Authentication Demystified

Basic Auth to Web Tokens in 60 minutes

Don't worry, my session content is better than my AI art!

Seth Petry-Johnson
seth@petry-johnson.com

My rookie mistake

Federated identity model

Which cryptographic hash function should I choose?

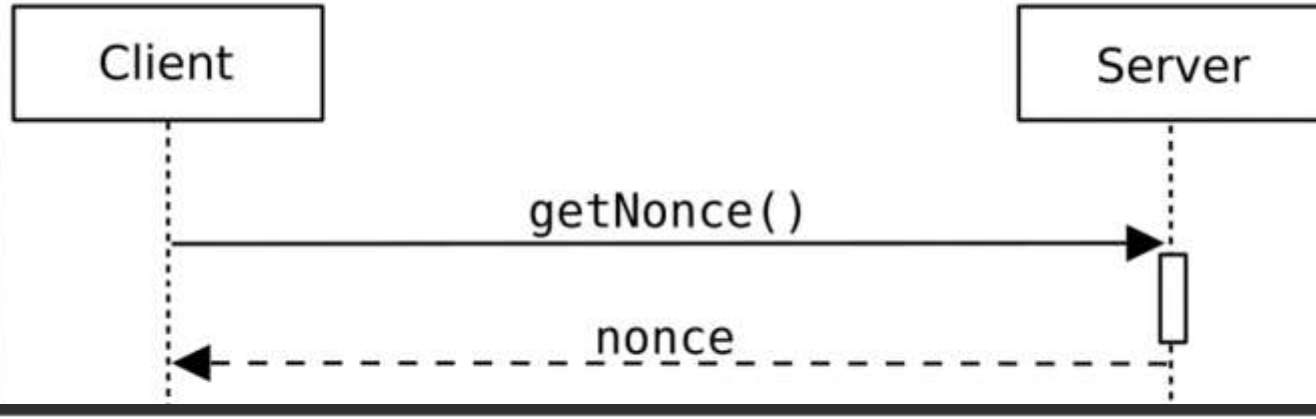
The .NET framework ships with 6 different hashing algorithms:

- MD5: 16 bytes (Time to hash 500MB: 1462 ms)

Securing OAuth Token with HMAC Validation

Implementing security measures in order to prevent the possible attacks is a need in usin
(HMAC) validation is such measure which involved a cryptographic hash function and us
as with any Message Authentication code. In this tutorial you will use the HMAC to valida
Identity Server.

- Preventing miss-use of OAuth Tokens



What's the difference between JWTs and a Bearer T

I'm learning something about Authorization like Basic, Digest, OAuth2.0, JWTs, and Bearer T
JWTs are used as an Access_Token in the OAuth2.0 standard. JWTs appears at RFC 7519, and
Token is at RFC 6750.

For example, the Bearer:

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

All about FIDO2, CTAP2 and WebAuthn

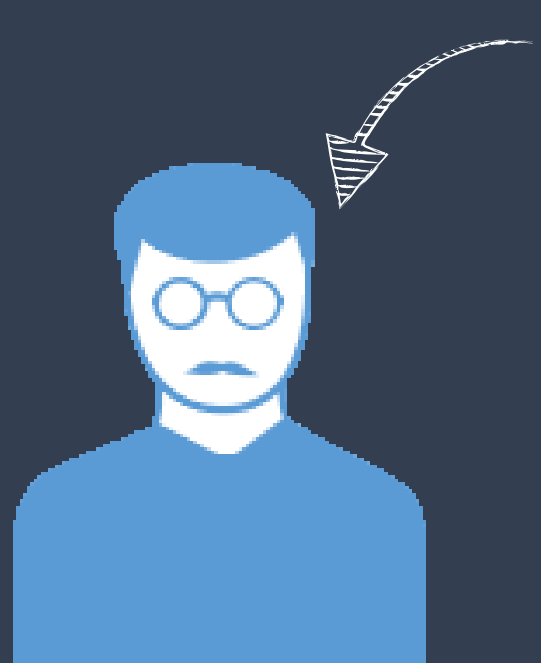
By Pamela Dingle

Published Nov 20 2018 09:00 AM 63.5K Views

This is a great week to be working in Identity Standards, as we at Microsoft celebrate the release of our fi
passwordless authentication at Xbox, Skype, Outlook.com and more. But what are the actual pieces of th
and FIDO2 CTAP2 specifications interact. We will start with the industry standards perspective, and then

Today’s goal: No more rookie mistakes!

This is not an advanced security session!

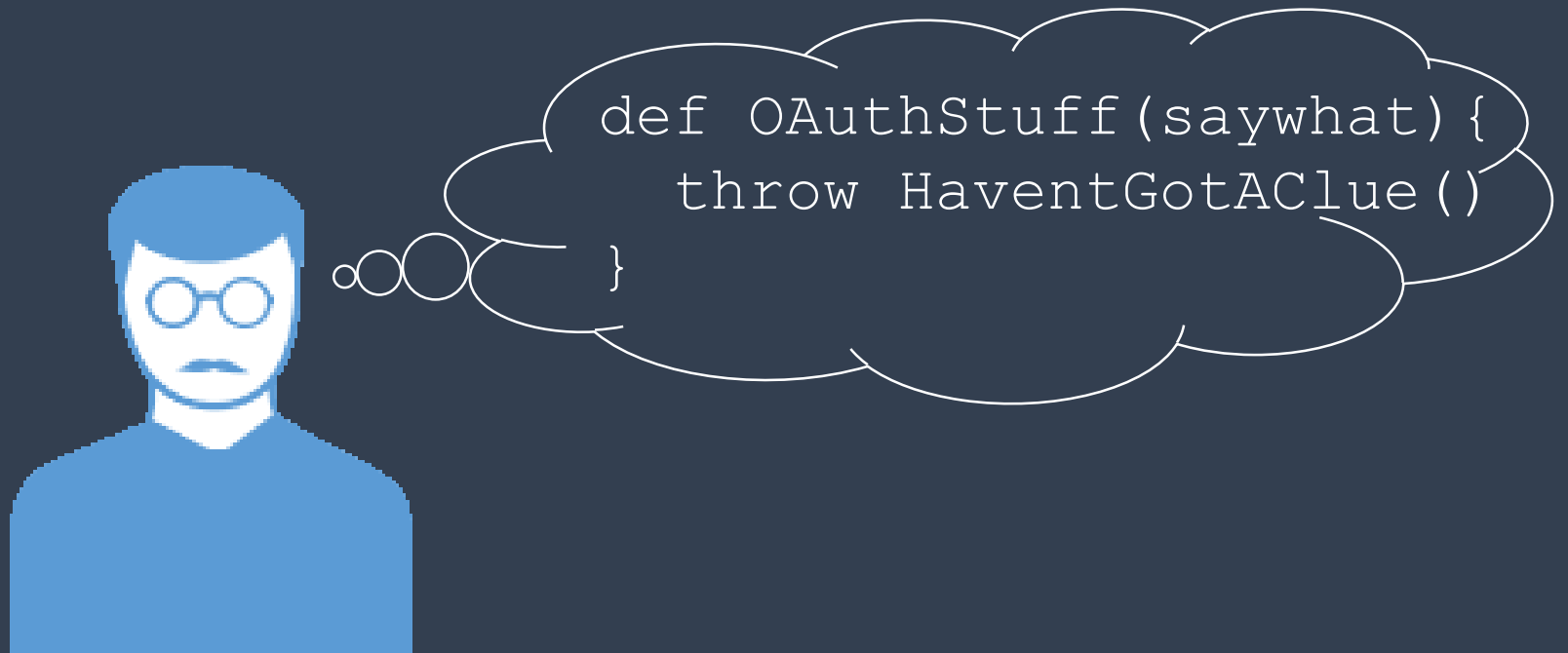


This is me.

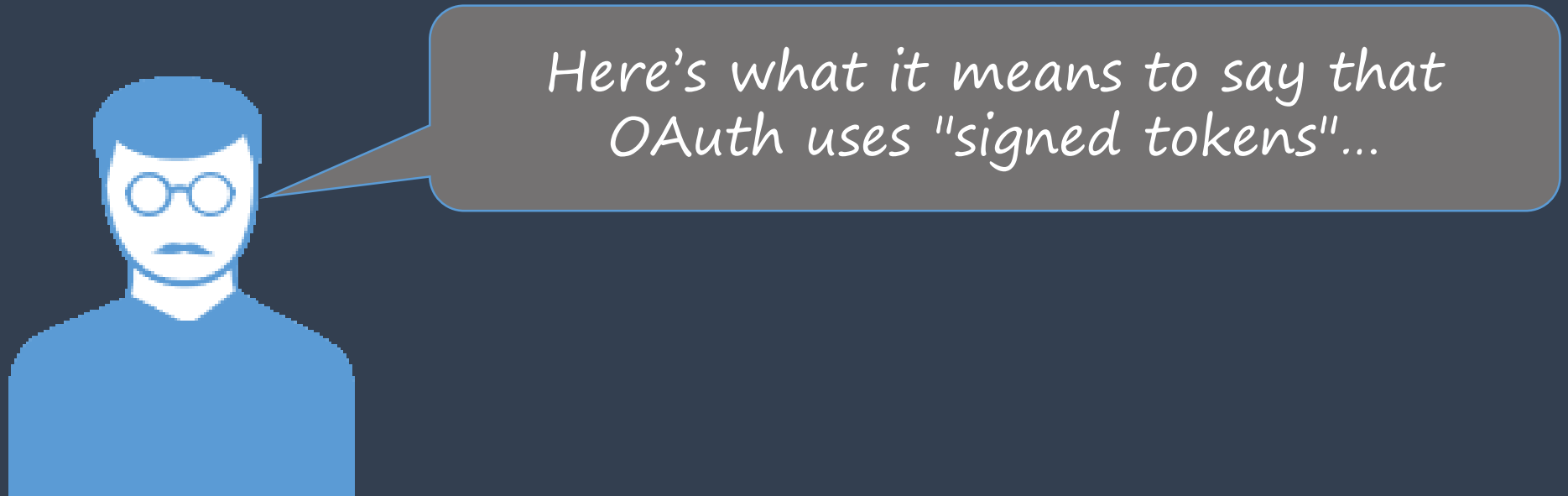
I am not a security expert!

(But I did stay at a Holiday Inn Express once...)

This is not “how to use <foo> in <lang>”



This is “understanding the options”



What's on the agenda?

Passwords
Basic Auth
API Keys as bearer tokens
API Keys as cryptographic keys
JSON Web Tokens
SAML
OAuth
OpenID Connect
Authenticator Apps
Passkeys

Identity *vs* Authentication *vs* Authorization

```
GET /foo/42
```

Identity *vs* Authentication *vs* Authorization

Person (or entity) making the request

```
GET /foo/42
```

Identity *vs* Authentication *vs* Authorization

How we **determine** the identity making the request

```
GET /foo/42
```

Identity *vs* Authentication *vs* Authorization

Confirm the identity **has permission** to perform the request

```
GET /foo/42
```

Authentication

Authentication

Three Factors of Authentication

Something you **know**
(password)

Something you **have**
(passkey / authenticator app)

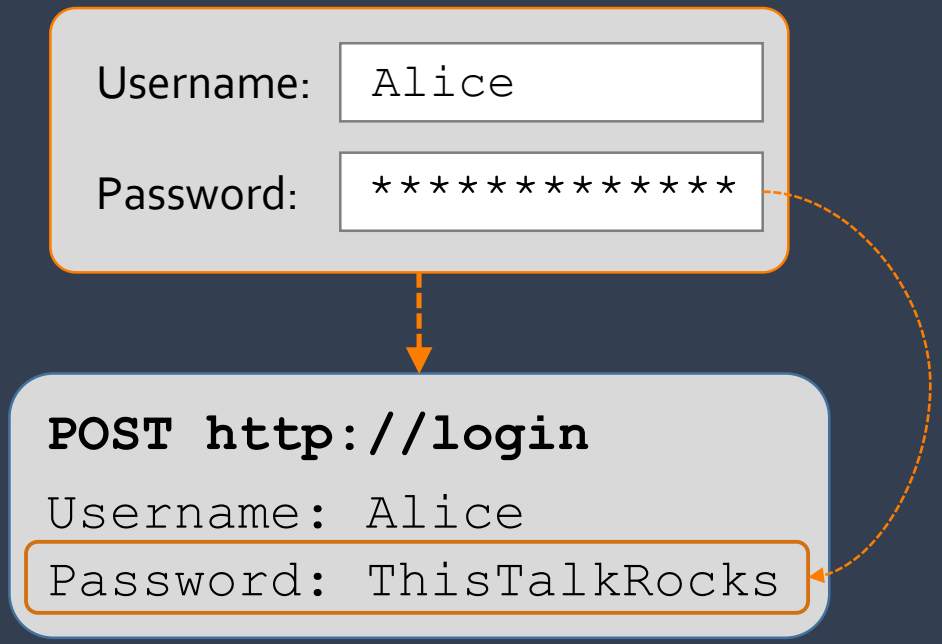
Something you **are**
(biometrics)

Authentication

Something you **know**: passwords

Authentication

Something you **know**: passwords



Authentication

Something you **know**: passwords

Username: Alice

Password: **********

POST http://login

Username: Alice

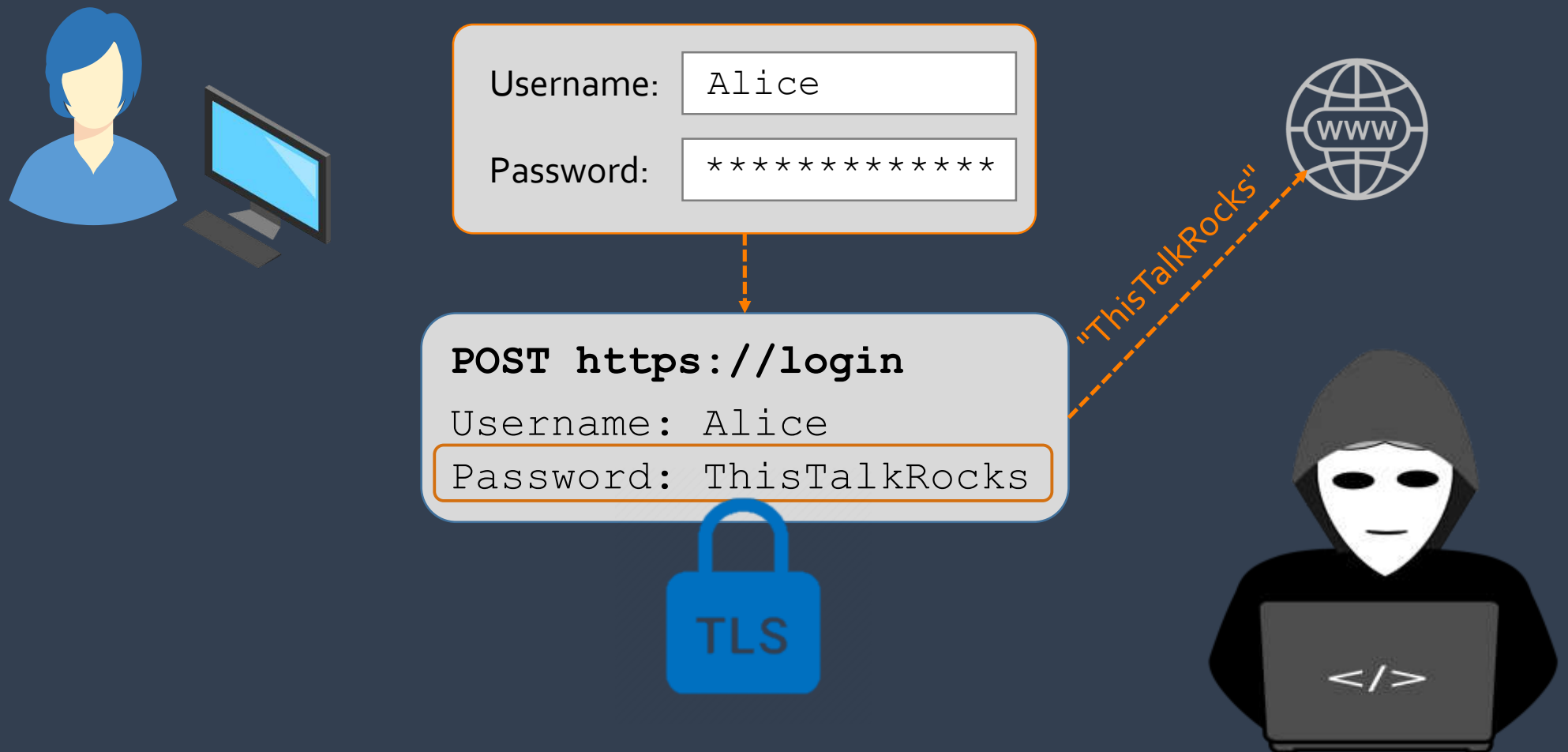
Password: ThisTalkRocks

"ThisTalkRocks"

Mwuahahaha!

