

ATTACKING .NET

MODERN SYSTEM APPLICATION
VULNERABILITIES COMPREHENSIVE
ANALYSIS



Attacking .NET

• May 20, 2024 •  31 min read

Table of contents

Code Access Security (CAS)

- › 1. Overly Permissive Policies
- › 2. Insecure Code Execution
- › 3. Misconfigured Evidence-Based Security

AllowPartiallyTrustedCaller attribute (APTCA)

- › 1. Elevation of Privilege
- › 2. Data Leakage

Distributed Component Object Model (DCOM)

- › 1. Authentication and Authorization
- › 2. Data Integrity and Confidentiality
- › 3. Denial of Service (DoS) Attacks
- › Using DCOMCNFG Utility
- › Programmatic Configuration in .NET

Timing vulnerabilities with CBC-mode symmetric

- › Encrypt-then-HMAC Implementation
- › Secure Encryption and Decryption Example

Race Conditions

- › Race Conditions in the Dispose Method
- › Race Conditions in Constructors

- › Race Conditions with Cached Objects

- › Race Conditions in Finalizers

App Secrets

- › Environment Variables

- › Secret Manager

- › Enable Secret Storage

- › Set, Access, and Manage Secrets

XML Processing

- › .NET Framework Options:

- › Win32 and COM-based Options:

Timing attacks

ViewState is love

Formatter Attacks

TemplateParser

ObjRefs

- › Leaking ObjRefs

- › Trust Issues

- › Known Suspects

- › Exploitation

Resources

Show less ^

Attacking .NET applications often involves exploiting weaknesses in the code or the runtime environment. One common attack vector is through input validation flaws, where an attacker inputs malicious data into an application, potentially leading to issues like SQL injection or cross-site scripting (XSS). For example, if a .NET

application does not properly sanitize user inputs before executing database queries, an attacker could inject SQL commands that manipulate the database, retrieve sensitive information, or cause data loss. Similarly, XSS attacks can occur if the application fails to properly encode user inputs before rendering them in the web browser, allowing attackers to execute malicious scripts in the context of other users' sessions.

Another significant aspect of attacking .NET applications involves reverse engineering and tampering with the compiled code. .NET applications are typically compiled into an intermediate language (IL) which can be decompiled back into readable source code with tools like ILSpy or dotPeek. This makes it easier for attackers to understand the application logic, discover hardcoded secrets, or modify the code to change its behavior. Additionally, attackers might exploit vulnerabilities in the .NET runtime itself, such as through deserialization attacks, where untrusted data is deserialized into objects, leading to arbitrary code execution. Securing .NET applications requires thorough input validation, proper use of cryptography, and adopting secure coding practices to mitigate these risks.

Code Access Security (CAS)

While Code Access Security (CAS) was designed to improve the security of .NET applications by restricting code permissions, several security issues can arise if it is not correctly implemented. Understanding these issues and how to mitigate them is crucial for maintaining secure applications.

1. Overly Permissive Policies

One common issue is the creation of overly permissive security policies. Granting too many permissions to code, especially third-party or less trusted components, can expose your application to various attacks.

COPY 

```
using System;  
using System.Security.Permissions;
```

```

class OverlyPermissiveExample
{
    public static void Main()
    {
        try
        {
            // This grants unrestricted access, which is dangerous
            PermissionSet allPerms = new
PermissionSet(PermissionState.Unrestricted);
            allPerms.Demand();

            Console.WriteLine("All permissions granted.");
        }
        catch (SecurityException se)
        {
            Console.WriteLine($"SecurityException: {se.Message}");
        }
    }
}

```

Mitigation: Always grant the minimum permissions necessary for the code to function. Use `PermissionSet` objects to precisely control the granted permissions.

```

using System;
using System.Security;
using System.Security.Permissions;

class LeastPrivilegeExample
{
    public static void Main()
    {
        try
        {
            // Grant only the necessary permissions

```

COPY 

```

        PermissionSet perms = new
PermissionSet(PermissionState.None);
        perms.AddPermission(new
FileIOPermission(FileIOPermissionAccess.Read, @"C:\example.txt"));
        perms.Demand();

        Console.WriteLine("Minimum necessary permissions
granted.");
    }
    catch (SecurityException se)
    {
        Console.WriteLine($"SecurityException: {se.Message}");
    }
}
}

```

2. Insecure Code Execution

Improper use of CAS can lead to insecure code execution, allowing attackers to execute arbitrary code with higher privileges.

Example of insecure execution:

```

using System;
using System.Reflection;
using System.Security;
using System.Security.Permissions;

class InsecureReflectionExample
{
    public static void Main()
    {
        try
        {
            // Execute a method using reflection without proper

```

COPY 


```

permission checks
    Type type = Type.GetType("System.IO.File, mscorlib");
    MethodInfo method = type.GetMethod("Delete", new Type[] {
typeof(string) });
    method.Invoke(null, new object[] { @"C:\example.txt" });

    Console.WriteLine("File deleted using reflection.");
}
catch (SecurityException se)
{
    Console.WriteLine($"SecurityException: {se.Message}");
}
}
}

```

Mitigation: Restrict reflection and dynamic code execution permissions. Always validate and sanitize inputs when using reflection.

COPY 

```

using System;
using System.Reflection;
using System.Security;
using System.Security.Permissions;

class SecureReflectionExample
{
    [ReflectionPermission(SecurityAction.Demand, Flags =
ReflectionPermissionFlag.RestrictedMemberAccess)]
    public static void Main()
    {
        try
        {
            // Restrict reflection permissions
            Type type = Type.GetType("System.IO.File, mscorlib");
            MethodInfo method = type.GetMethod("Delete", new Type[] {
typeof(string) });

```

```

        // Ensure only authorized code can perform the action
        if (method != null && method.IsPublic)
        {
            method.Invoke(null, new object[] { @"C:\example.txt"
});
            Console.WriteLine("File deleted using secure
reflection.");
        }
    }
    catch (SecurityException se)
    {
        Console.WriteLine($"SecurityException: {se.Message}");
    }
}
}

```

3. Misconfigured Evidence-Based Security

Evidence-based security can be misconfigured, leading to incorrect assignment of permissions to code. If the evidence is not properly validated, malicious code could gain more privileges than intended.

