncryptorDescriptor` contains any elements marked as "requires encryption", then `XmlKeyManager` will run those elements through the configured `IXmlEncryptor`'s `Encrypt` method, and it will persist the enciphered element rather than the plaintext element to the `IXmlRepository`. The output of the `Encrypt` method is an `EncryptedXmlInfo` object. This object is a wrapper which contains both the resultant enciphered `XElement` and the Type which represents an `IXmlDecryptor` which can be used to decipher the corresponding element.

There are four built-in concrete types which implement `IXmlEncryptor`:

- CertificateXmlEncryptor
- DpapiNGXmlEncryptor
- DpapiXmlEncryptor
- NullXmlEncryptor

See the key encryption at rest document for more information.

To change the default key-encryption-at-rest mechanism application-wide, register a custom `IXmlEncryptor` instance:

```
C#

services.Configure<KeyManagementOptions>(options => options.XmlEncryptor =
new MyCustomXmlEncryptor());
```

IXmlDecryptor

The `IXmlDecryptor` interface represents a type that knows how to decrypt an `XElement` that was enciphered via an `IXmlEncryptor`. It exposes a single API:

- Decrypt(XElement encryptedElement) : XElement

The `Decrypt` method undoes the encryption performed by `IXmlEncryptor.Encrypt`. Generally, each concrete `IXmlEncryptor` implementation will have a corresponding concrete `IXmlDecryptor` implementation.

Types which implement `IXmlDecryptor` should have one of the following two public constructors:

- .ctor(IServiceProvider)
- .ctor()

> ⓘ **Note**
>
> The `IServiceProvider` passed to the constructor may be null.

IKeyEscrowSink

The `IKeyEscrowSink` interface represents a type that can perform escrow of sensitive information. Recall that serialized descriptors might contain sensitive information (such as cryptographic material), and this is what led to the introduction of the IXmlEncryptor type in the first place. However, accidents happen, and key rings can be deleted or become corrupted.

The escrow interface provides an emergency escape hatch, allowing access to the raw serialized XML before it's transformed by any configured IXmlEncryptor. The interface exposes a single API:

- Store(Guid keyId, XElement element)

It's up to the `IKeyEscrowSink` implementation to handle the provided element in a secure manner consistent with business policy. One possible implementation could be for the escrow sink to encrypt the XML element using a known corporate X.509 certificate where the certificate's private key has been escrowed; the `CertificateXmlEncryptor` type can assist with this. The `IKeyEscrowSink` implementation is also responsible for persisting the provided element appropriately.

By default no escrow mechanism is enabled, though server administrators can configure this globally. It can also be configured programmatically via the `IDataProtectionBuilder.AddKeyEscrowSink` method as shown in the sample below. The `AddKeyEscrowSink` method overloads mirror the `IServiceCollection.AddSingleton` and `IServiceCollection.AddInstance` overloads, as `IKeyEscrowSink` instances are intended to be singletons. If multiple `IKeyEscrowSink` instances are registered, each one will be called during key generation, so keys can be escrowed to multiple mechanisms simultaneously.

There's no API to read material from an `IKeyEscrowSink` instance. This is consistent with the design theory of the escrow mechanism: it's intended to make the key material accessible to a trusted authority, and since the application is itself not a trusted authority, it shouldn't have access to its own escrowed material.

The following sample code demonstrates creating and registering an `IKeyEscrowSink` where keys are escrowed such that only members of "CONTOSODomain Admins" can recover them.

> **Note**
>
> To run this sample, you must be on a domain-joined Windows 8 / Windows Server 2012 machine, and the domain controller must be Windows Server 2012 or later.

C#

```
using System;
using System.IO;
using System.Xml.Linq;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.AspNetCore.DataProtection.XmlEncryption;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

public class Program
```

```
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi()
            .AddKeyEscrowSink(sp => new MyKeyEscrowSink(sp));
        var services = serviceCollection.BuildServiceProvider();

        // get a reference to the key manager and force a new key to be
generated
        Console.WriteLine("Generating new key...");
        var keyManager = services.GetService<IKeyManager>();
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddDays(7));
    }

    // A key escrow sink where keys are escrowed such that they
    // can be read by members of the CONTOSO\Domain Admins group.
    private class MyKeyEscrowSink : IKeyEscrowSink
    {
        private readonly IXmlEncryptor _escrowEncryptor;

        public MyKeyEscrowSink(IServiceProvider services)
        {
            // Assuming I'm on a machine that's a member of the CONTOSO
            // domain, I can use the Domain Admins SID to generate an
            // encrypted payload that only they can read. Sample SID from
            // https://technet.microsoft.com/library/cc778824(v=ws.10).aspx.
            _escrowEncryptor = new DpapiNGXmlEncryptor(
                "SID=S-1-5-21-1004336348-1177238915-682003330-512",
                DpapiNGProtectionDescriptorFlags.None,
                new LoggerFactory());
        }

        public void Store(Guid keyId, XElement element)
        {
            // Encrypt the key element to the escrow encryptor.
            var encryptedXmlInfo = _escrowEncryptor.Encrypt(element);

            // A real implementation would save the escrowed key to a
            // write-only file share or some other stable storage, but
            // in this sample we'll just write it out to the console.
            Console.WriteLine($"Escrowing key {keyId}");
            Console.WriteLine(encryptedXmlInfo.EncryptedElement);

            // Note: We cannot read the escrowed key material ourselves.
            // We need to get a member of CONTOSO\Domain Admins to read
            // it for us in the event we need to recover it.
        }
    }
}
```

```
/*
 * SAMPLE OUTPUT
 *
 * Generating new key...
 * Escrowing key 38e74534-c1b8-4b43-aea1-79e856a822e5
 * <encryptedKey>
 *   <!-- This key is encrypted with Windows DPAPI-NG. -->
 *   <!-- Rule: SID=S-1-5-21-1004336348-1177238915-682003330-512 -->
 *   <value>MIIIfAYJKoZIhvcNAQcDoIIIbTCCCGkCAQ...T5rA4g==</value>
 * </encryptedKey>
 */
```

Miscellaneous ASP.NET Core Data Protection APIs

Article • 06/03/2022

> ⚠ **Warning**
>
> Types that implement any of the following interfaces should be thread-safe for multiple callers.

ISecret

The `ISecret` interface represents a secret value, such as cryptographic key material. It contains the following API surface:

- `Length`: `int`
- `Dispose()`: `void`
- `WriteSecretIntoBuffer(ArraySegment<byte> buffer)`: `void`

The `WriteSecretIntoBuffer` method populates the supplied buffer with the raw secret value. The reason this API takes the buffer as a parameter rather than returning a `byte[]` directly is that this gives the caller the opportunity to pin the buffer object, limiting secret exposure to the managed garbage collector.

The `Secret` type is a concrete implementation of `ISecret` where the secret value is stored in in-process memory. On Windows platforms, the secret value is encrypted via CryptProtectMemory.

ASP.NET Core Data Protection implementation

Article • 06/03/2022

Authenticated encryption details in ASP.NET Core

Article • 10/21/2024

Calls to IDataProtector.Protect are authenticated encryption operations. The Protect method offers both confidentiality and authenticity, and it's tied to the purpose chain that was used to derive this particular IDataProtector instance from its root IDataProtectionProvider.

IDataProtector.Protect takes a byte[] plaintext parameter and produces a byte[] protected payload, whose format is described below. (There's also an extension method overload which takes a string plaintext parameter and returns a string protected payload. If this API is used the protected payload format will still have the below structure, but it will be base64url-encoded.)

Protected payload format

The protected payload format consists of three primary components:

- A 32-bit magic header that identifies the version of the data protection system.
- A 128-bit key id that identifies the key used to protect this particular payload.
- The remainder of the protected payload is specific to the encryptor encapsulated by this key. In the example below, the key represents an AES-256-CBC + HMACSHA256 encryptor, and the payload is further subdivided as follows:
  - A 128-bit key modifier.
  - A 128-bit initialization vector.
  - 48 bytes of AES-256-CBC output.
  - An HMACSHA256 authentication tag.

A sample protected payload is illustrated below.

```
09 F0 C9 F0 80 9C 81 0C 19 66 19 40 95 36 53 F8
AA FF EE 57 57 2F 40 4C 3F 7F CC 9D CC D9 32 3E
84 17 99 16 EC BA 1F 4A A1 18 45 1F 2D 13 7A 28
79 6B 86 9C F8 B7 84 F9 26 31 FC B1 86 0A F1 56
61 CF 14 58 D3 51 6F CF 36 50 85 82 08 2D 3F 73
5F B0 AD 9E 1A B2 AE 13 57 90 C8 F5 7C 95 4E 6A
8A AA 06 EF 43 CA 19 62 84 7C 11 B2 C8 71 9D AA
```

```
52 19 2E 5B 4C 1E 54 F0 55 BE 88 92 12 C1 4B 5E
52 C9 74 A0
```

From the payload format above the first 32 bits, or 4 bytes are the magic header identifying the version (09 F0 C9 F0)

The next 128 bits, or 16 bytes is the key identifier (80 9C 81 0C 19 66 19 40 95 36 53 F8 AA FF EE 57)

The remainder contains the payload and is specific to the format used.

> ⚠ **Warning**
>
> All payloads protected to a given key will begin with the same 20-byte (magic value, key id) header. Administrators can use this fact for diagnostic purposes to approximate when a payload was generated. For example, the payload above corresponds to key {aaaaaaaa-0b0b-1c1c-2d2d-333333333333}. If after checking the key repository you find that this specific key's activation date was 2015-01-01 and its expiration date was 2015-03-01, then it's reasonable to assume that the payload (if not tampered with) was generated within that window, give or take a small fudge factor on either side.

Subkey derivation and authenticated encryption in ASP.NET Core

Article • 01/04/2023

Most keys in the key ring will contain some form of entropy and will have algorithmic information stating "CBC-mode encryption + HMAC validation" or "GCM encryption + validation". In these cases, we refer to the embedded entropy as the master keying material (or KM) for this key, and we perform a key derivation function to derive the keys that will be used for the actual cryptographic operations.

> ⓘ **Note**
>
> Keys are abstract, and a custom implementation might not behave as below. If the key provides its own implementation of `IAuthenticatedEncryptor` rather than using one of our built-in factories, the mechanism described in this section no longer applies.

Additional authenticated data and subkey derivation

The `IAuthenticatedEncryptor` interface serves as the core interface for all authenticated encryption operations. Its `Encrypt` method takes two buffers: plaintext and additionalAuthenticatedData (AAD). The plaintext contents flow unchanged the call to `IDataProtector.Protect`, but the AAD is generated by the system and consists of three components:

1. The 32-bit magic header 09 F0 C9 F0 that identifies this version of the data protection system.
2. The 128-bit key id.
3. A variable-length string formed from the purpose chain that created the `IDataProtector` that's performing this operation.

Because the AAD is unique for the tuple of all three components, we can use it to derive new keys from KM instead of using KM itself in all of our cryptographic operations. For every call to `IAuthenticatedEncryptor.Encrypt`, the following key derivation process takes place:

```
( K_E, K_H ) = SP800_108_CTR_HMACSHA512(K_M, AAD, contextHeader || keyModifier)
```

Here, we're calling the NIST SP800-108 KDF in Counter Mode (see NIST SP800-108, Sec. 5.1) with the following parameters:

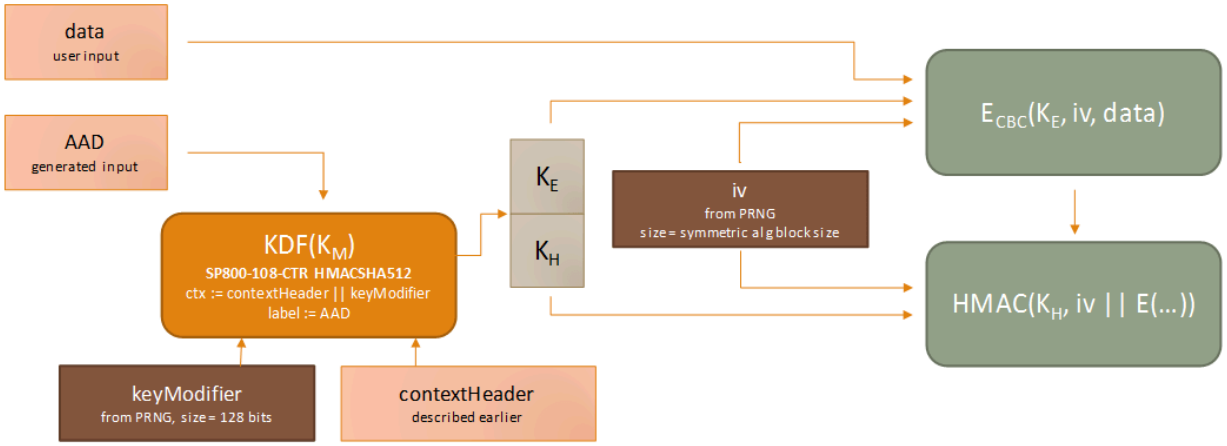
- Key derivation key (KDK) = `K_M`
- PRF = HMACSHA512
- label = additionalAuthenticatedData
- context = contextHeader || keyModifier

The context header is of variable length and essentially serves as a thumbprint of the algorithms for which we're deriving `K_E` and `K_H`. The key modifier is a 128-bit string randomly generated for each call to `Encrypt` and serves to ensure with overwhelming probability that KE and KH are unique for this specific authentication encryption operation, even if all other input to the KDF is constant.

For CBC-mode encryption + HMAC validation operations, `| K_E |` is the length of the symmetric block cipher key, and `| K_H |` is the digest size of the HMAC routine. For GCM encryption + validation operations, `| K_H | = 0`.

CBC-mode encryption + HMAC validation

Once `K_E` is generated via the above mechanism, we generate a random initialization vector and run the symmetric block cipher algorithm to encipher the plaintext. The initialization vector and ciphertext are then run through the HMAC routine initialized with the key `K_H` to produce the MAC. This process and the return value is represented graphically below.

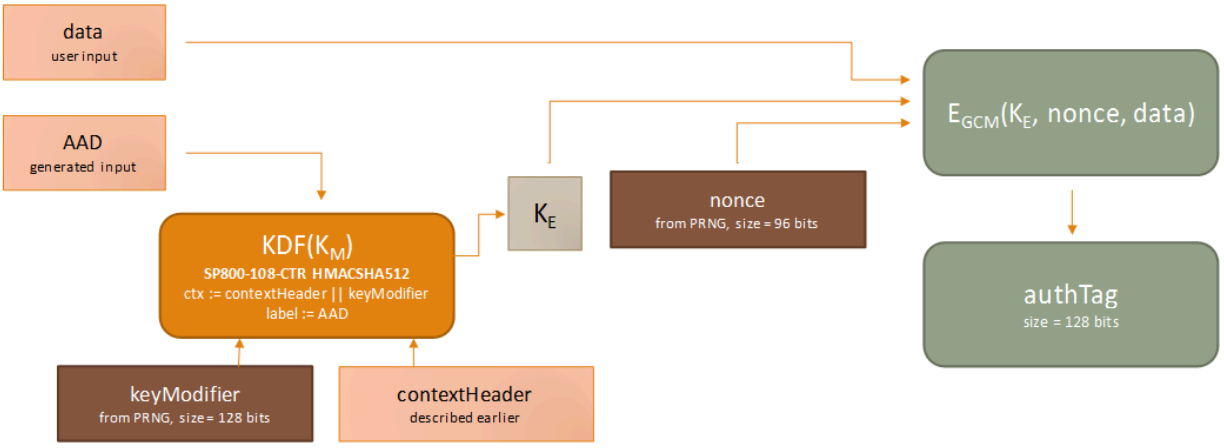


`output:= keyModifier || iv || E_cbc (K_E,iv,data) || HMAC(K_H, iv || E_cbc (K_E,iv,data))`

> ⓘ Note
>
> The `IDataProtector.Protect` implementation will prepend the magic header and key id to output before returning it to the caller. Because the magic header and key id are implicitly part of AAD, and because the key modifier is fed as input to the KDF, this means that every single byte of the final returned payload is authenticated by the MAC.

Galois/Counter Mode encryption + validation

Once `K_E` is generated via the above mechanism, we generate a random 96-bit nonce and run the symmetric block cipher algorithm to encipher the plaintext and produce the 128-bit authentication tag.



`output := keyModifier || nonce || E_gcm (K_E,nonce,data) || authTag`

> ⓘ Note
>
> Even though GCM natively supports the concept of AAD, we're still feeding AAD only to the original KDF, opting to pass an empty string into GCM for its AAD parameter. The reason for this is two-fold. First, to support agility we never want to use `K_M` directly as the encryption key. Additionally, GCM imposes very strict uniqueness requirements on its inputs. The probability that the GCM encryption routine is ever invoked on two or more distinct sets of input data with the same (key, nonce) pair must not exceed 2^-32. If we fix `K_E` we cannot perform more than 2^32 encryption operations before we run afoul of the 2^-32 limit. This might

seem like a very large number of operations, but a high-traffic web server can go through 4 billion requests in mere days, well within the normal lifetime for these keys. To stay compliant of the 2^-32 probability limit, we continue to use a 128-bit key modifier and 96-bit nonce, which radically extends the usable operation count for any given `K_M`. For simplicity of design we share the KDF code path between CBC and GCM operations, and since AAD is already considered in the KDF there's no need to forward it to the GCM routine.

Context headers in ASP.NET Core

Article • 06/03/2022

Background and theory

In the data protection system, a "key" means an object that can provide authenticated encryption services. Each key is identified by a unique id (a GUID), and it carries with it algorithmic information and entropic material. It's intended that each key carry unique entropy, but the system cannot enforce that, and we also need to account for developers who might change the key ring manually by modifying the algorithmic information of an existing key in the key ring. To achieve our security requirements given these cases the data protection system has a concept of cryptographic agility, which allows securely using a single entropic value across multiple cryptographic algorithms.

Most systems which support cryptographic agility do so by including some identifying information about the algorithm inside the payload. The algorithm's OID is generally a good candidate for this. However, one problem that we ran into is that there are multiple ways to specify the same algorithm: "AES" (CNG) and the managed Aes, AesManaged, AesCryptoServiceProvider, AesCng, and RijndaelManaged (given specific parameters) classes are all actually the same thing, and we'd need to maintain a mapping of all of these to the correct OID. If a developer wanted to provide a custom algorithm (or even another implementation of AES!), they'd have to tell us its OID. This extra registration step makes system configuration particularly painful.

Stepping back, we decided that we were approaching the problem from the wrong direction. An OID tells you what the algorithm is, but we don't actually care about this. If we need to use a single entropic value securely in two different algorithms, it's not necessary for us to know what the algorithms actually are. What we actually care about is how they behave. Any decent symmetric block cipher algorithm is also a strong pseudorandom permutation (PRP): fix the inputs (key, chaining mode, IV, plaintext) and the ciphertext output will with overwhelming probability be distinct from any other symmetric block cipher algorithm given the same inputs. Similarly, any decent keyed hash function is also a strong pseudorandom function (PRF), and given a fixed input set its output will overwhelmingly be distinct from any other keyed hash function.

We use this concept of strong PRPs and PRFs to build up a context header. This context header essentially acts as a stable thumbprint over the algorithms in use for any given operation, and it provides the cryptographic agility needed by the data protection system. This header is reproducible and is used later as part of the subkey derivation

process. There are two different ways to build the context header depending on the modes of operation of the underlying algorithms.

CBC-mode encryption + HMAC authentication

The context header consists of the following components:

- [16 bits] The value 00 00, which is a marker meaning "CBC encryption + HMAC authentication".
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The key length (in bytes, big-endian) of the HMAC algorithm. (Currently the key size always matches the digest size.)
- [32 bits] The digest size (in bytes, big-endian) of the HMAC algorithm.
- `EncCBC(K_E, IV, "")`, which is the output of the symmetric block cipher algorithm given an empty string input and where IV is an all-zero vector. The construction of `K_E` is described below.
- `MAC(K_H, "")`, which is the output of the HMAC algorithm given an empty string input. The construction of `K_H` is described below.

Ideally, we could pass all-zero vectors for `K_E` and `K_H`. However, we want to avoid the situation where the underlying algorithm checks for the existence of weak keys before performing any operations (notably DES and 3DES), which precludes using a simple or repeatable pattern like an all-zero vector.

Instead, we use the NIST SP800-108 KDF in Counter Mode (see NIST SP800-108, Sec. 5.1) with a zero-length key, label, and context and HMACSHA512 as the underlying PRF. We derive `| K_E | + | K_H |` bytes of output, then decompose the result into `K_E` and `K_H` themselves. Mathematically, this is represented as follows.

```
( K_E || K_H ) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")
```

Example: AES-192-CBC + HMACSHA256

As an example, consider the case where the symmetric block cipher algorithm is AES-192-CBC and the validation algorithm is HMACSHA256. The system would generate the context header using the following steps.