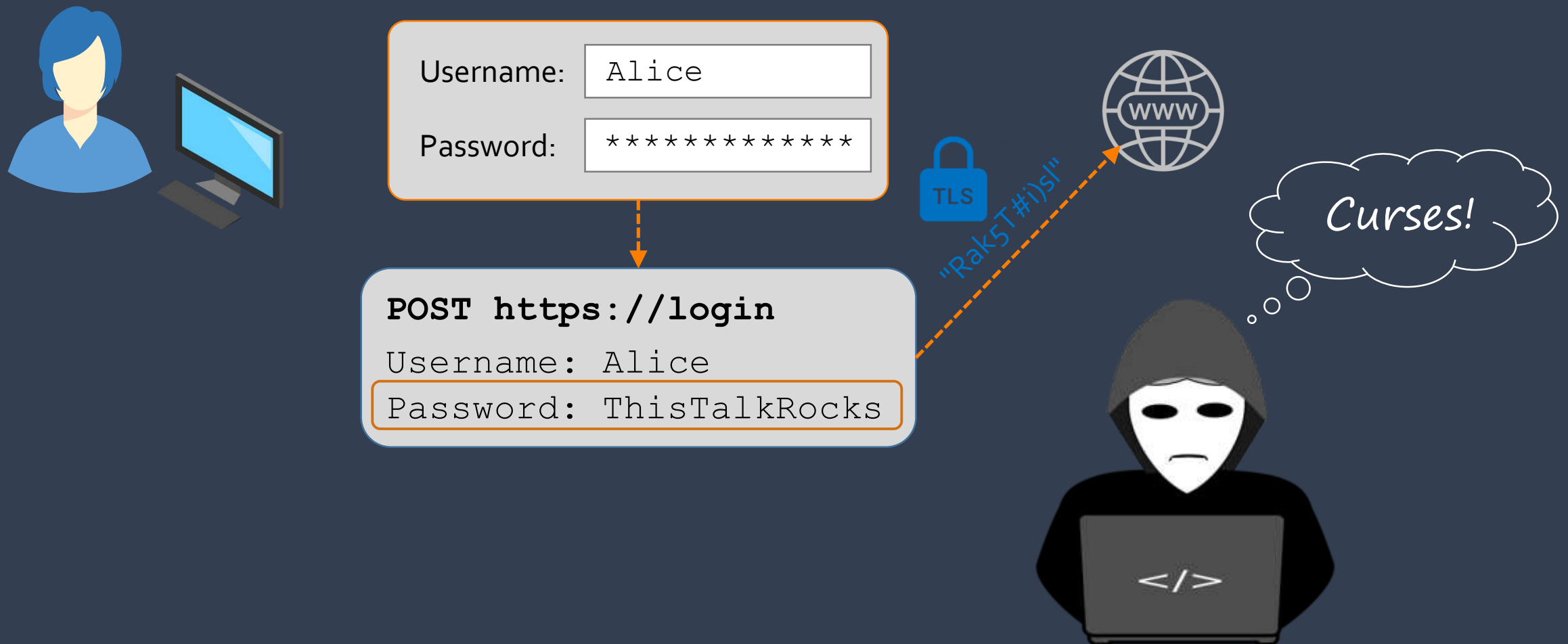
Authentication

Something you **know**: passwords



Authentication

Something you **know**: passwords



Authentication

Something you **know**: passwords

Username: Alice

Password: **********

POST https://login

Username: Alice

Password: ThisTalkRocks

"ThisTalkRocks"

Authentication

Something you **know**: passwords

```
POST https://login
Username: Alice
Password: ThisTalkRocks
```

Authentication

Something you **know**: passwords

Authentication

Something you **know**: passwords

```
POST https://login
Username: Alice
Password: ThisTalkRocks
```

```
Alice
{Encrypted}
```

Authentication

Something you **know**: passwords

POST https://login

Username: Alice

Password: ThisTalkRocks

ENCRYPT(ThisTalkRocks, key)

➔ {Encrypted}

Alice

{Encrypted}

Authentication

Something you **know**: passwords

POST https://login
Username: Alice
Password: ThisTalkRocks

ENCRYPT(ThisTalkRocks, key)
➔ {Encrypted}

DECRYPT({Encrypted}, key)
➔ "ThisTalkRocks"

Alice
{Encrypted}

Authentication

Something you **know**: passwords

```
POST https://login
Username: Alice
Password: ThisTalkRocks
```

```
ENCRYPT(ThisTalkRocks, key)
➔ {Encrypted}
```

```
DECRYPT({Encrypted}, key)
➔ "ThisTalkRocks"
```

We have no need to ever do this!

Alice
ThisTalkRocks

Authentication

Something you **know**: passwords

ENCRYPT(ThisTalkRocks, key)
➔ {Encrypted}

AES-128
OKd/h8mO+fz/ilMd3TeZaw==

HASH(ThisTalkRocks)
➔ {Hashed}

SHA-256
5BD06EA1A11ABF9ABB95A71AF24CF35FFB402CF4168BAF0382BEC264C59D4F

Authentication

Something you **know**: passwords

HASH(ThisTalkRocks)

➔ {Hashed}

Alice

{Hashed}

Authentication

Something you **know**: passwords

HASH(ThisTalkRocks)
➔ {Hashed}

Alice
{Hashed}

Additional reading

- Different hash algorithms
- Salting hashed passwords
- Rainbow tables

Authentication

Authentication

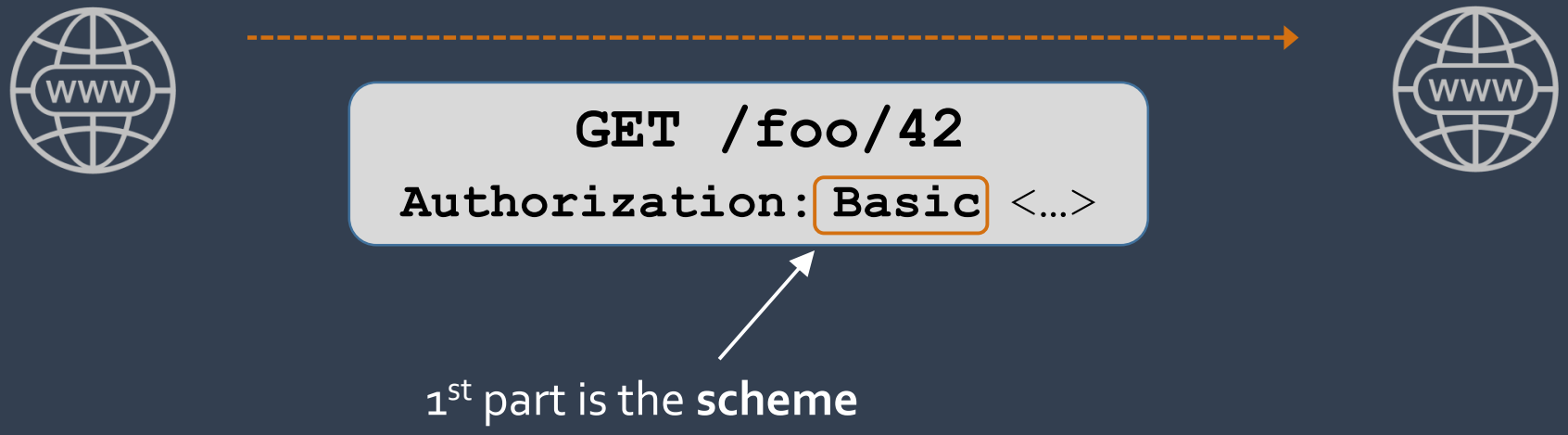
Basic Authentication


```
GET /foo/42
Authorization: Basic <...>
```

This is poorly named. Can be for authorization *or* authentication.

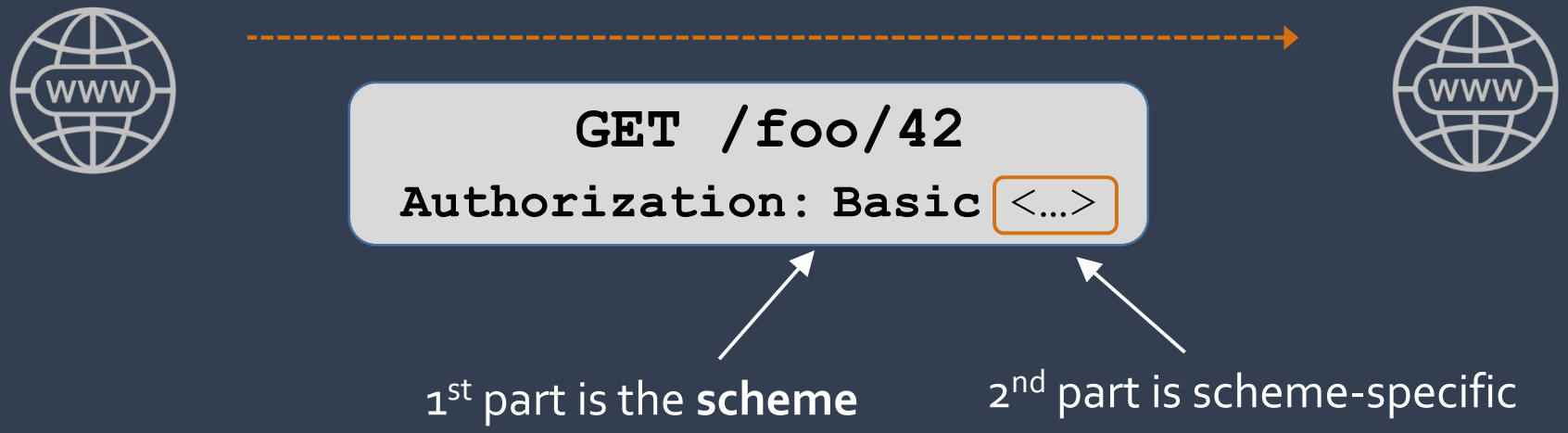
Authentication

Basic Authentication



Authentication

Basic Authentication



Authentication

Basic Authentication

GET /foo/42

Authorization: Basic <…>

```
Alice
ThisTalkRocks
```

Client needs to know the **username** and **password**

Authentication

Basic Authentication


```
GET /foo/42
Authorization: Basic <...>
```

```
Alice:ThisTalkRocks
```

Client **concatenates** those values together

Authentication

Basic Authentication


```
GET /foo/42
Authorization: Basic <...>
```

```
Base64Encode(Alice:ThisTalkRocks)
➔ QWxpY2U6VGhpc1RhbGtSb2Nrcw==
```

Client **Base64 encodes** the concatenated string

This is NOT a secure representation!
It can easily be reversed into "Alice:ThisTalkRocks"

Authentication

Basic Authentication

Encoded string
sent in HTTP Header

Authentication

Basic Authentication

GET /foo/42

Authorization: Basic QWxpY2U…

`Alice:ThisTalkRocks`

Server **Base64 decodes** the string into credentials

Authentication

Basic Authentication

```
Alice:ThisTalkRocks
```

There's one big issue w/ passwords

Authentication

Basic Authentication

```
Alice:ThisTalkRocks
```

Passwords are the
primary account credential

Authentication API Keys

```
Alice:O7SEkjRU2Um_UKyN3iqqmA
```

Use **API Keys** as limited-privilege passwords

Authentication API Keys

Alice
API 1: **FG75319573!&#1d**
API 2: **6Ha*41d841(b@21**

API Keys = usage-specific credentials

For API calls only; NOT for logging in

Authentication API Keys

Alice
API 1: **FG75319573!&#1d**
API 2: **6Ha*41d841(b@21**

API Keys = usage-specific credentials

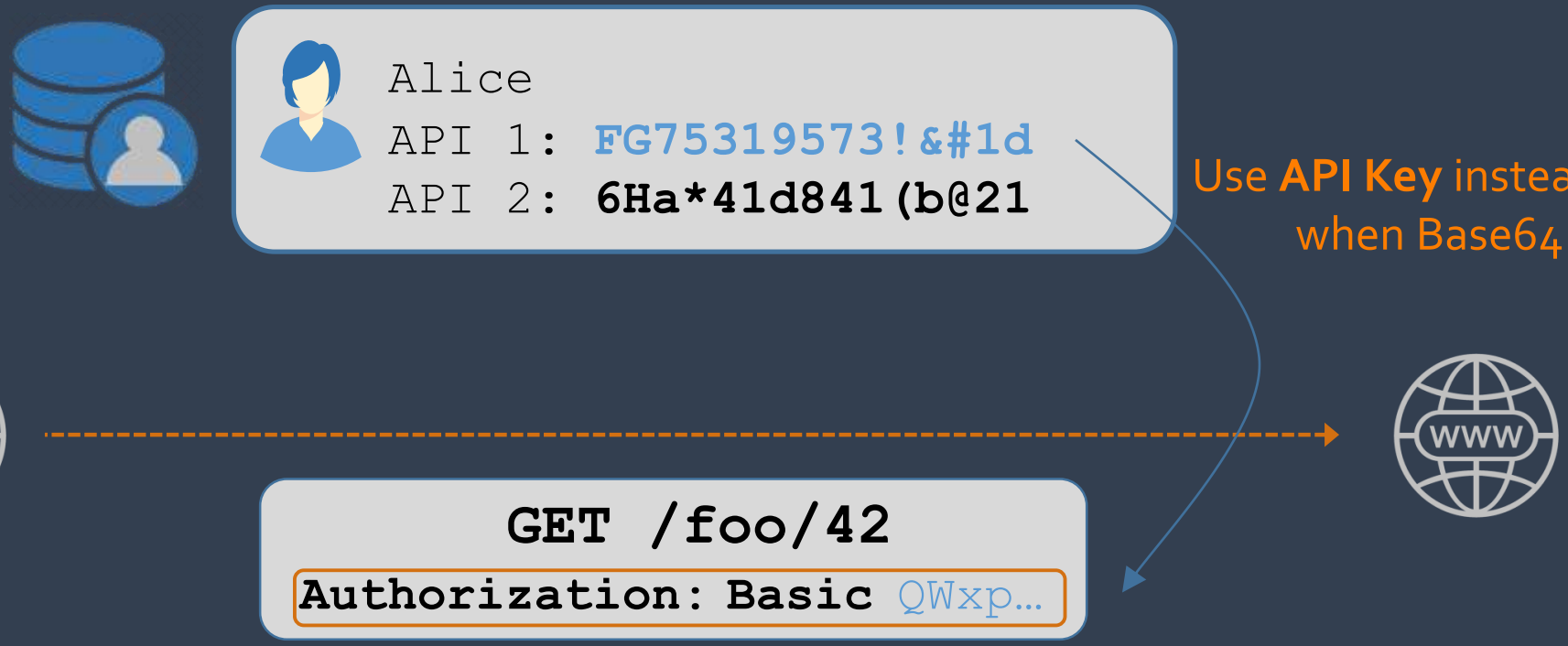
For API calls only; NOT for logging in

Do not use GUIDs for API Keys!

