JSON Web Tokens Will Improve Your Life

John SJ Anderson | **@genehack** | LinuxFest Northwest | 6 May 2017

Hi, I'm John.

a/k/a @genehack

infinityinteractive

VP, Tech

Infinity Interactive

This is my dog, Sammy

a/k/a @sammyGenehack

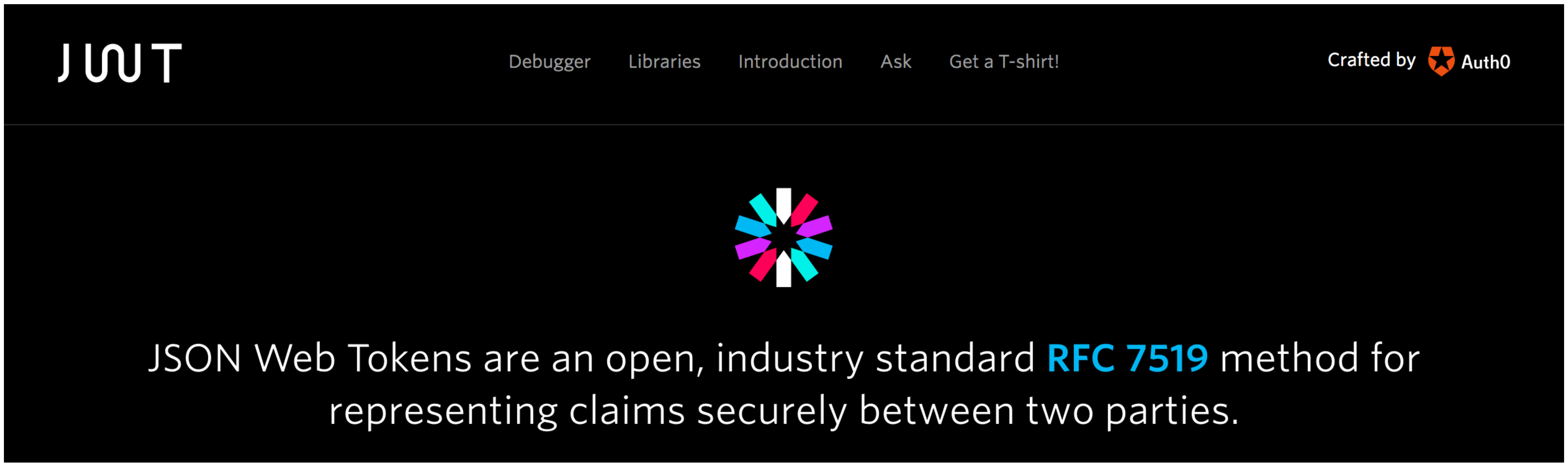
Now that I've been promoted into a management position, I don't get to do all that much coding anymore.

When I do end up with a coding project, Sammy helps me out from her cushion under my desk

So, what's a JWT?

who's heard of JWTs before this talk?
who's using JWTs?

jwt.io



JSON Web Tokens are an open, industry standard **RFC 7519** method for representing claims securely between two parties.

What Does That Even ***Mean***

Sadly, Sammy suffers from the horrible disease RSF -- Resting Stoned Face

Holy RFCs, Batman

- RFC 7519 - JSON Web Token (JWT)

Holy RFCs, Batman

- RFC 7515 - JSON Web Signature (JWS)
- RFC 7516 - JSON Web Encryption (JWE)
- RFC 7517 - JSON Web Key (JWK)
- RFC 7518 - JSON Web Algorithms (JWA)
- RFC 7519 - JSON Web Token (JWT)
- RFC 7520 - Examples of Protecting Content Using JSON Object

It turns out, it's not just RFC7519 you need to worry about...

Sammy found these RFCs a bit …dry

Think of JWTs as...

- A lightweight alternative to cookies (kinda, sorta)
    - ...that also works with CLI, mobile, or even desktop apps
- An authorization or access control mechanism
    - ...kinda like OAuth but without losing the will to live
- Cross-domain friendly

Made of stuff you already know

- Plain ol' JSON Objects (POJOs)
- Stringified, encoded, and cryptographically signed
- Transmitted over HTTP(S)

What do they look like?

- dot-delimited string (`'.'`)
- 3 parts
  - header
  - payload
  - signature
- Example: `xxx.yyyyy.zzz`

Dot-delim

Three parts...

Let's look at each of these three parts in a bit more detail

JWT teardown: header

- Plain ole JSON object
- Base64 encoded
- Typically metadata, such as token type and signing algorithm

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

JWT teardown: payload

- Another Base64-encoded POJO
- Contains "claims" – just key-value data
- Types of keys: reserved, public, private

```
{
  "name": "LinuxFest Northwest",
  "admin": false,
  "iat": 1488562999
}
```

JWT teardown: signature

- Encoded header POJO, plus
- Encoded payload POJO, plus
- A secret, plus
- Signing algorithm from header `alg` key

A word about my code samples

I'm going to be showing a fair amount of code in this talk -- maybe a bit too much code -- but I really wanted to drive home how simple and elegant JWTs are, and showing how they actually work on a code level is the best way to do that, IMO

In real practice, you'd almost certainly be using a library for most of the code I'm showing, but since the point is how uncomplicated most of these operations are, I wanted to give you some idea of what was happening inside that code

For each code sample, I'm going to show the whole code sample -- and it's going to be way too small to read. I'm doing that just so you can see, it's really not that much code. We'll then step through each one in much smaller 3 or 4 line chunks.

Making a JWT

```javascript
function base64EncodeJson (pojo) {
  var jsonString = JSON.stringify(pojo);
  var encoded    = new Buffer(jsonString).toString("base64");
  return encoded;
}

function hmacIt (string, secret) {
  var hmac = crypto.createHmac("sha256" , secret);

  hmac.update(string);

  var signature = hmac.digest("hex");

  return signature;
}

var header = {
  "alg": "HS256",
  "typ": "JWT"
};

var payload = {
  "name": "LinuxFest Northwest",
  "admin": false,
  "iat": 1488562999
};

var secret = "be wery, wery qwiet, we're hunting JWTs";

var encodedHeader  = base64EncodeJson(header);
var encodedPayload = base64EncodeJson(payload);

var signature = hmacIt(encodedHeader + "." + encodedPayload, secret);

var jwt = encodedHeader + "." + encodedPayload + "." + signature;
```

Helper functions

```
function base64EncodeJson (pojo) {
  var jsonString = JSON.stringify(pojo);
  var encoded    = new Buffer(jsonString)
                     .toString("base64");
  return encoded;
}
```

Helper functions

```
function hmacIt (string, secret) {
  var hmac = crypto.createHmac("sha256" , secret);

  hmac.update(string);

  var signature = hmac.digest("hex");

  return signature;
}
```

