Bypassing .NET Serialization Binders

Bypassing .NET Serialization Binders - Author not stated, codewhitesec.blogspot.com.

- Published: date not stated
- Original: <https://codewhitesec.blogspot.com/2022/06/bypassing-dotnet-serialization-binders.html>
- Preserved from: https://codewhitesec.blogspot.com/2022/06/bypassing-dotnet-serialization-binders.html (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Serialization binders are often used to validate types specified in the serialized data to prevent the deserialization of dangerous types that can have malicious side effects with the runtime serializers such as the `BinaryFormatter`.

In this blog post we'll have a look into cases where this can fail and consequently may allow to bypass validation. We'll also walk though two real-world examples of insecure serialization binders in the DevExpress framework (CVE-2022-28684) and Microsoft Exchange (CVE-2022-23277), that both allow remote code execution.

Introduction

Type Names

Type names are used to identify .NET types. In the fully qualified form (also known as *assembly qualified name*, AQN), it also contains the information on the assembly the type should be loaded from. This information comprises of the assembly's name as well as attributes specifying its version, culture, and a token of the public key it was signed with. Here is an (extensive) example of such an assembly qualified name:

```
System.Collections.Concurrent.ConcurrentBag`1+ListOperation[
    [System.Object, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]
],
System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

This assembly qualified name comprises of two parts with several components:

- Assembly Qualified Name (AQN)
- Type Full Name
- Namespace
- Type Name
- Generic Type Parameters Indicator
- Nested Type Name
- Generic Type Parameters
- Embedded Type AQN (EAQN)
- Assembly Full Name
- Assembly Name
- Assembly Attributes

You can see that the same breakdown can also be applied to the embedded type's AQN. For simplicity, the type info will be referred to as *type name* and the assembly info will be referred to as *assembly name* as these are the general terms used by .NET and thus also within this post.

The assembly and type information are used by the runtime to locate and bind the assembly. That software component is also sometimes referred to as the CLR Binder.

Serialization Binders

In its original intent, a `SerializationBinder` was supposed to work just like the runtime binder but only in the context of serialization/deserialization with the `BinaryFormatter`, `SoapFormatter`, and `NetDataContractSerializer`:

> Some users need to control which class to load, either because the class has moved between assemblies or a different version of the class is required on the server and client. — `SerializationBinder` Class

For that, a `SerializationBinder` provides two methods:

- `public virtual void BindToName(Type serializedType, out string assemblyName, out string typeName);`
- `public abstract Type BindToType(string assemblyName, string typeName);`

The `BindToName` gets called during serialization and allows to control the `assemblyName` and `typeName` values that get written to the serialized stream. On the other side, the `BindToType` gets called during deserialization and allows to control the `Type` being returned depending on the passed `assemblyName` and `typeName` that were read from the serialized stream. As the latter method is `abstract`, derived classes would need provide their own implementation of that method.

During the time .NET deserialization issues rose in 2017, the remark "`SerializationBinder` can also be used for security" was added to the `SerializationBinder` documentation. Later in 2020, that remark has been changed to the exact opposite:

[[image: image]]
(https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEj_vmpEFT_iKfBTOqe_v7RyQB3M6q

0lVEwcuGE9mTJKniiJOFXdb1SCtF9_2c9TgAgSEYmDsFRvDrleNXYA9Uesz1nFU_e34ZSOv-SEv6IT_jgBf85uadyZsz0jNEof6uEap8R0BzMs-VZypvStMiGJMx75PfASOu8AY1sf8cNiUiGsZrmgXE1FL_qb/s1600/diff-0b8845a039c9e5f799538d0a686cb70f597a79fb3b91069fcdc76a29b537a0a8.png)

That is probably why developers (mis-)use them as a security measure to prevent the deserialization of malicious types. And it is still widely used, even though those serializers have already been disapproved for obvious reasons.

But using a `SerializationBinder` for validating the type to be deserialized can be tricky and has pitfalls that may allow to bypass the validation depending on how it is implemented.

What could possibly go wrong?

For validating the specified type, developers can either

- work solely on the string representations of the specified assembly name and type name, or
- try to resolve the specified type and then work with the returned `Type`.

Each of these strategies has its own advantages and disadvantages.

Advantages/Disadvantages of Validation Before/After Type Binding

The advantage of the former is that type resolving is cost intensive and hence some advise against it to prevent a possible denial of service attacks.

On the other hand, however, the type name parsing is not that straight forward and the internal type parser/binder of .NET allows some unexpected quirks:

- whitespace characters (i. e., U+0009, U+000A, U+000D, U+0020) are generally ignored between tokens, in some cases even further characters
- type names can begin with a "`.`" (period), e. g., `.System.Data.DataSet`
- assembly names are case-insensitive and can be quoted, e. g., `MsCoRlIb` and `"mscorlib"`
- assembly attribute values can be quoted, even improperly, e. g., `PublicKeyToken="b77a5c561934e089"` and `PublicKeyToken='b77a5c561934e089`
- .NET Framework assemblies often only require the `PublicKey` / `PublicKeyToken` attribute, e. g., `System.Data.DataSet, System.Data, PublicKey=0000000000000000040000000000000` or `System.Data.DataSet, System.Data, PublicKeyToken=b77a5c561934e089`
- assembly attributes can be in arbitrary order, e. g., `System.Data, PublicKeyToken=b77a5c561934e089, Culture=neutral, Version=4.0.0.0`
- arbitrary additional assembly attributes are allowed, e. g., `System.Data, Foo=bar, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089, Baz=quux`
- assembly attributes can consist of almost arbitrary data (supported escape sequences: `\"`, `\'`, `\,`, `\/`, `\=`, `\\`, `\n`, `\r`, and `\t`)

This renders detecting known dangerous types based on their name basically impractical, which, by the way, is always a bad idea. Instead, only known safe types should be allowed and anything else should result in an exception being thrown.

In contrast to that, resolving the type before validation would allow to work with a normalized form of the type. But type resolution/binding may also fail. And depending on how the custom `SerializationBinder` handles such cases, it can allow attackers to bypass validation.

`SerializationBinder` Usages

If you keep in mind that the `SerializationBinder` was supposedly never meant to be used as a security measure (otherwise it would probably have been named `SerializationValidator` or similar), it gets more clear if you see how it is actually used by the `BinaryFormatter`, `SoapFormatter`, and `NetDataContractSerializer`:

- `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.ObjectReader.Bind(string, string)`
- `System.Runtime.Serialization.Formatters.Soap.SoapFormatter.ObjectReader.Bind(string, string)`
- `System.Runtime.Serialization.XmlObjectSerializerReadContextComplex.ResolveDataContractTypeInSharedTypeMode(string, string, out Assembly)`

Let's have a closer look at the first one, `ObjectReader.Bind(string, string)` used by `BinaryFormatter`:

```
1  // System.Runtime.Serialization.Formatters.Binary.ObjectReader
2  [SecurityCritical]
3  internal Type Bind(string assemblyString, string typeString)
4  {
5      Type type = null;
6      if (this.m_binder != null)
7      {
8          type = this.m_binder.BindToType(assemblyString, typeString);
9      }
10     if (type == null)
11     {
12         type = this.FastBindToType(assemblyString, typeString);
13     }
14     return type;
15 }
```

Here you can see that if the `SerializationBinder.BindToType(string, string)` call returns `null`, the fallback `ObjectReader.FastBindToType(string, string)` gets called.

```
// System.Runtime.Serialization.Formatters.Binary.ObjectReader
[SecurityCritical]
internal Type FastBindToType(string assemblyName, string typeName)
{
    Type type = null;
    ObjectReader.TypeNAssembly typeNAssembly = (ObjectReader.TypeNAssembly)
      this.typeCache.GetCachedValue(typeName);
    if (typeNAssembly == null || typeNAssembly.assemblyName != assemblyName)
    {
        Assembly assembly = null;
        if (this.bSimpleAssembly)
        {
            try
            {
                ObjectReader.sfileIOPermission.Assert();
                try
                {
                    assembly = ObjectReader.ResolveSimpleAssemblyName(new AssemblyName
                      (assemblyName));
                }
                finally
                {
                    CodeAccessPermission.RevertAssert();
                }
            }
            catch (Exception ex)
            {
            }
            if (assembly == null)
            {
                return null;
            }
            ObjectReader.GetSimplyNamedTypeFromAssembly(assembly, typeName, ref type);
        }
        else
        {
            try
            {
                ObjectReader.sfileIOPermission.Assert();
                try
                {
                    assembly = Assembly.Load(assemblyName);
                }
                finally
                {
                    CodeAccessPermission.RevertAssert();
                }
            }
            catch (Exception ex2)
            {
            }
            if (assembly == null)
            {
                return null;
            }
            type = FormatterServices.GetTypeFromAssembly(assembly, typeName);
        }
        if (type == null)
        {
            return null;
        }
        ObjectReader.CheckTypeForwardedTo(assembly, type.Assembly, type);
        typeNAssembly = new ObjectReader.TypeNAssembly();
        typeNAssembly.type = type;
        typeNAssembly.assemblyName = assemblyName;
        this.typeCache.SetCachedValue(typeNAssembly);
    }
    return typeNAssembly.type;
}
```

Here, if the `BinaryFormatter` uses `FormatterAssemblyStyle.Simple` (i. e., `bSimpleAssembly == true`, which is the default for `BinaryFormatter`), then the specified assembly name is used to create an `AssemblyName` instance and it is then attempted to load the corresponding assembly with it. This

must succeed, otherwise `ObjectReader.FastBindToType(string, string)` immediately returns with `null`. It is then tried to load the specified type with `ObjectReader.GetSimplyNamedTypeFromAssembly(Assembly, string, ref Type)`.

