PumaPrey.sln

Source Code Scan Results

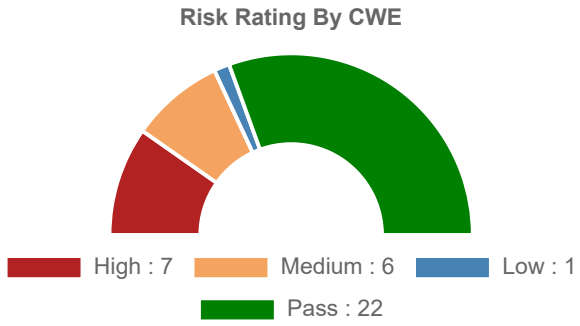
3/6/2020 9:53:19 PM

Scan Time: 0 min 08.47 sec

Engine Version: 0.9.7.2

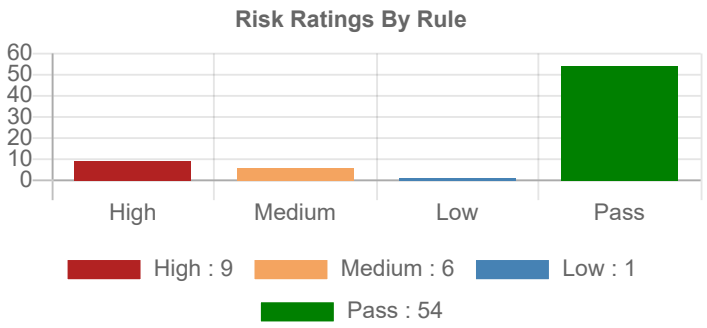
Common Weakness Enumeration (CWE)

	CWE Identifier	Instances
!	CWE-23: Relative Path Traversal	1
!	CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	1
!	CWE-295: Improper Certificate Validation	1
!	CWE-327: Use of a Broken or Risky Cryptographic Algorithm	3
!	CWE-502: Deserialization of Untrusted Data	2
!	CWE-601: URL Redirection to Untrusted Site	1
!	CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	1
⚠	CWE-306: Missing Authentication for Critical Function	3
⚠	CWE-307: Improper Restriction of Excessive Authentication Attempts	1
⚠	CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	1
⚠	CWE-352: Cross-Site Request Forgery (CSRF)	8
⚠	CWE-521: Weak Password Requirements	2
⚠	CWE-598: Information Exposure Through Query Strings in GET Request	1
!	CWE-20: Improper Input Validation	1
	Total	**27**



Puma Scan Rules

	Rule Identifier	Instances
!	SEC0003: Forms Authentication: Insecure Cookie	1
!	SEC0025: Weak Symmetric Algorithm	2
!	SEC0027: Weak Algorithm: MD5	1
!	SEC0029: Insecure Deserialization	1
!	SEC0030: Insecure Deserialization: Newtonsoft JSON	1
!	SEC0108: SQL Injection: Dynamic EF Query	1
!	SEC0109: Unvalidated MVC Redirect	1
!	SEC0111: Path Tampering: MVC File Result	1
!	SEC0113: Certificate Validation Disabled	1
⚠	SEC0004: Forms Authentication: Cookieless Mode	1
⚠	SEC0017: Identity Weak Password Complexity	2
⚠	SEC0018: Identity Password Lockout Disabled	1
⚠	SEC0019: Missing AntiForgery Token Attribute	8
⚠	SEC0115: Insecure Random Number Generator	1
⚠	SEC0120: Missing Authorize Attribute	3
!	SEC0023: Action Request Validation Disabled	1
	Total	**27**



Scan Details

SEC0003: Forms Authentication: Insecure Cookie

Description

Authentication cookies sent over HTTP connections can be stolen by attackers monitoring the network traffic, which can lead to session hijacking attacks.

Configure **secure** cookies by setting the requireSSL attribute to **true**.

References

- CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
- https://msdn.microsoft.com/en-us/library/1d3t3c61(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the requireSSL attribute is false.

```
 <system.web>
    ...
    <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx" timeout="30" />
        </authentication>
    ...
</system.web>
```

Secure Configuration: Explicitly set the requireSSL value to true.

```
<system.web>
    ...
    <authentication mode="Forms">
      <forms loginUrl="~/Account/Login.aspx" requireSSL="true"/>
    </authentication>
    ...
</system.web>
```

Scan Results

Skunk.csproj | Web.config:

```
19: <forms requireSSL="false"></forms>
```

SEC0025: Weak Symmetric Algorithm

Description

The DES, TripleDES, and RC2 classes use weak encryption algorithms and not considered secure for protecting sensitive information.

Use the AesManaged or AesCryptoServiceProvider algorithm (aka Rijndael) for symmetric encryption operations.

References

- CWE-327: Use of a Broken or Risky Cryptographic Algorithm
- http://csrc.nist.gov/groups/ST/toolkit/block_ciphers.html
- https://msdn.microsoft.com/en-us/library/system.security.cryptography.aesmanaged(v=vs.110).aspx

Code Example(s)

Insecure Code: The following example shows the DESCryptoServiceProvider algorithm being used to encrypt a sensitive piece of data.

```
using (MemoryStream mStream = new MemoryStream())
{
    //Input bytes
    byte[] inputBytes = Encoding.UTF8.GetBytes(plainText);

    SymmetricAlgorithm alg = new DESCryptoServiceProvider();

    //Set key and iv
    alg.Key = GetKey();
    alg.IV = GetIv();

    //Create the crypto stream
    CryptoStream cStream = new CryptoStream(mStream, alg.CreateEncryptor(), CryptoStreamMode.Write);
    cStream.Write(inputBytes, 0, inputBytes.Length);
    cStream.FlushFinalBlock();
    cStream.Close();

    //Get the output
    output = mStream.ToArray();

    //Close resources
    mStream.Close();
    alg.Clear();
}
```

Secure Code: Use the AesManaged algorithm for symmetric encryption.

```
using (MemoryStream mStream = new MemoryStream())
{
    //Input bytes
    byte[] inputBytes = Encoding.UTF8.GetBytes(plainText);

    SymmetricAlgorithm alg = new AesManaged();

    //Set key and iv
```

```
    alg.Key = GetKey();
    alg.IV = GetIv();

    //Create the crypto stream
    CryptoStream cStream = new CryptoStream(mStream, alg.CreateEncryptor(), CryptoStreamMode.Write);
    cStream.Write(inputBytes, 0, inputBytes.Length);
    cStream.FlushFinalBlock();
    cStream.Close();

    //Get the output
    output = mStream.ToArray();

    //Close resources
    mStream.Close();
    alg.Clear();
}
```

Scan Results

Common.csproj | Cryptography\Encryption.cs:

```
51: var crypto = new DESCryptoServiceProvider
```

Common.csproj | Cryptography\Encryption.cs:

```
106: var crypto = new DESCryptoServiceProvider
```

SEC0027: Weak Algorithm: MD5

Description

The MD5CryptoServiceProvider class uses the weak MD5 algorithm and is not an approved hashing algorithm.

Use the SHA256Managed (at least) preferably SHA512Managed for hashing operations.

ℹ This alone is not sufficient for password hashing, which requires a unique salt and an adaptive hashing algorithm / iterations. See the references below for more information on password hashing in .NET.

References

- CWE-327: Use of a Broken or Risky Cryptographic Algorithm
- https://msdn.microsoft.com/en-us/library/system.security.cryptography.md5cryptoserviceprovider(v=vs.110).aspx
- https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha256(v=vs.110).aspx
- https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha512(v=vs.110).aspx
- https://blogs.dropbox.com/tech/2016/09/how-dropbox-securely-stores-your-passwords/

Code Example(s)

Insecure Code: The following example shows the MD5 algorithm hashing a piece of data.

```
HashAlgorithm hash = new MD5CryptoServiceProvider();
byte[] bytes = hash.ComputeHash(input);
```

Secure Code: Use SHA256Managed or SHA512Managed for hashing operations (note: this is not enough for storing passwords).

```
HashAlgorithm hash = new SHA512Managed();
byte[] bytes = hash.ComputeHash(input);
```

Scan Results

Common.csproj | Cryptography\Hashing.cs:

```
47: HashAlgorithm hash = new MD5CryptoServiceProvider();
```

SEC0029: Insecure Deserialization

Description

Deserializing untrusted data using vulnerable libraries can allow attackers to execute arbitrary code and perform denial of service attacks against the server.
SEC0029 covers over 25 .NET libraries vulnerable to deserialization attacks. The most commonly used libraries are as follows:

- JavaScriptSerializer
- DataContractJsonSerializer
- BinaryFormatter
- SoapFormatter
- ObjectStateFormatter
- NetDataContractSerializer
- LosFormatter
- And many more...

Avoid deserializing untrusted data (e.g. request parameters, web service parameters, data from external services) using the above dangerous methods. In cases where deserialization is required, ensure that the application performs signature validation (e.g. HMAC) before deserializing the data.

The SEC0029 analyzer currently looks for insecure function calls. Data flow analysis is not yet performed to determine if the binary content being parsed is actually controllable by an attacker.

References

- CWE-502: Deserialization of Untrusted Data
- https://www.owasp.org/index.php/Top_10-2017_A8-Insecure_Deserialization
- https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf
- https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter
- https://docs.microsoft.com/en-us/dotnet/api/system.web.script.serialization.javascriptserializer
- https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.json.datacontractjsonserializer
- https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.soap.soapformatter
- https://docs.microsoft.com/en-us/dotnet/api/system.web.ui.objectstateformatter
- https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.netdatacontractserializer
- https://docs.microsoft.com/en-us/dotnet/api/system.web.ui.losformatter

Code Example(s)

Insecure Code: The following example shows the BinaryFormatter Deserialize method deserializing a byte array without HMAC signature validation.

```
using (MemoryStream stream = new MemoryStream())
{
    byte[] bytes = Convert.FromBase64String(model.UserData);
    stream.Write(bytes, 0, bytes.Length);
    BinaryFormatter formatter = new BinaryFormatter();
    return (User)formatter.Deserialize(stream);
}
```

Secure Code: Ensure that the application performs signature validation (e.g. HMAC) before deserializing the data. This snippet assumes that the serialized data has been concatenated with the signature (joined using a '.') and sent to the client.

```
//Validate user data signature
string[] args = model.UserData.Split('.');
byte[] userData = Convert.FromBase64String(args[0]);
byte[] hmac = Convert.FromBase64String(args[1]);

HMACSHA256 hmac = new HMACSHA256(_KEY);
byte[] verification = hmac.ComputeHash(userData);

if (!verification.SequenceEqual(hmac))
    throw new ArgumentException("Invalid signature detected.");

//If valid, process the deserialization
using (MemoryStream stream = new MemoryStream())
{
    byte[] bytes = Convert.FromBase64String(userData);
    stream.Write(bytes, 0, bytes.Length);
    BinaryFormatter formatter = new BinaryFormatter();
    return (User)formatter.Deserialize(stream);
}
```

Scan Results

Common.csproj | Deserialize\BinaryDeserialize.cs:

```
20: return (T)formatter.Deserialize(stream);
```

SEC0030: Insecure Deserialization: Newtonsoft JSON

Description

The Newtonsoft JSON DeserializeObject method can allow attackers to execute arbitrary code and perform denial of service attacks if the *TypeNameHandling* setting is set to a value other than None.

Validate incoming data with HMAC protection to ensure it has not been modified on the client. Explicitly set the TypeNameHandling setting to None (which is the default) to prevent JSON deserialization vulnerabilities.

References

- CWE-502: Deserialization of Untrusted Data
- https://www.owasp.org/index.php/Top_10-2017_A8-Insecure_Deserialization
- http://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_TypeNameHandling.htm

Code Example(s)

Insecure Code: The following example shows the *model.UserData* request parameter being deserialized with Newtonsoft JSON serialization TypeNameHandling enumeration set to All.

```
JsonSerializerSettings settings = new JsonSerializerSettings();
settings.TypeNameHandling = TypeNameHandling.Auto;
return JsonConvert.DeserializeObject<User>(model.UserData, settings);
```

Secure Code: Validate incoming data with HMAC protection to ensure it has not been modified on the client. Explicitly set the TypeNameHandling setting to None (which is the default) to prevent JSON deserialization vulnerabilities.

```
//Validate user data signature
string[] args = json.Split('.');
byte[] userData = Convert.FromBase64String(args[0]);
byte[] hash = Convert.FromBase64String(args[1]);

HMACSHA256 hmac = new HMACSHA256(_KEY);
byte[] verification = hmac.ComputeHash(userData);

if (!verification.SequenceEqual(hash))
    throw new ArgumentException("Invalid signature detected.");

JsonSerializerSettings settings = new JsonSerializerSettings();
settings.TypeNameHandling = TypeNameHandling.None;
return JsonConvert.DeserializeObject<T>(Encoding.UTF8.GetString(userData), settings);
```

Scan Results

Common.csproj | Deserialize\JsonDeserialize.cs:

```
16: TypeNameHandling = TypeNameHandling.All
```

SEC0108: SQL Injection: Dynamic EF Query

Description

Concatenating untrusted data into a dynamic SQL string and calling vulnerable Entity Framework (EF) methods can allow SQL Injection:

- ExecuteSqlCommand
- ExecuteSqlCommandAsync
- SqlQuery
- FromSql

To ensure calls to vulnerable EF methods are parameterized, pass parameters into the statement using the method's second argument: *params object[] parameters*.

References

- CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- https://www.owasp.org/index.php/Top_10-2017_A1-Injection
- https://msdn.microsoft.com/en-us/library/system.data.entity.database(v=vs.113).aspx
- https://docs.microsoft.com/en-us/ef/core/querying/raw-sql

Code Example(s)

Insecure Code: The following example shows dynamic SQL passed to the ExecuteSqlCommand Entity Database method.

```
using (DbDataContext context = new DbDataContext())
{
    string q = "DELETE FROM Items WHERE ProductCode = '" + model.ProductCode + "'";
    context.Database.ExecuteSqlCommand(q);
}
```

Secure Code: Call the overloaded ExecuteSqlCommand method that accepts a list of parameters in the second argument.

```
using (DbDataContext context = new DbDataContext())
{
    string q = "DELETE FROM Items WHERE ProductCode = @productCode";
    context.Database.ExecuteSqlCommand(q, model.ProductCode);
}
```

Scan Results

Fox.csproj | Controllers\HuntController.cs:

```
67: context.Database.ExecuteSqlCommand(q);
```

SEC0109: Unvalidated MVC Redirect

Description

Passing unvalidated redirect locations to the MVC *Controller.Redirect* method can allow attackers to send users to malicious web sites. This can allow attackers to perform phishing attacks and distribute malware to victims.

