# AppSec is too hard!?

Dr. Philippe De Ryck

https://PragmaticWebSecurity.com

# The road to AppSec hell is paved with good intentions

# What do we expect from developers to build secure applications?

# Follow secure coding guidelines!

# I am *Dr. Philippe De Ryck*

Pragmatic Web Security
Security for developers

Founder of Pragmatic Web Security

Google Developers Experts

Google Developer Expert

AMBASSADOR PROGRAM

Auth0 Ambassador

SecAppDev

SecAppDev organizer

# I help developers with security

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

https://pdr.online

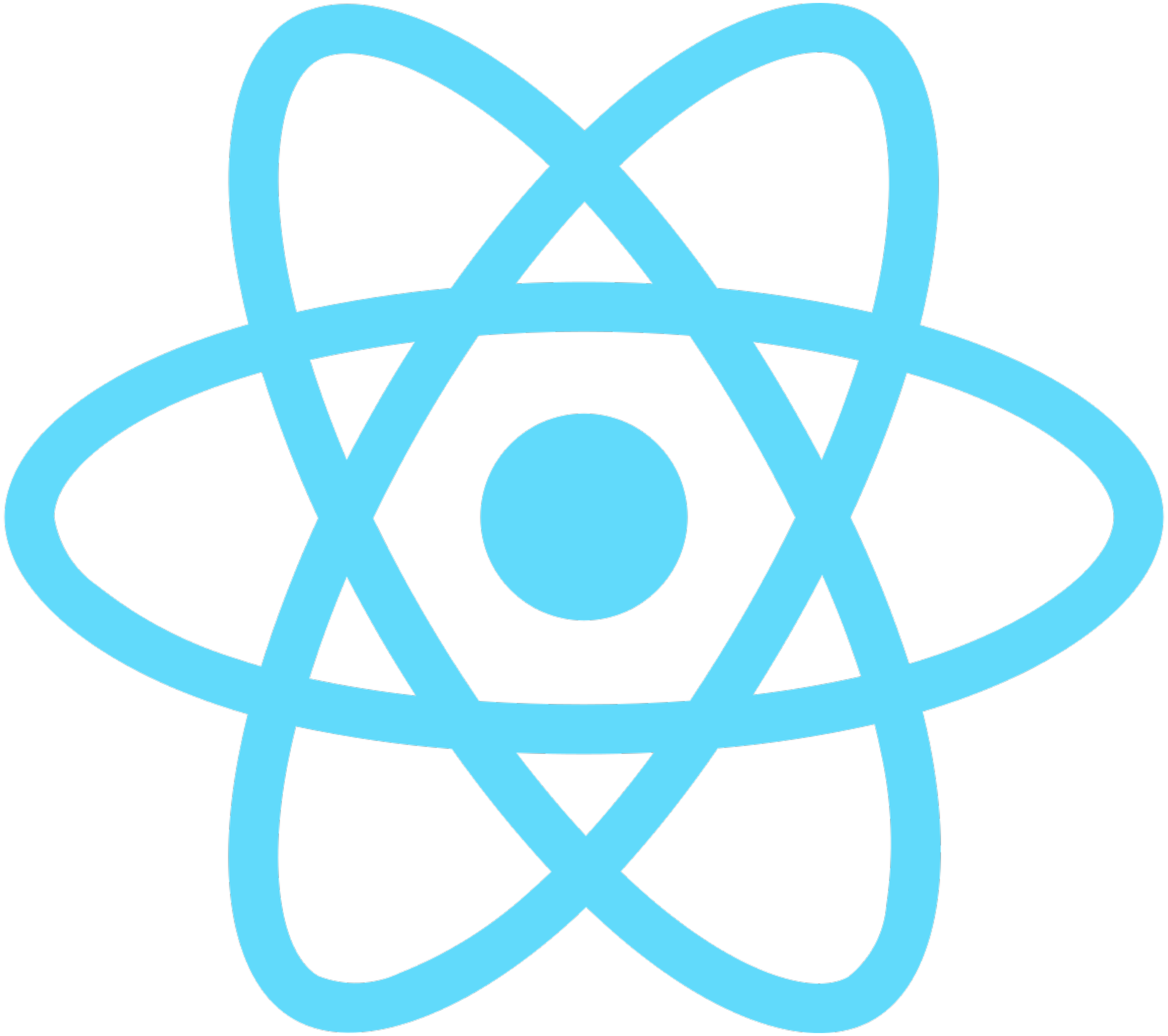
# GRAB A COPY OF THE SLIDES ...

https://pragmaticwebsecurity.com/talks

/in/PhilippeDeRyck

https://infosec.exchange/@PhilippeDeRyck

*Website icons created by Uniconlabs - Flaticon*



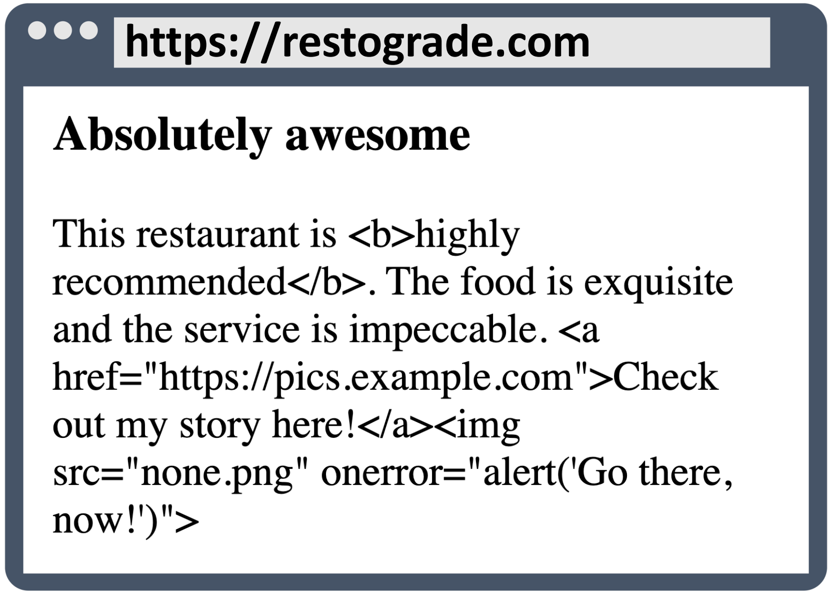
*A JSX template to combine data with HTML*

```
return ( <div>
  <h3>{ title }</h3>
  <p>{ review }</p>
</div>);
```

By default, React escapes values embedded in JSX before rendering them

*A review submitted by a malicious user*

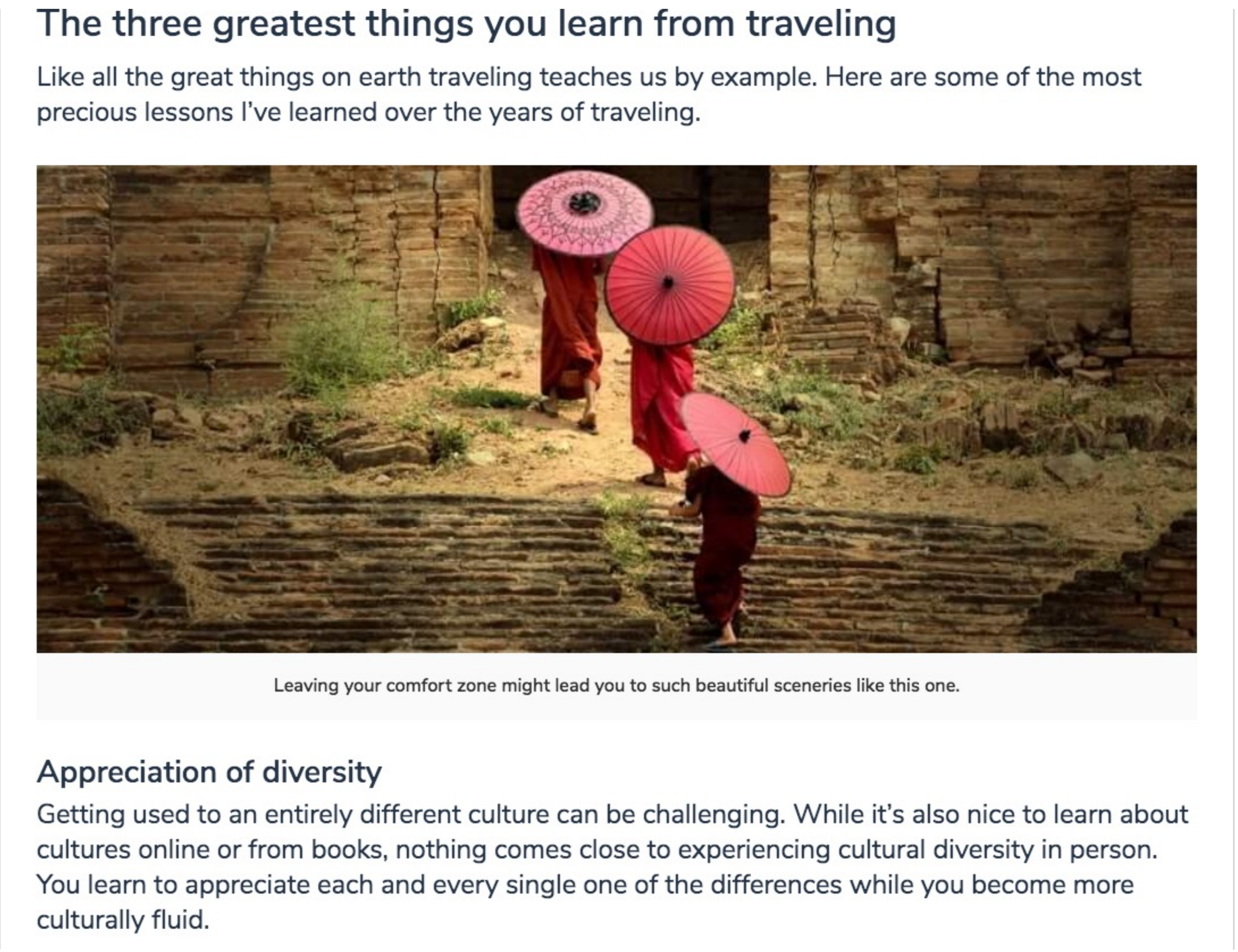
```
This restaurant is <b>highly recommended</b>. The
food is exquisite and the service is impeccable. <a
href="https://pics.example.com">Check out my story
here!</a><img src="none.png" onerror="alert('Go
there, now!')">
```



https://restograde.com

**Absolutely awesome**

This restaurant is <b>highly recommended</b>. The food is exquisite and the service is impeccable. <a href="https://pics.example.com">Check out my story here!</a><img src="none.png" onerror="alert('Go there, now!')">



