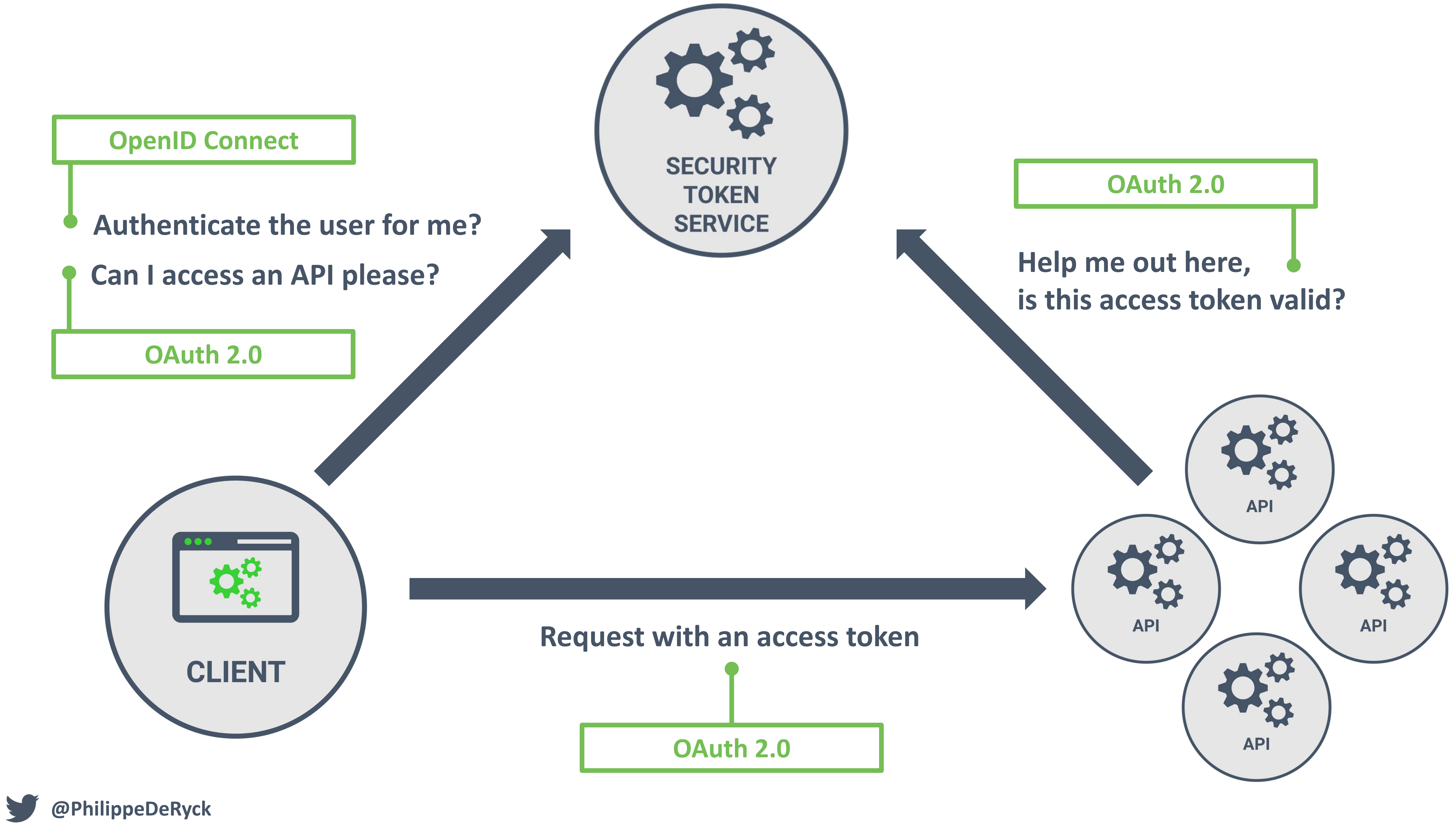
Securing OAuth 2.0 and OIDC in Frontends

Dr. Philippe De Ryck



OpenID Connect
Authenticate the user for me?
Can I access an API please?
OAuth 2.0
SECURITY
TOKEN
SERVICE
OAuth 2.0
Help me out here,
is this access token valid?
CLIENT
API
API
API
API
Request with an access token
OAuth 2.0
@PhilippeDeRyck

TERMINOLOGY

This session	OAuth 2.0	OpenID Connect
User	Resource Owner	End-User
API	Resource Server	
Security Token Service (STS)	Authorization Server	OpenID Provider
Client	Client	Relying Party

How do you secure tokens in the frontend?

I am *Dr. Philippe De Ryck*

Founder of Pragmatic Web Security

Google Developer Expert

Auth0 Ambassador

SecAppDev organizer

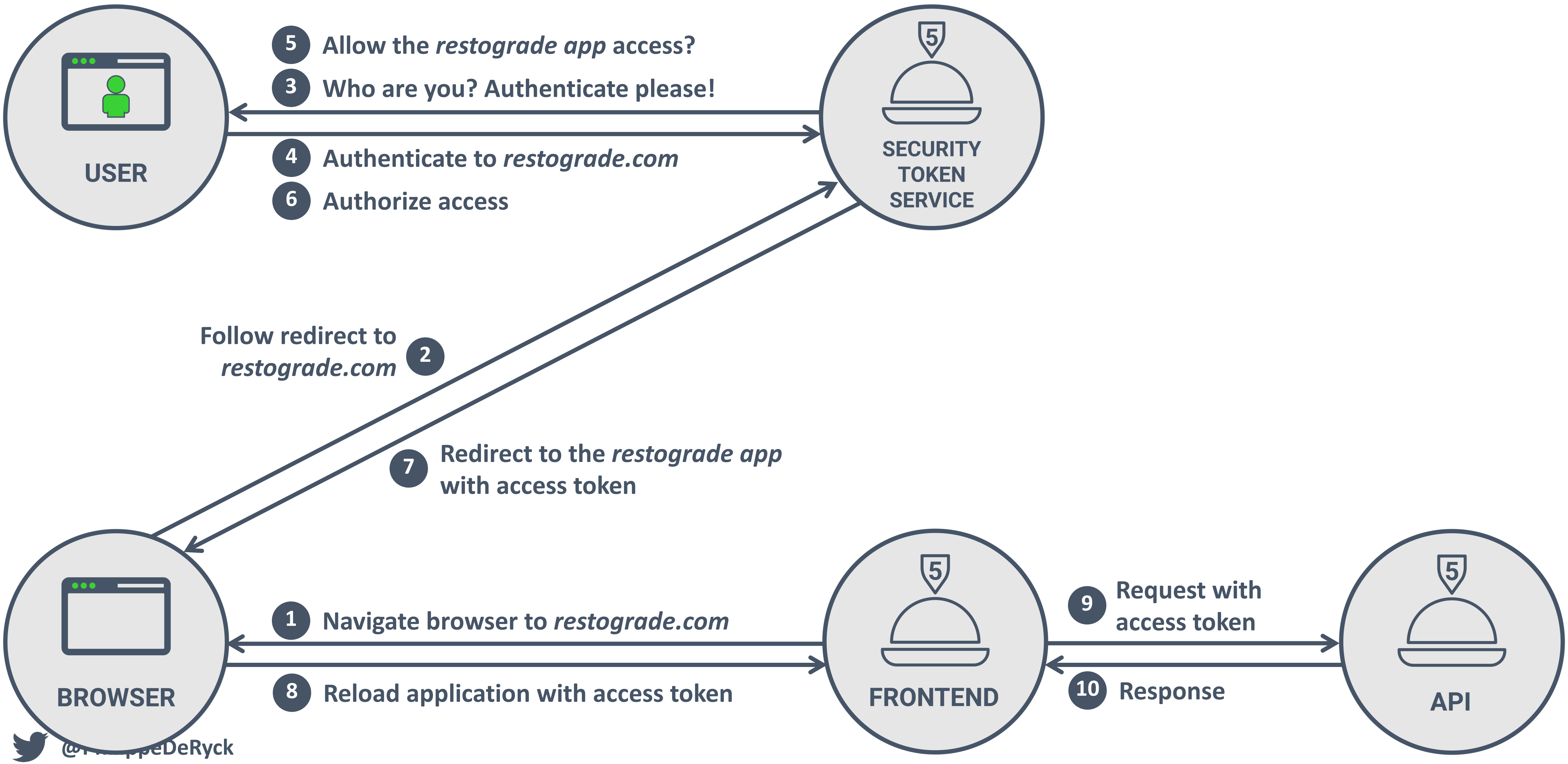
I help developers with security

- Hands-on in-depth security training
- Advanced online security courses
- Security advisory services