First, let `( K_E || K_H ) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")`, where `| K_E | = 192 bits` and `| K_H | = 256 bits` per the specified algorithms. This leads to `K_E = 5BB6..21DD` and `K_H = A04A..00A9` in the example below:

```
5B B6 C9 83 13 78 22 1D 8E 10 73 CA CF 65 8E B0
61 62 42 71 CB 83 21 DD A0 4A 05 00 5B AB C0 A2
49 6F A5 61 E3 E2 49 87 AA 63 55 CD 74 0A DA C4
B7 92 3D BF 59 90 00 A9
```

Next, compute `Enc_CBC (K_E, IV, "")` for AES-192-CBC given `IV = 0*` and `K_E` as above.

`result := F474B1872B3B53E4721DE19C0841DB6F`

Next, compute `MAC(K_H, "")` for HMACSHA256 given `K_H` as above.

`result := D4791184B996092EE1202F36E8608FA8FBD98ABDFF5402F264B1D7211536220C`

This produces the full context header below:

```
00 00 00 00 00 18 00 00 00 10 00 00 00 20 00 00
00 20 F4 74 B1 87 2B 3B 53 E4 72 1D E1 9C 08 41
DB 6F D4 79 11 84 B9 96 09 2E E1 20 2F 36 E8 60
8F A8 FB D9 8A BD FF 54 02 F2 64 B1 D7 21 15 36
22 0C
```

This context header is the thumbprint of the authenticated encryption algorithm pair (AES-192-CBC encryption + HMACSHA256 validation). The components, as described above are:

- the marker `(00 00)`
- the block cipher key length `(00 00 00 18)`
- the block cipher block size `(00 00 00 10)`
- the HMAC key length `(00 00 00 20)`

- the HMAC digest size `(00 00 00 20)`
- the block cipher PRP output `(F4 74 - DB 6F)` and
- the HMAC PRF output `(D4 79 - end)`.

> ⓘ **Note**
>
> The CBC-mode encryption + HMAC authentication context header is built the same way regardless of whether the algorithms implementations are provided by Windows CNG or by managed SymmetricAlgorithm and KeyedHashAlgorithm types. This allows applications running on different operating systems to reliably produce the same context header even though the implementations of the algorithms differ between OSes. (In practice, the KeyedHashAlgorithm doesn't have to be a proper HMAC. It can be any keyed hash algorithm type.)

Example: 3DES-192-CBC + HMACSHA1

First, let `( K_E || K_H ) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")`, where `| K_E | = 192 bits` and `| K_H | = 160 bits` per the specified algorithms. This leads to `K_E = A219..E2BB` and `K_H = DC4A..B464` in the example below:

```
A2 19 60 2F 83 A9 13 EA B0 61 3A 39 B8 A6 7E 22
61 D9 F8 6C 10 51 E2 BB DC 4A 00 D7 03 A2 48 3E
D1 F7 5A 34 EB 28 3E D7 D4 67 B4 64
```

Next, compute `Enc_CBC (K_E, IV, "")` for 3DES-192-CBC given `IV = 0*` and `K_E` as above.

`result := ABB100F81E53E10E`

Next, compute `MAC(K_H, "")` for HMACSHA1 given `K_H` as above.

`result := 76EB189B35CF03461DDF877CD9F4B1B4D63A7555`

This produces the full context header which is a thumbprint of the authenticated encryption algorithm pair (3DES-192-CBC encryption + HMACSHA1 validation), shown below:

```
00 00 00 00 00 18 00 00 00 08 00 00 00 14 00 00
00 14 AB B1 00 F8 1E 53 E1 0E 76 EB 18 9B 35 CF
03 46 1D DF 87 7C D9 F4 B1 B4 D6 3A 75 55
```

The components break down as follows:

- the marker `(00 00)`
- the block cipher key length `(00 00 00 18)`
- the block cipher block size `(00 00 00 08)`
- the HMAC key length `(00 00 00 14)`
- the HMAC digest size `(00 00 00 14)`
- the block cipher PRP output `(AB B1 - E1 0E)` and
- the HMAC PRF output `(76 EB - end)`.

Galois/Counter Mode encryption + authentication

The context header consists of the following components:

- [16 bits] The value 00 01, which is a marker meaning "GCM encryption + authentication".
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The nonce size (in bytes, big-endian) used during authenticated encryption operations. (For our system, this is fixed at nonce size = 96 bits.)
- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm. (For GCM, this is fixed at block size = 128 bits.)
- [32 bits] The authentication tag size (in bytes, big-endian) produced by the authenticated encryption function. (For our system, this is fixed at tag size = 128 bits.)
- [128 bits] The tag of `Enc_GCM (K_E, nonce, "")`, which is the output of the symmetric block cipher algorithm given an empty string input and where nonce is a 96-bit all-zero vector.

`K_E` is derived using the same mechanism as in the CBC encryption + HMAC authentication scenario. However, since there's no `K_H` in play here, we essentially have `| K_H | = 0`, and the algorithm collapses to the below form.

```
K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")
```

Example: AES-256-GCM

First, let `K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")`, where `| K_E | = 256 bits`.

```
K_E := 22BC6F1B171C08C4AE2F27444AF8FC8B3087A90006CAEA91FDCFB47C1B8733B8
```

Next, compute the authentication tag of `Enc_GCM (K_E, nonce, "")` for AES-256-GCM given `nonce = 096` and `K_E` as above.

```
result := E7DCCE66DF855A323A6BB7BD7A59BE45
```

This produces the full context header below:

```

00 01 00 00 00 20 00 00 00 0C 00 00 00 10 00 00
00 10 E7 DC CE 66 DF 85 5A 32 3A 6B B7 BD 7A 59
BE 45
```

The components break down as follows:

- the marker `(00 01)`
- the block cipher key length `(00 00 00 20)`
- the nonce size `(00 00 00 0C)`
- the block cipher block size `(00 00 00 10)`
- the authentication tag size `(00 00 00 10)` and
- the authentication tag from running the block cipher `(E7 DC - end)`.

Key management in ASP.NET Core

Article • 09/27/2024

The data protection system automatically manages the lifetime of master keys used to protect and unprotect payloads. Each key can exist in one of four stages:

- Created - the key exists in the key ring but has not yet been activated. The key shouldn't be used for new Protect operations until sufficient time has elapsed that the key has had a chance to propagate to all machines that are consuming this key ring.
- Active - the key exists in the key ring and should be used for all new Protect operations.
- Expired - the key has run its natural lifetime and should no longer be used for new Protect operations.
- Revoked - the key is compromised and must not be used for new Protect operations.

Created, active, and expired keys may all be used to unprotect incoming payloads. Revoked keys by default may not be used to unprotect payloads, but the application developer can override this behavior if necessary.

> ⚠ **Warning**
>
> The developer might be tempted to delete a key from the key ring (for example by deleting the corresponding file from the file system). At that point, all data protected by the key is permanently undecipherable, and there's no emergency override like there's with revoked keys. Deleting a key is truly destructive behavior.

Default key selection

When the data protection system reads the key ring from the backing repository, it will attempt to locate a "default" key from the key ring. The default key is used for new Protect operations.

The general heuristic is that the data protection system chooses the key with the most recent activation date as the default key. (There's a small fudge factor to allow for server-to-server clock skew.) If the key is expired or revoked, and if the application has

not disabled automatic key generation, then a new key will be generated with immediate activation per the key expiration and rolling policy below.

The reason the data protection system generates a new key immediately rather than falling back to a different key is that new key generation should be treated as an implicit expiration of all keys that were activated prior to the new key. The general idea is that new keys may have been configured with different algorithms or encryption-at-rest mechanisms than old keys, and the system should prefer the current configuration over falling back.

There's an exception. If the application developer has disabled automatic key generation, then the data protection system must choose something as the default key. In this fallback scenario, the system will choose the non-revoked key with the most recent activation date, with preference given to keys that have had time to propagate to other machines in the cluster. The fallback system may end up choosing an expired default key as a result. The fallback system will never choose a revoked key as the default key, and if the key ring is empty or every key has been revoked then the system will produce an error upon initialization.

Key expiration and rolling

When a key is created, it's automatically given an activation date of { now + 2 days } and an expiration date of { now + 90 days }. The 2-day delay before activation gives the key time to propagate through the system. That is, it allows other applications pointing at the backing store to observe the key at their next auto-refresh period, thus maximizing the chances that when the key ring does become active it has propagated to all applications that might need to use it.

If the default key will expire within 2 days and if the key ring doesn't already have a key that will be active upon expiration of the default key, then the data protection system will automatically persist a new key to the key ring. This new key has an activation date of { default key's expiration date } and an expiration date of { now + 90 days }. This allows the system to automatically roll keys on a regular basis with no interruption of service.

There might be circumstances where a key will be created with immediate activation. One example would be when the application hasn't run for a time and all keys in the key ring are expired. When this happens, the key is given an activation date of { now } without the normal 2-day activation delay.

The default key lifetime is 90 days, though this is configurable as in the following example.

C#

```
services.AddDataProtection()
       // use 14-day lifetime instead of 90-day lifetime
       .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
```

An administrator can also change the default system-wide, though an explicit call to `SetDefaultKeyLifetime` will override any system-wide policy. The default key lifetime cannot be shorter than 7 days.

Automatic key ring refresh

When the data protection system initializes, it reads the key ring from the underlying repository and caches it in memory. This cache allows Protect and Unprotect operations to proceed without hitting the backing store. The system will automatically check the backing store for changes approximately every 24 hours or when the current default key expires, whichever comes first.

> ⚠ **Warning**
>
> Developers should very rarely (if ever) need to use the key management APIs directly. The data protection system will perform automatic key management as described above.

The data protection system exposes an interface `IKeyManager` that can be used to inspect and make changes to the key ring. The DI system that provided the instance of `IDataProtectionProvider` can also provide an instance of `IKeyManager` for your consumption. Alternatively, you can pull the `IKeyManager` straight from the `IServiceProvider` as in the example below.

Any operation which modifies the key ring (creating a new key explicitly or performing a revocation) will invalidate the in-memory cache. The next call to `Protect` or `Unprotect` will cause the data protection system to reread the key ring and recreate the cache.

The sample below demonstrates using the `IKeyManager` interface to inspect and manipulate the key ring, including revoking existing keys and generating a new key manually.

C#

```
using System;
using System.IO;
```

```
using System.Threading;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // perform a protect operation to force the system to put at least
        // one key in the key ring

services.GetDataProtector("Sample.KeyManager.v1").Protect("payload");
        Console.WriteLine("Performed a protect operation.");
        Thread.Sleep(2000);

        // get a reference to the key manager
        var keyManager = services.GetService<IKeyManager>();

        // list all keys in the key ring
        var allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId:B}: Created =
{key.CreationDate:u}, IsRevoked = {key.IsRevoked}");
        }

        // revoke all keys in the key ring
        keyManager.RevokeAllKeys(DateTimeOffset.Now, reason: "Revocation
reason here.");
        Console.WriteLine("Revoked all existing keys.");

        // add a new key to the key ring with immediate activation and a 1-
month expiration
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddMonths(1));
        Console.WriteLine("Added a new key.");

        // list all keys in the key ring
        allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId:B}: Created =
{key.CreationDate:u}, IsRevoked = {key.IsRevoked}");
        }
```

```
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Performed a protect operation.
 * The key ring contains 1 key(s).
 * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18
22:20:49Z, IsRevoked = False
 * Revoked all existing keys.
 * Added a new key.
 * The key ring contains 2 key(s).
 * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18
22:20:49Z, IsRevoked = True
 * Key {2266fc40-e2fb-48c6-8ce2-5fde6b1493f7}: Created = 2015-03-18
22:20:51Z, IsRevoked = False
 */
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

Key storage

The data protection system has a heuristic whereby it attempts to deduce an appropriate key storage location and encryption-at-rest mechanism automatically. The key persistence mechanism is also configurable by the app developer. The following documents discuss the in-box implementations of these mechanisms:

- Key storage providers in ASP.NET Core
- Key encryption at rest in Windows and Azure using ASP.NET Core

Key storage providers in ASP.NET Core

Article • 10/30/2024

The data protection system employs a discovery mechanism by default to determine where cryptographic keys should be persisted. The developer can override the default discovery mechanism and manually specify the location.

> ⚠ **Warning**
>
> If you specify an explicit key persistence location, the data protection system deregisters the default key encryption at rest mechanism, so keys are no longer encrypted at rest. It's recommended that you additionally specify an explicit key encryption mechanism for production deployments.

File system

To configure a file system-based key repository, call the PersistKeysToFileSystem configuration routine as shown below. Provide a DirectoryInfo pointing to the repository where keys should be stored:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys\"));
}
```

The `DirectoryInfo` can point to a directory on the local machine, or it can point to a folder on a network share. If pointing to a directory on the local machine (and the scenario is that only apps on the local machine require access to use this repository), consider using Windows DPAPI (on Windows) to encrypt the keys at rest. Otherwise, consider using an X.509 certificate to encrypt keys at rest.

Azure Storage

The Azure.Extensions.AspNetCore.DataProtection.Blobs package allows storing data protection keys in Azure Blob Storage. Keys can be shared across several instances of a web app. Apps can share authentication cookies or CSRF protection across multiple servers.

To configure the Azure Blob Storage provider, call one of the PersistKeysToAzureBlobStorage overloads.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blob URI including SAS
token>"));
}
```

If the web app is running as an Azure service, connection string can be used to authenticate to Azure storage by using Azure.Storage.Blobs.

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

C#

```
string connectionString = "<connection_string>";
string containerName = "my-key-container";
string blobName = "keys.xml";
BlobContainerClient container = new BlobContainerClient(connectionString,
containerName);

// optional - provision the container automatically
await container.CreateIfNotExistsAsync();

BlobClient blobClient = container.GetBlobClient(blobName);

services.AddDataProtection()
    .PersistKeysToAzureBlobStorage(blobClient);
```

> ⓘ **Note**
>
> The connection string to your storage account can be found in the Azure Portal under the "Access Keys" section or by running the following CLI command:

Bash

```
az storage account show-connection-string --name <account_name> --
resource-group <resource_group>
```

Redis

The Microsoft.AspNetCore.DataProtection.StackExchangeRedis package allows storing data protection keys in a Redis cache. Keys can be shared across several instances of a web app. Apps can share authentication cookies or CSRF protection across multiple servers.

To configure on Redis, call one of the PersistKeysToStackExchangeRedis overloads:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    var redis = ConnectionMultiplexer.Connect("<URI>");
    services.AddDataProtection()
        .PersistKeysToStackExchangeRedis(redis, "DataProtection-Keys");
}
```

For more information, see the following topics:

- StackExchange.Redis ConnectionMultiplexer
- Azure Redis Cache
- ASP.NET Core DataProtection samples

Registry

Only applies to Windows deployments.

Sometimes the app might not have write access to the file system. Consider a scenario where an app is running as a virtual service account (such as *w3wp.exe*'s app pool identity). In these cases, the administrator can provision a registry key that's accessible by the service account identity. Call the PersistKeysToRegistry extension method as shown below. Provide a RegistryKey pointing to the location where cryptographic keys should be stored:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()

.PersistKeysToRegistry(Registry.CurrentUser.OpenSubKey(@"SOFTWARE\Sample\keys", true));
}
```

> **ⓘ Important**
>
> We recommend using **Windows DPAPI** to encrypt the keys at rest.

Entity Framework Core

The Microsoft.AspNetCore.DataProtection.EntityFrameworkCore package provides a mechanism for storing data protection keys to a database using Entity Framework Core. The `Microsoft.AspNetCore.DataProtection.EntityFrameworkCore` NuGet package must be added to the project file, it's not part of the Microsoft.AspNetCore.App metapackage.

With this package, keys can be shared across multiple instances of a web app.

To configure the EF Core provider, call the PersistKeysToDbContext method:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    // Add a DbContext to store your Database Keys
    services.AddDbContext<MyKeysContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("MyKeysConnection")));

    // using Microsoft.AspNetCore.DataProtection;
    services.AddDataProtection()
        .PersistKeysToDbContext<MyKeysContext>();

    services.AddDefaultIdentity<IdentityUser>()
```

```
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The generic parameter, `TContext`, must inherit from DbContext and implement IDataProtectionKeyContext:

C#

```
using Microsoft.AspNetCore.DataProtection.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using WebApp1.Data;

namespace WebApp1
{
    class MyKeysContext : DbContext, IDataProtectionKeyContext
    {
        // A recommended constructor overload when using EF Core
        // with dependency injection.
        public MyKeysContext(DbContextOptions<MyKeysContext> options)
            : base(options) { }

        // This maps to the table that stores keys.
        public DbSet<DataProtectionKey> DataProtectionKeys { get; set; }
    }
}
```

Create the `DataProtectionKeys` table.

Visual Studio

Execute the following commands in the **Package Manager Console** (PMC) window:

PowerShell

```
Add-Migration AddDataProtectionKeys -Context MyKeysContext
Update-Database -Context MyKeysContext
```

`MyKeysContext` is the `DbContext` defined in the preceding code sample. If you're using a `DbContext` with a different name, substitute your `DbContext` name for `MyKeysContext`.

The `DataProtectionKeys` class/entity adopts the structure shown in the following table.

Expand table

Property/Field	CLR Type	SQL Type
`Id`	`int`	`int`, PK, `IDENTITY(1,1)`, not null
`FriendlyName`	`string`	`nvarchar(MAX)`, null
`Xml`	`string`	`nvarchar(MAX)`, null

Custom key repository

If the in-box mechanisms aren't appropriate, the developer can specify their own key persistence mechanism by providing a custom IXmlRepository.

Key encryption at rest in Windows and Azure using ASP.NET Core

Article • 03/17/2023

The data protection system employs a discovery mechanism by default to determine how cryptographic keys should be encrypted at rest. The developer can override the discovery mechanism and manually specify how keys should be encrypted at rest.

> ⚠ **Warning**
>
> If you specify an explicit **key persistence location**, the data protection system deregisters the default key encryption at rest mechanism. Consequently, keys are no longer encrypted at rest. We recommend that you **specify an explicit key encryption mechanism** for production deployments. The encryption-at-rest mechanism options are described in this topic.

Azure Key Vault

To store keys in Azure Key Vault, configure the system with ProtectKeysWithAzureKeyVault in the `Startup` class:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blobUriWithSasToken>"))
        .ProtectKeysWithAzureKeyVault("<keyIdentifier>", "<clientId>", "
<clientSecret>");
}
```

For more information, see Configure ASP.NET Core Data Protection: ProtectKeysWithAzureKeyVault.

Windows DPAPI

Only applies to Windows deployments.

When Windows DPAPI is used, key material is encrypted with CryptProtectData before being persisted to storage. DPAPI is an appropriate encryption mechanism for data

that's never read outside of the current machine (though it's possible to back these keys up to Active Directory). To configure DPAPI key-at-rest encryption, call one of the ProtectKeysWithDpapi) extension methods:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    // Only the local user account can decrypt the keys
    services.AddDataProtection()
        .ProtectKeysWithDpapi();
}
```

If `ProtectKeysWithDpapi` is called with no parameters, only the current Windows user account can decipher the persisted key ring. You can optionally specify that any user account on the machine (not just the current user account) be able to decipher the key ring:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    // All user accounts on the machine can decrypt the keys
    services.AddDataProtection()
        .ProtectKeysWithDpapi(protectToLocalMachine: true);
}
```

X.509 certificate

If the app is spread across multiple machines, it may be convenient to distribute a shared X.509 certificate across the machines and configure the hosted apps to use the certificate for encryption of keys at rest:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()

.ProtectKeysWithCertificate("3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0");
}
```

Due to .NET Framework limitations, only certificates with CAPI private keys are supported. See the content below for possible workarounds to these limitations.

Windows DPAPI-NG

This mechanism is available only on Windows 8/Windows Server 2012 or later.

Beginning with Windows 8, Windows OS supports DPAPI-NG (also called CNG DPAPI). For more information, see About CNG DPAPI.

The principal is encoded as a protection descriptor rule. In the following example that calls ProtectKeysWithDpapiNG, only the domain-joined user with the specified SID can decrypt the key ring:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    // Uses the descriptor rule "SID=S-1-5-21-..."
    services.AddDataProtection()
        .ProtectKeysWithDpapiNG("SID=S-1-5-21-...",
        flags: DpapiNGProtectionDescriptorFlags.None);
}
```

There's also a parameterless overload of `ProtectKeysWithDpapiNG`. Use this convenience method to specify the rule "SID={CURRENT_ACCOUNT_SID}", where *CURRENT_ACCOUNT_SID* is the SID of the current Windows user account:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    // Use the descriptor rule "SID={current account SID}"
    services.AddDataProtection()
        .ProtectKeysWithDpapiNG();
}
```

In this scenario, the AD domain controller is responsible for distributing the encryption keys used by the DPAPI-NG operations. The target user can decipher the encrypted payload from any domain-joined machine (provided that the process is running under their identity).

Certificate-based encryption with Windows DPAPI-NG

If the app is running on Windows 8.1/Windows Server 2012 R2 or later, you can use Windows DPAPI-NG to perform certificate-based encryption. Use the rule descriptor

string "CERTIFICATE=HashId:THUMBPRINT", where *THUMBPRINT* is the hex-encoded SHA1 thumbprint of the certificate:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()

.ProtectKeysWithDpapiNG("CERTIFICATE=HashId:3BCE558E2...B5AEA2A9BD2575A0",
            flags: DpapiNGProtectionDescriptorFlags.None);
}
```

Any app pointed at this repository must be running on Windows 8.1/Windows Server 2012 R2 or later to decipher the keys.