```
// System.Runtime.Serialization.Formatters.Binary.ObjectReader
[SecurityCritical]
private static void GetSimplyNamedTypeFromAssembly(Assembly assm, string typeName, ref Type type)
{
    try
    {
        type = FormatterServices.GetTypeFromAssembly(assm, typeName);
    }
    catch (TypeLoadException)
    {
    }
    catch (FileNotFoundException)
    {
    }
    catch (FileLoadException)
    {
    }
    catch (BadImageFormatException)
    {
    }
    if (type == null)
    {
        type = Type.GetType(typeName, new Func<AssemblyName, Assembly>
          (ObjectReader.ResolveSimpleAssemblyName), new Func<Assembly, string, bool, Type>(new
          ObjectReader.TopLevelAssemblyTypeResolver(assm).ResolveType), false);
    }
}
```

This method first calls `FormatterServices.GetTypeFromAssembly(Assembly, string)` that tries to load the type from the already resolved assembly using `Assembly.GetType(string)` (not depicted here). But if that fails, it uses `Type.GetType(string, Func<AssemblyName, Assembly>, Func<Assembly, string, bool, Type>, bool)` with the specified type name as first parameter. Now if the specified type name happens to be a AQN, the type loading succeeds and it returns the type specified by the AQN regardless of the already loaded assembly.

That means, unless the custom `SerializationBinder.BindToType(string, string)` implementation uses the same algorithm as the `ObjectReader.FastBindToType(string, string)` method, it might be possible to get the custom `SerializationBinder` to fail while the `ObjectReader.FastBindToType(string, string)` still succeeds. And if the custom `SerializationBinder.BindToType(string, string)` method does not throw an exception on failure but silently returns `null` instead, it would also allow to bypass any type validation implemented in `SerializationBinder.BindToType(string, string)`.

This behavior already mentioned in Jonathan Birch's *Dangerous Contents - Securing .Net Deserialization* in 2017:

> **Don't return null for unexpected types** – this makes some serializers fall back to a default binder, allowing exploits.

Origin of the Assembly Name and Type Name

The assembly name and type name values passed to the `SerializationBinder.BindToType(string, string)` during deserialization originate from the serialized stream: the assembly name is read by `BinaryAssembly.Read(__BinaryParser)` and the type name by `BinaryObjectWithMapTyped.Read(__BinaryParser)`.

On the serializing side, these values are written to the stream by `BinaryAssembly.Write(__BinaryWrite)` and `BinaryObjectWithMapTyped.Write(__BinaryWriter)`. The written values originate from an `SerObjectInfoCache` instance, which are set in the two available constructors:

- `SerObjectInfoCache(string typeName, string assemblyName, bool hasTypeForwardedFrom)`
- `SerObjectInfoCache(Type type)`

In the latter case, the assembly name and type name are obtained from the `TypeInformation` returned by `BinaryFormatter.GetTypeInformation(Type)`. In the former case, however, the assembly name and type name are adopted from the `SerializationInfo` instance filled during serialization if the assembly name or type name was set explicitly via `SerializationInfo.AssemblyName` and `SerializationInfo.FullTypeName`, respectively.

That means, besides using `SerializationInfo.SetType(Type)`, it is also possible to set the assembly name and type name explicitly and independently as strings by using `SerializationInfo.AssemblyName` and `SerializationInfo.FullTypeName`:

```
[Serializable]
class Marshal : ISerializable
{
   public void GetObjectData(SerializationInfo info, StreamingContext context)
   {
      info.AssemblyName = "...";
      info.FullTypeName = "...";
   }
}
```

There is also another and probably more convenient way to specify an arbitrary assembly name and type name by using a custom `SerializationBinder` during serialization:

```
class CustomSerializationBinder : SerializationBinder
{
   public override void BindToName(Type serializedType, out string assemblyName, out string typeName)
   {
      assemblyName = "...";
      typeName     = "...";
   }

   public override Type BindToType(string assemblyName, string typeName)
   {
      throw new NotImplementedException();
   }
}
```

This allows to fiddle with all assembly names and type names that are used within the object graph to be serialized.

Common Pitfalls of Custom `SerializationBinder`s

There are two common pitfalls that can render a `SerializationBinder` bypassable:

- parsing the passed assembly name and type name differently than the .NET runtime does
- resolving the specified type differently than the .NET runtime does

We will demonstrate these with two case studies: the DevExpress framework (CVE-2022-28684) and Microsoft Exchange (CVE-2022-23277).

Case Study № 1: `SafeSerializationBinder` in DevExpress (CVE-2022-28684)

Despite its name, the `DevExpress.Data.Internal.SafeSerializationBinder` class of *DevExpress.Data* is not really a `SerializationBinder`. But its `Ensure(string, string)` method is used by the `DXSerializationBinder.BindToType(string, string)` method to check for safe and unsafe types.

```
// DevExpress.Data.Internal.SafeSerializationBinder.DXSerializationBinder
public sealed override Type BindToType(string assemblyName, string typeName)
{
    if (this.isNetDataContractsSerializationBinder && assemblyName == "0")
    {
        assemblyName = "mscorlib";
    }
    SafeSerializationBinder.Ensure(assemblyName, typeName);
    if (SafeSerializationBinder.EnsureAssemblyQualifiedTypeName(ref assemblyName, ref typeName))
    {
        SafeSerializationBinder.Ensure(assemblyName, typeName);
    }
    if (typeName == "DevExpress.XtraPivotGrid.PivotGridFieldFilterValues")
    {
        assemblyName = "DevExpress.PivotGrid.v21.2.Core";
    }
    typeName = DXAssemblyResolverEx.GetValidTypeName(typeName);
    assemblyName = DXAssemblyResolverEx.GetValidAssemblyName(assemblyName);
    Assembly assembly = SafeSerializationBinder.DXSerializationBinder.GetAssembly(assemblyName);
    if (!(assembly != null))
    {
        return null;
    }
    return assembly.GetType(typeName, false);
}
```

It does this by checking the assembly name and type name against a list of known unsafe types (i. e., `UnsafeTypes` class) and known safe types (i. e., `KnownTypes` class). To pass the validation, the former *must not* match while the latter *must* match as both `XtraSerializationSecurityTrace.UnsafeType(string, string)` and `XtraSerializationSecurityTrace.NotTrustedType(string, string)` result in an exception being thrown.

```
// DevExpress.Data.Internal.SafeSerializationBinder
public static void Ensure(string assemblyName, string typeName)
{
    if (SafeSerializationBinder.UnsafeTypes.Match(assemblyName, typeName))
    {
        XtraSerializationSecurityTrace.UnsafeType(assemblyName, typeName);
        return;
    }
    if (SafeSerializationBinder.KnownTypes.Match(assemblyName, typeName))
    {
        XtraSerializationSecurityTrace.TrustedType(assemblyName, typeName);
        return;
    }
    XtraSerializationSecurityTrace.NonTrustedType(assemblyName, typeName);
}
```

The check in each `Match(string, string)` method comprises of a match against so called type ranges and several full type names.

[[image: image]]
(https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEjZxQTOvsvuZEZzDbFVKTuXFxUimmGmwjwgJYljSG6kSGbHfdd84acRLv9RyYsAW0onwFrU418wMGhClv9ysH4XP1UQjS483jImUoOrDCXjtnbzBmFmWxDVovslLBBQ-pqv7IWDQ_KCofwOcbhnT7Jo2OQ2NGdcpPElBd4VN700Sc8E5ZxbO36iNePz/s1600/DevExpress_UnsafeTypes.Match.png)

A type range is basically a pair of assembly name and namespace prefix that the passed assembly name and type name are tested against.

```
// DevExpress.Data.Internal.SafeSerializationBinder.TypeRanges
internal static bool Match(KeyValuePair<string, string> asmAndType, string assembly, string type)
{
    string key = asmAndType.Key;
    if (assembly.StartsWith(key, StringComparison.InvariantCultureIgnoreCase))
    {
        if (assembly.Length == key.Length)
        {
            if (type.StartsWith(asmAndType.Value, StringComparison.InvariantCultureIgnoreCase))
            {
                return true;
            }
        }
        else if (assembly[key.Length] == ',' && type.StartsWith(asmAndType.Value,
          StringComparison.InvariantCultureIgnoreCase))
        {
            return true;
        }
    }
    return false;
}
```

Here is the definition of `UnsafeTypes.typeRanges` that `UnsafeTypes.Match(string, string)` tests against:

[[image: image]]
(https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEi_azs4bR9LlZkFvCxww5qTwc_BTvqar49AHR63icFS9lFKG1EUuybAsi6ih9piaRrCilkt3FdKCSnf8aKZazomMfhB_1kIQiwGFIYy1vmYeRJ7TM7J4Fj7i16W7_aB8CSmV3j2HQtz17oj-mNr0bWaYh30EIXCkL4H63cLrsyKYvwYWuyFh-97c_ZD/s1600/DevExpress_UnsafeTypes.typeRanges.png)

And here `UnsafeTypes.types`:

```
// DevExpress.Data.Internal.SafeSerializationBinder.UnsafeTypes
private static readonly HashSet<string> types = new HashSet<string>
  (StringComparer.InvariantCultureIgnoreCase)
{
    "System.DelegateSerializationHolder",
    "System.DelegateSerializationHolder+DelegateEntry",
    "System.Reflection.MemberInfoSerializationHolder",
    "System.UnitySerializationHolder",
    "System.Management.Automation.PSObject",
    "System.Workflow.ComponentModel.Serialization.ActivitySurrogateSelector",
    "System.AppDomainSetup",
    "System.Exception",
    "System.BadImageFormatException",
    "System.MissingFieldException",
    "System.MissingMemberException",
    "System.MissingMethodException",
    "System.IO.FileLoadException",
    "System.IO.FileNotFoundException",
    "System.Security.SecurityException",
    "System.Security.Permissions.PublisherIdentityPermissionAttribute",
    "System.Security.Permissions.PermissionSetAttribute",
    "System.Collections.Generic.ComparisonComparer`1",
    "System.Reflection.RuntimeAssembly",
    "System.Reflection.DefaultMemberAttribute",
    "System.Reflection.Emit.ModuleBuilderData",
    "System.Net.FileWebRequest",
    "System.Net.HttpWebRequest",
    "System.Media.SoundPlayer",
    "System.Configuration.ConfigurationException",
    "System.Runtime.Versioning.FrameworkName",
    "System.Drawing.Design.ToolboxItem",
    "System.RuntimeType",
    "System.RuntimeTypeHandle",
    "System.Resources.ResourceManager",
    "System.Reflection.RuntimeConstructorInfo",
    "System.Reflection.RuntimeEventInfo",
    "System.Reflection.RuntimeFieldInfo",
    "System.Reflection.RuntimeMethodInfo",
    "System.Reflection.RuntimePropertyInfo",
    "System.Reflection.RuntimeParameterInfo",
    "System.Reflection.RuntimeModule",
    "System.Reflection.RtFieldInfo",
    "System.Reflection.MdFieldInfo",
    "System.Reflection.ParameterInfo",
    "System.Reflection.Pointer",
    "System.Reflection.TypeDelegator",
    "System.Reflection.CustomAttributeTypedArgument",
    "System.Runtime.CompilerServices.RequiredAttributeAttribute",
    "System.Runtime.CompilerServices.StateMachineAttribute",
    "System.Security.Permissions.ResourcePermissionBase",
    "System.ComponentModel.LicenseException",
    "System.CodeDom.Compiler.TempFileCollection",
    "System.Data.DataSet"
};
```

This set basically comprises the types used in public gadgets such as those of *YSoSerial.Net*.

Remember that `SafeSerializationBinder.Ensure(string, string)` does not resolve the specified type but only works on the assembly names and type names read from the serialized stream. The type binding/resolution attempt happens after the string-based validation in `DXSerializationBinder.BindToType(string, string)` where `Assembly.GetType(string, bool)` is used to load the specified type from the specified assembly but without throwing an exception on error (i. e., the passed `false`).

We'll demonstrate how a `System.Data.DataSet` can be used to bypass validation in `SafeSerializationBinder.Ensure(string, string)` despite it is contained in `UnsafeTypes.types`.

As `DXSerializationBinder.BindToType(string, string)` can return `null` in two cases (`assembly == null` or `Assembly.GetType(string, bool)` returns `null`), it is possible to craft the assembly name and type name pair that does fail loading while the fallback `ObjectReader.FastBindToType(string, string)` still returns the proper type.

In the first attempt, we'll update the `ISerializable.GetObjectData(SerializationInfo, StreamingContext)` implementation of the *DataSet* gadget of *YSoSerial.Net* so that the assembly name is `mscorlib` and the type name the AQN of `System.Data.DataSet` :