Avoid performing redirect actions with user controllable data (e.g. request parameters). Consider validating redirect paths to allow relative paths inside of the application and deny absolute paths. All absolute paths must be validated against an approved list of external domains prior to redirecting the user.

References

- CWE-601: URL Redirection to Untrusted Site ('Open Redirect')
- https://msdn.microsoft.com/en-us/library/system.web.mvc.controller.redirect(v=vs.118).aspx

Code Example(s)

Insecure Code: The following example shows the Controller Redirect method called using an unvalidated request parameter.

```
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    [perform auth logic]

    return this.Redirect(model.ReturnUrl);
}
```

Secure Code: Validate the return URL is a local path (not absolute) to ensure an attacker cannot redirect the user to a malicious external domain.

```
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    [perform auth logic]

    if (Url.IsLocalUrl(returnUrl))
    {
        return this.Redirect(model.ReturnUrl);
    }
    else
    {
        return RedirectToAction("Index", "Home");
    }
}
```

Scan Results

Raccoon.csproj | Controllers\AccountController.cs:

```
449: return Redirect(returnUrl);
```

SEC0111: Path Tampering: MVC File Result

Description

Path traversal vulnerabilities occur when an application does not properly validate file paths for directory traversal (../) and other malicious characters. This can allow attackers to download, overwrite, or delete unauthorized files from the server. Ensure file paths are read from a trusted location, such as a static resource or configuration file. Do not send file paths in request parameters, which can be modified by an attacker.

The ASP.NET MVC *FilePathResult* and *FileStreamResult* actions are used to stream file content to the browser.

Failing to validate the file path used by these actions can allow path traversal vulnerabilities. Ensure that all user input is properly validated and sanitized before it is passed to the file API.

References

- CWE-23: Relative Path Traversal
- https://msdn.microsoft.com/en-us/library/system.web.mvc.filepathresult(v=vs.118).aspx
- https://docs.microsoft.com/en-us/previous-versions/aspnet/web-frameworks/dd492941(v=vs.118)

Code Example(s)

Insecure Code: The following example shows the *FilePathResult* action result using the *fileName* request parameter to construct the file path location.

```
[HttpPost]
public FileResult Download(string fileName)
{
    string filePath = ConfigurationManager.AppSettings["DownloadDirectory"].ToString();
    return new FilePathResult(filePath + fileName, "application/octet-stream");
}
```

Secure Code: Avoid using untrusted values when constructing a file path. Instead, store file paths in a trusted location, such as a configuration file, and use a unique identifier to construct the file name.

```
[HttpPost]
public FileResult Download(Guid fileId)
{
    string filePath = ConfigurationManager.AppSettings["DownloadDirectory"].ToString();
    filePath = string.Format("{0}{1}.pdf", filePath, fileId.ToString());
    return new FilePathResult(filePath, "application/octet-stream");
}
```

Scan Results

Raccoon.csproj | Controllers\HuntController.cs:

```
92: return new FilePathResult("C:\\share\\reports\\" + report.Name, "application/octet-stream");
```

SEC0113: Certificate Validation Disabled

Description

Disabling certificate validation is common in testing and development environments. Quite often, this is accidentally deployed to production, leaving the application vulnerable to man-in-the-middle attacks on insecure networks.

Do not override the *ServicePointManager.ServerCertificateValidationCallback* method to always return true. If the development team is using self-signed certificates, use certificate pinning to ensure the application will only communicate with the trusted server certificate.

References

- CWE-295: Improper Certificate Validation
- https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning
- https://msdn.microsoft.com/en-us/library/system.net.servicepointmanager.servercertificatevalidationcallback(v=vs.110).aspx

Code Example(s)

Insecure Code: The following example shows the *ServicePointManager.ServerCertificateValidationCallback* method being overridden to always return a *true* value.

```
using (var handler = new WebRequestHandler())
{
    handler.ServerCertificateValidationCallback += (sender, cert, chain, sslPolicyErrors) => true;

    using (var client = new HttpClient(handler))
    {
        var request = client.GetAsync(string.Format("{0}{1}", BASE_URL, endpoint)).ContinueWith((response) =>
            {
                var result = response.Result;
                var json = result.Content.ReadAsStringAsync();
                json.Wait();
                item = JsonConvert.DeserializeObject<T>(json.Result);
            }
        );
        request.Wait();
    }
}
```

Secure Code: This one is too easy. Don't override the frameworks certificate validation to always return true.

```
using (var handler = new WebRequestHandler())
{
    using (var client = new HttpClient(handler))
    {
        var request = client.GetAsync(string.Format("{0}{1}", BASE_URL, endpoint)).ContinueWith((response) =>
            {
                var result = response.Result;
                var json = result.Content.ReadAsStringAsync();
                json.Wait();
                item = JsonConvert.DeserializeObject<T>(json.Result);
            }
```

```
        );
        request.Wait();
    }
}
```

Scan Results

Common.csproj | Rest\RestClient.cs:

```
17: handler.ServerCertificateValidationCallback += (sender, cert, chain, sslPolicyErrors) => true;
```

⚠ SEC0004: Forms Authentication: Cookieless Mode

Description

Authentication cookies should not be sent in the URL. Doing so allows attackers to gain unauthorized access to authentication tokens (web server logs, referrer headers, and browser history) and more easily perform session fixation / hijacking attacks.

Configure cookie-based authentication by setting the *cookieless* attribute to **UseCookies**.

References

- CWE-598: Information Exposure Through Query Strings in GET Request
- https://msdn.microsoft.com/en-us/library/1d3t3c61(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the cookieless attribute is UseDeviceProfile, which can allow URL-based authentication tokens.

```
<system.web>
    ...
    <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx" />
    </authentication>
    ...
</system.web>
```

Secure Configuration: Explicitly set the cookieless attribute to UseCookies.

```
<system.web>
    ...
    <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx" cookieless="UseCookies" />
    </authentication>
    ...
</system.web>
```

Scan Results

Skunk.csproj | Web.config:

```
19: <forms requireSSL="false"></forms>
```

⚠ SEC0017: Identity Weak Password Complexity

Description

Weak passwords can allow attackers to easily guess user passwords using wordlist or brute force attacks. Enforcing a strict password complexity policy mitigates these attacks by significantly increasing the time to guess a user's valid password.

The following .NET Framework and Core password complexity classes are supported by SEC0017:

- Microsoft.AspNet.Identity.PasswordValidator
- Microsoft.AspNetCore.Identity.PasswordOptions

Out of the box, Visual Studio's template for ASP.NET Identity-based authentication requires only 6 characters. Increasing the minimum length from 6 to a value greater than 12 characters will significantly increase the strength of the passwords stored by the application.

Requiring all of the character types (upper, lower, numeric, and special) increases the computational power to crack passwords.

References

- CWE-521: Weak Password Requirements
- https://msdn.microsoft.com/en-us/library/microsoft.aspnet.identity.passwordvalidator(v=vs.108).aspx
- https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.passwordoptions

Code Example(s)

Insecure Code: The following example shows the *ApplicationUserManager* class configuring the *PasswordValidator* with a weak password complexity policy. This configuration, which is the default policy generated by Visual Studio, sets a minimum password length of 6 characters.

```
// Configure validation logic for passwords
manager.PasswordValidator = new PasswordValidator
{
    RequiredLength = 6
};
```

Secure Code: Update the *PasswordValidator* configuration to meet your organization's password complexity requirements. The following example shows a required length of 12 characters, along with at least one digit, upper, lower, and special character.

```
// Configure validation logic for passwords
manager.PasswordValidator = new PasswordValidator
{
    RequiredLength = 12,
    RequireDigit = true,
    RequireLowercase = true,
    RequireNonLetterOrDigit = true,
    RequireUppercase = true
};
```

Scan Results

Fox.csproj | App_Start\IdentityConfig.cs:

```
29: manager.PasswordValidator = new PasswordValidator
```

Raccoon.csproj | App_Start\IdentityConfig.cs:

```
54: manager.PasswordValidator = new PasswordValidator
```

⚠ SEC0018: Identity Password Lockout Disabled

Description

Password lockout mechanisms help prevent continuous brute force attacks again user accounts by disabling an account for a period of time after a number of invalid attempts.

The ASP.NET Identity SignInManager protects against brute force attacks if the lockout parameter is set to true.

References

- CWE-307: Improper Restriction of Excessive Authentication Attempts
- https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.signinmanager-1.passwordsigninasync
- https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.signinmanager-1.checkpasswordsigninasync

Code Example(s)

Insecure Code: The following example shows the *SignInManager.PasswordSignInAsync* method passing *false* in the *lockoutOnFailure* parameter. This disables the Identity password lockout mechanism.

```
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    var user = await _userManager.FindByNameAsync(model.Username);
    [...]
    var result = await _signInManager.CheckPasswordSignInAsync(user, model.Password, false);
    [...]
};
```

Secure Code: Change the *lockoutOnFailure* parameter value to *true*.

```
{
    var user = await _userManager.FindByNameAsync(model.Username);
    [...]
    var result = await _signInManager.CheckPasswordSignInAsync(user, model.Password, true);
    [...]
};
```

Scan Results

Raccoon.csproj | Controllers\AccountController.cs:

```
79: model.Password, model.RememberMe, shouldLockout: false);
```

⚠ SEC0019: Missing AntiForgery Token Attribute

Description

Cross Site Request Forgery attacks occur when a victim authenticates to a target web site and then visits a malicious web page. The malicious web page then sends a fake HTTP request (GET, POST, etc.) back to the target website. The victim's valid authentication cookie from the target web site is automatically included in the malicious request, sent to the target web site, and processed as a valid transaction under the victim's identity.

This rule searches for all actions decorated with HTTP verbs that typically modify data (POST, PUT, DELETE, and PATCH). Actions containing the [AllowAnonymous] attribute are not reported as CSRF attacks target authenticated users. Any identified actions that are missing the *ValidateAntiForgeryToken* attribute raise a diagnostic warning.

In ASP.NET MVC, the *ValidateAntiForgeryToken* attribute protects applications using authentication cookies from CSRF attacks. Actions with this attribute search the request parameters for the __RequestVerificationToken and validate the value prior to executing the request.

References

- CWE-352: Cross-Site Request Forgery (CSRF)
- https://msdn.microsoft.com/en-us/library/system.web.mvc.validateantiforgerytokenattribute(v=vs.118).aspx

Code Example(s)

Insecure Code: The following example shows the *Enter* action performing a transaction without using the *ValidateAntiForgeryToken* attribute.

```
[HttpPost]
public ActionResult Enter(int id, ContestEntryModel model)
{
    if (ModelState.IsValid)
    {
        submitContestEntry(id, model);
    }
}
```

Secure Code: By adding the *ValidateAntiForgeryToken* attribute, the application will validate the *__RequestVerificationToken* request parameter before entering the contest.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Enter(int id, ContestEntryModel model)
{
    if (ModelState.IsValid)
    {
        submitContestEntry(id, model);
    }
}
```

Scan Results

Fox.csproj | Controllers\HuntController.cs:

```
52: public void Post([FromBody]string value)
```

Fox.csproj | Controllers\HuntController.cs:

```
57: public void Put(int id, [FromBody]string value)
```

Fox.csproj | Controllers\HuntController.cs:

```
62: public void Delete(string id)
```

Fox.csproj | Controllers\ValuesController.cs:

```
26: public void Post([FromBody]string value)
```

Fox.csproj | Controllers\ValuesController.cs:

```
31: public void Put(int id, [FromBody]string value)
```

Fox.csproj | Controllers\ValuesController.cs:

```
36: public void Delete(int id)
```

Raccoon.csproj | Controllers\HuntController.cs:

```
58: public ActionResult Enter(int id, HuntModel model)
```

Raccoon.csproj | Controllers\HuntController.cs:

```
77: public ActionResult Download(string fileName)
```

⚠ SEC0115: Insecure Random Number Generator

Description

System.Random is a statistical random number generator that does not generate sufficiently random values for use in a security context.

Generate values used in a security context (e.g., encryption keys, initialization vectors, random passwords, authentication tokens) using the System.Security.Cryptography.RNGCryptoServiceProvider.

References

- CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)
- https://msdn.microsoft.com/en-us/library/system.security.cryptography.rngcryptoserviceprovider(v=vs.110).aspx

Code Example(s)

Insecure Code: The following example shows the *System.Random* algorithm creating a random value.

```
public static byte[] GenerateRandomBytes(int length)
{
    var random = new Random();
    byte[] bytes = new byte[length];
    random.NextBytes(bytes);
    return bytes;
}
```

Secure Code: Use the *RNGCryptoServiceProvider* class to create random values for use in security operations.

```
public static byte[] GenerateSecureRandomBytes(int length)
{
    var random = new RNGCryptoServiceProvider();
    byte[] bytes = new byte[length];
    random.GetNonZeroBytes(bytes);
    return bytes;
}
```

Scan Results

Common.csproj | Cryptography\Random.cs:

```
16: System.Random random = new System.Random();
```

⚠ SEC0120: Missing Authorize Attribute

Description

Missing Authorization occurs when an application does not properly verify an authenticated user's access to functionality, data, or resources. In many cases, applications do not check policy, claim, or role-based access control rules during a request. This can allow attackers to invoke privileged functionality, such as changing their role or directly browsing to an administrative interface in the application.

Access control is managed in ASP.NET MVC, Web API, and .NET Core Controllers and Actions using the **Authorize** attribute. By default, Actions missing the Authorize attribute can be invoked by anonymous users. Review the SEC0120 warnings and determine if the Authorize attribute needs to be present to protect against anonymous access.

References

- CWE-306: Missing Authentication for Critical Function
- https://www.owasp.org/index.php/Top_10-2017_A2-Broken_Authentication
- https://www.owasp.org/index.php/Top_10-2017_A5-Broken_Access_Control
- https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authorization.authorizeattribute
- https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.authorizeattribute

Code Example(s)

Insecure Code: The following example shows the *Enter* action missing the *Authorization* attribute.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Enter(int id, ContestEntryModel model)
{
    if (ModelState.IsValid)
    {
        submitContestEntry(id, model);
    }
}
```

Secure Code: By adding the *Authorization* attribute with a valid policy, the application restricts access to users meeting the *EnterContest* permissions.

```
[Authorize(Policy = Scopes.EnterContent)]
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Enter(int id, ContestEntryModel model)
{
    if (ModelState.IsValid)
    {
        submitContestEntry(id, model);
    }
}
```

Scan Results

Fox.csproj | Controllers\HuntController.cs:

```
52: public void Post([FromBody]string value)
```

Fox.csproj | Controllers\HuntController.cs:

```
57: public void Put(int id, [FromBody]string value)
```

Fox.csproj | Controllers\HuntController.cs:

```
62: public void Delete(string id)
```

❗ SEC0023: Action Request Validation Disabled

Description

Request validation performs blacklist input validation for XSS payloads found in form and URL request parameters. Request validation has known bypass issues and does not prevent all XSS attacks, but it does provide a strong countermeasure for most payloads targeting a HTML context.