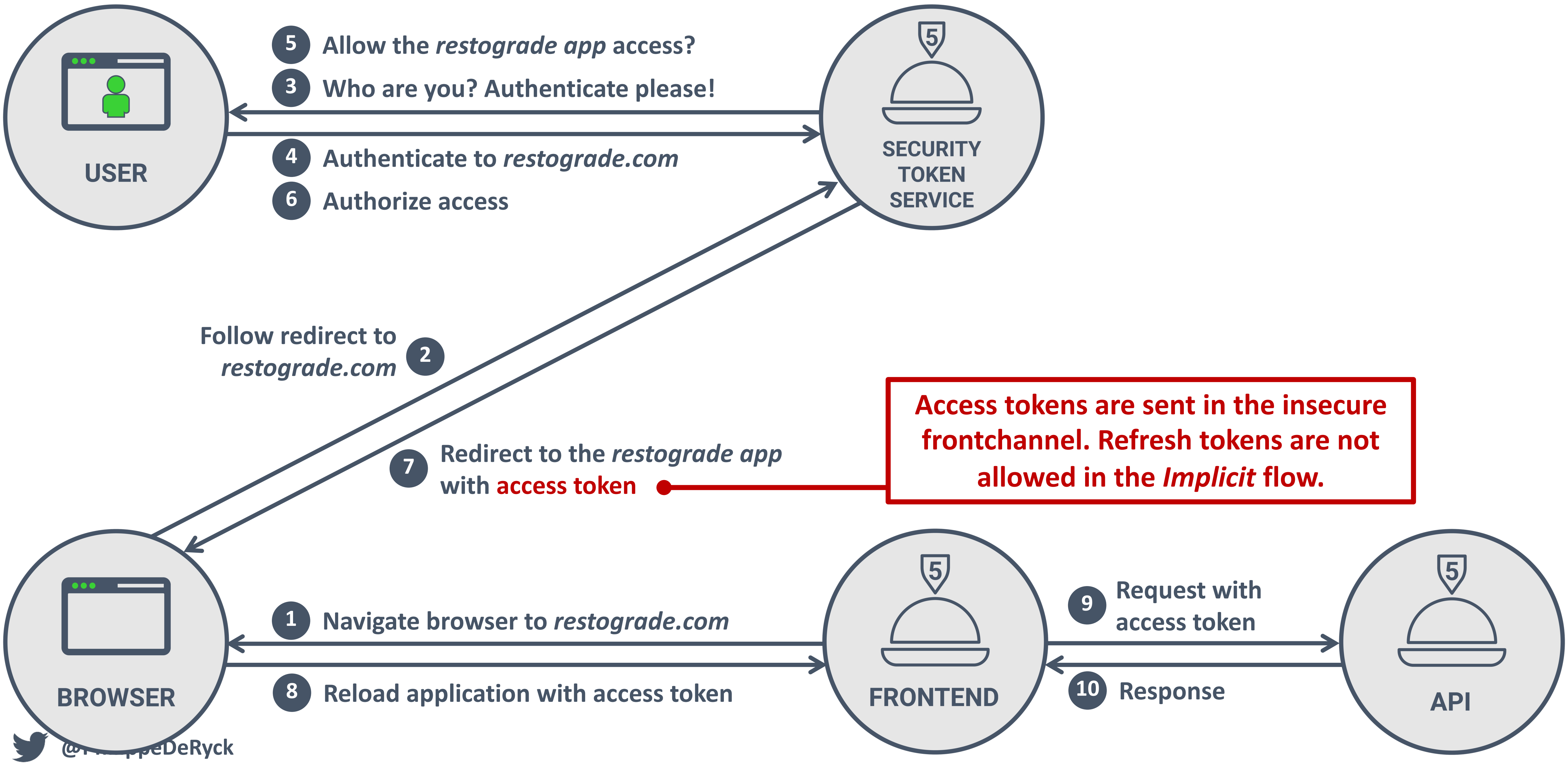
https://pragmaticwebsecurity.com

OAUTH 2.0 AND OIDC IN SPAS

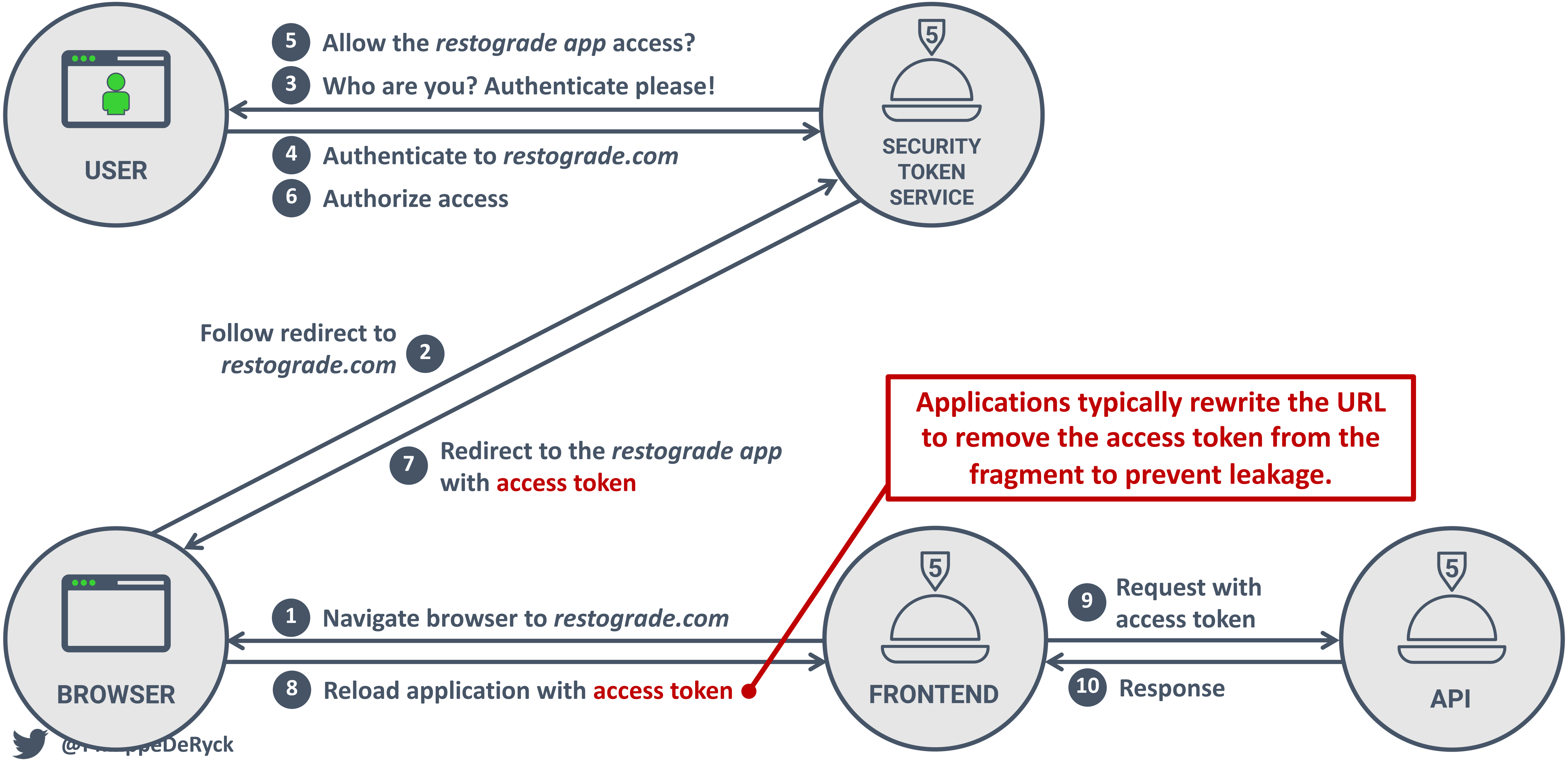
THE *IMPLICIT* FLOW



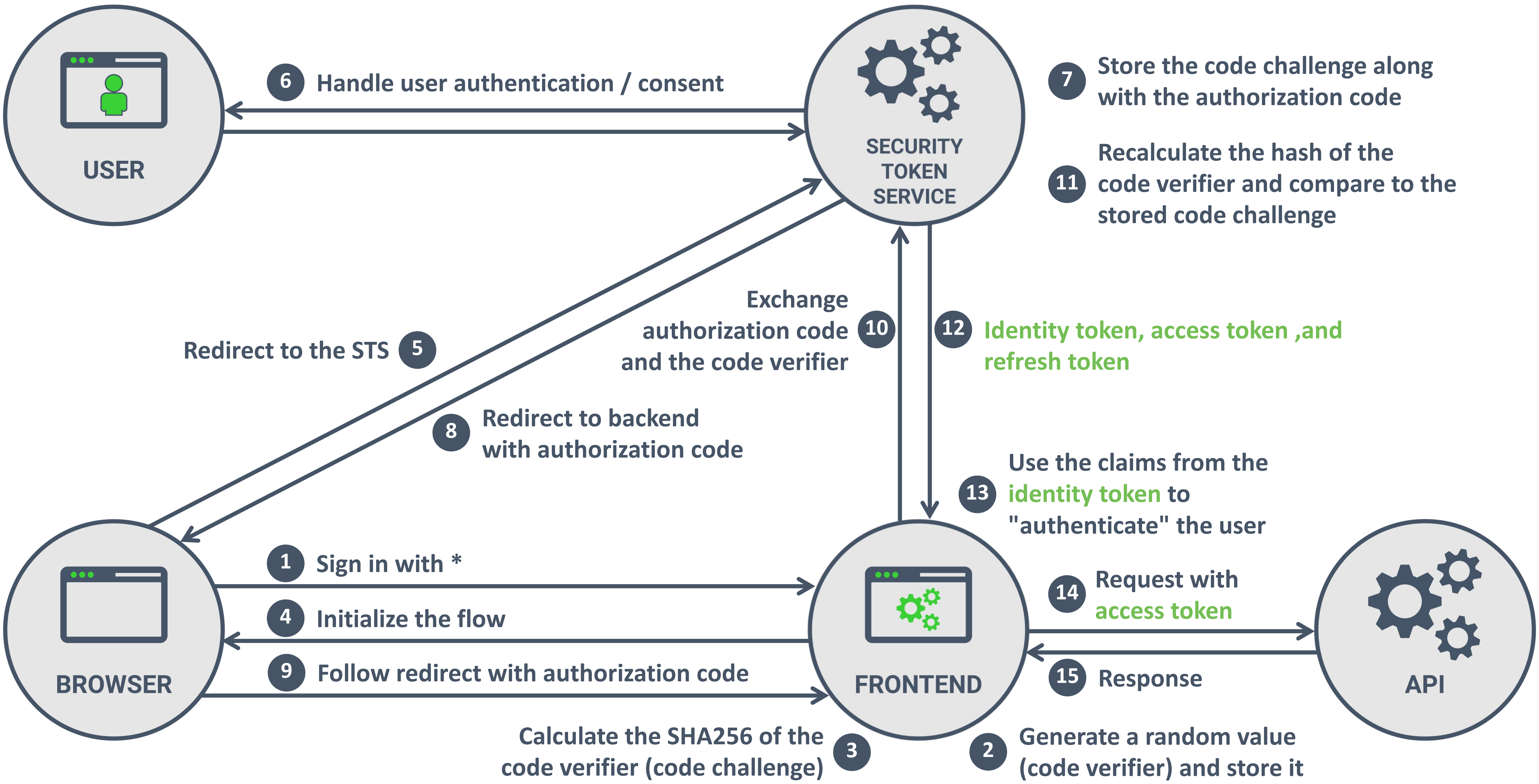
The *Implicit* Flow



THE *IMPLICIT* FLOW



The OAuth 2.0 Security Best Practices and OAuth 2.1 specifications deprecate the *Implicit* flow



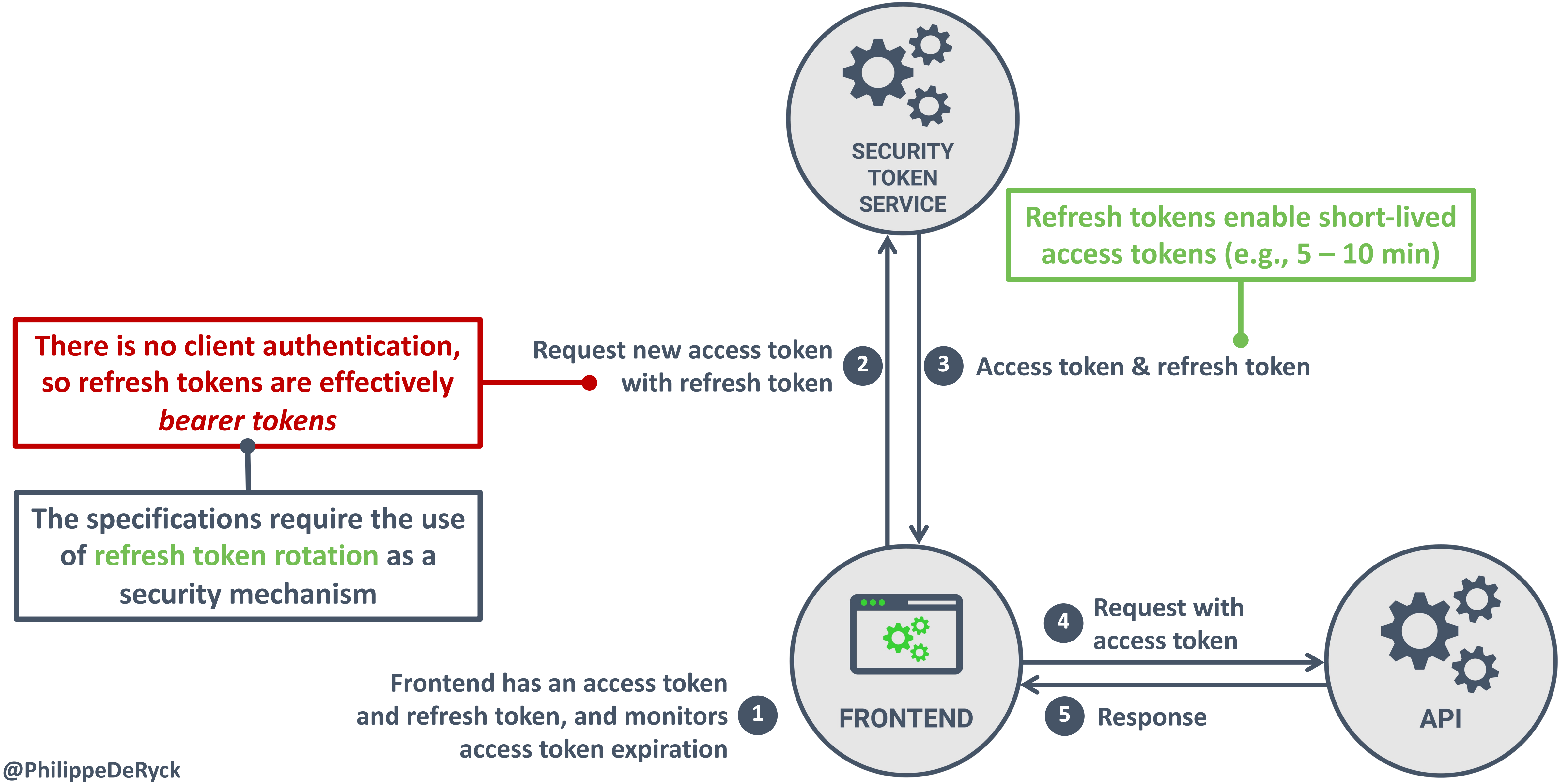
@PhilippeDeRyck

Frontends use the Authorization Code flow

The authorization code flow with PKCE allows the user to delegate authority to an application to access APIs on their behalf

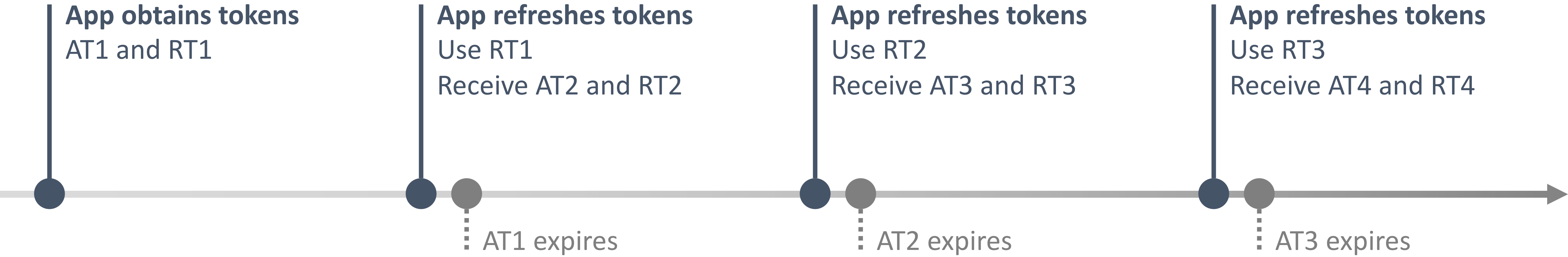
USING REFRESH TOKENS

The *Refresh Token* flow



Refresh token rotation

- Refresh token rotation is required for using refresh tokens in the browser
  - Part of the ***OAuth 2.0 for Browser-Based Apps*** proposal
  - Refresh tokens are used once to obtain a new access token and new refresh token
  - Previously used refresh tokens become invalid



REFRESH TOKEN ROTATION

2 *The request to exchange a refresh token*

```
POST https://sts.restograde.com/oauth/token

 grant_type=refresh_token
&client_id=DtsTliLAWq3JXIwaoPQzl8vXhNI6qGnb
&refresh_token=8xLOxBtZp8
```