https://bit.ly/**GuidsArentKeys**

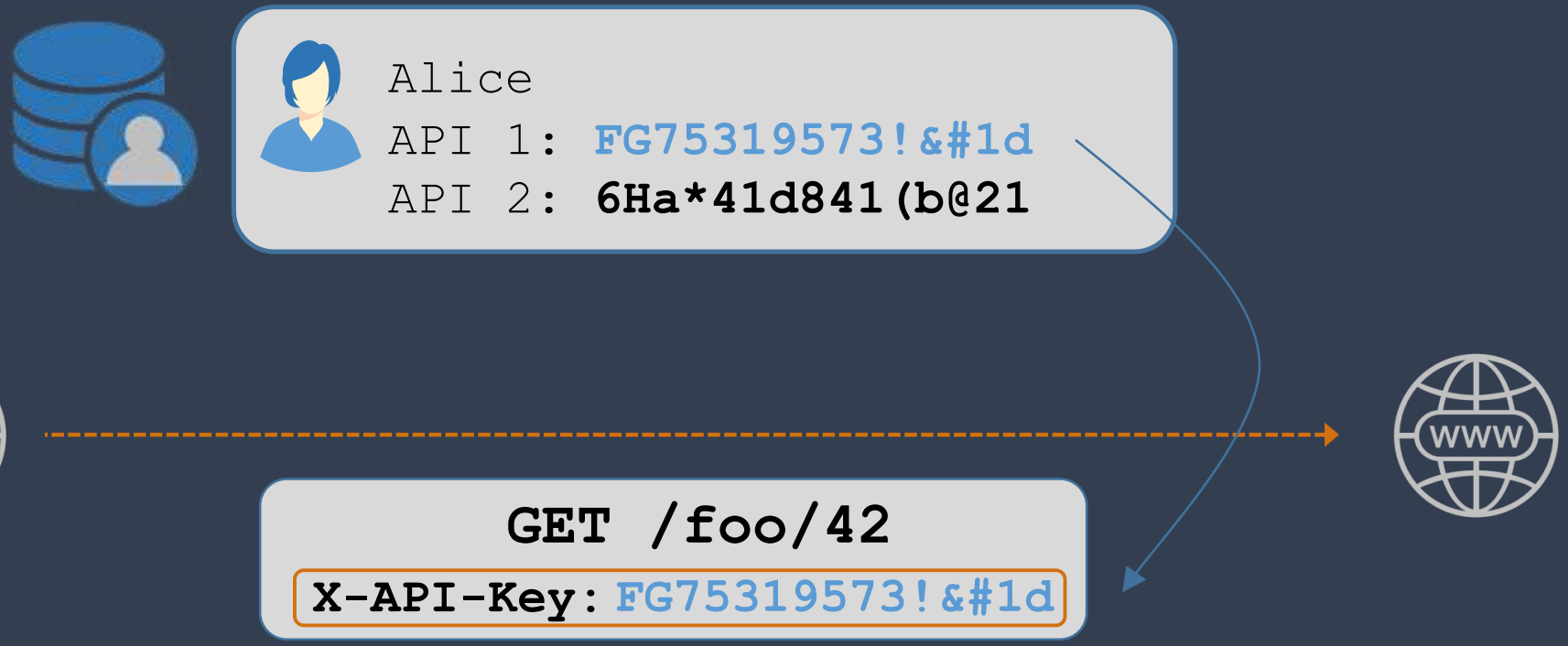
Authentication

API Keys as **passwords**



Authentication

API Keys as **passwords**



If you only pass an API Key, then
it **must be unique** within your system

Authentication

API Keys as **cryptographic keys**

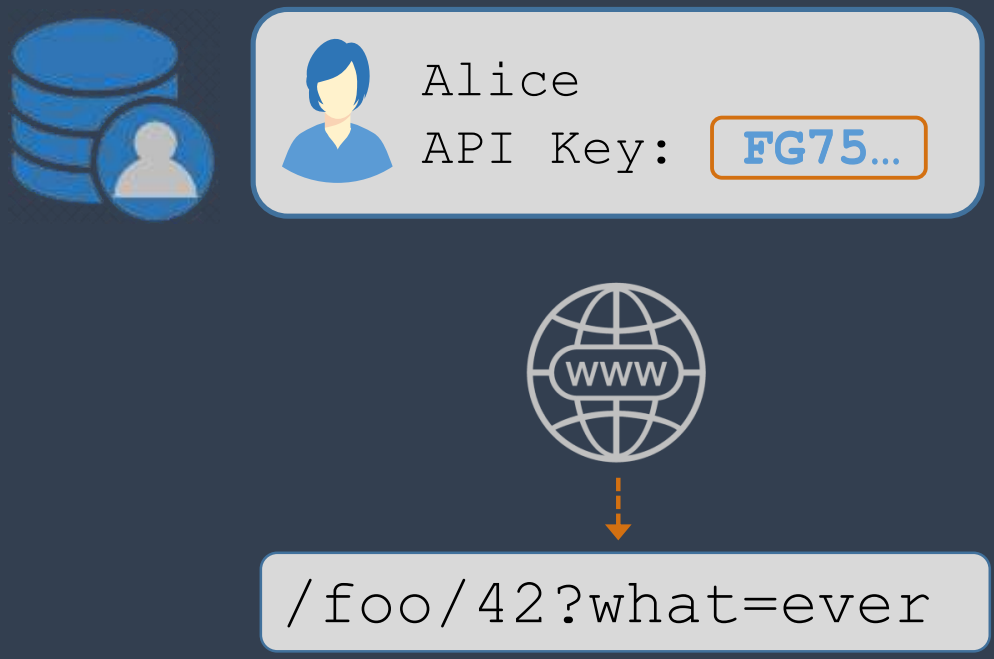
Authentication

API Keys as **cryptographic keys**

Alice
API Key: `FG75319573!&#1d`

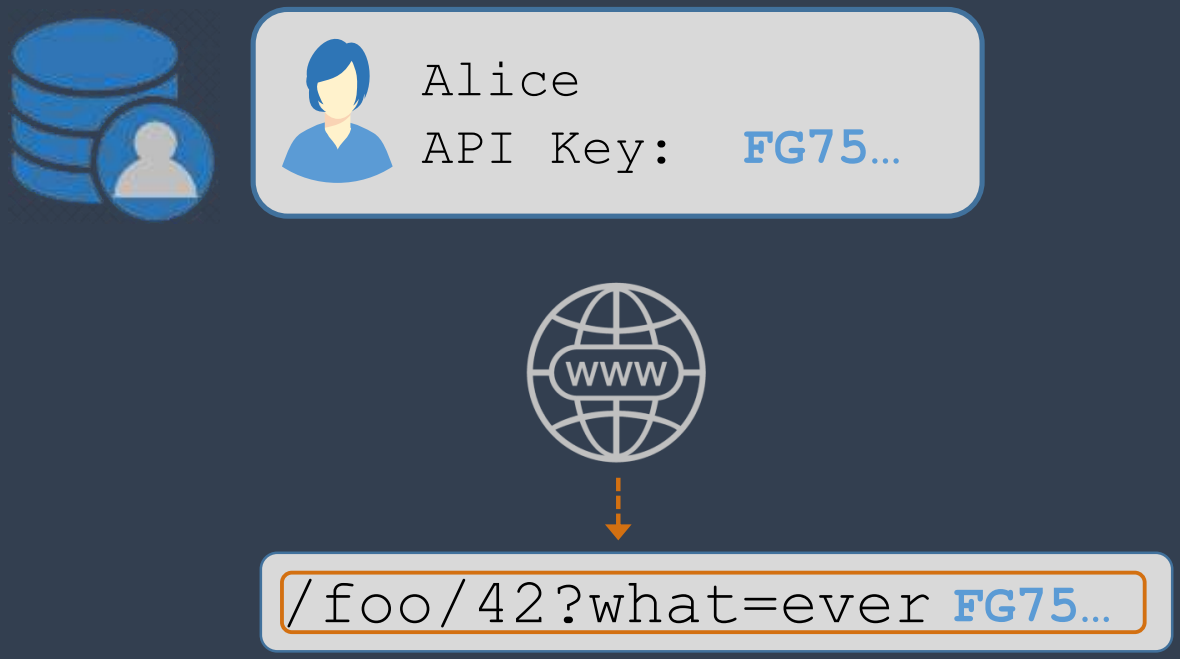
Authentication

API Keys as **cryptographic keys**



Authentication

API Keys as **cryptographic keys**



Authentication

API Keys as **cryptographic keys**

```
Alice
API Key:   FG75…
```

HASH(/foo/42?what=ever **FG75…**)

Authentication

API Keys as **cryptographic keys**

Authentication

API Keys as **cryptographic keys**

Alice
API Key: FG75…

HASH(/foo/42?what=ever FG75…)
8aa53271528…

GET /foo/42?what=ever
X-Sig: 8aa53271528…

The API Key isn't included in the request!

Authentication

API Keys as **cryptographic keys**

```
GET /foo/42?what=ever
X-Sig: 8aa53271528…
```

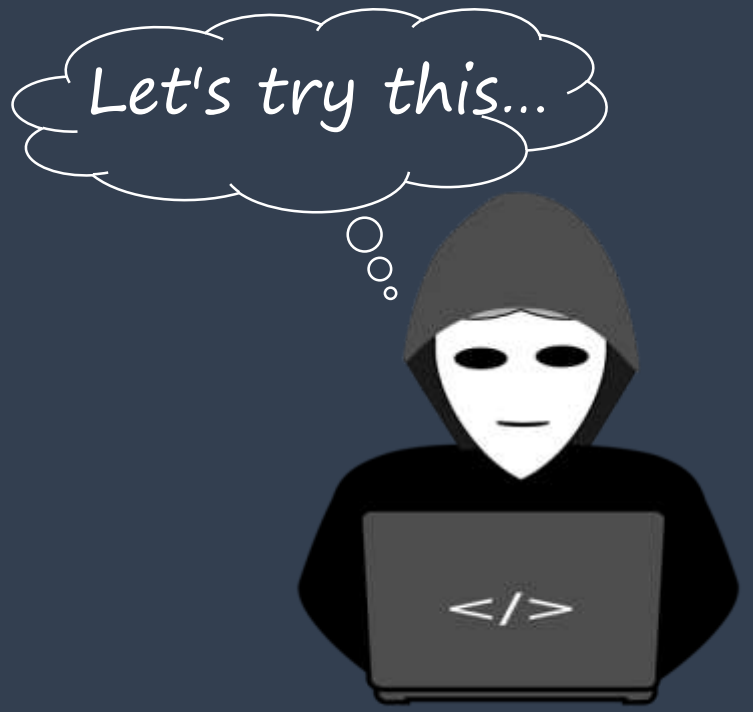
```
GET /foo/42?what=evs
X-Sig: 8aa53271528…
```

Authentication

API Keys as **cryptographic keys**

```
GET /foo/42?what=evs
X-Sig: 8aa53271528…
```

```
Alice
API Key:  FG75…
```



Authentication

API Keys as **cryptographic keys**

GET /foo/42?what=**evs**
X-Sig: 8aa53271528…

Alice
API Key: FG75…

HASH(/foo/42?what=**evs** FG75… **)**
↳ VeZxG76M4mKp…

Authentication

API Keys as **cryptographic keys**

Authentication

API Keys as **cryptographic keys**

```
GET /foo/42?what=ever
X-Sig: 8aa53271528…
```

```
Alice
API Key:  FG75…
```

How does the server know **which** user API key to validate with?

Authentication

API Keys as **cryptographic keys**

```
GET /foo/42?what=ever
Authorization: ApiKey <id>:<sig>
```

Alice
API Key: FG75…

Authentication

API Keys as **cryptographic keys**

```
GET /foo/42?what=ever
Authorization: ApiKey <id>:<sig>
```

Could be some primary key in your system, like email or USER_ID

Alice
API Key: FG75…

Authentication

API Keys as **cryptographic keys**

```
GET /foo/42?what=ever
Authorization: ApiKey <id>:<sig>
```

```
Alice
API Key1: FG75…
API Key2: G72n…
API Key3: a904…
```

But then it's hard to support multiple keys per user

Authentication

API Keys as **cryptographic keys**

```
GET /foo/42?what=ever
Authorization: ApiKey <key_id>:<sig>
```

Alice

JF3Am:	FG75…
nP4!g:	G72n…
tHa35:	a904…

Track API Keys as a **key pair**:
Pass the public ID with the request
Private key used to sign requests

Key IDs **uniquely identify the API Key**

Authentication

API Keys as **cryptographic keys**

```
GET /foo/42?what=ever
Authorization: ApiKey <key_id>:<sig>
```

```
Alice
    JF3Am: FG75…
    nP4!g: G72n…
    tHa35: a904…
```

Track API Keys as a **key pair**:
Pass the public ID with the request
Private key used to sign requests

Key IDs **uniquely identify the API Key**

Authentication

API Keys as **cryptographic keys**

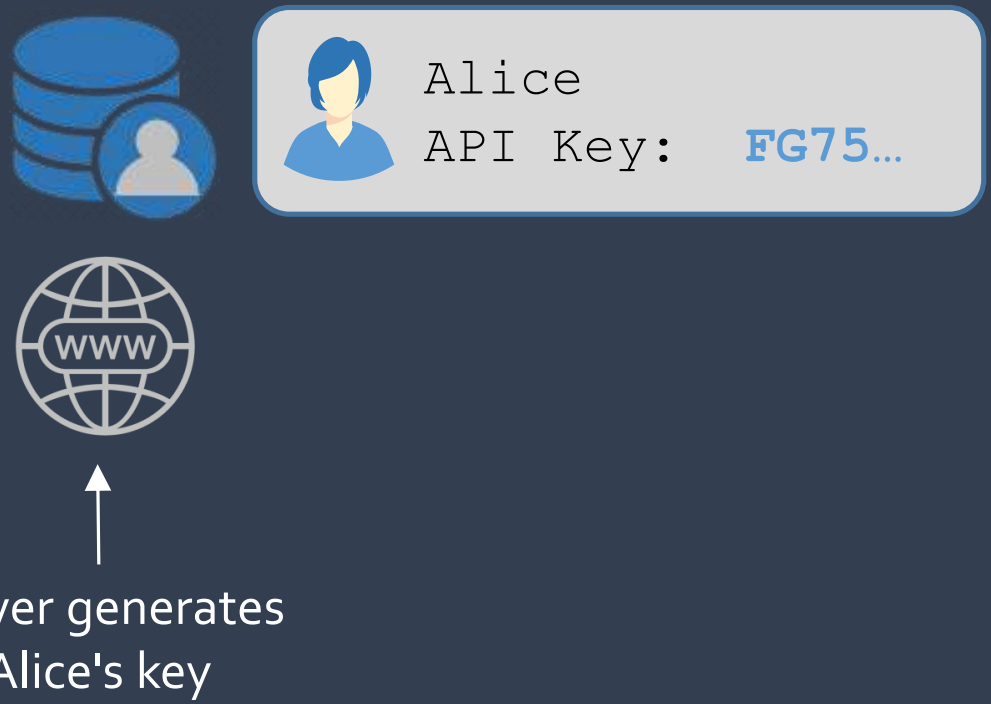
Hash-based **M**essage **A**uthentication **C**ode
(HMAC)