Request validation is enabled by default during model binding to dynamic HTML request parameters, but can be disabled on controllers and actions properties using the **ValidateInput(false)** attribute. The following countermeasures can help validate data and filter XSS payloads:

- Avoid accepting HTML input from untrusted data sources. Leave request validation enabled and consider accepting a different data format, such as markdown.
- Create a custom action filter that sanitizes request parameters containing HTML using the AntiXss HTML Sanitizer class to strip potentially dangerous script from untrusted data before consuming the data.

ℹ Version 4.3.0 is the recommended HTML sanitizer library version

References

- CWE-20: Improper Input Validation
- https://msdn.microsoft.com/en-us/library/system.web.mvc.validateinputattribute.aspx
- https://wpl.codeplex.com/

Code Example(s)

Insecure Code: The following example shows the *Save* action using the *ValidateInput(false)* attribute to disable request validation.

```
[HttpPost]
[ValidateInput(false)]
public ActionResult Save(int id, ProductFeedbackModel model)
{

}
```

Secure Code: The following example removes the *ValidateInput(false)* attribute to enable request validation.

```
[HttpPost]
public ActionResult Save(int id, ProductFeedbackModel model)
{

}
```

Scan Results

Raccoon.csproj | Controllers\HuntController.cs:

```
57: [ValidateInput(false)]
```

✔ SEC0001: Debug Build Enabled

Description

Binaries compiled in debug mode can leak detailed stack traces and debugging messages to attackers.

Disable debug builds by setting the debug attribute to false.

References

- CWE-11: ASP.NET Misconfiguration: Creating Debug Binary
- https://msdn.microsoft.com/en-us/library/s10awwz0(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the debug attribute is false.

```
<system.web>
    ...
    <compilation debug="true" targetFramework="4.5"/>
    ...
</system.web>
```

Secure Configuration: Remove the debug attribute, or explicitly set the value to false.

```
<system.web>
    ...
    <compilation debug="false" targetFramework="4.5"/>
    ...
</system.web>
```

Scan Results

SEC0001 ran successfully without any identified instances.

✔ SEC0002: Custom Errors Disabled

Description

Displaying stack traces in the browser can leak information to attackers and help them gain information for launching additional attacks.

Enable custom errors by setting the mode to On or RemoteOnly.

References

- CWE-12: ASP.NET Misconfiguration: Missing Custom Error Page
- https://msdn.microsoft.com/en-us/library/h0hfz6fc(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the mode attribute is RemoteOnly.

```
<system.web>
    ...
    <customErrors mode="Off" defaultRedirect="/home/error"/>
    ...
</system.web>
```

Secure Configuration: Explicitly set the mode value to RemoteOnly or On, or simply remove the mode attribute.

```
<system.web>
    ...
    <customErrors mode="RemoteOnly|On" defaultRedirect="/home/error"/>
    ...
</system.web>
```

Scan Results

SEC0002 ran successfully without any identified instances.

✔ SEC0005: Forms Authentication: Cross App Redirects

Description

Enabling cross-application redirects can allow unvalidated redirect attacks via the returnUrl parameter during the login process.

Disable cross-application redirects to by setting the *enableCrossAppRedirects* attribute to **false**.

References

- CWE-601: URL Redirection to Untrusted Site
- https://msdn.microsoft.com/en-us/library/1d3t3c61(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The following example shows the *enableCrossAppRedirects* attribute set to true.

```
<system.web>
    ...
    <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx" enableCrossAppRedirects="true" />
    </authentication>
    ...
</system.web>
```

Secure Configuration: Explicitly set the enableCrossAppRedirects attribute to false, or simply remove the insecure configuration. The default value for enableCrossAppRedirects is false.

```
<system.web>
    ...
    <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx" enableCrossAppRedirects="false" />
    </authentication>
    ...
</system.web>
```

Scan Results

SEC0005 ran successfully without any identified instances.

✔ SEC0006: Forms Authentication: Weak Cookie Protection

Description

Forms Authentication cookies must use strong encryption and message authentication code (MAC) validation to protect the cookie value from inspection and tampering.

Configure the forms element's *protection* attribute to **All** to enable cookie data validation and encryption.

References

- CWE-565: Reliance on Cookies without Validation and Integrity Checking
- https://msdn.microsoft.com/en-us/library/1d3t3c61(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the *protection* attribute is *All*. Setting the value to any of the options below weakens the authentication cookie protections.

```
<system.web>
    ...
    <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx" protection="None|Encryption|Validation" />
    </authentication>
    ...
</system.web>
```

Secure Configuration: Explicitly set the *protection* attribute to *All*, or simply remove the insecure configuration.

```
<system.web>
    ...
    <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx" protection="All" />
    </authentication>
    ...
</system.web>
```

Scan Results

SEC0006 ran successfully without any identified instances.

✔ SEC0007: Forms Authentication: Weak Timeout

Description

Excessive authentication timeout values provide attackers with a large window of opportunity to hijack user's authentication tokens.

Configure the forms timeout value to meet your organization's timeout policy. If your organization does not have a timeout policy, the following guidance can be used:

App	Timeout
High Security	15 minutes
Medium Security	30 minutes
Low Security	60 minutes

The default forms authentication timeout value is *30* minutes.

References

- CWE-613: Insufficient Session Expiration
- https://msdn.microsoft.com/en-us/library/1d3t3c61(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The following examples shows the forms authentication cookie timeout set to 480 minutes (8 hours).

```
<system.web>
    ...
    <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx" timeout="480" />
    </authentication>
    ...
</system.web>
```

Secure Configuration: Explicitly set the *timeout* attribute to an appropriate value.

```
<system.web>
    ...
    <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx" timeout="15" />
    </authentication>
    ...
</system.web>
```

Scan Results

SEC0007 ran successfully without any identified instances.

✔ SEC0008: Header Checking Disabled

Description

Disabling the HTTP Runtime header checking protection opens the application up to HTTP Header Injection (aka Response Splitting) attacks.

Enable the header checking protection by setting the httpRuntime element's enableHeaderChecking attribute to true, which is the default value.

References

- CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers
- https://msdn.microsoft.com/en-us/library/e1f13641(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the *enableHeaderChecking* attribute is *true*.

```
<system.web>
    ...
    <httpRuntime enableHeaderChecking="false" />
    ...
</system.web>
```

Secure Configuration: Explicitly set the *enableHeaderChecking* attribute to true, or simply remove the insecure configuration.

```
<system.web>
    ...
    <httpRuntime enableHeaderChecking="true" />
    ...
</system.web>
```

Scan Results

SEC0008 ran successfully without any identified instances.

✔ SEC0009: Version Header Enabled

Description

The Version HTTP response header sends the ASP.NET framework version to the client's browser. This information can help an attacker identify vulnerabilities in the server's framework version and should be disabled in production.

Disable the version response header by setting the httpRuntime element's *enableVersionHeader* attribute to **false**.

References

- CWE-200: Information Exposure
- https://msdn.microsoft.com/en-us/library/e1f13641(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the *enableVersionHeader* attribute is *true*.

```
<system.web>
    ...
    <httpRuntime enableVersionHeader="true" />
    ...
</system.web>
```

Secure Configuration: Explicitly set the *enableVersionHeader* attribute to false, or simply remove the insecure configuration.

```
<system.web>
    ...
    <httpRuntime enableVersionHeader="false" />
    ...
</system.web>
```

Scan Results

SEC0009 was not enabled and did not run.

✔ SEC0010: Event Validation Disabled

Description

Event validation prevents unauthorized post backs in web form applications. Disabling this feature can allow attackers to forge requests from controls not visible or enabled on a given web form.

Enable event validation by setting the page element's *eventValidation* attribute to **true**.

References

- CWE-807: Reliance on Untrusted Inputs in a Security Decision
- https://msdn.microsoft.com/en-us/library/950xf363(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the *enableEventValidation* attribute is *true*.

```
<system.web>
    ...
    <pages enableEventValidation="false" />
    ...
</system.web>
```

Secure Configuration: Explicitly set the *enableEventValidation* attribute to true, or simply remove the insecure configuration.

```
<system.web>
    ...
    <pages enableEventValidation="true" />
    ...
</system.web>
```

Scan Results

SEC0010 ran successfully without any identified instances.

✔ SEC0011: View State Mac Disabled

Description

The ViewStateMac protection prevents tampering with the web forms view state and event validation hidden fields. Disabling this feature can allow attackers to manipulate these fields in the browser and bypass several security features in the .NET framework.

Enable the view state mac protection by setting the page element's viewStateMac attribute to true.

ℹ As of .NET version 4.5.1, the ViewStateMac can no longer be disabled.

References

- CWE-807: Reliance on Untrusted Inputs in a Security Decision
- https://msdn.microsoft.com/en-us/library/950xf363(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the *enableViewStateMac* attribute is *true*.

```
<system.web>
    ...
    <pages enableViewStateMac="false" />
    ...
</system.web>
```

Secure Configuration: Explicitly set the *enableViewStateMac* attribute to true, or simply remove the insecure configuration.

```
<system.web>
    ...
    <pages enableViewStateMac="true" />
    ...
</system.web>
```

Scan Results

SEC0011 ran successfully without any identified instances.

✔ SEC0012: Request Validation Disabled

Description

The ValidateRequest protection denies known malicious XSS payloads found in form and URL request parameters. Request validation has known bypass issues and does not prevent all XSS attacks, but it does provide a strong countermeasure for most payloads targeting a HTML context.

Request validation should be enabled by setting the page element's requestValidation attribute to true. Then, consider making exceptions or overriding the default behavior on individual request parameters.

References

- CWE-20: Improper Input Validation
- https://msdn.microsoft.com/en-us/library/950xf363(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the *validateRequest* attribute is *true*.

```
<system.web>
    ...
    <pages validateRequest="false" />
    ...
</system.web>
```

Secure Configuration: Explicitly set the *validateRequest* attribute to true, or simply remove the insecure configuration.

```
<system.web>
    ...
    <pages validateRequest="true" />
    ...
</system.web>
```

Scan Results

SEC0012 ran successfully without any identified instances.

✔ SEC0013: View State Encryption Disabled

Description

The web forms view state hidden field is base64 encoded by default, which can be easily decoded. Applications placing sensitive data into the view state are vulnerable to information leakage issues via the view state parameter.

Configure the pages element's *viewStateEncryptionMode* attribute to **Always** to encrypt the view state data with the .NET machine key.

References

- CWE-200: Information Exposure
- https://msdn.microsoft.com/en-us/library/950xf363(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the *viewStateEncryptionMode* attribute is *Auto*, which only encrypts view state data for controls that request encryption.

```
<system.web>
    ...
    <pages viewStateEncryptionMode="Never" />
    ...
</system.web>
```

Secure Configuration: Explicitly set the *viewStateEncryptionMode* attribute to *Always*.

```
<system.web>
    ...
    <pages viewStateEncryptionMode="Always" />
    ...
</system.web>
```

Scan Results

SEC0013 ran successfully without any identified instances.

✔ SEC0014: Insecure HTTP Cookies

Description

Cookies containing authentication tokens, session tokens, and other state management credentials must be protected in transit across a network.

Set the httpCookie element's *requireSSL* attribute to **true** to prevent the browser from transmitting cookies over HTTP.

References

- CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
- https://msdn.microsoft.com/library/ms228262(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the *requireSSL* attribute is *false*.

```
<system.web>
    ...
    <httpCookies requireSSL="false" />
    ...
</system.web>
```

Secure Configuration: Explicitly set the *requireSSL* attribute to *true*.

```
<system.web>
    ...
    <httpCookies requireSSL="true" />
    ...
</system.web>
```

Scan Results

SEC0014 ran successfully without any identified instances.

✔ SEC0015: Cookies Accessible Via Script

Description

Cookies that do not have the httpOnly attribute set are accessible in the browser by scripts. This can allow attackers to inject malicious scripts into the site and extract authentication cookie values to a remote server.

Setting the httpCookie element's *httpOnlyCookies* attribute to **true** will help prevent client-side session hijacking attempts.

References

- CWE-1004: Sensitive Cookie Without 'HttpOnly' Flag
- https://msdn.microsoft.com/library/ms228262(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the *httpOnlyCookies* attribute is *false*.

```
<system.web>
    ...
    <httpCookies httpOnlyCookies="false" />
    ...
</system.web>
```

Secure Configuration: Explicitly set the *httpOnlyCookies* attribute to *true*.

```
<system.web>
    ...
    <httpCookies httpOnlyCookies="true" />
    ...
</system.web>
```

Scan Results

SEC0015 ran successfully without any identified instances.

✔ SEC0016: Cleartext Machine Key

Description

The machine key element defines keys to use for encryption, decryption, and HMAC validation of authentication cookies, view state, event validation, and other framework fields. The validation and decryption key values should not be stored in configuration files in cleartext.

Cleartext machine key validation and decryption values became a high risk to .NET Web Forms applications when @jared_mclaren presented a talk titled RCEvil.NET - A Super Serial Story at @BSidesIowa 2019. This talk released a tool that enables remote attackers knowing the machine keys to anonymously gain Remote Code Execution (RCE) on IIS web servers.

Encrypt the machineKey section of the configuration file using aspnet_regiis.exe.