The latest refresh token obtained from the STS

3 *The response from the Security Token Service*

```
{
  "access_token": "eyJhbGciO...du6TY9w",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "mTVeKoIZYy",
}
```

A new access token

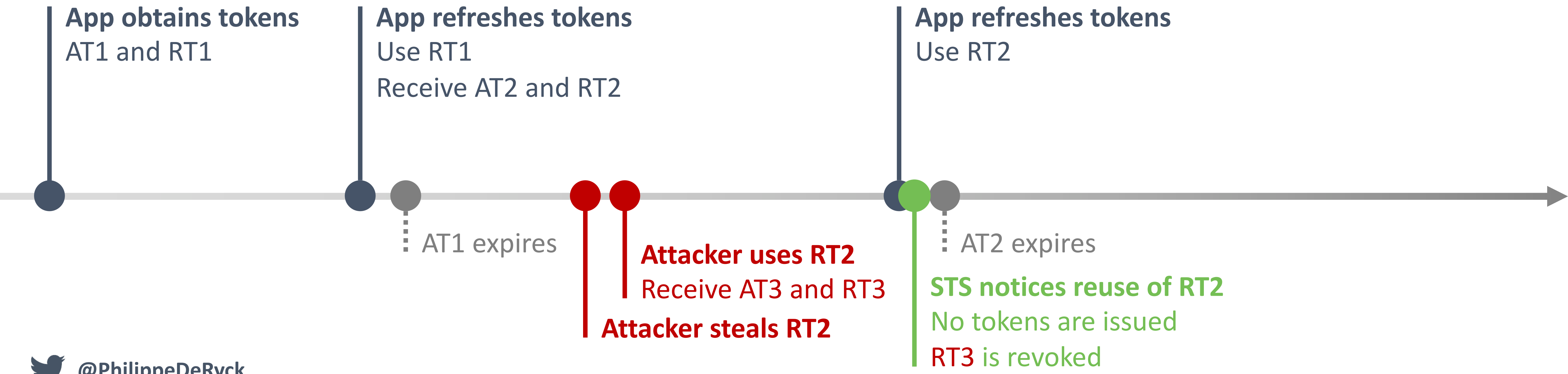
A new refresh token to be used in the next flow

What should the STS do when it detects refresh token re-use?

- **A** Nothing
- **B** Issue a new access token, but not a new refresh token
- **C** Not issue new tokens
- **D** Revoke all tokens associated with the re-used refresh token

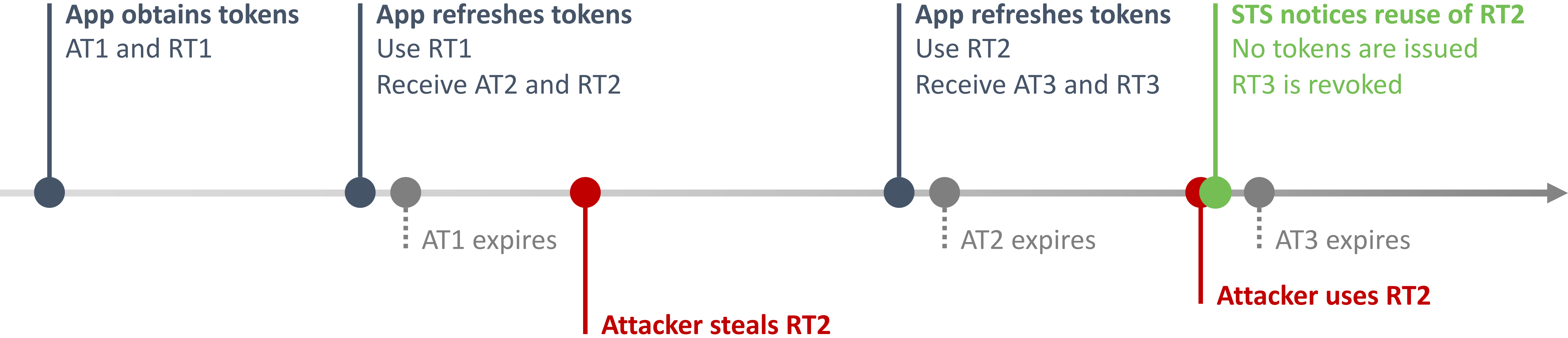
Detecting refresh token abuse

- When the STS detects the re-use of a refresh token, something is wrong
  - The refresh token is immediately revoked, preventing abuse
- To ensure security, the STS revokes the entire token chain of this refresh token
  - The abuse of *RT2* leads to the revocation of *RT3*, *RT4*, …



DETECTING REFRESH TOKEN ABUSE

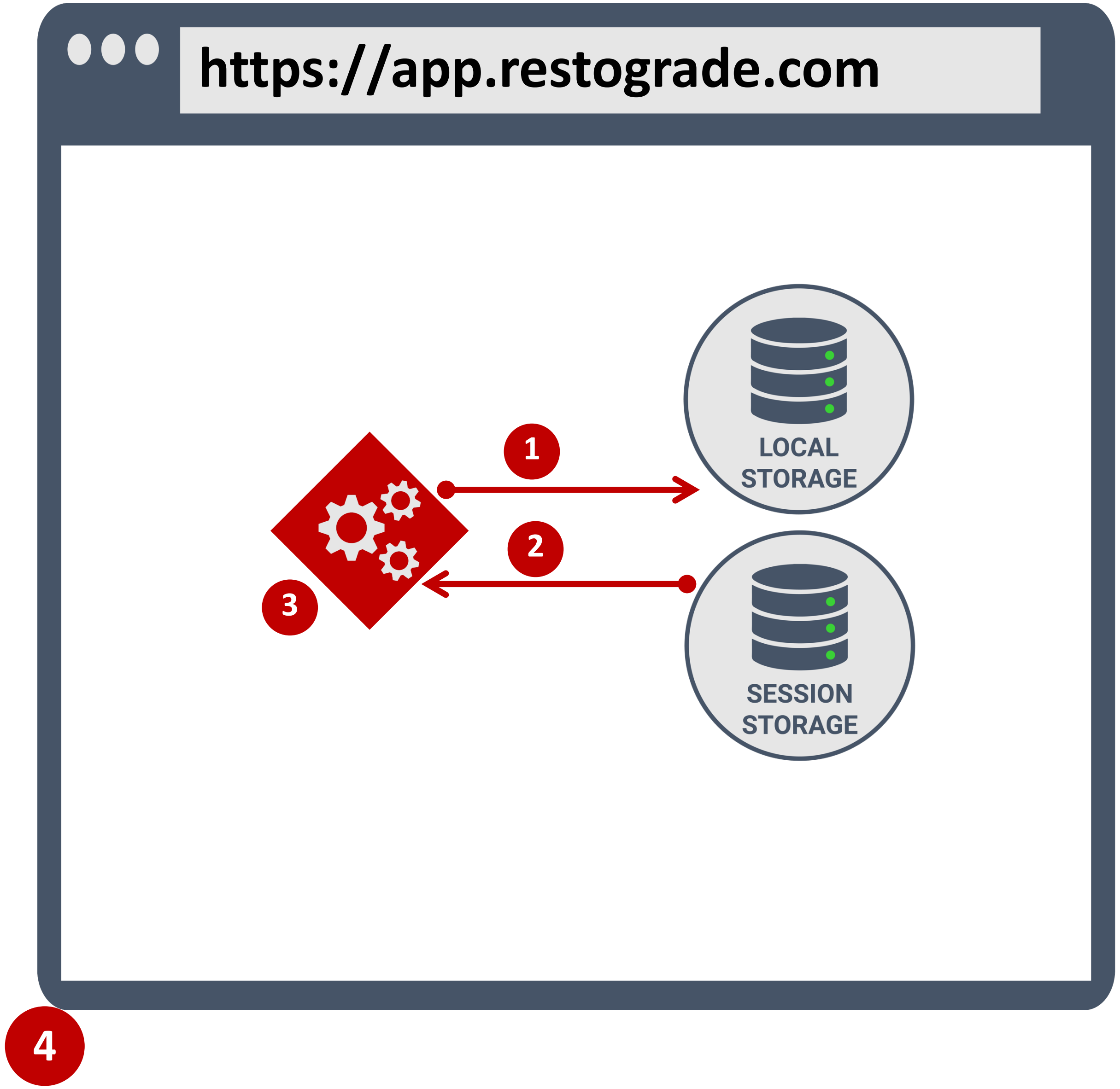
- When the STS detects the re-use of a refresh token, something is wrong
  - The refresh token is immediately revoked, preventing immediate abuse
- To ensure security, the STS revokes the entire token chain of this refresh token
  - The abuse of *RT2* leads to the revocation of *RT3*, *RT4*, …



Refresh token rotation in action

ATTACKING OAUTH 2.0 IN FRONTENDS

The common perception of malicious JavaScript



1. Request all data from localStorage/sessionStorage
2. Return all data to the JS code requesting it
3. Send data to a server controlled by the attacker
4. Abuse the stolen data (access token, refresh token)

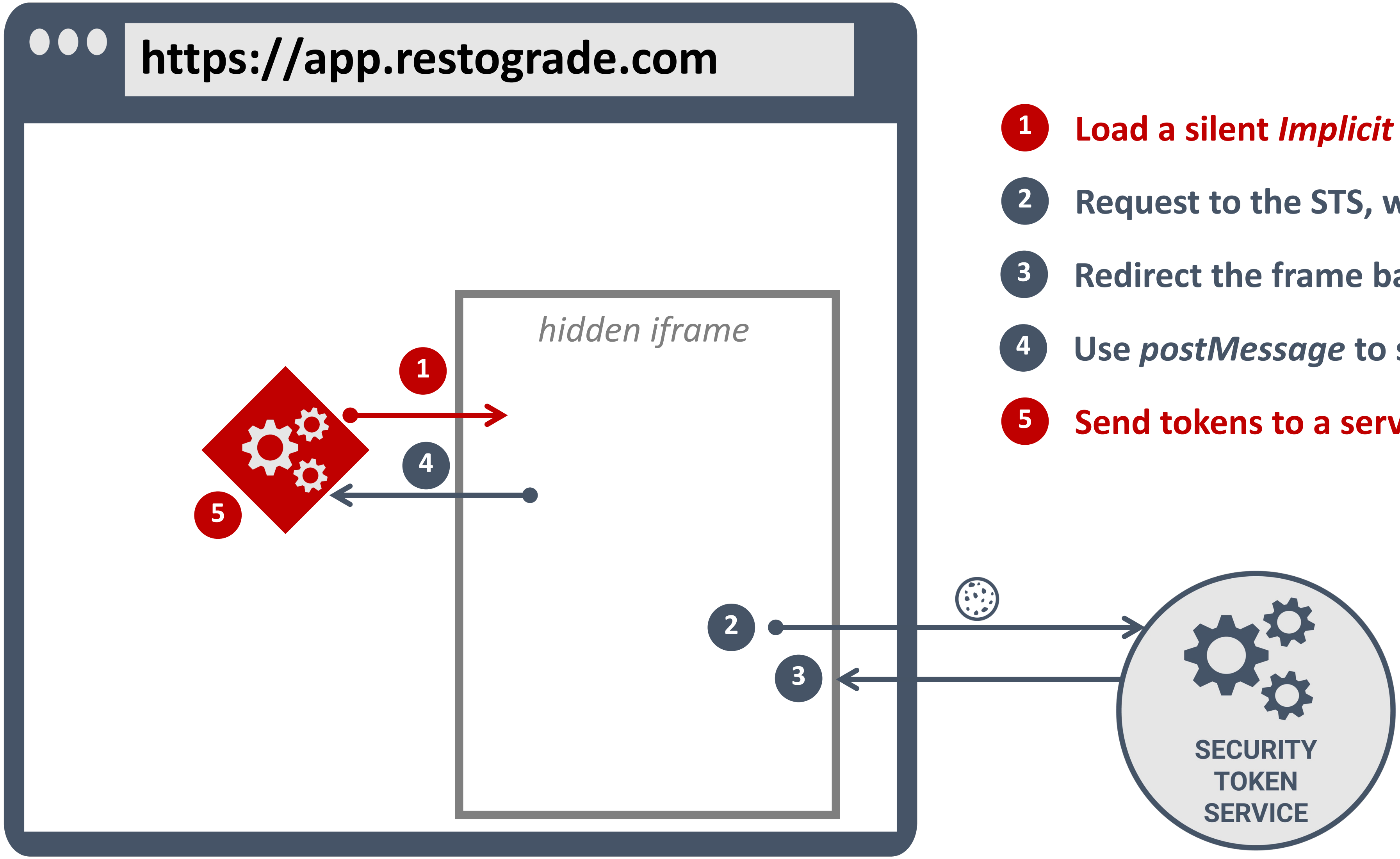
Short-lived access tokens reduce the impact of stolen access tokens