## The three greatest things you learn from traveling

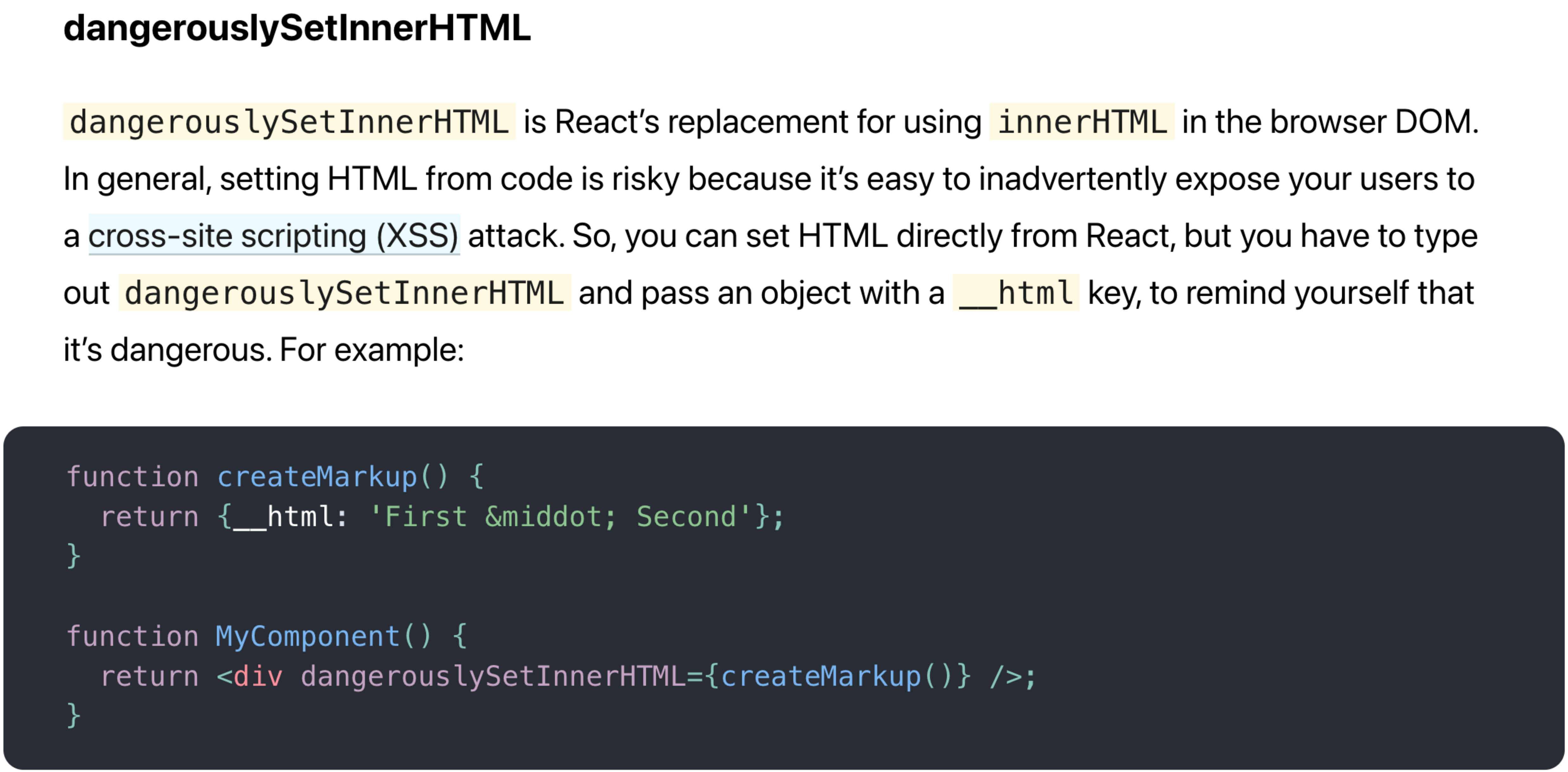
Like all the great things on earth traveling teaches us by example. Here are some of the most precious lessons I've learned over the years of traveling.

Leaving your comfort zone might lead you to such beautiful sceneries like this one.

### Appreciation of diversity

Getting used to an entirely different culture can be challenging. While it's also nice to learn about cultures online or from books, nothing comes close to experiencing cultural diversity in person. You learn to appreciate each and every single one of the differences while you become more culturally fluid.

# dangerouslySetInnerHTML



## dangerouslySetInnerHTML

`dangerouslySetInnerHTML` is React’s replacement for using `innerHTML` in the browser DOM. In general, setting HTML from code is risky because it’s easy to inadvertently expose your users to a cross-site scripting (XSS) attack. So, you can set HTML directly from React, but you have to type out `dangerouslySetInnerHTML` and pass an object with a `__html` key, to remind yourself that it’s dangerous. For example:

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

*https://reactjs.org/docs/dom-elements.html*

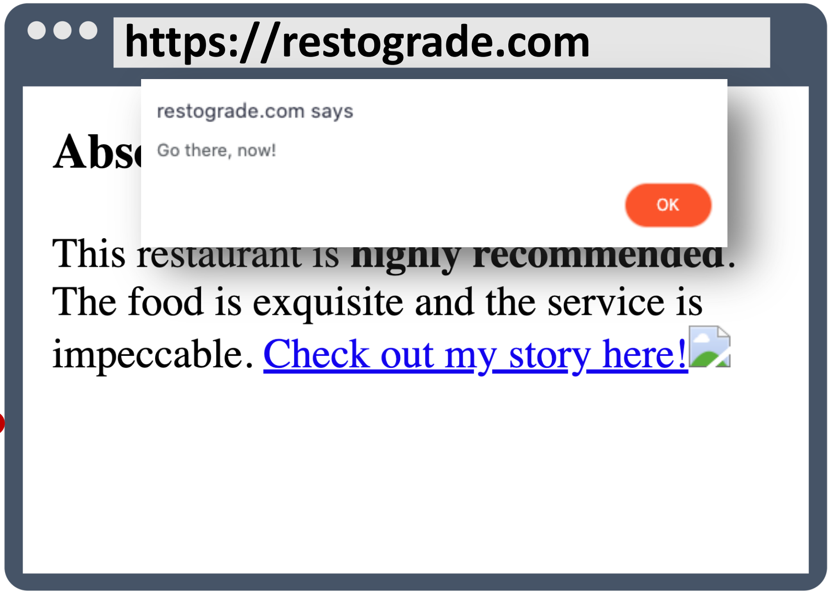
*A JSX template to render user-provided HTML*

```
return ( <div>
<h3>{ title }</h3>
<p dangerouslySetInnerHTML={{__html: review}}></p>
</div>);
```

***dangerouslySetInnerHTML* exposes the *innerHTML* property**

*A review submitted by a malicious user*

```
This restaurant is <b>highly recommended</b>. The
food is exquisite and the service is impeccable. <a
href="https://pics.example.com">Check out my story
here!</a><img src="none.png" onerror="alert('Go
there, now!')">
```



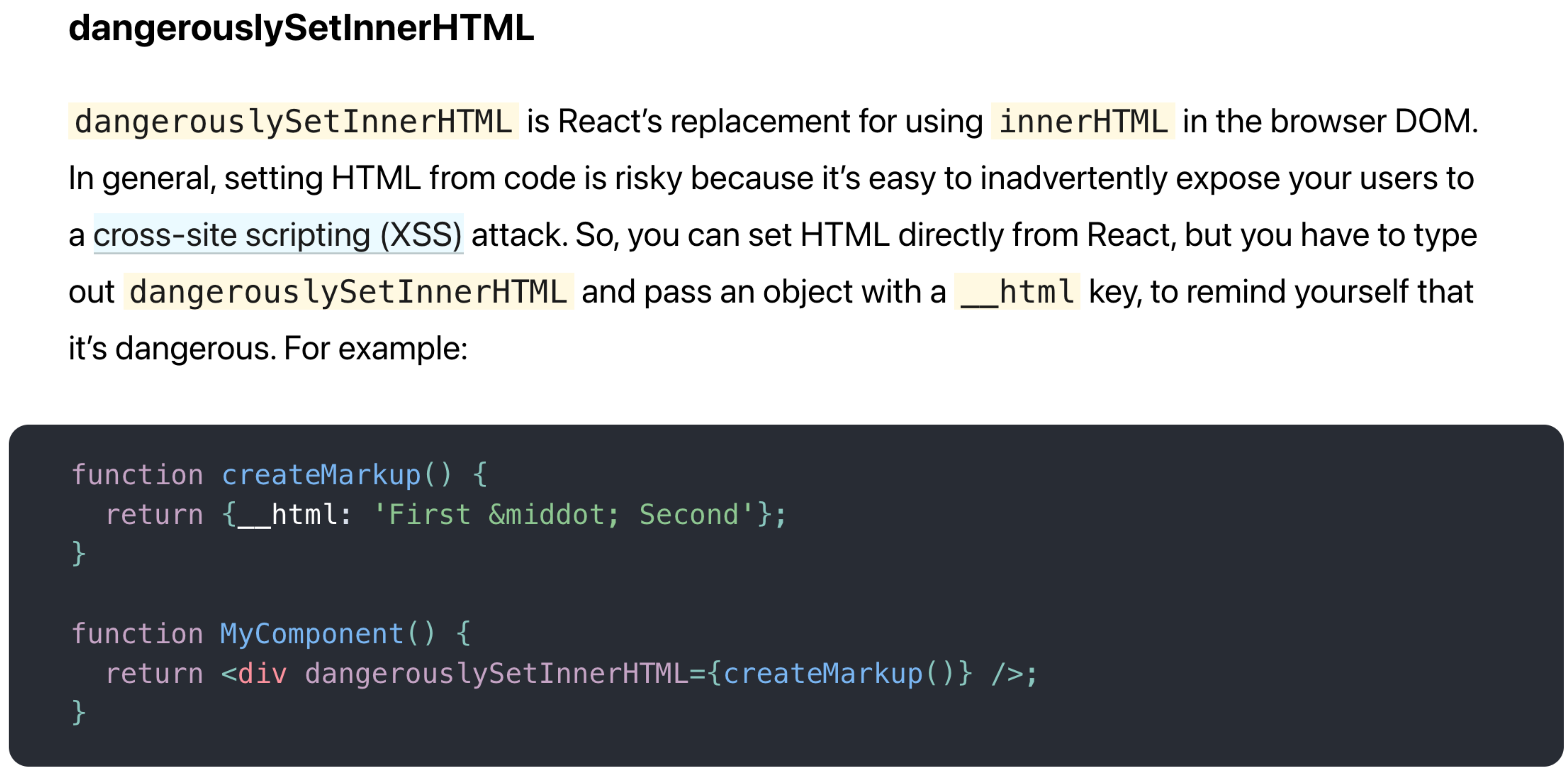
restograde.com says

Go there, now!

Abs

This restaurant is **highly recommended**. The food is exquisite and the service is impeccable. Check out my story here!