References

- CWE-312: Cleartext Storage of Sensitive Information
- https://msdn.microsoft.com/en-us/library/w8h3skw9(v=vs.100).aspx

Code Example(s)

Insecure Configuration: Secrets stored in the web.config file are not encrypted by default.

```
<system.web>
    ...
    <machineKey validationKey="NOTASECRETANYMORE" decryptionKey="NOTASECRETANYMORE" validation="SHA1" decryption="AES"/>
    ...
</system.web>
```

Secure Configuration: Use the aspnet_regiis.exe utility to encrypt the *machineKey* element of the configuration file. Run this command from the root of your app.

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\aspnet_regiis.exe -pef "system.web/machineKey" . -prov "DataProtectionConfigu
```

```
<system.web>
    ...
    <machineKey configProtectionProvider="DataProtectionConfigurationProvider">
        <EncryptedData>
            <CipherData>
                <CipherValue>85c0b357d397d3e63f03e5b6ae299d66</CipherValue>
            </CipherData>
        </EncryptedData>
    </machineKey>
    ...
</system.web>
```

Scan Results

SEC0016 ran successfully without any identified instances.

✔ SEC0020: Weak Session Timeout

Description

If session data is used by the application for authentication, excessive timeout values provide attackers with a large window of opportunity to hijack user's session tokens.

Configure the session timeout value to meet your organization's timeout policy. If your organization does not have a timeout policy, the following guidance can be used:

App	Timeout
High Security	15 minutes
Medium Security	30 minutes
Low Security	60 minutes

The default session timeout value is 20 minutes.

References

- CWE-613: Insufficient Session Expiration
- https://msdn.microsoft.com/en-us/library/h6bb9cz9(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The example below shows an excessive timeout value of 480 minutes (8 hours).

```
<system.web>
    ...
    <sessionState timeout="480" />
    ...
</system.web>
```

Secure Configuration: Explicitly set the *timeout* attribute to an appropriate value.

```
<system.web>
    ...
    <sessionState timeout="15" />
    ...
</system.web>
```

Scan Results

SEC0020 ran successfully without any identified instances.

✔ SEC0021: State Server Mode

Description

The session StateServer mode transports session data insecurely to a remote server. The remote server also does not require system authentication to access the session data for an application. This risk depends entirely on the sensitivity of the data stored in the user's session.

If the session data is considered sensitive, consider adding an external control (e.g. IPSEC) that provides mutual authentication and transport security.

References

- CWE-319: Cleartext Transmission of Sensitive Information
- https://msdn.microsoft.com/en-us/library/h6bb9cz9(v=vs.100).aspx

Code Example(s)

Insecure Configuration: The default value for the *mode* attribute is *InProc*, which uses in memory session storage on the web server. Options such as *StateServer* allow another server to handle session management in a multi-node web farm scenario.

```
<system.web>
    ...
    <sessionState mode="StateServer" />
    ...
</system.web>
```

Secure Configuration: Use a different distributed session state provider, such as *SQLServer*, that supports system authentication and encrypted data transmission. Or, implement an external control for system to system authentication and secure transmission, such as *IPSec*.

```
<system.web>
    ...
    <sessionState mode="SQLServer" />
    ...
</system.web>
```

Scan Results

SEC0021 ran successfully without any identified instances.

✔ SEC0022: Model Request Validation Disabled

Description

Request validation performs blacklist input validation for XSS payloads found in form and URL request parameters. Request validation has known bypass issues and does not prevent all XSS attacks, but it does provide a strong countermeasure for most payloads targeting a HTML context.

Request validation is enabled by default during model binding to dynamic HTML request parameters, but can be disabled on individual model properties using the **AllowHtml** attribute. The following countermeasures can help validate data and filter XSS payloads:

- Avoid accepting HTML input from untrusted data sources. Leave request validation enabled and consider accepting a different data format, such as markdown.
- Create a custom validation attribute that performs strict validation on the given property
- Create a custom action filter that sanitizes request parameters containing HTML using the AntiXss HTML Sanitizer class to strip potentially dangerous script from untrusted data before consuming the data.

ℹ Version 4.3.0 is the recommended HTML sanitizer library version

References

- CWE-20: Improper Input Validation
- https://msdn.microsoft.com/en-us/library/system.web.mvc.allowhtmlattribute(v=vs.118).aspx
- https://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.validationattribute.aspx
- https://wpl.codeplex.com/

Code Example(s)

Insecure Code: The following example shows the *ProductFeedback* model setting the *AllowHtml* attribute on the *Feedback* property.

```
public class ProductFeedbackModel
{
        [Display(Name = "Rating")]
        public int Rating { get; set; }

        [Display(Name = "Feedback")]
        [AllowHtml]
        public string Feedback { get; set; }
}
```

Secure Code: The following example removes the *AllowHtml* attribute to enable request validation.

```
public class ProductFeedbackModel
{
        [Display(Name = "Rating")]
        public int Rating { get; set; }

        [Display(Name = "Feedback")]
```

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

Scan Results

SEC0022 ran successfully without any identified instances.

✔ SEC0024: Unencoded Response Write

Description

Data is written to the browser using the HttpResonse.Write method. This can result in Cross-Site Scripting (XSS) vulnerabilities if the data source is considered untrusted or dynamic (request parameters, database, web service, etc.).

Ensure all dynamic data is properly encoded in the browser. Consider using the AntiXssEncoder library to neutralize dangerous data before writing it the browser. For example, the HtmlEncode method should be used to encode data displayed in an HTML context.

References

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- https://msdn.microsoft.com/en-us/library/1463ysyw(v=vs.110).aspx

Code Example(s)

Insecure Code: The following example shows the Raw Response.Write method writing a dynamic request parameter into HTML without HTML encoding.

```
string user = Request["user"]?.ToString();
Response.Write("We're sorry" + user + "but that contest is not valid.  Please click your back button and try again.");
```

Secure Code: Ensure HTML encoding is applied prior to writing dynamic data into the browser's HTML context.

```
string user = Request["user"]?.ToString();
Response.Write("We're sorry" + Encoder.HtmlEncode(user) + "but that contest is not valid.  Please click your back button ar
```

Scan Results

SEC0024 ran successfully without any identified instances.

✔ SEC0026: Weak Cipher Mode

Description

Symmetric algorithms use a Cipher Mode to repeatedly apply a cryptographic transformation to multiple blocks of data. Weaknesses in the Cipher Mode can allow cryptographic attacks to compromise the integrity or confidentiality of data.

SEC0026 identifies usage of the weak Cipher Modes in symmetric encryption operations including CFB, CTS, ECB, and OFB. Each of these cipher modes contain cryptographic weaknesses. For example, Electronic Codebook (ECB) mode encrypts blocks individually without using an initialization vector, which fails to provide entropy for identical plaintext blocks being encrypted with the same encryption key. This can allow attackers to identify patterns and repetition in ciphertext, and may lead to the discovery of the original encryption key.

The .NET cryptography libraries do not contain support for the strongest Cipher Modes: Counter Mode (CTR) and Galois Counter Mode (GCM). Third-party encryption libraries, such as Bouncy Castle can be used instead of the native .NET cryptography libraries.

If a product is required to use the .NET cryptography libraries, use the *CipherMode.CBC* option for symmetric block cipher operations. However, the *CipherMode.CBC* mode is vulnerable to Padding Oracle attacks. See SEC0124 for details on Cipher Block Chaining (CBC) padding weaknesses and defenses.

References

- CWE-327: Use of a Broken or Risky Cryptographic Algorithm
- https://msdn.microsoft.com/en-us/library/system.security.cryptography.ciphermode(v=vs.110).aspx

Code Example(s)

Insecure Code: The following example shows the AES algorithm setting the Mode to the CipherMode.ECB option.

```
using (MemoryStream mStream = new MemoryStream())
{
    //Input bytes
    byte[] inputBytes = Encoding.UTF8.GetBytes(plainText);

    SymmetricAlgorithm alg = Aes.Create();
    alg.Mode = CipherMode.ECB;

    //Set key and iv
    alg.Key = GetKey();
    alg.IV = GetIv();

    //Create the crypto stream
    CryptoStream cStream = new CryptoStream(mStream
      , alg.CreateEncryptor()
      , CryptoStreamMode.Write);
    cStream.Write(inputBytes, 0, inputBytes.Length);
    cStream.FlushFinalBlock();
    cStream.Close();

    //Get the output
    output = mStream.ToArray();

    //Close resources
    mStream.Close();
```

```
    alg.Clear();
}
```

Secure Code: Set the AES algorithm mode set to CipherMode.CBC option, perform an integrity check generate a unique initialization vector for each input.

```
//Perform integrity check on incoming data
string[] args = model.ProtectedData.Split('.');
byte[] ciphertext = Convert.FromBase64String(args[0]);
byte[] hmac = Convert.FromBase64String(args[1]);

HMACSHA256 hmac = new HMACSHA256(_KEY);
byte[] verification = hmac.ComputeHash(ciphertext);

if (!verification.SequenceEqual(hmac))
    throw new ArgumentException("Invalid signature detected.");

using (MemoryStream mStream = new MemoryStream())
{
    SymmetricAlgorithm alg = Aes.Create();
    alg.Mode = CipherMode.CBC;

    //Set key and iv
    alg.Key = GetKey();
    alg.IV = GetIv();

    //Create the crypto stream
    CryptoStream cStream = new CryptoStream(mStream
      , alg.CreateDecryptor()
      , CryptoStreamMode.Write);
    cStream.Write(ciphertext, 0, inputBytes.Length);
    cStream.FlushFinalBlock();
    cStream.Close();

    //Get the cleartext
    byte[] cleartext = mStream.ToArray();

    //Close resources
    mStream.Close();
    alg.Clear();
}
```

Scan Results

SEC0026 ran successfully without any identified instances.

✔ SEC0028: Weak Algorithm: SHA1

Description

The SHA1 algorithm has known collision weaknesses and should not be used for security operations in new applications.

Use the SHA256Managed (at least) preferably SHA512Managed for hashing operations.

ℹ This alone is not sufficient for password hashing, which requires a unique salt and adaptive hashing algorithm. See the references below for more information on password hashing in .NET.

References

- CWE-327: Use of a Broken or Risky Cryptographic Algorithm
- https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha1managed(v=vs.110).aspx
- https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha256(v=vs.110).aspx
- https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha512(v=vs.110).aspx
- https://blogs.dropbox.com/tech/2016/09/how-dropbox-securely-stores-your-passwords/

Code Example(s)

Insecure Code: The following example shows the SHA1 algorithm hashing a piece of data.

```
HashAlgorithm hash = new SHA1CryptoServiceProvider();
byte[] bytes = hash.ComputeHash(input);
```

Secure Code: Use SHA256Managed or SHA512Managed for hashing operations (note: this is not enough for storing passwords).

```
HashAlgorithm hash = new SHA512Managed();
byte[] bytes = hash.ComputeHash(input);
```

Scan Results

SEC0028 ran successfully without any identified instances.

✔ SEC0031: Command Injection: Process.Start

Description

Concatenating untrusted data into operating system commands can allow attackers to execute arbitrary commands against the server's operating system.

Defending against command injection in the .NET ecosystem is more difficult than other injection categories because no special encoding method exists to approve safe characters and escape evil characters.

To prevent command injection, follow this pattern:

- Does the user really need to directly control the *fileName* passed to this command? The answer is usually No. Consider moving the *fileName* values to server-side storage and look up the value using a non-injectable value such as a GUID, integer, etc. passed in from the request. For example, consider an application that allows the user to run two commands: calc.exe and process.exe.

Command Name	GUID
calc.exe	06C67D8C-CAD5-4003-B065-1089CFF0D1E9
process.exe	A180A8AF-C667-4074-A768-C95F13819E2A

  Requests pass in a valid GUID and the application selects the associated process. If the GUID is not valid, throw an exception. This ensures that the user can only run a process that exists in your list of approved processes.
- Validate incoming *arguments* against a strict list of approved values or characters only and reject everything else. Consider using the server-side lookup tables (as described above) for argument values as well. Avoid using *string* values whenever possible, instead opting for non-injectable data types (e.g. GUID, integer, date, decimal, etc.). If *string* values must be passed as an *argument*, then use regular expressions to restrict the characters to avoid special characters. For example: [A-za-z0-9]+
- Reject the special characters shown to the right in *Figure 1*.

References

- CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- https://www.owasp.org/index.php/Top_10-2017_A1-Injection
- https://msdn.microsoft.com/en-us/library/system.diagnostics.process.start(v=vs.110).aspx

Code Example(s)

Figure 1 The following characters are control characters in various shells (cmd, bash, etc.) and should be stripped from untrusted string values being passed to the *fileName* or *arguments* parameters.

```
& < > [ ] { } ^ = ; ! ' + , ` ~ [white space].
```

Insecure Code: The following example shows the *model.FileName* parameter being passed to the *Process.Start* method's *fileName* parameter. This allows an attacker to execute arbitrary commands on the server.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Run(ProcessViewModel model)
{
    Process p = Process.Start(model.FileName);
    model.ExitCode = p.ExitCode;
    return View(model);
}
```

Secure Code: Rather than allowing the user to control the file name directly, look up the *fileName* in a server-side collection. The following example verifies the file id against the command entries in a server-side configuration file before executing the command.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Run(ProcessViewModel model)
{
    //Pull valid commands from the configuration file
    List<Command> commands = GetCommands();

    //Verify the command exists
    Command c = commands.FirstOrDefault(i => i.Id == model.FileId);
    if (c == null)
        throw new ArgumentException("Invalid file name parameter");

    Process p = Process.Start(c.FileName);
    model.ExitCode = p.ExitCode;
    return View(model);
}
```

Scan Results

SEC0031 ran successfully without any identified instances.

✔ SEC0032: Command Injection: ProcessStartInfo

Description

Concatenating untrusted data into operating system commands can allow attackers to execute arbitrary commands against the server's operating system.

Defending against command injection in the .NET ecosystem is more difficult than other injection categories because no special encoding method exists to allow safe characters and escape evil characters.

To prevent command injection, follow this pattern:

- Does the user really need to directly control the *fileName* passed to this command? The answer is usually No. Consider moving the *fileName* values to server-side storage and look up the value using a non-injectable value such as a GUID, integer, etc. passed in from the request. For example, consider an application that allows the user to run two commands: calc.exe and process.exe.

Command Name	GUID
calc.exe	06C67D8C-CAD5-4003-B065-1089CFF0D1E9
process.exe	A180A8AF-C667-4074-A768-C95F13819E2A

  Requests pass in a valid GUID and the application selects the associated process. If the GUID is not valid, throw an exception. This ensures that the user can only run a process that exists in your list of approved processes.
- Validate incoming *arguments* against a strict list of approved values or characters and reject everything else. Consider using the server-side lookup tables (as described above) for argument values as well. Avoid using *string* values whenever possible, instead opting for non-injectable data types (e.g. GUID, integer, date, decimal, etc.). If *string* values must be passed as an *argument*, then use regular expressions to restrict the characters to avoid special characters. For example: [A-za-z0-9]+
- Reject the special characters shown to the right in *Figure 1*.

References

- CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- https://www.owasp.org/index.php/Top_10-2017_A1-Injection
- https://msdn.microsoft.com/en-us/library/system.diagnostics.processstartinfo(v=vs.110).aspx

Code Example(s)

Figure 1 The following characters are control characters in various shells (cmd, bash, etc.) and should be stripped from untrusted string values being passed to the *fileName* or *arguments* parameters.

