Replace the linked URL with the target page. Then reload and visit your HTML page. Click the link and see if you get redirected to your site automatically or after the required user interactions.

Bypassing Open-Redirect Protection

As a bug bounty hunter, I find open redirects in almost all the web targets I attack. Why are open redirects still so prevalent in web applications today? Sites prevent open redirects by validating the URL used to redirect the user, making the root cause of open redirects failed URL validation. And, unfortunately, URL validation is extremely difficult to get right.

Here, you can see the components of a URL. The way the browser redirects the user depends on how the browser differentiates between these components:

```
scheme://userinfo@hostname:port/path?query#fragment
```

The URL validator needs to predict how the browser will redirect the user and reject URLs that will result in a redirect offsite. Browsers redirect users to the location indicated by the hostname section of the URL. However, URLs don't always follow the strict format shown in this example. They can be malformed, have their components out of order, contain characters that the browser does not know how to decode, or have extra or missing components. For example, how would the browser redirect this URL?

```
https://user:password:8080/example.com@attacker.com
```

When you visit this link in different browsers, you will see that different browsers handle this URL differently. Sometimes validators don't account for all the edge cases that can cause the browser to behave unexpectedly. In this case, you could try to bypass the protection by using a few strategies, which I'll go over in this section.

Using Browser Autocorrect

First, you can use browser autocorrect features to construct alternative URLs that redirect offsite. Modern browsers often autocorrect URLs that don't have the correct components, in order to correct mangled URLs caused by user typos. For example, Chrome will interpret all of these URLs as pointing to *https://attacker.com*:

```
https:attacker.com
https;attacker.com
https:\/\/attacker.com
https:/\/\attacker.com
```

These quirks can help you bypass URL validation based on a blocklist. For example, if the validator rejects any redirect URL that contains the strings `https://` or `http://`, you can use an alternative string, like `https;`, to achieve the same results.

Most modern browsers also automatically correct backslashes (\) to forward slashes (/), meaning they'll treat these URLs as the same:

```
https:\\example.com
https://example.com
```

If the validator doesn't recognize this behavior, the inconsistency could lead to bugs. For example, the following URL is potentially problematic:

```
https://attacker.com\@example.com
```

Unless the validator treats the backslash as a path separator, it will interpret the hostname to be *example.com*, and treat *attacker.com* as the username portion of the URL. But if the browser autocorrects the backslash to a forward slash, it will redirect the user to *attacker.com*, and treat *@example.com* as the path portion of the URL, forming the following valid URL:

```
https://attacker.com/@example.com
```

Exploiting Flawed Validator Logic

Another way you can bypass the open-redirect validator is by exploiting loopholes in the validator's logic. For example, as a common defense against open redirects, the URL validator often checks if the redirect URL starts with, contains, or ends with the site's domain name. You can bypass this type of protection by creating a subdomain or directory with the target's domain name:

```
https://example.com/login?redir=http://example.com.attacker.com
https://example.com/login?redir=http://attacker.com/example.com
```

To prevent attacks like these from succeeding, the validator might accept only URLs that both start and end with a domain listed on the allowlist. However, it's possible to construct a URL that satisfies both of these rules. Take a look at this one:

```
https://example.com/login?redir=https://example.com.attacker.com/example.com
```

This URL redirects to *attacker.com*, despite beginning and ending with the target domain. The browser will interpret the first *example.com* as the subdomain name and the second one as the filepath.

Or you could use the at symbol (@) to make the first *example.com* the username portion of the URL:

```
https://example.com/login?redir=https://example.com@attacker.com/example.com
```

Custom-built URL validators are prone to attacks like these, because developers often don't consider all edge cases.

Using Data URLs

You can also manipulate the scheme portion of the URL to fool the validator. As mentioned in Chapter 6, data URLs use the `data:` scheme to embed small files in a URL. They are constructed in this format:

```
data:MEDIA_TYPE[;base64],DATA
```

For example, you can send a plaintext message with the data scheme like this:

```
data:text/plain,hello!
```

The optional base64 specification allows you to send base64-encoded messages. For example, this is the base64-encoded version of the preceding message:

```
data:text/plain;base64,aGVsbG8h
```

You can use the `data:` scheme to construct a base64-encoded redirect URL that evades the validator. For example, this URL will redirect to *example.com*:

```
data:text/html;base64,
PHNjcmlwdD5sb2NhdGlvbj0iaHR0cHM6Ly9leGFtcGxlLmNvbSI8L3NjcmlwdD4=
```

The data encoded in this URL, *PHNjcmlwdD5sb2NhdGlvbj0iaHR0cHM6Ly9leGFtcGxlLmNvbSI8L3NjcmlwdD4=*, is the base64-encoded version of this script:

```
<script>location="https://example.com"</script>
```

This is a piece of JavaScript code wrapped between HTML `<script>` tags. It sets the location of the browser to *https://example.com*, forcing the browser to redirect there. You can insert this data URL into the redirection parameter to bypass blocklists:

```
https://example.com/login?redir=data:text/html;base64,
PHNjcmlwdD5sb2NhdGlvbj0iaHR0cHM6Ly9leGFtcGxlLmNvbSI8L3NjcmlwdD4=
```

Exploiting URL Decoding

URLs sent over the internet can contain only *ASCII characters*, which include a set of characters commonly used in the English language and a few special characters. But since URLs often need to contain special characters or characters from other languages, people encode characters by using URL encoding. URL encoding converts a character into a percentage sign, followed by two hex digits; for example, `%2f`. This is the URL-encoded version of the slash character (`/`).

When validators validate URLs, or when browsers redirect users, they have to first find out what is contained in the URL by decoding any characters that are URL encoded. If there is any inconsistency between how the validator and browsers decode URLs, you could exploit that to your advantage.