**This property is dangerous, since React does not apply any protection at all**



## dangerouslySetInnerHTML

`dangerouslySetInnerHTML` is React's replacement for using `innerHTML` in the browser DOM. In general, setting HTML from code is risky because it's easy to inadvertently expose your users to a cross-site scripting (XSS) attack. So, you can set HTML directly from React, but you have to type out `dangerouslySetInnerHTML` and pass an object with a `__html` key, to remind yourself that it's dangerous. For example:

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

*https://reactjs.org/docs/dom-elements.html*

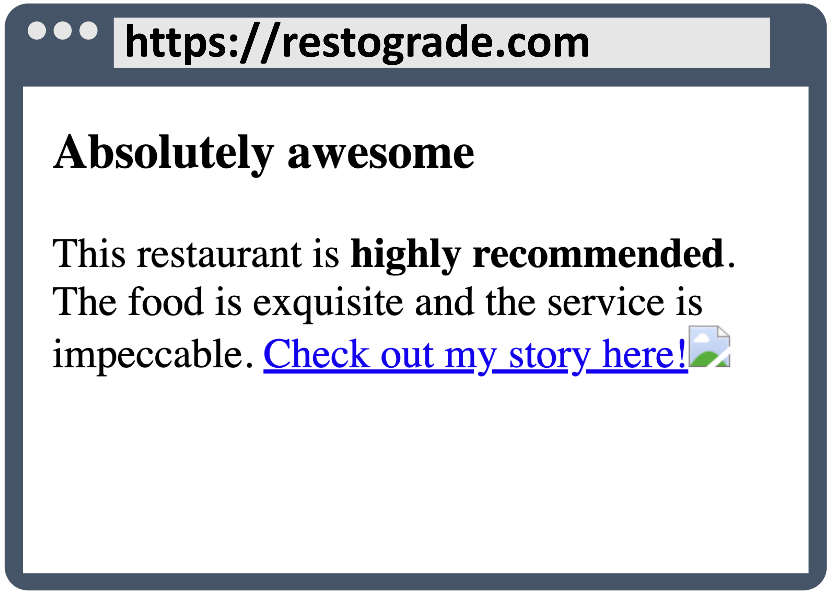
*A JSX template to render user-provided HTML*

```
import DOMPurify from 'dompurify';

return ( <div>
  <h3>{ title }</h3>
  <p dangerouslySetInnerHTML={{__html: DOMPurify.sanitize(review)}}></p>
</div>);
```

**DOMPurify turns untrusted HTML in safe HTML, making it safe to include in the page**

*A review submitted by a malicious user*

```
This restaurant is <b>highly recommended</b>. The
food is exquisite and the service is impeccable. <a
href="https://pics.example.com">Check out my story
here!</a><img src="none.png" onerror="alert('Go
there, now!')">
```



https://restograde.com

**Absolutely awesome**

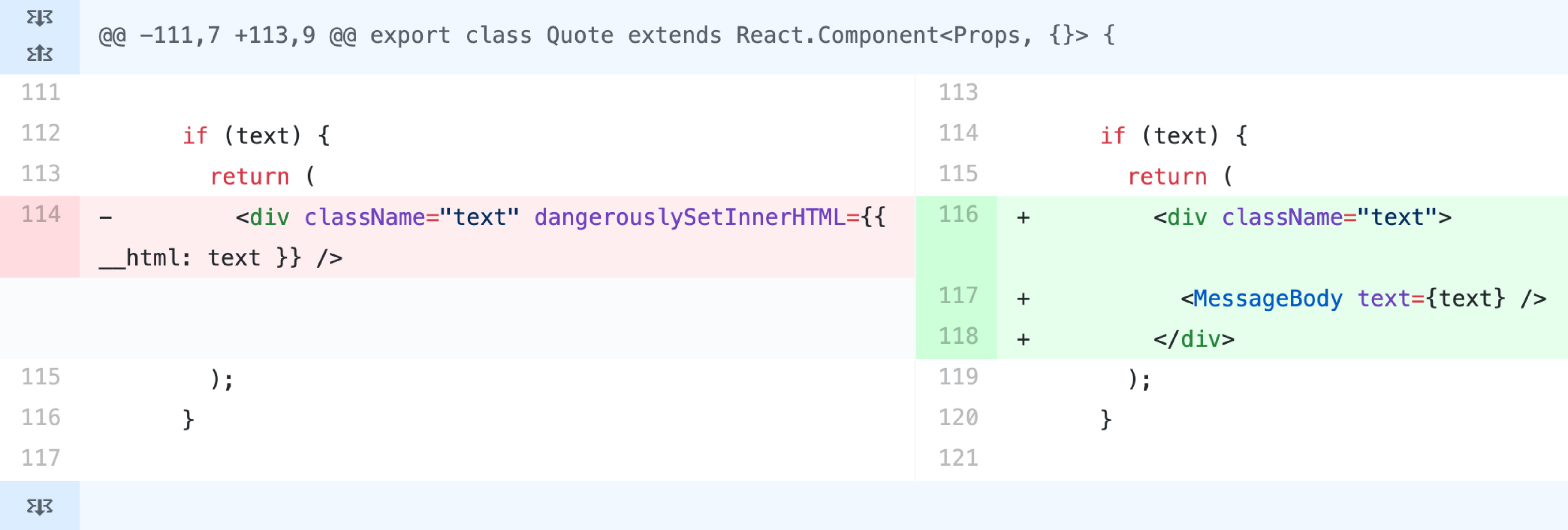
This restaurant is **highly recommended**. The food is exquisite and the service is impeccable. Check out my story here!

# Avoid accidental misuse of dangerous features

***Explicitly mark dangerous application features as dangerous to raise developer awareness***

*Photo by Stephen Andrews on Unsplash*

# Signal Messenger



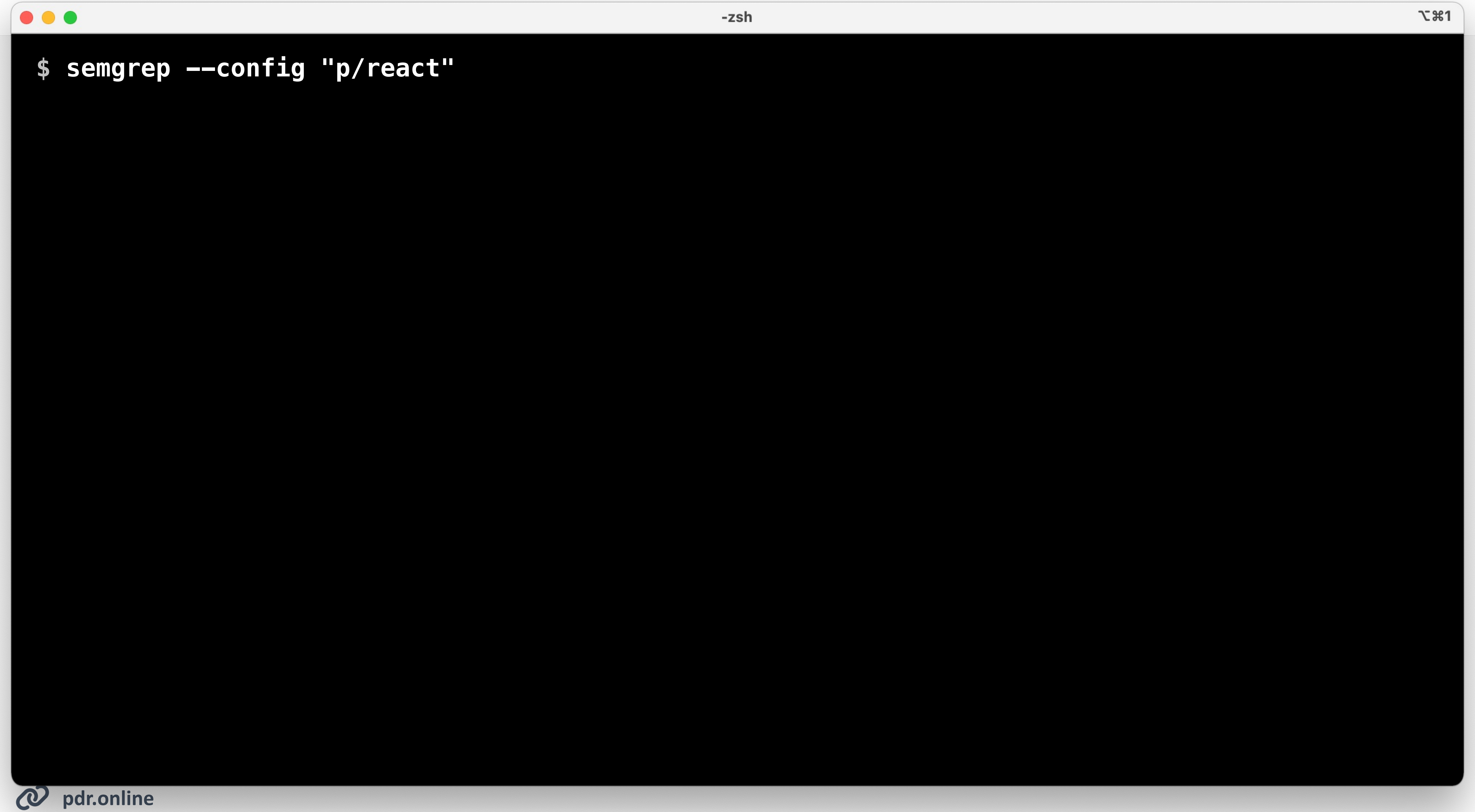
```diff
@@ -111,7 +113,9 @@ export class Quote extends React.Component<Props, {}> {

    if (text) {
      return (
-       <div className="text" dangerouslySetInnerHTML={{ __html: text }} />
+       <div className="text">
+         <MessageBody text={text} />
+       </div>
      );
    }

```

*https://github.com/signalapp/Signal-Desktop/commit/4e5c8965ff72576a9e20850dd30d9985f4073192#diff-8bba204372da85d8cceed81278b7eec*

