Cookieless DuoDrop: IIS Auth Bypass & App Pool Privesc in ASP.NET Framework (CVE-2023-36899 & CVE-2023-36560)

Cookieless DuoDrop: IIS Auth Bypass & App Pool Privesc in ASP.NET Framework (CVE-2023-36899 & CVE-2023-36560) - Soroush Dalili, soroush.me.

- Published: date not stated
- Original: <https://soroush.me/blog/cookieless-duodrop-iis-auth-bypass-app-pool-privesc-in-asp-net-framework-cve-2023-36899>
- Preserved from: https://soroush.me/blog/cookieless-duodrop-iis-auth-bypass-app-pool-privesc-in-asp-net-framework-cve-2023-36899 (stored) on 2026-08-11
- 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.

Cookieless DuoDrop: IIS Auth Bypass & App Pool Privesc in ASP.NET Framework (CVE-2023-36899 & CVE-2023-36560)

[image: Cookieless DuoDrop: IIS Auth Bypass & App Pool Privesc in ASP.NET Framework (CVE-2023-36899 & CVE-2023-36560)]

Introduction

In modern web development, while cookies are the go-to method for transmitting session IDs, the .NET Framework also provides an alternative: encoding the session ID directly in the URL. This method is useful to clients that do not support cookies. An example of this URL encoding can be seen here:

Copy

```
1https://[targetdomain]/(S(aaaaaaaaaaaaaaaaaaaaaaaa))/default.aspx
```

This technique is known as the “cookieless” feature in the .NET Framework:

https://learn.microsoft.com/en-us/previous-versions/dotnet/articles/aa479314(v=msdn.10))

Many developers and security testers overlook this option, primarily because of its rarity in real-world applications. Historically, this has turned it into a goldmine for discovering client-side vulnerabilities, such as session fixation, session hijacking, HTML injection, and cross-site scripting. Additionally, this feature can be leveraged to circumvent path-based firewall rules that aren’t configured to recognize the cookieless approach.

For a deeper dive into security issues stemming from the use of cookieless sessions, consider these references:

- https://blog.isec.pl/all-is-xss-that-comes-to-the-net/ (A highly recommended read)
- https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/march/security-briefs-protect-your-site-with-url-rewriting
- https://www.sans.org/blog/session-attacks-and-asp-net-part-2/

Importantly, due to inherent security concerns, the cookieless feature was omitted from .NET Core and subsequent .NET versions. You can learn more about this decision in the following discussions:

- https://github.com/dotnet/aspnetcore/issues/37978
- https://github.com/dotnet/AspNetCore.Docs/blob/main/aspnetcore/fundamentals/app-state.md

However, let’s not forget the vast number of web applications still humming along on the classic .NET Framework (with the capital ‘F’)!

Finding the vulnerability

I was initially trying to find a new method to improve my IIS Short File Name Disclosure technique. As part of this, I realised that the cookieless part can be used twice within the path, and I quickly wrote a Twitter (X) post about how WAFs can be potentially bypassed using this:

https://twitter.com/irsdl/status/1640390106312835072

[image: Cookieless DuoDrop: IIS Auth Bypass & App Pool Privesc in ASP.NET Framework (CVE-2023-36899 & CVE-2023-36560)]

However, later on I identified a strange anomaly when the cookieless pattern was repeated twice. This resulted in two vulnerabilities reported to Microsoft as their impact and the exploitation were different:

- IIS restricted path bypass leading to potential authentication and path-filtration bypass
- Application Pool confusion leading to potential privilege escalations

Microsoft addressed both of these issues as part of one patch under CVE-2023-36899.

I got the following comment from Microsoft when I was trying to see why one of them was assessed as a duplicate reducing the bounty:

IIS Restricted Path Bypass

The cookieless feature of .NET Framework could be abused to access protected directories or those blocked by URL filters in IIS. For instance, on the victim.com website, consider:

- The page: `/webform/protected/target1.aspx` within the `/protected/` directory that enforces Basic authentication.
- The page: `/webform/bin/target2.aspx` that was temporarily moved to the `/bin/` folder, making it inaccessible.