Refresh token rotation prevents re-use of stolen refresh tokens

The underestimated threat model of malicious JS

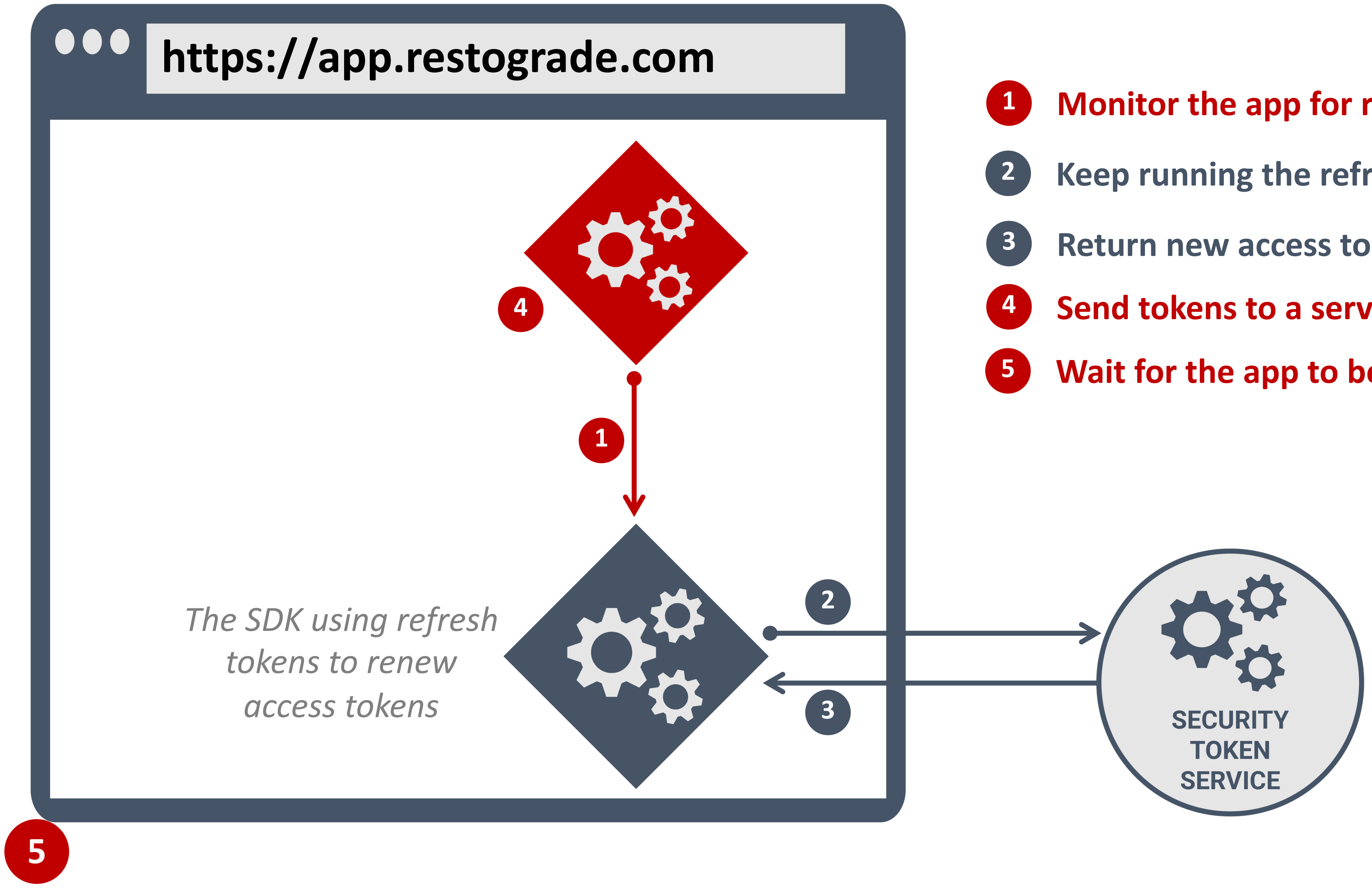
- Malicious JS code runs in the same environment as the application code
  - All code running in that environment has the same privileges
  - The browser does not and cannot differentiate between code sources
- When the browser executes malicious JS, the attacker controls the application
  - Stealing data from storage areas is a simplistic attack payload
- The malicious code can perform any action the legitimate application can
  - Any storage area reachable to the application is reachable to the attacker
  - Function definitions in the runtime environment can be manipulated
  - Iframe-based flows relying on cookies in the browser can be executed by the attacker
  - API requests sent by the malicious code are indistinguishable from legitimate requests

STEALING ACCESS TOKENS WITH THE *IMPLICIT* SILENT RENEW



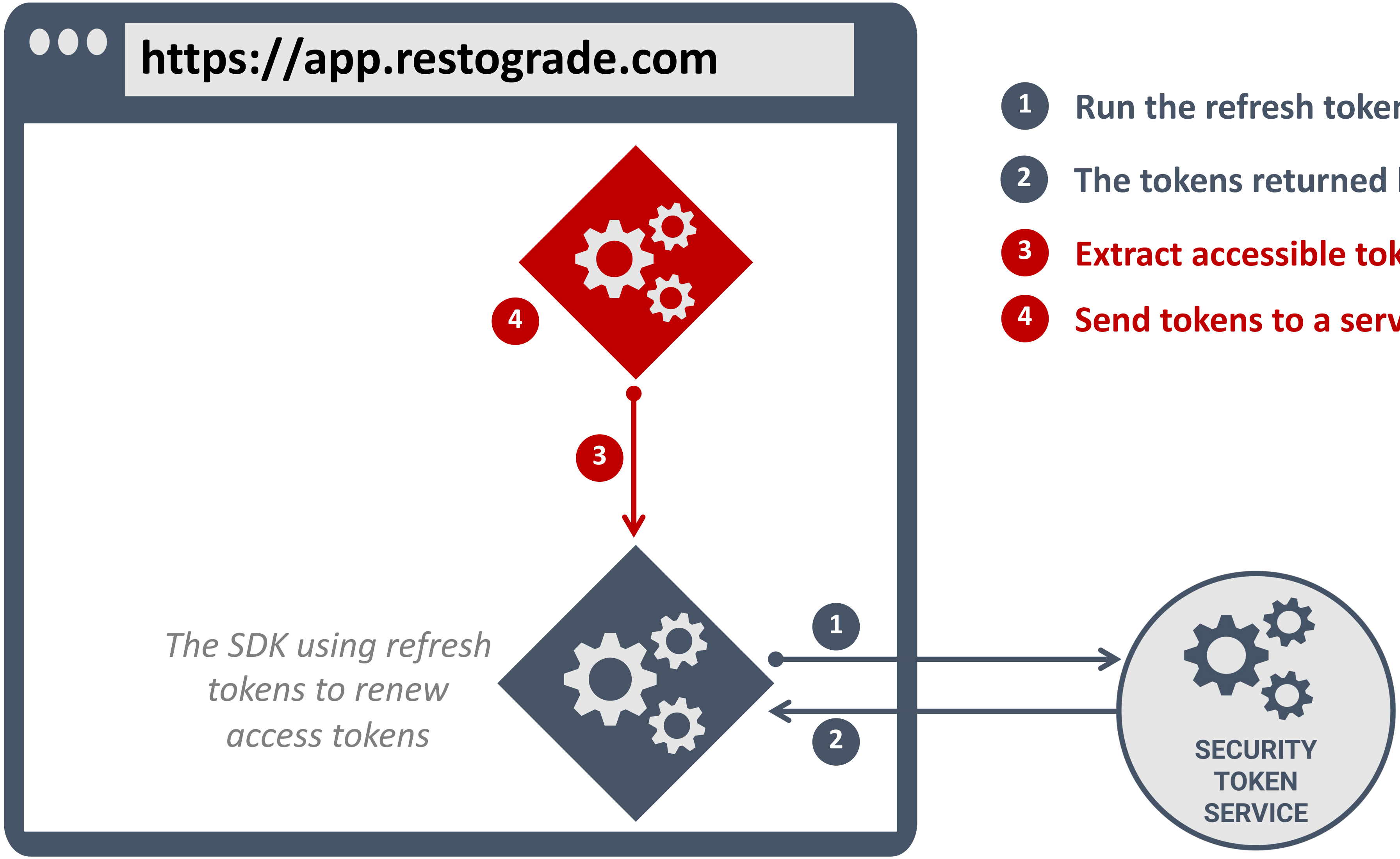
1. Load a silent *Implicit* flow in an iframe
2. Request to the STS, with cookies if available
3. Redirect the frame back to the app, with tokens in URL
4. Use *postMessage* to send tokens to parent frame
5. Send tokens to a server controlled by the attacker

SIDESTEPPING REFRESH TOKEN ROTATION



1. Monitor the app for refresh tokens (if available)
2. Keep running the refresh flow when needed
3. Return new access tokens and refresh tokens
4. Send tokens to a server controlled by the attacker
5. Wait for the app to become inactive to use RT

STEALING ACCESS TOKENS FROM THE LEGITIMATE SDK



1. Run the refresh token flow to get fresh tokens
2. The tokens returned by the STS
3. Extract accessible tokens from the JS environment
4. Send tokens to a server controlled by the attacker

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.

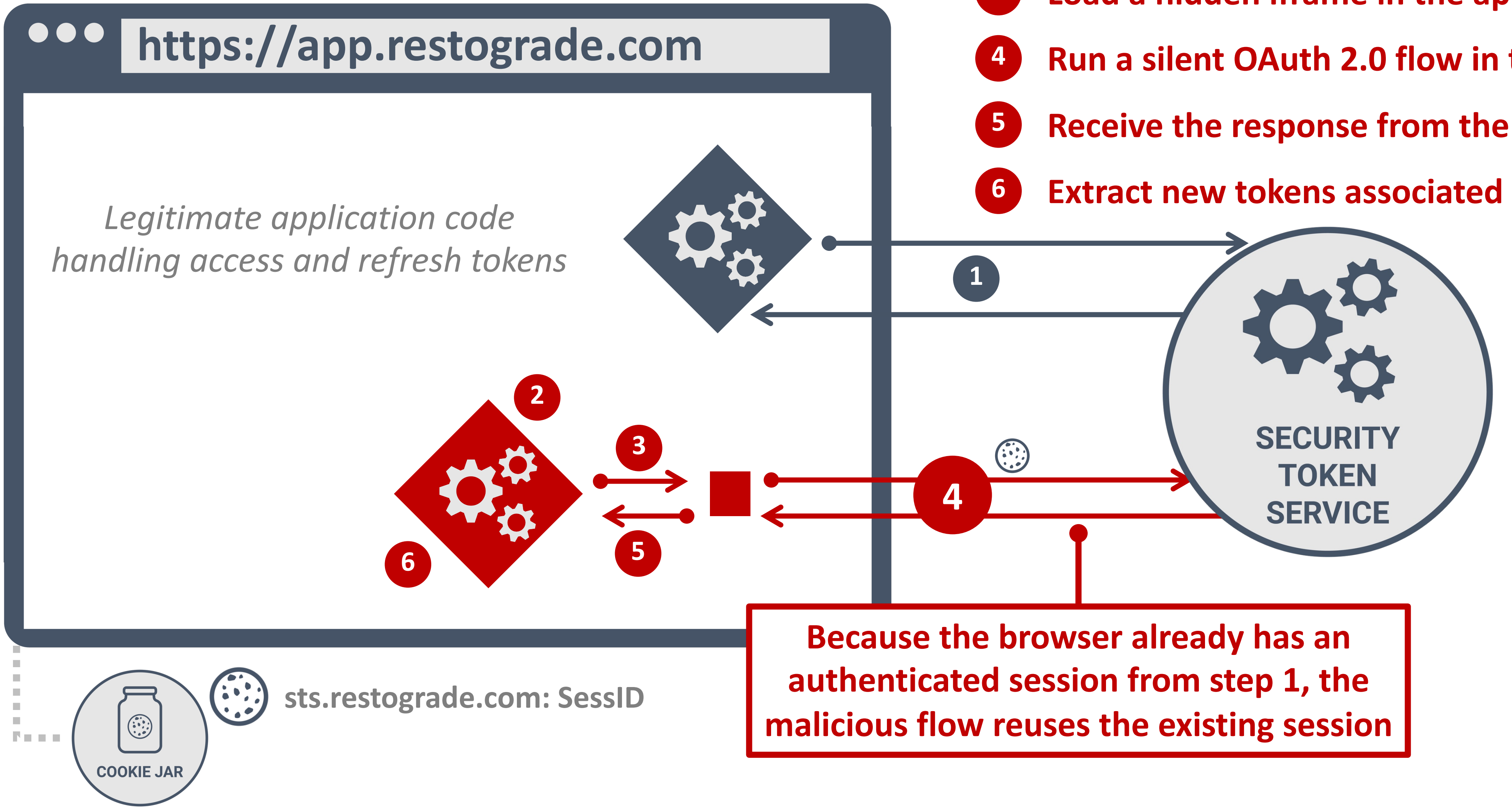
16 April 2020 · OAuth 2.0 & OpenID Connect · OAuth 2.0, LocalStorage, XSS

