# 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.

Code: csharp

// RememberMeUtil.cs:13  
public static HttpCookie createCookie(CustomMembershipUser user)  
{  
 <SNIP>  
}  
  
// RememberMeUtil.cs:27  
public static CustomMembershipUser validateCookieAndReturnUser(string cookie)  
{  
 <SNIP>  
}

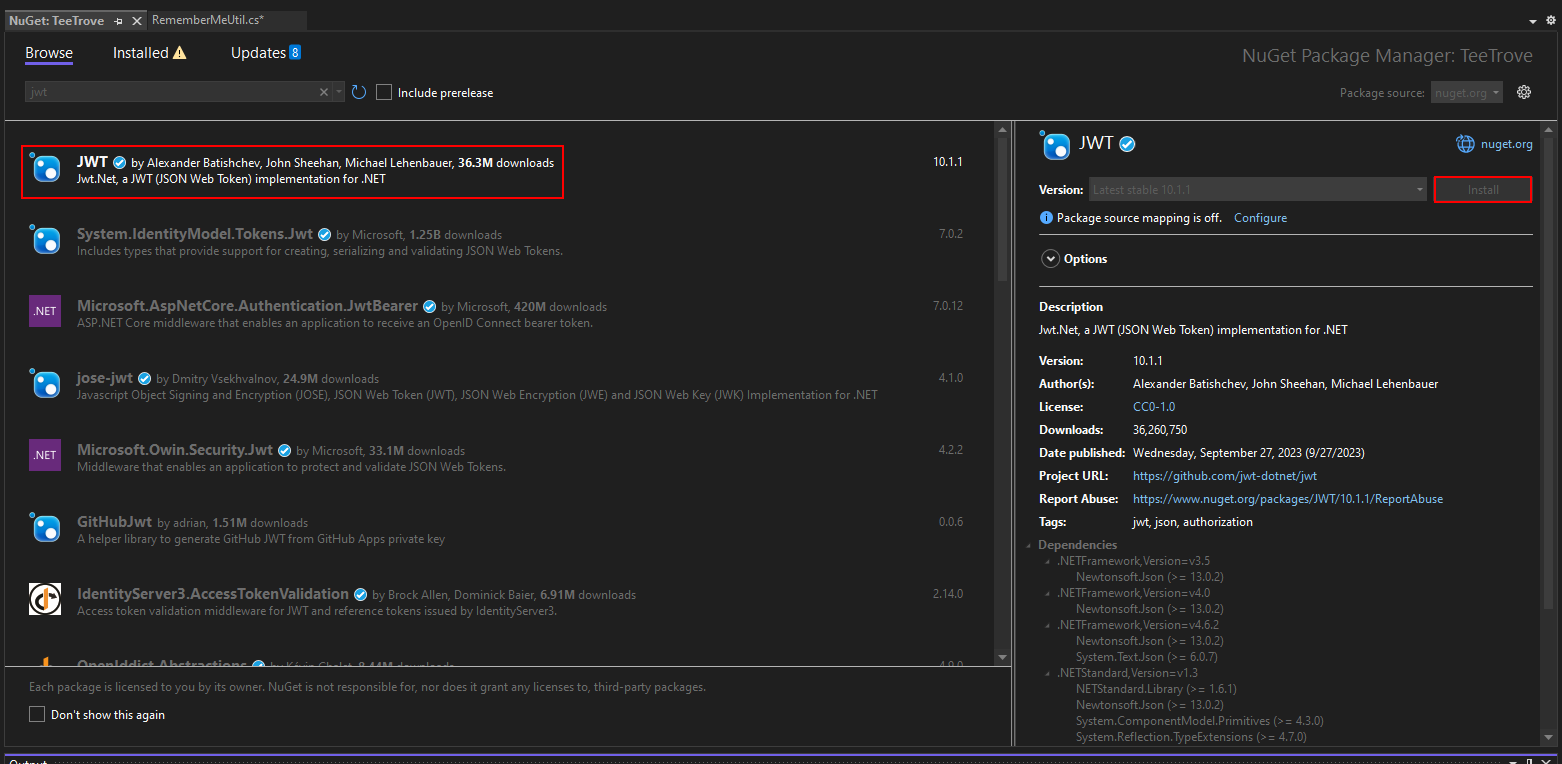
Right now, the serialized data looks like this:

Code: js

{"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](https://github.com/jwt-dotnet/jwt) package with the NuGet Package Manager.