```
& < > [ ] { } ^ = ; ! ' + , ` ~ [white space].
```

Insecure Code: The following example shows the *model.FileName* parameter being passed to the *ProcessStartInfo* constructor's *fileName* parameter. This allows an attacker to execute arbitrary commands on the server.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Run(ProcessViewModel model)
{
    ProcessStartInfo info = new ProcessStartInfo()
    {
        FileName = model.FileName,
    };
    Process p = Process.Start(info);
    model.ExitCode = p.ExitCode;
    return View(model);
}
```

Secure Code: Rather than allowing the user to control the file name directly, look up the *fileName* in a server-side collection. The following example verifies the file id against the command entries in a server-side configuration file before executing the command.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Run(ProcessViewModel model)
{
    //Pull valid commands from the configuration file
    List<Command> commands = GetCommands();

    //Verify the command exists
    Command c = commands.FirstOrDefault(i => i.Id == model.FileId);
    if (c == null)
        throw new ArgumentException("Invalid file name parameter");

    ProcessStartInfo info = new ProcessStartInfo()
    {
        FileName = c.FileName,
    };
    Process p = Process.Start(info);
    model.ExitCode = p.ExitCode;
    return View(model);
}
```

Scan Results

SEC0032 ran successfully without any identified instances.

✔ SEC0033: Insecure HTTP Cookie Transport

Description

Cookies containing authentication tokens, session tokens, and other state management credentials must be protected in transit across a network.

The following vulnerable sinks are covered by SEC0033:

- Microsoft.AspNetCore.Http.CookieOptions
- System.Web.HttpCookie

Set the cookie options' **Secure** property to *true* to prevent the browser from transmitting cookies over HTTP.

References

- CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
- https://www.owasp.org/index.php/Session_Management_Cheat_Sheet
- https://docs.microsoft.com/en-us/dotnet/api/system.web.httpcookie.secure
- https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.cookieoptions.secure

Code Example(s)

Insecure Code: The following example shows the *CookieOptions Secure* property set to **false**, which allows the cookie to be sent over insecure HTTP connections.

```
CookieOptions options = new CookieOptions()
{
    Secure = false,
};
```

Secure Code: Set the *CookieOptions Secure* property to **true** to require the cookie to be sent over secure HTTPS connections.

```
CookieOptions options = new CookieOptions()
{
    Secure = true,
};
```

Scan Results

SEC0033 ran successfully without any identified instances.

✔ SEC0034: HTTP Cookie Accessible via Script

Description

Cookies containing authentication tokens, session tokens, and other state management credentials should be protected from malicious JavaScript running in the browser. Setting the httpOnly attribute to false can allow attackers to inject malicious scripts into the site and extract authentication cookie values to a remote server.

The following vulnerable sinks are covered by SEC0034:

- Microsoft.AspNetCore.Http.CookieOptions
- System.Web.HttpCookie

Configure the cookie options' *httpOnly* property to **true**, which prevents cookie access from scripts running in the browser.

References

- CWE-1004: Sensitive Cookie Without 'HttpOnly' Flag
- https://www.owasp.org/index.php/Session_Management_Cheat_Sheet
- https://docs.microsoft.com/en-us/dotnet/api/system.web.httpcookie.httponly
- https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.cookieoptions.httponly

Code Example(s)

Insecure Code: The following example shows the *CookieOptions HttpOnly* property set to **false**, which allows the cookie to be read via JavaScript running in the browser.

```
CookieOptions options = new CookieOptions()
{
    HttpOnly = false,
};
```

Secure Code: Set the *CookieOptions HttpOnly* property to **true** to prevent JavaScript from reading the cookie value.

```
 CookieOptions options = new CookieOptions()
{
    HttpOnly = true,
};
```

Scan Results

SEC0034 ran successfully without any identified instances.

✔ SEC0035: XPathDocument External Entity Expansion

Description

XML External Entity (XXE) vulnerabilities occur when applications process untrusted XML data without disabling external entities and DTD processing. Processing untrusted XML data with a vulnerable parser can allow attackers to extract data from the server, perform denial of service attacks, and in some cases gain remote code execution.

.NET Framework v4.5.1 (And Older)

The XPathDocument class is vulnerable to XXE attacks when loading XML from a file URI, TextReader, or Stream. The only secure way to parse XML using the XPathDocument class in .NET Framework versions prior to 4.5.2 is to load the content using an XmlReader object, which by default does not have external entities enabled.

.NET Framework v4.5.2 (And Newer)

The XPathDocument class was patched to securely parse XML content. Applications targeting .NET 4.5.2 and greater are not vulnerable (by default) to XXE attacks when using the XPathDocument class.

The SEC0035 analyzer looks for usage of the XPathDocument class in projects targeting .NET Framework 4.5.1 and older. Diagnostic warnings will be raised (regardless of the source XML data) if the constructor is loading XML data from a file URI, TextReader, or Stream.

References

- CWE-611: Improper Restriction of XML External Entity Reference
- https://www.owasp.org/index.php/Top_10-2017_A4-XML_External_Entities_(XXE)
- https://docs.microsoft.com/en-us/dotnet/api/system.xml.xpath.xpathdocument
- https://www.jardinesoftware.net/2016/09/12/xxe-in-net-and-xpathdocument/

Code Example(s)

Insecure Code: The following example shows the XPathDocument loading XML content using a file URI stored in the *model.FilePath* variable:

```
XPathDocument xmlDoc = new XPathDocument(model.FilePath);
XPathNavigator nav = xmlDoc.CreateNavigator();
```

Secure Code: Load the XML content into an *XmlReader* object first. Then, pass the XmlReader into the XPathDocument constructor.

```
XmlReader reader = XmlReader.Create(model.FilePath);
XPathDocument xmlDoc = new XPathDocument(reader);
XPathNavigator nav = xmlDoc.CreateNavigator();
```

Scan Results

SEC0035 ran successfully without any identified instances.

✔ SEC0036: XML Reader External Entity Expansion

Description

XML External Entity (XXE) vulnerabilities occur when applications process untrusted XML data without disabling external entities and DTD processing. Processing untrusted XML data with a vulnerable parser can allow attackers to extract data from the server, perform denial of service attacks, and in some cases gain remote code execution.

The XmlReaderSettings and XmlTextReader classes are vulnerable to XXE attacks when setting the DtdProcessing property to DtdProcessing.Parse or the ProhibitDtd property to false.

To prevent XmlReader XXE attacks, avoid using the deprecated ProhibitDtd property. Set the DtdProcessing property to DtdProcessing.Ignore or DtdProcessing.Prohibit.

The SEC0036 analyzer currently looks for insecure XmlReaderSettings and XmlTextReader configuration. Data flow analysis is not performed to determine if the XML content being parsed is actually controllable by an attacker.

References

- CWE-611: Improper Restriction of XML External Entity Reference
- https://www.owasp.org/index.php/Top_10-2017_A4-XML_External_Entities_(XXE)
- https://www.jardinesoftware.net/2016/05/26/xxe-and-net/
- https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlreadersettings.dtdprocessing
- https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlreadersettings.prohibitdtd
- https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmltextreader.dtdprocessing
- https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmltextreader.prohibitdtd

Code Example(s)

Insecure Code: The following example shows the XmlReaderSettings *DtdProcessing* property set to *Parse*, which allows external entities to be parsed by the XmlReader:

```
XmlReaderSettings rs = new XmlReaderSettings
{
    DtdProcessing = DtdProcessing.Parse,
};

XmlReader reader = XmlReader.Create("evil.xml", rs);
while (reader.Read())
```

Secure Code: Configure the *XmlReaderSettings* to *Prohibit* parsing external XML entities.

```
XmlReaderSettings rs = new XmlReaderSettings
{
    DtdProcessing = DtdProcessing.Prohibit,
};
```

```
XmlReader reader = XmlReader.Create("evil.xml", rs);
while (reader.Read())
```

Scan Results

SEC0036 ran successfully without any identified instances.

✔ SEC0037: XML Document External Entity Expansion

Description

XML External Entity (XXE) vulnerabilities occur when applications process untrusted XML data without disabling external entities and DTD processing. Processing untrusted XML data with a vulnerable parser can allow attackers to extract data from the server, perform denial of service attacks, and in some cases gain remote code execution.

The XmlDocument class is vulnerable to XXE attacks when setting the XmlResolver property to resolve external entities.

To prevent XmlDocument XXE attacks, set the XmlResolver property to null.

The SEC0037 analyzer currently looks for insecure XmlDocument configuration. Data flow analysis is not performed to determine if the XML content being parsed is actually controllable by an attacker.

References

- CWE-611: Improper Restriction of XML External Entity Reference
- https://www.owasp.org/index.php/Top_10-2017_A4-XML_External_Entities_(XXE)
- https://www.jardinesoftware.net/2016/05/26/xxe-and-net/
- https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmldocument.xmlresolver

Code Example(s)

Insecure Code: The following example shows the XmlDocument *XmlResolver* property set to resolve external entities:

```
XmlUrlResolver resolver = new XmlUrlResolver();
resolver.Credentials = CredentialCache.DefaultCredentials;

XmlDocument xmlDoc = new XmlDocument();
xmlDoc.XmlResolver = resolver;
xmlDoc.LoadXml(xml);
```

Secure Code: Configure the XmlDocument *XmlResolver* property to *null* to prevent resolving external XML entities.

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.XmlResolver = null;
xmlDoc.LoadXml(xml);
```

Scan Results

SEC0037 ran successfully without any identified instances.

✔ SEC0038: Directory Listing Enabled

Description

Directory listing provides a complete index of the resources located in a web directory. Enabling directory listing can expose sensitive resources such as application binaries, configuration files, and static content that should not be exposed.

Unless directory listing is required to meet the application's functional requirements, disable the listing by setting the **directoryBrowse** element's **enabled** attribute to false.

References

- CWE-548: Information Exposure Through Directory Listing
- https://www.owasp.org/index.php/Top_10-2017_A6-Security_Misconfiguration
- https://docs.microsoft.com/en-us/iis/configuration/system.webserver/directorybrowse

Code Example(s)

Insecure Code: The following example shows the *directoryBrowse enabled* attribute set to **true**, which enables directory listing information in the browser.

```
<system.webServer>
  <directoryBrowse enabled="true"/>
</system.webServer>
```

Secure Code: Set the *directoryBrowse enabled* attribute to **false**. Or, remove the **directoryBrowse** element to inherit the default value of **false**.

```
<system.webServer>
  <directoryBrowse enabled="false"/>
</system.webServer>
```

Scan Results

SEC0038 ran successfully without any identified instances.

✔ SEC0100: Raw Inline Expression

Description

Data is written to the browser using a raw write: <%= var %>. This can result in Cross-Site Scripting (XSS) vulnerabilities if the data source is considered untrusted or dynamic (request parameters, database, web service, etc.).

Instead of using a raw write, use the inline HTML encoded shortcut (<%: var %>) to automatically HTML encode data before writing it to the browser.

References

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- https://software-security.sans.org/developer-how-to/developer-guide-xss

Code Example(s)

Insecure Code: The following example shows a raw inline expression writing a dynamic request parameter to the browser.

```
<h2>
    Welcome <%= Request["UserName"].ToString() %>
</h2>
```

Secure Code: Replace the raw inline expression with the secure HTML encode inline shortcut.

```
<h2>
    Welcome <%: Request["UserName"].ToString() %>
</h2>
```

Scan Results

SEC0100 ran successfully without any identified instances.

✔ SEC0101: Raw Binding Expression

Description

Data is written to the browser using a raw binding expression: <%# Item.Variable %>. This can result in Cross-Site Scripting (XSS) vulnerabilities if the data source is considered untrusted or dynamic (request parameters, database, web service, etc.).

Instead of using a raw binding expression, use the HTML encoded binding shortcut (<%#: Item.Variable %>) to automatically HTML encode data before writing it to the browser.

References

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- https://software-security.sans.org/developer-how-to/developer-guide-xss

Code Example(s)

Insecure Code: The following example shows a raw binding expression writing a dynamic database value to the browser.

```
<asp:GridView ID="gv" runat="server" ItemType="Data.Product">
    <Columns>
        <asp:TemplateField HeaderText="Product">
            <ItemTemplate>
                <%# Item.ProductName %>
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

Secure Code: Replace the raw binding expression with the secure HTML encode binding shortcut (<%#:).

```
<asp:GridView ID="gv" runat="server" ItemType="Data.Product">
    <Columns>
        <asp:TemplateField HeaderText="Product">
            <ItemTemplate>
                <%#: Item.ProductName %>
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

Scan Results

SEC0101 ran successfully without any identified instances.

✔ SEC0102: Raw Razor Method

Description

Data is written to the browser using a raw Razor helper method: @Html.Raw(Model.Variable). This can result in Cross-Site Scripting (XSS) vulnerabilities if the data source is considered untrusted or dynamic (request parameters, database, web service, etc.).

Instead of using the raw Razor helper method, use a Razor helper that performs automatic HTML encoding before writing it to the browser.

References

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- https://software-security.sans.org/developer-how-to/developer-guide-xss

Code Example(s)

Insecure Code: The following example shows the Raw Razor helper method writing a dynamic value to the view.

```
<div class="loginDisplay">
    @Html.Raw(string.Format("Welcome <span class=\"bold\">{0}</span>!", Model.UserName))
</div>
```

Secure Code: Replace the Raw Razor method with a Razor helper method that automatically HTML encodes the output data.

```
<div class="loginDisplay">
    Welcome <span class="bold">@Model.UserName</span>!
</div>
```

Scan Results

SEC0102 ran successfully without any identified instances.