Example of misconfigured evidence:

```

using System;
using System.Security;
using System.Security.Policy;
using System.Security.Permissions;

class MisconfiguredEvidenceExample
{
    public static void Main()
    {
        try

```

COPY 


```

    {
        // Incorrectly assigning evidence
        Evidence evidence = new Evidence();
        evidence.AddHostEvidence(new
Url("http://untrustedsource.com"));

        PermissionSet perms = new
PermissionSet(PermissionState.None);
        perms.AddPermission(new
FileIOPermission(FileIOPermissionAccess.Read, @"C:\example.txt"));

        // Incorrectly assuming the evidence is trusted
        if
(perms.IsSubsetOf(AppDomain.CurrentDomain.PermissionSet))
        {
            Console.WriteLine("Permissions incorrectly granted
based on evidence.");
        }
    }
    catch (SecurityException se)
    {
        Console.WriteLine($"SecurityException: {se.Message}");
    }
}
}

```

Mitigation: Ensure evidence is accurately and securely assessed before granting permissions. Use strong evidence such as digital signatures rather than relying solely on URLs or other easily spoofed sources.

COPY 

```

using System;
using System.Security;
using System.Security.Policy;
using System.Security.Permissions;

```

```

class SecureEvidenceExample
{
    public static void Main()
    {
        try
        {
            // Properly validate and assign evidence
            Evidence evidence = new Evidence();
            evidence.AddHostEvidence(new Zone(SecurityZone.Intranet));

            PermissionSet perms = new
PermissionSet(PermissionState.None);
            perms.AddPermission(new
FileIOPermission(FileIOPermissionAccess.Read, @"C:\example.txt"));

            // Check if the evidence is correctly trusted
            if (evidence.GetHostEvidence<Zone>().SecurityZone ==
SecurityZone.Intranet)
            {
                perms.Demand();
                Console.WriteLine("Permissions correctly granted based
on secure evidence.");
            }
        }
        catch (SecurityException se)
        {
            Console.WriteLine($"SecurityException: {se.Message}");
        }
    }
}

```

AllowPartiallyTrustedCaller attribute (APTCA)

The `AllowPartiallyTrustedCallersAttribute` (APTCA) in .NET is a powerful attribute that can introduce significant security risks if not properly understood and managed.

This attribute allows assemblies marked with it to be called by partially trusted code, such as code from the internet or other less trusted sources. While it can be useful in scenarios where an assembly needs to be accessible by a wide range of clients, it can also expose the assembly to various security vulnerabilities.

1. **Partially Trusted Code:** Code that does not have full permissions to execute, often due to its origin or the security policies applied to it.
2. **Fully Trusted Code:** Code that has unrestricted permissions to execute and access resources on the system.
3. **APTCA:** An attribute that allows partially trusted code to call into an assembly, potentially exposing it to misuse.

1. Elevation of Privilege

Allowing partially trusted callers can lead to an elevation of privilege if the assembly is not designed to handle untrusted input securely. This can happen if internal methods assume that all callers are fully trusted and therefore do not perform necessary security checks.

Example of a vulnerability:

```
using System;
using System.Security;

[assembly: AllowPartiallyTrustedCallers]

public class SecuritySensitiveClass
{
    public void SensitiveOperation()
    {
        // Perform a sensitive operation that assumes full trust
        Console.WriteLine("Sensitive operation performed.");
    }
}
```

COPY 

```
}  
}
```

In this example, a sensitive operation is exposed to partially trusted callers, potentially leading to security breaches.

Mitigation:

- Implement rigorous input validation and security checks.
- Use the `SecurityCritical` attribute to mark methods or classes that should only be accessed by fully trusted code.

```
using System;  
using System.Security;  
  
[assembly: AllowPartiallyTrustedCallers]  
  
public class SecuritySensitiveClass  
{  
    [SecurityCritical]  
    public void SensitiveOperation()  
    {  
        // Perform a sensitive operation that should only be accessed  
        by fully trusted code  
        Console.WriteLine("Sensitive operation performed.");  
    }  
}
```

COPY 

2. Data Leakage

Partially trusted callers may exploit exposed APIs to access sensitive data that should be restricted.

Example of data leakage:

COPY 

```
using System;
using System.Security;

[assembly: AllowPartiallyTrustedCallers]

public class SensitiveDataClass
{
    public string GetSensitiveData()
    {
        // Return sensitive data
        return "Sensitive data";
    }
}
```

In this example, a method returning sensitive data is exposed to partially trusted callers.

Mitigation:

- Restrict access to sensitive methods using security attributes.
- Ensure that sensitive data is not exposed to partially trusted code unless absolutely necessary.

COPY 

```
using System;
using System.Security;

[assembly: AllowPartiallyTrustedCallers]

public class SensitiveDataClass
{
```

```
[SecurityCritical]
public string GetSensitiveData()
{
    // Return sensitive data that should only be accessed by fully
trusted code
    return "Sensitive data";
}
}
```

To manage and inspect the APTCA settings for assemblies, developers can use tools like the .NET Framework Configuration Tool (`caspol`) or PowerShell.

Inspecting APTCA Settings:

- Using `caspol`:

```
caspol -listgroups
```

COPY 

Using PowerShell to check if an assembly is marked with APTCA:

```
[System.Reflection.Assembly]::LoadFile("C:\path\to\your\assembly.dll")
.GetCustomAttributes($true) | Where-Object { $_ -is
[System.Security.AllowPartiallyTrustedCallersAttribute] }
```

COPY 

Distributed Component Object Model (DCOM)

Distributed Component Object Model (DCOM) is a Microsoft technology that allows software components to communicate over a network. While DCOM facilitates distributed computing and can be very powerful, it also introduces several security

issues, especially in the context of .NET applications. Understanding these security issues and how to mitigate them is crucial for developers working with DCOM.

1. Authentication and Authorization

DCOM allows remote procedure calls over a network, which can be intercepted or exploited if proper authentication and authorization mechanisms are not in place. Without proper security, unauthorized users can gain access to sensitive operations or data.

Mitigation:

- Use strong authentication methods such as NTLM or Kerberos.
- Implement role-based access control (RBAC) to ensure that only authorized users can invoke methods on DCOM objects.

2. Data Integrity and Confidentiality

Data transmitted over DCOM can be intercepted by attackers if it is not properly encrypted, leading to data tampering or leakage.

Mitigation:

- Use encryption to protect data in transit. This can be achieved by configuring DCOM to use secure protocols.
- Ensure that sensitive data is encrypted before being sent over the network.

3. Denial of Service (DoS) Attacks

DCOM services can be susceptible to DoS attacks, where an attacker sends a large number of requests to overwhelm the server, making it unavailable to legitimate users.

Mitigation:

- Implement rate limiting to restrict the number of requests a client can make in a given period.
- Use network firewalls and intrusion detection/prevention systems to detect and block malicious traffic.

DCOM security settings can be configured using the DCOMCNFG utility or programmatically through .NET code.