The actual data

```
var header = {
  "alg": "HS256",
  "typ": "JWT"
};

var payload = {
  "name": "LinuxFest Northwest",
  "admin": false,
  "iat": 1488562999
};

var secret = "be wery, wery qwiet, we're hunting JWTs";
```

Really generating the signature

```
var encodedHeader  = base64EncodeJson(header);
var encodedPayload = base64EncodeJson(payload);

var signature = hmacIt(
  encodedHeader + "." + encodedPayload,
  secret
);

var jwt = encodedHeader
        + "." + encodedPayload
        + "." + signature;
```

Hand-rolled, artisanal JWT

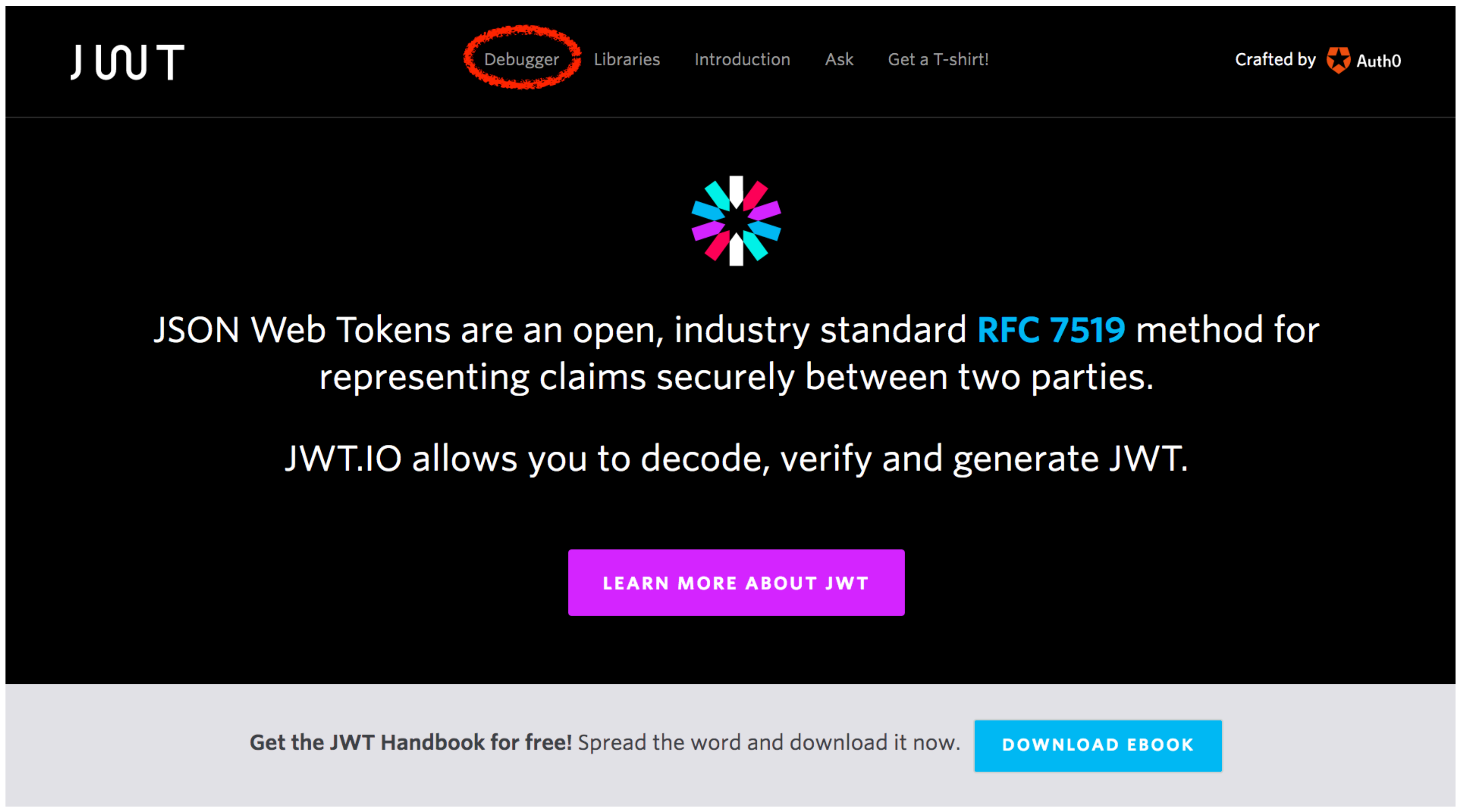
```
console.log(jwt);

'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJuYW1lIjoiTGludXhGZXN0IE5vcnRod2VzdCIsImFkbWluIjpmYWxzZSwiaWF0IjoxNDg4NTYyOTk5fQ==.
3e87e3e7a2d1614193ed308b666bda51a55aeec8c8af2374d32041f7b61130b7'
```

Key bit:

The signature means you can detect any attempts to modify the header or the payload.