Custom key encryption

If the in-box mechanisms aren't appropriate, the developer can specify their own key encryption mechanism by providing a custom IXmlEncryptor.

Key immutability and key settings in ASP.NET Core

Article • 06/03/2022

Once an object is persisted to the backing store, its representation is forever fixed. New data can be added to the backing store, but existing data can never be mutated. The primary purpose of this behavior is to prevent data corruption.

One consequence of this behavior is that once a key is written to the backing store, it's immutable. Its creation, activation, and expiration dates can never be changed, though it can revoked by using `IKeyManager`. Additionally, its underlying algorithmic information, master keying material, and encryption at rest properties are also immutable.

If the developer changes any setting that affects key persistence, those changes won't go into effect until the next time a key is generated, either via an explicit call to `IKeyManager.CreateNewKey` or via the data protection system's own automatic key generation behavior. The settings that affect key persistence are as follows:

- The default key lifetime
- The key encryption at rest mechanism
- The algorithmic information contained within the key

If you need these settings to kick in earlier than the next automatic key rolling time, consider making an explicit call to `IKeyManager.CreateNewKey` to force the creation of a new key. Remember to provide an explicit activation date ({ now + 2 days } is a good rule of thumb to allow time for the change to propagate) and expiration date in the call.

> **Tip**
>
> All applications touching the repository should specify the same settings with the `IDataProtectionBuilder` extension methods. Otherwise, the properties of the persisted key will be dependent on the particular application that invoked the key generation routines.

Key storage format in ASP.NET Core

Article • 10/21/2024

Objects are stored at rest in XML representation. The default directory for key storage is:

- Windows: *%LOCALAPPDATA%\ASP.NET\DataProtection-Keys*
- macOS / Linux: *$HOME/.aspnet/DataProtection-Keys*

The <key> element

Keys exist as top-level objects in the key repository. By convention keys have the filename `key-{guid}.xml`, where {guid} is the id of the key. Each such file contains a single key. The format of the file is as follows.

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<key id="aaaaaaaa-0b0b-1c1c-2d2d-333333333333" version="1">
  <creationDate>2015-03-19T23:32:02.3949887Z</creationDate>
  <activationDate>2015-03-19T23:32:02.3839429Z</activationDate>
  <expirationDate>2015-06-17T23:32:02.3839429Z</expirationDate>
  <descriptor deserializerType="{deserializerType}">
    <descriptor>
      <encryption algorithm="AES_256_CBC" />
      <validation algorithm="HMACSHA256" />
      <enc:encryptedSecret decryptorType="{decryptorType}" xmlns:enc="...">
        <encryptedKey>
          <!-- This key is encrypted with Windows DPAPI. -->
          <value>AQAAANCM...8/zeP8lcwAg==</value>
        </encryptedKey>
      </enc:encryptedSecret>
    </descriptor>
  </descriptor>
</key>
```

The <key> element contains the following attributes and child elements:

- The key id. This value is treated as authoritative; the filename is simply a nicety for human readability.
- The version of the <key> element, currently fixed at 1.
- The key's creation, activation, and expiration dates.
- A <descriptor> element, which contains information on the authenticated encryption implementation contained within this key.

In the above example, the key's id is {aaaaaaaa-0b0b-1c1c-2d2d-333333333333}, it was created and activated on March 19, 2015, and it has a lifetime of 90 days. (Occasionally the activation date might be slightly before the creation date as in this example. This is due to a nit in how the APIs work and is harmless in practice.)

The <descriptor> element

The outer <descriptor> element contains an attribute deserializerType, which is the assembly-qualified name of a type which implements IAuthenticatedEncryptorDescriptorDeserializer. This type is responsible for reading the inner <descriptor> element and for parsing the information contained within.

The particular format of the <descriptor> element depends on the authenticated encryptor implementation encapsulated by the key, and each deserializer type expects a slightly different format for this. In general, though, this element will contain algorithmic information (names, types, OIDs, or similar) and secret key material. In the above example, the descriptor specifies that this key wraps AES-256-CBC encryption + HMACSHA256 validation.

The <encryptedSecret> element

An **<encryptedSecret>** element which contains the encrypted form of the secret key material may be present if encryption of secrets at rest is enabled. The attribute `decryptorType` is the assembly-qualified name of a type which implements IXmlDecryptor. This type is responsible for reading the inner **<encryptedKey>** element and decrypting it to recover the original plaintext.

As with `<descriptor>`, the particular format of the `<encryptedSecret>` element depends on the at-rest encryption mechanism in use. In the above example, the master key is encrypted using Windows DPAPI per the comment.

The <revocation> element

Revocations exist as top-level objects in the key repository. By convention revocations have the filename *`revocation-{timestamp}.xml`* (for revoking all keys before a specific date) or *`revocation-{guid}.xml`* (for revoking a specific key). Each file contains a single <revocation> element.

For revocations of individual keys, the file contents will be as below.

```
XML
```

```xml
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T22:45:30.2616742Z</revocationDate>
  <key id="bbbbbbbb-1c1c-2d2d-3e3e-444444444444" />
  <reason>human-readable reason</reason>
</revocation>
```

In this case, only the specified key is revoked. If the key id is "*", however, as in the below example, all keys whose creation date is prior to the specified revocation date are revoked.

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T15:45:45.7366491-07:00</revocationDate>
  <!-- All keys created before the revocation date are revoked. -->
  <key id="*" />
  <reason>human-readable reason</reason>
</revocation>
```

The <reason> element is never read by the system. It's simply a convenient place to store a human-readable reason for revocation.

Ephemeral data protection providers in ASP.NET Core

Article • 02/28/2023

There are scenarios where an application needs a throwaway `IDataProtectionProvider`. For example, the developer might just be experimenting in a one-off console application, or the application itself is transient (it's scripted or a unit test project). To support these scenarios the Microsoft.AspNetCore.DataProtection package includes a type `EphemeralDataProtectionProvider`. This type provides a basic implementation of `IDataProtectionProvider` whose key repository is held solely in-memory and isn't written out to any backing store.

Each instance of `EphemeralDataProtectionProvider` uses its own unique primary key. Therefore, if an `IDataProtector` rooted at an `EphemeralDataProtectionProvider` generates a protected payload, that payload can only be unprotected by an equivalent `IDataProtector` (given the same purpose chain) rooted at the same `EphemeralDataProtectionProvider` instance.

The following sample demonstrates instantiating an `EphemeralDataProtectionProvider` and using it to protect and unprotect data.

C#

```
using System;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        const string purpose = "Ephemeral.App.v1";

        // create an ephemeral provider and demonstrate that it can round-
trip a payload
        var provider = new EphemeralDataProtectionProvider();
        var protector = provider.CreateProtector(purpose);
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
```

```
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        // if I create a new ephemeral provider, it won't be able to
unprotect existing
        // payloads, even if I specify the same purpose
        provider = new EphemeralDataProtectionProvider();
        protector = provider.CreateProtector(purpose);
        unprotectedPayload = protector.Unprotect(protectedPayload); //
THROWS
    }
}

/*
* SAMPLE OUTPUT
*
* Enter input: Hello!
* Protect returned: CfDJ8AAAAAAAAAAAAAAAAAAA...uGoxWLjGKtm1SkNACQ
* Unprotect returned: Hello!
* << throws CryptographicException >>
*/
```

Compatibility in ASP.NET Core

Article • 06/03/2022

- Replacing ASP.NET <machineKey> in ASP.NET Core

Replace the ASP.NET machineKey in ASP.NET Core

Article • 06/03/2022

The implementation of the `<machineKey>` element in ASP.NET is replaceable. This allows most calls to ASP.NET cryptographic routines to be routed through a replacement data protection mechanism, including the new data protection system.

Package installation

> **Note**
>
> The new data protection system can only be installed into an existing ASP.NET application targeting .NET 4.5.1 or later. Installation will fail if the application targets .NET 4.5 or lower.

To install the new data protection system into an existing ASP.NET 4.5.1+ project, install the package Microsoft.AspNetCore.DataProtection.SystemWeb. This will instantiate the data protection system using the default configuration settings.

When you install the package, it inserts a line into *Web.config* that tells ASP.NET to use it for most cryptographic operations, including forms authentication, view state, and calls to MachineKey.Protect. It does not use the data protection API. The line that's inserted reads as follows.

XML

```xml
<machineKey compatibilityMode="Framework45" dataProtectorType="..." />
```

> **Tip**
>
> You can tell if the new data protection system is active by inspecting fields like `__VIEWSTATE`, which should begin with "CfDJ8" as in the example below. "CfDJ8" is the base64 representation of the magic "09 F0 C9 F0" header that identifies a payload protected by the data protection system.

HTML

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="CfDJ8AWPr2EQPTBGs3L2GCZOpk...">
```

Package configuration

The data protection system is instantiated with a default zero-setup configuration. However, since by default keys are persisted to the local file system, this won't work for applications which are deployed in a farm. To resolve this, you can provide configuration by creating a type which subclasses DataProtectionStartup and overrides its ConfigureServices method.

Below is an example of a custom data protection startup type which configured both where keys are persisted and how they're encrypted at rest. It also overrides the default app isolation policy by providing its own application name.

C#

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.SystemWeb;
using Microsoft.Extensions.DependencyInjection;

namespace DataProtectionDemo
{
    public class MyDataProtectionStartup : DataProtectionStartup
    {
        public override void ConfigureServices(IServiceCollection services)
        {
            services.AddDataProtection()
                .SetApplicationName("my-app")
                .PersistKeysToFileSystem(new
DirectoryInfo(@"\\server\share\myapp-keys\"))
                .ProtectKeysWithCertificate("thumbprint");
        }
    }
}
```

Tip

You can also use `<machineKey applicationName="my-app" ... />` in place of an explicit call to SetApplicationName. This is a convenience mechanism to avoid forcing the developer to create a DataProtectionStartup-derived type if all they wanted to configure was setting the application name.

To enable this custom configuration, go back to Web.config and look for the `<appSettings>` element that the package install added to the config file. It will look like the following markup:

XML

```
<appSettings>
  <!--
  If you want to customize the behavior of the ASP.NET Core Data Protection
stack, set the
  "aspnet:dataProtectionStartupType" switch below to be the fully-qualified
name of a
  type which subclasses
Microsoft.AspNetCore.DataProtection.SystemWeb.DataProtectionStartup.
  -->
  <add key="aspnet:dataProtectionStartupType" value="" />
</appSettings>
```

Fill in the blank value with the assembly-qualified name of the DataProtectionStartup-derived type you just created. If the name of the application is DataProtectionDemo, this would look like the below.

XML

```
<add key="aspnet:dataProtectionStartupType"
     value="DataProtectionDemo.MyDataProtectionStartup, DataProtectionDemo"
/>
```

The newly-configured data protection system is now ready for use inside the application.

Safe storage of app secrets in development in ASP.NET Core

Article • 11/04/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Rick Anderson and Kirk Larkin

View or download sample code (how to download)

This article explains how to manage sensitive data for an ASP.NET Core app on a development machine. Never store passwords or other sensitive data in source code or configuration files. Production secrets shouldn't be used for development or test. Secrets shouldn't be deployed with the app. Production secrets should be accessed through a controlled means like Azure Key Vault. Azure test and production secrets can be stored and protected with the Azure Key Vault configuration provider.

For more information on authentication for deployed test and production apps, see Secure authentication flows.

To use user secrets in a .NET console app, see this GitHub issue.

Environment variables

Environment variables are used to avoid storage of app secrets in code or in local configuration files. Environment variables override configuration values for all previously specified configuration sources.

Consider an ASP.NET Core web app in which **Individual User Accounts** security is enabled. A default database connection string is included in the project's `appsettings.json` file with the key `DefaultConnection`. The default connection string is for LocalDB, which runs in user mode and doesn't require a password. During app deployment, the `DefaultConnection` key value can be overridden with an environment

variable's value. The environment variable may store the complete connection string with sensitive credentials.

> ⚠ **Warning**
>
> Environment variables are generally stored in plain, unencrypted text. If the machine or process is compromised, environment variables can be accessed by untrusted parties. Additional measures to prevent disclosure of user secrets may be required.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` separator is not supported by Bash. The double underscore, `__`, is:

- Supported by all platforms.
- Automatically replaced by a colon, `:`.

Secret Manager

The Secret Manager tool stores sensitive data during application development. In this context, a piece of sensitive data is an app secret. App secrets are stored in a separate location from the project tree. The app secrets are associated with a specific project or shared across several projects. The app secrets aren't checked into source control.

> ⚠ **Warning**
>
> The Secret Manager tool doesn't encrypt the stored secrets and shouldn't be treated as a trusted store. It's for development purposes only. The keys and values are stored in a JSON configuration file in the user profile directory.

How the Secret Manager tool works

The Secret Manager tool hides implementation details, such as where and how the values are stored. You can use the tool without knowing these implementation details. The values are stored in a JSON file in the local machine's user profile folder:

Windows

File system path:

```
%APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json
```

In the preceding file paths, replace `<user_secrets_id>` with the `UserSecretsId` value specified in the project file.

Don't write code that depends on the location or format of data saved with the Secret Manager tool. These implementation details may change. For example, the secret values aren't encrypted.

Enable secret storage

The Secret Manager tool operates on project-specific configuration settings stored in your user profile.

Use the CLI

The Secret Manager tool includes an `init` command. To use user secrets, run the following command in the project directory:

.NET CLI

```
dotnet user-secrets init
```

The preceding command adds a `UserSecretsId` element within a `PropertyGroup` of the project file. By default, the inner text of `UserSecretsId` is a GUID. The inner text is arbitrary, but is unique to the project.

XML

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.1</TargetFramework>
  <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>
</PropertyGroup>
```

Use Visual Studio

In Visual Studio, right-click the project in Solution Explorer, and select **Manage User Secrets** from the context menu. This gesture adds a `UserSecretsId` element, populated with a GUID, to the project file.

If `GenerateAssemblyInfo` is `false`

If the generation of assembly info attributes is disabled, manually add the UserSecretsIdAttribute in `AssemblyInfo.cs`. For example:

C#

```
[assembly: UserSecretsId("your_user_secrets_id")]
```

When manually adding the `UserSecretsId` attribute to `AssemblyInfo.cs`, the `UserSecretsId` value must match the value in the project file.

Set a secret

Define an app secret consisting of a key and its value. The secret is associated with the project's `UserSecretsId` value. For example, run the following command from the directory in which the project file exists:

.NET CLI

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345"
```

In the preceding example, the colon denotes that `Movies` is an object literal with a `ServiceApiKey` property.

The Secret Manager tool can be used from other directories too. Use the `--project` option to supply the file system path at which the project file exists. For example:

.NET CLI

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345" --project
"C:\apps\WebApp1\src\WebApp1"
```

JSON structure flattening in Visual Studio

Visual Studio's **Manage User Secrets** gesture opens a `secrets.json` file in the text editor. Replace the contents of `secrets.json` with the key-value pairs to be stored. For example:

JSON

```
{
  "Movies": {
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
    "ServiceApiKey": "12345"
  }
}
```

The JSON structure is flattened after modifications via `dotnet user-secrets remove` or `dotnet user-secrets set`. For example, running `dotnet user-secrets remove "Movies:ConnectionString"` collapses the `Movies` object literal. The modified file resembles the following JSON:

JSON

```
{
  "Movies:ServiceApiKey": "12345"
}
```

Set multiple secrets

A batch of secrets can be set by piping JSON to the `set` command. In the following example, the `input.json` file's contents are piped to the `set` command.

Windows

Open a command shell, and execute the following command:

.NET CLI

```
type .\input.json | dotnet user-secrets set
```

Access a secret

To access a secret, complete the following steps:

1. Register the user secrets configuration source
2. Read the secret via the Configuration API

Register the user secrets configuration source

The user secrets configuration provider registers the appropriate configuration source with the .NET Configuration API.

ASP.NET Core web apps created with dotnet new or Visual Studio generate the following code:

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults. The initialized `WebApplicationBuilder` (`builder`) provides default configuration and calls AddUserSecrets when the EnvironmentName is Development:

Read the secret via the Configuration API

Consider the following examples of reading the `Movies:ServiceApiKey` key:

Program.cs file:

```
C#

var builder = WebApplication.CreateBuilder(args);
var movieApiKey = builder.Configuration["Movies:ServiceApiKey"];

var app = builder.Build();

app.MapGet("/", () => movieApiKey);

app.Run();
```

Razor Pages page model:

```
C#

public class IndexModel : PageModel
{
    private readonly IConfiguration _config;

    public IndexModel(IConfiguration config)
    {
```

```
        _config = config;
    }

    public void OnGet()
    {
        var moviesApiKey = _config["Movies:ServiceApiKey"];

        // call Movies service with the API key
    }
}
```

For more information, see Configuration in ASP.NET Core.

Map secrets to a POCO

Mapping an entire object literal to a POCO (a simple .NET class with properties) is useful for aggregating related properties.

Assume the app's `secrets.json` file contains the following two secrets:

JSON

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

To map the preceding secrets to a POCO, use the .NET Configuration API's object graph binding feature. The following code binds to a custom `MovieSettings` POCO and accesses the `ServiceApiKey` property value:

C#

```
var moviesConfig =
    Configuration.GetSection("Movies").Get<MovieSettings>();
_moviesApiKey = moviesConfig.ServiceApiKey;
```

The `Movies:ConnectionString` and `Movies:ServiceApiKey` secrets are mapped to the respective properties in `MovieSettings`:

C#

```
public class MovieSettings
{
    public string ConnectionString { get; set; }
```

```
    public string ServiceApiKey { get; set; }
}
```

String replacement with secrets

Storing passwords in plain text is insecure. Never store secrets in a configuration file such as `appsettings.json`, which might get checked in to a source code repository.

For example, a database connection string stored in `appsettings.json` should not include a password. Instead, store the password as a secret, and include the password in the connection string at runtime. For example:

.NET CLI

```
dotnet user-secrets set "DbPassword" "`<secret value>`"
```

Replace the `<secret value>` placeholder in the preceding example with the password value. Set the secret's value on a SqlConnectionStringBuilder object's Password property to include it as the password value in the connection string:

C#

```
using System.Data.SqlClient;

var builder = WebApplication.CreateBuilder(args);

var conStrBuilder = new SqlConnectionStringBuilder(
        builder.Configuration.GetConnectionString("Movies"));
conStrBuilder.Password = builder.Configuration["DbPassword"];
var connection = conStrBuilder.ConnectionString;

var app = builder.Build();

app.MapGet("/", () => connection);

app.Run();
```

List the secrets

Assume the app's `secrets.json` file contains the following two secrets:

JSON

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the project file exists:

.NET CLI

```
dotnet user-secrets list
```

The following output appears:

Console

```
Movies:ConnectionString = Server=(localdb)\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true
Movies:ServiceApiKey = 12345
```

In the preceding example, a colon in the key names denotes the object hierarchy within `secrets.json`.

Remove a single secret

Assume the app's `secrets.json` file contains the following two secrets:

JSON

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the project file exists:

.NET CLI

```
dotnet user-secrets remove "Movies:ConnectionString"
```

The app's `secrets.json` file was modified to remove the key-value pair associated with the `Movies:ConnectionString` key:

JSON

```
{
  "Movies": {
    "ServiceApiKey": "12345"
  }
}
```

`dotnet user-secrets list` displays the following message:

Console

```
Movies:ServiceApiKey = 12345
```

Remove all secrets

Assume the app's `secrets.json` file contains the following two secrets:

JSON

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the project file exists:

.NET CLI

```
dotnet user-secrets clear
```

All user secrets for the app have been deleted from the `secrets.json` file:

JSON