✔ SEC0103: Raw WriteLiteral Method

Description

Data is written to the browser using the raw WriteLiteral method. This can result in Cross-Site Scripting (XSS) vulnerabilities if the data source is considered untrusted or dynamic (request parameters, database, web service, etc.).

Instead of using the raw WriteLiteral method, use a Razor helper that performs automatic HTML encoding before writing it to the browser.

References

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- https://msdn.microsoft.com/en-us/library/system.web.webpages.webpagebase.writeliteral(v=vs.111).aspx
- https://msdn.microsoft.com/en-us/library/system.web.webpages.html.htmlhelper_methods(v=vs.111).aspx

Code Example(s)

Insecure Code: The following example shows the Raw WriteLiteral method writing a dynamic value to the view.

```
<div class="loginDisplay">
@{
    WriteLiteral(string.Format("Welcome <span class=\"bold\">{0}</span>!", Model.UserName));
}
</div>
```

Secure Code: Replace the WriteLiteral method with a Razor helper method that automatically HTML encodes the output data.

```
<div class="loginDisplay">
    Welcome <span class="bold">@Model.UserName</span>!
</div>
```

Scan Results

SEC0103 ran successfully without any identified instances.

✔ SEC0104: Unencoded WebForms Property

Description

Data is written to the browser using a WebForms property that does not perform output encoding. This can result in Cross-Site Scripting (XSS) vulnerabilities if the data source is considered untrusted or dynamic (request parameters, database, web service, etc.).

WebForms controls are often found in HTML contexts, but can also appear in other contexts such as JavaScript, HTML Attribute, or URL. Fixing the vulnerability requires the appropriate Web Protection Library (aka AntiXSS) context-specific method to encode the data before setting the WebForms property.

References

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- https://msdn.microsoft.com/en-us/library/system.web.security.antixss.antixssencoder(v=vs.110).aspx
- https://wpl.codeplex.com/

Code Example(s)

Insecure Code: The following example shows the Literal.Text property set to a dynamic database value.

```
litDetails.Text = product.ProductDescription;
```

Secure Code: Encode data passed to the Literal.Text property with the Web Protection Library's (aka AntiXSS) appropriate context-specific method.

```
litDetails.Text = Encoder.HtmlEncode(product.ProductDescription);
```

Scan Results

SEC0104 ran successfully without any identified instances.

✔ SEC0105: Unencoded Label Text

Description

Data is written to the browser using the raw Label.Text method. This can result in Cross-Site Scripting (XSS) vulnerabilities if the data source is considered untrusted or dynamic (request parameters, database, web service, etc.).

Label controls are often found in HTML contexts, but can also appear in other contexts such as JavaScript, HTML Attribute, or URL. Fixing the vulnerability requires the appropriate Web Protection Library (aka AntiXSS) context-specific method to encode the data before setting the Label.Text property.

References

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- https://msdn.microsoft.com/en-us/library/system.windows.controls.label(v=vs.110).aspx
- https://msdn.microsoft.com/en-us/library/system.web.security.antixss.antixssencoder(v=vs.110).aspx
- https://wpl.codeplex.com/

Code Example(s)

Insecure Code: The following example shows the Label.Text property set to a dynamic database value.

```
lblDetails.Text = product.ProductDescription;
```

Secure Code: Encode data passed to the Label.Text property with the Web Protection Library's (aka AntiXSS) appropriate context-specific method.

```
lblDetails.Text = Encoder.HtmlEncode(product.ProductDescription);
```

Scan Results

SEC0105 ran successfully without any identified instances.

✔ SEC0106: SQL Injection: Dynamic LINQ Query

Description

Concatenating untrusted data into a dynamic SQL string and calling vulnerable LINQ methods can allow SQL Injection:

- ExecuteQuery
- ExecuteCommand

To ensure calls to vulnerable LINQ methods are parameterized, pass parameters into the statement using the method's second argument: *params object[] parameters*.

References

- CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- https://www.owasp.org/index.php/Top_10-2017_A1-Injection
- https://msdn.microsoft.com/en-us/library/bb361109(v=vs.110).aspx

Code Example(s)

Insecure Code: The following example shows dynamic SQL passed to the ExecuteQuery LINQ method.

```
using (DbDataContext context = new DbDataContext())
{
    string q = "SELECT Name from Items where ProductCode = " + model.ProductCode;
    name = context.ExecuteQuery<string>(q).SingleOrDefault().ToString();
}
```

Secure Code: Call the overloaded ExecuteQuery method that accepts a list of parameters in the second argument.

```
using (DbDataContext context = new DbDataContext())
{
    string q = "SELECT Name from Items where ProductCode = {0}";
    name = context.ExecuteQuery<string>(q, model.ProductCode).SingleOrDefault().ToString();
}
```

Scan Results

SEC0106 ran successfully without any identified instances.

✔ SEC0107: SQL Injection: ADO.NET

Description

ADO.NET classes allow applications to communicate directly to a backend database without using an Object Relational Mapping (ORM) framework. SEC0107 identifies the following classes allowing dynamic SQL statements to be constructed and executed:

- System.Data.SqlClient.SqlCommand
- Microsoft.Data.Sqlite.SqliteCommand
- System.Data.OleDb.OleDbCommand
- System.Data.Odbc.OdbcCommand
- IBM.Data.DB2.DB2Command

Ensure that calls to these methods do not concatenate untrusted data into dynamic SQL statements sent to the *CommandText*. Use parameter placeholders or stored procedures to prevent SQL Injection attacks.

References

- CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- https://www.owasp.org/index.php/Top_10-2017_A1-Injection
- https://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand(v=vs.110).aspx
- https://docs.microsoft.com/en-us/dotnet/api/microsoft.data.sqlite.sqlitecommand
- https://docs.microsoft.com/en-us/dotnet/api/system.data.oledb.oledbcommand
- https://docs.microsoft.com/en-us/dotnet/api/system.data.odbc.odbccommand

Code Example(s)

Insecure Code: The following example shows a snippet concatenating a request parameter into a dynamic SQL statement. The statement is executed using the *ExecuteScalar* method.

```
SqlCommand cmd = new SqlCommand("select count(*) from Users where UserName = '" + model.UserName + "'", cn);
string result = cmd.ExecuteScalar().ToString();
```

Secure Code: Parameterize the query using the standard ADO.NET placeholders and set the parameter value.

```
SqlCommand cmd = new SqlCommand("select count(*) from Users where UserName = @UserName", cn);
SqlParameter parm = new SqlParameter("@UserName", NVarChar);
parm.Value = model.UserName;
cmd.Parameters.Add(parm);
string result = cmd.ExecuteScalar().ToString();
```

Scan Results

SEC0107 ran successfully without any identified instances.

✔ SEC0110: Unvalidated Web Forms Redirect

Description

Passing unvalidated redirect locations to the *Response.Redirect* method can allow attackers to send users to malicious web sites. This can allow attackers to perform phishing attacks and distribute malware to victims.

Avoid performing redirect actions with user controllable data (e.g. request parameters). Consider validating redirect paths to allow relative paths inside of the application and deny absolute paths. All absolute paths must be validated against an approved list of external domains prior to redirecting the user.

References

- CWE-601: URL Redirection to Untrusted Site ('Open Redirect')
- https://msdn.microsoft.com/en-us/library/a8wa7sdt(v=vs.110).aspx

Code Example(s)

Insecure Code: The following example shows the *Response.Redirect* method called using an unvalidated request parameter.

```
protected void LoginUser_LoggedIn(object sender, EventArgs e)
{
    if (Request.QueryString["ReturnUrl"] != null)
        Response.Redirect(Request.QueryString["ReturnUrl"]);
}
```

Secure Code: Validate the return URL is a relative path (not absolute) to ensure an attacker cannot redirect the user to a malicious external domain.

```
protected void LoginUser_LoggedIn(object sender, EventArgs e)
{
    Uri targetUri = null;

    if (Uri.TryCreate(Request.QueryString["ReturnUrl"], UriKind.Relative, out targetUri))
    {
        Response.Redirect(targetUri.ToString());
    }
    else
    {
        Response.Redirect("~/default.aspx");
    }
}
```

Scan Results

SEC0110 ran successfully without any identified instances.

✔ SEC0112: Path Tampering: Unvalidated File Path

Description

Path traversal vulnerabilities occur when an application does not properly validate file paths for directory traversal (../) and other malicious characters. This can allow attackers to download, overwrite, or delete unauthorized files from the server. Ensure file paths are read from a trusted location, such as a static resource or configuration file. Do not send file paths in request parameters, which can be modified by an attacker.

SEC0112 scans the *System.IO.FileStream* API that is commonly called with dynamic file path data.

Dynamic file paths passed to a file stream require strict validation. Ensure file paths are read from a trusted location, such as a static resource or configuration file. Do not send file paths in request parameters, which can be modified by an attacker to contain dangerous characters (../).

References

- CWE-23: Relative Path Traversal
- https://msdn.microsoft.com/en-us/library/system.io.filestream(v=vs.110).aspx

Code Example(s)

Insecure Code: The following example shows a *FileStream* being constructed from a dynamic parameter to determine the file path location.

```
public ActionResult Index(string fileName)
{
    using (Stream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read, FileShare.Read))
    {
        return new FileStreamResult(stream, fileName);
    }
}
```

Secure Code: Avoid using untrusted values when constructing a file path. Instead, store file paths in a trusted location, such as a configuration file, and use a unique identifier to construct the file name.

```
public ActionResult Index(Guid fileId)
{
    string path = Path.Combine(ConfigurationManager.AppSettings["DownloadPath"], fileId.ToString());

    //NOTE: YOU MAY STILL NEED TO PERFORM ENTITLEMENT AUTHORIZATION BEFORE RETURNING THE FILE

    using (Stream stream = new FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read))
    {
        return new FileStreamResult(stream, fileName);
    }
}
```

Scan Results

SEC0112 ran successfully without any identified instances.

✔ SEC0114: LDAP Injection Directory Entry

Description

LDAP Injection vulnerabilities occur when untrusted data is concatenated into a LDAP Path or Filter expression without properly escaping control characters. This can allow attackers to change the meaning of an LDAP query and gain access to resources for which they are not authorized.

Fixing the LDAP Injection Directory Entry vulnerability requires untrusted data to be encoded using the appropriate Web Protection Library (aka AntiXSS) LDAP encoding method:

- Encoder.LdapDistinguishedNameEncode()

References

- CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
- https://www.owasp.org/index.php/Top_10-2017_A1-Injection
- https://msdn.microsoft.com/en-us/library/system.directoryservices.directoryentry(v=vs.110).aspx
- https://msdn.microsoft.com/en-us/library/system.directoryservices.directorysearcher(v=vs.110).aspx
- https://wpl.codeplex.com/

Code Example(s)

Insecure Code: The following example shows untrusted data being concatenated into directory entry constructor path without escaping LDAP control characters.

```
DirectoryEntry entry = new DirectoryEntry(string.Format("LDAP://DC={0}, DC=COM/", model.Domain));
DirectorySearcher searcher = new DirectorySearcher(entry);
searcher.SearchScope = SearchScope.Subtree;
searcher.Filter = "(name={BobbyTables})";
SearchResultCollection resultCollection = searcher.FindAll();
```

Secure Code: Encode untrusted data passed to the DirectoryEntry path argument with the Web Protection Library's (aka AntiXSS) LdapDistinguishedNameEncode method.

```
DirectoryEntry entry = new DirectoryEntry(string.Format("LDAP://DC={0}, DC=COM/", Encoder.LdapDistinguishedNameEncode(model
DirectorySearcher searcher = new DirectorySearcher(entry);
searcher.SearchScope = SearchScope.Subtree;
searcher.Filter = "(name={BobbyTables})";
SearchResultCollection resultCollection = searcher.FindAll();
```

Scan Results

SEC0114 ran successfully without any identified instances.

✔ SEC0116: Path Tampering: Unvalidated File Path

Description

Path traversal vulnerabilities occur when an application does not properly validate file paths for directory traversal (../) and other malicious characters. This can allow attackers to download, overwrite, or delete unauthorized files from the server.

Ensure file paths are read from a trusted location, such as a static resource or configuration file. Do not send file paths in request parameters, which can be modified by an attacker to contain dangerous characters (../).

SEC0116 covers several APIs that are commonly called with dynamic file path data. Any of the following method calls require strict validation on the files being consumed by the API:

- System.IO.File
- Delete
- OpenText
- OpenWrite
- Read
- ReadAllBytes
- ReadAllLines
- ReadAllText
- ReadLines
- WriteAllBytes
- WriteAllLines
- WriteAllText

References

- CWE-23: Relative Path Traversal
- https://msdn.microsoft.com/en-us/library/system.io.file(v=vs.110).aspx

Code Example(s)

Insecure Code: The following example shows the *File.Delete* method called using a dynamic parameter from a REST service to construct the file path location.

```
public HttpResponseMessage Delete(string file)
{
    string path = Path.Combine(ConfigurationManager.AppSettings["DownloadPath"], file);
    File.Delete(path);
    return Request.CreateResponse(HttpStatusCode.OK);
}
```

Secure Code: Avoid using untrusted values when constructing a file path. Instead, store file paths in a trusted location, such as a configuration file, and use a unique identifier to construct the file name.

```
[HttpPost]
public HttpResponseMessage Delete(Guid fileId)
{
    string path = Path.Combine(ConfigurationManager.AppSettings["DownloadPath"], fileId.ToString());
    //NOTE: YOU STILL NEED TO AUTHORIZE THE USERS ENTITLEMENTS ON DELETING THIS FILE :)
    File.Delete(path);
    return Request.CreateResponse(HttpStatusCode.OK);
}
```

Scan Results

SEC0116 ran successfully without any identified instances.

✔ SEC0117: LDAP Injection Path Assignment

Description

LDAP Injection vulnerabilities occur when untrusted data is concatenated into a LDAP Path or Filter expression without properly escaping control characters. This can allow attackers to change the meaning of an LDAP query and gain access to resources for which they are not authorized.