Validation

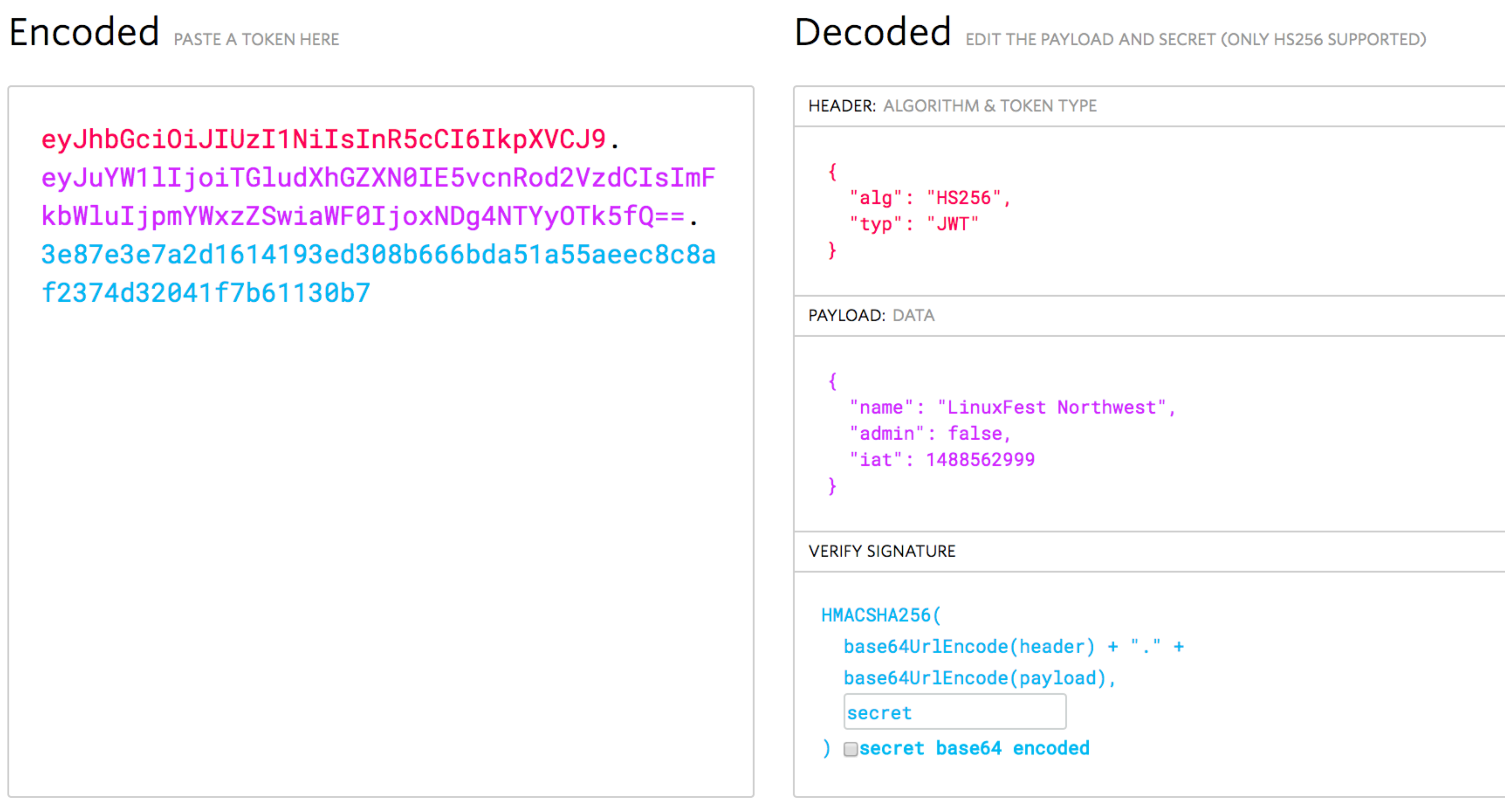


JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.

JWT.IO allows you to decode, verify and generate JWT.

Get the JWT Handbook for free! Spread the word and download it now.

Validation



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJuYW1lIjoiTGludXhGZXN0IE5vcnRod2VzdCIsImF
kbWluIjpmYWxzZSwiaWF0IjoxNDg4NTYyOTk5fQ==.
3e87e3e7a2d1614193ed308b666bda51a55aeec8c8a
f2374d32041f7b61130b7
```

Decoded EDIT THE PAYLOAD AND SECRET (ONLY HS256 SUPPORTED)

HEADER: ALGORITHM & TOKEN TYPE

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

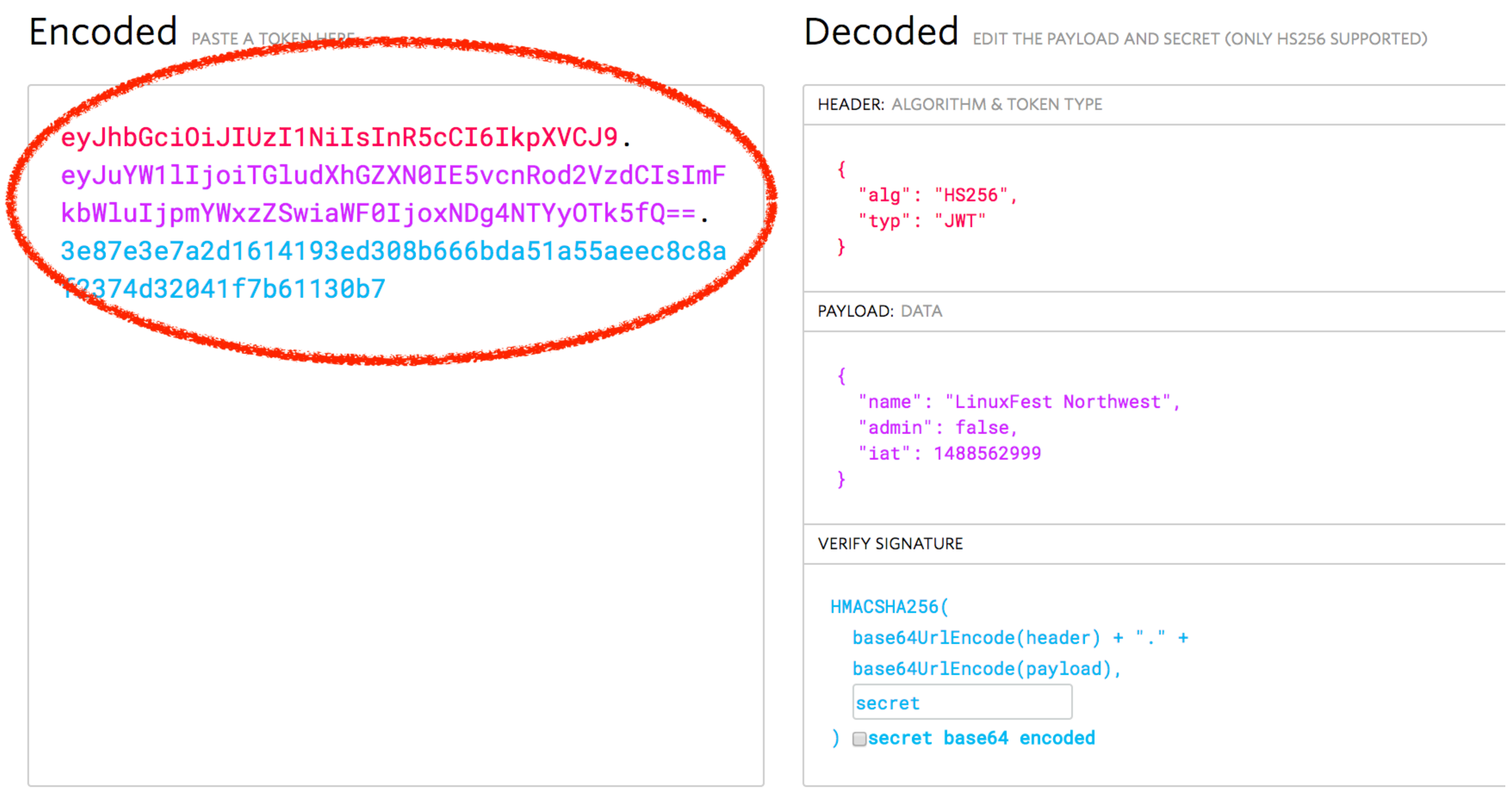
PAYLOAD: DATA