```
{}
```

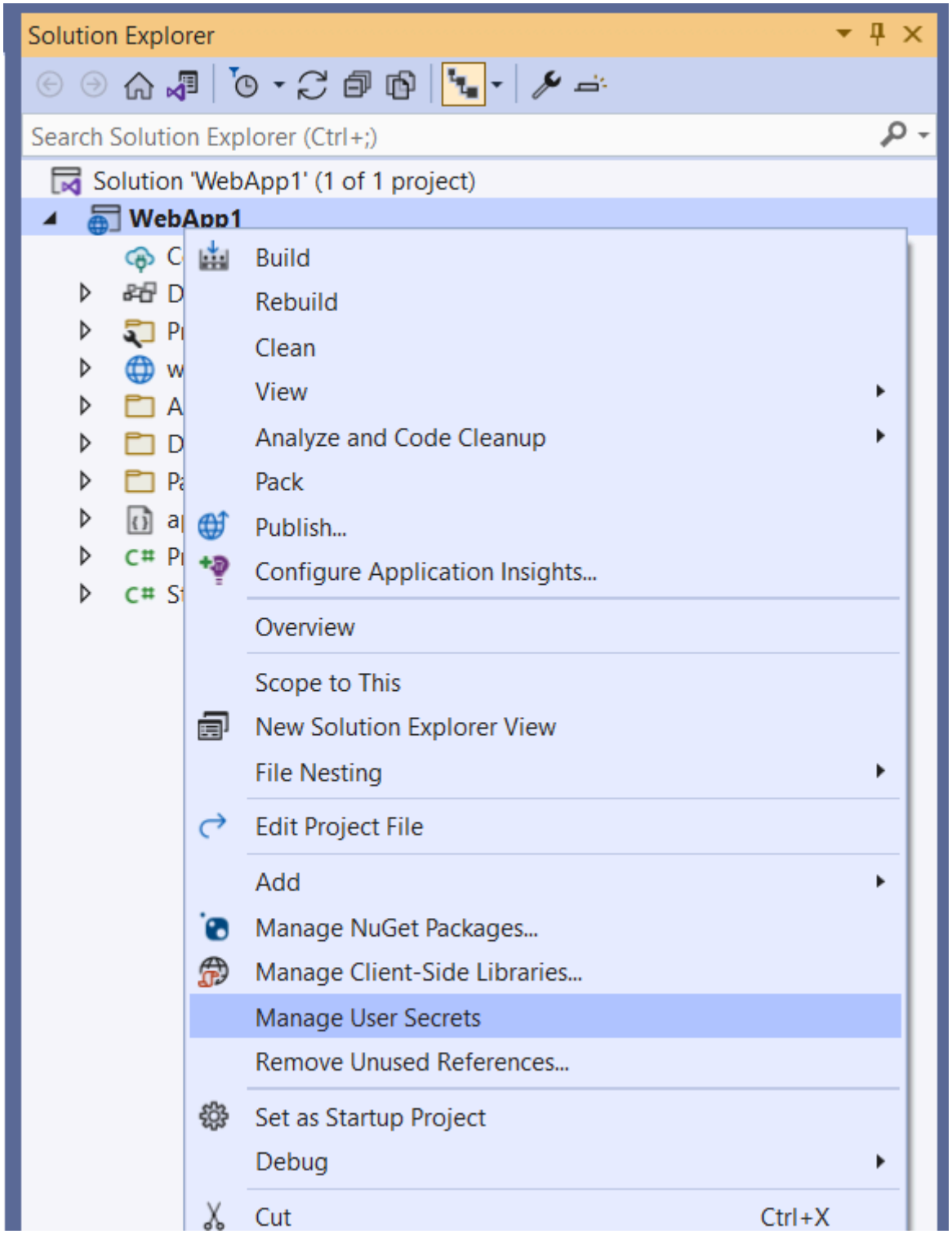
Running `dotnet user-secrets list` displays the following message:

Console

```
No secrets configured for this application.
```

Manage user secrets with Visual Studio

To manage user secrets in Visual Studio, right click the project in solution explorer and select **Manage User Secrets**:



Migrating User Secrets from ASP.NET Framework to ASP.NET Core

See this GitHub issue.

User secrets in non-web applications

Projects that target `Microsoft.NET.Sdk.Web` automatically include support for user secrets. For projects that target `Microsoft.NET.Sdk`, such as console applications, install the configuration extension and user secrets NuGet packages explicitly.

Using PowerShell:

PowerShell

```
Install-Package Microsoft.Extensions.Configuration
Install-Package Microsoft.Extensions.Configuration.UserSecrets
```

Using the .NET CLI:

.NET CLI

```
dotnet add package Microsoft.Extensions.Configuration
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
```

Once the packages are installed, initialize the project and set secrets the same way as for a web app. The following example shows a console application that retrieves the value of a secret that was set with the key "AppSecret":

C#

```
using Microsoft.Extensions.Configuration;

namespace ConsoleApp;

class Program
{
    static void Main(string[] args)
    {
        IConfigurationRoot config = new ConfigurationBuilder()
            .AddUserSecrets<Program>()
            .Build();

        Console.WriteLine(config["AppSecret"]);
    }
}
```

Additional resources

- See this issue and this issue for information on accessing user secrets from IIS.

- Configuration in ASP.NET Core
- Azure Key Vault configuration provider in ASP.NET Core

Manage JSON Web Tokens in development with dotnet user-jwts

Article • 09/27/2024

By Rick Anderson

The `dotnet user-jwts` command line tool can create and manage app specific local JSON Web Tokens (JWTs).

Synopsis

.NET CLI

```
dotnet user-jwts [<PROJECT>] [command]
dotnet user-jwts [command] -h|--help
```

Description

Creates and manages project specific local JSON Web Tokens.

Arguments

`PROJECT | SOLUTION`

The MSBuild project to apply a command on. If a project is not specified, MSBuild searches the current working directory for a file that has a file extension that ends in *proj* and uses that file.

Commands

Expand table

Command	Description
clear	Delete all issued JWTs for a project.
create	Issue a new JSON Web Token.
remove	Delete a given JWT.

Command	Description
key	Display or reset the signing key used to issue JWTs.
list	Lists the JWTs issued for the project.
print	Display the details of a given JWT.

Create

Usage: `dotnet user-jwts create [options]`

Expand table

Option	Description
-p | --project	The path of the project to operate on. Defaults to the project in the current directory.
--scheme	The scheme name to use for the generated token. Defaults to 'Bearer'.
-n | --name	The name of the user to create the JWT for. Defaults to the current environment user.
--audience	The audiences to create the JWT for. Defaults to the URLs configured in the project's launchSettings.json.
--issuer	The issuer of the JWT. Defaults to 'dotnet-user-jwts'.
--scope	A scope claim to add to the JWT. Specify once for each scope.
--role	A role claim to add to the JWT. Specify once for each role.
--claim	Claims to add to the JWT. Specify once for each claim in the format "name=value".
--not-before	The UTC date & time the JWT should not be valid before in the format 'yyyy-MM-dd [[HH:mm[[:ss]]]]'. Defaults to the date & time the JWT is created.
--expires-on	The UTC date & time the JWT should expire in the format 'yyyy-MM-dd [[[[HH:mm]]:ss]]'. Defaults to 6 months after the --not-before date. Do not use this option in conjunction with the --valid-for option.
--valid-for	The period the JWT should expire after. Specify using a number followed by duration type like 'd' for days, 'h' for hours, 'm' for minutes, and 's' for seconds, for example 365d'. Do not use this option in conjunction with the --expires-on option.
-o | --output	The format to use for displaying output from the command. Can be one of 'default', 'token', or 'json'.
-h | --help	Show help information

Examples

Run the following commands to create an empty web project and add the Microsoft.AspNetCore.Authentication.JwtBearer NuGet package:

.NET CLI

```
dotnet new web -o MyJWT
cd MyJWT
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Replace the contents of `Program.cs` with the following code:

C#

```
using System.Security.Claims;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var app = builder.Build();

app.UseAuthorization();

app.MapGet("/", () => "Hello, World!");
app.MapGet("/secret", (ClaimsPrincipal user) => $"Hello
{user.Identity?.Name}. My secret")
    .RequireAuthorization();

app.Run();
```

In the preceding code, a GET request to `/secret` returns an `401 Unauthorized` error. A production app might get the JWT from a Security token service (STS), perhaps in response to logging in via a set of credentials. For the purpose of working with the API during local development, the `dotnet user-jwts` command line tool can be used to create and manage app-specific local JWTs.

The `user-jwts` tool is similar in concept to the user-secrets tool, it can be used to manage values for the app that are only valid for the developer on the local machine. In fact, the user-jwts tool utilizes the `user-secrets` infrastructure to manage the key that the JWTs are signed with, ensuring it's stored safely in the user profile.

The `user-jwts` tool hides implementation details, such as where and how the values are stored. The tool can be used without knowing the implementation details. The values

are stored in a JSON file in the local machine's user profile folder:

Windows

File system path:

`%APPDATA%\Microsoft\UserSecrets\<secrets_GUID>\user-jwts.json`

Create a JWT

The following command creates a local JWT:

.NET CLI

```
dotnet user-jwts create
```

The preceding command creates a JWT and updates the project's `appsettings.Development.json` file with JSON similar to the following:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Authentication": {
    "Schemes": {
      "Bearer": {
        "ValidAudiences": [
          "http://localhost:8401",
          "https://localhost:44308",
          "http://localhost:5182",
          "https://localhost:7076"
        ],
        "ValidIssuer": "dotnet-user-jwts"
      }
    }
  }
}
```

Copy the JWT and the `ID` created in the preceding command. Use a tool like Curl to test `/secret`:

.NET CLI

```
curl -i -H "Authorization: Bearer {token}" https://localhost:{port}/secret
```

Where `{token}` is the previously generated JWT.

Display JWT security information

The following command displays the JWT security information, including expiration, scopes, roles, token header and payload, and the compact token:

.NET CLI

```
dotnet user-jwts print {ID} --show-all
```

Create a token for a specific user and scope

See Create in this topic for supported create options.

The following command creates a JWT for a user named `MyTestUser`:

.NET CLI

```
dotnet user-jwts create --name MyTestUser --scope "myapi:secrets"
```

The preceding command has output similar to the following:

.NET CLI

```
New JWT saved with ID '43e0b748'.
Name: MyTestUser
Scopes: myapi:secrets

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.{Remaining token deleted}
```

The preceding token can be used to test the `/secret2` endpoint in the following code:

C#

```
using System.Security.Claims;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
```

```
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var app = builder.Build();

app.MapGet("/", () => "Hello, World!");
app.MapGet("/secret", (ClaimsPrincipal user) => $"Hello
{user.Identity?.Name}. My secret")
    .RequireAuthorization();
app.MapGet("/secret2", () => "This is a different secret!")
    .RequireAuthorization(p => p.RequireClaim("scope", "myapi:secrets"));

app.Run();
```

Azure Key Vault configuration provider in ASP.NET Core

Article • 10/30/2024

This article explains how to use the Azure Key Vault configuration provider to load app configuration values from Azure Key Vault secrets. Azure Key Vault is a cloud-based service that helps safeguard cryptographic keys and secrets used by apps and services. Common scenarios for using Azure Key Vault with ASP.NET Core apps include:

- Controlling access to sensitive configuration data.
- Meeting the requirement for FIPS 140-2 Level 2 validated Hardware Security Modules (HSMs) when storing configuration data.

Packages

Add package references for the following packages:

- Azure.Extensions.AspNetCore.Configuration.Secrets
- Azure.Identity

Sample app

The sample app runs in either of two modes determined by the `#define` preprocessor directive at the top of `Program.cs`:

- `Certificate`: Demonstrates using an Azure Key Vault Client ID and X.509 certificate to access secrets stored in Azure Key Vault. This sample can be run from any location, whether deployed to Azure App Service or any host that can serve an ASP.NET Core app.
- `Managed`: Demonstrates how to use Managed identities for Azure resources. The managed identity authenticates the app to Azure Key Vault with Managed identities for Azure resources without storing credentials in the app's code or configuration. The `Managed` version of the sample must be deployed to Azure. Follow the guidance in the Use the managed identities for Azure resources section.

For more information configuring a sample app using preprocessor directives (`#define`), see Overview of ASP.NET Core.

View or download sample code (how to download)

Secret storage in the Development environment

Set secrets locally using Secret Manager. When the sample app runs on the local machine in the Development environment, secrets are loaded from the local user secrets store.

Secret Manager requires a `<UserSecretsId>` property in the app's project file. Set the property value (`{GUID}`) to any unique GUID:

XML

```
<PropertyGroup>
  <UserSecretsId>{GUID}</UserSecretsId>
</PropertyGroup>
```

Secrets are created as name-value pairs. Hierarchical values (configuration sections) use a `:` (colon) as a separator in ASP.NET Core configuration key names.

Secret Manager is used from a command shell opened to the project's content root, where `{SECRET NAME}` is the name and `{SECRET VALUE}` is the value:

.NET CLI

```
dotnet user-secrets set "{SECRET NAME}" "{SECRET VALUE}"
```

Execute the following commands in a command shell from the project's content root to set the secrets for the sample app:

.NET CLI

```
dotnet user-secrets set "SecretName" "secret_value_1_dev"
dotnet user-secrets set "Section:SecretName" "secret_value_2_dev"
```

When these secrets are stored in Azure Key Vault in the Secret storage in the Production environment with Azure Key Vault section, the `_dev` suffix is changed to `_prod`. The suffix provides a visual cue in the app's output indicating the source of the configuration values.

Secret storage in the Production environment with Azure Key Vault

Complete the following steps to create an Azure Key Vault and store the sample app's secrets in it. For more information, see Quickstart: Set and retrieve a secret from Azure Key Vault using Azure CLI.

1. Open Azure Cloud Shell using any one of the following methods in the Azure portal:

   - Select **Try It** in the upper-right corner of a code block. Use the search string "Azure CLI" in the text box.
   - Open Cloud Shell in your browser with the **Launch Cloud Shell** button.
   - Select the **Cloud Shell** button on the menu in the upper-right corner of the Azure portal.

   For more information, see Azure CLI and Overview of Azure Cloud Shell.

2. If you aren't already authenticated, sign in with the `az login` command.

3. Create a resource group with the following command, where `{RESOURCE GROUP NAME}` is the new resource group's name and `{LOCATION}` is the Azure region:

   Azure CLI

   ```
   az group create --name "{RESOURCE GROUP NAME}" --location {LOCATION}
   ```

4. Create a Key Vault in the resource group with the following command, where `{KEY VAULT NAME}` is the new vault's name and `{LOCATION}` is the Azure region:

   Azure CLI

   ```
   az keyvault create --name {KEY VAULT NAME} --resource-group "{RESOURCE GROUP NAME}" --location {LOCATION}
   ```

5. Create secrets in the vault as name-value pairs.

   Azure Key Vault secret names are limited to alphanumeric characters and dashes. Hierarchical values (configuration sections) use `--` (two dashes) as a delimiter, as colons aren't allowed in Key Vault secret names. Colons delimit a section from a subkey in ASP.NET Core configuration. The two-dash sequence is replaced with a colon when the secrets are loaded into the app's configuration.

   The following secrets are for use with the sample app. The values include a `_prod` suffix to distinguish them from the `_dev` suffix values loaded in the Development

environment from Secret Manager. Replace `{KEY VAULT NAME}` with the name of the Key Vault you created in the prior step:

Azure CLI

```
az keyvault secret set --vault-name {KEY VAULT NAME} --name
"SecretName" --value "secret_value_1_prod"
az keyvault secret set --vault-name {KEY VAULT NAME} --name "Section--
SecretName" --value "secret_value_2_prod"
```

Use Application ID and X.509 certificate for non-Azure-hosted apps

Configure Azure Key Vault and the app to use an Microsoft Entra ID Application ID and X.509 certificate to authenticate to a vault **when the app is hosted outside of Azure**. For more information, see About keys, secrets, and certificates.

> **Note**
>
> Although using an Application ID and X.509 certificate is supported for apps hosted in Azure, it's not recommended. Instead, use Managed identities for Azure resources when hosting an app in Azure. Managed identities don't require storing a certificate in the app or in the development environment.

The sample app uses an Application ID and X.509 certificate when the `#define` preprocessor directive at the top of `Program.cs` is set to `Certificate`.

1. Create a PKCS#12 archive (*.pfx*) certificate. Options for creating certificates include New-SelfSignedCertificate on Windows and OpenSSL.
2. Install the certificate into the current user's personal certificate store. Marking the key as exportable is optional. Note the certificate's thumbprint, which is used later in this process.
3. Export the PKCS#12 archive (*.pfx*) certificate as a DER-encoded certificate (*.cer*).
4. Register the app with Microsoft Entra ID (**App registrations**).
5. Upload the DER-encoded certificate (*.cer*) to Microsoft Entra ID:
   a. Select the app in Microsoft Entra ID.
   b. Navigate to **Certificates & secrets**.
   c. Select **Upload certificate** to upload the certificate, which contains the public key. A *.cer*, *.pem*, or *.crt* certificate is acceptable.

6. Store the Key Vault name, Application ID, and certificate thumbprint in the app's `appsettings.json` file.
7. Navigate to **Key Vaults** in the Azure portal.
8. Select the Key Vault you created in the Secret storage in the Production environment with Azure Key Vault section.
9. Select **Access policies**.
10. Select **Add Access Policy**.
11. Open **Secret permissions** and provide the app with **Get** and **List** permissions.
12. Select **Select principal** and select the registered app by name. Select the **Select** button.
13. Select **OK**.
14. Select **Save**.
15. Deploy the app.

The `Certificate` sample app obtains its configuration values from IConfigurationRoot with the same name as the secret name:

- Non-hierarchical values: The value for `SecretName` is obtained with `config["SecretName"]`.
- Hierarchical values (sections): Use `:` (colon) notation or the GetSection method. Use either of these approaches to obtain the configuration value:
  - `config["Section:SecretName"]`
  - `config.GetSection("Section")["SecretName"]`

The X.509 certificate is managed by the OS. The app calls AddAzureKeyVault with values supplied by the `appsettings.json` file:

C#

```
using System.Security.Cryptography.X509Certificates;
using Azure.Identity;

var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsProduction())
{
    using var x509Store = new X509Store(StoreLocation.CurrentUser);

    x509Store.Open(OpenFlags.ReadOnly);

    var x509Certificate = x509Store.Certificates
        .Find(
            X509FindType.FindByThumbprint,
            builder.Configuration["AzureADCertThumbprint"],
            validOnly: false)
```

```
        .OfType<X509Certificate2>()
        .Single();

    builder.Configuration.AddAzureKeyVault(
        new
Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"),
        new ClientCertificateCredential(
            builder.Configuration["AzureADDirectoryId"],
            builder.Configuration["AzureADApplicationId"],
            x509Certificate));
}

var app = builder.Build();
```

Example values:

- Key Vault name: `contosovault`
- Application ID: `00001111-aaaa-2222-bbbb-3333cccc4444`
- Certificate thumbprint: `fe14593dd66b2406c5269d742d04b6e1ab03adb1`

`appsettings.json`:

JSON

```
{
  "KeyVaultName": "Key Vault Name",
  "AzureADApplicationId": "Azure AD Application ID",
  "AzureADCertThumbprint": "Azure AD Certificate Thumbprint",
  "AzureADDirectoryId": "Azure AD Directory ID"
}
```

When you run the app, a webpage shows the loaded secret values. In the Development environment, secret values load with the `_dev` suffix. In the Production environment, the values load with the `_prod` suffix.

Use managed identities for Azure resources

An app deployed to Azure can take advantage of Managed identities for Azure resources. A managed identity allows the app to authenticate with Azure Key Vault using Microsoft Entra ID authentication without storing credentials in the app's code or configuration.

The sample app uses a system-assigned managed identity when the `#define` preprocessor directive at the top of `Program.cs` is set to `Managed`. To create a managed identity for an Azure App Service app, see How to use managed identities for App

Service and Azure Functions. Once the managed identity has been created, note the app's Object ID shown in the Azure portal on the **Identity** panel of the App Service.

Enter the vault name into the app's `appsettings.json` file. The sample app doesn't require an Application ID and Password (Client Secret) when set to the `Managed` version, so you can ignore those configuration entries. The app is deployed to Azure, and Azure authenticates the app to access Azure Key Vault only using the vault name stored in the `appsettings.json` file.

Deploy the sample app to Azure App Service.

Using Azure CLI and the app's Object ID, provide the app with `list` and `get` permissions to access the vault:

Azure CLI

```
az keyvault set-policy --name {KEY VAULT NAME} --object-id {OBJECT ID} --
secret-permissions get list
```

Restart the app using Azure CLI, PowerShell, or the Azure portal.

The sample app creates an instance of the DefaultAzureCredential class. The credential attempts to obtain an access token from environment for Azure resources:

C#

```
using Azure.Identity;

var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsProduction())
{
    builder.Configuration.AddAzureKeyVault(
        new
Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"),
        new DefaultAzureCredential());
}
```

Key Vault name example value: `contosovault`

`appsettings.json`:

JSON