Fixing the LDAP Injection Path Assignment vulnerability requires untrusted data to be encoded using the appropriate Web Protection Library (aka AntiXSS) LDAP encoding method:

- Encoder.LdapDistinguishedNameEncode()

References

- CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
- https://www.owasp.org/index.php/Top_10-2017_A1-Injection
- https://msdn.microsoft.com/en-us/library/system.directoryservices.directoryentry(v=vs.110).aspx
- https://msdn.microsoft.com/en-us/library/system.directoryservices.directorysearcher(v=vs.110).aspx]
- https://wpl.codeplex.com/

Code Example(s)

Insecure Code: The following example shows untrusted data being concatenated into the *DirectoryEntry* Path property without escaping LDAP control characters.

```
DirectoryEntry entry = new DirectoryEntry();
entry.Path = string.Format("LDAP://DC={0},DC=COM,CN=Users", model.Domain);
entry.Username = model.UserName;
entry.Password = model.Password;
DirectorySearcher searcher = new DirectorySearcher(entry);
searcher.SearchScope = SearchScope.Subtree;
searcher.Filter = $"(samaccountname=DOMAIN\\BobbyTables)";
SearchResult result = searcher.FindOne();
```

Secure Code: Encode untrusted data passed to the DirectoryEntry.Path property with the Web Protection Library's (aka AntiXSS) LdapDistinguishedNameEncode method.

```
DirectoryEntry entry = new DirectoryEntry();
entry.Path = string.Format("LDAP://DC={0},DC=COM,CN=Users", Encoder.LdapDistinguishedNameEncode(model.Domain));
entry.Username = model.UserName;
entry.Password = model.Password;
DirectorySearcher searcher = new DirectorySearcher(entry);
searcher.SearchScope = SearchScope.Subtree;
searcher.Filter = $"(samaccountname=DOMAIN\\BobbyTables)";
SearchResult result = searcher.FindOne();
```

Scan Results

SEC0117 ran successfully without any identified instances.

✔ SEC0118: LDAP Injection Directory Searcher

Description

LDAP Injection vulnerabilities occur when untrusted data is concatenated into a LDAP Path or Filter expression without properly escaping control characters. This can allow attackers to change the meaning of an LDAP query and gain access to resources for which they are not authorized.

Fixing the LDAP Injection Directory Searcher vulnerability requires untrusted data to be encoded using the Web Protection Library (aka AntiXSS) LDAP encoding method:

- Encoder.LdapFilterEncode()

References

- CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
- https://www.owasp.org/index.php/Top_10-2017_A1-Injection
- https://msdn.microsoft.com/en-us/library/system.directoryservices.directoryentry(v=vs.110).aspx
- https://msdn.microsoft.com/en-us/library/system.directoryservices.directorysearcher(v=vs.110).aspx
- https://wpl.codeplex.com/

Code Example(s)

Insecure Code: The following example shows untrusted data being concatenated into directory searcher constructor's Filter property without escaping LDAP control characters.

```
DirectoryEntry entry = new DirectoryEntry("LDAP://DC=example.com, DC=COM/");
DirectorySearcher searcher = new DirectorySearcher(entry, string.Format("(name={0})", model.UserName);
searcher.SearchScope = SearchScope.Subtree;
SearchResultCollection resultCollection = searcher.FindAll();
```

Secure Code: Encode untrusted data passed to the *DirectorySearcher* Filter argument with the Web Protection Library's (aka AntiXSS) LdapFilterEncode method.

```
DirectoryEntry entry = new DirectoryEntry("LDAP://DC=example.com, DC=COM/");
DirectorySearcher searcher = new DirectorySearcher(entry, string.Format("(name={0})", Encoder.LdapFilterEncode(model.UserNa
searcher.SearchScope = SearchScope.Subtree;
SearchResultCollection resultCollection = searcher.FindAll();
```

Scan Results

SEC0118 ran successfully without any identified instances.

✔ SEC0119: LDAP Injection Filter Assignment

Description

LDAP Injection vulnerabilities occur when untrusted data is concatenated into a LDAP Path or Filter expression without properly escaping control characters. This can allow attackers to change the meaning of an LDAP query and gain access to resources for which they are not authorized.

Fixing the LDAP Injection Filter Assignment vulnerability requires untrusted data to be encoded using the Web Protection Library (aka AntiXSS) LDAP encoding method:

- Encoder.LdapFilterEncode()

References

- CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
- https://www.owasp.org/index.php/Top_10-2017_A1-Injection
- https://msdn.microsoft.com/en-us/library/system.directoryservices.directoryentry(v=vs.110).aspx
- https://msdn.microsoft.com/en-us/library/system.directoryservices.directorysearcher(v=vs.110).aspx
- https://wpl.codeplex.com/

Code Example(s)

Insecure Code: The following example shows untrusted data being concatenated into the *DirectorySearcher* Filter property without escaping LDAP control characters.

```
DirectoryEntry entry = new DirectoryEntry("LDAP://DC=example.com, DC=COM");
DirectorySearcher searcher = new DirectorySearcher(entry);
searcher.SearchScope = SearchScope.Subtree;
searcher.Filter = string.Format("(name={0})", model.UserName);
SearchResultCollection resultCollection = searcher.FindAll();
```

Secure Code: Encode untrusted data passed to the *DirectorySearcher.Filter* property with the Web Protection Library's (aka AntiXSS) LdapFilterEncode method.

```
DirectoryEntry entry = new DirectoryEntry("LDAP://DC=example.com, DC=COM");
DirectorySearcher searcher = new DirectorySearcher(entry);
searcher.SearchScope = SearchScope.Subtree;
searcher.Filter = string.Format("(name={0})", Encoder.LdapFilterEncode(model.UserName));
SearchResultCollection resultCollection = searcher.FindAll();
```

Scan Results

SEC0119 ran successfully without any identified instances.

✔ SEC0121: CORS Allow Origin Wildcard

Description

Cross-Origin Resource Sharing (CORS) allows a service to disable the browser's Same-origin policy, which prevents scripts on an attacker-controlled domain from accessing resources and data hosted on a different domain. The CORS Access-Control-Allow-Origin HTTP header specifies the domain with permission to invoke a cross-origin service and view the response data. Configuring the Access-Control-Allow-Origin header with a wildcard (*) can allow code running on an attacker-controlled domain to view responses containing sensitive data.

SEC0121 identifies .NET CORS misconfigurations using the *AllowAnyOrigin()* method.

Avoid setting the Access-Control-Allow-Origin header to a wildcard (*). Instead, configure the service to validate the incoming Origin header value against a trusted list of domains. Return the incoming accepted domain in the Access-Control-Allow-Origin header value, otherwise default the Access-Control-Allow-Origin value to a known safe origin.

References

- CWE-942: Overly Permissive Cross-domain Whitelist
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin
- https://www.youtube.com/watch?reload=9&v=wgkj4ZgxI4c
- https://docs.microsoft.com/en-us/aspnet/core/security/cors?view=aspnetcore-2.2

Code Example(s)

Insecure Code: The following example shows a CORS configuration using the *AllowAnyOrigin* configuration, which sets the *Access-Control-Allow-Origin* header to the wildcard (*)

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    ...
    app.UseCors(builder => builder.AllowAnyOrigin());
    ...
}
```

Secure Code: The following example shows the *ConfigureServices* method creating a *_secureOrigin* policy with a trusted list of domains. Then, the *Configure* method configures the CORS policy to usee the *_secureOrigin* policy.

```
private readonly string secureOrigin = "_secureOrigin";

public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy(MyAllowSpecificOrigins,
        builder =>
        {
            builder.WithOrigins("https://www.pumasecurity.io",
                                "https://www.pumascan.com");
```

```
        });
    });
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   ...
   app.UseCors(secureOrigin);
   ...
}
```

Scan Results

SEC0121 ran successfully without any identified instances.

✔ SEC0122: JWT Signature Validation Disabled

Description

The JSON Web Tokens (JWT) header and payload values are base64 encoded, which can be decoded, tampered, and replayed to gain access to protected resources.

Web service APIs relying on JSON Web Tokens (JWT) for authentication and authorization must sign each JWT with a private key or secret. Each web service endpoint must require JWT signature validation prior to decoding and using the token to access protected resources.
In ASP.NET Core, configure the Authentication service's JwtBearer options to require signed tokens:

- **RequireSignedTokens**: Rejects JWTs that do not have a signature.

References

- CWE-347: Improper Verification of Cryptographic Signature
- https://cheatsheets.pragmaticwebsecurity.com/cheatsheets/jwt.pdf
- https://tools.ietf.org/html/rfc7519#section-11.2
- https://docs.microsoft.com/en-us/dotnet/api/microsoft.identitymodel.tokens.tokenvalidationparameters

Code Example(s)

Insecure Code: The following example shows the *TokenValidationParameters.RequireSignedTokens* property set to false.

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
  .AddJwtBearer(options =>
  {
      options.TokenValidationParameters = new TokenValidationParameters
      {
          [...]
          RequireSignedTokens = false,
      };
  });
```

Secure Code: Set the *TokenValidationParameters.RequireSignedTokens* property to true.

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
  .AddJwtBearer(options =>
  {
      options.TokenValidationParameters = new TokenValidationParameters
      {
          [...]
          RequireSignedTokens = true,
      };
  });
```

Scan Results

SEC0122 ran successfully without any identified instances.

✔ SEC0123: JWT Expiration Disabled

Description

JSON Web Tokens (JWT) payloads that do not have an Expiration Time (exp) will never expire, which allows an attacker to indefinitely replay a compromised token. Web service APIs relying on JSON Web Tokens (JWT) for authentication and authorization must enforce token expiration by requiring and validating the Expiration Time (exp) claim.

In ASP.NET Core, configure the Authentication service's JwtBearer options to require to require the exp claim and validate the token's lifetime:

- **RequireExpirationTime**: Requires tokens to have the 'exp' claim.
- **ValidateLifetime**: Requires the token's lifetime to be validated.

References

- CWE-613: Insufficient Session Expiration
- https://cheatsheets.pragmaticwebsecurity.com/cheatsheets/jwt.pdf
- https://tools.ietf.org/html/rfc7519#section-4.1.4
- https://docs.microsoft.com/en-us/dotnet/api/microsoft.identitymodel.tokens.tokenvalidationparameters

Code Example(s)

Insecure Code: The following example shows the *TokenValidationParameters RequireExpirationTime* and *ValidateLifetime* properties set to false.

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
  .AddJwtBearer(options =>
  {
      options.TokenValidationParameters = new TokenValidationParameters
      {
          [...]
          RequireExpirationTime = false,
          ValidateLifetime = false,
      };
  });
```

Secure Code: Configure the *TokenValidationParameters RequireExpirationTime* and *ValidateLifetime* properties to true.

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
  .AddJwtBearer(options =>
  {
      options.TokenValidationParameters = new TokenValidationParameters
      {
          [...]
          RequireExpirationTime = true,
          ValidateLifetime = true,
      };
  });
```

Scan Results

SEC0123 ran successfully without any identified instances.

✔ SEC0124: Weak Cipher Mode Padding

Description

Padding Oracle attacks can occur in applications that decrypt ciphertext values without an integrity check. This is commonly identified when decrypting untrusted values from an HTTP request or web service. Applications that meet the following criteria are vulnerable to padding oracle attacks:

- Data encryption uses a block cipher algorithm (e.g. 3DES or AES) in Cipher Block Chaining (CBC) mode.
- Error handling responds differently (e.g. error message, error code, or response time) when given invalid ciphertext with valid padding and invalid ciphertext with invalid padding (aka a side channel attack).

Protecting against padding oracle attacks requires applications to verify the ciphertext's integrity prior to decryption. To do this, create an HMAC of the ciphertext value with a private key and send both the ciphertext and HMAC values to the client. During decryption, read the ciphertext parameter, generate a new HMAC value, and compare the result to the original HMAC value. If the values HMAC match, proceed with decrypting the ciphertext. If the HMAC values do not match, respond with a consistent response code and error message to the client.

Alternatively, third-party encryption libraries provide support for stronger Cipher Modes: Counter Mode (CTR) and Galois Counter Mode (GCM), such as Bouncy Castle can be used instead of the native .NET cryptography libraries.

SEC0124 identifies usage of the CBC padding mode. It does not currently perform data flow analysis to see if the incoming bytes have been passed through an acceptable HMAC method to suppress finding. See the User Guide for details on creating false positive exceptions.

References

- CWE-347: Improper Verification of Cryptographic Signature
- https://www.owasp.org/index.php/Testing_for_Padding_Oracle_(OTG-CRYPST-002)
- https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.ciphermode
- http://resources.infosecinstitute.com/padding-oracle-attack-2/

Code Example(s)

Insecure Code: The following example shows the AES algorithm decrypting an untrusted parameter using the *CipherMode.CBC* mode without an integrity check.

```
//Perform integrity check on incoming data
byte[] ciphertext = Convert.FromBase64String(model.ProtectedData);

using (MemoryStream mStream = new MemoryStream())
{
    //Input bytes
    byte[] inputBytes = Encoding.UTF8.GetBytes(plainText);

    SymmetricAlgorithm alg = Aes.Create();
    alg.Mode = CipherMode.CBC;

    //Set key and iv
    alg.Key = GetKey();
```

```
    alg.IV = GetIv();

    //Create the crypto stream
    CryptoStream cStream = new CryptoStream(mStream
      , alg.CreateEncryptor()
      , CryptoStreamMode.Write);
    cStream.Write(inputBytes, 0, inputBytes.Length);
    cStream.FlushFinalBlock();
    cStream.Close();

    //Get the output
    output = mStream.ToArray();

    //Close resources
    mStream.Close();
    alg.Clear();
}
```

Secure Code: Set the AES algorithm mode set to *CipherMode.CBC* option and perform an integrity check on the incoming ciphertext.