```
{
  "name": "LinuxFest Northwest",
  "admin": false,
  "iat": 1488562999
}
```

VERIFY SIGNATURE

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

Validation


Encoded PASTE A TOKEN HERE
Decoded EDIT THE PAYLOAD AND SECRET (ONLY HS256 SUPPORTED)
HEADER: ALGORITHM & TOKEN TYPE
PAYLOAD: DATA
VERIFY SIGNATURE
secret base64 encoded

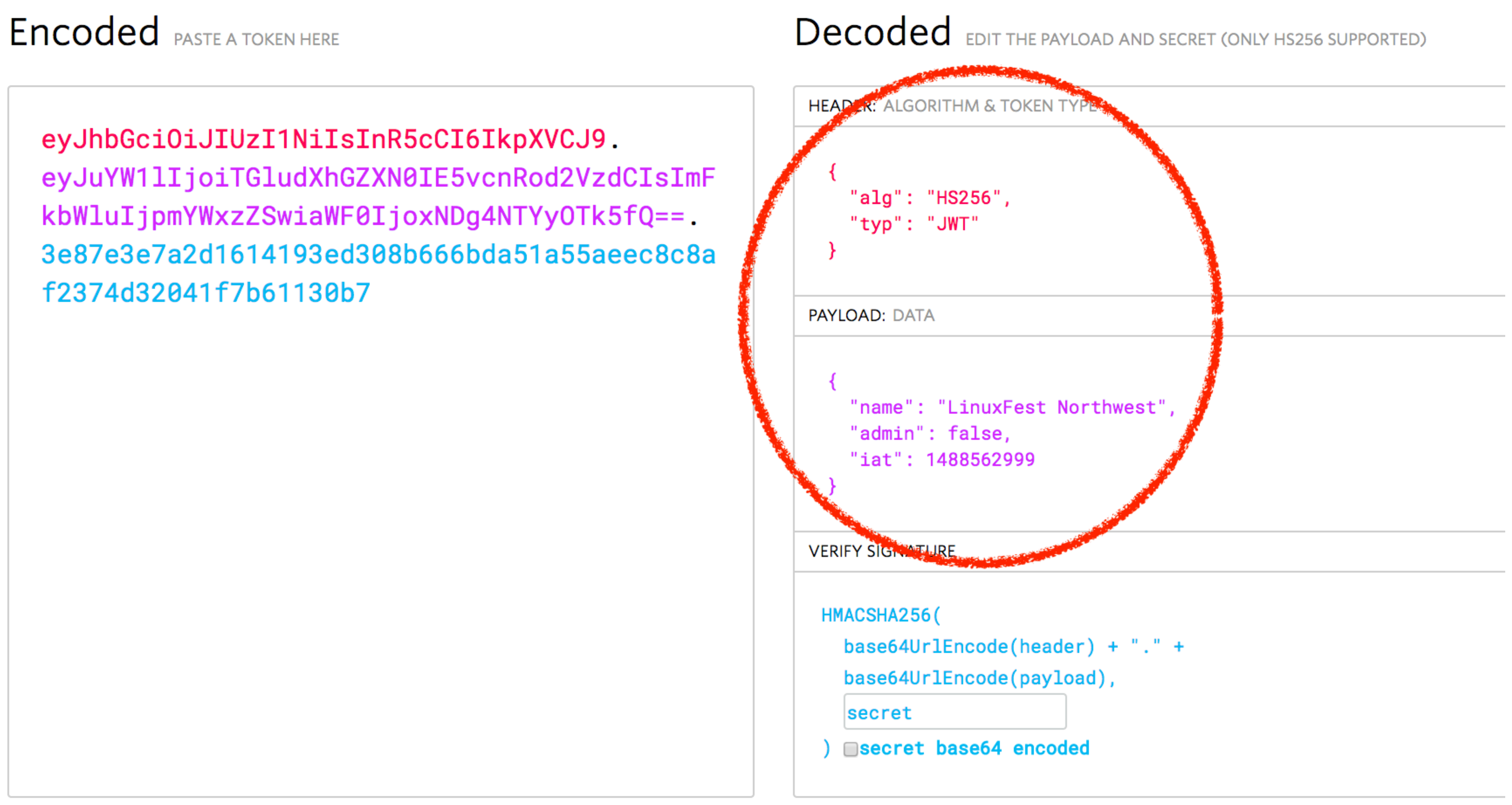
```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJuYW1lIjoiTGludXhGZXN0IE5vcnRod2VzdCIsImF
kbWluIjpmYWxzZSwiaWF0IjoxNDg4NTYyOTk5fQ==.
3e87e3e7a2d1614193ed308b666bda51a55aeec8c8a
f2374d32041f7b61130b7
```

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

```
{
  "name": "LinuxFest Northwest",
  "admin": false,
  "iat": 1488562999
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

Validation



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJuYW1lIjoiTGludXhGZXN0IE5vcnRod2VzdCIsImF
kbWluIjpmYWxzZSwiaWF0IjoxNDg4NTYyOTk5fQ==.
3e87e3e7a2d1614193ed308b666bda51a55aeec8c8a
f2374d32041f7b61130b7
```

Decoded EDIT THE PAYLOAD AND SECRET (ONLY HS256 SUPPORTED)

HEADER: ALGORITHM & TOKEN TYPE

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

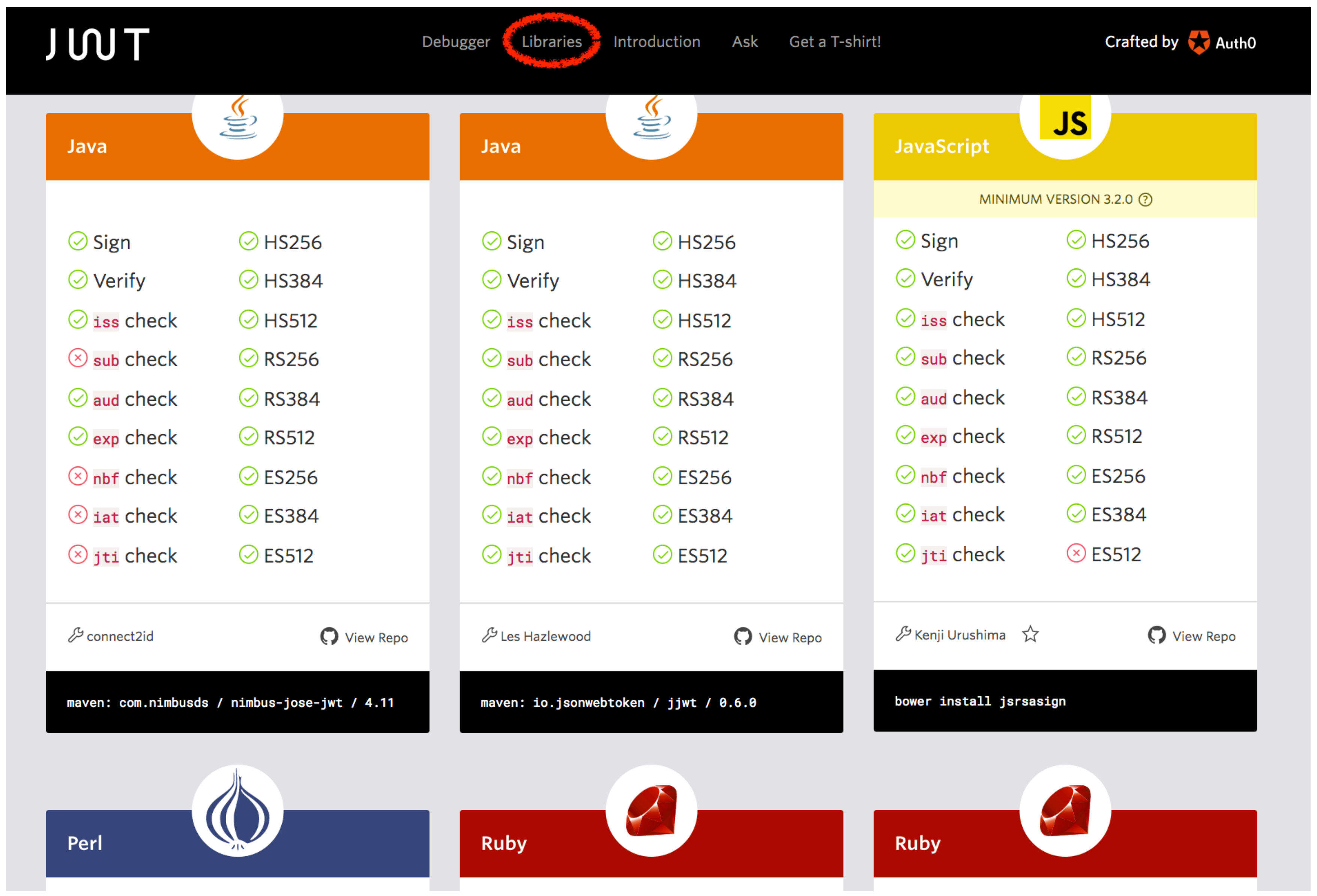
PAYLOAD: DATA