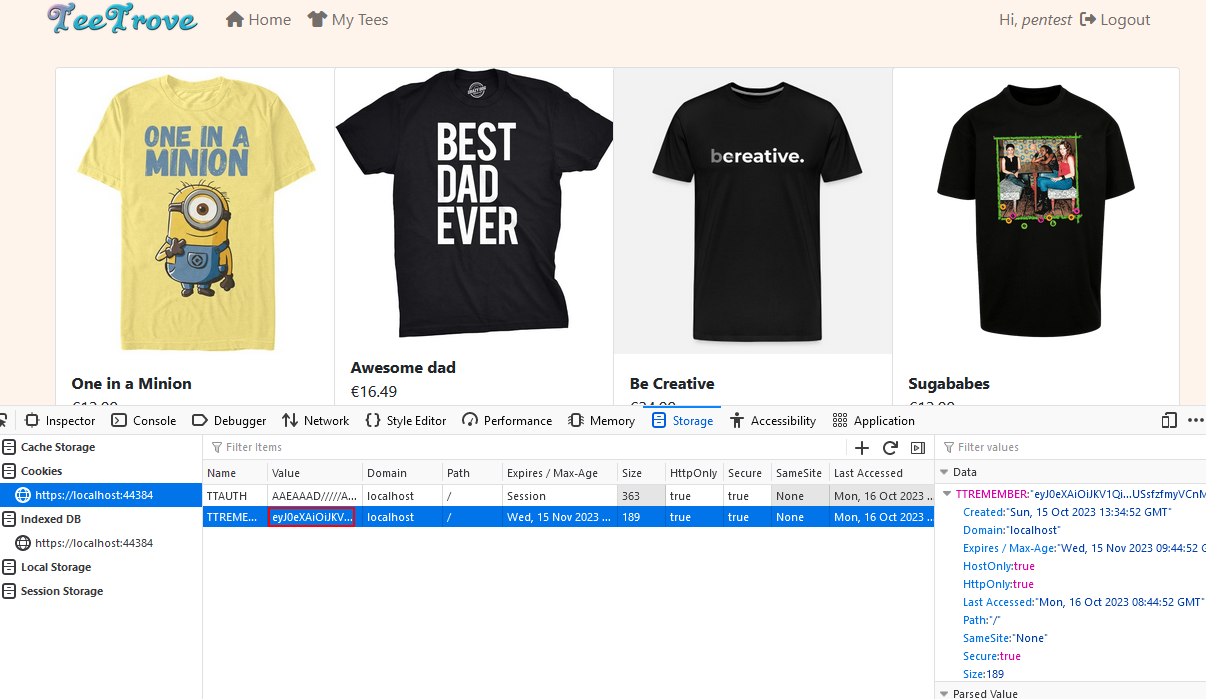


With the package installed, we can modify the createCookie method like so (original code is commented out). Here we are generating a JWT which contains the two claims (Username and RememberToken) and is signed with a secret key (JWT\_SECRET) to prevent tampering.