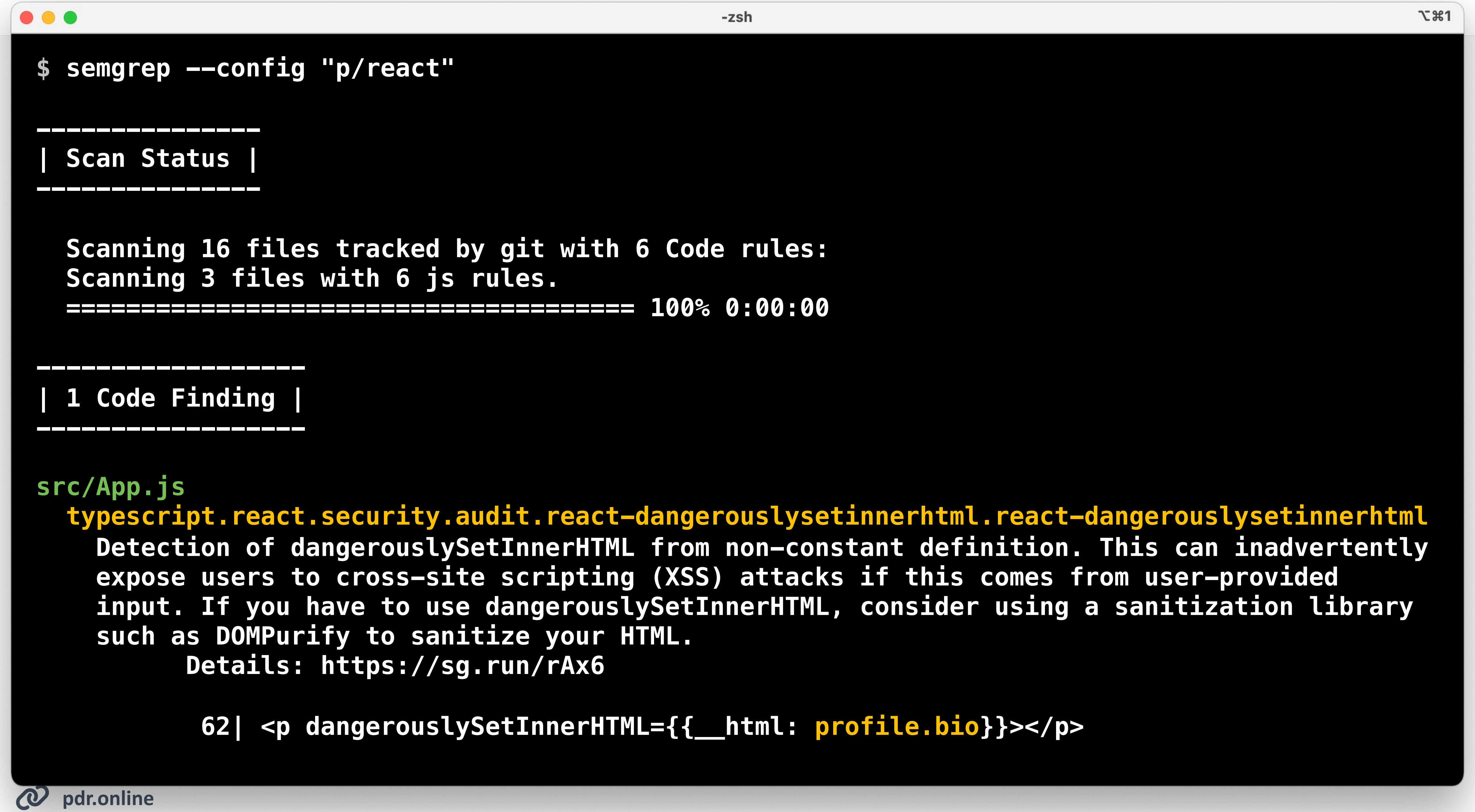
# Marking things as dangerous is not enough

***Explicitly marking dangerous features prevents accidental mis-use, but does not magically enable developers to use the feature securely***

## How can we check if our application is using *dangerouslySetInnerHTML* securely?



```
$ semgrep --config "p/react"
```



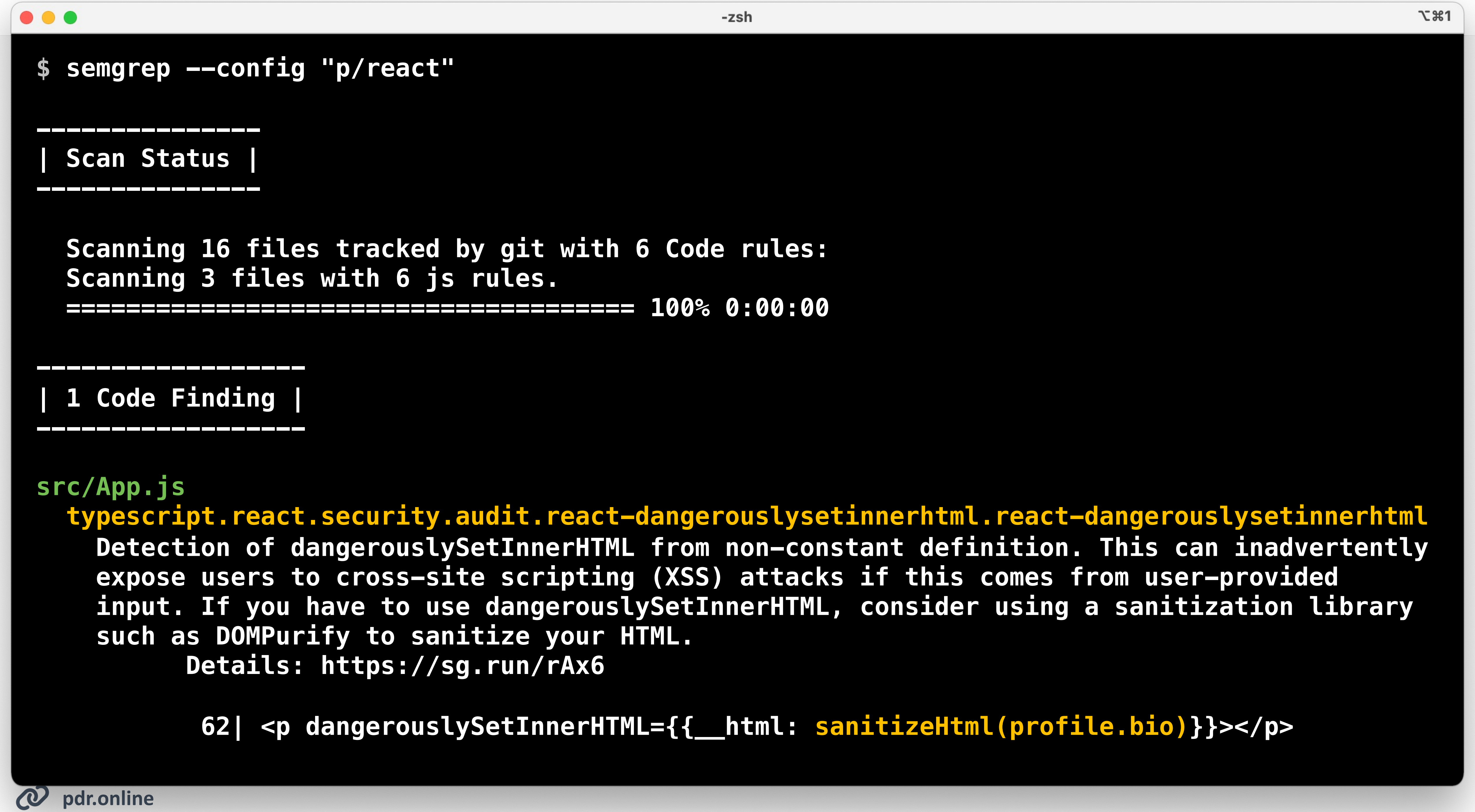
```
$ semgrep --config "p/react"

---------------
| Scan Status |
---------------

  Scanning 16 files tracked by git with 6 Code rules:
  Scanning 3 files with 6 js rules.
  ====================================== 100% 0:00:00

------------------
| 1 Code Finding |
------------------

src/App.js
  typescript.react.security.audit.react-dangerouslysetinnerhtml.react-dangerouslysetinnerhtml
    Detection of dangerouslySetInnerHTML from non-constant definition. This can inadvertently
    expose users to cross-site scripting (XSS) attacks if this comes from user-provided
    input. If you have to use dangerouslySetInnerHTML, consider using a sanitization library
    such as DOMPurify to sanitize your HTML.
          Details: https://sg.run/rAx6

           62| <p dangerouslySetInnerHTML={{__html: profile.bio}}></p>
```

*A JSX template using DOMPurify directly*

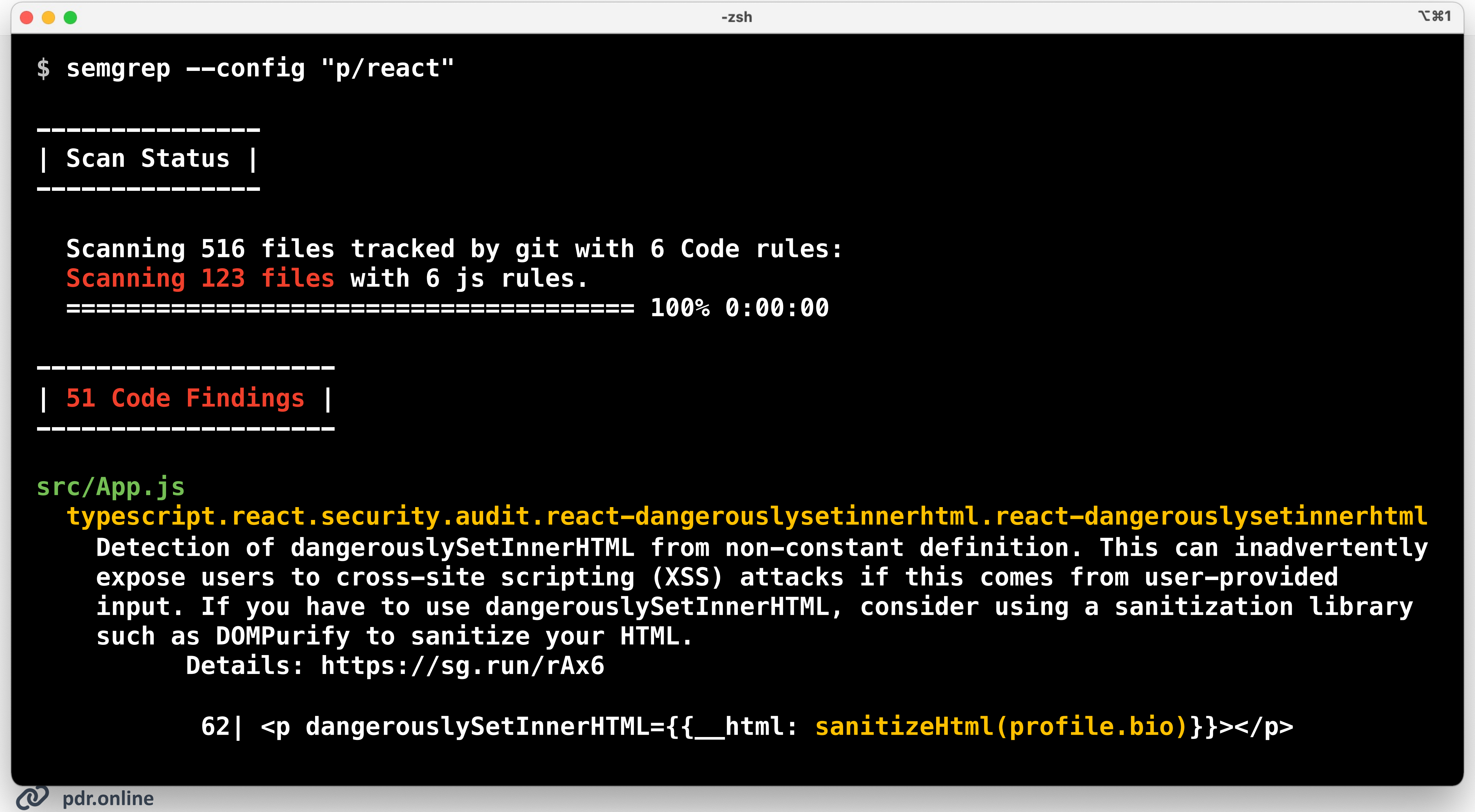
```jsx
import DOMPurify from 'dompurify';

return ( <div>
  <h3>{ title }</h3>
  <p dangerouslySetInnerHTML={{__html: DOMPurify.sanitize(review)}}></p>
</div>);
```