Authentication API Keys

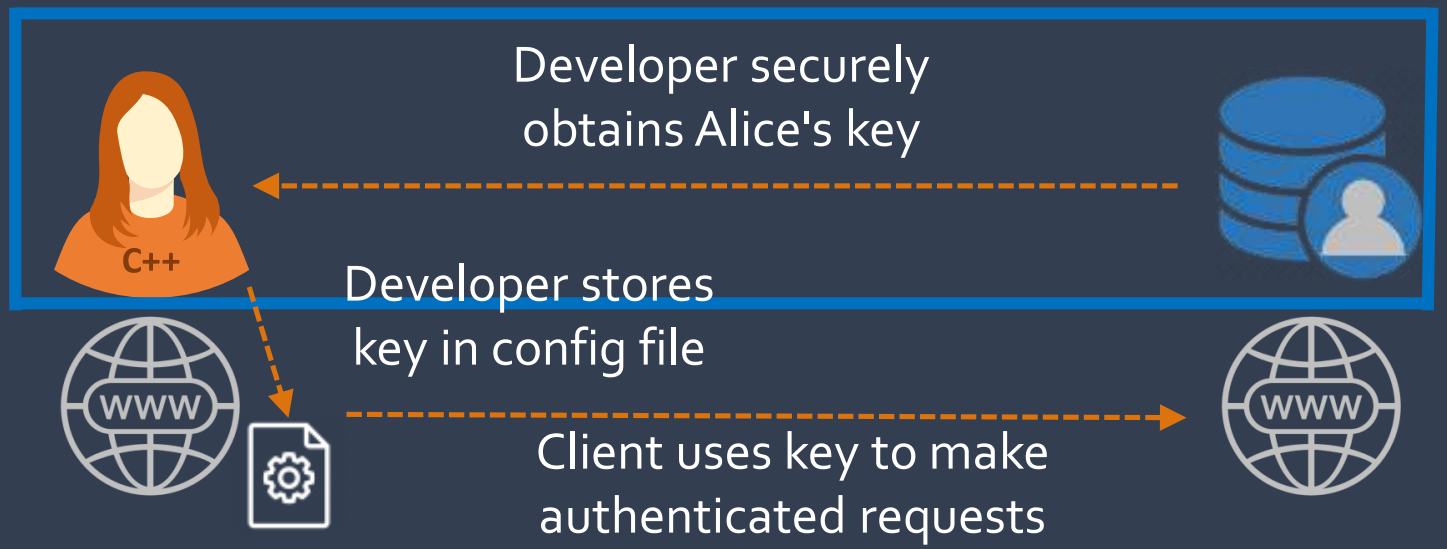
To use API Keys, the client must be able to **securely store** the key

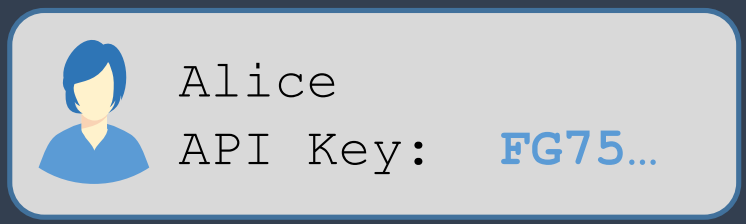
Authentication API Keys

To use API Keys, the client must be able to **securely store** the key



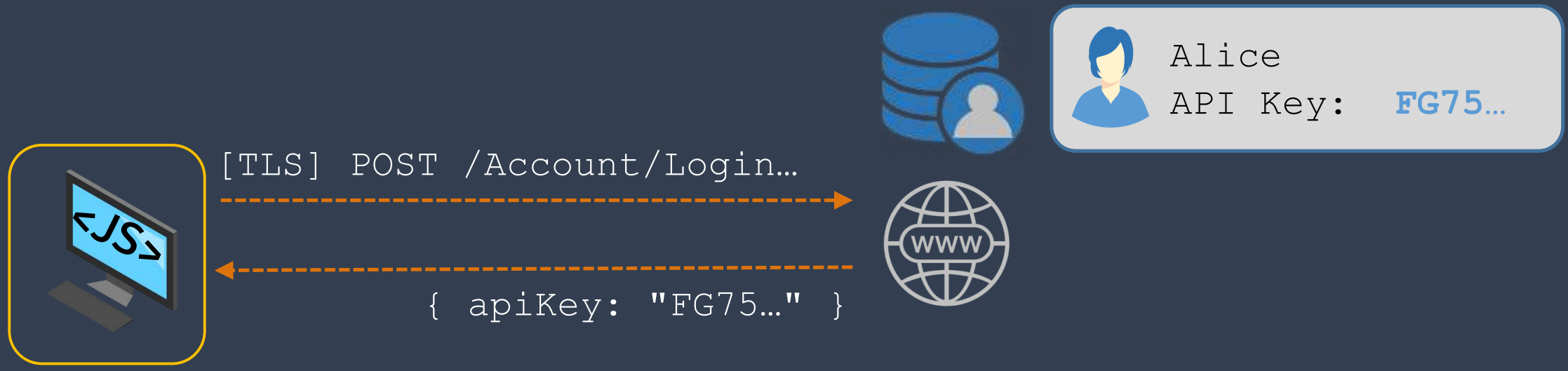
Authentication API Keys





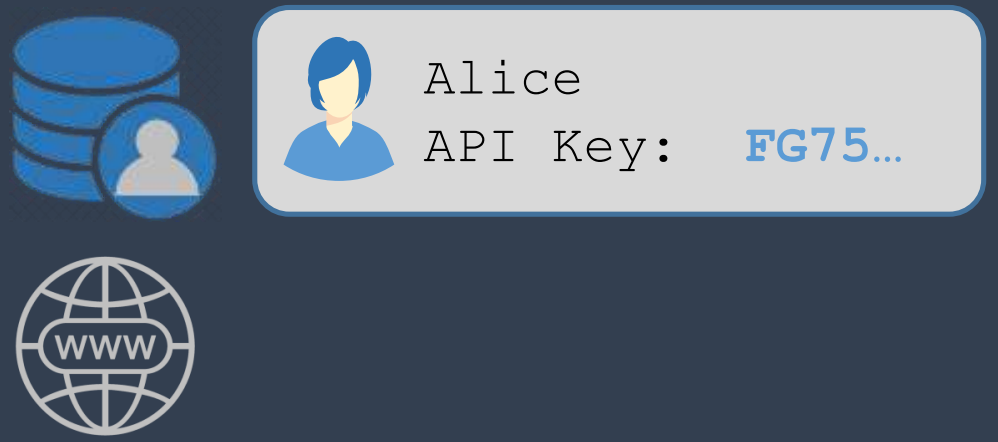
To use API Keys, the client must be able to **securely store** the key

Authentication API Keys



JavaScript client
can't pre-load the API key

Authentication API Keys



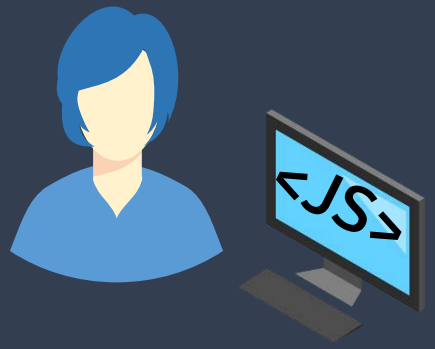
JavaScript is an insecure client;
it **cannot** securely store the key!

JSON Web Tokens

JWT / "Jot" / "Jot tokens"

JSON Web Tokens

JWT / "Jot" / "Jot tokens"



[TLS] POST /Account/Login…

Private Key: b8wM…

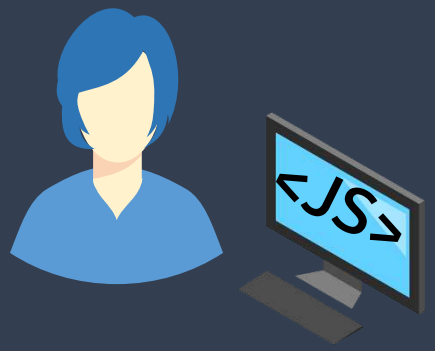
```
{
 auth_OK: true,
 user: "Alice"
}
```

"Claims" are just pieces of data

JSON Web Tokens JWT / "Jot" / "Jot tokens"

JSON Web Tokens

JWT / "Jot" / "Jot tokens"



[TLS] POST /Account/Login…

Private Key: b8wM…

```
{
 auth_OK: true,
 user: "Alice"
}
```

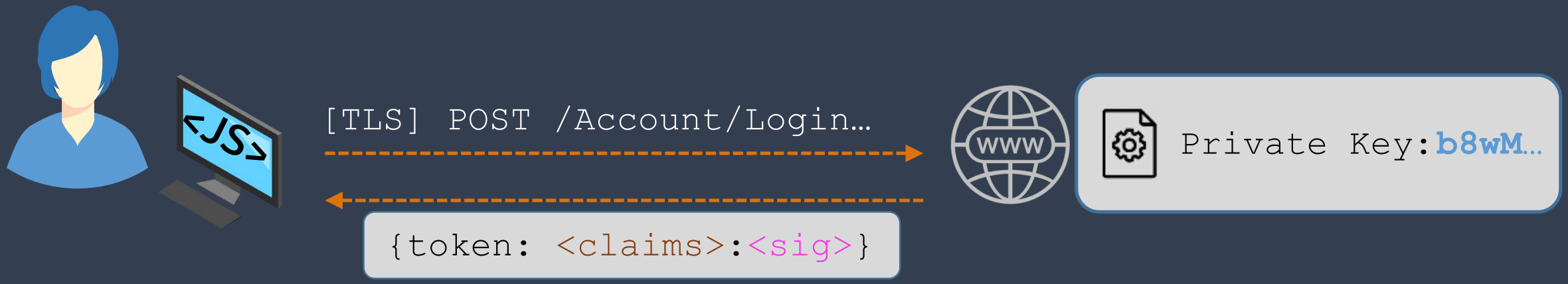
HASH(<claims>:<key>)

↳ **m46e9vb3F28…**

{token: <claims>:<sig>}

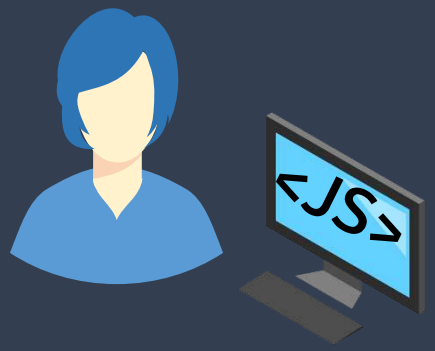
JSON Web Tokens

JWT / "Jot" / "Jot tokens"



JSON Web Tokens

JWT / "Jot" / "Jot tokens"



[TLS] POST /Account/Login…

Private Key: b8wM…

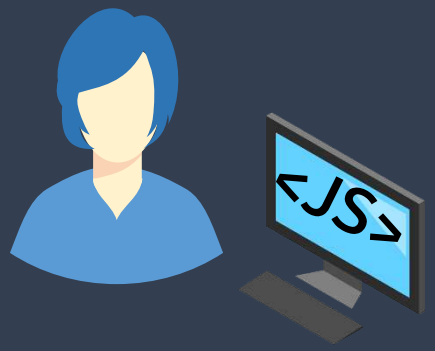
HASH(<claims>:<key>)

m46e9vb3F28…

GET /foo/42
{token: <claims>:<sig>}

JSON Web Tokens

JWT / "Jot" / "Jot tokens"



[TLS] POST /Account/Login…

Private Key: b8wM…

```
 GET /foo/42
{token: <claims>:<sig>}
```

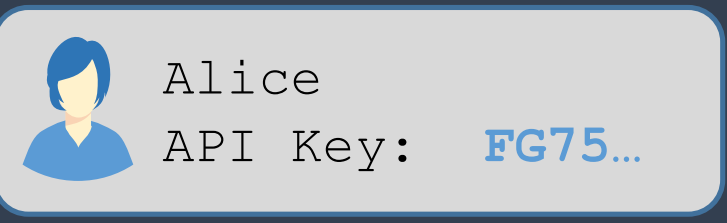
m46e9vb3F28…

```
{
 auth_OK: true,
 user: "Alice"
}
```

Authentication

JWT / "Jot" / "Jot tokens"

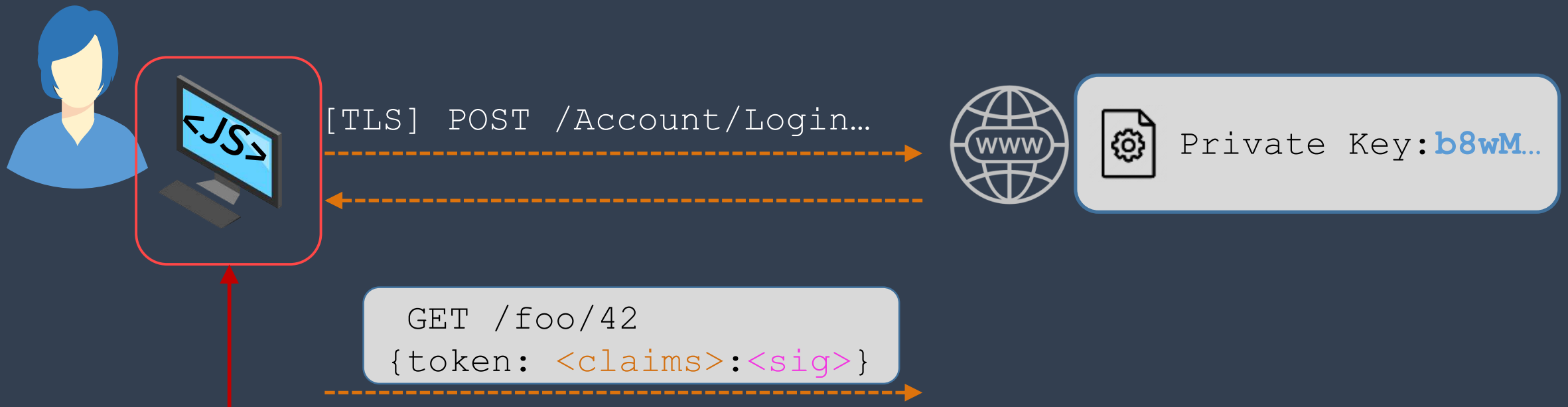

```
{ apiKey: "FG75…" }
```



JavaScript is an insecure client;
it **cannot** securely store the key!

JSON Web Tokens

JWT / "Jot" / "Jot tokens"



JavaScript is an insecure client;
how does it securely store the token?