Normally, accessing the pages through these URLs would be blocked in IIS:

Copy

```
1https://victim.com/webform/protected/target1.aspx
2
3https://victim.com/webform/bin/target2.aspx
```

However, the cookieless feature can be exploited to access these pages with the following patterns:

Copy

```
1https://victim.com/webform/(S(X))/prot/(S(X))ected/target1.aspx
2
3https://victim.com/webform/(S(X))/b/(S(X))in/target2.aspx
```

Here is how IIS was configured as an example to set authentication for the `/protected/` path:

[image: Cookieless DuoDrop: IIS Auth Bypass & App Pool Privesc in ASP.NET Framework (CVE-2023-36899 & CVE-2023-36560)]

When trying the standard approach, IIS authentication for the `/protected/` path behaves as expected, redirecting unauthorized users to the login page:

[image: Cookieless DuoDrop: IIS Auth Bypass & App Pool Privesc in ASP.NET Framework (CVE-2023-36899 & CVE-2023-36560)]

Still, the bypass technique allows access without authentication, using the Anonymous user. This can sometimes lead to errors if the system expects a specific profile:

[image: Cookieless DuoDrop: IIS Auth Bypass & App Pool Privesc in ASP.NET Framework (CVE-2023-36899 & CVE-2023-36560)]

The `target1.aspx` code was:

Copy

```
1Here I am, I am a protected page, how can you be here?!

2You username is: <%=HttpContext.Current.User.Identity.Name%>
```

The vulnerability stems from the way cookieless paths are rewritten in the .NET Framework. The following code was responsible for the final rewrite:

https://referencesource.microsoft.com/#System.Web/HttpResponse.cs,50b59e7205970b81

Copy

```
1internal String RemoveAppPathModifier(string virtualPath) {

2   if (String.IsNullOrEmpty(_appPathModifier))

3       return virtualPath;

4

5   int pos = virtualPath.IndexOf(_appPathModifier, StringComparison.Ordinal);

6

7   if (pos <= 0 || virtualPath[pos-1] != '/')

8       return virtualPath;

9

10    return virtualPath.Substring(0, pos-1) + virtualPath.Substring(pos + _appPathModifier.Length);

11}
```

The `RemoveAppPathModifier` method used by the `RemoveCookielessValuesFromPath` method of the `CookielessHelperClass` class as can be seen here:

https://referencesource.microsoft.com/#System.Web/Security/CookielessHelper.cs,113

By the time the function is invoked, the initial cookieless value is already removed. Due to this behavior, the path doesn't adhere to restriction rules, bypassing authentication or filter checks. Therefore, it changes the `/prot/(S(X))ected/` path to `/protected/` facilitating the observed bypasses. A

screenshot, provided below, captures the `RemoveAppPathModifier` method in action during the debugging of the .NET Framework:

[image: Cookieless DuoDrop: IIS Auth Bypass & App Pool Privesc in ASP.NET Framework (CVE-2023-36899 & CVE-2023-36560)]

Application Pool Confusion

Another notable issue involves how IIS manages Application Pools, potentially leading to privilege escalations or security bypasses. It's possible to manipulate the cookieless feature in .NET Framework to compel an IIS application to run using its parent's Application Pool instead of its own.

To illustrate:

- The root (`/`) of the website is running with the `DefaultAppPool` Application Pool
- The `/classic/` application uses the `.NET v4.5 Classic` Application Pool
- The `/classic/nodotnet/` application uses the `NoManagedCodeClassic` Application Pool, which doesn't support Managed Code.