Using DCOMCNFG Utility

1. Launch DCOMCNFG:

```
dcomcnfg
```

COPY 

2. Configure Application Security:

- In the DCOMCNFG utility, navigate to "Component Services" > "Computers" > "My Computer" > "DCOM Config".
- Select the application you want to configure.
- Right-click and select "Properties".
- Go to the "Security" tab and configure the required permissions for "Launch and Activation", "Access", and "Configuration".

Programmatic Configuration in .NET

You can use .NET to programmatically configure DCOM security settings for a specific application. Below is an example demonstrating how to configure DCOM settings:

```
using System;  
using System.Runtime.InteropServices;
```

COPY 


```

class DcomSecurityExample
{
    // Constants for DCOM security settings
    const int RPC_C_AUTHN_LEVEL_DEFAULT = 0;
    const int RPC_C_IMP_LEVEL_IDENTIFY = 2;
    const int EOAC_NONE = 0;

    [DllImport("ole32.dll")]
    private static extern int CoInitializeSecurity(
        IntPtr pSecDesc,
        int cAuthSvc,
        IntPtr asAuthSvc,
        IntPtr pReserved1,
        int dwAuthnLevel,
        int dwImpLevel,
        IntPtr pAuthList,
        int dwCapabilities,
        IntPtr pReserved3
    );

    public static void ConfigureDcomSecurity()
    {
        // Initialize COM security settings
        int result = CoInitializeSecurity(
            IntPtr.Zero,
            -1,
            IntPtr.Zero,
            IntPtr.Zero,
            RPC_C_AUTHN_LEVEL_DEFAULT,
            RPC_C_IMP_LEVEL_IDENTIFY,
            IntPtr.Zero,
            EOAC_NONE,
            IntPtr.Zero
        );

        if (result != 0)
        {

```

```

        throw new Exception($"CoInitializeSecurity failed with
error code {result}");
    }

    Console.WriteLine("DCOM security configured successfully.");
}

static void Main()
{
    try
    {
        ConfigureDcomSecurity();
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error: {ex.Message}");
    }
}
}

```

Timing vulnerabilities with CBC-mode symmetric

Timing vulnerabilities in CBC-mode symmetric decryption with padding can expose applications to padding oracle attacks. This type of attack allows an attacker to decrypt ciphertext without knowing the encryption key by exploiting the timing differences in decryption error messages.

A padding oracle attack leverages the difference in processing time or error messages to determine if the padding of decrypted data is valid. When using block ciphers with modes like CBC (Cipher Block Chaining), data often needs to be padded to fit the block size. Common padding schemes include PKCS#7, where padding bytes are added to make the final block fit the block size.

Let's illustrate a basic decryption process that is vulnerable to padding oracle attacks:

```
using System;
using System.IO;
using System.Security.Cryptography;

public class VulnerableDecryptor
{
    private static readonly byte[] Key =
Convert.FromBase64String("your-base64-key-here");
    private static readonly byte[] IV =
Convert.FromBase64String("your-base64-iv-here");

    public static string Decrypt(byte[] cipherText)
    {
        using (Aes aes = Aes.Create())
        {
            aes.Key = Key;
            aes.IV = IV;
            aes.Mode = CipherMode.CBC;
            aes.Padding = PaddingMode.PKCS7;

            try
            {
                using (ICryptoTransform decryptor =
aes.CreateDecryptor(aes.Key, aes.IV))
                    using (MemoryStream ms = new MemoryStream(cipherText))
                        using (CryptoStream cs = new CryptoStream(ms,
decryptor, CryptoStreamMode.Read))
                            using (StreamReader reader = new StreamReader(cs))
                            {
                                return reader.ReadToEnd();
                            }
            }
            catch (CryptographicException)
            {
                return null; // Invalid padding
            }
        }
    }
}
```

```
}  
}  
}
```

In this example, if an attacker can detect whether the padding is valid or not based on the response time or an error message, they can perform a padding oracle attack.

To prevent padding oracle attacks, it is essential to use an HMAC (Hash-based Message Authentication Code) to verify the integrity of the encrypted data before attempting decryption. This approach ensures that any tampered data is rejected before decryption, making timing attacks infeasible.

Encrypt-then-HMAC Implementation

Here's how to implement encrypt-then-HMAC:

1. **Encrypt the data.**
2. **Compute an HMAC of the ciphertext.**
3. **Append the HMAC to the ciphertext.**

When decrypting, the steps are:

1. **Verify the HMAC.**
2. **Decrypt the data only if the HMAC is valid.**

Secure Encryption and Decryption Example

```
using System;  
using System.IO;  
using System.Security.Cryptography;  
using System.Text;  
  
public class SecureEncryptor
```

COPY 

```

{
    private static readonly byte[] Key =
Convert.FromBase64String("your-base64-key-here");
    private static readonly byte[] IV =
Convert.FromBase64String("your-base64-iv-here");
    private static readonly byte[] HmacKey =
Convert.FromBase64String("your-hmac-key-here");

    public static byte[] EncryptAndHmac(string plainText)
    {
        using (Aes aes = Aes.Create())
        {
            aes.Key = Key;
            aes.IV = IV;
            aes.Mode = CipherMode.CBC;
            aes.Padding = PaddingMode.PKCS7;

            using (ICryptoTransform encryptor =
aes.CreateEncryptor(aes.Key, aes.IV))
            using (MemoryStream ms = new MemoryStream())
            using (CryptoStream cs = new CryptoStream(ms, encryptor,
CryptoStreamMode.Write))
            using (StreamWriter writer = new StreamWriter(cs))
            {
                writer.Write(plainText);
                writer.Flush();
                cs.FlushFinalBlock();

                byte[] cipherText = ms.ToArray();
                byte[] hmac = ComputeHmac(cipherText);

                byte[] result = new byte[cipherText.Length +
hmac.Length];
                Buffer.BlockCopy(cipherText, 0, result, 0,
cipherText.Length);
                Buffer.BlockCopy(hmac, 0, result, cipherText.Length,
hmac.Length);
            }
        }
    }
}

```

```

        return result;
    }
}

public static string DecryptAndVerifyHmac(byte[]
cipherTextWithHmac)
{
    byte[] cipherText = new byte[cipherTextWithHmac.Length - 32];
    // assuming SHA-256 HMAC
    byte[] hmac = new byte[32];

    Buffer.BlockCopy(cipherTextWithHmac, 0, cipherText, 0,
cipherText.Length);
    Buffer.BlockCopy(cipherTextWithHmac, cipherText.Length, hmac,
0, hmac.Length);

    byte[] computedHmac = ComputeHmac(cipherText);
    if (!CryptographicEquals(computedHmac, hmac))
    {
        throw new CryptographicException("Invalid HMAC.");
    }

    using (Aes aes = Aes.Create())
    {
        aes.Key = Key;
        aes.IV = IV;
        aes.Mode = CipherMode.CBC;
        aes.Padding = PaddingMode.PKCS7;

        using (ICryptoTransform decryptor =
aes.CreateDecryptor(aes.Key, aes.IV))
            using (MemoryStream ms = new MemoryStream(cipherText))
                using (CryptoStream cs = new CryptoStream(ms, decryptor,
CryptoStreamMode.Read))
                    using (StreamReader reader = new StreamReader(cs))