```
{
  "KeyVaultName": "Key Vault Name"
```

```
}
```

For apps that use a user-assigned managed identity, configure the managed identity's Client ID using one of the following approaches:

1. Set the `AZURE_CLIENT_ID` environment variable.
2. Set the DefaultAzureCredentialOptions.ManagedIdentityClientId property when calling `AddAzureKeyVault`:

   C#

   ```
   builder.Configuration.AddAzureKeyVault(
       new
   Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"
   ),
       new DefaultAzureCredential(new DefaultAzureCredentialOptions
       {
           ManagedIdentityClientId =
   builder.Configuration["AzureADManagedIdentityClientId"]
       }));
   ```

When you run the app, a webpage shows the loaded secret values. In the Development environment, secret values have the `_dev` suffix because they're provided by Secret Manager. In the Production environment, the values load with the `_prod` suffix because they're provided by Azure Key Vault.

If you receive an `Access denied` error, confirm that the app is registered with Microsoft Entra ID and provided access to the vault. Confirm that you've restarted the service in Azure.

For information on using the provider with a managed identity and Azure Pipelines, see Create an Azure Resource Manager service connection to a VM with a managed service identity.

Configuration options

`AddAzureKeyVault` can accept an AzureKeyVaultConfigurationOptions object:

C#

```
// using Azure.Extensions.AspNetCore.Configuration.Secrets;

builder.Configuration.AddAzureKeyVault(
    new
```

```
Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"),
    new DefaultAzureCredential(),
    new AzureKeyVaultConfigurationOptions
    {
        // ...
    });
```

The `AzureKeyVaultConfigurationOptions` object contains the following properties:

Expand table

Property	Description
Manager	KeyVaultSecretManager instance used to control secret loading.
ReloadInterval	`TimeSpan` to wait between attempts at polling the vault for changes. The default value is `null` (configuration isn't reloaded).

Use a key name prefix

`AddAzureKeyVault` provides an overload that accepts an implementation of KeyVaultSecretManager, which allows you to control how Key Vault secrets are converted into configuration keys. For example, you can implement the interface to load secret values based on a prefix value you provide at app startup. This technique allows you, for example, to load secrets based on the version of the app.

> ⚠ **Warning**
>
> Don't use prefixes on Key Vault secrets to:
>
> - Place secrets for multiple apps into the same vault.
> - Place environmental secrets (for example, *development* versus *production* secrets) into the same vault.
>
> Different apps and development/production environments should use separate Key Vaults to isolate app environments for the highest level of security.

In the following example, a secret is established in Key Vault (and using Secret Manager for the Development environment) for `5000-AppSecret` (periods aren't allowed in Key Vault secret names). This secret represents an app secret for version 5.0.0.0 of the app. For another version of the app, 5.1.0.0, a secret is added to the vault (and using Secret

Manager) for `5100-AppSecret`. Each app version loads its versioned secret value into its configuration as `AppSecret`, removing the version as it loads the secret.

`AddAzureKeyVault` is called with a custom `KeyVaultSecretManager` implementation:

C#

```
// using Azure.Extensions.AspNetCore.Configuration.Secrets;

builder.Configuration.AddAzureKeyVault(
    new
Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"),
    new DefaultAzureCredential(),
    new SamplePrefixKeyVaultSecretManager("5000"));
```

The implementation reacts to the version prefixes of secrets to load the proper secret into configuration:

- `Load` loads a secret when its name starts with the prefix. Other secrets aren't loaded.
- `GetKey`:
  - Removes the prefix from the secret name.
  - Replaces two dashes in any name with the `KeyDelimiter`, which is the delimiter used in configuration (usually a colon). Azure Key Vault doesn't allow a colon in secret names.

C#

```
public class SamplePrefixKeyVaultSecretManager : KeyVaultSecretManager
{
    private readonly string _prefix;

    public SamplePrefixKeyVaultSecretManager(string prefix)
        => _prefix = $"{prefix}-";

    public override bool Load(SecretProperties properties)
        => properties.Name.StartsWith(_prefix);

    public override string GetKey(KeyVaultSecret secret)
        => secret.Name[_prefix.Length..].Replace("--",
ConfigurationPath.KeyDelimiter);
}
```

The `Load` method is called by a provider algorithm that iterates through the vault secrets to find the version-prefixed secrets. When a version prefix is found with `Load`, the algorithm uses the `GetKey` method to return the configuration name of the secret

name. It removes the version prefix from the secret's name. The rest of the secret name is returned for loading into the app's configuration name-value pairs.

When this approach is implemented:

1. The app's version specified in the app's project file. In the following example, the app's version is set to `5.0.0.0`:

   XML

   ```xml
   <PropertyGroup>
     <Version>5.0.0.0</Version>
   </PropertyGroup>
   ```

2. Confirm that a `<UserSecretsId>` property is present in the app's project file, where `{GUID}` is a user-supplied GUID:

   XML

   ```xml
   <PropertyGroup>
     <UserSecretsId>{GUID}</UserSecretsId>
   </PropertyGroup>
   ```

   Save the following secrets locally with Secret Manager:

   .NET CLI

   ```
   dotnet user-secrets set "5000-AppSecret" "5.0.0.0_secret_value_dev"
   dotnet user-secrets set "5100-AppSecret" "5.1.0.0_secret_value_dev"
   ```

3. Secrets are saved in Azure Key Vault using the following Azure CLI commands:

   Azure CLI

   ```
   az keyvault secret set --vault-name {KEY VAULT NAME} --name "5000-
   AppSecret" --value "5.0.0.0_secret_value_prod"
   az keyvault secret set --vault-name {KEY VAULT NAME} --name "5100-
   AppSecret" --value "5.1.0.0_secret_value_prod"
   ```

4. When the app is run, the Key Vault secrets are loaded. The string secret for `5000-AppSecret` is matched to the app's version specified in the app's project file (`5.0.0.0`).

5. The version, `5000` (with the dash), is stripped from the key name. Throughout the app, reading configuration with the key `AppSecret` loads the secret value.

6. If the app's version is changed in the project file to `5.1.0.0` and the app is run again, the secret value returned is `5.1.0.0_secret_value_dev` in the Development environment and `5.1.0.0_secret_value_prod` in Production.

> **Note**
>
> You can also provide your own SecretClient implementation to `AddAzureKeyVault`. A custom client permits sharing a single instance of the client across the app.

Bind an array to a class

The provider can read configuration values into an array for binding to a POCO array.

When reading from a configuration source that allows keys to contain colon (`:`) separators, a numeric key segment is used to distinguish the keys that make up an array (`:0:`, `:1:`, ... `:{n}:`). For more information, see Configuration: Bind an array to a class.

Azure Key Vault keys can't use a colon as a separator. The approach described in this article uses double dashes (`--`) as a separator for hierarchical values (sections). Array keys are stored in Azure Key Vault with double dashes and numeric key segments (`--0--`, `--1--`, ... `--{n}--`).

Examine the following Serilog logging provider configuration provided by a JSON file. There are two object literals defined in the `WriteTo` array that reflect two Serilog *sinks*, which describe destinations for logging output:

JSON

```
"Serilog": {
  "WriteTo": [
    {
      "Name": "AzureTableStorage",
      "Args": {
        "storageTableName": "logs",
        "connectionString": "DefaultEnd...ountKey=Eby8...GMGw=="
      }
    },
    {
      "Name": "AzureDocumentDB",
      "Args": {
        "endpointUrl": "https://contoso.documents.azure.com:443",
        "authorizationKey": "Eby8...GMGw=="
      }
    }
```

```
  ]
}
```

The configuration shown in the preceding JSON file is stored in Azure Key Vault using double dash (`--`) notation and numeric segments:

Expand table

Key	Value
`Serilog--WriteTo--0--Name`	`AzureTableStorage`
`Serilog--WriteTo--0--Args--storageTableName`	`logs`
`Serilog--WriteTo--0--Args--connectionString`	`DefaultEnd...ountKey=Eby8...GMGw==`
`Serilog--WriteTo--1--Name`	`AzureDocumentDB`
`Serilog--WriteTo--1--Args--endpointUrl`	`https://contoso.documents.azure.com:443`
`Serilog--WriteTo--1--Args--authorizationKey`	`Eby8...GMGw==`

Reload secrets

By default, secrets are cached by the configuration provider for the app's lifetime. Secrets that have been subsequently disabled or updated in the vault are ignored by the app.

To reload secrets, call IConfigurationRoot.Reload:

C#

```
config.Reload();
```

To reload secrets periodically, at a specified interval, set the AzureKeyVaultConfigurationOptions.ReloadInterval property. For more information, see Configuration options.

Disabled and expired secrets

Expired secrets are included by default in the configuration provider. To exclude values for these secrets in app configuration, update the expired secret or provide the configuration using a custom configuration provider:

C#

```
class SampleKeyVaultSecretManager : KeyVaultSecretManager
{
  public override bool Load(SecretProperties properties) =>
    properties.ExpiresOn.HasValue &&
    properties.ExpiresOn.Value > DateTimeOffset.Now;
}
```

Pass this custom `KeyVaultSecretManager` to `AddAzureKeyVault`:

C#

```
// using Azure.Extensions.AspNetCore.Configuration.Secrets;

builder.Configuration.AddAzureKeyVault(
    new
Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"),
    new DefaultAzureCredential(),
    new SampleKeyVaultSecretManager());
```

Disabled secrets cannot be retrieved from Key Vault and are never included.

Troubleshoot

When the app fails to load configuration using the provider, an error message is written to the ASP.NET Core Logging infrastructure. The following conditions will prevent configuration from loading:

- The app or certificate isn't configured correctly in Microsoft Entra ID.
- The vault doesn't exist in Azure Key Vault.
- The app isn't authorized to access the vault.
- The access policy doesn't include `Get` and `List` permissions.
- In the vault, the configuration data (name-value pair) is incorrectly named, missing, or disabled.
- The app has the wrong Key Vault name (`KeyVaultName`), Microsoft Entra ID Application ID (`AzureADApplicationId`), or Microsoft Entra ID certificate thumbprint (`AzureADCertThumbprint`), or Microsoft Entra ID Directory ID (`AzureADDirectoryId`).
- When adding the Key Vault access policy for the app, the policy was created, but the **Save** button wasn't selected in the **Access policies** UI.

Additional resources

- View or download sample code (how to download)
- Configuration in ASP.NET Core
- Microsoft Azure: Key Vault Documentation
- How to generate and transfer HSM-protected keys for Azure Key Vault
- Quickstart: Set and retrieve a secret from Azure Key Vault by using a .NET web app
- Tutorial: How to use Azure Key Vault with Azure Windows Virtual Machine in .NET

Enforce HTTPS in ASP.NET Core

Article • 10/25/2024

By David Galvan and Rick Anderson

this article shows how to:

- Require HTTPS for all requests.
- Redirect all HTTP requests to HTTPS.

No API can prevent a client from sending sensitive data on the first request.

⚠ **Warning**

API projects

Do **not** use RequireHttpsAttribute on Web APIs that receive sensitive information. `RequireHttpsAttribute` uses HTTP status codes to redirect browsers from HTTP to HTTPS. API clients may not understand or obey redirects from HTTP to HTTPS. Such clients may send information over HTTP. Web APIs should either:

- Not listen on HTTP.
- Close the connection with status code 400 (Bad Request) and not serve the request.

To disable HTTP redirection in an API, set the `ASPNETCORE_URLS` environment variable or use the `--urls` command line flag. For more information, see Use multiple environments in ASP.NET Core and 8 ways to set the URLs for an ASP.NET Core app by Andrew Lock.

HSTS and API projects

The default API projects don't include HSTS because HSTS is generally a browser only instruction. Other callers, such as phone or desktop apps, do **not** obey the instruction. Even within browsers, a single authenticated call to an API over HTTP has risks on insecure networks. The secure approach is to configure API projects to only listen to and respond over HTTPS.

HTTP redirection to HTTPS causes ERR_INVALID_REDIRECT on the CORS preflight request

Requests to an endpoint using HTTP that are redirected to HTTPS by UseHttpsRedirection fail with `ERR_INVALID_REDIRECT` on the CORS preflight request.

API projects can reject HTTP requests rather than use `UseHttpsRedirection` to redirect requests to HTTPS.

Require HTTPS

We recommend that production ASP.NET Core web apps use:

- HTTPS Redirection Middleware (UseHttpsRedirection) to redirect HTTP requests to HTTPS.
- HSTS Middleware (UseHsts) to send HTTP Strict Transport Security Protocol (HSTS) headers to clients.

> **Note**
>
> Apps deployed in a reverse proxy configuration allow the proxy to handle connection security (HTTPS). If the proxy also handles HTTPS redirection, there's no need to use HTTPS Redirection Middleware. If the proxy server also handles writing HSTS headers (for example, native HSTS support in IIS 10.0 (1709) or later), HSTS Middleware isn't required by the app. For more information, see Opt-out of HTTPS/HSTS on project creation.

UseHttpsRedirection

The following code calls UseHttpsRedirection in the `Program.cs` file:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
```

```
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding highlighted code:

- Uses the default HttpsRedirectionOptions.RedirectStatusCode (Status307TemporaryRedirect).
- Uses the default HttpsRedirectionOptions.HttpsPort (null) unless overridden by the `ASPNETCORE_HTTPS_PORT` environment variable or IServerAddressesFeature.

We recommend using temporary redirects rather than permanent redirects. Link caching can cause unstable behavior in development environments. If you prefer to send a permanent redirect status code when the app is in a non-Development environment, see the Configure permanent redirects in production section. We recommend using HSTS to signal to clients that only secure resource requests should be sent to the app (only in production).

Port configuration

A port must be available for the middleware to redirect an insecure request to HTTPS. If no port is available:

- Redirection to HTTPS doesn't occur.
- The middleware logs the warning "Failed to determine the https port for redirect."

Specify the HTTPS port using any of the following approaches:

- Set HttpsRedirectionOptions.HttpsPort.
- Set the `https_port` host setting:
  - In host configuration.
  - By setting the `ASPNETCORE_HTTPS_PORT` environment variable.
  - By adding a top-level entry in `appsettings.json`:

JSON

```
{
  "https_port": 443,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

- Indicate a port with the secure scheme using the ASPNETCORE_URLS environment variable. The environment variable configures the server. The middleware indirectly discovers the HTTPS port via IServerAddressesFeature. This approach doesn't work in reverse proxy deployments.
- The ASP.NET Core web templates set an HTTPS URL in `Properties/launchsettings.json` for both Kestrel and IIS Express. `launchsettings.json` is only used on the local machine.
- Configure an HTTPS URL endpoint for a public-facing edge deployment of Kestrel server or HTTP.sys server. Only **one HTTPS port** is used by the app. The middleware discovers the port via IServerAddressesFeature.

> **Note**
>
> When an app is run in a reverse proxy configuration, IServerAddressesFeature isn't available. Set the port using one of the other approaches described in this section.

Edge deployments

When Kestrel or HTTP.sys is used as a public-facing edge server, Kestrel or HTTP.sys must be configured to listen on both:

- The secure port where the client is redirected (typically, 443 in production and 5001 in development).
- The insecure port (typically, 80 in production and 5000 in development).

The insecure port must be accessible by the client in order for the app to receive an insecure request and redirect the client to the secure port.

For more information, see Kestrel endpoint configuration or HTTP.sys web server implementation in ASP.NET Core.

Deployment scenarios

Any firewall between the client and server must also have communication ports open for traffic.

If requests are forwarded in a reverse proxy configuration, use Forwarded Headers Middleware before calling HTTPS Redirection Middleware. Forwarded Headers Middleware updates the `Request.Scheme`, using the `X-Forwarded-Proto` header. The middleware permits redirect URIs and other security policies to work correctly. When Forwarded Headers Middleware isn't used, the backend app might not receive the correct scheme and end up in a redirect loop. A common end user error message is that too many redirects have occurred.

When deploying to Azure App Service, follow the guidance in Tutorial: Bind an existing custom SSL certificate to Azure Web Apps.

Options

The following highlighted code calls AddHttpsRedirection to configure middleware options:

C#

```
using static Microsoft.AspNetCore.Http.StatusCodes;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddHsts(options =>
{
    options.Preload = true;
    options.IncludeSubDomains = true;
    options.MaxAge = TimeSpan.FromDays(60);
    options.ExcludedHosts.Add("example.com");
    options.ExcludedHosts.Add("www.example.com");
});

builder.Services.AddHttpsRedirection(options =>
{
    options.RedirectStatusCode = Status307TemporaryRedirect;
    options.HttpsPort = 5001;
});
```

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Calling `AddHttpsRedirection` is only necessary to change the values of `HttpsPort` or `RedirectStatusCode`.

The preceding highlighted code:

- Sets HttpsRedirectionOptions.RedirectStatusCode to Status307TemporaryRedirect, which is the default value. Use the fields of the StatusCodes class for assignments to `RedirectStatusCode`.
- Sets the HTTPS port to 5001.

Configure permanent redirects in production

The middleware defaults to sending a Status307TemporaryRedirect with all redirects. If you prefer to send a permanent redirect status code when the app is in a non-Development environment, wrap the middleware options configuration in a conditional check for a non-Development environment.

When configuring services in `Program.cs`:

C#

```
using static Microsoft.AspNetCore.Http.StatusCodes;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
```

```
if (!builder.Environment.IsDevelopment())
{
    builder.Services.AddHttpsRedirection(options =>
    {
        options.RedirectStatusCode = Status308PermanentRedirect;
        options.HttpsPort = 443;
    });
}

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");

    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

HTTPS Redirection Middleware alternative approach

An alternative to using HTTPS Redirection Middleware (`UseHttpsRedirection`) is to use URL Rewriting Middleware (`AddRedirectToHttps`). `AddRedirectToHttps` can also set the status code and port when the redirect is executed. For more information, see URL Rewriting Middleware.

When redirecting to HTTPS without the requirement for additional redirect rules, we recommend using HTTPS Redirection Middleware (`UseHttpsRedirection`) described in this article.

HTTP Strict Transport Security Protocol (HSTS)

Per OWASP, HTTP Strict Transport Security (HSTS) is an opt-in security enhancement that's specified by a web app through the use of a response header. When a browser that supports HSTS receives this header:

- The browser stores configuration for the domain that prevents sending any communication over HTTP. The browser forces all communication over HTTPS.
- The browser prevents the user from using untrusted or invalid certificates. The browser disables prompts that allow a user to temporarily trust such a certificate.

Because HSTS is enforced by the client, it has some limitations:

- The client must support HSTS.
- HSTS requires at least one successful HTTPS request to establish the HSTS policy.
- The application must check every HTTP request and redirect or reject the HTTP request.

ASP.NET Core implements HSTS with the UseHsts extension method. The following code calls `UseHsts` when the app isn't in development mode:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

`UseHsts` isn't recommended in development because the HSTS settings are highly cacheable by browsers. By default, `UseHsts` excludes the local loopback address.

For production environments that are implementing HTTPS for the first time, set the initial HstsOptions.MaxAge to a small value using one of the TimeSpan methods. Set the value from hours to no more than a single day in case you need to revert the HTTPS infrastructure to HTTP. After you're confident in the sustainability of the HTTPS configuration, increase the HSTS `max-age` value; a commonly used value is one year.

The following highlighted code:

C#

```
using static Microsoft.AspNetCore.Http.StatusCodes;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddHsts(options =>
{
    options.Preload = true;
    options.IncludeSubDomains = true;
    options.MaxAge = TimeSpan.FromDays(60);
    options.ExcludedHosts.Add("example.com");
    options.ExcludedHosts.Add("www.example.com");
});

builder.Services.AddHttpsRedirection(options =>
{
    options.RedirectStatusCode = Status307TemporaryRedirect;
    options.HttpsPort = 5001;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

- Sets the preload parameter of the `Strict-Transport-Security` header. Preload isn't part of the RFC HSTS specification, but is supported by web browsers to preload HSTS sites on fresh install. For more information, see https://hstspreload.org/.
- Enables includeSubDomain, which applies the HSTS policy to Host subdomains.
- Explicitly sets the `max-age` parameter of the `Strict-Transport-Security` header to 60 days. If not set, defaults to 30 days. For more information, see the max-age directive.

- Adds `example.com` to the list of hosts to exclude.

`UseHsts` excludes the following loopback hosts:

- `localhost` : The IPv4 loopback address.
- `127.0.0.1` : The IPv4 loopback address.
- `[::1]` : The IPv6 loopback address.