*A JSX template using DOMPurify through an intermediate function*

```jsx
import DOMPurify from 'dompurify';

return ( <div>
  <h3>{ title }</h3>
  <p dangerouslySetInnerHTML={{__html: sanitizeHtml(profile.bio)}}></p>
</div>);
```



```
$ semgrep --config "p/react"

---------------
| Scan Status |
---------------

  Scanning 16 files tracked by git with 6 Code rules:
  Scanning 3 files with 6 js rules.
  ====================================== 100% 0:00:00

------------------
| 1 Code Finding |
------------------

src/App.js
  typescript.react.security.audit.react-dangerouslysetinnerhtml.react-dangerouslysetinnerhtml
    Detection of dangerouslySetInnerHTML from non-constant definition. This can inadvertently
    expose users to cross-site scripting (XSS) attacks if this comes from user-provided
    input. If you have to use dangerouslySetInnerHTML, consider using a sanitization library
    such as DOMPurify to sanitize your HTML.
          Details: https://sg.run/rAx6

           62| <p dangerouslySetInnerHTML={{__html: sanitizeHtml(profile.bio)}}></p>
```



```
$ semgrep --config "p/react"

---------------
| Scan Status |
---------------

  Scanning 516 files tracked by git with 6 Code rules:
  Scanning 123 files with 6 js rules.
  ====================================== 100% 0:00:00

--------------------
| 51 Code Findings |
--------------------

src/App.js
  typescript.react.security.audit.react-dangerouslysetinnerhtml.react-dangerouslysetinnerhtml
    Detection of dangerouslySetInnerHTML from non-constant definition. This can inadvertently
    expose users to cross-site scripting (XSS) attacks if this comes from user-provided
    input. If you have to use dangerouslySetInnerHTML, consider using a sanitization library
    such as DOMPurify to sanitize your HTML.
          Details: https://sg.run/rAx6

           62| <p dangerouslySetInnerHTML={{__html: sanitizeHtml(profile.bio)}}></p>
```

*Using a* SafeHtml *component, avoiding the use of* dangerouslySetInnerHTML

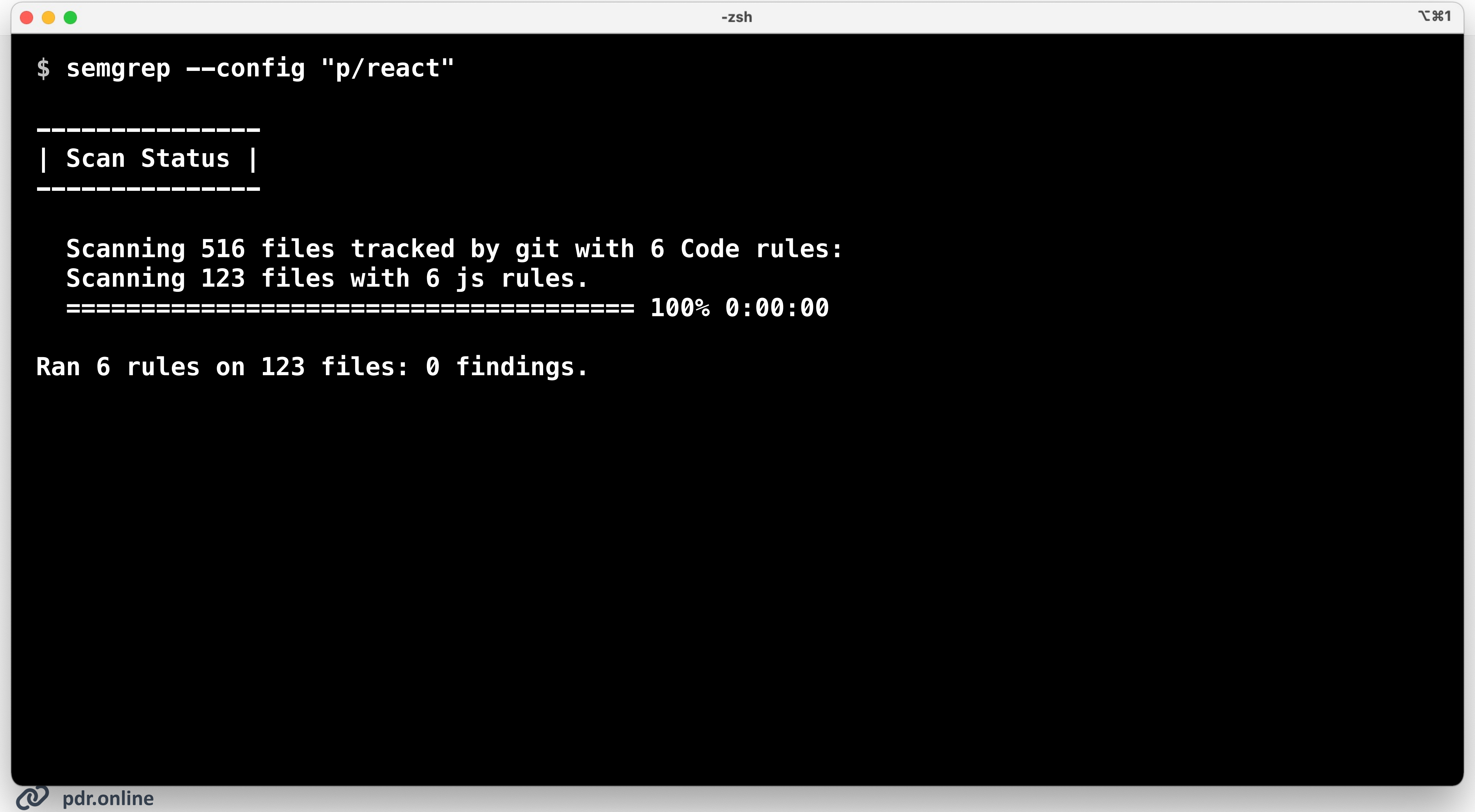
```
import SafeHtml from './SafeHtml';

return ( <div>
  <h3>{ title }</h3>
  <SafeHtml element="p" html={{review}}></SafeHtml>
</div>);
```

*The* SafeHtml *component*

```
import React from 'react';
import DOMPurify from 'dompurify';

// This function will render HTML safely using DOMPurify
function SafeHtml({ element, html }){
    return React.createElement(element, {
       dangerouslySetInnerHTML: { __html: DOMPurify.sanitize(html) }
    });
}
export default SafeHtml;
```

# Encapsulate security behavior in libraries

***Offering the right abstractions absolves developers of the responsibility to apply detailed secure coding guidelines***

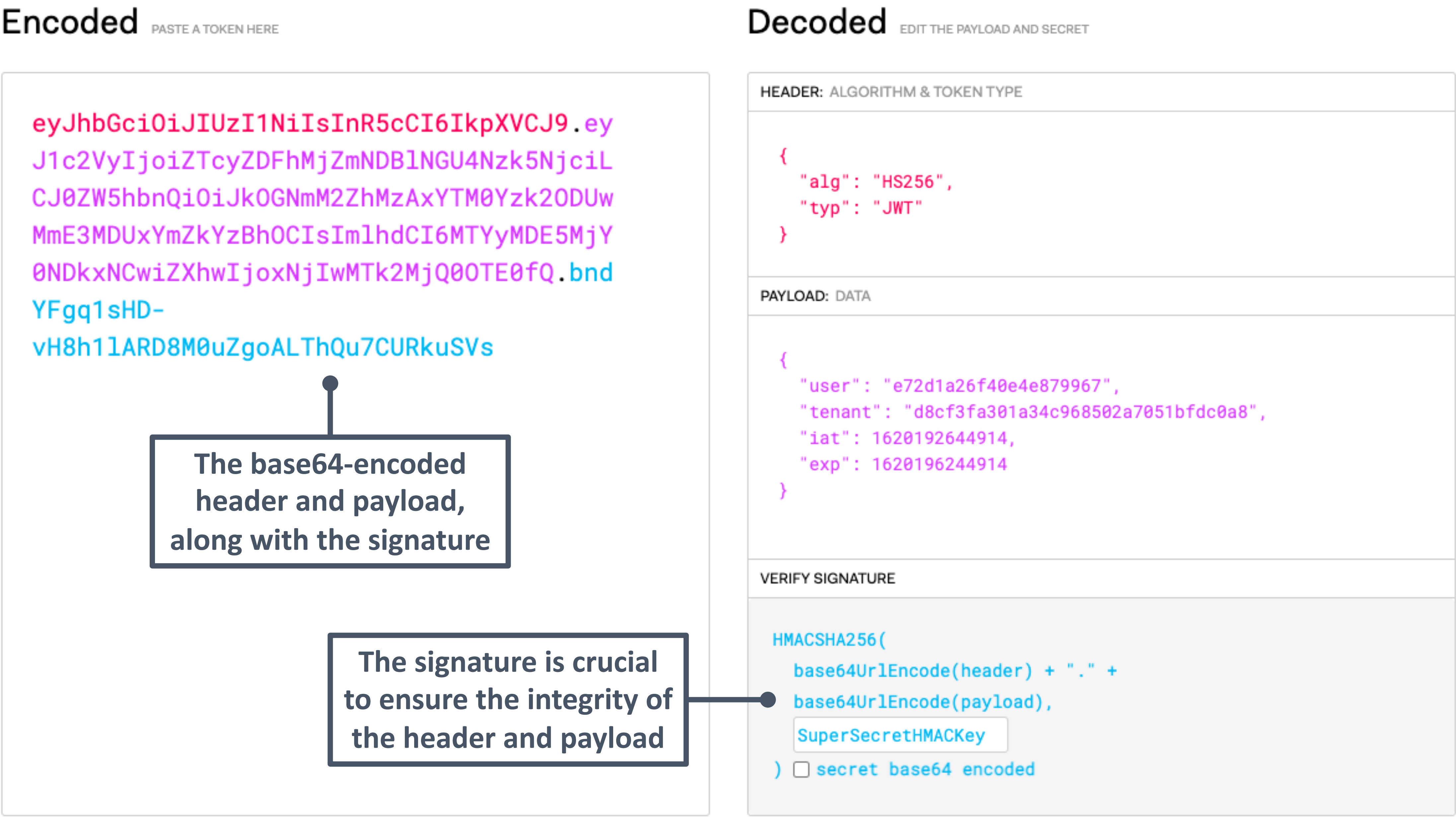


```
$ semgrep --config "p/react"

---------------
| Scan Status |
---------------

  Scanning 516 files tracked by git with 6 Code rules:
  Scanning 123 files with 6 js rules.
  ====================================== 100% 0:00:00

Ran 6 rules on 123 files: 0 findings.
```