```

```

        {
            return reader.ReadToEnd();
        }
    }
}

private static byte[] ComputeHmac(byte[] data)
{
    using (HMACSHA256 hmac = new HMACSHA256(HmacKey))
    {
        return hmac.ComputeHash(data);
    }
}

private static bool CryptographicEquals(byte[] a, byte[] b)
{
    if (a.Length != b.Length) return false;
    int diff = 0;
    for (int i = 0; i < a.Length; i++)
    {
        diff |= a[i] ^ b[i];
    }
    return diff == 0;
}
}

```

Race Conditions

Race conditions in .NET applications, particularly concerning security vulnerabilities, can arise in various scenarios. Let's explore these race conditions and their potential impacts, along with best practices to mitigate them.

Race Conditions in the Dispose Method

When implementing the `Dispose` method of a class, synchronization is crucial to avoid race conditions. Consider this example:

```
void Dispose()
{
    if (myObj != null)
    {
        Cleanup(myObj);
        myObj = null;
    }
}
```

Without proper synchronization, it's possible for multiple threads to enter the `Dispose` method simultaneously, leading to `Cleanup` being called more than once. This can result in improper disposal of resources, potentially causing security vulnerabilities, especially with resource handles like files.

Race Conditions in Constructors

In scenarios where other threads might access class members before constructors complete, race conditions can arise. It's essential to review all class constructors to ensure there are no security concerns if this occurs. Synchronizing threads may be necessary to prevent such issues.

Race Conditions with Cached Objects

Code that caches security information or utilizes code access security operations can be vulnerable to race conditions if not properly synchronized. Consider the following example:

```
void SomeSecureFunction()
{
    if (SomeDemandPasses())
    {
        fCallersOk = true;
        DoOtherWork();
    }
}
```



```

        fCallersOk = false;
    }
}

void DoOtherWork()
{
    if (fCallersOk)
    {
        DoSomethingTrusted();
    }
    else
    {
        DemandSomething();
        DoSomethingTrusted();
    }
}

```

If `DoOtherWork` can be called from another thread with the same object, an untrusted caller might bypass a security demand. This highlights the importance of proper synchronization, especially when caching security-related information.

Race Conditions in Finalizers

Objects with finalizers that reference static or unmanaged resources can encounter race conditions. If multiple objects share a resource manipulated in a class's finalizer, all access to that resource must be synchronized to prevent race conditions.

App Secrets

When developing [ASP.NET](#) Core applications, managing sensitive data like passwords and API keys is crucial. Storing such information directly in code or configuration files is insecure and not recommended. Instead, [ASP.NET](#) Core provides mechanisms for safely managing app secrets during development. Let's explore how to handle app secrets securely in [ASP.NET](#) Core applications.

Environment Variables

Environment variables are commonly used to store sensitive data, as they override configuration values from other sources. However, it's important to note that environment variables are generally stored in plain text, which could pose a security risk if the system is compromised.

Secret Manager

The Secret Manager tool in [ASP.NET](#) Core provides a way to store sensitive data securely during development. This tool stores app secrets in a separate location from the project tree and ensures they are not checked into source control. However, it's essential to understand that the Secret Manager tool doesn't encrypt the stored secrets and should only be used for development purposes.

The Secret Manager tool operates on project-specific configuration settings stored in the user profile directory. It hides implementation details and stores values in a JSON configuration file. However, developers should not rely on the location or format of data saved by the Secret Manager tool, as these details may change in the future.

Enable Secret Storage

To enable secret storage using the Secret Manager tool, developers can use either the .NET CLI or Visual Studio. The `dotnet user-secrets init` command initializes the Secret Manager for the project and adds a `UserSecretsId` element to the project file. Visual Studio also provides a convenient way to manage user secrets through its UI.

Set, Access, and Manage Secrets

Once secret storage is enabled, developers can set, access, and manage secrets using commands like `dotnet user-secrets set`, `dotnet user-secrets list`, and `dotnet user-secrets remove`. Secrets can be accessed in code using the Configuration API, allowing for secure retrieval of sensitive data at runtime.

COPY 


```
// Non-compliant code with hardcoded sensitive data
public class UserService
```

```

{
    public void ConnectToDatabase()
    {
        string connectionString =
"Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
        // Connect to the database using the hardcoded connection
string
    }
}

```

In the non-compliant code above, sensitive data such as the database connection string is hardcoded directly into the source code, which poses a security risk.

COPY 

```

// Compliant code using environment variables for sensitive data
public class UserService
{
    public void ConnectToDatabase()
    {
        string connectionString =
Environment.GetEnvironmentVariable("DB_CONNECTION_STRING");
        if (connectionString != null)
        {
            // Connect to the database using the connection string
retrieved from environment variable
        }
        else
        {
            // Handle the case when environment variable is not set
        }
    }
}

```

In the compliant code above, sensitive data is stored in environment variables, and the application retrieves the connection string from the environment variable at runtime. This approach improves security by keeping sensitive information out of the source code.

XML Processing

When processing XML in .NET applications, it's important to consider security issues to prevent vulnerabilities such as XML External Entity (XXE) attacks and denial of service attacks. Let's discuss the various XML processing options available in .NET and their security implications:

.NET Framework Options:

1. LINQ to XML (C# / Visual Basic):

- **Processing Type:** In-memory
- **Description:** LINQ to XML provides intuitive document creation and transformation capabilities, similar to SQL for objects, relational data, and XML data.
- **Security Considerations:** LINQ to XML is relatively safe as it operates in-memory and doesn't rely on external resources. However, developers should be cautious when parsing untrusted XML data to avoid injection attacks.

```
// Non-compliant code: Hardcoded XML string
using System;
using System.Xml.Linq;

public class Program
{
    public static void Main(string[] args)
    {
        string xmlString = "<root><person><name>John</name>
```

COPY 

```
<age>30</age></person></root>";
```

```
    // Parsing XML directly from a hardcoded string  
    XElement xml = XElement.Parse(xmlString);  
  
    Console.WriteLine(xml);  
}  
}
```

COPY 

```
// Compliant code: XML from file or trusted source  
using System;  
using System.Xml.Linq;  
  
public class Program  
{  
    public static void Main(string[] args)  
    {  
        // Read XML from a file or a trusted source  
        XElement xml = XElement.Load("data.xml");  
  
        Console.WriteLine(xml);  
    }  
}
```