A C# file named `AppPoolPrint.aspx`, accessible across all the above applications, reveals the current Application Pool name:

Copy

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

2<%

3string appPoolName = System.Environment.GetEnvironmentVariable("APP_POOL_ID");

4Response.Write("App Pool Name: " + appPoolName);

5%>
```

Based on the regular structure, accessing this page would result in:

Copy

```
/AppPoolPrint.aspx -> DefaultAppPool

2

/classic/AppPoolPrint.aspx -> .NET v4.5 Classic

4

/classic/nodotnet/AppPoolPrint.aspx -> Error: 404 Not Found (as Managed Code isn't supported)
```

However, by using the cookieless pattern twice, we can run this page using its parent Application Pool:

Copy

```
/(S(X))/(S(X))/classic/AppPoolPrint.aspx -> DefaultAppPool

2

/(S(X))/(S(X))/classic/nodotnet/AppPoolPrint.aspx -> DefaultAppPool

4

/classic/(S(X))/(S(X))/nodotnet/AppPoolPrint.aspx -> .NET v4.5 Classic
```

This allows even the pages within `/classic/nodotnet/` (which shouldn't execute Managed Code) to run the ASPX page using their parent's Application Pools. This behavior can lead to privilege escalation on IIS.

A new variant after the CVE-2023-36899 patch has been reported to Microsoft. This variant operates only on specific files, and I cannot discuss it in further detail at the moment.

Furthermore, the patch only disabled the aggressive path replacement by default configuration. Thus, it's possible to reintroduce the problematic behavior using the following settings in the `web.config`:

Copy

```
<appSettings>
  <add key="aspnet:RestoreAggressiveCookielessPathRemoval" value="true" />
</appSettings>
```

Update 15/11/2023

Microsoft has now addressed the new variant under CVE-2023-36560 released on 14 Nov. 2023. Here is the details:

The `PathInfo` feature of ASP.NET Framework web forms such as `.aspx` and `.ashx` pages (potentially other services which support path parameters) can be abused with the help of the Cookieless feature. Similar to the original report, this issue can lead to bypassing IIS authentication mechanism as well as running the page using the parent application pool which can lead to privilege escalation.

Consider that the following pages exist on the victim.com website:

Copy

```
/WebForm/protected/target1.aspx -> the /protected/ folder requires Basic Authentication

/WebForm/bin/target2.aspx -> an ASPX file has been moved to the /bin/ folder so no one can access it!
```

Accessing the above pages using the following URLs is not normally possible in IIS:

Copy

```
/WebForm/protected/target1.aspx

/WebForm/bin/target2.aspx
```

However, it is possible to still access these pages using the following Cookieless patterns and with the `PathInfo` (adding the Cookieless pattern after the .ASPX extension):

Copy

```
1/WebForm/pro/(S(X))tected/target1.aspx/(S(X))/

2

3/WebForm/b/(S(X))in/target2.aspx/(S(X))/
```

Application pool confusion can also be performed similar to the original report. As a reminder, the following shows the original application pool results using the previously provided C# test page:

Copy

```
1/AppPoolPrint.aspx -> DefaultAppPool

2

3/classic/AppPoolPrint.aspx -> .NET v4.5 Classic

4

5/classic/nodotnet/AppPoolPrint.aspx -> Error: 404 Not Found (as Managed Code isn't supported)
```

Now with the new bypass variant:

Copy

```
1/cla/(S(X))ssic/AppPoolPrint.aspx/(S(X))/ -> DefaultAppPool

2

3/cla/(S(X))ssic/nodotnet/AppPoolPrint.aspx/(S(X))/ -> DefaultAppPool

4

5/classic/nodot/(S(X))net/AppPoolPrint.aspx/(S(X))/ -> .NET v4.5 Classic
```

As a result, even pages within the `/classic/nodotnet/` application -which should not be able to run Managed Code- could run the ASPX page using its parents' Application Pools. This could lead to escalation or privileges on IIS.

MSRC experience notes for this variant:

- The issue was submitted on August 10, 2023 but was rejected on August 21, 2023 due to "unsatisfactory quality".

- I reported it again on August 22, 2023 by pasting the original report rather than referencing it.
- I was told a few weeks later that my issue is a duplicate and someone (must be Markus Wulftange according to the released advisory page) has reported it before me on August 9, 2023!