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;
}
```

STEALING ALL TOKENS WITH THE SILENT RENEW

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



So, are we screwed?

- A Yes
- B No

Yes. XSS is game over!

On the security of tokens in browser-based applications

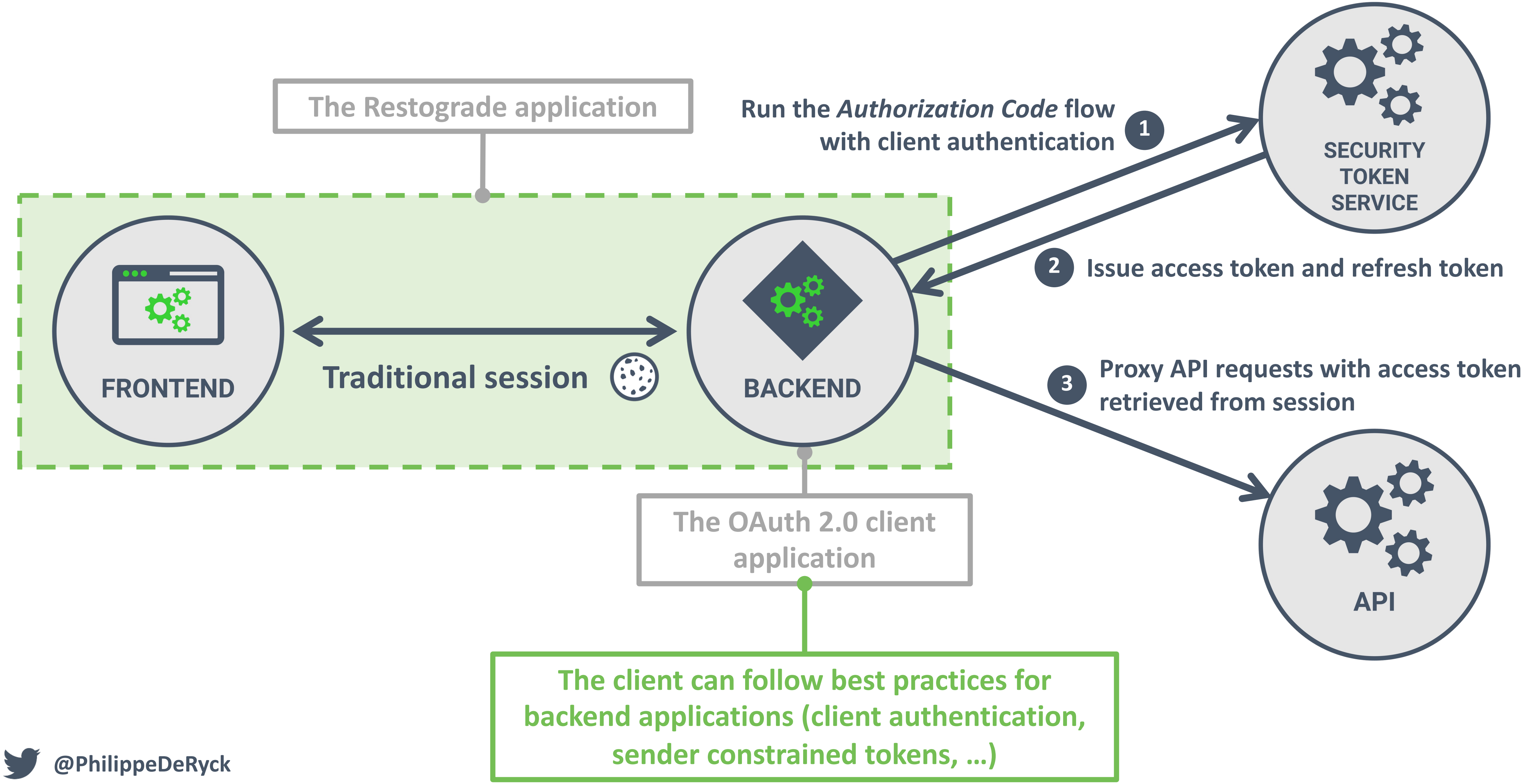
- Refresh token rotation is a great and clean feature
  - Eliminates the need for messy iframe-based patterns
  - In light of malicious JavaScript, **refresh token rotation *is not a security measure***

- An isolation mechanism with workers addresses some scenarios
  - JavaScript needs to "tunnel" all requests through the worker
  - Malicious code would still be able to send requests through the worker
  - Attackers can always request an independent set of tokens

- Restricting access tokens with proof-of-possession does not work in browsers
  - Legitimate JavaScript needs access to the tokens, also exposing them to malicious code
  - Attackers can always request an independent set of tokens using their own secret

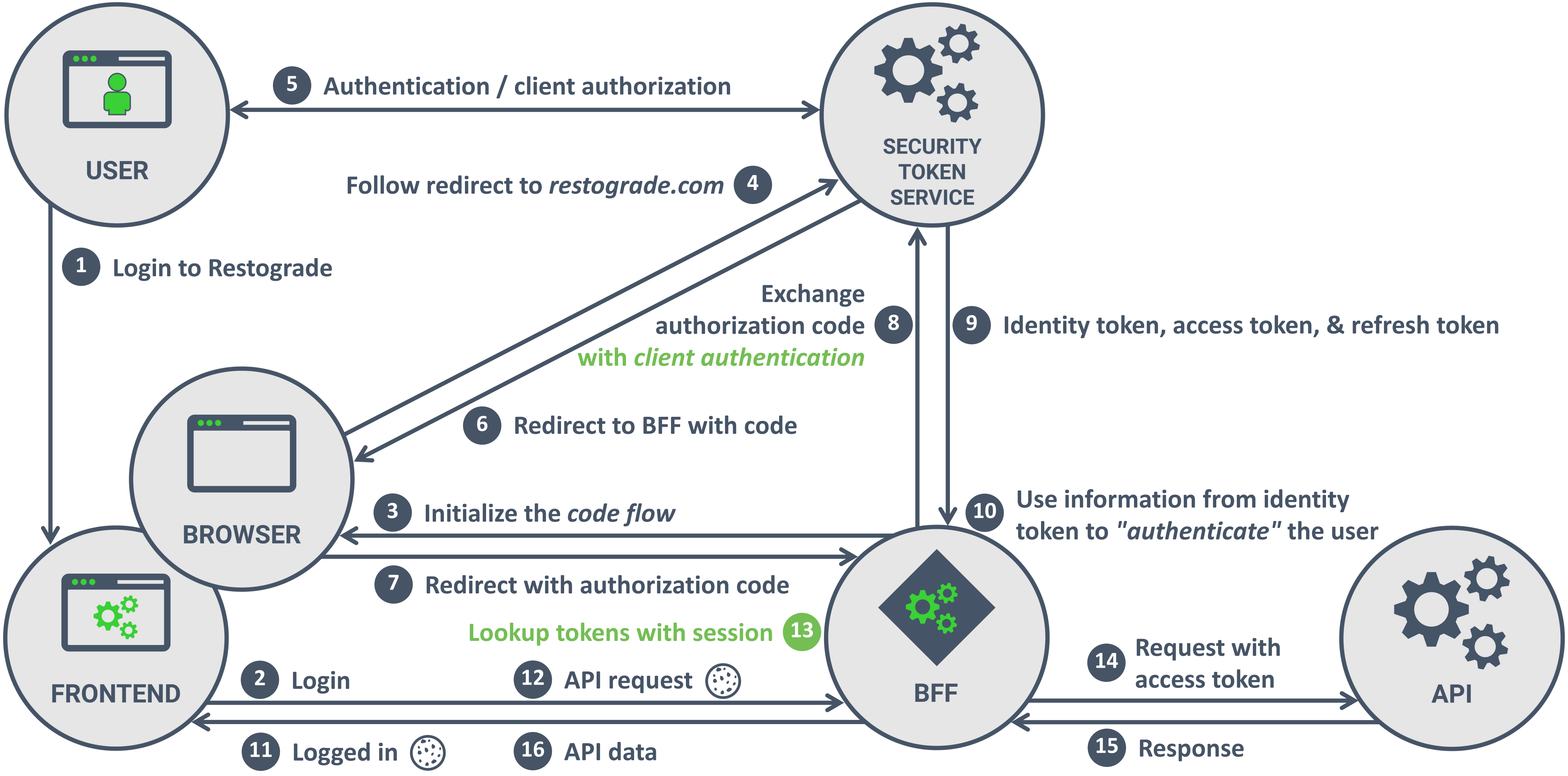
The backend-for-frontend pattern

THE CONCEPT OF A BACKEND-FOR-FRONTEND



@PhilippeDeRyck

THE DETAILS OF A BACKEND-FOR-FRONTEND

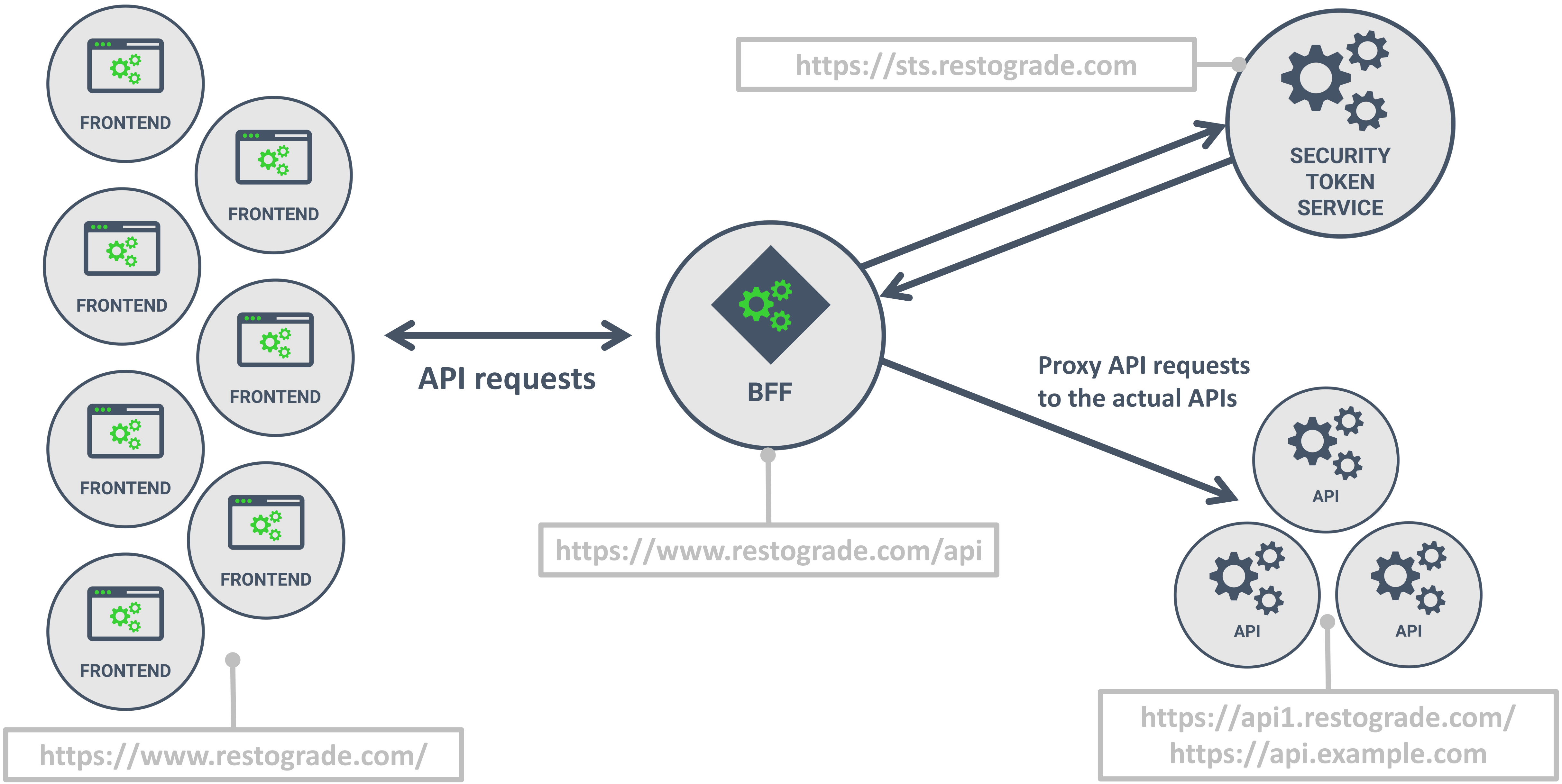


The backend-for-frontend pattern

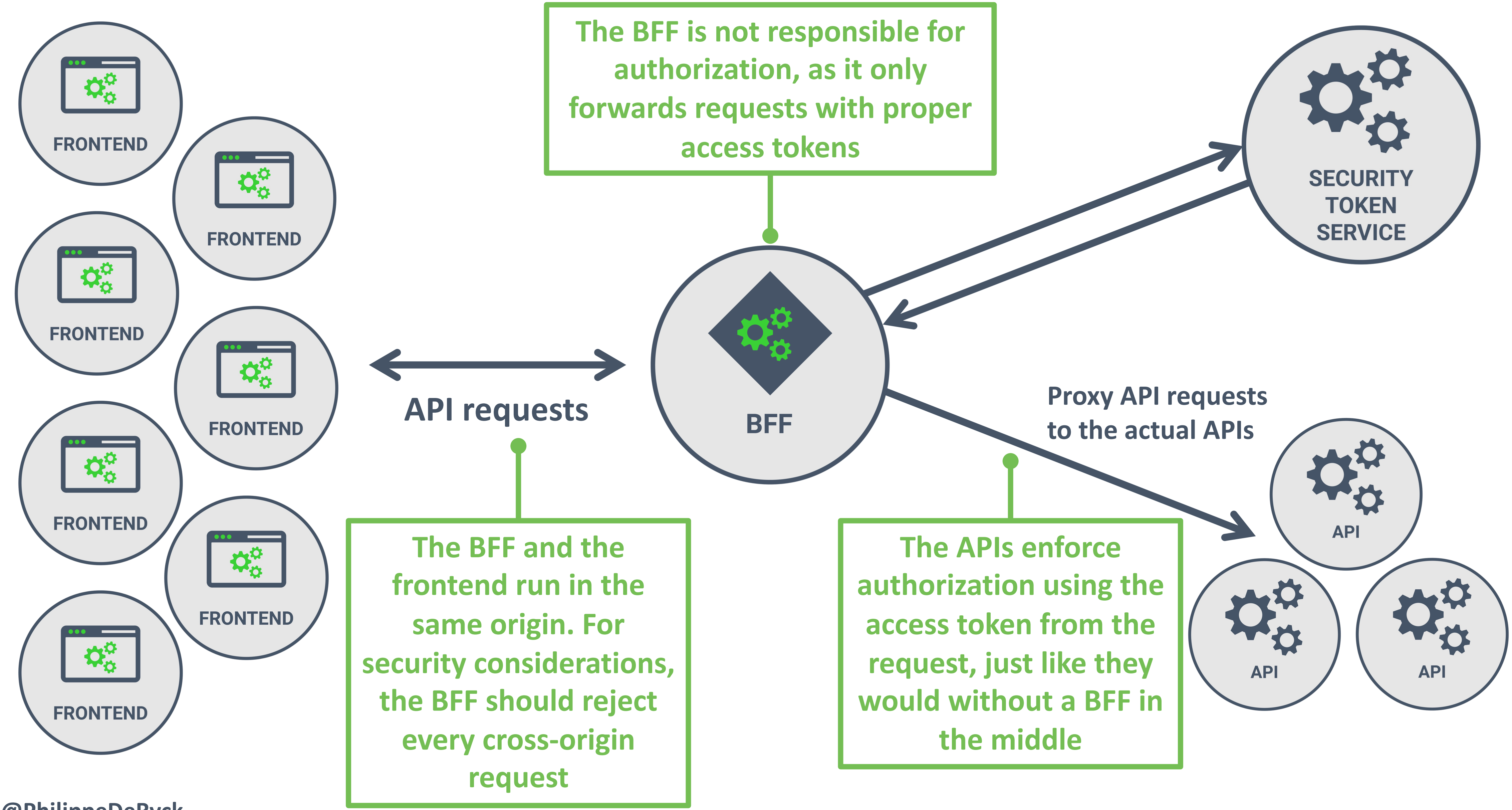
- The frontend uses a dedicated backend-for-frontend (BFF) for API access
  - The BFF mainly forwards calls to the actual APIs
  - The BFF attaches access tokens to outgoing requests to authorize the API calls
- BFFs are already used to aggregate different backend systems in a single API
  - Common pattern to join various microservices into a single frontend-specific API
  - Useful to chain different operations together without pushing that to the client
  - ***From a security perspective, BFFs make a lot of sense***
- The BFF becomes the OAuth 2.0 / OIDC client application
  - The BFF runs on a server, so it acts as a ***confidential client***
  - The BFF can apply all security best practices for backend client applications

BFF IMPLEMENTATION DETAILS

Fitting a BFF into the architecture

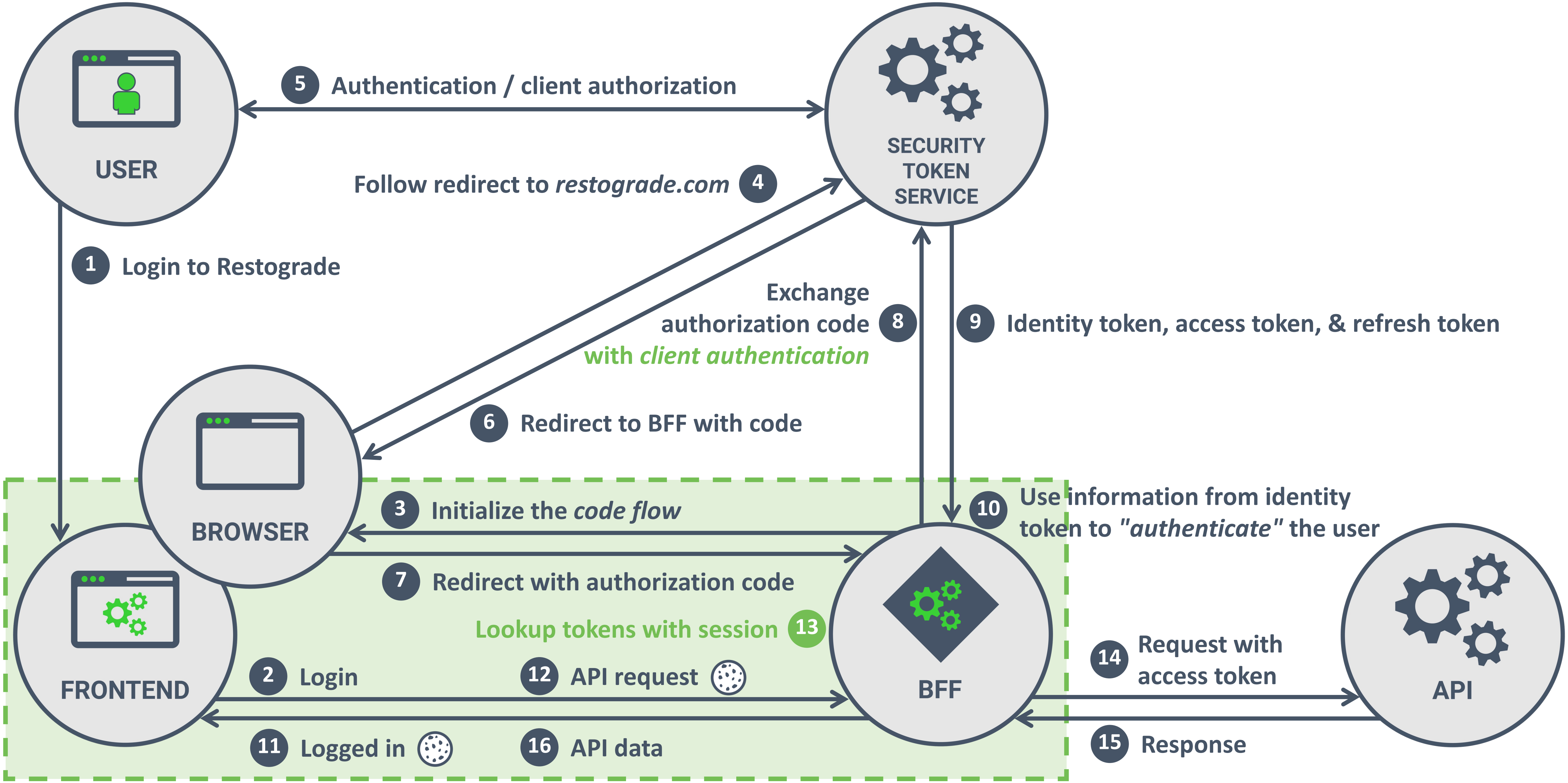


A BFF TRANSLATES SESSIONS INTO ACCESS TOKENS



@PhilippeDeRyck

SESSIONS BETWEEN THE FRONTEND AND THE BFF



Cookie Security Settings

- The BFF uses cookies to manage the session with the frontend
  - Cookies work perfectly when frontend and BFF are deployed in the same domain
  - Browsers handle cookies automatically, so no need to write code in the frontend

Security best practices for setting a cookie