```
diff --git a/ysoserial/Generators/DataSetGenerator.cs b/ysoserial/Generators/DataSetGenerator.cs
index ae4beb8..1755e62 100644
--- a/ysoserial/Generators/DataSetGenerator.cs
+++ b/ysoserial/Generators/DataSetGenerator.cs
@@ -62,7 +62,8 @@ namespace ysoserial.Generators

         public void GetObjectData(SerializationInfo info, StreamingContext context)
         {
-            info.SetType(typeof(System.Data.DataSet));
+            info.AssemblyName = "mscorlib";
+            info.FullTypeName = typeof(System.Data.DataSet).AssemblyQualifiedName;
             info.AddValue("DataSet.RemotingFormat", System.Data.SerializationFormat.Binary);
             info.AddValue("DataSet.DataSetName", "");
             info.AddValue("DataSet.Namespace", "");
```

With a breakpoint at `DXSerializationBinder.BindToType(string, string)` , we'll see that the first call to `SafeSerializationBinder.Ensure(string, string)` gets passed. This is because we use the AQN of `System.Data.DataSet` as type name while `UnsafeTypes.types` only contains the full name `System.Data.DataSet` instead. And as the pair of assembly name `mscorlib` and type name prefix `System.` is contained in `KnownTypes.typeRanges` , it will pass validation.

But now the assembly name and type name are passed to `SafeSerializationBinder.EnsureAssemblyQualifiedTypeName(string, string)` :

```
// DevExpress.Data.Internal.SafeSerializationBinder
internal static bool EnsureAssemblyQualifiedTypeName(ref string assemblyName, ref string typeName)
{
    string a = assemblyName;
    int num = typeName.Length - 1;
    bool flag = false;
    bool flag2 = false;
    int num2;
    while ((num2 = typeName.LastIndexOf(',', num)) > 0)
    {
        flag2 = true;
        int num3 = num2 + 1;
        while (char.IsWhiteSpace(typeName[num3]))
        {
            num3++;
        }
        if (flag)
        {
            assemblyName = typeName.Substring(num3);
            typeName = typeName.Substring(0, num2);
            return a != assemblyName;
        }
        if (typeName.IndexOf("version=", num3, num - num3, StringComparison.OrdinalIgnoreCase) ==
          num3)
        {
            int num4 = typeName.LastIndexOf(']', typeName.Length - 1);
            if (num4 > num2)
            {
                return false;
            }
            flag = true;
        }
        num = num2 - 1;
    }
    if (!flag2)
    {
        return false;
    }
    int num5 = typeName.LastIndexOf(']', typeName.Length - 1);
    num2 = typeName.LastIndexOf(',', typeName.Length - 1);
    if (num5 > num2)
    {
        return false;
    }
    int num6 = num2 + 1;
    while (char.IsWhiteSpace(typeName[num6]))
    {
        num6++;
    }
    assemblyName = typeName.Substring(num6);
    typeName = typeName.Substring(0, num2);
    return a != assemblyName;
}
```

That method probably tries to extract the type name and assembly name from an AQN passed in the `typeName`. It does this by looking for the last position of `,` in `typeName` and whether the part behind that position starts with `version=`. If that's not the case, the loop looks for the second last, then the third last, and so on. If `version=` was found, the algorithm assumes that the next iteration would also contain the assembly name (remember, the version is the first assembly attribute in the normalized form), `flag` gets set to `true` and in the next loop the position of the preceeding `,` marks the delimiter between the type name and assembly name. At the end, the passed `assemblyName` value stored in `a` and the extracted `assemblyName` values get compared. If they differ, `true` gets returned an the extracted assembly name and type name are checked by another call to `SafeSerializationBinder.Ensure(string, string)`.

With our AQN passed as type name, `SafeSerializationBinder.EnsureAssemblyQualifiedTypeName(string, string)` extracts the proper values so that the call to `SafeSerializationBinder.Ensure(string, string)` throws

an exception. That didn't work.

So in what cases does `SafeSerializationBinder.EnsureAssemblyQualifiedTypeName(string, string)` return `false` so that the second call to `SafeSerializationBinder.Ensure(string, string)` does not happen?

There are five `return` statements: three always return `false` (lines 28, 36, and 42) and the other two only return `false` when the passed `assemblyName` value equals the extracted assembly name (lines 21 and 51).

Let's first look at those always returning `false`: in two cases (line 28 and 42), the condition depends on whether the `typeName` contains a `]` after the last `,`. We can achieve that by adding a custom assembly attribute to our AQN that contains a `]`, which is perfectly valid:

```
diff --git a/ysoserial/Generators/DataSetGenerator.cs b/ysoserial/Generators/DataSetGenerator.cs
index ae4beb8..1755e62 100644
--- a/ysoserial/Generators/DataSetGenerator.cs
+++ b/ysoserial/Generators/DataSetGenerator.cs
@@ -62,7 +62,8 @@ namespace ysoserial.Generators

         public void GetObjectData(SerializationInfo info, StreamingContext context)
         {
-            info.SetType(typeof(System.Data.DataSet));
+            info.AssemblyName = "mscorlib";
+            info.FullTypeName = typeof(System.Data.DataSet).AssemblyQualifiedName + ", x=]";
             info.AddValue("DataSet.RemotingFormat", System.Data.SerializationFormat.Binary);
             info.AddValue("DataSet.DataSetName", "");
             info.AddValue("DataSet.Namespace", "");
```

Now the `SafeSerializationBinder.EnsureAssemblyQualifiedTypeName(string, string)` returns `false` without updating the `typeName` or `assemblyName` values. Loading the *mscorlib* assembly will succeed but the specified `DataSet` type won't be found in it so that `DXSerializationBinder.BindToType(string, string)` also returns `null` and the `ObjectReader.FastBindToType(string, string)` attempts to load the type, which finally succeeds.

Case Study № 2: `ChainedSerializationBinder` in Exchange Server (CVE-2022-23277)

After my colleage @frycos published his story on *Searching for Deserialization Protection Bypasses in Microsoft Exchange (CVE-2022–21969)*, I was curious whether it was possible to still bypass the security measures implemented in the `Microsoft.Exchange.Diagnostics.ChainedSerializationBinder` class.

The `ChainedSerializationBinder` is used for a `BinaryFormatter` instance created by `Microsoft.Exchange.Diagnostics.ExchangeBinaryFormatterFactory.CreateBinaryFormatter(DeserializeLocation, bool, string[], string[])` to resolve the specified type and then test it against a set of allowed and disallowed types to abort deserialization in case of a violation.

[[image: image]]
(https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEj2kZRFk8loBotKMP73iGE-2xITOCGPXxTiQPwqAC0wM4MV5HKEDHGYYnhO9fuV7f4pFi-nrPCRB9jC5wYfybAusp_IchCfEU03gYsRJouYfGMaa5dRSlZ8vZd7uDC3z2UtEbByjeLzH8qemZlWano24prCxCZY61WhHxfY1aGdqm_WCuyZsoFjHYBM/s1600/Exchange_ExchangeBinaryFormatterFactory.CreateBinaryFormatter.png)

Within the `ChainedSerializationBinder.BindToType(string, string)` method, the passed assembly name and type name parameters are forwarded to `InternalBindToType(string, string)` (not depicted here) and then to `LoadType(string, string)`. Note that only if the type was loaded successfully, it gets validated using the `ValidateTypeToDeserialize(Type)` method.

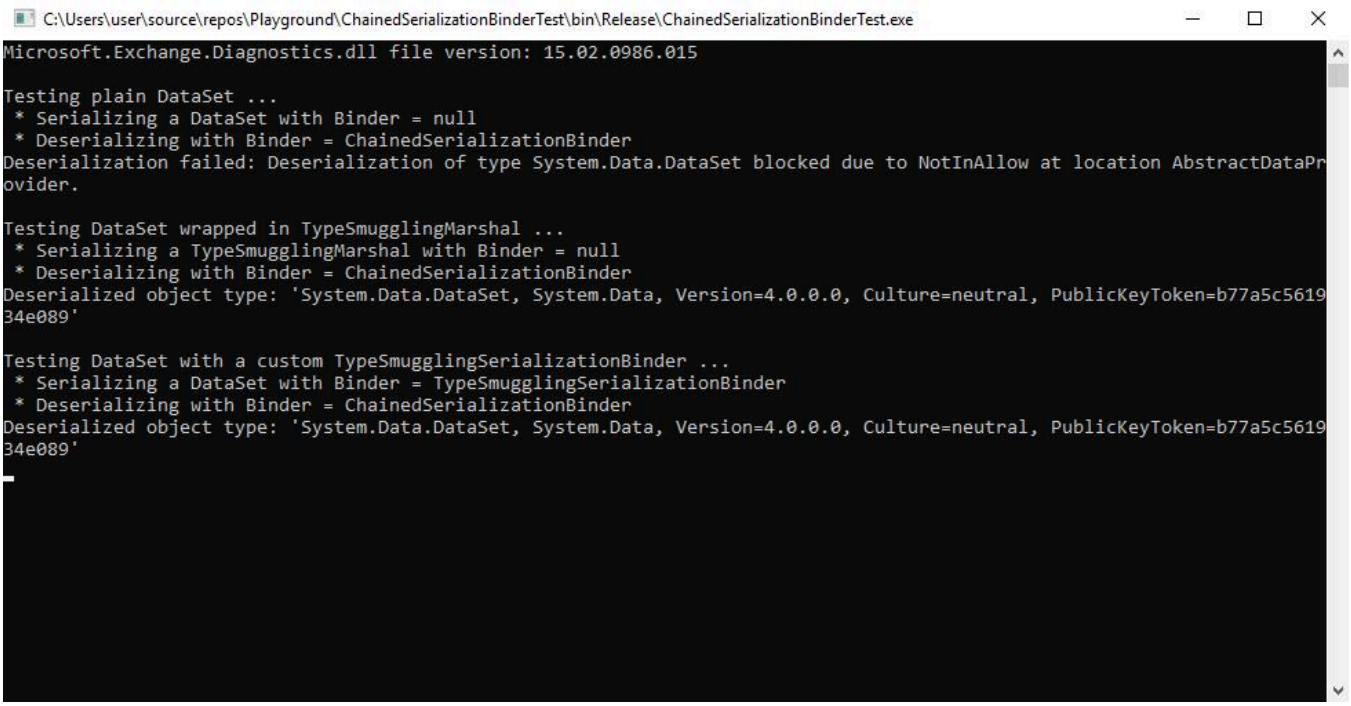
```
1  // Microsoft.Exchange.Diagnostics.ChainedSerializationBinder
2  public override Type BindToType(string assemblyName, string typeName)
3  {
4      if (this.serializationOnly)
5      {
6          throw new InvalidOperationException("ChainedSerializationBinder was created for
             serialization only.  This instance cannot be used for deserialization.");
7      }
8      Type type = this.InternalBindToType(assemblyName, typeName);
9      if (type != null)
10     {
11         this.ValidateTypeToDeserialize(type);
12     }
13     return type;
14 }
```

Inside `LoadType(string, string)`, it is attempted to load the type by combining both values in various ways, either via `Type.GetType(string)` or by iterating the already loaded assemblies and then using `Assembly.GetType(string)` on it. If loading of the type fails, `LoadType(string, string)` returns `null` and then `BindToType(string, string)` also returns `null` while the validation via `ValidateTypeToDeserialize(Type)` only happens if the type was successfully loaded.