# Scaling security with encapsulation and tooling

***Encapsulating security behavior and using proper tooling makes it easier to apply security best practices at scale***

JWT



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
J1c2VyIjoiZTcyZDFhMjZmNDBlNGU4Nzk5NjciL
CJ0ZW5hbnQiOiJkOGNmM2ZhMzAxYTM0Yzk2ODUw
MmE3MDUxYmZkYzBhOCIsImlhdCI6MTYyMDE5MjY
0NDkxNCwiZXhwIjoxNjIwMTk2MjQ0OTE0fQ.bnd
YFgq1sHD-
vH8h1lARD8M0uZgoALThQu7CURkuSVs
```

The base64-encoded header and payload, along with the signature

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

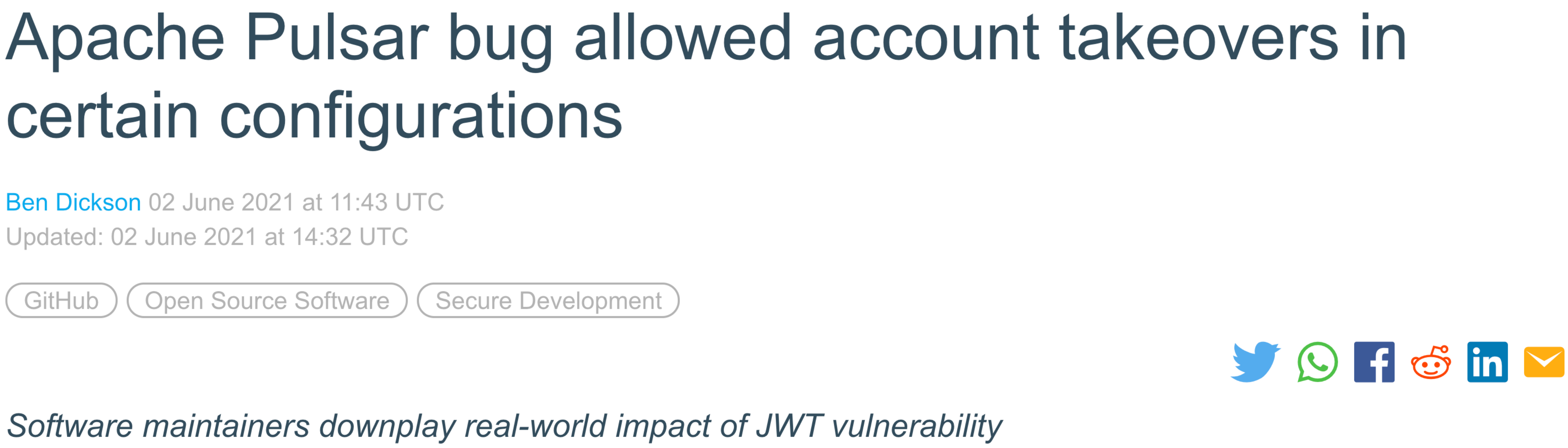
PAYLOAD: DATA

```
{
  "user": "e72d1a26f40e4e879967",
  "tenant": "d8cf3fa301a34c968502a7051bfdc0a8",
  "iat": 1620192644914,
  "exp": 1620196244914
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  SuperSecretHMACKey
) ☐ secret base64 encoded
```

The signature is crucial to ensure the integrity of the header and payload



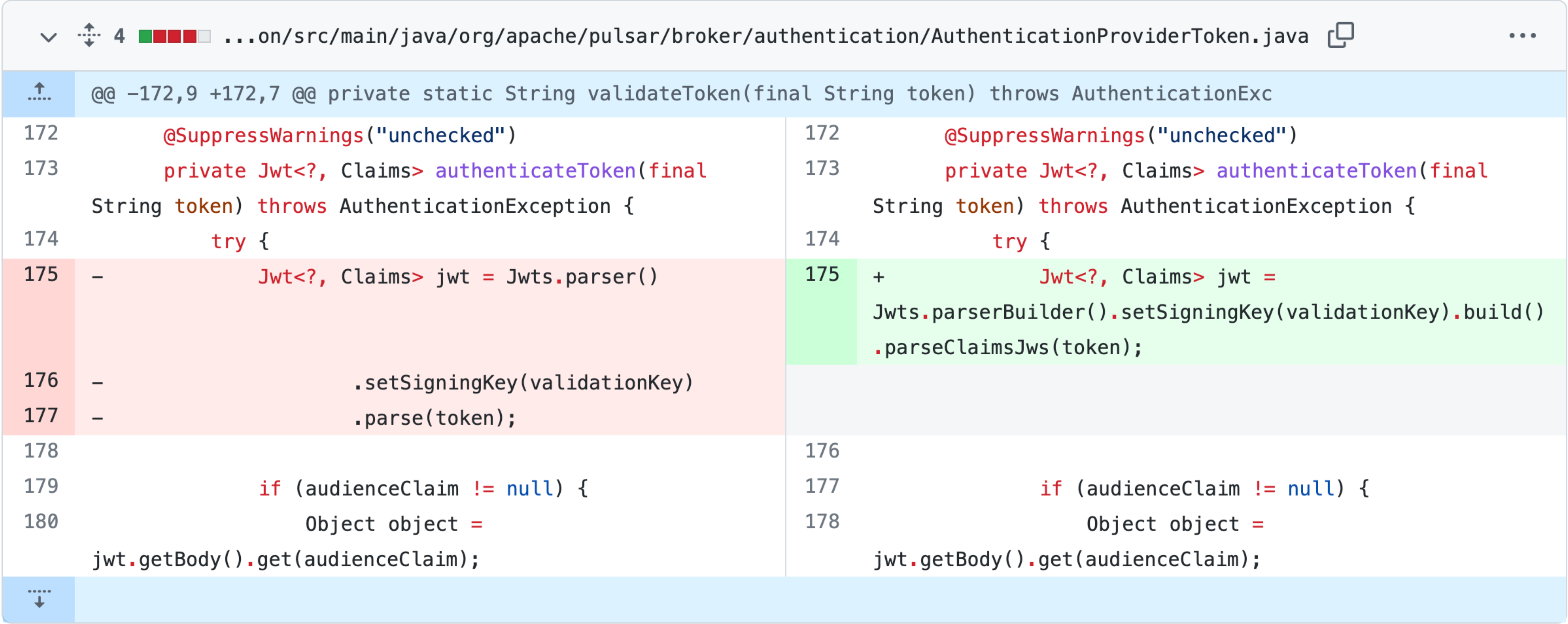
# Apache Pulsar bug allowed account takeovers in certain configurations

Ben Dickson 02 June 2021 at 11:43 UTC
Updated: 02 June 2021 at 14:32 UTC

GitHub | Open Source Software | Secure Development

*Software maintainers downplay real-world impact of JWT vulnerability*

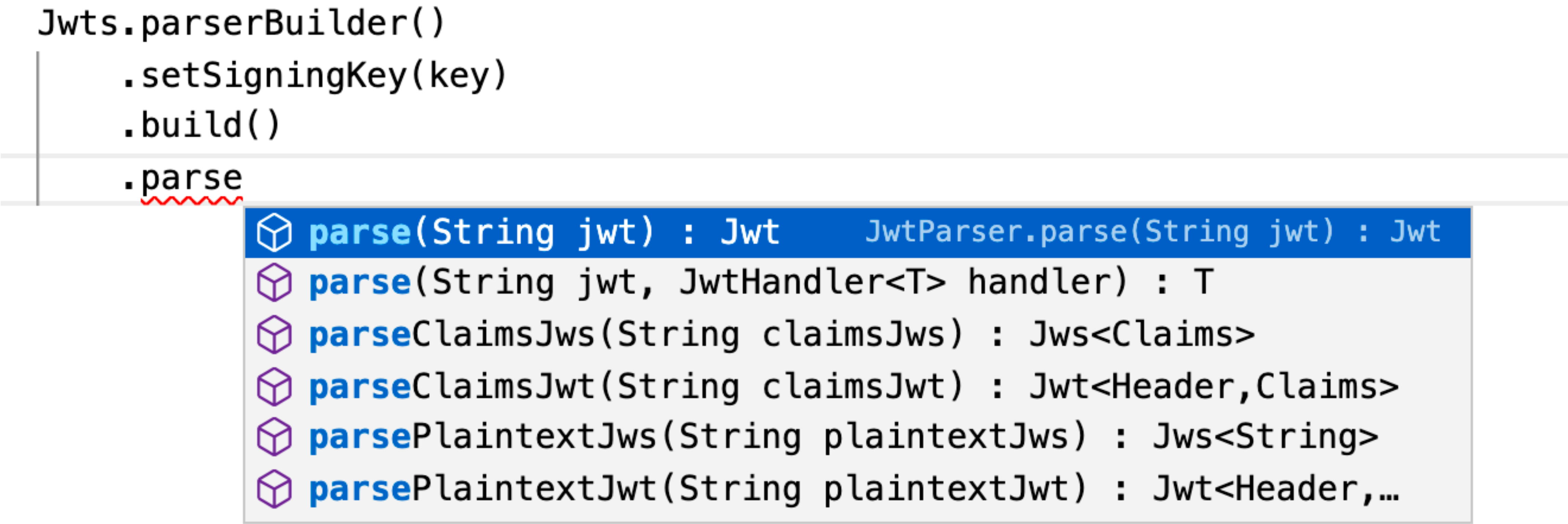
*https://portswigger.net/daily-swig/apache-pulsar-bug-allowed-account-takeovers-in-certain-configurations*



4 ...on/src/main/java/org/apache/pulsar/broker/authentication/AuthenticationProviderToken.java

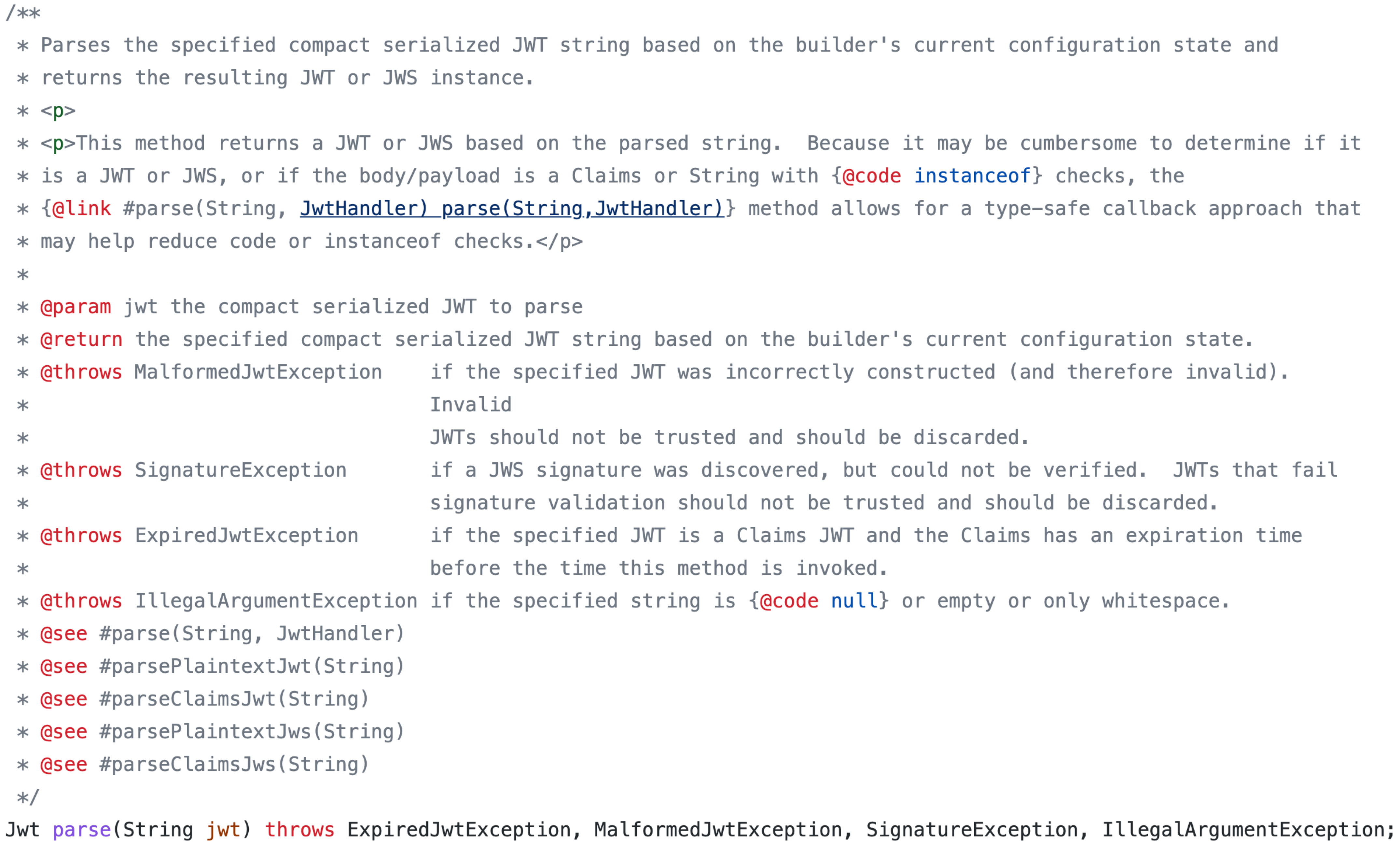
```diff
@@ -172,9 +172,7 @@ private static String validateToken(final String token) throws AuthenticationExc
     @SuppressWarnings("unchecked")
     private Jwt<?, Claims> authenticateToken(final String token) throws AuthenticationException {
         try {
-            Jwt<?, Claims> jwt = Jwts.parser()
-                    .setSigningKey(validationKey)
-                    .parse(token);
+            Jwt<?, Claims> jwt = Jwts.parserBuilder().setSigningKey(validationKey).build().parseClaimsJws(token);
 