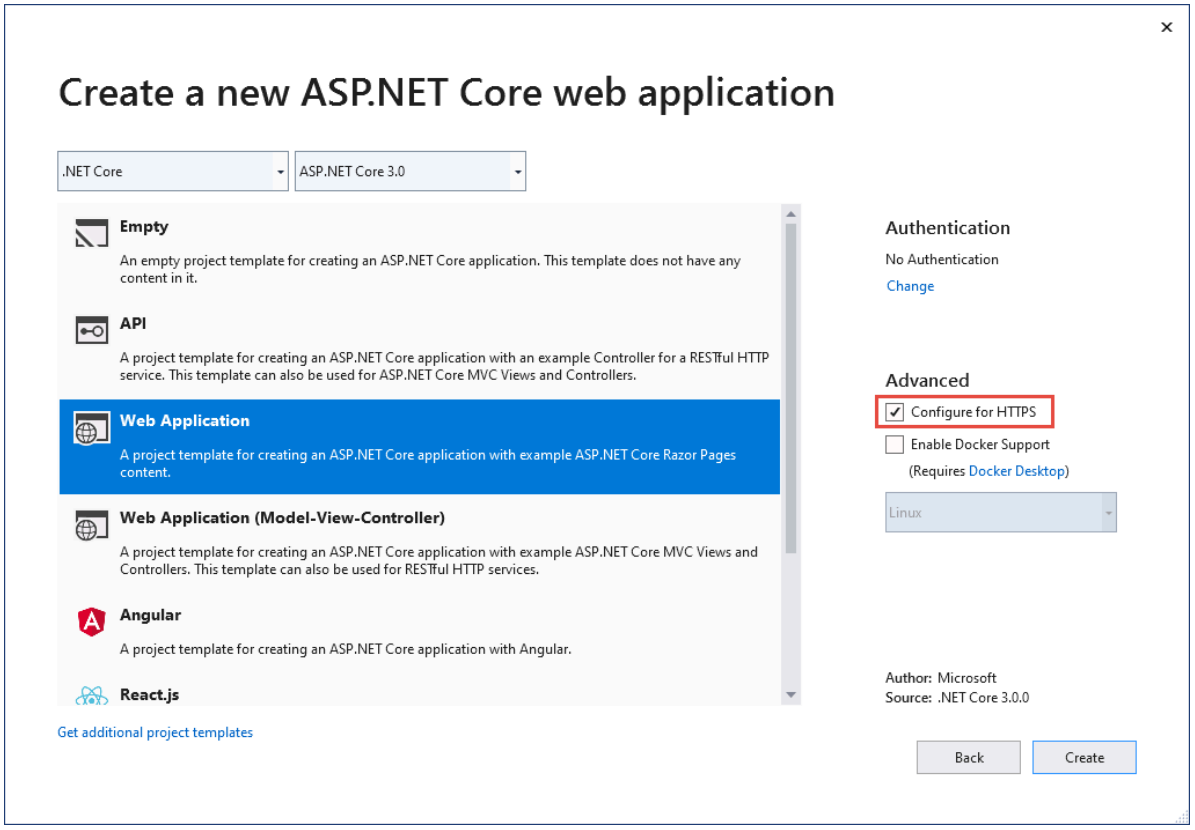
Opt-out of HTTPS/HSTS on project creation

In some backend service scenarios where connection security is handled at the public-facing edge of the network, configuring connection security at each node isn't required. Web apps that are generated from the templates in Visual Studio or from the dotnet new command enable HTTPS redirection and HSTS. For deployments that don't require these scenarios, you can opt-out of HTTPS/HSTS when the app is created from the template.

To opt-out of HTTPS/HSTS:

Visual Studio

Uncheck the **Configure for HTTPS** checkbox.



Trust the ASP.NET Core HTTPS development certificate

The .NET Core SDK includes an HTTPS development certificate. The certificate is installed as part of the first-run experience. For example, `dotnet --info` produces a variation of the following output:

cli

```
ASP.NET Core
------------
Successfully installed the ASP.NET Core HTTPS Development Certificate.
To trust the certificate run 'dotnet dev-certs https --trust' (Windows and
macOS only).
For establishing trust on other platforms refer to the platform specific
documentation.
For more information on configuring HTTPS see
https://go.microsoft.com/fwlink/?linkid=848054.
```

Installing the .NET Core SDK installs the ASP.NET Core HTTPS development certificate to the local user certificate store. The certificate has been installed, but it's not trusted. To trust the certificate, perform the one-time step to run the `dotnet dev-certs` tool:

.NET CLI

```
dotnet dev-certs https --trust
```

The following command provides help on the `dotnet dev-certs` tool:

.NET CLI

```
dotnet dev-certs https --help
```

> ⚠ **Warning**
>
> Do not create a development certificate in an environment that will be redistributed, such as a container image or virtual machine. Doing so can lead to spoofing and elevation of privilege. To help prevent this, set the `DOTNET_GENERATE_ASPNET_CERTIFICATE` environment variable to `false` prior to calling the .NET CLI for the first time. This will skip the automatic generation of the ASP.NET Core development certificate during the CLI's first-run experience.

How to set up a developer certificate for Docker

See this GitHub issue.

Linux-specific considerations

Linux distros differ substantially in how they mark certificates as trusted. While `dotnet dev-certs` is expected to be broadly applicable it is only officially supported on Ubuntu and Fedora and specifically aims to ensure trust in Firefox and Chromium-based browsers (Edge, Chrome, and Chromium).

Dependencies

To establish OpenSSL trust, the `openssl` tool must be on the path.

To establish browser trust (for example in Edge or Firefox), the `certutil` tool must be on the path.

OpenSSL trust

When the ASP.NET Core development certificate is trusted, it is exported to a folder in the current user's home directory. To have OpenSSL (and clients that consume it) pick up this folder, you need to set the `SSL_CERT_DIR` environment variable. You can either do this in a single session by running a command like `export SSL_CERT_DIR=$HOME/.aspnet/dev-certs/trust:/usr/lib/ssl/certs` (the exact value will be in the output when `--verbose` is passed) or by adding it your (distro- and shell-specific) configuration file (for example `.profile`).

This is required to make tools like `curl` trust the development certificate. Or, alternatively, you can pass `-CAfile` or `-CApath` to each individual `curl` invocation.

Note that this requires 1.1.1h or later or 3.0.0 or later, depending on which major version you're using.

If OpenSSL trust gets into a bad state (for example if `dotnet dev-certs https --clean` fails to remove it), it is frequently possible to set things right using the c_rehash tool.

Overrides

If you're using another browser with its own Network Security Services (NSS) store, you can use the `DOTNET_DEV_CERTS_NSSDB_PATHS` environment variable to specify a colon-delimited list of NSS directories (for example, the directory containing `cert9.db`) to which to add the development certificate.

If you store the certificates you want OpenSSL to trust in a specific directory, you can use the `DOTNET_DEV_CERTS_OPENSSL_CERTIFICATE_DIRECTORY` environment variable to indicate where that is.

> ⚠ **Warning**
>
> If you set either of these variables, it is important that they are set to the same values each time trust is updated. If they change, the tool won't know about certificates in the former locations (for example to clean them up).

Using sudo

As on other platforms, development certificates are stored and trusted separately for each user. As a result, if you run `dotnet dev-certs` as a different user (for example by using `sudo`), it is *that* user (for example `root`) that will trust the development certificate.

Trust HTTPS certificate on Linux with linux-dev-certs

linux-dev-certs is an open-source, community-supported, .NET global tool that provides a convenient way to create and trust a developer certificate on Linux. The tool is not maintained or supported by Microsoft.

The following commands install the tool and create a trusted developer certificate:

cli

```
dotnet tool update -g linux-dev-certs
dotnet linux-dev-certs install
```

For more information or to report issues, see the linux-dev-certs GitHub repository.

SUSE Linux Enterprise Server

See this GitHub issue

Troubleshoot certificate problems such as certificate not trusted

This section provides help when the ASP.NET Core HTTPS development certificate has been installed and trusted, but you still have browser warnings that the certificate is not trusted. The ASP.NET Core HTTPS development certificate is used by Kestrel.

To repair the IIS Express certificate, see this Stackoverflow issue.

All platforms - certificate not trusted

Run the following commands:

.NET CLI

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Close any browser instances open. Open a new browser window to app. Certificate trust is cached by browsers.

dotnet dev-certs https --clean Fails

The preceding commands solve most browser trust issues. If the browser is still not trusting the certificate, follow the platform-specific suggestions that follow.

Docker - certificate not trusted

- Delete the *C:\Users{USER}\AppData\Roaming\ASP.NET\Https* folder.
- Clean the solution. Delete the *bin* and *obj* folders.
- Restart the development tool. For example, Visual Studio or Visual Studio Code.

Windows - certificate not trusted

- Check the certificates in the certificate store. There should be a `localhost` certificate with the `ASP.NET Core HTTPS development certificate` friendly name both under `Current User > Personal > Certificates` and `Current User > Trusted root certification authorities > Certificates`
- Remove all the found certificates from both Personal and Trusted root certification authorities. Do **not** remove the IIS Express localhost certificate.

- Run the following commands:

.NET CLI

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Close any browser instances open. Open a new browser window to app.

OS X - certificate not trusted

- Open KeyChain Access.
- Select the System keychain.
- Check for the presence of a localhost certificate.
- Check that it contains a `+` symbol on the icon to indicate it's trusted for all users.
- Remove the certificate from the system keychain.
- Run the following commands:

.NET CLI

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Close any browser instances open. Open a new browser window to app.

See HTTPS Error using IIS Express (dotnet/AspNetCore #16892) for troubleshooting certificate issues with Visual Studio.

Linux certificate not trusted

Check that the certificate being configured for trust is the user HTTPS developer certificate that will be used by the Kestrel server.

Check the current user default HTTPS developer Kestrel certificate at the following location:

```
ls -la ~/.dotnet/corefx/cryptography/x509stores/my
```

The HTTPS developer Kestrel certificate file is the SHA1 thumbprint. When the file is deleted via `dotnet dev-certs https --clean`, it's regenerated when needed with a

different thumbprint. Check the thumbprint of the exported certificate matches with the following command:

```
openssl x509 -noout -fingerprint -sha1 -inform pem -in /usr/local/share/ca-certificates/aspnet/https.crt
```

If the certificate doesn't match, it could be one of the following:

- An old certificate.
- An exported a developer certificate for the root user. For this case, export the certificate.

The root user certificate can be checked at:

```
ls -la /root/.dotnet/corefx/cryptography/x509stores/my
```

IIS Express SSL certificate used with Visual Studio

To fix problems with the IIS Express certificate, select **Repair** from the Visual Studio installer. For more information, see this GitHub issue.

Group policy prevents self-signed certificates from being trusted

In some cases, group policy may prevent self-signed certificates from being trusted. For more information, see this GitHub issue.

Additional information

- Configure ASP.NET Core to work with proxy servers and load balancers
- Host ASP.NET Core on Linux with Nginx: HTTPS configuration
- How to Set Up SSL on IIS
- Configure endpoints for the ASP.NET Core Kestrel web server
- OWASP HSTS browser support

Hosting ASP.NET Core images with Docker over HTTPS

Article • 09/10/2024

By Rick Anderson

ASP.NET Core uses HTTPS by default. HTTPS relies on certificates for trust, identity, and encryption.

This document explains how to run pre-built container images with HTTPS using the .NET command-line interface (CLI). For instructions on how to run Docker in development with Visual Studio, see Developing ASP.NET Core Applications with Docker over HTTPS.

This sample requires Docker 17.06 or later of the Docker client.

Prerequisites

The current .NET SDK.

Certificates

A certificate from a certificate authority is required for production hosting for a domain. Let's Encrypt is a certificate authority that offers free certificates.

This document uses self-signed development certificates for hosting pre-built images over `localhost`. The instructions are similar to using production certificates. The certificate generated by `dotnet dev-certs` is for use with `localhost` only and should ***not*** be used in an environment like Kubernetes. To support HTTPS within a Kubernetes cluster, use the tools provided by Manage TLS Certificates in a Cluster to setup TLS within pods.

Use dotnet dev-certs to create self-signed certificates for development and testing.

For production certs:

- The `dotnet dev-certs` tool is not required.
- Certificates do not need to be stored in the location used in the instructions. Any location should work, although storing certs within your site directory is not recommended.

The instructions contained in the following section volume mount certificates into containers using Docker's `-v` command-line option. You could add certificates into container images with a `COPY` command in a *Dockerfile*, but it's not recommended. Copying certificates into an image isn't recommended for the following reasons:

- It's difficult to use the same image for testing with developer certificates.
- It's difficult to use the same image for Hosting with production certificates.
- There is significant risk of certificate disclosure.

Running pre-built container images with HTTPS

Use the following instructions for your operating system configuration.

Windows using Linux containers

Generate a certificate and configure the local machine:

.NET CLI

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p
<CREDENTIAL_PLACEHOLDER>
dotnet dev-certs https --trust
```

In the preceding commands, replace `<CREDENTIAL_PLACEHOLDER>` with a password.

Run the container image with ASP.NET Core configured for HTTPS in a command shell:

Console

```
docker pull mcr.microsoft.com/dotnet/samples:aspnetapp
docker run --rm -it -p 8000:80 -p 8001:443 -e
ASPNETCORE_URLS="https://+;http://+" -e ASPNETCORE_HTTPS_PORTS=8001 -e
ASPNETCORE_Kestrel__Certificates__Default__Password="
<CREDENTIAL_PLACEHOLDER>" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx -v
%USERPROFILE%\.aspnet\https:/https/
mcr.microsoft.com/dotnet/samples:aspnetapp
```

In the preceding code, replace `<CREDENTIAL_PLACEHOLDER>` with the password. The password must match the password used for the certificate.

When using PowerShell, replace `%USERPROFILE%` with `$env:USERPROFILE`.

Note: The certificate in this case must be a `.pfx` file. Utilizing a `.crt` or `.key` file with or without the password isn't supported with the sample container. For example, when specifying a `.crt` file, the container may return error messages such as 'The server mode SSL must use a certificate with the associated private key.'. When using WSL, validate the mount path to ensure that the certificate loads correctly.

macOS or Linux

Generate certificate and configure local machine:

.NET CLI

```
dotnet dev-certs https -ep ${HOME}/.aspnet/https/aspnetapp.pfx -p
<CREDENTIAL_PLACEHOLDER>
dotnet dev-certs https --trust
```

On Linux, `dotnet dev-certs https --trust` requires .NET 9 SDK or later. For Linux on .NET 8.0.401 SDK and earlier, see your Linux distribution's documentation for trusting a certificate.

In the preceding commands, replace `<CREDENTIAL_PLACEHOLDER>` with a password.

Run the container image with ASP.NET Core configured for HTTPS:

Console

```
docker pull mcr.microsoft.com/dotnet/samples:aspnetapp
docker run --rm -it -p 8000:80 -p 8001:443 -e
ASPNETCORE_URLS="https://+;http://+" -e ASPNETCORE_HTTPS_PORTS=8001 -e
ASPNETCORE_Kestrel__Certificates__Default__Password="
<CREDENTIAL_PLACEHOLDER>" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx -v
${HOME}/.aspnet/https:/https/ mcr.microsoft.com/dotnet/samples:aspnetapp
```

In the preceding code, replace `<CREDENTIAL_PLACEHOLDER>` with the password. The password must match the password used for the certificate.

Windows using Windows containers

Generate certificate and configure local machine:

.NET CLI

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p
<CREDENTIAL_PLACEHOLDER>
```

```
dotnet dev-certs https --trust
```

In the preceding commands, replace `<CREDENTIAL_PLACEHOLDER>` with a password. When using PowerShell, replace `%USERPROFILE%` with `$env:USERPROFILE`.

Run the container image with ASP.NET Core configured for HTTPS:

Console

```
docker pull mcr.microsoft.com/dotnet/samples:aspnetapp
docker run --rm -it -p 8000:80 -p 8001:443 -e
ASPNETCORE_URLS="https://+;http://+" -e ASPNETCORE_HTTPS_PORTS=8001 -e
ASPNETCORE_Kestrel__Certificates__Default__Password="
<CREDENTIAL_PLACEHOLDER>" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=c:\https\aspnetapp.pfx -v
%USERPROFILE%\.aspnet\https:C:\https\ --user ContainerAdministrator
mcr.microsoft.com/dotnet/samples:aspnetapp
```

NOTE: `<CREDENTIAL_PLACEHOLDER>` is a placeholder for the Kestrel certificates default password.

The password must match the password used for the certificate. When using PowerShell, replace `%USERPROFILE%` with `$env:USERPROFILE`.

Developing ASP.NET Core Applications with Docker over HTTPS

See Developing ASP.NET Core Applications with Docker over HTTPS for information and samples on how to develop ASP.NET Core applications with HTTPS in Docker containers.

See also

- Developing ASP.NET Core Applications with Docker over HTTPS
- dotnet dev-certs

Hosting ASP.NET Core images with Docker Compose over HTTPS

Article • 09/10/2024

ASP.NET Core uses HTTPS by default. HTTPS relies on certificates for trust, identity, and encryption.

This document explains how to run pre-built container images with HTTPS.

See Developing ASP.NET Core Applications with Docker over HTTPS for development scenarios.

This sample requires Docker 17.06 or later of the Docker client.

Prerequisites

The .NET Core 2.2 SDK or later is required for some of the instructions in this document.

Certificates

A certificate from a certificate authority is required for production hosting for a domain. Let's Encrypt is a certificate authority that offers free certificates.

This document uses self-signed development certificates for hosting pre-built images over `localhost`. The instructions are similar to using production certificates.

For production certificates:

- The `dotnet dev-certs` tool is not required.
- Certificates don't need to be stored in the location used in the instructions. Store the certificates in any location outside the site directory.

The instructions contained in the following section volume mount certificates into containers using the `volumes` property in *docker-compose.yml*. You could add certificates into container images with a `COPY` command in a *Dockerfile*, but it's not recommended. Copying certificates into an image isn't recommended for the following reasons:

- It makes it difficult to use the same image for testing with developer certificates.
- It makes it difficult to use the same image for Hosting with production certificates.
- There is significant risk of certificate disclosure.

Starting a container with https support using docker compose

Use the following instructions for your operating system configuration.

Windows using Linux containers

Generate certificate and configure local machine:

PowerShell

```
dotnet dev-certs https -ep "$env:USERPROFILE\.aspnet\https\aspnetapp.pfx"  -
p $CREDENTIAL_PLACEHOLDER$
dotnet dev-certs https --trust
```

The previous command using the .NET CLI:

.NET CLI

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p
$CREDENTIAL_PLACEHOLDER$
dotnet dev-certs https --trust
```

In the preceding commands, replace `$CREDENTIAL_PLACEHOLDER$` with a password.

Create a *docker-compose.debug.yml* file with the following content:

YAML

```
version: '3.4'

services:
  webapp:
    image: mcr.microsoft.com/dotnet/samples:aspnetapp
    ports:
      - 80
      - 443
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password=password
      - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx
    volumes:
      - ~/.aspnet/https:/https:ro
```

The password specified in the docker compose file must match the password used for the certificate.

Start the container with ASP.NET Core configured for HTTPS:

Console

```
docker-compose -f "docker-compose.debug.yml" up -d
```

macOS or Linux

Generate certificate and configure local machine:

.NET CLI

```
dotnet dev-certs https -ep ${HOME}/.aspnet/https/aspnetapp.pfx -p
$CREDENTIAL_PLACEHOLDER$
dotnet dev-certs https --trust
```

On Linux, `dotnet dev-certs https --trust` requires .NET 9 SDK or later. For Linux on .NET 8.0.401 SDK and earlier, see your Linux distribution's documentation for trusting a certificate.

In the preceding commands, replace `$CREDENTIAL_PLACEHOLDER$` with a password.

Create a *docker-compose.debug.yml* file with the following content:

YAML

```
version: '3.4'

services:
  webapp:
    image: mcr.microsoft.com/dotnet/samples:aspnetapp
    ports:
      - 80
      - 443
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password=password
      - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx
    volumes:
      - ~/.aspnet/https:/https:ro
```

The password specified in the docker compose file must match the password used for the certificate.

Start the container with ASP.NET Core configured for HTTPS:

Console

```
docker-compose -f "docker-compose.debug.yml" up -d
```

Windows using Windows containers

Generate certificate and configure local machine:

.NET CLI

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p
$CREDENTIAL_PLACEHOLDER$
dotnet dev-certs https --trust
```

In the preceding commands, replace `$CREDENTIAL_PLACEHOLDER$` with a password.

Create a *docker-compose.debug.yml* file with the following content:

YAML

```
version: '3.4'

services:
  webapp:
    image: mcr.microsoft.com/dotnet/samples:aspnetapp
    ports:
      - 80
      - 443
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password=password
      -
ASPNETCORE_Kestrel__Certificates__Default__Path=C:\https\aspnetapp.pfx
    volumes:
      - ${USERPROFILE}\.aspnet\https:C:\https:ro
```

The password specified in the docker compose file must match the password used for the certificate.

Start the container with ASP.NET Core configured for HTTPS:

Console

```
docker-compose -f "docker-compose.debug.yml" up -d
```

See also

- dotnet dev-certs

EU General Data Protection Regulation (GDPR) support in ASP.NET Core

Article • 06/17/2024

By Rick Anderson

ASP.NET Core provides APIs and templates to help meet some of the EU General Data Protection Regulation (GDPR) requirements:

- The project templates include extension points and stubbed markup that you can replace with your privacy and cookie use policy.
- The `Pages/Privacy.cshtml` page or `Views/Home/Privacy.cshtml` view provides a page to detail your site's privacy policy.