```
Set-Cookie: __Host-session=…; Secure; HttpOnly; SameSite=strict
```

- Modern best practices for cookies require the following settings
  - Enable the ***Secure*** flag to restrict the cookie for HTTPS use only
  - Enable the ***HttpOnly*** flag to prevent JS-based access and memory-level attacks
  - Enable the ***SameSite=strict*** flag to prevent CSRF attacks
  - Add the ***__Host-*** attribute to the name of the cookie to prevent subdomain-based attacks

Rejecting malicious requests

- Attackers can attempt sending *CSRF* requests from the victim's browser
  - A fraudulent request from a different domain to the BFF carrying cookies
  - The ***SameSite*** flag stops most of these scenarios, ***but not from malicious subdomains***
- An API following security best practices is unlikely to suffer from this problem
  - Carefully respect the semantics of HTTP verbs (GET vs POST vs …)
  - Always enforce a proper non-form content type on incoming request bodies
- It is recommended to use CORS as a secondary defense to stop potential attacks
  - Include a request header to force attackers to send requests from JavaScript
    - E.g., include a static ***BFF: rocks*** header
  - The browser will always send a preflight with ***Origin*** header on cross-origin requests to your API
  - Your API's CORS configuration will reject any requests coming from different origins

Is a BFF stateful or stateless?

A BFF CAN BE STATEFUL OR STATELESS

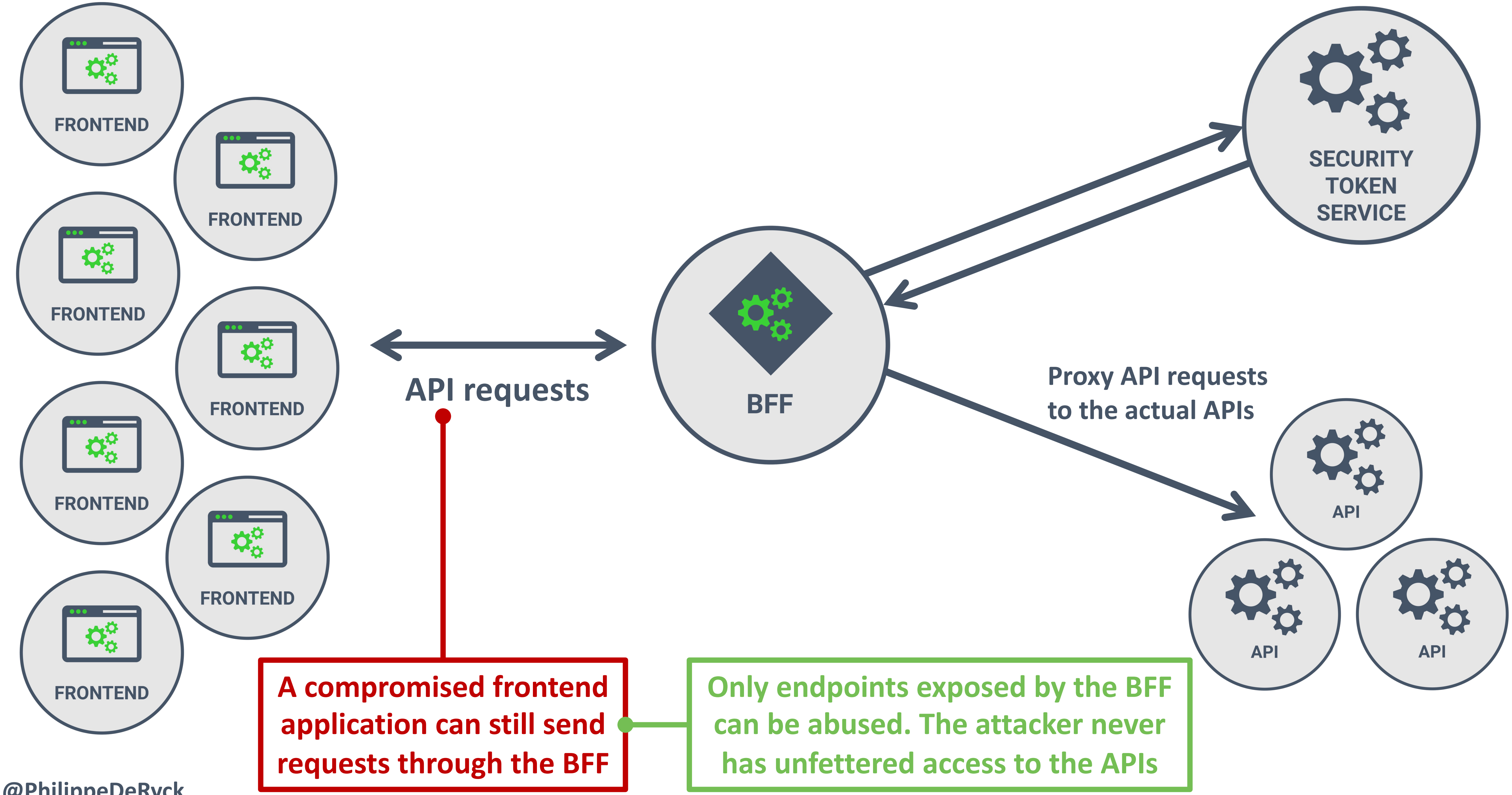
- BFF sessions can be implemented with or without server-side state
  - Server-side state keeps tokens on the server and issues a session ID in a cookie
  - Client-side state puts tokens into a session object and stores the object in a cookie

- Client-side sessions are often not recommended, due to lack of control
  - The session cookie has bearer token properties, so theft leads to abuse
  - Revoking existing state becomes difficult without server-side control over the session
  - In a BFF scenario, ***revocation is available through the OAuth 2.0 refresh tokens***

- Client-side sessions in a BFF have strict security requirements
  - Integrity protection of the data is crucial to avoid attacks
  - Confidentiality (i.e., encryption) is not mandatory, but strongly recommended

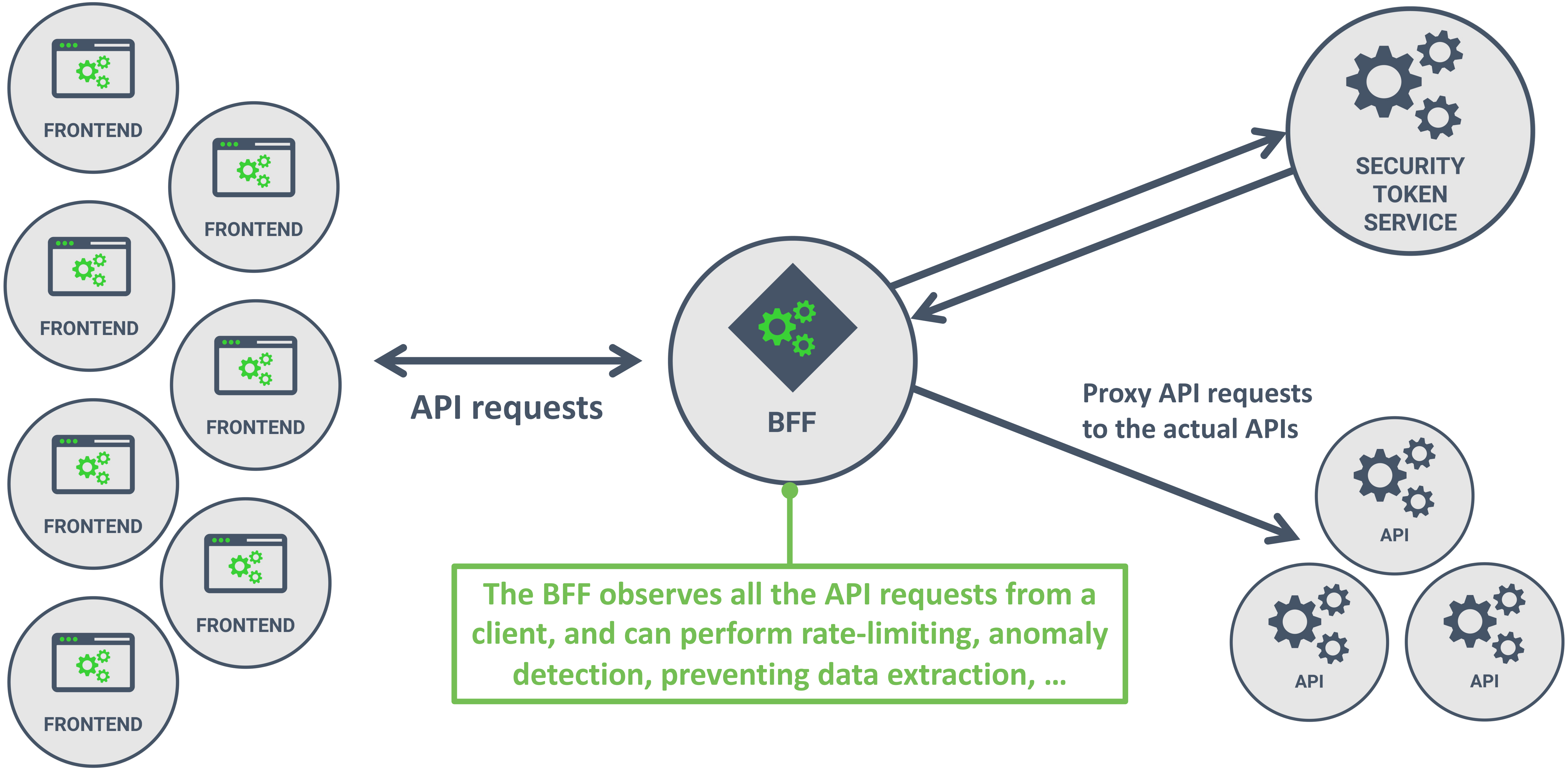
That’s great Philippe, but what about XSS?

A BFF CANNOT STOP XSS ATTACKS EITHER



@PhilippeDeRyck

A BFF CAN ACT AS AN ADDITIONAL LAYER OF DEFENSE



@PhilippeDeRyck

BFFs rely on core building blocks of the web

- Same-origin requests between a frontend and a backend are straightforward
  - Browsers do not restrict same-origin requests
  - The BFF can reject all cross-origin requests to avoid Cross-Site Request Forgery attacks
- Cookies work well within the same origin, even with privacy-sensitive browsers
  - The cookie is essential for the BFF to track the user's state (which contains access tokens)
  - Cookies will always be present on same-origin requests, regardless of how they are sent
- BFFs see all requests from the frontend and can detect malicious behavior
  - A BFF can correlate requests to a single frontend, enabling additional checks
  - Examples include rate limiting, anomaly detection, preventing data extraction, …

BFF Security Framework :: Duende

docs.duendesoftware.com/identityserver/v5/bff/

Duende.

Search...

- Overview
- Fundamentals
- Quickstarts
- User Interaction
- Requesting Tokens
- Protecting APIs
- Data Stores and Persistence
- Diagnostics

Home > BFF Security Framework

Edit this page

BFF Security Framework

The Duende.BFF (Backend for Frontend) security framework packages up guidance and the necessary components to secure browser-based frontends (e.g. SPAs or Blazor WASM applications) with ASP.NET Core backends.