JSON Web Tokens

JWT / "Jot" / "Jot tokens"

```
{
 user: "Alice"
 isAdmin: true
}
```

Anyone that can view the token
can view the claims

Anyone that can view the token
can replay the token

JSON Web Tokens

JWT / "Jot" / "Jot tokens"

```
{
 user: "Alice"
 isAdmin: true
}
```

Not exposed to JS

```
Set-Cookie: jwt={token}; Secure; HttpOnly;
```

Only sent over HTTPS

JSON Web Tokens

JWT / "Jot" / "Jot tokens"

```
{
 user: "Alice"
 isAdmin: true
}
```

```
{
 currentTheme: "dark",
 hasVerifiedAddr: true
}
```

```
jwt={token}; Secure; HttpOnly;
```

Server-side code uses this token

JSON Web Tokens

JWT / "Jot" / "Jot tokens"

```
{
 user: "Alice"
 isAdmin: true
}
```

```
jwt={token}; Secure; HttpOnly;
```

Server-side code uses this token

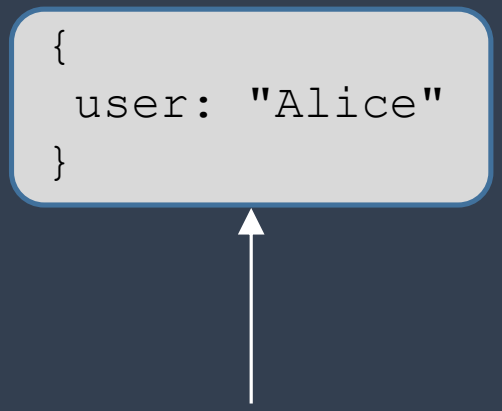
```
{
 currentTheme: "dark",
 hasVerifiedAddr: true
}
```

Put non-sensitive stuff into **LocalStorage** where JS can get to it

Client-side code uses this token

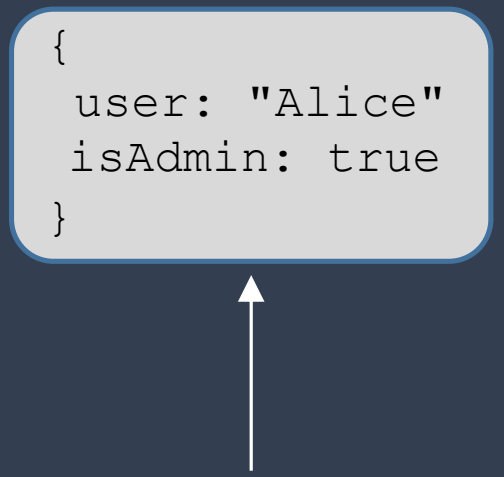
JSON Web Tokens

JWT / "Jot" / "Jot tokens"



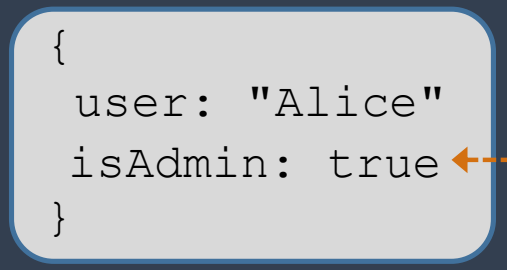
JSON Web Tokens

JWT / "Jot" / "Jot tokens"



JSON Web Tokens

JWT / "Jot" / "Jot tokens"



What if Alice's permissions *change* after the token is created?

JSON Web Tokens

JWT / "Jot" / "Jot tokens"

```
{
 user: "Alice"
 isAdmin: true
 expiry: {date}
}
```

What if Alice's permissions *change* after the token is created?

JSON Web Tokens

JWT / "Jot" / "Jot tokens"

```
{
 user: "Alice"
 isAdmin: true
 expiry: {date}
}
```

What if Alice's permissions *change* after the token is created?

JSON Web Tokens

Refresh tokens

```
{
 user: "Alice"
 isAdmin: true
 expiry: {short}
}
```

Short-lived **access token**

Server trusts token for auth, **without** hitting database

```
{
 user: "Alice"
 expiry: {long}
}
```

Long-lived **refresh token**

JSON Web Tokens

Refresh tokens

```
{
 user: "Alice"
 isAdmin: true
 expiry: {short}
}
```

Short-lived **access token**

Server trusts token for auth,
without hitting database

```
{
 user: "Alice"
 expiry: {long}
}
```

Long-lived **refresh token**

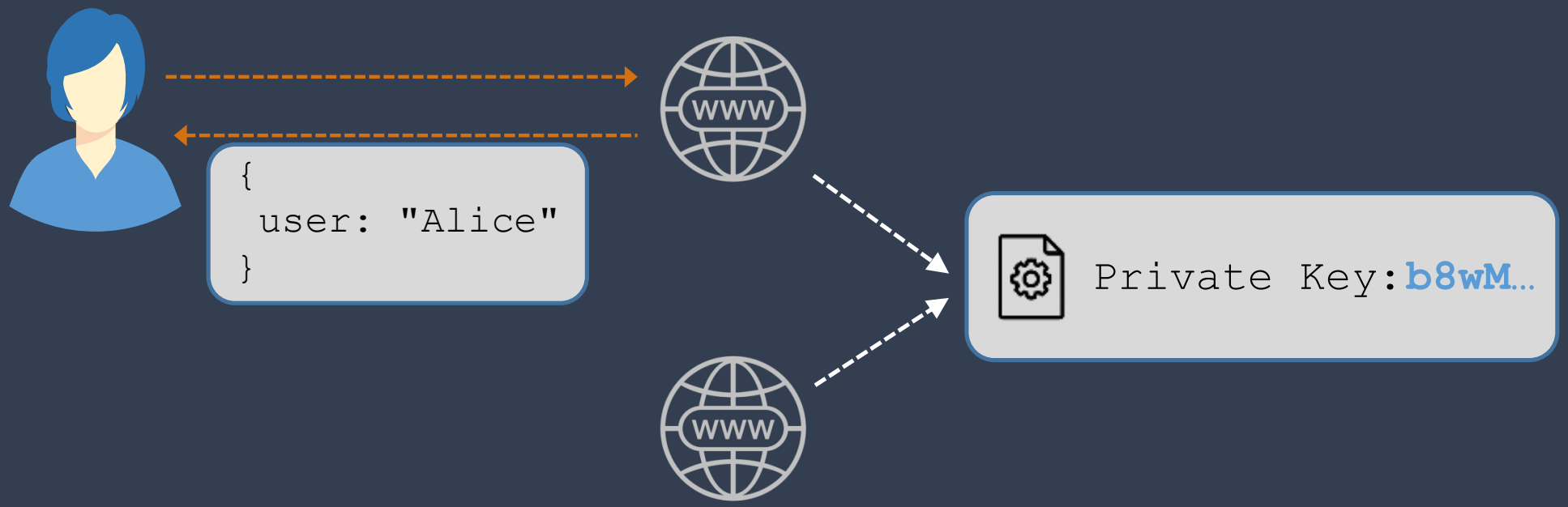
Server **re-queries** database,
then issues a new access token

JSON Web Tokens

Web tokens for SSO

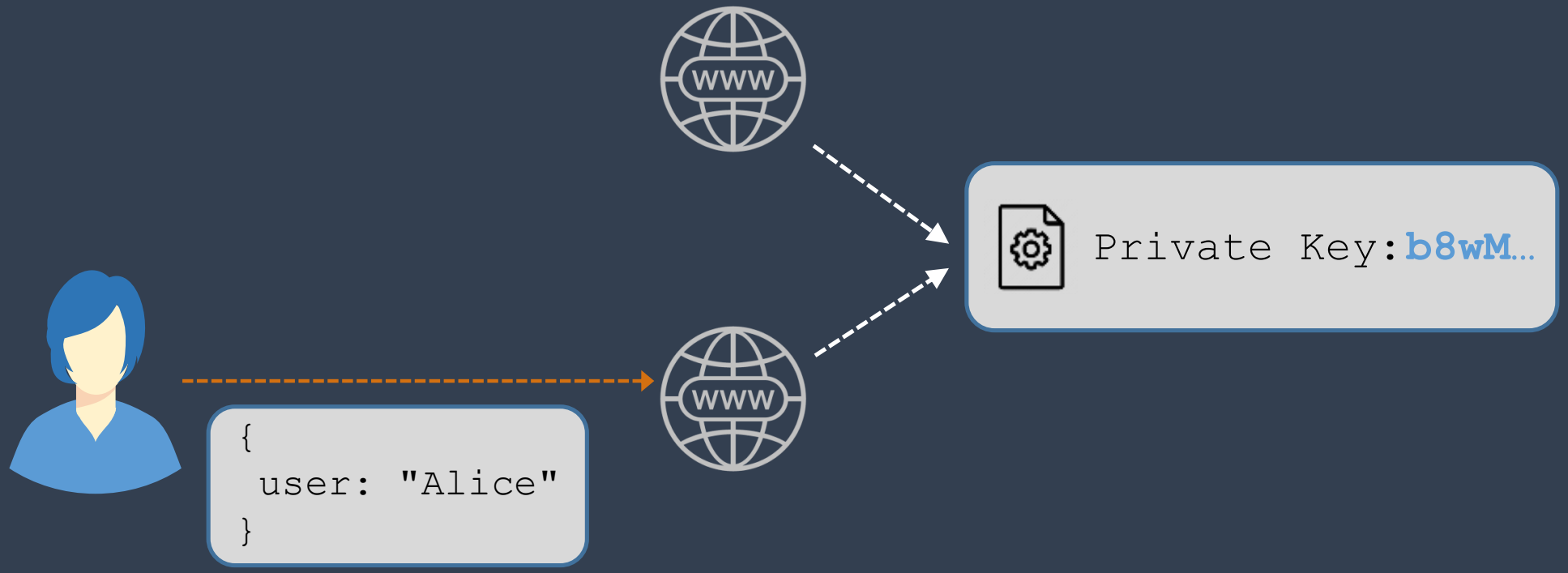
JSON Web Tokens

Web tokens for SSO



JSON Web Tokens

Web tokens for SSO



Agenda

Passwords
Basic Auth
API Keys as bearer tokens
API Keys as cryptographic keys
JSON Web Tokens
SAML
OAuth
OpenID Connect
Authenticator Apps
Passkeys

Credential-based authentication

Claims-based authentication

Agenda

Passwords
Basic Auth
API Keys as bearer tokens
API Keys as cryptographic keys
JSON Web Tokens
SAML
OAuth
OpenID Connect
Authenticator Apps
Passkeys

Credential-based authentication

Claims-based authentication

SAML

SAML is like an enterprise-grade,
XML version of JSON Web Tokens

SAML

Service Provider (SP)

Alice wants to **use** the **service provider**

Identity Provider (IdP)

Alice can **authenticate** to the **identity provider**

SAML

Service Provider (SP)

Identity Provider (IdP)

Alice accesses the SP without an authenticated session

SAML

Service Provider (SP)

Identity Provider (IdP)

Alice redirected to Identity Provider, where she logs in

SAML

Service Provider (SP)

Identity Provider (IdP)

Identity Provider creates
signed claims
stating her identity

```
{ email: alice@… }
```

SAML

Service Provider (SP)

Identity Provider (IdP)

{ email: alice@… }

Alice accesses Service Provider

Signed claims are verified and used to authenticate her

SAML

Service Provider (SP)

Identity Provider (IdP)

There is no shared private key!

How does token validation work?

SAML

Service Provider (SP)

Identity Provider (IdP)

SAML uses
public key cryptography
instead of shared private keys

SAML

Service Provider (SP)

Identity Provider (IdP)

Public key cryptography

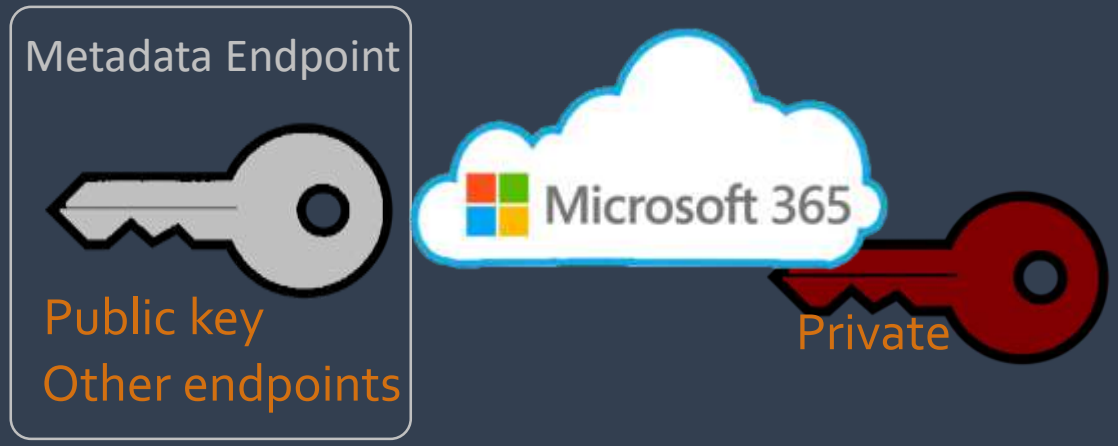
A message **signed** with this…

… can be **verified** with this

SAML Public key cryptography

Service Provider (SP)

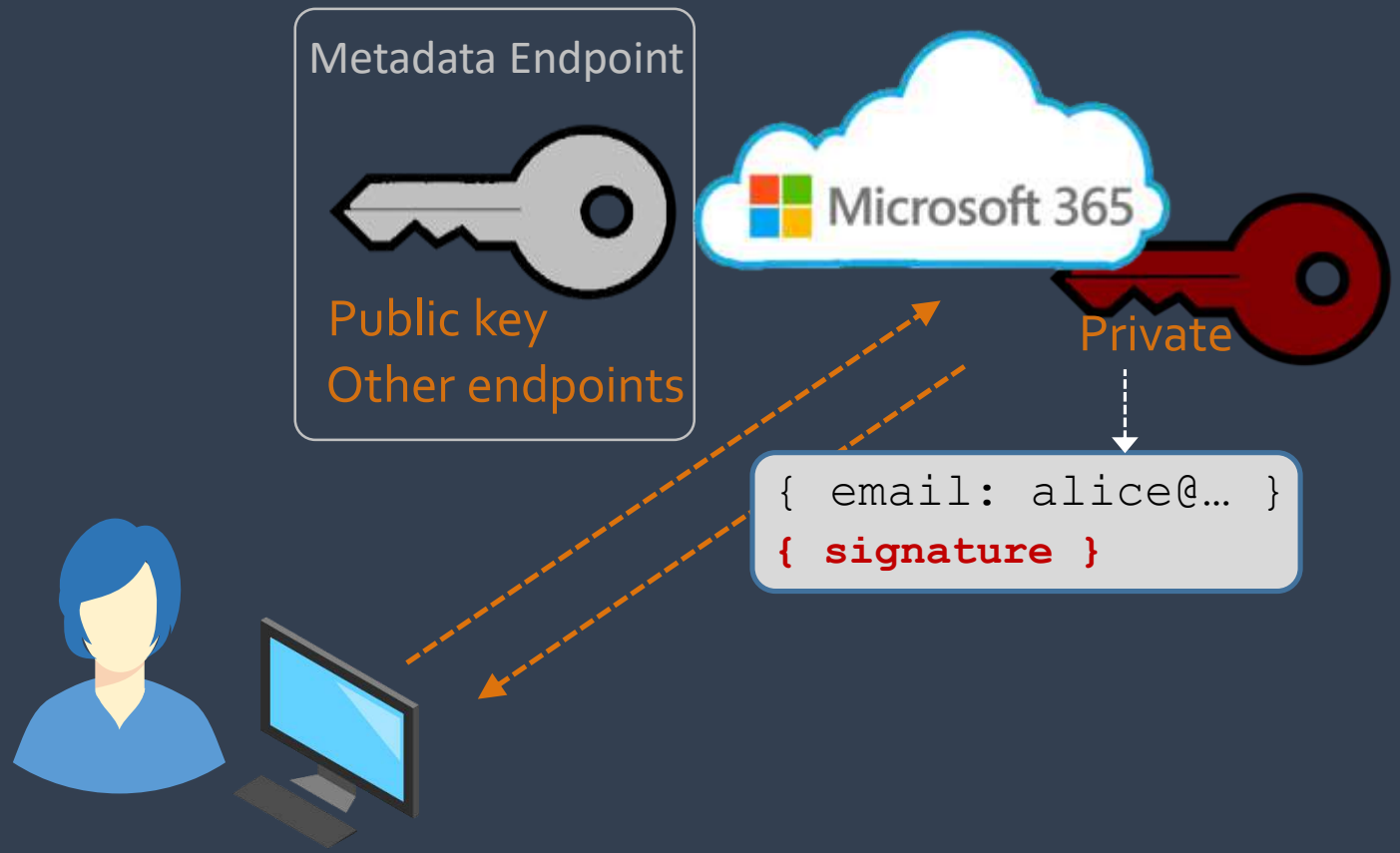
Identity Provider (IdP)