2. System.Xml.XmlReader:

- **Processing Type:** Stream-based
- **Description:** XmlReader provides a fast, non-cached, forward-only way to access XML data.
- **Security Considerations:** XmlReader is vulnerable to XXE attacks if not configured properly. Developers should disable external entities and DTD processing when parsing untrusted XML data to mitigate this risk.

```
// Non-compliant code: XmlReader without secure settings
using System;
using System.Xml;

public class Program
{
    public static void Main(string[] args)
    {
        // Create an XmlReader without secure settings
        XmlReader reader = XmlReader.Create("data.xml");

        while (reader.Read())
        {
            // Process XML data
            Console.WriteLine(reader.Name);
        }

        reader.Close();
    }
}
```

```
// Compliant code: XmlReader with secure settings
using System;
using System.Xml;

public class Program
{
    public static void Main(string[] args)
    {
        // Create an XmlReader with secure settings
        XmlReaderSettings settings = new XmlReaderSettings();
        settings.DtdProcessing = DtdProcessing.Parse; // Disable DTD
processing
        settings.XmlResolver = null; // Disable external entities
    }
}
```

```

        XmlReader reader = XmlReader.Create("data.xml", settings);

        while (reader.Read())
        {
            // Process XML data
            Console.WriteLine(reader.Name);
        }

        reader.Close();
    }
}

```

3. System.Xml.XmlWriter:

- **Processing Type:** Stream-based
- **Description:** XmlWriter provides a fast, non-cached, forward-only way to generate XML data.
- **Security Considerations:** XmlWriter itself doesn't pose significant security risks. However, care should be taken to ensure that the generated XML doesn't contain sensitive information or vulnerabilities.

```

// Non-compliant code: XmlWriter without secure settings
using System;
using System.Xml;

public class Program
{
    public static void Main(string[] args)
    {
        // Create an XmlWriter without secure settings
        XmlWriter writer = XmlWriter.Create("output.xml");

        writer.WriteStartElement("root");
    }
}

```

COPY 

```
        writer.WriteElementString("name", "John");
        writer.WriteElementString("age", "30");
        writer.WriteEndElement();

        writer.Close();
    }
}
```

COPY 

```
// Compliant code: XmlWriter with secure settings
using System;
using System.Xml;

public class Program
{
    public static void Main(string[] args)
    {
        // Create an XmlWriter with secure settings
        XmlWriterSettings settings = new XmlWriterSettings();
        settings.OmitXmlDeclaration = false; // Include XML
declaration
        settings.Indent = true; // Indent XML for readability

        XmlWriter writer = XmlWriter.Create("output.xml", settings);

        writer.WriteStartElement("root");
        writer.WriteElementString("name", "John");
        writer.WriteElementString("age", "30");
        writer.WriteEndElement();

        writer.Close();
    }
}
```


4. System.Xml.XmlDocument:

- **Processing Type:** In-memory
- **Description:** XmlDocument implements the W3C Document Object Model (DOM) and allows for creation, insertion, removal, and modification of nodes using familiar DOM methods and properties.
- **Security Considerations:** XmlDocument is susceptible to XXE attacks if untrusted XML is parsed without proper configuration. Developers should disable external entities and DTD processing to mitigate this risk.

COPY 

```
// Non-compliant code: XmlDocument without secure settings
using System;
using System.Xml;

public class Program
{
    public static void Main(string[] args)
    {
        // Create an XmlDocument without secure settings
        XmlDocument doc = new XmlDocument();
        doc.Load("data.xml");

        XmlNodeList nodes = doc.SelectNodes("//person");

        foreach (XmlNode node in nodes)
        {
            // Process XML data

            Console.WriteLine(node.SelectSingleNode("name").InnerText);
            Console.WriteLine(node.SelectSingleNode("age").InnerText);
        }
    }
}
```

```
// Compliant code: XmlDocument with secure settings
using System;
using System.Xml;

public class Program
{
    public static void Main(string[] args)
    {
        // Create an XmlDocument with secure settings
        XmlDocument doc = new XmlDocument();
        XmlReaderSettings settings = new XmlReaderSettings();
        settings.DtdProcessing = DtdProcessing.Parse; // Disable DTD
processing
        settings.XmlResolver = null; // Disable external entities

        using (XmlReader reader = XmlReader.Create("data.xml",
settings))
        {
            doc.Load(reader);
        }

        XmlNodeList nodes = doc.SelectNodes("//person");

        foreach (XmlNode node in nodes)
        {
            // Process XML data

            Console.WriteLine(node.SelectSingleNode("name").InnerText);
            Console.WriteLine(node.SelectSingleNode("age").InnerText);
        }
    }
}
```

5. System.Xml.XPath.XPathNavigator:

- **Processing Type:** In-memory
- **Description:** XPathNavigator offers editing options and navigation capabilities using a cursor model.
- **Security Considerations:** Similar to XmlDocument, XPathNavigator is vulnerable to XXE attacks if not configured securely. Developers should follow best practices to prevent XXE vulnerabilities.

COPY 

```
// Non-compliant code: XPathNavigator without secure settings
using System;
using System.Xml.XPath;

public class Program
{
    public static void Main(string[] args)
    {
        // Create an XPathNavigator without secure settings
        XPathDocument doc = new XPathDocument("data.xml");
        XPathNavigator nav = doc.CreateNavigator();

        XPathNodeIterator nodes = nav.Select("//person");

        while (nodes.MoveNext())
        {
            // Process XML data

            Console.WriteLine(nodes.Current.SelectSingleNode("name").Value);

            Console.WriteLine(nodes.Current.SelectSingleNode("age").Value);
        }
    }
}
```

```

// Compliant code: XPathNavigator with secure settings
using System;
using System.Xml;
using System.Xml.XPath;

public class Program
{
    public static void Main(string[] args)
    {
        // Create an XPathNavigator with secure settings
        XPathDocument doc = new XPathDocument("data.xml");
        XPathNavigator nav = doc.CreateNavigator();

        XPathNodeIterator nodes = nav.Select("//person");

        while (nodes.MoveNext())
        {
            // Process XML data

            Console.WriteLine(nodes.Current.SelectSingleNode("name").Value);

            Console.WriteLine(nodes.Current.SelectSingleNode("age").Value);
        }
    }
}

```

6. XslCompiledTransform:

- **Processing Type:** In-memory
- **Description:** XslCompiledTransform provides options for transforming XML data using XSL transformations.
- **Security Considerations:** XslCompiledTransform can be vulnerable to XXE attacks if untrusted XML is transformed without proper precautions.

Developers should ensure that input XML is sanitized to prevent injection attacks.