             if (audienceClaim != null) {
                 Object object = jwt.getBody().get(audienceClaim);
```

*https://github.com/apache/pulsar/pull/9172/commits/94247dac93542bbcb45fb7104f7204363aad7441*



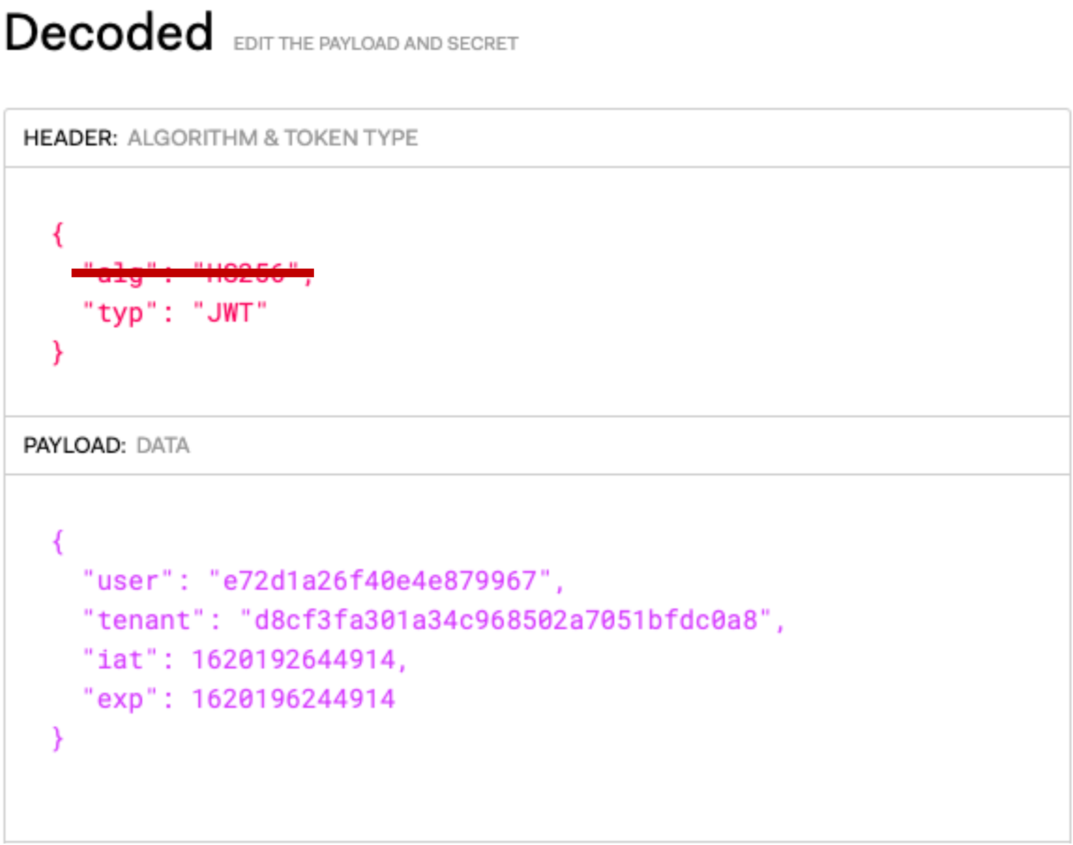
```
Jwts.parserBuilder()
    .setSigningKey(key)
    .build()
    .parse
```

```
parse(String jwt) : Jwt          JwtParser.parse(String jwt) : Jwt
parse(String jwt, JwtHandler<T> handler) : T
parseClaimsJws(String claimsJws) : Jws<Claims>
parseClaimsJwt(String claimsJwt) : Jwt<Header,Claims>
parsePlaintextJws(String plaintextJws) : Jws<String>
parsePlaintextJwt(String plaintextJwt) : Jwt<Header,…
```



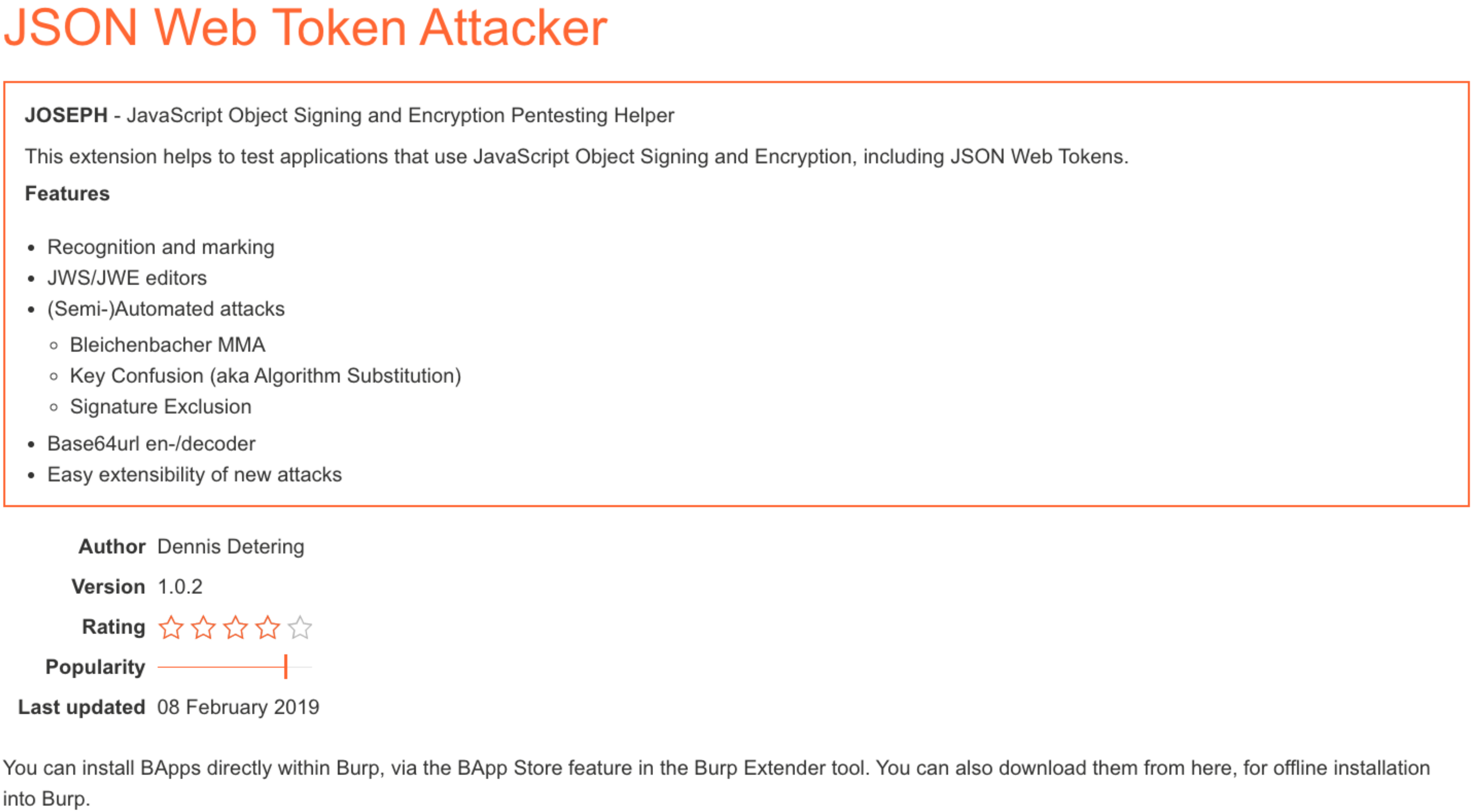
```
/**
 * Parses the specified compact serialized JWT string based on the builder's current configuration state and
 * returns the resulting JWT or JWS instance.
 * <p>
 * <p>This method returns a JWT or JWS based on the parsed string.  Because it may be cumbersome to determine if it
 * is a JWT or JWS, or if the body/payload is a Claims or String with {@code instanceof} checks, the
 * {@link #parse(String, JwtHandler) parse(String,JwtHandler)} method allows for a type-safe callback approach that
 * may help reduce code or instanceof checks.</p>
 *
 * @param jwt the compact serialized JWT to parse
 * @return the specified compact serialized JWT string based on the builder's current configuration state.
 * @throws MalformedJwtException    if the specified JWT was incorrectly constructed (and therefore invalid).
 *                                  Invalid
 *                                  JWTs should not be trusted and should be discarded.
 * @throws SignatureException       if a JWS signature was discovered, but could not be verified.  JWTs that fail
 *                                  signature validation should not be trusted and should be discarded.
 * @throws ExpiredJwtException      if the specified JWT is a Claims JWT and the Claims has an expiration time
 *                                  before the time this method is invoked.
 * @throws IllegalArgumentException if the specified string is {@code null} or empty or only whitespace.
 * @see #parse(String, JwtHandler)
 * @see #parsePlaintextJwt(String)
 * @see #parseClaimsJwt(String)
 * @see #parsePlaintextJws(String)
 * @see #parseClaimsJws(String)
 */
Jwt parse(String jwt) throws ExpiredJwtException, MalformedJwtException, SignatureException, IllegalArgumentException;
```

*https://github.com/jwtk/jjwt/blob/master/api/src/main/java/io/jsonwebtoken/JwtParser.java*



# alg: none

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0
.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
.
```



# JSON Web Token Attacker

**JOSEPH** - JavaScript Object Signing and Encryption Pentesting Helper

This extension helps to test applications that use JavaScript Object Signing and Encryption, including JSON Web Tokens.