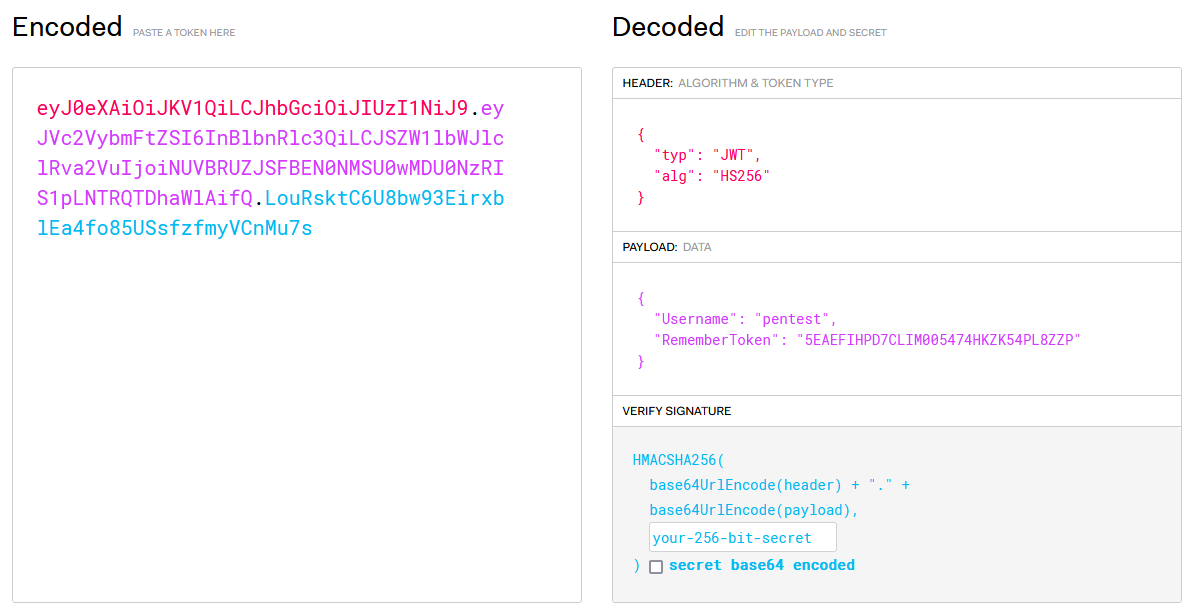
Code: csharp

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](https://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.

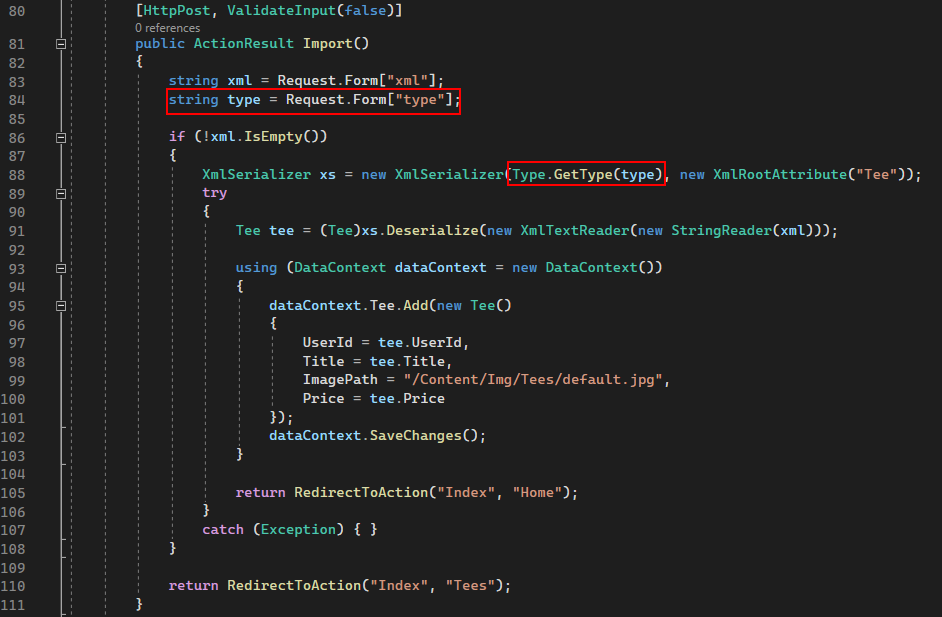
Code: csharp

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.

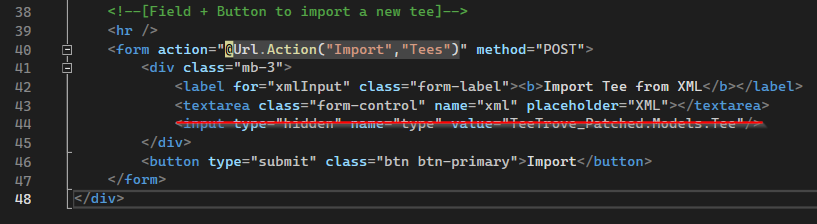


In Controllers.TeeController we can make the following change:

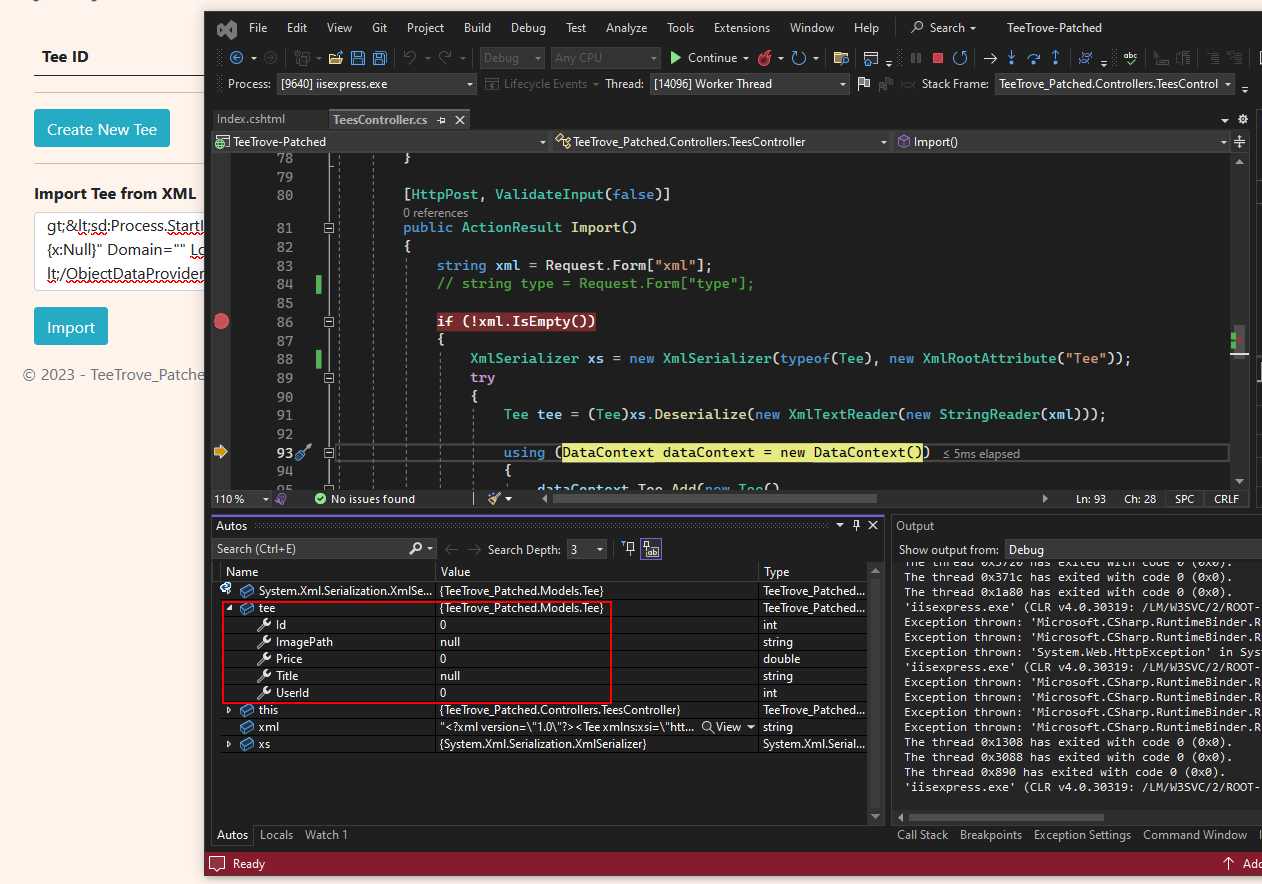
Code: csharp

<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:



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.

Code: csharp

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:

Code: csharp

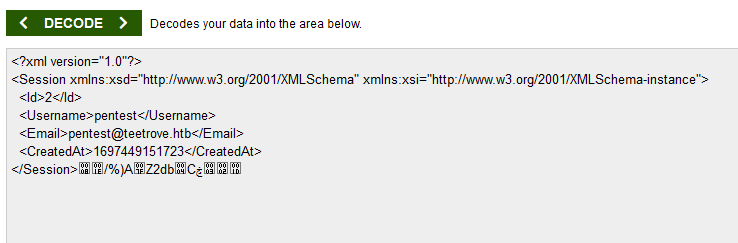
<SNIP>  
  
if (AuthCookieUtil.validateSignedCookie(authCookie.Value))  
{  
 //BinaryFormatter bf = new BinaryFormatter();  
 XmlSerializer xs = new XmlSerializer(typeof(Session));  
 Session session = null;  
 try  
 {  
 MemoryStream ms = new MemoryStream(Convert.FromBase64String(authCookie.Value.Split('.')[0]));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

Code: csharp

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).