Win32 and COM-based Options:

1. XmlLite:

- **Description:** XmlLite is a fast, secure, non-caching, forward-only XML parser suitable for building high-performance XML apps. It works with any language that can use dynamic link libraries (DLLs).
- **Security Considerations:** XmlLite is relatively safe, but developers should handle XML data securely to prevent injection attacks.

2. MSXML:

- **Description:** MSXML is a COM-based technology included with the Windows operating system for processing XML. It provides a native implementation of the DOM with support for XPath and XSLT.
- **Security Considerations:** MSXML is prone to various security vulnerabilities if not used carefully. Developers should keep MSXML updated and apply security patches regularly to mitigate risks.

When working with XML processing options in .NET, developers should always validate and sanitize input XML data, disable external entities and DTD processing, and follow best practices to prevent security vulnerabilities. Additionally, staying updated on security advisories and patches for XML-related libraries is essential to ensure the security of .NET applications.

Timing attacks

In .NET, timing attacks are a concern in security where attackers exploit variations in the execution time of cryptographic algorithms to gain information about secret data, such as passwords or encryption keys. Here's how you can address timing attacks in .NET:

1. Implementing Fixed-Time String Comparison:

COPY 

```
public bool Equals(string str1, string str2)
{
    if (str1.Length != str2.Length)
    {
        return false;
    }

    int result = 0;

    for (var i = 0; i < str1.Length; i++)
    {
        result |= str1[i] ^ str2[i];
    }

    return result == 0;
}
```

- This code performs a fixed-time comparison of two strings to mitigate timing attacks. It compares each character of the strings using the XOR operator (^) and bitwise OR (|), ensuring that the execution time does not reveal information about the compared strings.

2. Disabling Optimization and Inlining:

To prevent the JIT compiler from optimizing the code and potentially introducing timing vulnerabilities, you can use the `MethodImpl` attribute with `NoInlining` and `NoOptimization` options:

COPY 

```
[MethodImpl(MethodImplOptions.NoInlining |
MethodImplOptions.NoOptimization)]
public bool Equals(string str1, string str2)
```

```
{  
    // Implementation remains the same  
}
```

- This ensures that the method is not inlined or optimized, maintaining a consistent execution time.
3. **Using `CryptographicOperations.FixedTimeEquals`** (since .NET Core 2.1):
.NET Core provides a built-in method for fixed-time comparison of byte arrays:

```
var result = CryptographicOperations.FixedTimeEquals(  
    Encoding.UTF8.GetBytes(str1), Encoding.UTF8.GetBytes(str2)  
);
```

COPY 

This method compares two byte arrays in constant time, mitigating timing attacks.

ViewState is love

ViewState attacks in .NET can be a serious concern for web application security. Attackers can exploit ViewState deserialization vulnerabilities to execute arbitrary code and gain unauthorized access to the application. Here's an overview of ViewState attacks and how they can be mitigated:

1. **Understanding ViewState Deserialization:** ViewState is a mechanism in [ASP.NET](#) that allows state information to be persisted across postbacks. It serializes the state of server-side controls and sends it to the client as a hidden field in the web form. The client sends the ViewState back to the server on subsequent requests, where it is deserialized and used to restore the control's state.
2. **Exploiting ViewState Deserialization:** Attackers can manipulate the ViewState parameter to execute arbitrary code on the server. They can modify the serialized ViewState data to include malicious code, such as remote code execution

payloads or WebShell logic. By submitting the manipulated ViewState, the attacker can trigger the deserialization process on the server and execute the injected code.

3. **Mitigating ViewState Attacks:** To mitigate ViewState attacks, consider the following measures:

- **Validate and Sanitize Input:** Implement strict input validation and data sanitization to prevent malicious input from being processed by the server.
- **Use Secure ViewState Encryption:** Encrypt ViewState data using strong encryption algorithms and secure keys to prevent tampering and manipulation by attackers.
- **Apply Strict Deserialization Controls:** Use .NET framework features like `ActivitySurrogateSelectorFromFile` to control the deserialization process and prevent the execution of unauthorized code.
- **Monitor and Log ViewState Activity:** Implement logging and monitoring mechanisms to track ViewState usage and detect any suspicious or unauthorized activities.
- **Update and Patch:** Regularly update the .NET framework and apply security patches to address known vulnerabilities and security issues related to ViewState deserialization.

4. **Example of ViewState Attack:** Below is an example of how an attacker can exploit ViewState deserialization to execute arbitrary code in a .NET application:

```
// Malicious ViewState payload
__VIEWSTATE=<yso_generated_content>&__VIEWSTATEGENERATOR=60AF4XXX&
```

COPY 

- The attacker crafts a malicious ViewState payload containing serialized data with injected code. When submitted to the server, the ViewState is deserialized, and

the injected code is executed, allowing the attacker to gain unauthorized access or execute arbitrary commands.

COPY 

```
// Non-compliant ViewState Handling

// Assume this is the vulnerable code that handles ViewState
deserialization
public class ViewStateHandler : Page
{
    // This method handles the deserialization of ViewState
    protected override void LoadViewState(object savedState)
    {
        string viewState = savedState as string;
        // Deserialize the viewState without proper validation
        // This is vulnerable to ViewState manipulation attacks
        ViewState = DeserializeViewState(viewState);
    }

    // Method to deserialize the ViewState
    private object DeserializeViewState(string viewState)
    {
        // Vulnerable deserialization logic
        // Attackers can manipulate the viewState parameter to execute
        arbitrary code
        return ViewStateSerializer.Deserialize(viewState);
    }
}
```

In this non-compliant code, ViewState deserialization is performed without proper validation and sanitization of the input data. This leaves the application vulnerable to ViewState manipulation attacks where attackers can inject malicious code into the serialized ViewState data.

```
// Compliant ViewState Handling
```

```
// Implement proper ViewState validation and secure deserialization
```

```
public class ViewStateHandler : Page
```

```
{  
    // Override the LoadViewState method to handle secure ViewState  
    deserialization  
    protected override void LoadViewState(object savedState)  
    {  
        // Perform input validation to ensure savedState is not null  
        and is of the expected type  
        if (savedState != null && savedState is string viewState)  
        {  
            // Validate the ViewState integrity before deserialization  
            if (IsValidViewState(viewState))  
            {  
                // Deserialize the ViewState securely  
                ViewState = DeserializeViewState(viewState);  
            }  
            else  
            {  
                // Log or handle invalid ViewState  
                HandleInvalidViewState();  
            }  
        }  
    }  
}
```

```
// Method to validate the integrity of ViewState
```

```
private bool IsValidViewState(string viewState)
```

```
{
```

```
    // Implement validation logic to check ViewState integrity,  
    such as encryption, integrity check, etc.
```

```
    // Ensure that the viewState parameter has not been tampered  
    with by attackers
```

```
    // Example: Check if ViewState has been tampered using  
    cryptographic integrity checks
```

```

        return ViewStateIntegrityValidator.Validate(viewState);
    }

    // Method to deserialize the ViewState securely
    private object DeserializeViewState(string viewState)
    {
        // Implement secure deserialization logic
        // Use a trusted and secure deserialization mechanism
        // Example: Deserialize the ViewState using a secure library
        like Newtonsoft.Json
        return SecureViewStateDeserializer.Deserialize(viewState);
    }

    // Method to handle invalid ViewState
    private void HandleInvalidViewState()
    {
        // Implement appropriate action for handling invalid ViewState
        // Example: Log the event, reject the request, or display an
        error message to the user
        Logger.Log("Invalid ViewState detected. Request rejected.");
    }
}