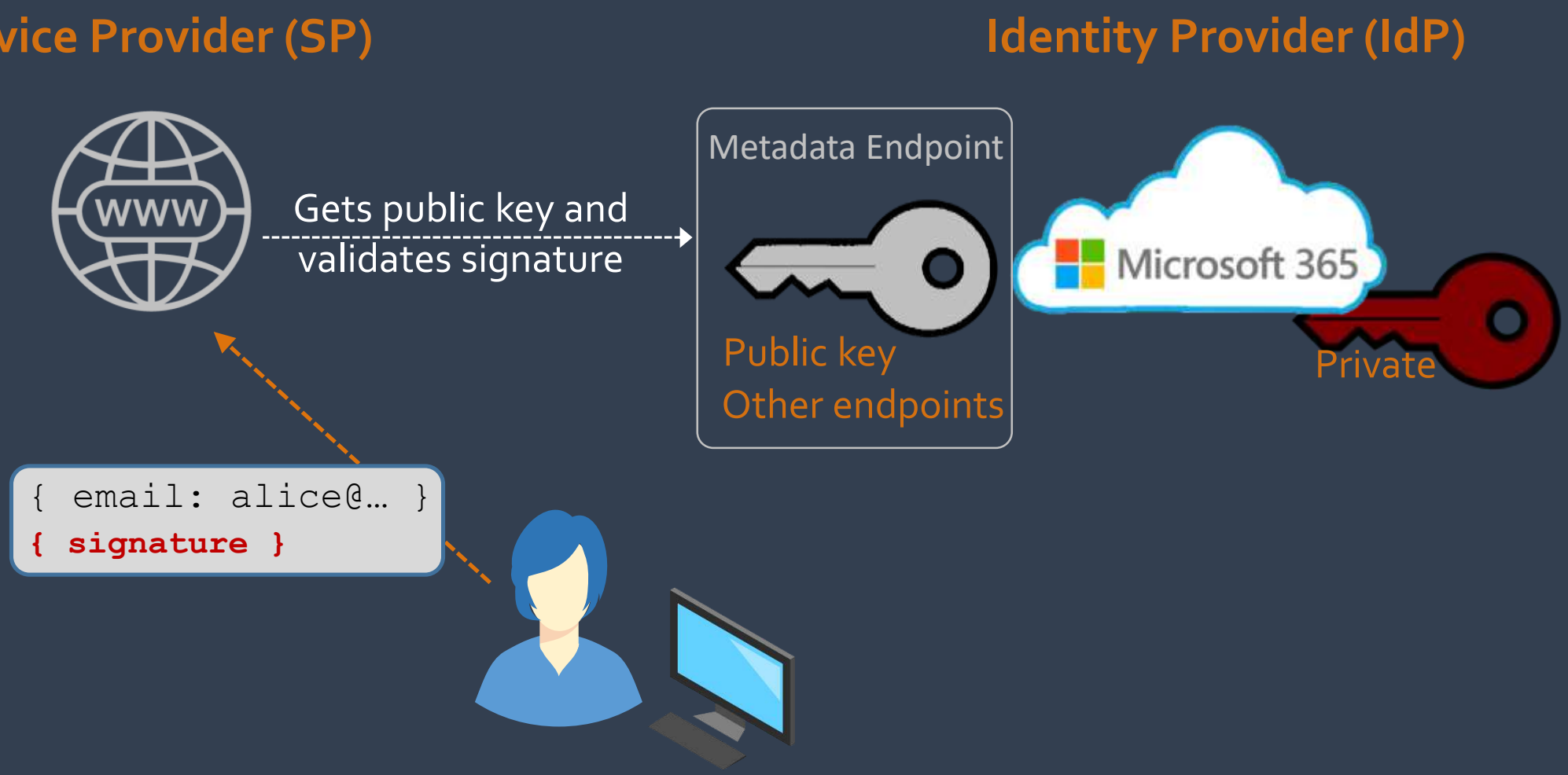
SAML Public key cryptography

Service Provider (SP)

Identity Provider (IdP)



SAML Public key cryptography



SAML

Service Provider (SP)

Nothin' much to do here!

Identity Provider (IdP)

Multi-factor authentication [MFA]

- Password
- Authenticator App
- Key card
- Email access validation
- "Hey, Bob said I could log in, is that cool?"

OAuth 2.0

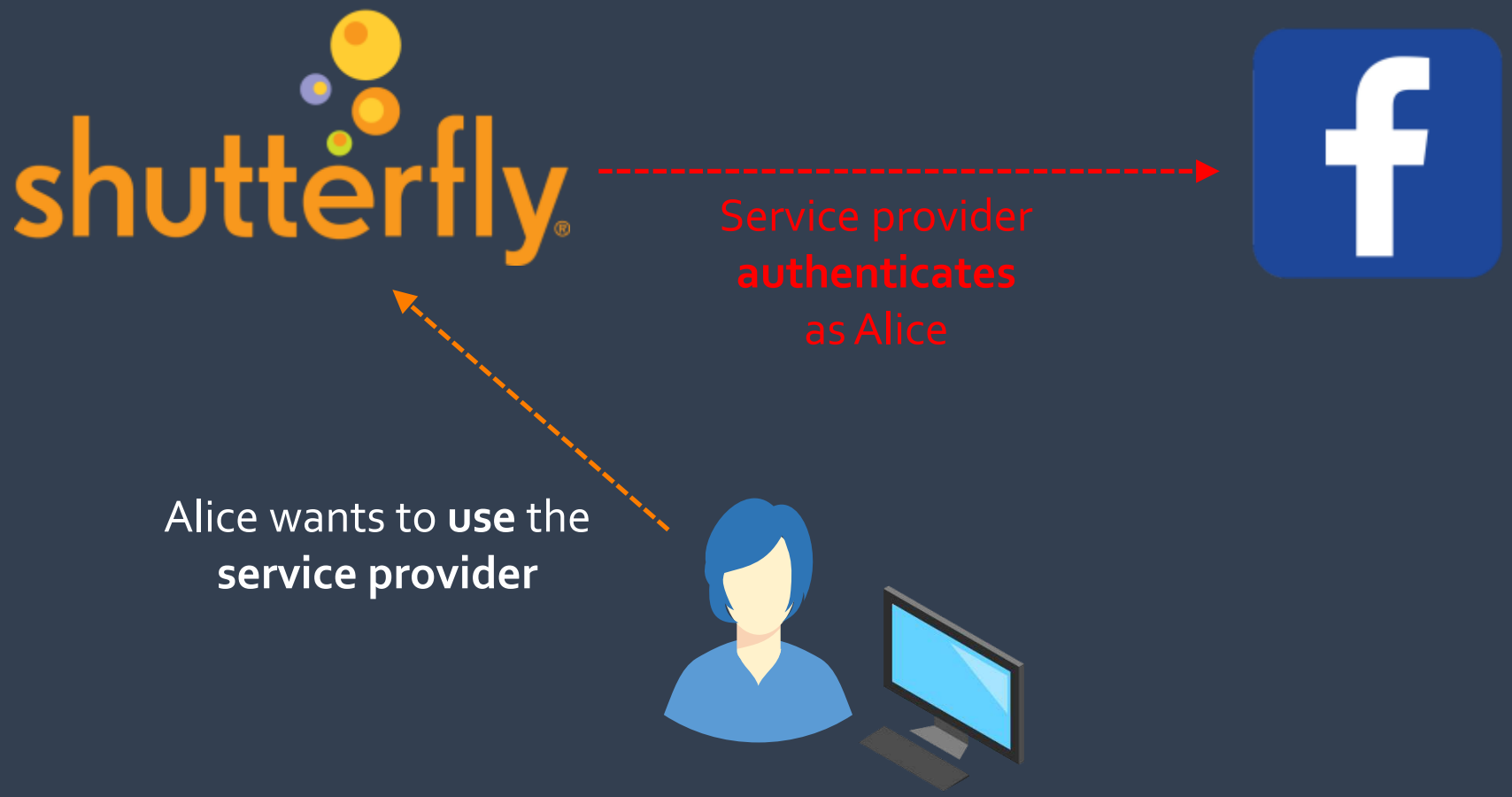
Service Provider (SP)

Resource Server

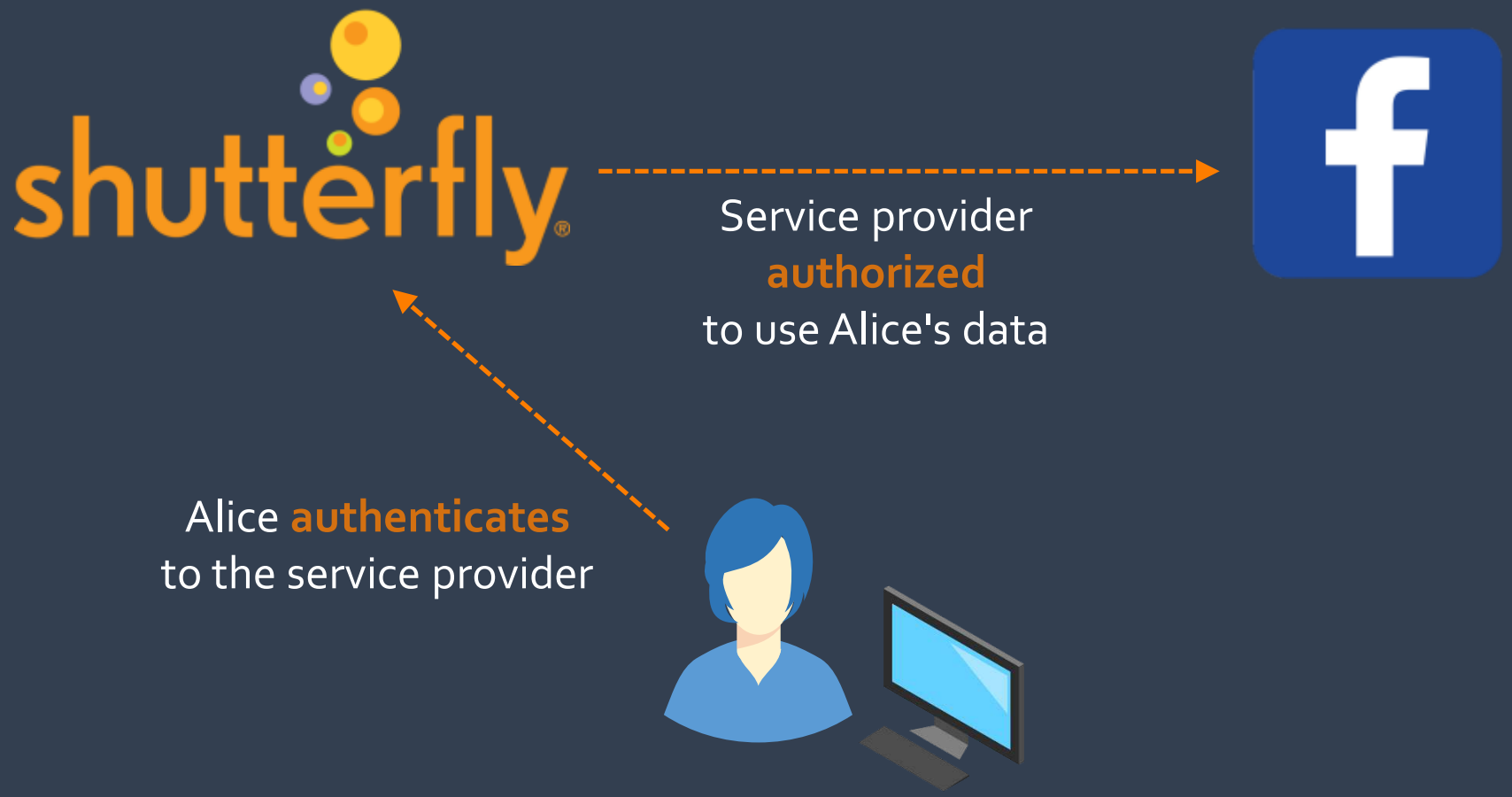
Service provider accesses
Alice's data in 3rd party system

Alice wants to **use** the
service provider

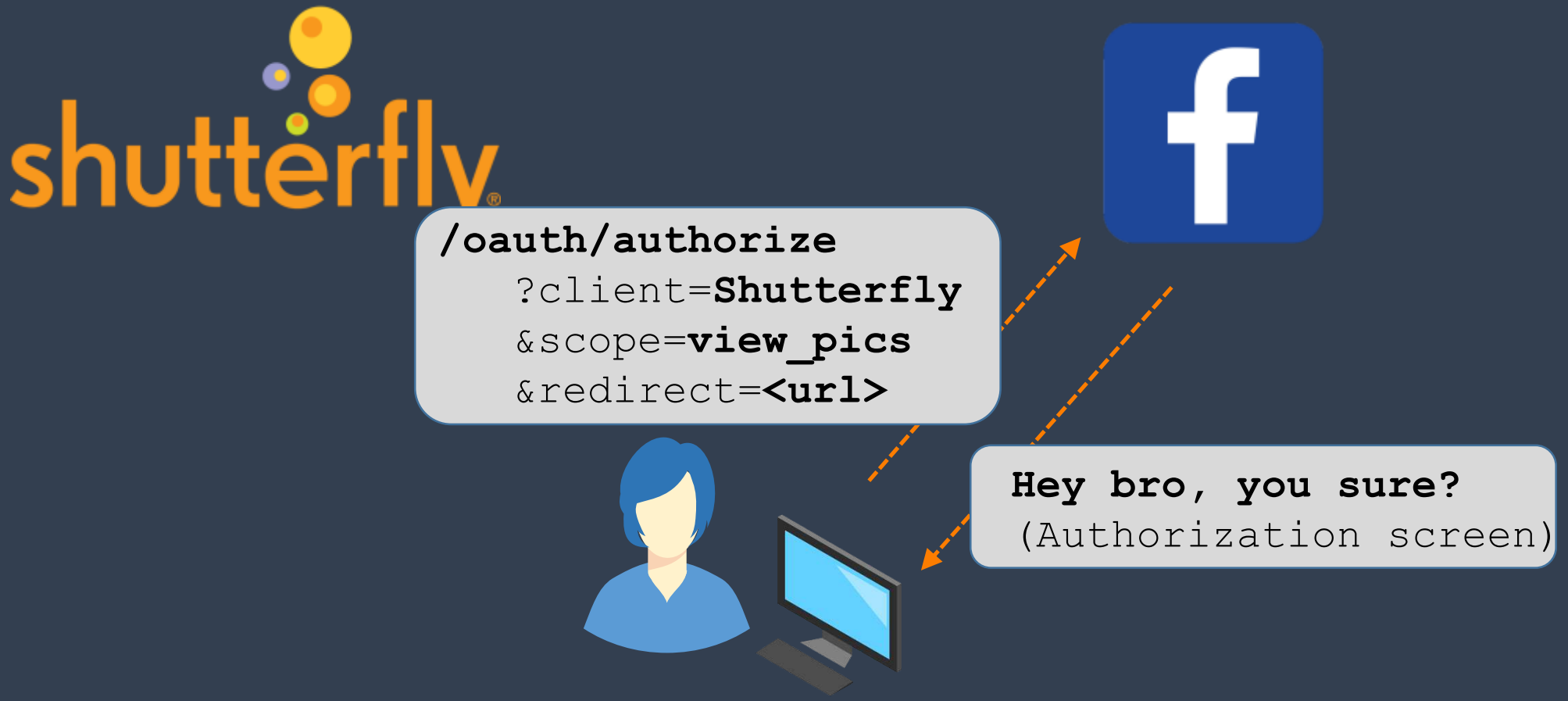
OAuth 2.0


shutterfly
Service provider
authenticates
as Alice
Alice wants to **use** the
service provider