To enable the default cookie consent feature like that found in the ASP.NET Core 2.2 templates in a current ASP.NET Core template generated app, add the following highlighted code to `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<CookiePolicyOptions>(options =>
{
    // This lambda determines whether user consent for non-essential
    // cookies is needed for a given request.
    options.CheckConsentNeeded = context => true;

    options.MinimumSameSitePolicy = SameSiteMode.None;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseCookiePolicy();

app.UseRouting();

app.UseAuthorization();
```

```
app.MapRazorPages();

app.Run();
```

In the preceding code, CookiePolicyOptions and UseCookiePolicy are used.

- Add the cookie consent partial to the `_Layout.cshtml` file:

CSHTML

```
            @*Previous markup removed for brevity*@
    </header>
    <div class="container">
        <partial name="_CookieConsentPartial" />
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2022 - WebGDPR - <a asp-area="" asp-
page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js">
</script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

- Add the `_CookieConsentPartial.cshtml` file to the project:

CSHTML

```
@using Microsoft.AspNetCore.Http.Features

@{
    var consentFeature = Context.Features.Get<ITrackingConsentFeature>
();
    var showBanner = !consentFeature?.CanTrack ?? false;
    var cookieString = consentFeature?.CreateConsentCookie();
}

@if (showBanner)
{
```

```
    <div id="cookieConsent" class="alert alert-info alert-dismissible
fade show" role="alert">
        Use this space to summarize your privacy and cookie use policy.
<a asp-page="/Privacy">Learn More</a>.
        <button type="button" class="accept-policy close" data-bs-
dismiss="alert" aria-label="Close" data-cookie-string="@cookieString">
            <span aria-hidden="true">Accept</span>
        </button>
    </div>
    <script>
        (function () {
            var button = document.querySelector("#cookieConsent
button[data-cookie-string]");
            button.addEventListener("click", function (event) {
                document.cookie = button.dataset.cookieString;
            }, false);
        })();
    </script>
}
```

- Select the ASP.NET Core 2.2 version of this article to read about the cookie consent feature.

Customize the cookie consent value

Specify the value used to track if the user consented to the cookie use policy using the CookiePolicyOptions.ConsentCookieValue property:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<CookiePolicyOptions>(options =>
{
    options.CheckConsentNeeded = context => true;
    options.MinimumSameSitePolicy = SameSiteMode.None;
    options.ConsentCookieValue = "true";
});

var app = builder.Build();
```

Encryption at rest

Some databases and storage mechanisms allow for encryption at rest. Encryption at rest:

- Encrypts stored data automatically.

- Encrypts without configuration, programming, or other work for the software that accesses the data.
- Is the easiest and safest option.
- Allows the database to manage keys and encryption.

For example:

- Microsoft SQL and Azure SQL provide Transparent Data Encryption (TDE).
- SQL Azure encrypts the database by default
- Azure Blobs, Files, Table, and Queue Storage are encrypted by default.

For databases that don't provide built-in encryption at rest, you may be able to use disk encryption to provide the same protection. For example:

- BitLocker for Windows Server
- Linux:
  - eCryptfs
  - EncFS.

Additional resources

- Microsoft.com/GDPR

Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core

Article • 10/21/2024

By Fiyaz Hasan and Rick Anderson

Cross-site request forgery is an attack against web-hosted apps whereby a malicious web app can influence the interaction between a client browser and a web app that trusts that browser. These attacks are possible because web browsers send some types of authentication tokens automatically with every request to a website. This form of exploit is also known as a *one-click attack* or *session riding* because the attack takes advantage of the user's previously authenticated session. Cross-site request forgery is also known as XSRF or CSRF.

An example of a CSRF attack:

1. A user signs into `www.good-banking-site.example.com` using forms authentication. The server authenticates the user and issues a response that includes an authentication cookie. The site is vulnerable to attack because it trusts any request that it receives with a valid authentication cookie.

2. The user visits a malicious site, `www.bad-crook-site.example.com`.

   The malicious site, `www.bad-crook-site.example.com`, contains an HTML form similar to the following example:

   HTML

   ```
   <h1>Congratulations! You're a Winner!</h1>
   <form action="https://good-banking-site.com/api/account" method="post">
       <input type="hidden" name="Transaction" value="withdraw" />
       <input type="hidden" name="Amount" value="1000000" />
       <input type="submit" value="Click to collect your prize!" />
   </form>
   ```

   Notice that the form's `action` posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.

3. The user selects the submit button. The browser makes the request and automatically includes the authentication cookie for the requested domain, `www.good-banking-site.example.com`.

4. The request runs on the `www.good-banking-site.example.com` server with the user's authentication context and can perform any action that an authenticated user is allowed to perform.

In addition to the scenario where the user selects the button to submit the form, the malicious site could:

- Run a script that automatically submits the form.
- Send the form submission as an AJAX request.
- Hide the form using CSS.

These alternative scenarios don't require any action or input from the user other than initially visiting the malicious site.

Using HTTPS doesn't prevent a CSRF attack. The malicious site can send an `https://www.good-banking-site.com/` request as easily as it can send an insecure request.

Some attacks target endpoints that respond to GET requests, in which case an image tag can be used to perform the action. This form of attack is common on forum sites that permit images but block JavaScript. Apps that change state on GET requests, where variables or resources are altered, are vulnerable to malicious attacks. **GET requests that change state are insecure. A best practice is to never change state on a GET request.**

CSRF attacks are possible against web apps that use cookies for authentication because:

- Browsers store cookies issued by a web app.
- Stored cookies include session cookies for authenticated users.
- Browsers send all of the cookies associated with a domain to the web app every request regardless of how the request to app was generated within the browser.

However, CSRF attacks aren't limited to exploiting cookies. For example, Basic and Digest authentication are also vulnerable. After a user signs in with Basic or Digest authentication, the browser automatically sends the credentials until the session ends.

In this context, *session* refers to the client-side session during which the user is authenticated. It's unrelated to server-side sessions or ASP.NET Core Session Middleware.

Users can protect against CSRF vulnerabilities by taking precautions:

- Sign out of web apps when finished using them.
- Clear browser cookies periodically.

However, CSRF vulnerabilities are fundamentally a problem with the web app, not the end user.

Authentication fundamentals

Cookie-based authentication is a popular form of authentication. Token-based authentication systems are growing in popularity, especially for Single Page Applications (SPAs).

Cookie-based authentication

When a user authenticates using their username and password they're issued a token containing an authentication ticket. The token can be used for authentication and authorization. The token is stored as a cookie that's sent with every request the client makes. Generating and validating this cookie is performed with the Cookie Authentication Middleware. The middleware serializes a user principal into an encrypted cookie. On subsequent requests, the middleware validates the cookie, recreates the principal, and assigns the principal to the HttpContext.User property.

Token-based authentication

When a user is authenticated, they're issued a token (not an antiforgery token). The token contains user information in the form of claims or a reference token that points the app to user state maintained in the app. When a user attempts to access a resource that requires authentication, the token is sent to the app with an extra authorization header in the form of a Bearer token. This approach makes the app stateless. In each subsequent request, the token is passed in the request for server-side validation. This token isn't *encrypted*; it's *encoded*. On the server, the token is decoded to access its information. To send the token on subsequent requests, store the token in the browser's local storage. Placing a token in the browser local storage and retrieving it and using it as a bearer token provides protection against CSRF attacks. However, should the app be vulnerable to script injection via XSS or a compromised external JavaScript file, a cyberattacker could retrieve any value from local storage and send it to themselves. ASP.NET Core encodes all server side output from variables by default, reducing the risk of XSS. If you override this behavior by using Html.Raw or custom code with untrusted input then you may increase the risk of XSS.

Don't be concerned about CSRF vulnerability if the token is stored in the browser's local storage. CSRF is a concern when the token is stored in a cookie. For more information, see the GitHub issue SPA code sample adds two cookies.

Multiple apps hosted at one domain

Shared hosting environments are vulnerable to session hijacking, sign-in CSRF, and other attacks.

Although `example1.contoso.net` and `example2.contoso.net` are different hosts, there's an implicit trust relationship between hosts under the `*.contoso.net` domain. This implicit trust relationship allows potentially untrusted hosts to affect each other's cookies (the same-origin policies that govern AJAX requests don't necessarily apply to HTTP cookies).

Attacks that exploit trusted cookies between apps hosted on the same domain can be prevented by not sharing domains. When each app is hosted on its own domain, there's no implicit cookie trust relationship to exploit.

Antiforgery in ASP.NET Core

> ⚠ **Warning**
>
> ASP.NET Core implements antiforgery using ASP.NET Core Data Protection. The data protection stack must be configured to work in a server farm. For more information, see Configuring data protection.

Antiforgery middleware is added to the Dependency injection container when one of the following APIs is called in `Program.cs`:

- AddMvc
- MapRazorPages
- MapControllerRoute
- AddRazorComponents

For more information, see Antiforgery with Minimal APIs.

The FormTagHelper injects antiforgery tokens into HTML form elements. The following markup in a Razor file automatically generates antiforgery tokens:

CSHTML

```
<form method="post">
    <!-- ... -->
</form>
```

Similarly, IHtmlHelper.BeginForm generates antiforgery tokens by default if the form's method isn't GET.

The automatic generation of antiforgery tokens for HTML form elements happens when the `<form>` tag contains the `method="post"` attribute and either of the following are true:

- The action attribute is empty (`action=""`).
- The action attribute isn't supplied (`<form method="post">`).

Automatic generation of antiforgery tokens for HTML form elements can be disabled:

- Explicitly disable antiforgery tokens with the `asp-antiforgery` attribute:

  CSHTML

  ```
  <form method="post" asp-antiforgery="false">
      <!-- ... -->
  </form>
  ```

- The form element is opted-out of Tag Helpers by using the Tag Helper ! opt-out symbol:

  CSHTML

  ```
  <!form method="post">
      <!-- ... -->
  </!form>
  ```

- Remove the `FormTagHelper` from the view. The `FormTagHelper` can be removed from a view by adding the following directive to the Razor view:

  CSHTML

  ```
  @removeTagHelper Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper,
  Microsoft.AspNetCore.Mvc.TagHelpers
  ```

> ⓘ Note
>
> Razor Pages are automatically protected from XSRF/CSRF. For more information, see XSRF/CSRF and Razor Pages.

The most common approach to protecting against CSRF attacks is to use the *Synchronizer Token Pattern* (STP). STP is used when the user requests a page with form data:

1. The server sends a token associated with the current user's identity to the client.
2. The client sends back the token to the server for verification.
3. If the server receives a token that doesn't match the authenticated user's identity, the request is rejected.

The token is unique and unpredictable. The token can also be used to ensure proper sequencing of a series of requests (for example, ensuring the request sequence of: page 1 > page 2 > page 3). All of the forms in ASP.NET Core MVC and Razor Pages templates generate antiforgery tokens. The following pair of view examples generates antiforgery tokens:

CSHTML

```
<form asp-action="Index" asp-controller="Home" method="post">
    <!-- ... -->
</form>

@using (Html.BeginForm("Index", "Home"))
{
    <!-- ... -->
}
```

Explicitly add an antiforgery token to a `<form>` element without using Tag Helpers with the HTML helper @Html.AntiForgeryToken:

CSHTML

```
<form asp-action="Index" asp-controller="Home" method="post">
    @Html.AntiForgeryToken()

    <!-- ... -->
</form>
```

In each of the preceding cases, ASP.NET Core adds a hidden form field similar to the following example:

HTML

```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ...
s2-m9Yw">
```

ASP.NET Core includes three filters for working with antiforgery tokens:

- ValidateAntiForgeryToken
- AutoValidateAntiforgeryToken

- IgnoreAntiforgeryToken

Antiforgery with `AddControllers`

Calling AddControllers does ***not*** enable antiforgery tokens. AddControllersWithViews must be called to have built-in antiforgery token support.

Multiple browser tabs and the Synchronizer Token Pattern

With the Synchronizer Token Pattern, only the most recently loaded page contains a valid antiforgery token. Using multiple tabs can be problematic. For example, if a user opens multiple tabs:

- Only the most recently loaded tab contains a valid antiforgery token.
- Requests made from previously loaded tabs fail with an error: `Antiforgery token validation failed. The antiforgery cookie token and request token do not match`

Consider alternative CSRF protection patterns if this poses an issue.

Configure antiforgery with `AntiforgeryOptions`

Customize AntiforgeryOptions in `Program.cs`:

C#

```
builder.Services.AddAntiforgery(options =>
{
    // Set Cookie properties using CookieBuilder properties†.
    options.FormFieldName = "AntiforgeryFieldname";
    options.HeaderName = "X-CSRF-TOKEN-HEADERNAME";
    options.SuppressXFrameOptionsHeader = false;
});
```

Set the antiforgery `Cookie` properties using the properties of the CookieBuilder class, as shown in the following table.

Expand table

Option	Description
Cookie	Determines the settings used to create the antiforgery cookies.

Option	Description
FormFieldName	The name of the hidden form field used by the antiforgery system to render antiforgery tokens in views.
HeaderName	The name of the header used by the antiforgery system. If `null`, the system considers only form data.
SuppressXFrameOptionsHeader	Specifies whether to suppress generation of the `X-Frame-Options` header. By default, the header is generated with a value of "SAMEORIGIN". Defaults to `false`.

For more information, see CookieAuthenticationOptions.

Generate antiforgery tokens with `IAntiforgery`

IAntiforgery provides the API to configure antiforgery features. `IAntiforgery` can be requested in `Program.cs` using WebApplication.Services. The following example uses middleware from the app's home page to generate an antiforgery token and send it in the response as a cookie:

C#

```
app.UseRouting();

app.UseAuthorization();

var antiforgery = app.Services.GetRequiredService<IAntiforgery>();

app.Use((context, next) =>
{
    var requestPath = context.Request.Path.Value;

    if (string.Equals(requestPath, "/", StringComparison.OrdinalIgnoreCase)
        || string.Equals(requestPath, "/index.html",
StringComparison.OrdinalIgnoreCase))
    {
        var tokenSet = antiforgery.GetAndStoreTokens(context);
        context.Response.Cookies.Append("XSRF-TOKEN",
tokenSet.RequestToken!,
            new CookieOptions { HttpOnly = false });
    }

    return next(context);
});
```

The preceding example sets a cookie named `XSRF-TOKEN`. The client can read this cookie and provide its value as a header attached to AJAX requests. For example, Angular

includes built-in XSRF protection that reads a cookie named `XSRF-TOKEN` by default.

Require antiforgery validation

The ValidateAntiForgeryToken action filter can be applied to an individual action, a controller, or globally. Requests made to actions that have this filter applied are blocked unless the request includes a valid antiforgery token:

```
C#

[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index()
{
    // ...

    return RedirectToAction();
}
```

The `ValidateAntiForgeryToken` attribute requires a token for requests to the action methods it marks, including HTTP GET requests. If the `ValidateAntiForgeryToken` attribute is applied across the app's controllers, it can be overridden with the `IgnoreAntiforgeryToken` attribute.

Automatically validate antiforgery tokens for unsafe HTTP methods only

Instead of broadly applying the `ValidateAntiForgeryToken` attribute and then overriding it with `IgnoreAntiforgeryToken` attributes, the AutoValidateAntiforgeryToken attribute can be used. This attribute works identically to the `ValidateAntiForgeryToken` attribute, except that it doesn't require tokens for requests made using the following HTTP methods:

- GET
- HEAD
- OPTIONS
- TRACE

We recommend use of `AutoValidateAntiforgeryToken` broadly for non-API scenarios. This attribute ensures POST actions are protected by default. The alternative is to ignore antiforgery tokens by default, unless `ValidateAntiForgeryToken` is applied to individual action methods. It's more likely in this scenario for a POST action method to be left

unprotected by mistake, leaving the app vulnerable to CSRF attacks. All POSTs should send the antiforgery token.

APIs don't have an automatic mechanism for sending the non-cookie part of the token. The implementation probably depends on the client code implementation. Some examples are shown below:

Class-level example:

```
C#

[AutoValidateAntiforgeryToken]
public class HomeController : Controller
```

Global example:

```
C#

builder.Services.AddControllersWithViews(options =>
{
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute());
});
```

Override global or controller antiforgery attributes

The IgnoreAntiforgeryToken filter is used to eliminate the need for an antiforgery token for a given action (or controller). When applied, this filter overrides `ValidateAntiForgeryToken` and `AutoValidateAntiforgeryToken` filters specified at a higher level (globally or on a controller).

```
C#

[IgnoreAntiforgeryToken]
public IActionResult IndexOverride()
{
    // ...

    return RedirectToAction();
}
```

Refresh tokens after authentication

Tokens should be refreshed after the user is authenticated by redirecting the user to a view or Razor Pages page.

JavaScript, AJAX, and SPAs

In traditional HTML-based apps, antiforgery tokens are passed to the server using hidden form fields. In modern JavaScript-based apps and SPAs, many requests are made programmatically. These AJAX requests may use other techniques, such as request headers or cookies, to send the token.

If cookies are used to store authentication tokens and to authenticate API requests on the server, CSRF is a potential problem. If local storage is used to store the token, CSRF vulnerability might be mitigated because values from local storage aren't sent automatically to the server with every request. Using local storage to store the antiforgery token on the client and sending the token as a request header is a recommended approach.

Blazor

For more information, see ASP.NET Core Blazor authentication and authorization.

JavaScript

Using JavaScript with views, the token can be created using a service from within the view. Inject the IAntiforgery service into the view and call GetAndStoreTokens:

CSHTML

```
@inject Microsoft.AspNetCore.Antiforgery.IAntiforgery Antiforgery

@{
    ViewData["Title"] = "JavaScript";

    var requestToken = Antiforgery.GetAndStoreTokens(Context).RequestToken;
}

<input id="RequestVerificationToken" type="hidden" value="@requestToken" />

<button id="button" class="btn btn-primary">Submit with Token</button>
<div id="result" class="mt-2"></div>

@section Scripts {
<script>
    document.addEventListener("DOMContentLoaded", () => {
        const resultElement = document.getElementById("result");

        document.getElementById("button").addEventListener("click", async ()
=> {

            const response = await fetch("@Url.Action("FetchEndpoint")", {
```

```
                method: "POST",
                headers: {
                    RequestVerificationToken:

document.getElementById("RequestVerificationToken").value
                }
            });

            if (response.ok) {
                resultElement.innerText = await response.text();
            } else {
                resultElement.innerText = `Request Failed:
${response.status}`
            }
        });
    });
</script>
}
```

The preceding example uses JavaScript to read the hidden field value for the AJAX POST header.

This approach eliminates the need to deal directly with setting cookies from the server or reading them from the client. However, when injecting the IAntiforgery service isn't possible, use JavaScript to access tokens in cookies:

- Access tokens in an additional request to the server, typically usually `same-origin`.
- Use the cookie's contents to create a header with the token's value.

Assuming the script sends the token in a request header called `X-XSRF-TOKEN`, configure the antiforgery service to look for the `X-XSRF-TOKEN` header:

C#

```
builder.Services.AddAntiforgery(options => options.HeaderName = "X-XSRF-
TOKEN");
```

The following example adds a protected endpoint that writes the request token to a JavaScript-readable cookie:

C#

```
app.UseAuthorization();
app.MapGet("antiforgery/token", (IAntiforgery forgeryService, HttpContext
context) =>
{
    var tokens = forgeryService.GetAndStoreTokens(context);
    context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken!,
            new CookieOptions { HttpOnly = false });
```

```
    return Results.Ok();
}).RequireAuthorization();
```

The following example uses JavaScript to make an AJAX request to obtain the token and make another request with the appropriate header:

JavaScript

```
var response = await fetch("/antiforgery/token", {
    method: "GET",
    headers: { "Authorization": authorizationToken }
});

if (response.ok) {
    // https://developer.mozilla.org/docs/web/api/document/cookie
    const xsrfToken = document.cookie
        .split("; ")
        .find(row => row.startsWith("XSRF-TOKEN="))
        .split("=")[1];

    response = await fetch("/JavaScript/FetchEndpoint", {
        method: "POST",
        headers: { "X-XSRF-TOKEN": xsrfToken, "Authorization":
authorizationToken }
    });

    if (response.ok) {
        resultElement.innerText = await response.text();
    } else {
        resultElement.innerText = `Request Failed: ${response.status}`
    }
} else {
    resultElement.innerText = `Request Failed: ${response.status}`
}
```

> **Note**
>
> When the antiforgery token is provided in both the request header and in the form payload, only the token in the header is validated.

Antiforgery with Minimal APIs

Call AddAntiforgery and UseAntiforgery(IApplicationBuilder) to register antiforgery services in DI. Antiforgery tokens are used to mitigate cross-site request forgery attacks.

C#

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The antiforgery middleware:

- Does ***not*** short-circuit the execution of the rest of the request pipeline.
- Sets the IAntiforgeryValidationFeature in the HttpContext.Features of the current request.

The antiforgery token is only validated if:

- The endpoint contains metadata implementing IAntiforgeryMetadata where `RequiresValidation=true`.
- The HTTP method associated with the endpoint is a relevant HTTP method. The relevant methods are all HTTP methods except for TRACE, OPTIONS, HEAD, and GET.
- The request is associated with a valid endpoint.

Note: When enabled manually, the antiforgery middleware must run after the authentication and authorization middleware to prevent reading form data when the user is unauthenticated.

By default, minimal APIs that accept form data require antiforgery token validation.

Consider the following `GenerateForm` method:

C#