```

In the compliant code:

- ViewState deserialization is performed securely with proper input validation and integrity checks.
- The `LoadViewState` method validates the integrity of the ViewState before deserialization to prevent manipulation by attackers.
- If the ViewState is determined to be invalid, appropriate actions are taken, such as logging the event or rejecting the request.
- Secure deserialization mechanisms are used to deserialize the ViewState, mitigating the risk of arbitrary code execution.

Formatter Attacks

Serialization attacks are a class of security vulnerabilities that exploit the deserialization process in software applications. These attacks occur when untrusted data is deserialized in a way that allows an attacker to manipulate the serialized data to execute arbitrary code, compromise the integrity of the application, or perform other malicious actions. Here are some common types of attacks associated with serialization:

1. **Deserialization of Untrusted Data:** This is the most common type of serialization attack. Insecure deserialization occurs when an application deserializes data from an untrusted or malicious source without properly validating or sanitizing it. Attackers can craft malicious serialized objects that exploit vulnerabilities in the deserialization process, leading to remote code execution, denial of service, or other security compromises.
2. **Remote Code Execution (RCE):** In a remote code execution attack, an attacker exploits vulnerabilities in the deserialization process to execute arbitrary code on the target system. By crafting a malicious serialized object that contains executable code, an attacker can trigger the execution of this code when it is deserialized by the application. This can lead to the compromise of sensitive data, unauthorized access to system resources, or complete control over the affected system.

```
// Non-compliant Code:
```

```
// Assume this is the vulnerable code that handles deserialization  
with a BinaryFormatter
```

```
public class DeserializationHandler  
{
```

```
    // This method deserializes the data using BinaryFormatter without  
proper validation
```

```
    public object Deserialize(byte[] data)  
    {
```

COPY 

```

        using (MemoryStream memoryStream = new MemoryStream(data))
        {
            BinaryFormatter binaryFormatter = new BinaryFormatter();
            // Deserialize the data without specifying a
SerializationBinder
            return binaryFormatter.Deserialize(memoryStream);
        }
    }
}

```

In this non-compliant code, deserialization is performed using a BinaryFormatter without specifying a SerializationBinder. This leaves the application vulnerable to deserialization attacks where attackers can manipulate the serialized data to execute arbitrary code.

COPY 

```

// Compliant Code:

// Implement proper deserialization with a BinaryFormatter and a
custom secure SerializationBinder
public class DeserializationHandler
{
    // This method deserializes the data securely using a custom
secure SerializationBinder
    public object Deserialize(byte[] data)
    {
        using (MemoryStream memoryStream = new MemoryStream(data))
        {
            BinaryFormatter binaryFormatter = new BinaryFormatter();
            // Specify a custom secure SerializationBinder to validate
the types during deserialization
            binaryFormatter.Binder = new SecureSerializationBinder();
            // Deserialize the data with the custom
SerializationBinder
            return binaryFormatter.Deserialize(memoryStream);
        }
    }
}

```

```

    }
}

// Custom secure SerializationBinder implementation to validate types
during deserialization
public class SecureSerializationBinder : SerializationBinder
{
    // Override the BindToType method to control the type binding
    during deserialization
    public override Type BindToType(string assemblyName, string
typeName)
    {
        // Implement validation logic to ensure only trusted types are
allowed
        // Example: Validate the assemblyName and typeName against a
whitelist of allowed types
        if (IsTypeAllowed(assemblyName, typeName))
        {
            // Return the validated type for deserialization
            return Type.GetType(typeName);
        }
        else
        {
            // Throw an exception for disallowed types
            throw new SecurityException("Type validation failed. Type
is not allowed.");
        }
    }

    // Method to validate if a type is allowed for deserialization
    private bool IsTypeAllowed(string assemblyName, string typeName)
    {
        // Implement validation logic to check if the type is allowed
for deserialization
        // Example: Check if the assemblyName and typeName belong to a
whitelist of allowed types

```

```

        // Return true if the type is allowed, false otherwise
        // Example: return true if (assemblyName ==
        "MyTrustedAssembly" && typeName == "MyTrustedType");
        return true; // Placeholder implementation
    }
}

```

In the compliant code:

- Deserialization is performed securely using a BinaryFormatter with a custom secure SerializationBinder.
- The custom SerializationBinder (`SecureSerializationBinder`) is responsible for validating the types during deserialization to ensure only trusted types are allowed.
- During deserialization, the `BindToType` method of the `SecureSerializationBinder` is invoked to control the type binding process.
- The `IsTypeAllowed` method within the `SecureSerializationBinder` implements the validation logic to check if a type is allowed for deserialization based on a whitelist of allowed types.
- If a type is not allowed, the deserialization process throws a `SecurityException` to prevent the loading of disallowed types.

By following these compliant practices, the .NET application can effectively mitigate deserialization attacks and enhance its overall security posture.

TemplateParser

The exploitation of ASP.NET TemplateParser, as described in the provided excerpt, illustrates a sophisticated attack vector that targets the deserialization process in ASP.NET applications, particularly focusing on SharePoint environments. Let's break down the key points and issues discussed in the text:

1. Introduction to SharePoint Custom .aspx Pages:

- SharePoint allows users to upload and create custom .aspx pages within the system.
- There's a fundamental assumption that untrusted users can create these pages but are restricted from adding server-side code.

2. Security Measures in SharePoint:

- SharePoint implements the `SPPageParserFilter` class, extending ASP.NET's `PageParserFilter`, to validate server control types against a Safe Controls list during the tokenization step in `TemplateParser.ProcessBeginTag()` method.
- This validation aims to prevent untrusted users from adding unauthorized server-side code.

3. Challenges and Ideas with TemplateParser:

- The `TemplateParser` has certain policies:
 - Control over types is limited to custom server controls and the `@Register` directive.
 - Server control tag names have constraints.
 - Server control type resolution is done using `Assembly.Load()` and `GetType()`.
 - Property types are derived from specified server control types via reflection.
- Challenges include controlling property types and custom server control types.