```
{
  "name": "LinuxFest Northwest",
  "admin": false,
  "iat": 1488562999
}
```

VERIFY SIGNATURE

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

Libraries for **DAYS**



Java

✓ Sign	✓ HS256
✓ Verify	✓ HS384
✓ `iss` check	✓ HS512
✗ `sub` check	✓ RS256
✓ `aud` check	✓ RS384
✓ `exp` check	✓ RS512
✗ `nbf` check	✓ ES256
✗ `iat` check	✓ ES384
✗ `jti` check	✓ ES512

connect2id — View Repo

```
maven: com.nimbusds / nimbus-jose-jwt / 4.11
```

Java

✓ Sign	✓ HS256
✓ Verify	✓ HS384
✓ `iss` check	✓ HS512
✓ `sub` check	✓ RS256
✓ `aud` check	✓ RS384
✓ `exp` check	✓ RS512
✓ `nbf` check	✓ ES256
✓ `iat` check	✓ ES384
✓ `jti` check	✓ ES512

Les Hazlewood — View Repo

```
maven: io.jsonwebtoken / jjwt / 0.6.0
```

JavaScript

MINIMUM VERSION 3.2.0

✓ Sign	✓ HS256
✓ Verify	✓ HS384
✓ `iss` check	✓ HS512
✓ `sub` check	✓ RS256
✓ `aud` check	✓ RS384
✓ `exp` check	✓ RS512
✓ `nbf` check	✓ ES256
✓ `iat` check	✓ ES384
✓ `jti` check	✗ ES512

Kenji Urushima — View Repo

```
bower install jsrsasign
```

Perl

Ruby

Ruby

Libraries for **DAYS**

- .NET, Python, Node, Java, Javascript, Ruby, Perl, Go, PHP
- Haskell, Rust, Lua, Scala, Clojure, ObjectiveC, Swift, ***Delphi***
- Support for your favorite language/platform is probably not an issue

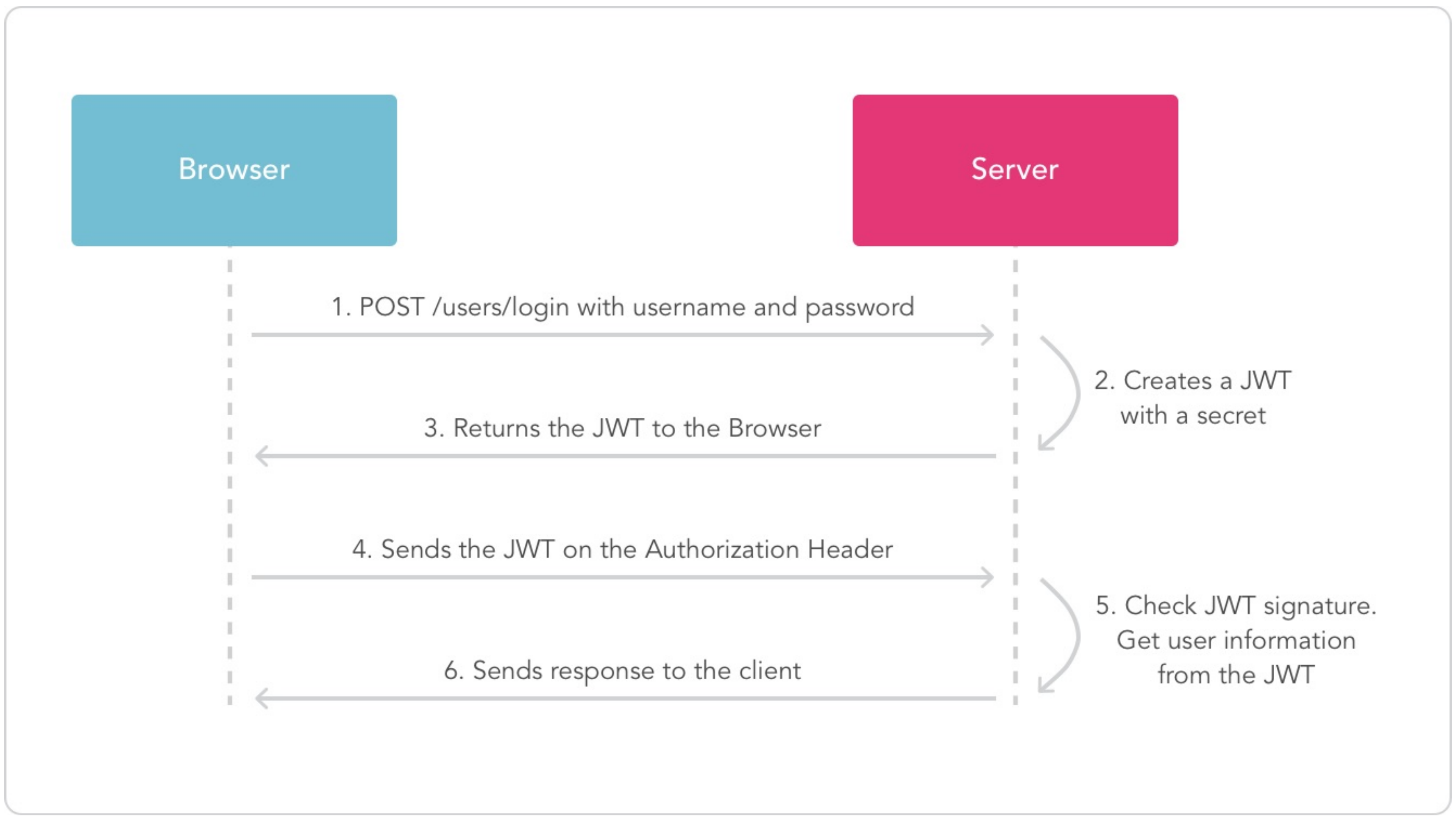
Frequently more than 1 library for a given platform, with varying degrees of support

OK, you've got my attention

At this point, Sammy perked back up a little bit. Now that we were past the RFC reading stage, and she'd seen how simple and elegant JWTs were conceptually, she starting asking...

How do I *actually use* JWTs?

Basic auth/authz usage


Browser
Server
1. POST /users/login with username and password
2. Creates a JWT with a secret
3. Returns the JWT to the Browser
4. Sends the JWT on the Authorization Header
5. Check JWT signature. Get user information from the JWT
6. Sends response to the client

(image stolen from jwt.io)

there are couple of different ways, because JWTs are intentionally pretty flexible. but one way you'll probably end up using them is as part of a fairly standard authentication/ authorization type flow

Things to be aware of

- Payload/header **NOT** encrypted
  - ...don't send anything sensitive!
- Need to control expiration, re-issue, etc.
- Some APIs will send a fresh JWT to the client *per-request*
- Sites other than issuing site can receive JWT
  - ...but they must share the secret to validate

How is it actually transmitted?

- Up to you! Various methods:
  - As part of the URL in a `GET`
  - In a `POST` body
  - In the `Authorization` header using Bearer scheme:

```
Authorization: Bearer <token>
```

How would you actually use this in an app?

Sammy isn't very big on theory. She likes to see the actual implementation so she can really understand what's going on...

Generate a token on login

```
app.post('/user/login', app.wrap(user_login));

var jwt = require('jwt'); // helper wrapper around 'jsonwebtoken'

function * user_login (req, res) {
  if (! (req.body.email && req.body.password)) {
    res.status(400);
    res.json({message: 'invalid request'});
    return;
  }

  var user = yield _fetch_user_by_email(req.body.email);

  var claims;

  if (_pw_validate(user.password, req.body.password)) {
    claims = { user_id: user.id };
  } else {
    res.status(401);
    res.header('WWW-Authenticate', 'Bearer realm=myapp');
    res.json({ message: 'authorization required' });
    return;
  }

  // sign the claim set and return the token in a header
  var token = jwt.sign(claims);

  res.append('X-MyApp-Token', token);

  res.status(200);
}
```

Node code for generating a token after a successful login

Generate a token on login

```
app.post('/user/login', app.wrap(user_login));

var jwt = require('jwt'); // helper wrapper around 'jsonwebtoken'

function * user_login (req, res) {
  if (! (req.body.email && req.body.password)) {
    res.status(400);
    res.json({message: 'invalid request'});
    return;
  }
```

This code is from an express app, so first we have to declare a route

Then we import a helper library that's just a thin wrapper around the jsonwebtoken NPM library.

Generate a token on login

```
var user = yield _fetch_user_by_email(req.body.email);

var claims;
if (_pw_validate(user.password, req.body.password)) {
  claims = { user_id: user.id };
}
else {
  res.status(401);
  res.header('WWW-Authenticate', 'Bearer realm=myapp');
  res.json({ message: 'authorization required' });
  return;
}
```

Generate a token on login

```
  // sign the claim set and return the token in a header
  var token = jwt.sign(claims);

  res.append('X-MyApp-Token', token);

  res.status(200);
}
```

OK, that's how you make one.

How do you validate it?

Validate with a middleware

```
// enable JWT-verification middleware

var jwt = require('jwt'); // helper wrapper around 'jsonwebtoken'

app.use(function (req, res, next) {
  // initialize the jwt object
  req.jwt = {};

  // now parse the Authorization header if it exists
  Promise.resolve(req.headers.authorization).then(function (auth) {
    // If the Authorization header is present and employs the correct
    // Bearar scheme, extract the token and attempt to verify it.

    if (auth) {
      var scheme  = auth.split(' ')[0];
      var token   = auth.split(' ')[1];

      if (scheme == 'Bearer') {
        return jwt.verify(token).catch(function (error) {
          throw new Error('failed to verify claim');
        });
      }
    }

    throw new Error('authorization not attempted');
  })
  .then(function (payload) {
    req.jwt = payload;
    next();
  })
  .catch(function (error) {
    // Allow login without JWT
    if (req.path == '/user/login' && req.method == 'POST') {
      return next();
    }

    res.status(401);
    res.header('WWW-Authenticate', 'Bearer realm=myapp');
    res.json({ message: 'authorization required' });
  });
});
```

Again, because this is Express, we can do validation in a middleware. That'll make sure it happens on every request.

Validate with a middleware

```
// enable JWT-verification middleware

var jwt = require('jwt'); // helper wrapper around 'jsonwebtoken'

app.use(function (req, res, next) {
  // initialize the jwt object
  req.jwt = {};
```

Validate with a middleware

```
// now parse the Authorization header if it exists
Promise.resolve(req.headers.authorization).then(function (auth) {
  // If the Authorization header is present and employs the correct
  // Bearar scheme, extract the token and attempt to verify it.

  if (auth) {
    var scheme  = auth.split(' ')[0];
    var token   = auth.split(' ')[1];

    if (scheme === 'Bearer') {
      return jwt.verify(token).catch(function (error) {
        throw new Error('failed to verify claim');
      });
    }
  }

  throw new Error('authorization not attempted');
})
```

Validate with a middleware

```
.then(function (payload) {
  req.jwt = payload;
  next();
})
```

Validate with a middleware

```
  .catch(function (error) {
    // Allow login without JWT
    if (req.path == '/user/login' && req.method == 'POST') {
      return next();
    }