OAuth 2.0


shutterfly
Service provider
authorized
to use Alice's data
Alice authenticates
to the service provider

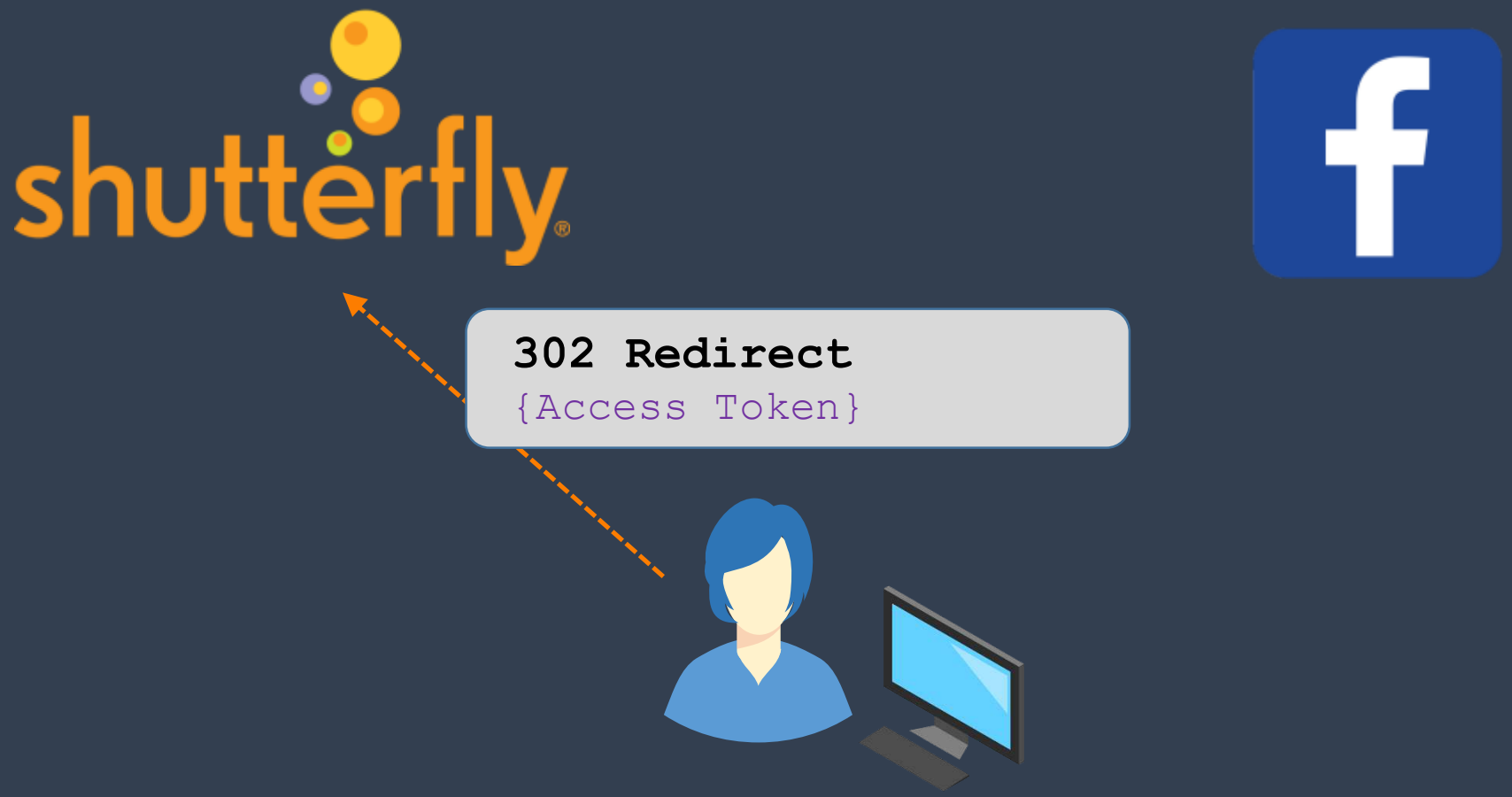
OAuth 2.0


shutterfly
/oauth/authorize
?client=Shutterfly
&scope=view_pics
&redirect=<url>
Hey bro, you sure?
(Authorization screen)

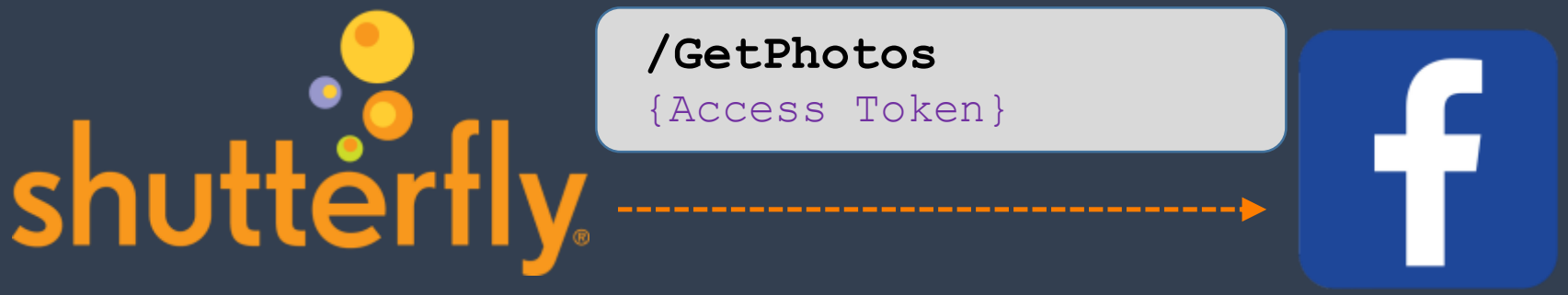
OAuth 2.0

shutterfly

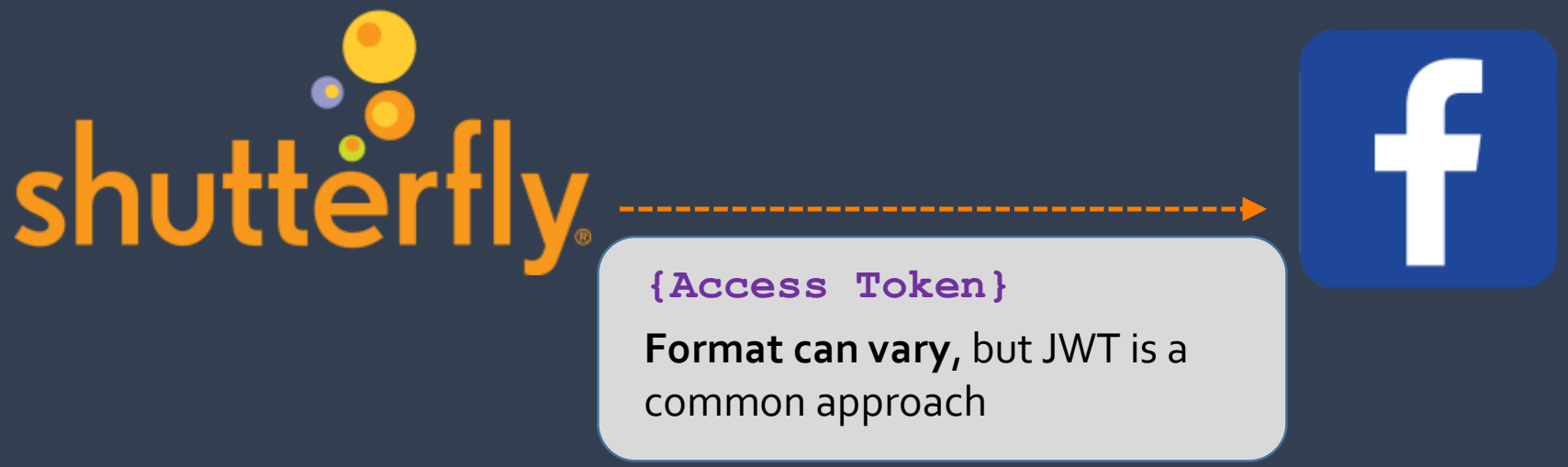
OAuth 2.0



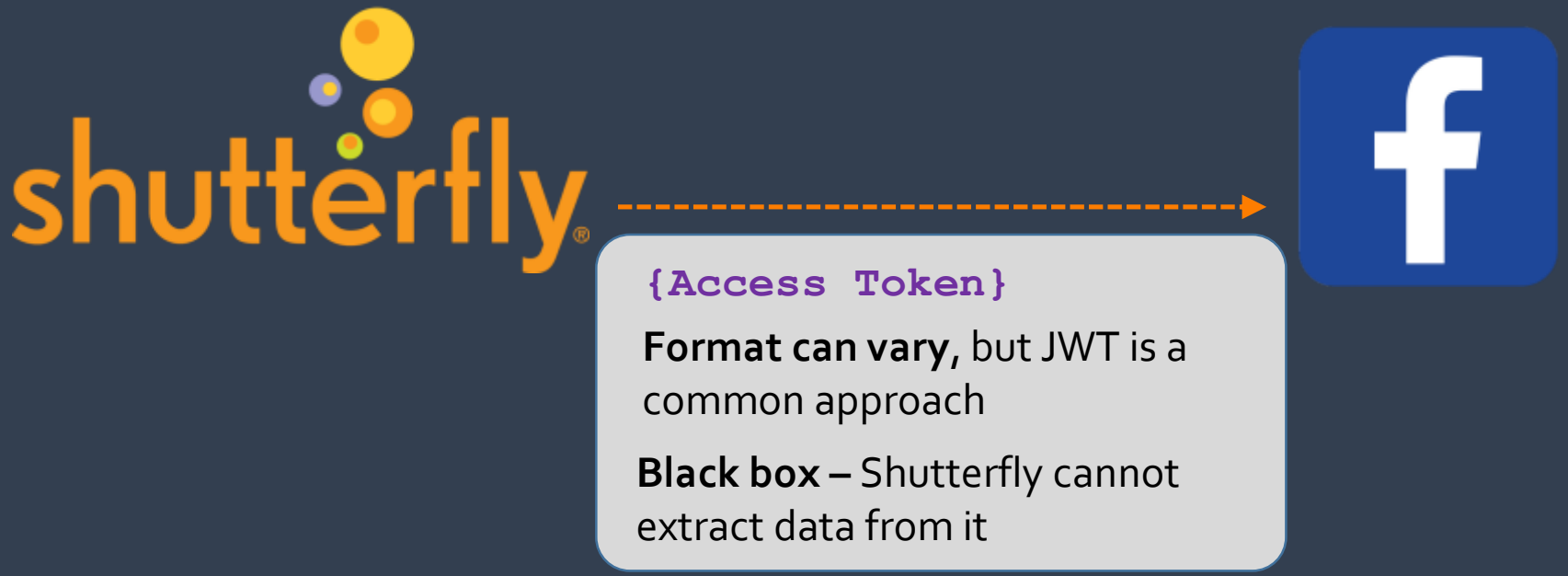
OAuth 2.0



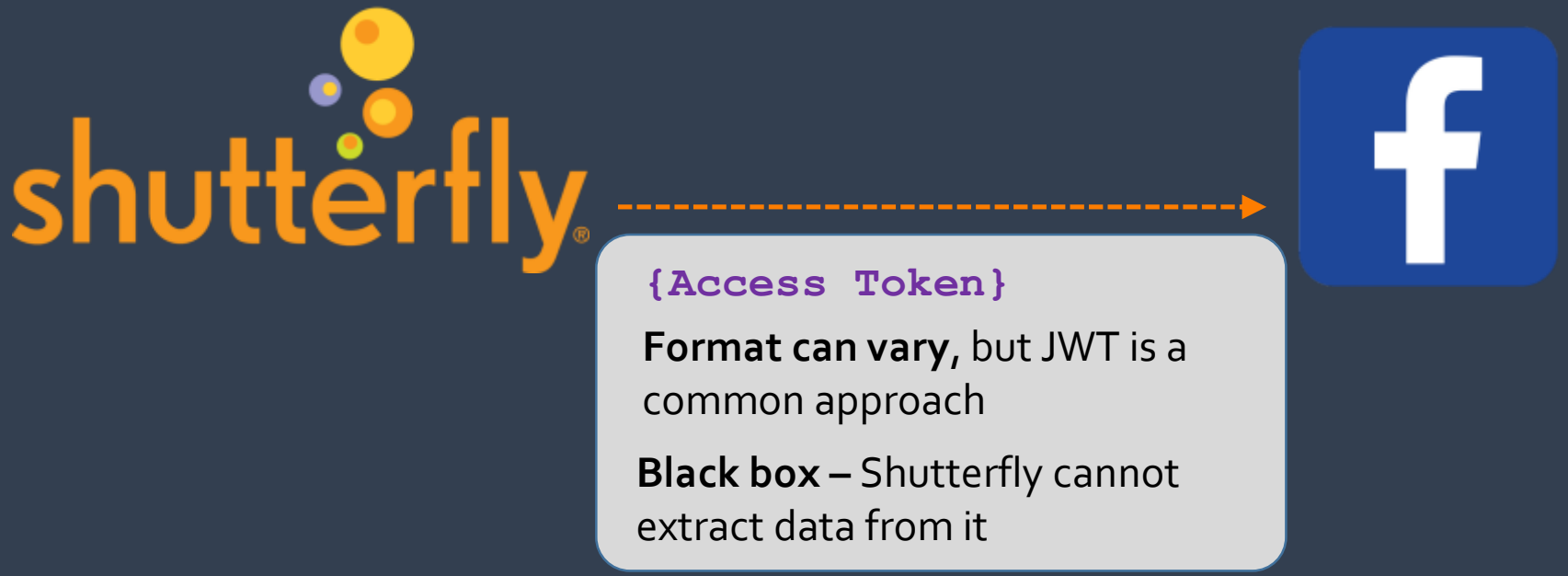
OAuth 2.0


shutterfly
{Access Token}
Format can vary, but JWT is a common approach

OAuth 2.0


shutterfly
{Access Token}
Format can vary, but JWT is a common approach
Black box – Shutterfly cannot extract data from it

OAuth is **not** an **authentication protocol!**



OAuth is **not** an **authentication protocol!**

`{Access Token}`

OAuth access tokens
DO NOT allow
"Log in with Facebook"

OAuth is **not** an **authentication protocol!**

`{Access Token}`

OAuth access tokens
DO NOT allow
"Log in with Facebook"

WHO authorized the token?