Duende.BFF is part of the IdentityServer Business Edition or higher. The same license and

https://docs.duendesoftware.com/identityserver/v5/bff/

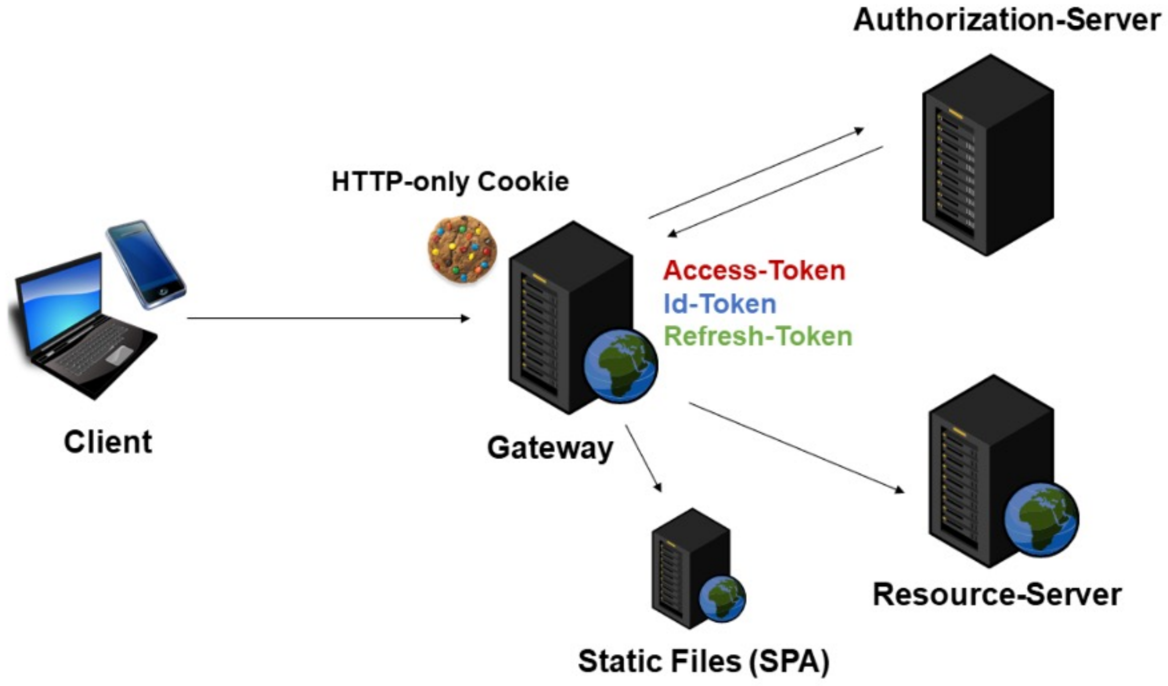
GitHub - manfredsteyer/yarp-a

github.com/manfredsteyer/yarp-auth-proxy

readme.md

Proof of Concept for an Auth Gateway for SPA

... aka Auth Reverse Proxy ... aka Backend for Frontend (BFF) ... aka Forward Authentication Service ...

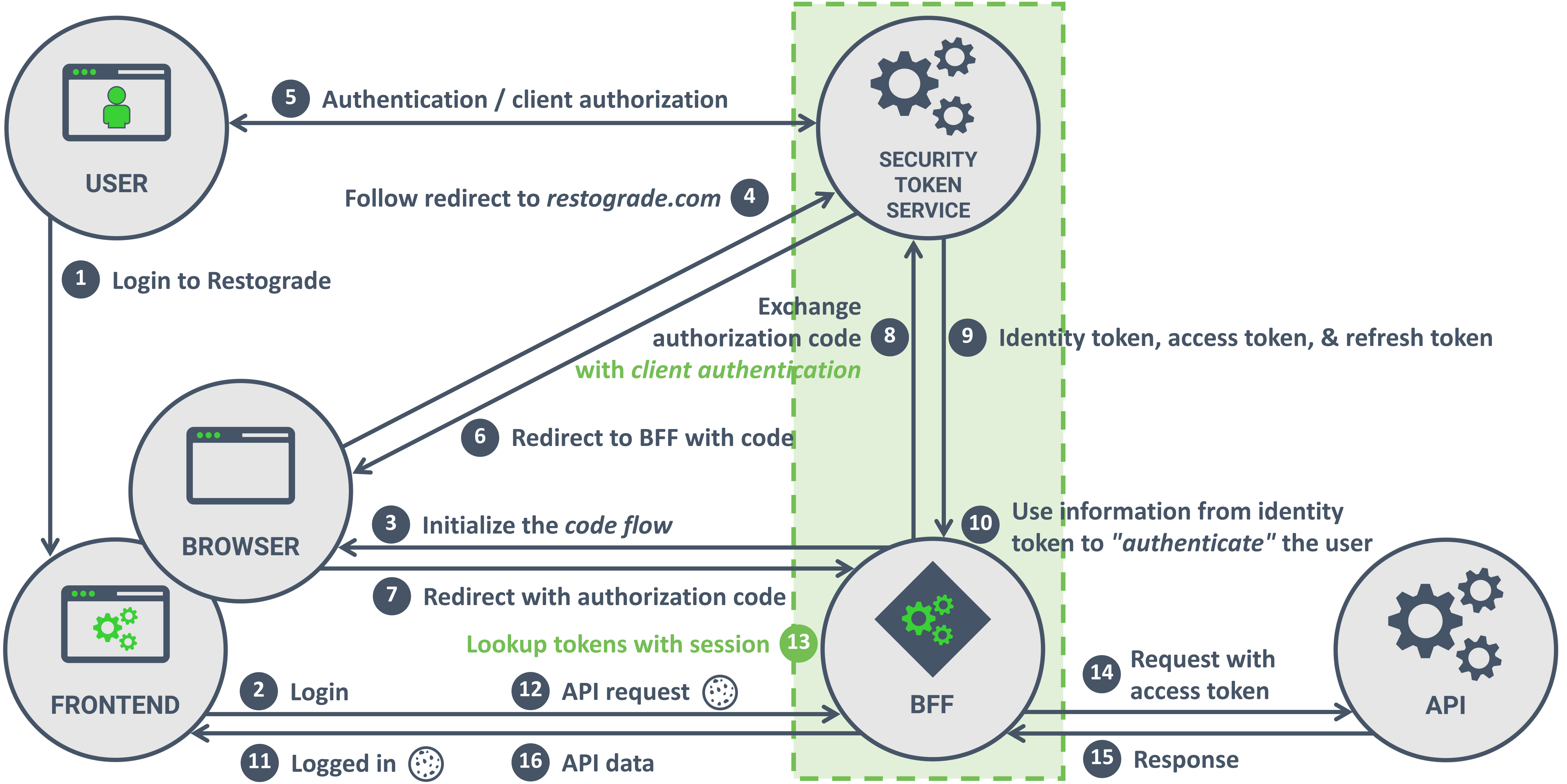


> This gateway shifts the use of security standards such as OAuth2 and OpenId Connect to the server side. This drastically simplifies the implementation of the SPA and makes your solution more secure.

https://github.com/manfredsteyer/yarp-auth-proxy

OAuth 2.0 security benefits of a BFF

CLIENT AUTHENTICATION WITH A BFF



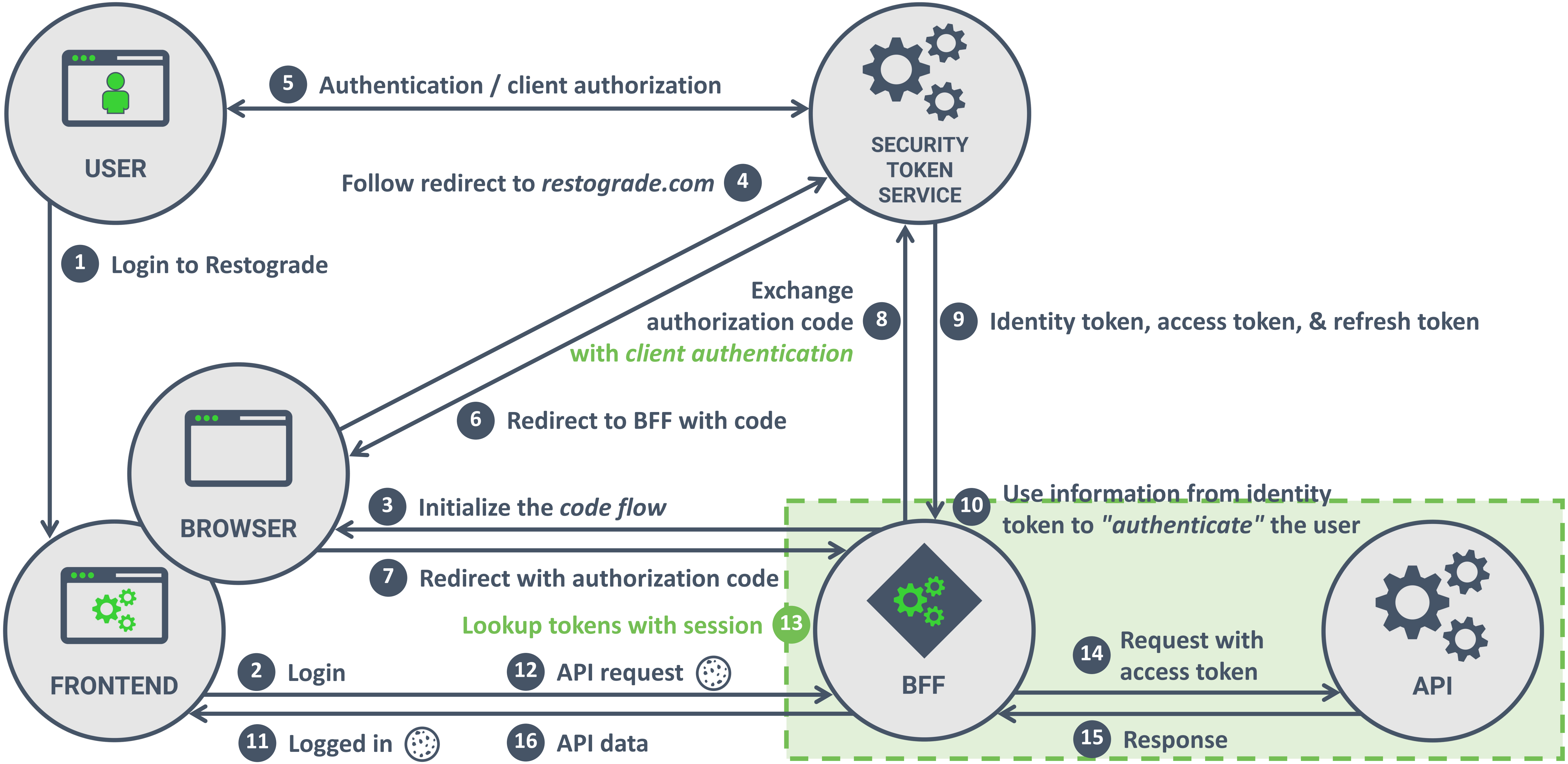
Client authentication as the BFF

- The BFF is the client application authenticating to the STS
  - Typical client authentication involves using a string-based shared secret
  - Backend clients can use more advanced key-based authentication mechanisms
    - Instead of a shared secret, the client has the private key and the STS has the public key
    - The secret is only exposed to one party, reducing the attack surface
- The specifications define two key-based options
  - ***mTLS*** uses client-side TLS certificates to implement authentication (RFC 8705)
  - ***JWT bearer tokens*** illustrate possession of a key to implement authentication (RFC 7523)
- Client authentication mitigates two potential attacks against the frontend
  - Stolen authorization codes become useless without client credentials or the PKCE verifier
  - Stolen refresh tokens cannot be used without authenticating as the BFF client

Refresh tokens in a BFF

- Confidential clients need to authenticate when using refresh tokens
  - Even though a BFF's refresh tokens are not bearer tokens, they should be well-protected
  - Key-based client authentication with proper key management is an important defense
  - Refresh token rotation can be used to ensure the continuous renewal of refresh tokens
- The BFF has a session with the frontend and associates tokens with the session
  - Sessions often have a limited lifetime and timeout after inactivity
  - When a session expires, the BFF can no longer use the associated tokens
  - There is no need for refresh tokens to have longer lifetimes or less strict properties
- The secure use of refresh tokens allows for shorter lifetimes for access tokens
  - With a short lifetime, the window of abuse for a stolen access token can be reduced

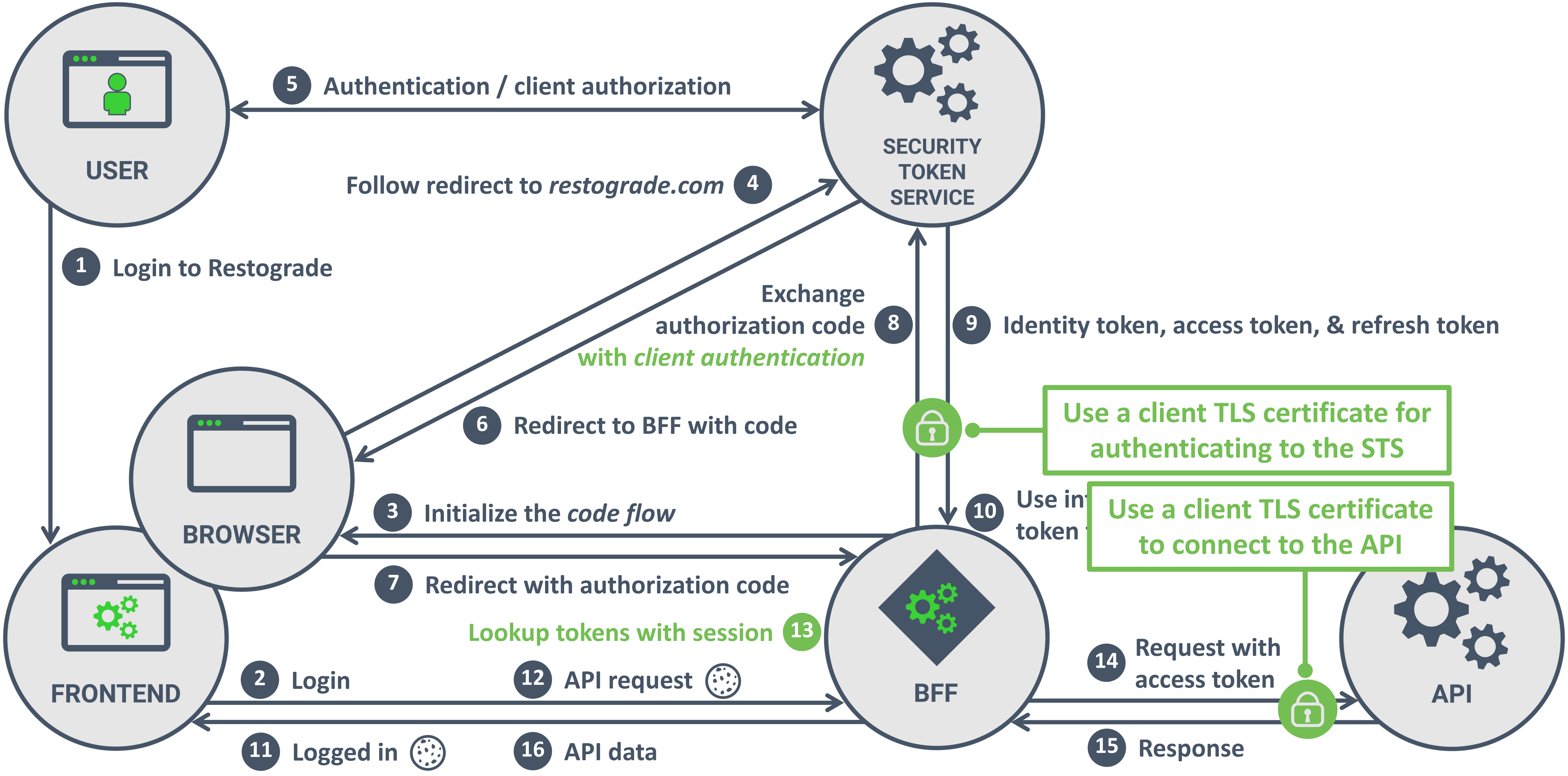
SENDER CONSTRAINED TOKENS WITH A BFF



Sender constrained tokens with a BFF

- Whenever possible, access tokens should not be bearer tokens
  - A BFF has enough control over its environment to use sender constrained access tokens
  - Such tokens can only be used when the holder illustrates possession of a secret
- Multiple proposals attempted to solve the bearer token problem
  - Various specs in the OAuth 2.0 world discuss *proof-of-possession* mechanisms
  - Browsers played around with offering token binding using TLS channels
- Currently, two important mechanisms for sender constrained tokens exist
  - Tokens can be tied to a TLS certificate, which requires the client to use ***mTLS***
    - mTLS is well-understood and supported by many STS products
  - ***Demonstration of Proof-of-Possession*** offers a similar application-layer mechanism
    - dPoP is still experimental and not widely supported

SENDER CONSTRAINED TOKENS WITH A BFF AND mTLS



SENDER CONSTRAINED TOKENS WITH A BFF AND MTLS

A JWT access token with an embedded certificate fingerprint