**Features**

- Recognition and marking
- JWS/JWE editors
- (Semi-)Automated attacks
  - Bleichenbacher MMA
  - Key Confusion (aka Algorithm Substitution)
  - Signature Exclusion
- Base64url en-/decoder
- Easy extensibility of new attacks

| | |
|---|---|
| **Author** | Dennis Detering |
| **Version** | 1.0.2 |
| **Rating** | ★★★★☆ |
| **Popularity** | |
| **Last updated** | 08 February 2019 |

You can install BApps directly within Burp, via the BApp Store feature in the Burp Extender tool. You can also download them from here, for offline installation into Burp.

*https://portswigger.net/bappstore/82d6c60490b540369d6d5d01822bdf61*

**Ben Knight** Senior Security Consultant

April 16, 2020

## JSON Web Token Validation Bypass in Auth0 Authentication API

Ben discusses a JSON Web Token validation bypass issue disclosed to Auth0 in their Authentication API.

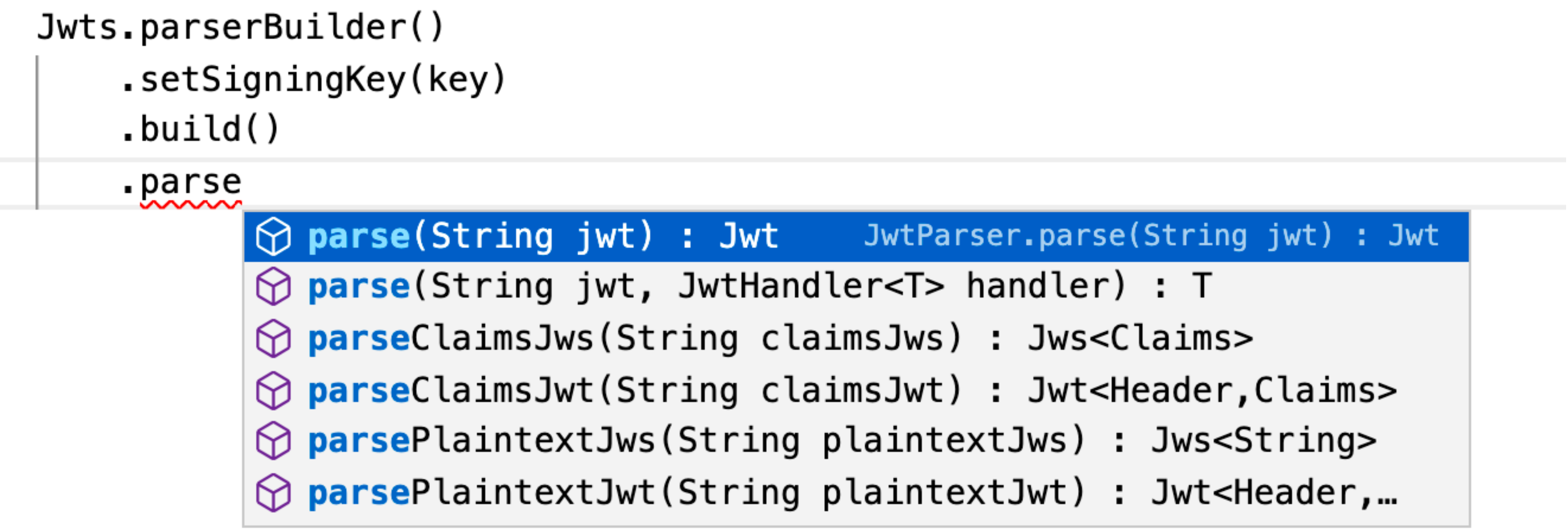
```
alg: NoNe

alg: NONe

alg: nOnE
```

*https://insomniasec.com/blog/auth0-jwt-validation-bypass*

# Include common pitfalls in your test scenarios

***Test your applications to ensure JWTs with "alg:none" are rejected.***

# AppSec is too hard!



```
Jwts.parserBuilder()
    .setSigningKey(key)
    .build()
    .parse
```

```
parse(String jwt) : Jwt          JwtParser.parse(String jwt) : Jwt
parse(String jwt, JwtHandler<T> handler) : T
parseClaimsJws(String claimsJws) : Jws<Claims>
parseClaimsJwt(String claimsJwt) : Jwt<Header,Claims>
parsePlaintextJws(String plaintextJws) : Jws<String>
parsePlaintextJwt(String plaintextJwt) : Jwt<Header,…
```

# What does it take to handle a JWT correctly?

## Choosing the JWT signature scheme

- HMACs or digital signatures?

## Deciding on the signing algorithm

- RS256, PS256, ES256, or EdDSA?

## Verifying the validity of the JWT

- Correctly verifying the signature
- Checking the timestamps (*`nbf`* and *`exp`*)
- Checking the issuer and audience claims (*`iss`* and *`aud`*)

# What does it take to handle a JWT correctly?

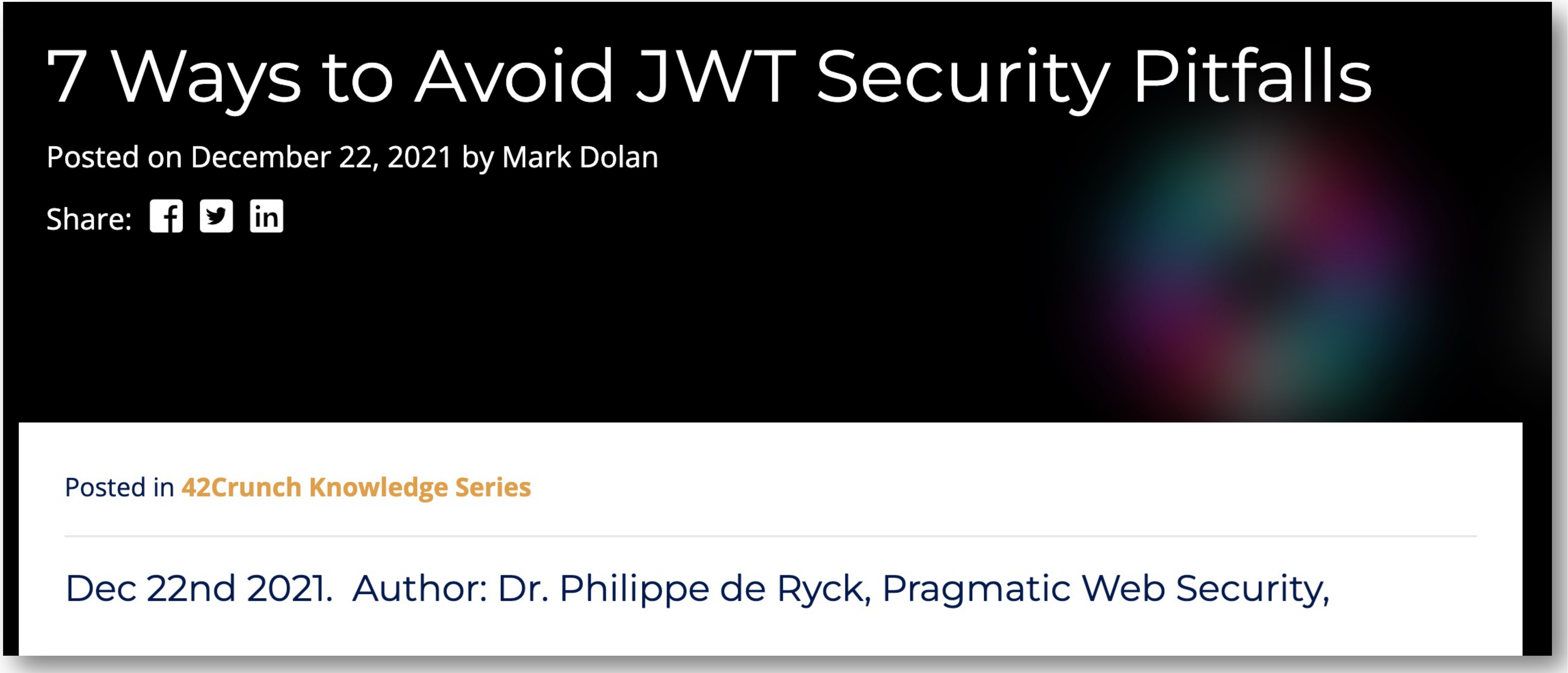
Choosing the JWT signature scheme

Deciding on the signing algorithm

Verifying the validity of the JWT

Using key identifiers to support key rotation

Using explicit JWT typing to avoid token confusion attacks



# 7 Ways to Avoid JWT Security Pitfalls

Posted on December 22, 2021 by Mark Dolan

Share:

Posted in **42Crunch Knowledge Series**

Dec 22nd 2021. Author: Dr. Philippe de Ryck, Pragmatic Web Security,

*https://42crunch.com/7-ways-to-avoid-jwt-pitfalls/*

```
Claims claims =
  Security.verifyAuthenticationToken(token);
```

# Encapsulate security behavior in libraries

***Offering the right abstractions absolves developers of the responsibility to apply detailed secure coding guidelines***

# The Show Must Go On: Securing Netflix Studios At Scale

Netflix Technology Blog Follow
Sep 13 · 11 min read

*Written by Jose Fernandez, Arthur Gonigberg, Julia Knecht, and Patrick Thomas*

In 2017, Netflix Studios was hitting an inflection point from a period of merely rapid growth to the sort of explosive growth that throws "how do we scale?" into every conversation. The vision was to create a "Studio in the Cloud", with applications supporting every part of the business from pitch to play. The security team was working diligently to support this effort, faced with two apparently contradictory priorities:

*https://netflixtechblog.com/the-show-must-go-on-securing-netflix-studios-at-scale-19b801c86479*

# Security as the "paved road"

*Incorporate security in the entire lifecycle,
from setup to development to deployment.*

*Getting security "for free" with regular tasks makes everything better!*

# Key takeaways

1. Security awareness and coding guidelines are only the beginning
2. Encapsulate security behavior to simplify your codebase
3. Leverage tooling to scale security across your organization

# Thank you!

Reach out to discuss
how I can help you with security

https://pragmaticwebsecurity.com