Open Redirection in Misconfigured OAuth

Ravi Solanki[1], Prof. Chandresh Parekh[2]

[1]*Student, M.Tech, School of Information Technology & Cyber Security, Raksha Shakti University*

[2]*Dean, School of Information Technology & Cyber Security, Raksha Shakti University*

Abstract- **OAuth2.0 is generally used by online facility providers worldwide. OAuth security-related banners appear from time to time, and mismanagement of the protocol caused many difficulties. It verifies the user's identity for the requested website without revealing the password to the website. When a web application receives untrustworthy input, it causes the request to be readdressed to the underlying URL without any input, redirects and forwards are potential. The user-agent redirection system in OAuth is the vulnerable links because hard for developers and operators to the right way read, understand and implement all the subtle but significant requirements. In this discussion, we begin by identifying the security community's understanding of the OAuth redirection threats. The current process of the OAuth requirement, as well as its circulating best practice, will be discussed. We announce new OAuth redirection attack technologies that activity the interaction of URL construing issues with redirection controlling in majority browsers and mobile applications. In explicit, it allows attackers to hijack third party app accounts, gain access to sensitive personal info, or take special actions on behalf of affected users.**

Index terms- **Oauth2.0, Misconfigured, web Applications, Open Redirection.**

I.INTRODUCTION

OAuth is an open standard for token-based authentication and authorization over the Internet. OAuth, known as "oh-auth", allows end-user information to be used for third-party services such as Facebook, without revealing the user's password. We all know that a recent Facebook breach was caused by a leak of access tokens. The session token / access token or OAuth token is very sensitive data because if an attacker receives this information, your account can be logged into your account without knowing your password.

II. OVERVIEW OF OAUTH

A. OAUTH 2.0

OAuth is the authoritative framework for web applications. It verifies the user's identity for the requested website without revealing the password to the website. This may seem complex at basic, but give an illustration: the user wants to log into the website. He goes to the signup page and finds three options to login - via Facebook, Google, LinkedIn. When a user connects on one of them, he validates himself on the website.

B. OPEN REDIRECT

According to OWASP, when a web application receives unreliable inputs, invalid redirects and forwards are possible, causing the web application to redirect request to URL with no input. By change entrusted URL input on a malevolent site, an attacker can successfully establish a phishing trick and take user credentials. Since the server name on the revised link is identical to the original site, phishing efforts may have a more reliable presence. Unrelated redirects and subsequent attacks can also be used to maliciously create a URL that periapt under the app's access control check, and then guarantor the attacker to particular tasks that they cannot normally use.

III. SECURE VS UNSECURE URL REDIRECTS

A. RISKY URL REDIRECTS

If the verification or additional method of control isn't implemented to verify the accuracy of the URL, the code below is vulnerable. This vulnerability is often used as part of a phishing scam by redirecting users to malicious sites. If authentication isn't applied, a malicious user may create a hyperlink to redirect their users to an undetected malicious website, for example:

http://example.com/example.php?url=http://malicious.example.com

The user specifies a link to the first trusted site (example.com) and doesn't perceive the redirection which will occur.

C. SECURE URL REDIRECTS

When we want to automatically redirect a user to another page (without the Visitant activity such as clicking a hyperlink) You can run the code this way.

PHP:

```
<? php
/ * Redirected Browser * /
Title ("Location: http://www.mysite.com/");
?>
```

In the above illustration, the URL is implicitly declared in the code and cannot be done be changed by the attacker.

D. REDIRECT AND FORWARD CAN BE USED SAFELY

Avoid redirects and further use. If used, do not allow URLs to be the user's input for the destination. This can usually be done. In this case, you need to have a way to verify the URL.

If the user cannot avoid input, make sure the value supplied is valid, compatible with the application, and that the user is authorized. Force all redirects to visit the first lead, let the user know they are visiting your site

IV. TYPICAL PATTERNS OF URL VALIDATOR

During a large evaluation of real-world OAuth implementations, we noticed that URL validators of OAuth providers behave differently. In this section we will list most of the types of URL validator behaviours we have seen, and we will discuss each of them in the next section

A. DOMAIN WHITELIST

Some OAuth providers, especially those with legacy, allow clients to be explicitly configured without redirect_ury. They only check the domain portion of the URL to make sure the scheme is HTTP or HTTP. Some of them also whitelist all subdomains of the configured domain. In such cases, if the domain domain.tl is whitelisted, https: //sub.domain.tld/a/b will still be a valid redirect URL.

B. PREFIX MATCHING

Most OAuth providers require users to configure redirect_url when registering an OAuth client. However, most of them only verify the redirection provided in the request with prefix matching. In that case, suppose a developer verifies the registered https: //domain.tld/a as redirect_url, https: //domain.tld/abc. Note that some implementations parse and validate domains in addition to pre-matching.

C. ARBITRARY SCHEME

We have also seen OAuth providers checking for strict compatibility for domains and paths, but allow for any custom scheme. Their intent is to allow developers to use OAuth for native applications. In such cases, a URL such as x: //domain.tld/a is allowed.