```
// Microsoft.Exchange.Diagnostics.ChainedSerializationBinder
protected Type LoadType(string assemblyName, string typeName)
{
    Type type = null;
    try
    {
        type = Type.GetType(string.Format("{0}, {1}", typeName, assemblyName));
    }
    catch (TypeLoadException)
    {
    }
    catch (FileLoadException)
    {
    }
    if (type == null)
    {
        string shortName = assemblyName.Split(new char[]
        {
            ','
        })[0];
        try
        {
            type = Type.GetType(string.Format("{0}, {1}", typeName, shortName));
        }
        catch (TypeLoadException)
        {
        }
        catch (FileLoadException)
        {
        }
        if (type == null)
        {
            Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
            IEnumerable<Assembly> source = from x in assemblies
            where shortName == x.FullName.Split(new char[]
            {
                ','
            })[0]
            select x;
            Assembly assembly = source.Any<Assembly>() ? source.First<Assembly>() : null;
            if (assembly != null)
            {
                type = assembly.GetType(typeName);
            }
            else
            {
                foreach (Assembly assembly2 in assemblies)
                {
                    try
                    {
                        type = assembly2.GetType(typeName);
                        if (type != null)
                        {
                            break;
                        }
                    }
                    catch
                    {
                    }
                }
            }
        }
    }
    return type;
}
```

When the `ChainedSerializationBinder.BindToType(string, string)` method returns to the `ObjectReader.Bind(string, string)` method, the fallback method `ObjectReader.FastBindToType(string, string)` gets called for resolving the type. Now as `ChainedSerializationBinder.BindToType(string, string)` uses a different algorithm to resolve the type than `ObjectReader.FastBindToType(string, string)` does, it is possible to bypass the validation of `ChainedSerializationBinder` via the aforementioned tricks.

Here either of the two ways (a custom marshal class or a custom `SerializationBinder` during serialization) do work. The following demonstrates this with `System.Data.DataSet`:



Conclusion

The insecure serializers `BinaryFormatter`, `SoapFormatter`, and `NetDataContractSerializer` should no longer be used and legacy code should be migrated to the preferred alternatives.

If you happen to encounter a `SerializationBinder`, check how the type resolution and/or validation is implemented and whether `BindToType(string, string)` has a case that returns `null` so that the fallback `ObjectReader.FastBindToType(string, string)` may get a chance to resolve the type instead.