```
//Perform integrity check on incoming data
string[] args = model.ProtectedData.Split('.');
byte[] ciphertext = Convert.FromBase64String(args[0]);
byte[] hmac = Convert.FromBase64String(args[1]);

HMACSHA256 hmac = new HMACSHA256(_KEY);
byte[] verification = hmac.ComputeHash(ciphertext);

if (!verification.SequenceEqual(hmac))
    throw new ArgumentException("Invalid signature detected.");

using (MemoryStream mStream = new MemoryStream())
{
    SymmetricAlgorithm alg = Aes.Create();
    alg.Mode = CipherMode.CBC;

    //Set key and iv
    alg.Key = GetKey();
    alg.IV = GetIv();

    //Create the crypto stream
    CryptoStream cStream = new CryptoStream(mStream
      , alg.CreateDecryptor()
      , CryptoStreamMode.Write);
    cStream.Write(ciphertext, 0, inputBytes.Length);
    cStream.FlushFinalBlock();
    cStream.Close();

    //Get the cleartext
    byte[] cleartext = mStream.ToArray();

    //Close resources
    mStream.Close();
    alg.Clear();
}
```

Scan Results

SEC0124 ran successfully without any identified instances.

✔ SEC0125: Weak RSA Key Length

Description

Due to advances in cryptanalysis attacks and cloud computing capabilities, the National Institute of Standards and Technology (NIST) deprecated 1024-bit RSA keys on January 1, 2011.

The Certificate Authority Browser Forum, along with the latest version of all browsers, currently mandates a minimum key size of 2048-bits for all RSA keys.

References

- CWE-326: Inadequate Encryption Strength
- https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

Code Example(s)

Insecure Code: The following example shows the *RSACryptoServiceProvider* initializing a 1024-bit key pair.

```
RSACryptoServiceProvider alg = new RSACryptoServiceProvider(1024);
```

Secure Code: Configure the *RSACryptoServiceProvider* with at least a 2048-bit key pair.

```
RSACryptoServiceProvider alg = new RSACryptoServiceProvider(2048);
```

Scan Results

SEC0125 ran successfully without any identified instances.

✔ SEC0126: SQL Injection: Dynamic NHibernate Query

Description

Concatenating untrusted data into a dynamic SQL string and calling vulnerable NHibernate Framework methods can allow SQL Injection. To ensure calls to vulnerable NHibernate Framework methods are parameterized, pass positional or named parameters in the statement. The following NHibernate methods allow for raw SQL queries to be executed:

- CreateQuery
- CreateSqlQuery

To ensure calls to vulnerable NHibernate methods are parameterized, use named parameters in the raw SQL query. Then, set the named parameter values when executing the query.

References

- CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- https://www.owasp.org/index.php/Top_10-2017_A1-Injection
- https://nhibernate.info/doc/nhibernate-reference/querysql.html

Code Example(s)

Insecure Code: The following example shows a request parameter being concatenated into a dynamic SQL statement, which is executed by the NHibernate *CreateQuery* method.

```
string q = "SELECT * FROM Items WHERE ProductCode = '" + model.ProductCode + "'";

var cfg = new Configuration();
ISessionFactory sessions = cfg.BuildSessionFactory();
ISession session = sessions.OpenSession();

var query = session.CreateQuery(q);
var product = query.List<Product>().FirstOrDefault();
```

Secure Code: Use NHibernate named parameters to prevent SQL Injection. The following examples shows the **:productCode** bind parameter

```
string q = "SELECT * FROM Items WHERE ProductCode = :productCode";

var cfg = new Configuration();
ISessionFactory sessions = cfg.BuildSessionFactory();
ISession session = sessions.OpenSession();

var query = session.CreateQuery(q);
var product = query
  .SetString("productCode", model.ProductCode)
  .List<Product>().FirstOrDefault();
```

Scan Results

SEC0126 ran successfully without any identified instances.

✔ SEC0127: XPath Injection

Description

Concatenating untrusted data into XPath queries allow attackers to change the meaning of the XPath query. The following XPath methods allow for raw queries to be executed:

- SelectNodes
- SelectSingleNode

.NET does not provide a built in library for parameterizing XPath queries. To prevent XPath Injection, ensure dynamic values are sanitized with strong input validation. Common methods include converting untrusted data to a strongly typed object (e.g. Int, Decimal, DateTime, etc.) or performing regular expression validation to restrict the characters (e.g. [A-za-z0-9]+).

For this reason, Puma Scan does not have a default cleanse method that allows SEC0127 findings to be eliminated. To remove a SEC0127 finding, register a Custom Cleanse Method that identifies your approved validation method for preventing XPath vulnerabilities.

References

- CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection')
- https://www.owasp.org/index.php/XPATH_Injection
- https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlnode.selectnodes
- https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlnode.selectsinglenode

Code Example(s)

Insecure Code: The following example shows an XPath query against a backend XML user store. The XPath query concatenates the *model.UserName* request parameter into the query.

```
public IActionResult Post(Model model)
{
    var doc = new XmlDocument { XmlResolver = null };
    doc.Load(configuration["UserXmlPath"]);
    var results = doc.SelectNodes("/Users/User[id='" + model.UserName + "']");

    //Process login
    [...]

    return View();
}
```

Secure Code: Before

```
public IActionResult Post(Model model)
{
    var userName = model.UserName;
```

```
    if (!Regex.IsMatch(userName, @"[A-za-z0-9]+"))
    {
        throw new ArgumentException("Invalid username parameter.");
    }

    var doc = new XmlDocument { XmlResolver = null };
    doc.Load(configuration["UserXmlPath"]);
    var results = doc.SelectNodes("/Users/User[id='" + userName + "']");

    //Process login
    [...]

    return View();
}
```

Scan Results

SEC0127 ran successfully without any identified instances.

✔ SEC0128: LDAP Authentication Disabled

Description

Disabling LDAP Authentication configures insecure connections to the backend LDAP provider. Using the *DirectoryEntry AuthenticationType* property's *Anonymous* or *None* option allows an anonymous or basic authentication connection to the LDAP provider.

Set the the *DirectoryEntry AuthenticationType* property to *Secure*, which requests Kerberos authentication under the security context of the calling thread or as a provider username and password.

References

- CWE-287: Improper Authentication
- https://docs.microsoft.com/en-us/dotnet/api/system.directoryservices.authenticationtypes

Code Example(s)

Insecure Code: The following example shows an *DirectoryEntry* with the *AuthenticationType* set to *Anonymous*, which means no authentication is performed.

```
DirectoryEntry entry = new DirectoryEntry("LDAP://DC=PUMA}, DC=COM/");
entry.AuthenticationType = AuthenticationTypes.Anonymous;
```

Secure Code: Set the *AuthenticationType* set to *Secure*, which requests Kerberos authentication under the security context of the calling thread or as a provider username and password.

```
DirectoryEntry entry = new DirectoryEntry("LDAP://DC=PUMA}, DC=COM/");
entry.AuthenticationType = AuthenticationTypes.Secure;
```

Scan Results

SEC0128 ran successfully without any identified instances.

✔ SEC0129: Server-side Request Forgery

Description

Server-side Request Forgery (SSRF) vulnerabilities occur when an application requests data from a URL supplied from an untrusted location (e.g. request parameter, web service API, database). Attackers can supply a malicious URL to the application, which can allow unauthorized access to secrets, database data, and files hosted on the server. Common examples include cloud meta-data endpoints (e.g. http://169.254.169.254/), database administrator REST APIs, or local file URIs (e.g. file://).

SEC0129 identifies untrusted URLs passed to the following libraries:

- System.Net.Http.HttpClient
- System.Net.HttpWebRequest

To prevent Server-side Request Forgery (SSRF) vulnerabilities, follow this pattern:

- Does the user really need to directly control the full *URL* passed to the application? The answer is usually No. Consider moving the *URL* value to server-side storage and look up the value using a non-injectable value such as a GUID, integer, etc. passed in from the request. For example, consider an application that allows the user to request data from two URIs:

URI	ID
https://pumasecurity.io/api/rules	06C67D8C-CAD5-4003-B065-1089CFF0D1E9
https://pumascan.com/api/rules	A180A8AF-C667-4074-A768-C95F13819E2A

  Requests pass in a valid GUID and the application selects the associated URL. If the GUID is not valid, throw an exception. This ensures that the user can only access approved URIs.
- Validate incoming *URLs* against a strict list of approved URIs reject everything else.

References

- CWE-918: Server-Side Request Forgery (SSRF)
- https://www.owasp.org/index.php/Server_Side_Request_Forgery
- https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient
- https://docs.microsoft.com/en-us/dotnet/api/system.net.httpwebrequest

Code Example(s)

Insecure Code: The following example shows a *url* request parameter being passed to the *HttpClient GetAsync* method without validating the URL against a list of trusted URIs.

```
public async IActionResult Get(string url)
{
   var client = new HttpClient();
   var request = client.GetAsync(url);
   var json = await result.Content.ReadAsStringAsync();
   return JsonConvert.DeserializeObject<GetResult>(json);
}
```

Secure Code:

```
public async IActionResult Get(Guid urlId)
{
   //Pull valid endpoints from the configuration file
   List<Endpoint> endpoints = GetEndpoints();

   //Verify the endpoint exists
   Endpoint e = endpoints.FirstOrDefault(i => i.Id == urlId);
   if (e == null)
     throw new ArgumentException("Invalid endpoint id.");

   var client = new HttpClient();
   var request = client.GetAsync(e.Url);
   var json = await result.Content.ReadAsStringAsync();
   return JsonConvert.DeserializeObject<GetResult>(json);
}
```

Scan Results

SEC0129 ran successfully without any identified instances.

✔ SEC0130: Hard-Coded Credential

Description

Hard-coding credentials in source code allows anyone with access to the source control repository or the binary files to obtain the credential. Rotating hard-coded secrets is also difficult because the application or service must be redeployed to change the value. SEC0130 detects hard-coded credentials used to create the following:

- System.Net.NetworkCredential
- Microsoft.IdentityModel.Clients.ActiveDirectory.ClientCredential

Rather than storing credentials in source code, store secrets in a dedicated secrets management system (Azure Key Vault, Amazon KMS, Google KMS, Hashicorp Vault) separate from the application or service consuming the secret values.

References

- CWE-798: Use of Hard-coded Credentials
- https://www.owasp.org/index.php/Use_of_hard-coded_password
- https://docs.microsoft.com/en-us/dotnet/api/system.net.networkcredential
- https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets
- https://docs.microsoft.com/en-us/aspnet/core/security/key-vault-configuration

Code Example(s)

Insecure Code: The following example shows a hard-coded SMTP *NetworkCredential*.

```
using (SmtpClient client = new SmtpClient())
{
    client.Credentials = new NetworkCredential("noreply@pumasecurity.io"
      , "supersecretpassword");
}
```

Secure Code: Start by burning the credential. Read the new network credential from an external secrets management system, such as Azure Vault.

```
using (SmtpClient client = new SmtpClient())
{
    client.Credentials = new NetworkCredential("noreply@pumasecurity.io"
      , getVaultSecret("smtp/noreply/credential"));
}
```

Scan Results

SEC0130 ran successfully without any identified instances.

✔ SEC0131: Hard-Coded Secret

Description

Hard-coding secrets such as encryption keys, initialization vectors, and passwords in source code allows anyone with access to the source control repository or the binary files to obtain the values. Rotating hard-coded secrets is also difficult because the application or service must be redeployed to change the value.
SEC0131 searches for hard-coded variable names matching the following regular expressions. See the Rule Options to customize the search expressions.

- ^[Kk][Ee][Yy]$
- ^[Ii][Vv]$
- ^[Pp][Aa][Ss][Ss][Ww][Oo][Rr][Dd]$

Rather than storing secrets in source code, store secrets in a dedicated secrets management system (Azure Key Vault, Amazon KMS, Google KMS, Hashicorp Vault) separate from the application or service consuming the secret values.

References

- CWE-798: Use of Hard-coded Credentials
- https://www.owasp.org/index.php/Use_of_hard-coded_password
- https://docs.microsoft.com/en-us/dotnet/api/system.net.networkcredential
- https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets
- https://docs.microsoft.com/en-us/aspnet/core/security/key-vault-configuration

Code Example(s)

Insecure Code: The following example shows a hard-coded encryption key and initialization vector.

```
public class Symmetric
{
   private readonly string KEY = "D87D016B70393029";
   private readonly byte[] IV = { 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110 };
   ...
}
```

Secure Code: Start by burning the hard-coded values. Read the new symmetric encryption key from the Vault, and generate a random IV for each encryption operation.

```
public class Symmetric
{
   private readonly byte[] KEY;
   private readonly byte[] IV;
   public Symmetric()
   {
      KEY = getVaultSecret("puma/services/key");
      IV = getCryptoRandomIV();
   }
```

```
    ...
}
```

Scan Results

SEC0131 ran successfully without any identified instances.

Scan Configuration

The Puma Scan configuration options affect the scanner's performance and the quality of the findings. The following sections capture the settings used in this scan that could affect the results. Read more.

General Settings

Data Flow Analysis: Enabled

Puma Scan performs data flow analysis in many analyzers to determine if the source of an input comes from an untrusted source (e.g. request parameter, web service API, etc.) to help reduce false positives. This setting turns the data flow feature on (true) or off (false). If you are experiencing performance issues with Puma Scan, disabling this feature will improve performance. Disabling this feature will also produce more findings (which may be false positives) for analysis. The default value is true.

Data Flow Report Indeterminates: Disabled

Puma Scan performs data flow analysis in many analyzers to determine if the source of an input comes from an untrusted source (e.g. request parameter, web service API, etc.). In some cases, the data flow analyzer may be unable to perform a complete trace and cannot confidently determine if a vulnerability exists. These sinks are marked as indeterminate. This setting tells the scanner if indeterminate issues should be reported in the scan results (true) or be suppressed by the scanner (false). The default is false.

Disabled Rules

Each Puma Scan analyzer (SEC###) provides a configuration option to turn the rule on (true) or off (false). The default value is true. The following rule(s) were disabled during this scan:

```
SEC0009
```

Custom Cleanse Methods

Puma Scan rules performing data flow analysis will see data passing through each method in the call chain. Data passing through valid cleanse methods will cause the vulnerability to be suppressed from the scan results. The following custom cleanse method(s) were registered during this scan:


```
{
  "RuleIds": [
    "SEC0111"
  ],
  "Flag": "Web",
  "Syntax": "InvocationExpressionSyntax",
  "Namespace": "Puma.Prey.Common.Validation",
  "Type": "Validator",
  "Method": "IsValidFileName"
}
```


Suppressed False Positives

Puma Scan provides security analysts and engineers the ability to suppress false positives in an exceptions list. The following false positive(s) were registered during this scan:


```
{
  "RuleIds": [
    "SEC0013"
  ],
  "Path": "Skunk\\Web.config",
  "StartLine": "10",
  "EndLine": "10",
  "Pattern": "",
  "Expires": null,
  "Checksum": "",
  "ApprovedBy": "Eric Johnson",
  "Reason": "Sensitive data is not stored in the ViewState object.",
  "Timestamp": "2018-10-25T22:27:58.6444584Z"
}
```