V. BROWSER EXPLOITATION

The in general, to successfully use OAuth redirect_security, the first step is to find a way to leak the code or access the victim's token. Anonymous redirects do this by using Open Redirect on the website hosting the OAuth client, we focus on finding bypasses For the URL verifier. In other words, we break the URL behind the validation-2. The methods we use to skip URL validators are categorized below. In this section, the URL contains all the green text host components. The URL on the left side of the arrow (on) indicates the validation of the OAuth provider's URL, while the URL on the right shows the commentary on the browsers.

A. FOOTING IN ENCODING / DECODING

Encoding / decoding is complex and it is easy to fix errors, which have been popular for decades. A comprehensive study and summarized in the Unicode Security Guide [25], many classic Unicode attack tactics still apply today. We present here three attack vectors operating in several implementations. We hope there are more attack vectors that can use the encoding / decoding error.

B. CONSUMPTION IS HIGH

If user credentials are allowed, test with the following vector:

https://attacker%ff@benign.com

https://attackernign.com

If sub-domains are allowed, test with the following vector:

https://attacker%ff.benign.com
https://attackernign.com
Explanation: When the decoder on the server meets a larger character than the ASCII range, it tries to decode it using Unicode with forwarding characters. Such weaknesses [25] have been described as XSS attacks technology. Here we use it to build redirection exploits.

C. MARK PAY ATTENTION TO THE QUESTION MARK

Attack Case 1(Error Decoding by Server):
https://attacker.com%ff@benign.com
https://attacker.com? @benign.com
Explanation: When validating a domain, the parser captures benign.com as a domain. Was the printable converted to%% before the URL was output? Therefore, the browser sends a request to the attacker.com. We found this method in the bug report

D. THE BEST FIT MAPPING

Assault Case 1:
https://attacker.com／.benign.com
https://attacker.com/.benign.comExplanation: The parser has a full-width character, but browsers like Edge or some previous versions of IE normalize it to a half-width character.
Assault Case 2:
https://benign.com／@attacker.com
https://benign.com／@attacker.com
Explanation: The parser normalizes the full-width character to the half-width character, while the browser has the full-width character.

VI. EVIL SLASH TRICK

Most browsers treat both as / and path separators, and when the address bar contains the user input URL, most browsers automatically become \ /. According to the URL standard [8], this is the desired behavior. However, both the URL verifier and the browser can go wrong.

- The forward slash is not considered a path separator, the browser does.

https://attacker.com\@benign.com
https://attacker.com/@benign.com
Explanation: The parser does not behave as a separator and captures Benign.com as a domain, the browser exchanges / and sends the request to the attacker.

- Parser treats forward slash as path separator, browser does not.

https://benign.com\@attacker.com
https://benign.com\@attacker.com
Explanation: This attack relies on the new Safari bug we first exposed, working on the latest version of Safari at the time of writing. When performing redirection, Safari allows user-information and is not considered a path separator. When the parser acts as a Path Separator and manages it in the output, Safari is redirected to Attacker.com..

VII. PRACTICAL EXPLOITATION

A. CODE INJECTION

OAuth has a policy to protect against code leakage via redirect_url. Authentication-1 needs to be redirected to level the authentication request and the token exchange request, which is good for this purpose. As long as the token exchange request arrives, the AS request to store the redirect_url to the authorization request. We have noticed that some ASA tokens cannot verify redirect_url unless they appear in the conversion request. If the token is not provided by its customer at the request, the relief is invalid. In fact, this may explain our observation that the number of implemented injections is vulnerable to code injection. This problem has also been observed and mentioned [13], and some alternative countermeasures have been proposed, such as floating, code-bound state, or PKCE.
As an attacker, the simplest and most effective technique to try is to change the response_type from "code" to "token" and test the underlying flow if it supports it. By doing this, the attacker can directly access_token and skip any code injection. This is an old attack known as the application attack, which was discussed in 2014. However, in practice, it works quite well these days.
Another obstacle to using code injection is the state variable. There is a misunderstanding among developers and security researchers that session-bound state variables can prevent code injection attacks. The truth is that only the code-bound state variable can prevent code injection, while the session-bound state variable only prevents CSRF.

Worse, in fact, there are many implementation flaws for state certification. In many cases, an attacker can reuse any valid state or create a valid session-state pair by stopping an OAuth authentication request.

B. BURGLARY BLINDLY

The OAuth redirection vulnerability caused by the URL parser we discussed affects the vulnerability of the provider and all of its OAuth clients. Meanwhile, most implementations support auto-compliance mechanisms that allow automatic authentication after the first time, giving the attacker the ability to perform CSRF style stealth attacks. A very stolen technology The OAuth authority should create an image that represents the URL. Attackers may also place malicious images on some online social platforms. Of course, two conditions must be met for this attack to work.

1. The user is logged into the provider (AS) and the login session is still valid.
2. There is no consent page, which means that auto-consent applies because the user is usually given access to the customer.