4. Exploitation Techniques:

- The text explores the idea of using generic types to control property types and manipulating the `Namespace` attribute to trick .NET into ignoring parts of the server control type.

- A specific exploit scenario involves using a null character (\0) to bypass parsing restrictions, enabling the execution of arbitrary code.
- The exploit involves crafting custom server control types that utilize interesting methods or TypeConverters, ultimately leading to remote code execution.

COPY 

```
<%@ Register
    TagPrefix="x"
    Assembly="System.Data.Services, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089"

    Namespace="System.Data.Services.Internal.ExpandedWrapper`1[[System.Dat
    eTime,mscorlib]]&#0;"
%>
<x:y
    runat="server"
    ExpandedElement="foobar"
/>
```

Explanation:

- This non-compliant code snippet attempts to exploit the ASP.NET TemplateParser.
- It uses the `@ Register` directive to register a custom server control type (`ExpandedWrapper<DateTime>`) from the `System.Data.Services` assembly.
- By appending `�` to the Namespace attribute, it tricks .NET into ignoring parts of the server control type, allowing arbitrary code execution.
- The `ExpandedElement` property is set to "foobar", potentially triggering a TypeConverter or method associated with the DateTime type.

```
<%@ Register
    TagPrefix="x"
    Assembly="MyAssembly"
    Namespace="MyNamespace"
%>
<x:MyCustomControl
    runat="server"
    MyProperty="safeValue"
/>
```

Explanation:

- This compliant code snippet demonstrates the secure usage of the ASP.NET `TemplateParser`.
- It registers a custom server control (`MyCustomControl`) from a trusted assembly (`MyAssembly`) and namespace (`MyNamespace`).
- The properties of the control are set to safe values, avoiding any potential security vulnerabilities.
- This code adheres to best practices for input validation and secure coding, mitigating the risk of exploitation.

ObjRefs

When .NET Remoting services are exposed via HTTP, they can be provided using either a standalone listener or integrated into an ASP.NET web application via IIS. In the latter scenario, the call stack involves components like

`HttpRemotingHandlerFactory`, `HttpRemotingHandler`, and `HttpHandlerTransportSink`.

Leaking ObjRefs

A critical vulnerability arises from the limited exception handling in the call stack, particularly in `SoapServerFormatterSink` and `BinaryServerFormatterSink`. These

components handle exceptions by creating and serializing a `ReturnMessage` object, which includes the current `LogicalCallContext`.

To exploit this, two conditions must be met:

1. Exception handling in `SoapServerFormatterSink` or `BinaryServerFormatterSink` is reached.
2. A class instance deriving from `MarshalByRefObject` is stored in the `LogicalCallContext`.

Trust Issues

It's possible to overwrite trusted values from the `HttpRequest` with values from corresponding HTTP headers, such as `__RequestVerb` and `__requestUri`. By manipulating these headers, attackers can pass validation and trigger exception handling in `SoapServerFormatterSink` or `BinaryServerFormatterSink`.

Known Suspects

Certain libraries, like `System.Diagnostics` and `DataDog.Trace`, use the `LogicalCallContext` to store `MarshalByRefObject` instances, potentially leaking `ObjRefs`.

Exploitation

With a leaked `ObjRef`, attackers can exploit deserialization in .NET Remoting via HTTP. While default configurations restrict direct remote code execution, bypass techniques exist to achieve it.

COPY 

```
// Non-compliant code vulnerable to HTTP header manipulation
public class HttpHandlerTransportSink
{
    public void HandleRequest(HttpContext context)
    {
        HttpRequest request = context.Request;
```

```

        BaseTransportHeaders baseTransportHeaders = new
BaseTransportHeaders();
        baseTransportHeaders["__RequestVerb"] = request.HttpMethod;
        baseTransportHeaders.RequestUri =
(string)context.Items["__requestUri"];

        // Copy HTTP headers to base transport headers
        NameValueCollection headers = request.Headers;
        foreach (string headerName in headers.AllKeys)
        {
            string headerValue = headers[headerName];
            baseTransportHeaders[headerName] = headerValue;
        }

        // Process request using base transport headers
        // This allows HTTP header manipulation
        ProcessRequest(baseTransportHeaders);
    }
}

```

COPY 

```

// Compliant code with improved HTTP header parsing
public class HttpHandlerTransportSink
{
    public void HandleRequest(HttpContext context)
    {
        HttpRequest request = context.Request;
        BaseTransportHeaders baseTransportHeaders = new
BaseTransportHeaders();

        // Check if late HTTP header parsing is enabled
        if (!AppSettings.LateHttpRequestParsing)
        {
            // Set trusted values directly from HttpRequest
            baseTransportHeaders["__RequestVerb"] =
request.HttpMethod;

```

```

        baseTransportHeaders.RequestUri =
(string)context.Items["__requestUri"];
    }
    else
    {
        // Late HTTP header parsing enabled, prevent overwriting
trusted values
        baseTransportHeaders["__RequestVerb"] = "POST"; // Default
to POST
        baseTransportHeaders.RequestUri = "/";
    }

    // Copy HTTP headers to base transport headers
    NameValueCollection headers = request.Headers;
    foreach (string headerName in headers.AllKeys)
    {
        string headerValue = headers[headerName];
        baseTransportHeaders[headerName] = headerValue;
    }

    // Process request using base transport headers
    ProcessRequest(baseTransportHeaders);
}
}

```

In the compliant code, the application checks whether late HTTP header parsing is enabled before populating the base transport headers. If it's enabled, the code prevents overwriting of trusted values from the `HttpRequest` with untrusted ones from HTTP headers, thus mitigating the vulnerability.

Resources

- [code-white](#)
- [OWASP](#)

- Sonarsource
- DevSecOps Guides

Subscribe to our newsletter

Read articles from **DevSecOpsGuides** directly inside your inbox. Subscribe to the newsletter, and don't miss out.

SUBSCRIBE

dotnet

C#

Devops

DevSecOps

Written by



Reza Rashidi

Follow

Published on



DevSecOpsGuides

Follow

MORE ARTICLES

RR

Reza Rashidi

RR

Reza Rashidi



Attacking Rust

"Attacking Rust" delves into the intricacies of identifying and mitigating security vulnerabilities ...



Attacking Java

Attacking Java applications requires a nuanced understanding of both the language's intricacies and ...

RR Reza Rashidi



Attacking PHP

In modern PHP applications, attackers exploit various vulnerabilities to compromise systems, steal d...



Write on Hashnode

Powered by [Hashnode](#) - Home for tech writers and readers