    res.status(401);
    res.header('WWW-Authenticate', 'Bearer realm=myapp');
    res.json({ message: 'authorization required' });
  });
});
```

That's cool. What else you got?

At this point, Sammy was pretty impressed and happy, and was making plans to use JWTs in all her future projects. But she wondered if there was anything else JWTs could do for her...

Recurring dilemma: 'lightweight' access control

This is actually the feature of JWTs that inspired me to give this talk, because JWTs provide an easy solution for a problem that I feel like I've run into time and time again in my coding career

Recurring dilemma: 'lightweight' access control

- Option 1: leave it wide open
  - a/k/a the MongoDB or WTF,YOLO! pattern
- Option 2: implement full authn/authz subsystem
  - ...again
- Option 3: OAuth

😢😢😢😢😢😢

leave it wide open

or implement full login ...and then user management, admin screens, etc.

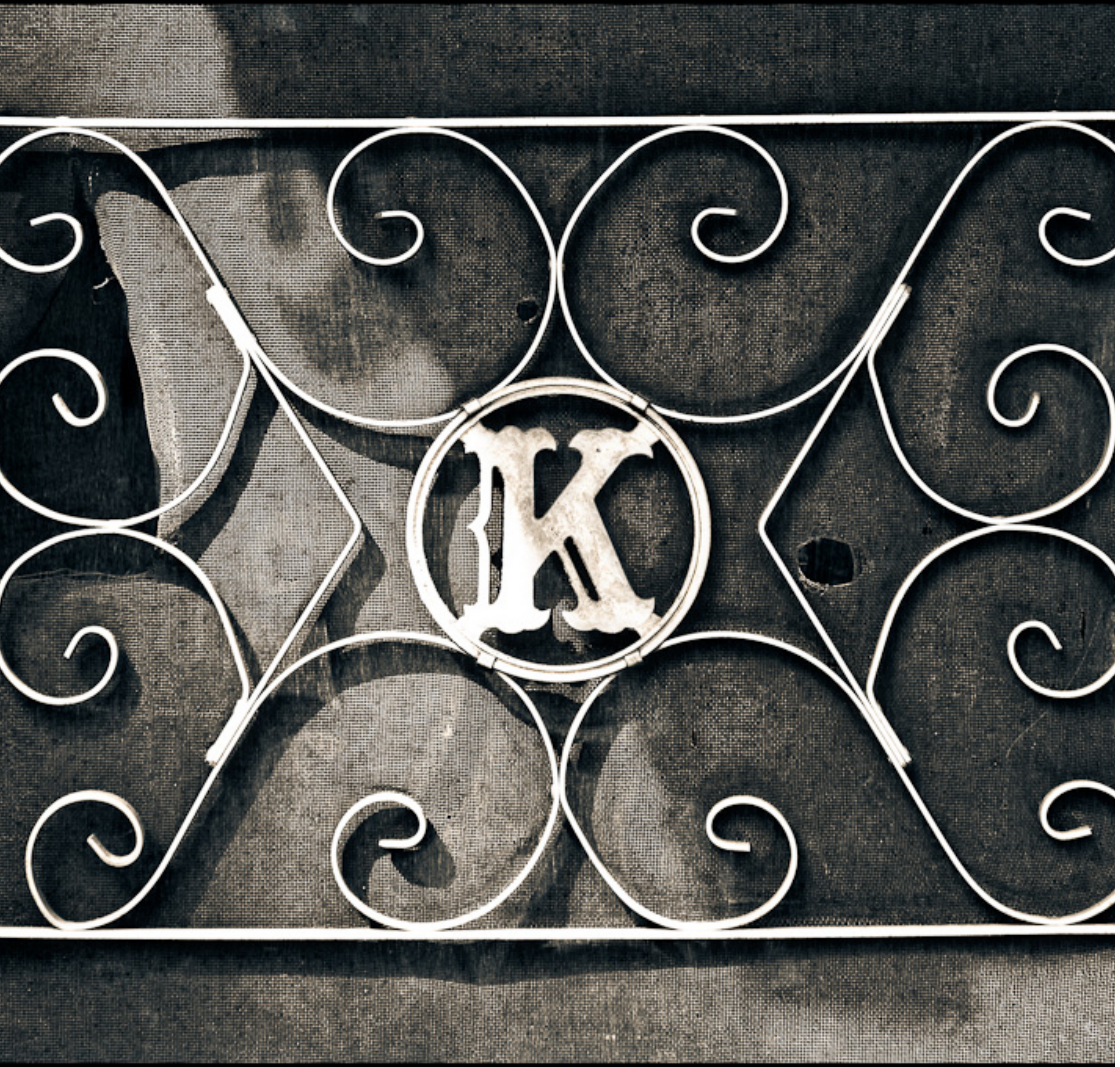
or oauth -- but i've never had a good experience using oauth. anybody here *like* oauth?

Where's the middle ground?

photo credits
* open = https://www.flickr.com/photos/keoni101/5356662124/

Where's the middle ground?

photo credits
* vault = https://www.flickr.com/photos/goodfeeling/24796188573/



A screen door for APIs

photo credit = https://www.flickr.com/photos/slemmon/4938498564

Authorization without authentication

- Scenario:
  - You have an API
  - You don't want to make anybody authenticate to use it
  - You don't want it wide open to the Internet either
- a/k/a authz without authn

Solution: JWT with RSA keys

- Alternative to secret in previous scenario: RSA key-pair
- Can include the public key in the JWT header using JWK
  - JSON Web Key, natch
  - Allows API *client* to produce claims in a verifiable way

To set it up:

- Give authorized API client an RSA key-pair
- Record the fingerprint of the public key (important later!)
- You can even let the client generate the key-pair
- You just need the public key fingerprint

On the client side:

- They make a JWT, using the private key to sign
- They include the public key in the header
- Include `iat` (issued-at) and `exp` (expires) claims
- Send JWT in with API request

On the API side:

- Get the public key out of the header
- Validate the signature using the public key
- Validate that public key fingerprint is white-listed
  - Signature produced with private key
  - Public key is white-listed
- Therefore we know JWT is valid!

Things to be aware of:

- You still want to validate `iat` and `exp` and any other rules
  - Your library should probably do that stuff for you, mostly
- Again, nothing is encrypted, so don't plan on sensitive stuff in the payload or header

Client side code

```perl
use Crypt::JWT qw(encode_jwt);
use Crypt::PK::RSA;
use HTTP::Request;

# generate a JWT and POST a request
my $pri_key = Crypt::PK::RSA->new('./key.pri');
my $pub_key = Crypt::PK::RSA->new('./key.pub');

my $token = encode_jwt(
  alg           => 'RS512',
  extra_headers => {
    jwk   => $pub_key->export_key_jwk('public', 1),
    nonce => undef ,
  },
  key           => $pri_key ,
  payload       => { iat => time() },
  relative_exp  => 1800,
);

HTTP::Request->new(
  'POST' => 'https://example.com/endpoint',
  ['Authorization' => "Bearer $token"],
  encode_json({ request => 'body' })
);
```

Client side code

```
use Crypt::JWT qw(encode_jwt);
use Crypt::PK::RSA;
use HTTP::Request;

my $pri_key = Crypt::PK::RSA->new('./key.pri');
my $pub_key = Crypt::PK::RSA->new('./key.pub');
```

maybe don't store your keys in files in the same directory as your code...

Client side code

```
my $token = encode_jwt(
  alg           => 'RS512',
  extra_headers => {
    jwk   => $pub_key->export_key_jwk('public', 1),
    nonce => undef ,
  },
  key           => $pri_key ,
  payload       => { iat => time() },
  relative_exp  => 1800,
);
```

Client side code

```
HTTP::Request->new(
  'POST' => 'https://example.com/endpoint',
  ['Authorization' => "Bearer $token"],
  encode_json({ request => 'body' })
);
```

Critical bit:

adding the public key to the header