VIII. CONCLUSION

For OAuth, I strongly suggest reading the OAuth 2.0 Security Best Current Practice Draft and checking every security threat against your implementation. The best and simplest solution is to use simple string comparisons for URL validation, to avoid redirect_url related weaknesses. For some reason, to use a format such as a domain whitelist, the provider must make sure that validation-1 and validation-2 are implemented correctly, or refer to Section 3.2. Optional code to reduce injection. If the provider uses URL pattern matching, make sure the other API endpoint/webpage does not have a URL matching the pattern. It is a good practice to use a specific subdomain for the authentication endpoint.

As a URL parser, I suggest developers use the popular libraries URL parser if possible. If developers need to implement the URL parser manually, it is safer to follow the latest WHATWG standard. Examine all components involved in URL processing and note encoding / decoding issues.

REFERENCES

[1] R. Fielding, “Hypertext Transfer Protocol -- HTTP/1.1,” [Online]. Available: https://tools.ietf.org/html/rfc2616. [Accessed 1999].

[2] W. Denniss, “OAuth 2.0 for Native Apps,” 2017. [Online]. Available: https://tools.ietf.org/html /rfc8252.

[3] D. Hardt, “The OAuth 2.0 authorization framework (No. RFC 6749).,” 2012. [Online]. Available: https://tools.ietf.org/html/rfc6749.

[4] R. Yang, W. C. Lau and S. Shi, “Breaking and Fixing Mobile App Authentication with OAuth2.0-based Protocols,” in International Conference on Applied Cryptography and Network Security, 2017.

[5] R. Yang, W. C. Lau and T. Liu, “Signing into One Billion Mobile App Accounts Effortlessly with OAuth2.0,” in Black Hat Europe, 2016.

[6] T. Berners-Lee, “Uniform Resource Locators (URL),” 1994. [Online]. Available: https://tools.ietf.org/html/rfc1738.

[7] T. Berners-Lee, “Uniform Resource Identifier (URI): Generic Syntax,” 2005. [Online]. Available: https://tools.ietf.org/html/rfc3986.

[8] WhatWG, “URL Living Standard,” 2018. [Online]. Available: https://url.spec.whatwg.org.

[9] E. T. Lodderstedt, “OAuth 2.0 Threat Model and Security Considerations,” 2013. [Online]. Available: https://tools.ietf.org/html/rfc6819.

[10] N. Sakimura, “OpenID Connect Core 1.0,” 2014. [Online]. Available: https://openid.net/specs/ openid-connect-core-1_0-final.html.

[11] J. Wang, “Covert Redirect Vulnerability,” 2014. [Online]. Available: http://tetraph.com/covert _redirect/.

[12] J. Bradley, “Covert Redirect and its real impact on OAuth and OpenID Connect,” 2014. [Online]. Available: http://www.thread-safe.com/2014/05/covert-redirect-and-its-real-impact-on.html.

[13] E. T. Lodderstedt, “OAuth 2.0 Security Best Current Practice (draft 07),” 2018. [Online]. Available: https://tools.ietf.org/html/draft-ietf-oauth-security-topics-07.

[14] Paypal, “Stricter Redirect Checks Required on Log In With PayPal Applications,” 2015. [Online]. Available: https://www.paypal-engineering.com/2015/03/30/stricter-redirect-

checks-required-on-log-in-with-paypal-applications/.

[15] Facebook, “Strict URI Matching,” 2017. [Online]. Available: https://developers.facebook.com/blog/post/2017/12/18/strict-uri-matching/.

[16] E. Homakov, “How I hacked Github again,” 2014. [Online]. Available: http://homakov.blogspot.com/2014/02/how-i-hacked-github-again.html.

[17] J. Wang, “Microsoft Live Online Service OAuth 2.0 Covert Redirect Web Security Bugs (Information Leakage & Open Redirect),” 2014. [Online]. Available: http://www.tetraph.com/blog/covert-redirect/microsoft-lives-oauth-2-0-covert-redirect- vulnerablity/.

[18] prakharprasad, “Slack OAuth2 "redirect_uri" Bypass,” Hackerone, 2014. [Online]. Available: https://hackerone.com/reports/2575.

[19] ethancruize, “Stealing Users OAUTH Tokens via redirect_uri,” Hackerone, 2018. [Online]. Available: https://hackerone.com/reports/405100.

[20] N. B. S. Harsha, “Oauth 2.0 redirection bypass cheat sheet,” 2016. [Online]. Available: http://nbsriharsha.blogspot.com/?view=sidebar.

[21] filedescriptor, “Bypassing callback_url validation on Digits,” Hackerone, 2016. [Online]. Available: https://hackerone.com/reports/108113.

[22] filedescriptor, “Internet Explorer has a URL problem,” 2016. [Online]. Available: https://blog.innerht.ml/internet-explorer-has-a-url-problem/.

[23] Y. Tian, “1000 Ways To Die In Mobile OAuth,” in Black Hat USA, 2016.