```
{
  "sub": "b6rdGPsO2iBKB7sO2i",
  "aud": "https://api.example.com",
  "azp": "lY5g0BKB7Mow4yDlb6rdGPsO2i1g7Osv",
  "iss": "https://sts.restograde.com/",
  "exp": 1419356238,
  "iat": 1419350238,
  "scope": "read write",
  "cnf": {
    "x5t#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05O89jdN-dg2"
  }
}
```

The fingerprint of the cert

@PhilippeDeRyck

Sender constrained tokens with a BFF and mTLS

- The *cnf* claim contains information about the proof-of-possession key
  - JWT access tokens directly embed the *cnf* claim in the token
  - For reference access tokens, the STS provides the *cnf* claim during introspection
- The only responsibility for a client is using mTLS with a client certificate
  - An STS that supports sender constrained access tokens will use the certificate fingerprint
    - The hash in the ***x5t#S256*** value uniquely identifies the certificate and its public key
  - An API enforcing proof-of-possession will look for the *cnf* claim can verify the fingerprint
    - If the connection is setup with the right certificate, the client must possess the private key
- Sender constrained access tokens are much harder to abuse
  - An attacker would need to completely compromise a client to abuse access tokens

SECURITY BEST PRACTICES FOR BFFS

- Use a cryptographic client authentication mechanism for the BFF
  - mTLS is well supported by server-side frameworks and STS implementations
- The use of mTLS enables using sender constrained access tokens
  - Ensure that the APIs support the use of mTLS and sender constrained access tokens
  - Make this proof-of-possession mechanism mandatory for all BFFs
- Keep token lifetimes as short as possible
  - Access tokens should be short-lived, since refresh tokens are available to a BFF
  - Refresh tokens should not extend the expected lifetime of a session
- Cookie-based sessions should apply the latest cookie security settings
  - Use the ***__Host-*** prefix and the ***Secure***, ***HttpOnly***, and ***SameSite*** cookie flags
  - Ensure confidentiality and integrity of client-side session data

Is a BFF the right choice?

Using a BFF is a trade-off

Benefits	Drawbacks
No tokens in the browser	Every frontend needs a BFF
The frontend becomes easier	More development and maintenance effort
Less issues with third-party cookie blocking	Non-believers hate BFFs
OAuth 2.0 flows with a confidential client	
Ability to use sender constrained tokens	
Minimal server-side attack surface	

Sensitive Single Page Applications should definitely consider using a BFF

Summary

Securing OAuth 2.0 in SPAs

BFFs for sensitive Single Page Applications

- Single Page Applications struggle with OAuth 2.0 security best practices
  - Public clients cannot authenticate to the STS
  - Tokens cannot be adequately protected against theft
  - Proof-of-possession mechanisms are more difficult to implement in the browser

- A Backend-For-Frontend moves OAuth 2.0 aspects into a backend
  - The BFF is the OAuth 2.0 client application, following backend security best practices
  - The frontend maintains a session with the BFF, which keeps tokens in a session
  - The BFF forwards requests to the API and attaches user-specific tokens

- A BFF has numerous security benefits for sensitive frontend web applications
  - OAuth 2.0 requires client authentication and tokens are not available in the browser
  - Additional defenses can be deployed at the BFF level

Thank you!

Connect on social media for more in-depth security content

@PhilippeDeRyck

/in/PhilippeDeRyck