Double Encoding

First, try to double- or triple-URL-encode certain special characters in your payload. For example, you could URL-encode the slash character in *https://example.com/@attacker.com*. Here is the URL with a URL-encoded slash:

```
https://example.com%2f@attacker.com
```

And here is the URL with a double-URL-encoded slash:

```
https://example.com%252f@attacker.com
```

Finally, here is the URL with a triple-URL-encoded slash:

```
https://example.com%25252f@attacker.com
```

Whenever a mismatch exists between how the validator and the browser decode these special characters, you can exploit the mismatch to induce an open redirect. For example, some validators might decode these URLs completely, then assume the URL redirects to *example.com*, since *@attacker.com* is in the path portion of the URL. However, the browsers might decode the URL incompletely, and instead treat *example.com%25252f* as the username portion of the URL.

On the other hand, if the validator doesn't double-decode URLs, but the browser does, you can use a payload like this one:

```
https://attacker.com%252f@example.com
```

The validator would see *example.com* as the hostname. But the browser would redirect to *attacker.com*, because *@example.com* becomes the path portion of the URL, like this:

```
https://attacker.com/@example.com
```

Non-ASCII Characters

You can sometimes exploit inconsistencies in the way the validator and browsers decode non-ASCII characters. For example, let's say that this URL has passed URL validation:

```
https://attacker.com%ff.example.com
```

`%ff` is the character ÿ, which is a non-ASCII character. The validator has determined that *example.com* is the domain name, and *attacker.comÿ* is the subdomain name. Several scenarios could happen. Sometimes browsers decode non-ASCII characters into question marks. In this case, *example.com* would become part of the URL query, not the hostname, and the browser would navigate to *attacker.com* instead:

```
https://attacker.com?.example.com
```

Another common scenario is that browsers will attempt to find a "most alike" character. For example, if the character ╱ (%E2%95%B1) appears in a URL like this, the validator might determine that the hostname is *example.com*:

```
https://attacker.com╱.example.com
```

But the browser converts the slash look-alike character into an actual slash, making *attacker.com* the hostname instead:

```
https://attacker.com/.example.com
```

Browsers normalize URLs this way often in an attempt to be user-friendly. In addition to similar symbols, you can use character sets in other languages to bypass filters. The *Unicode* standard is a set of codes developed to represent all of the world's languages on the computer. You can find a list of Unicode characters at *http://www.unicode.org/charts/*. Use the Unicode chart to find look-alike characters and insert them in URLs to bypass filters. The *Cyrillic* character set is especially useful since it contains many characters similar to ASCII characters.

Combining Exploit Techniques

To defeat more-sophisticated URL validators, combine multiple strategies to bypass layered defenses. I've found the following payload to be useful:

```
https://example.com%252f@attacker.com/example.com
```

This URL bypasses protection that checks only that a URL contains, starts with, or ends with an allowlisted hostname by making the URL both start and end with *example.com*. Most browsers will interpret *example.com%252f* as the username portion of the URL. But if the validator over-decodes the URL, it will confuse *example.com* as the hostname portion:

```
    https://example.com/@attacker.com/example.com
```

You can use many more methods to defeat URL validators. In this section, I've provided an overview of the most common ones. Try each of them to check for weaknesses in the validator you are testing. If you have time, experiment with URLs to invent new ways of bypassing URL validators. For example, try inserting random non-ASCII characters into a URL, or intentionally messing up its different components, and see how browsers interpret it.

Escalating the Attack

Attackers could use open redirects by themselves to make their phishing attacks more credible. For example, they could send this URL in an email to a user: *https://example.com/login?next=https://attacker.com/fake_login.html.*

Though this URL would first lead users to the legitimate website, it would redirect them to the attacker's site after login. The attacker could host a fake login page on a malicious site that mirrors the legitimate site's login page, and prompt the user to log in again with a message like this one:

> Sorry! The password you provided was incorrect. Please enter your username and password again.

Believing they've entered an incorrect password, the user would provide their credentials to the attacker's site. At this point, the attacker's site could even redirect the user back to the legitimate site to keep the victim from realizing that their credentials were stolen.

Since organizations can't prevent phishing completely (because those attacks depend on human judgment), security teams will often dismiss open redirects as trivial bugs if reported on their own. But open redirects can often serve as a part of a bug chain to achieve a bigger impact. For example, an open redirect can help you bypass URL blocklists and allowlists. Take this URL, for example:

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

This URL will pass even well-implemented URL validators, because the URL is technically still on the legitimate website. Open redirects can, therefore, help you maximize the impact of vulnerabilities like server-side request forgery (SSRF), which I'll discuss in Chapter 13. If a site utilizes an allowlist to prevent SSRFs and allows requests to only a list of predefined URLs, an attacker can utilize an open redirect within those allowlisted pages to redirect the request anywhere.

You could also use open redirects to steal credentials and OAuth tokens. Often, when a page redirects to another site, browsers will include the originating URL as a referer HTTP request header. When the originating URL contains sensitive information, like authentication tokens, attackers can induce an open redirect to steal the tokens via the referer header. (Even when there is no open redirect on the sensitive endpoint, there are ways to smuggle tokens offsite by using open redirect chains. I'll go into detail about how these attacks work in Chapter 20.)

Finding Your First Open Redirect!

You're ready to find your first open redirect. Follow the steps covered in this chapter to test your target applications:

1. Search for redirect URL parameters. These might be vulnerable to parameter-based open redirect.
2. Search for pages that perform referer-based redirects. These are candidates for a referer-based open redirect.
3. Test the pages and parameters you've found for open redirects.
4. If the server blocks the open redirect, try the protection bypass techniques mentioned in this chapter.
5. Brainstorm ways of using the open redirect in your other bug chains!