```
extra_headers => {
  jwk    => $pub_key->export_key_jwk('public', 1),
},
```

Key: find an RSA library that supports export to JWK format!

API side

```perl
use Crypt::JWT qw(decode_jwt);
use Crypt::PK::RSA;
use Dancer;
use Try::Tiny;

my $auth_header = request_header 'Authorization' ;

my $token;
status_401 unless ( $token ) = $auth_header =~ /^Bearer (.*)$/;

# try to decode it and confirm valid sig,
# and valid iat and exp claims
my( $header, $payload );
try {
  ( $header, $payload ) = decode_jwt(
    token => $token , decode_header => 1 ,
    accepted_alg => 'RS512' ,
    verify_iat => 1 , verify_exp => 1
  );
};

# no catch block, just drop the error, we're out of here in that case
status_401 unless $header and $payload;

# check that expiration time is less than one hour
status_401 unless $payload->{exp} - $payload->{iat} < 3600;

# check that the included public key is on the whitelist
my $pk = Crypt::PK::RSA->new;
$pk->import_key($header->{jwk});
my $thumbprint = $pk->export_key_jwk_thumbprint;
status_401 unless config->{whitelist}{$thumbprint};

# if we get here, we're all good!
...
```

API side: get the token

```
use Crypt::JWT qw(decode_jwt);
use Crypt::PK::RSA;
use Dancer;
use Try::Tiny;

my $auth_header = request_header 'Authorization' ;

my $token;
status_401 unless ( $token ) = $auth_header =~ /^Bearer (.*)$/;
```

API side: decode the token

```
# try to decode it and confirm valid sig,
# and valid iat and exp claims
my( $header, $payload );
try {
  ( $header, $payload ) = decode_jwt(
    token => $token , decode_header => 1 ,
    accepted_alg => 'RS512' ,
    verify_iat => 1 , verify_exp => 1
  );
};

# no catch block, just drop the error, we're out of here in that case
status_401 unless $header and $payload;
```

API side: decode the token

- Key in header wrong? **FAILS**
- Not right algorithm? **FAILS**
- Doesn't have `iat` and `exp`? **FAILS**

ALL that validation is happening inside the library, so I don't have to worry about it.

- Me? ***WINS***

API side: more checks

- We specify in the API docs that tokens can only be valid for one hour
    - Have to check that ourselves
- Also need to make sure this isn't some *random* RSA keypair
    - Need to make sure we know this public key

API side: more validation

```
# check that expiration time is less than one hour
status_401 unless $payload->{exp} - $payload->{iat} < 3600;

# check that the included public key is on the whitelist
my $pk = Crypt::PK::RSA->new;
$pk->import_key($header->{jwk});

my $thumbprint = $pk->export_key_jwk_thumbprint;
status_401 unless config->{whitelist}{$thumbprint};
```

our api has a rule that the token can't have an expires time more than 1 hour into the future.

we also need to make sure the public key fingerprint is on the allowed list

one weakness with this scheme is, if a valid token leaks, that allows access to the API until it expires -- so make sure you chose an allowable access window based on an evaluation of that potential impact

API side: THAT'S ALL FOLKS

```
# if we get here, we're all good!
```

- We know the public key in the header by its fingerprint,
- so we know the private key was used to sign the JWT
  - (or it wouldn't validate)
- and therefore the JWT is from the private key holder
  - (who is, by definition, authorized!)

IMPORTANT NOTE!

This does, of course, depend on the client keeping the private key actually ***private***

…but revocation is as simple as removing the fingerprint from the whitelist.

More advanced usage

- Encrypted payloads (JWE)
- Nested JWT

See those RFCs!

Conclusions

- JWTs solve some really common problems.
- JWTs solve them in a pretty elegant way.
- This is really pretty damn cool.

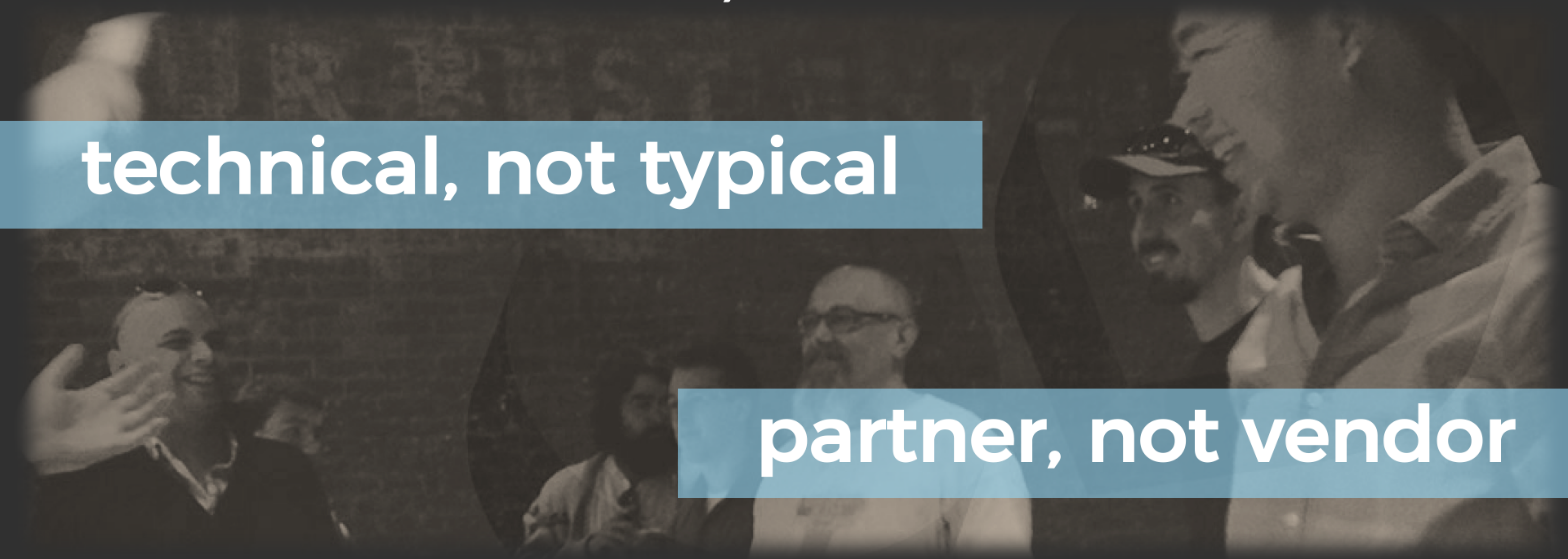
Conclusions

- JWTs solve some really common problems.
- JWTs solve them in a pretty elegant way.
- This is really pretty damn cool**!!!**
- You should think about using JWTs.

Thanks!

- JWT.io / auth0.com folks
- LFNW organizers
- *YOU!*

infinityinteractive



We solve problems with technology.

consulting development web mobile devops

Questions?