```
public static string GenerateForm(string action,
    AntiforgeryTokenSet token, bool UseToken=true)
{
    string tokenInput = "";
    if (UseToken)
    {
        tokenInput = $@"<input name=""{token.FormFieldName}""
                        type=""hidden"" value=""{token.RequestToken}"" />";
    }

    return $@"
```

```
    <html><body>
        <form action=""{action}"" method=""POST"" enctype=""multipart/form-
data"">
            {tokenInput}
            <input type=""text"" name=""name"" />
            <input type=""date"" name=""dueDate"" />
            <input type=""checkbox"" name=""isCompleted"" />
            <input type=""submit"" />
        </form>
    </body></html>
";
}
```

The preceding code has three arguments, the action, the antiforgery token, and a `bool` indicating whether the token should be used.

Consider the following sample:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

// Pass token
app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    return Results.Content(MyHtml.GenerateForm("/todo", token),
"text/html");
});

// Don't pass a token, fails
app.MapGet("/SkipToken", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    return Results.Content(MyHtml.GenerateForm("/todo",token, false ),
"text/html");
});

// Post to /todo2. DisableAntiforgery on that endpoint so no token needed.
app.MapGet("/DisableAntiforgery", (HttpContext context, IAntiforgery
antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    return Results.Content(MyHtml.GenerateForm("/todo2", token, false),
```

```
"text/html");
});

app.MapPost("/todo", ([FromForm] Todo todo) => Results.Ok(todo));

app.MapPost("/todo2", ([FromForm] Todo todo) => Results.Ok(todo))
                                                    .DisableAntiforgery();

app.Run();

class Todo
{
    public required string Name { get; set; }
    public bool IsCompleted { get; set; }
    public DateTime DueDate { get; set; }
}

public static class MyHtml
{
    public static string GenerateForm(string action,
        AntiforgeryTokenSet token, bool UseToken=true)
    {
        string tokenInput = "";
        if (UseToken)
        {
            tokenInput = $@"<input name=""{token.FormFieldName}""
                                type=""hidden"" value=""{token.RequestToken}""
/>";
        }

        return $@"
        <html><body>
            <form action=""{action}"" method=""POST""
enctype=""multipart/form-data"">
                {tokenInput}
                <input type=""text"" name=""name"" />
                <input type=""date"" name=""dueDate"" />
                <input type=""checkbox"" name=""isCompleted"" />
                <input type=""submit"" />
            </form>
        </body></html>
    ";
    }
}
```

In the preceding code, posts to:

- `/todo` require a valid antiforgery token.
- `/todo2` do ***not*** require a valid antiforgery token because DisableAntiforgery is called.

C#

```
app.MapPost("/todo", ([FromForm] Todo todo) => Results.Ok(todo));

app.MapPost("/todo2", ([FromForm] Todo todo) => Results.Ok(todo))
                                                .DisableAntiforgery();
```

A POST to:

- `/todo` from the form generated by the `/` endpoint succeeds because the antiforgery token is valid.
- `/todo` from the form generated by the `/SkipToken` fails because the antiforgery is not included.
- `/todo2` from the form generated by the `/DisableAntiforgery` endpoint succeeds because the antiforgery is not required.

C#

```
app.MapPost("/todo", ([FromForm] Todo todo) => Results.Ok(todo));

app.MapPost("/todo2", ([FromForm] Todo todo) => Results.Ok(todo))
                                                .DisableAntiforgery();
```

When a form is submitted without a valid antiforgery token:

- In the development environment, an exception is thrown.
- In the production environment, a message is logged.

Windows authentication and antiforgery cookies

When using Windows Authentication, application endpoints must be protected against CSRF attacks in the same way as done for cookies. The browser implicitly sends the authentication context to the server and endpoints need to be protected against CSRF attacks.

Extend antiforgery

The IAntiforgeryAdditionalDataProvider type allows developers to extend the behavior of the anti-CSRF system by round-tripping additional data in each token. The GetAdditionalData method is called each time a field token is generated, and the return value is embedded within the generated token. An implementer could return a timestamp, a nonce, or any other value and then call ValidateAdditionalData to validate

this data when the token is validated. The client's username is already embedded in the generated tokens, so there's no need to include this information. If a token includes supplemental data but no `IAntiForgeryAdditionalDataProvider` is configured, the supplemental data isn't validated.

Additional resources

- Host ASP.NET Core in a web farm

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Prevent open redirect attacks in ASP.NET Core

Article • 09/27/2024

A web app that redirects to a URL that's specified via the request such as the querystring or form data can potentially be tampered with to redirect users to an external, malicious URL. This tampering is called an open redirection attack.

Whenever your application logic redirects to a specified URL, you must verify that the redirection URL hasn't been tampered with. ASP.NET Core has built-in functionality to help protect apps from open redirect (also known as open redirection) attacks.

What is an open redirect attack?

Web applications frequently redirect users to a login page when they access resources that require authentication. The redirection typically includes a `returnUrl` querystring parameter so that the user can be returned to the originally requested URL after they have successfully logged in. After the user authenticates, they're redirected to the URL they had originally requested.

Because the destination URL is specified in the querystring of the request, a malicious user could tamper with the querystring. A tampered querystring could allow the site to redirect the user to an external, malicious site. This technique is called an open redirect (or redirection) attack.

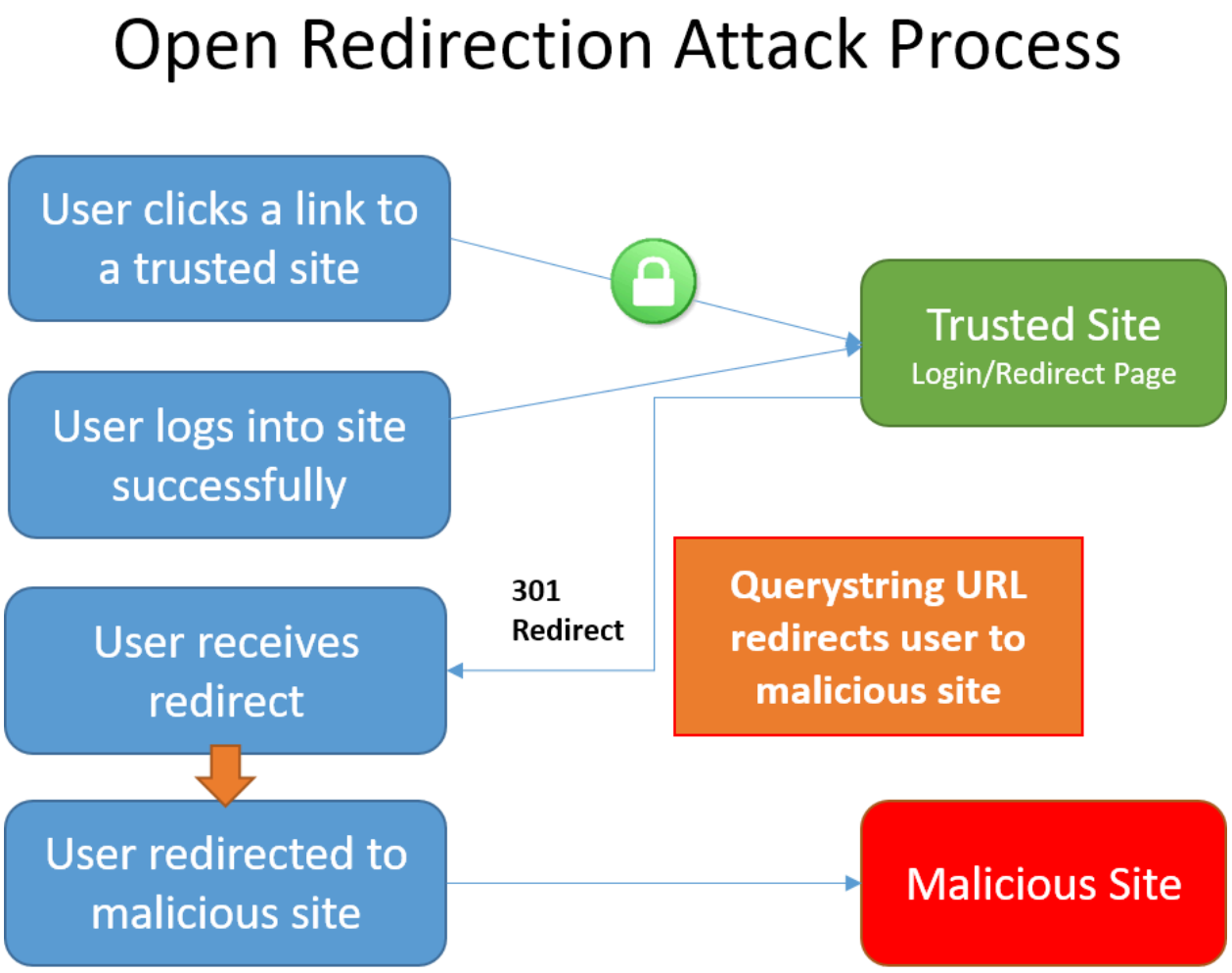
An example attack

A malicious user can develop an attack intended to allow the malicious user access to a user's credentials or sensitive information. To begin the attack, the malicious user convinces the user to click a link to your site's login page with a `returnUrl` querystring value added to the URL. For example, consider an app at `contoso.com` that includes a login page at `http://contoso.com/Account/LogOn?returnUrl=/Home/About`. The attack follows these steps:

1. The user clicks a malicious link to `http://contoso.com/Account/LogOn?returnUrl=http://contoso1.com/Account/LogOn` (the second URL is "contoso**1**.com", not "contoso.com").
2. The user logs in successfully.

3. The user is redirected (by the site) to `http://contoso1.com/Account/LogOn` (a malicious site that looks exactly like real site).
4. The user logs in again (giving malicious site their credentials) and is redirected back to the real site.

The user likely believes that their first attempt to log in failed and that their second attempt is successful. The user most likely remains unaware that their credentials are compromised.



In addition to login pages, some sites provide redirect pages or endpoints. Imagine your app has a page with an open redirect, `/Home/Redirect`. A cyberattacker could create, for example, a link in an email that goes to `[yoursite]/Home/Redirect?url=http://phishingsite.com/Home/Login`. A typical user will look at the URL and see it begins with your site name. Trusting that, they will click the link. The open redirect would then send the user to the phishing site, which looks identical to yours, and the user would likely login to what they believe is your site.

Protecting against open redirect attacks

When developing web applications, treat all user-provided data as untrustworthy. If your application has functionality that redirects the user based on the contents of the URL, ensure that such redirects are only done locally within your app (or to a known URL, not any URL that may be supplied in the querystring).

LocalRedirect

Use the `LocalRedirect` helper method from the base `Controller` class:

C#

```
public IActionResult SomeAction(string redirectUrl)
{
    return LocalRedirect(redirectUrl);
}
```

`LocalRedirect` will throw an exception if a non-local URL is specified. Otherwise, it behaves just like the `Redirect` method.

IsLocalUrl

Use the IsLocalUrl method to test URLs before redirecting:

The following example shows how to check whether a URL is local before redirecting.

C#

```
private IActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction(nameof(HomeController.Index), "Home");
    }
}
```

The `IsLocalUrl` method protects users from being inadvertently redirected to a malicious site. You can log the details of the URL that was provided when a non-local URL is supplied in a situation where you expected a local URL. Logging redirect URLs may help in diagnosing redirection attacks.

Prevent Cross-Site Scripting (XSS) in ASP.NET Core

Article • 09/27/2024

By Rick Anderson

Cross-Site Scripting (XSS) is a security vulnerability that enables a cyberattacker to place client side scripts (usually JavaScript) into web pages. When other users load affected pages, the cyberattacker's scripts run, enabling the cyberattacker to steal cookies and session tokens, change the contents of the web page through DOM manipulation, or redirect the browser to another page. XSS vulnerabilities generally occur when an application takes user input and outputs it to a page without validating, encoding or escaping it.

This article applies primarily to ASP.NET Core MVC with views, Razor Pages, and other apps that return HTML that may be vulnerable to XSS. Web APIs that return data in the form of HTML, XML, or JSON can trigger XSS attacks in their client apps if they don't properly sanitize user input, depending on how much trust the client app places in the API. For example, if an API accepts user-generated content and returns it in an HTML response, a cyberattacker could inject malicious scripts into the content that executes when the response is rendered in the user's browser.

To prevent XSS attacks, web APIs should implement input validation and output encoding. Input validation ensures that user input meets expected criteria and doesn't include malicious code. Output encoding ensures that any data returned by the API is properly sanitized so that it can't be executed as code by the user's browser. For more information, see this GitHub issue.

Protecting your application against XSS

At a basic level, XSS works by tricking your application into inserting a `<script>` tag into your rendered page, or by inserting an `On*` event into an element. Developers should use the following prevention steps to avoid introducing XSS into their applications:

1. Never put untrusted data into your HTML input, unless you follow the rest of the steps below. Untrusted data is any data that may be controlled by a cyberattacker, such as HTML form inputs, query strings, HTTP headers, or even data sourced from a database, as a cyberattacker may be able to breach your database even if they can't breach your application.

2. Before putting untrusted data inside an HTML element, ensure it's HTML encoded. HTML encoding takes characters such as < and changes them into a safe form like <

3. Before putting untrusted data into an HTML attribute, ensure it's HTML encoded. HTML attribute encoding is a superset of HTML encoding and encodes additional characters such as " and ".

4. Before putting untrusted data into JavaScript, place the data in an HTML element whose contents you retrieve at runtime. If this isn't possible, then ensure the data is JavaScript encoded. JavaScript encoding takes dangerous characters for JavaScript and replaces them with their hex, for example, < would be encoded as `\u003C`.

5. Before putting untrusted data into a URL query string ensure it's URL encoded.

HTML Encoding using Razor

The Razor engine used in MVC automatically encodes all output sourced from variables, unless you work really hard to prevent it doing so. It uses HTML attribute encoding rules whenever you use the @ directive. As HTML attribute encoding is a superset of HTML encoding this means you don't have to concern yourself with whether you should use HTML encoding or HTML attribute encoding. You must ensure that you only use @ in an HTML context, not when attempting to insert untrusted input directly into JavaScript. Tag helpers will also encode input you use in tag parameters.

Take the following Razor view:

CSHTML

```
@{
    var untrustedInput = "<\"123\">";
}

@untrustedInput
```

This view outputs the contents of the *untrustedInput* variable. This variable includes some characters which are used in XSS attacks, namely <, " and >. Examining the source shows the rendered output encoded as:

HTML

```
&lt;&quot;123&quot;&gt;
```

> ⚠ **Warning**
>
> ASP.NET Core MVC provides an `HtmlString` class which isn't automatically encoded upon output. This should never be used in combination with untrusted input as this will expose an XSS vulnerability.

JavaScript Encoding using Razor

There may be times you want to insert a value into JavaScript to process in your view. There are two ways to do this. The safest way to insert values is to place the value in a data attribute of a tag and retrieve it in your JavaScript. For example:

CSHTML

```
@{
    var untrustedInput = "<script>alert(1)</script>";
}

<div id="injectedData"
     data-untrustedinput="@untrustedInput" />

<div id="scriptedWrite" />
<div id="scriptedWrite-html5" />

<script>
    var injectedData = document.getElementById("injectedData");

    // All clients
    var clientSideUntrustedInputOldStyle =
        injectedData.getAttribute("data-untrustedinput");

    // HTML 5 clients only
    var clientSideUntrustedInputHtml5 =
        injectedData.dataset.untrustedinput;

    // Put the injected, untrusted data into the scriptedWrite div tag.
    // Do NOT use document.write() on dynamically generated data as it
    // can lead to XSS.

    document.getElementById("scriptedWrite").innerText +=
clientSideUntrustedInputOldStyle;

    // Or you can use createElement() to dynamically create document
elements
    // This time we're using textContent to ensure the data is properly
encoded.
    var x = document.createElement("div");
    x.textContent = clientSideUntrustedInputHtml5;
    document.body.appendChild(x);
```

```
    // You can also use createTextNode on an element to ensure data is
properly encoded.
    var y = document.createElement("div");
    y.appendChild(document.createTextNode(clientSideUntrustedInputHtml5));
    document.body.appendChild(y);

</script>
```

The preceding markup generates the following HTML:

HTML

```
<div id="injectedData"
     data-untrustedinput="&lt;script&gt;alert(1)&lt;/script&gt;" />

<div id="scriptedWrite" />
<div id="scriptedWrite-html5" />

<script>
    var injectedData = document.getElementById("injectedData");

    // All clients
    var clientSideUntrustedInputOldStyle =
        injectedData.getAttribute("data-untrustedinput");

    // HTML 5 clients only
    var clientSideUntrustedInputHtml5 =
        injectedData.dataset.untrustedinput;

    // Put the injected, untrusted data into the scriptedWrite div tag.
    // Do NOT use document.write() on dynamically generated data as it can
    // lead to XSS.

    document.getElementById("scriptedWrite").innerText +=
clientSideUntrustedInputOldStyle;

    // Or you can use createElement() to dynamically create document
elements
    // This time we're using textContent to ensure the data is properly
encoded.
    var x = document.createElement("div");
    x.textContent = clientSideUntrustedInputHtml5;
    document.body.appendChild(x);

    // You can also use createTextNode on an element to ensure data is
properly encoded.
    var y = document.createElement("div");
    y.appendChild(document.createTextNode(clientSideUntrustedInputHtml5));
    document.body.appendChild(y);

</script>
```

The preceding code generates the following output:

```
<script>alert(1)</script>
<script>alert(1)</script>
<script>alert(1)</script>
```

> ⚠ **Warning**
>
> Do ***NOT*** concatenate untrusted input in JavaScript to create DOM elements or use `document.write()` on dynamically generated content.
>
> Use one of the following approaches to prevent code from being exposed to DOM-based XSS:
>
> - `createElement()` and assign property values with appropriate methods or properties such as `node.textContent=` or `node.InnerText=`.
> - `document.CreateTextNode()` and append it in the appropriate DOM location.
> - `element.SetAttribute()`
> - `element[attribute]=`

Accessing encoders in code

The HTML, JavaScript and URL encoders are available to your code in two ways:

- Inject them via dependency injection.
- Use the default encoders contained in the `System.Text.Encodings.Web` namespace.

When using the default encoders, then any customizations applied to character ranges to be treated as safe won't take effect. The default encoders use the safest encoding rules possible.

To use the configurable encoders via DI your constructors should take an *HtmlEncoder*, *JavaScriptEncoder* and *UrlEncoder* parameter as appropriate. For example;

C#

```
public class HomeController : Controller
{
    HtmlEncoder _htmlEncoder;
    JavaScriptEncoder _javaScriptEncoder;
    UrlEncoder _urlEncoder;
```

```
    public HomeController(HtmlEncoder htmlEncoder,
                          JavaScriptEncoder javascriptEncoder,
                          UrlEncoder urlEncoder)
    {
        _htmlEncoder = htmlEncoder;
        _javaScriptEncoder = javascriptEncoder;
        _urlEncoder = urlEncoder;
    }
}
```

Encoding URL Parameters

If you want to build a URL query string with untrusted input as a value use the `UrlEncoder` to encode the value. For example,

C#

```
var example = "\"Quoted Value with spaces and &\"";
var encodedValue = _urlEncoder.Encode(example);
```

After encoding the encodedValue variable contains `%22Quoted%20Value%20with%20spaces%20and%20%26%22`. Spaces, quotes, punctuation and other unsafe characters are percent encoded to their hexadecimal value, for example a space character will become %20.

> ⚠ **Warning**
>
> Don't use untrusted input as part of a URL path. Always pass untrusted input as a query string value.

Customizing the Encoders

By default encoders use a safe list limited to the Basic Latin Unicode range and encode all characters outside of that range as their character code equivalents. This behavior also affects Razor TagHelper and HtmlHelper rendering as it uses the encoders to output your strings.

The reasoning behind this is to protect against unknown or future browser bugs (previous browser bugs have tripped up parsing based on the processing of non-English characters). If your web site makes heavy use of non-Latin characters, such as Chinese, Cyrillic or others this is probably not the behavior you want.