HOW was it authorized?

WHICH service was the token issued to?

OpenID Connect **is** an authentication protocol!

{Access Token}

{OIDC Token}

WHO authorized the token?

HOW was it authorized?

WHICH service was the token issued to?

OpenID Connect **is** an authentication protocol!

{Access Token}

{OIDC Token} ----------→

WHO authorized the token?

HOW was it authorized?

WHICH service was the token issued to?

OpenID Connect DOES allow "Log in with Facebook"

Token format for securely exchanging data

Authorization protocol allowing one app to securely access a user's data in another app

Identity protocol built on top of OAuth 2.0

Identity protocol using XML and SOAP

Agenda

Passwords
Basic Auth
API Keys as bearer tokens
API Keys as cryptographic keys
JSON Web Tokens
SAML
OpenID Connect
OAuth
Authenticator Apps
Passkeys

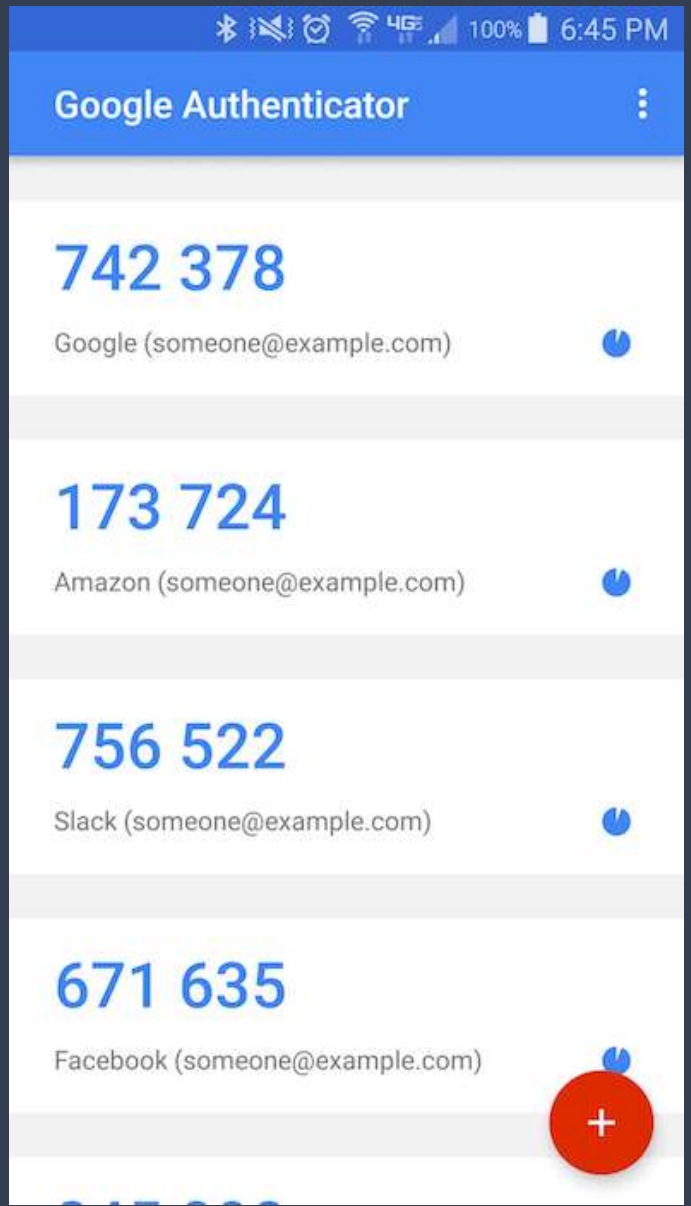
Credential-based authentication

Claims-based identity

Authorization, not authentication

Authenticator Apps

Time-based One-Time Password (TOTP)



Authenticator Apps

Time-based One-Time Password (TOTP)

Authenticator Apps

Time-based One-Time Password (TOTP)

Authenticator Apps

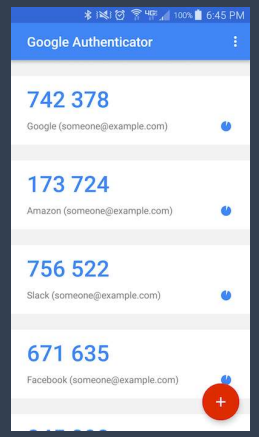
Time-based One-Time Password (TOTP)

Authenticates

```
otpauth://totp/<issuer>:<email>
  ?secret={secret}&issuer=<issuer>
  &digits=6
```

Authenticator Apps

Time-based One-Time Password (TOTP)



{secret}

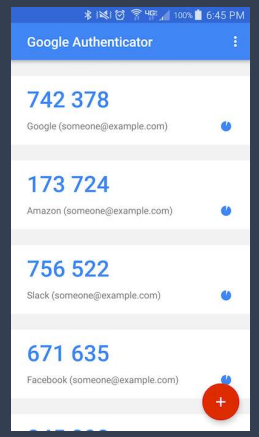
$$Tcounter = floor(\frac{current\ time}{time\ step})$$

HMAC("{secret}", *Tcounter*)

HMAC("{secret}", *Tcounter*)

Authenticator Apps

Time-based One-Time Password (TOTP)



{secret}

$$Tcounter = floor(\frac{current\ time}{time\ step})$$

HMAC("{secret}", *Tcounter*)
➔ 742 378

HMAC("{secret}", *Tcounter*)
➔ 742 378

Passkeys

"Passwordless" is the new hotness

Passwords

Captured in transit ♦ Reused across sites

API Keys / JWT Tokens

Shared secret key stolen in transit or data breach

SAML / OIDC

Vulnerable to phishing ♦ Identity provider might use passwords

Passkeys

"Passwordless" is the new hotness

Passwords

Captured in transit ♦ Reused across sites

Passkeys

Resistant to phishing / replay attacks
Keys stored securely on *user's* device

API Keys / JWT Tokens

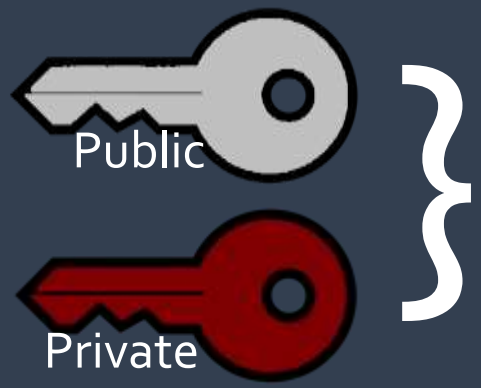
Shared secret key stolen in transit or data breach

SAML / OIDC

Vulnerable to phishing ♦ Identity provider might use passwords

Passkeys

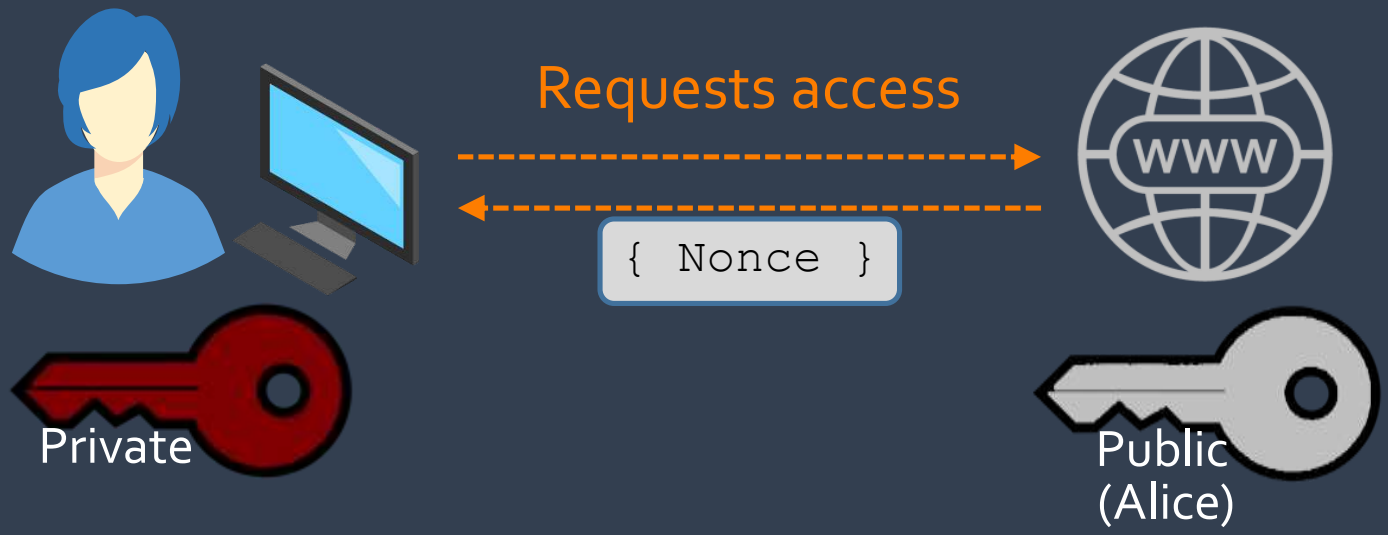
"Passwordless" is the new hotness



"Passkey" is basically a public/private key pair

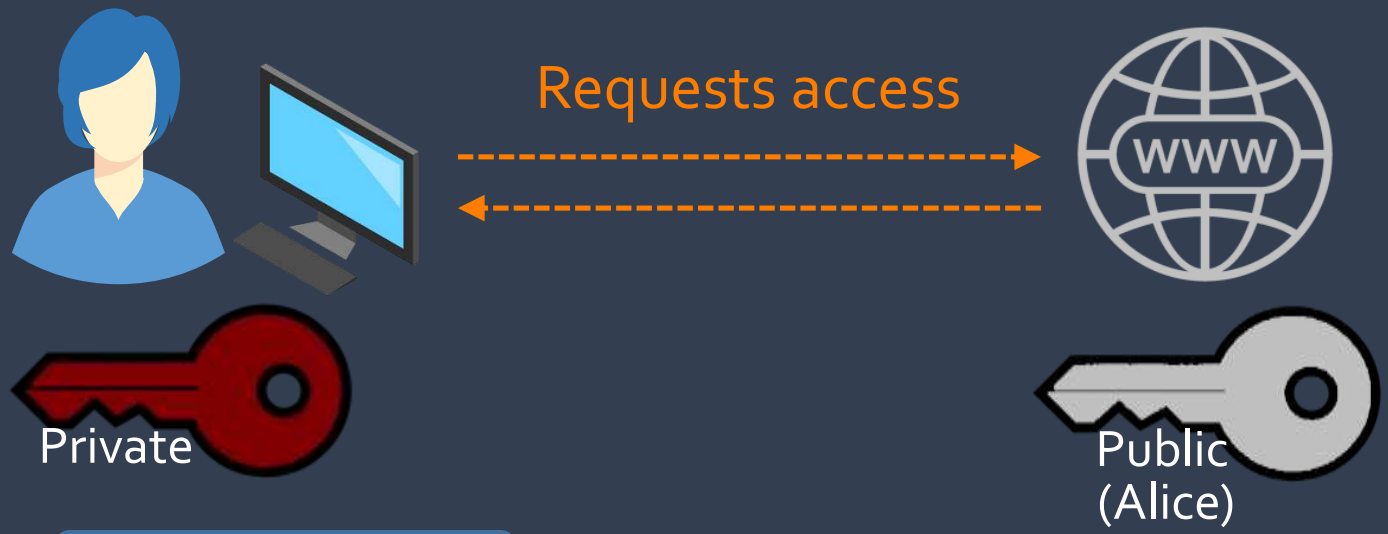
Passkeys

"Passwordless" is the new hotness



Passkeys

"Passwordless" is the new hotness



Nonce value is
signed with private key

Passkeys

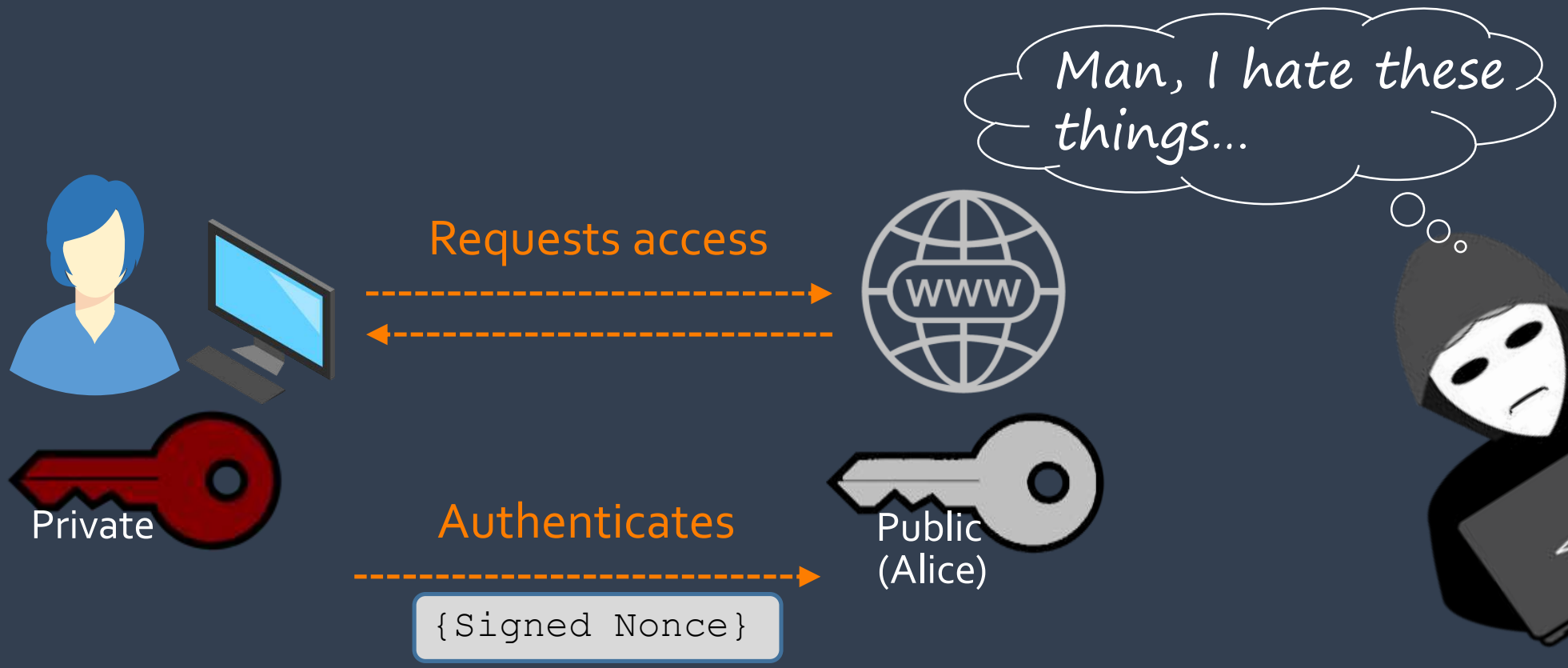
"Passwordless" is the new hotness

Nonce value is
signed with private key

Signed nonce is
verified with public key

Passkeys

"Passwordless" is the new hotness

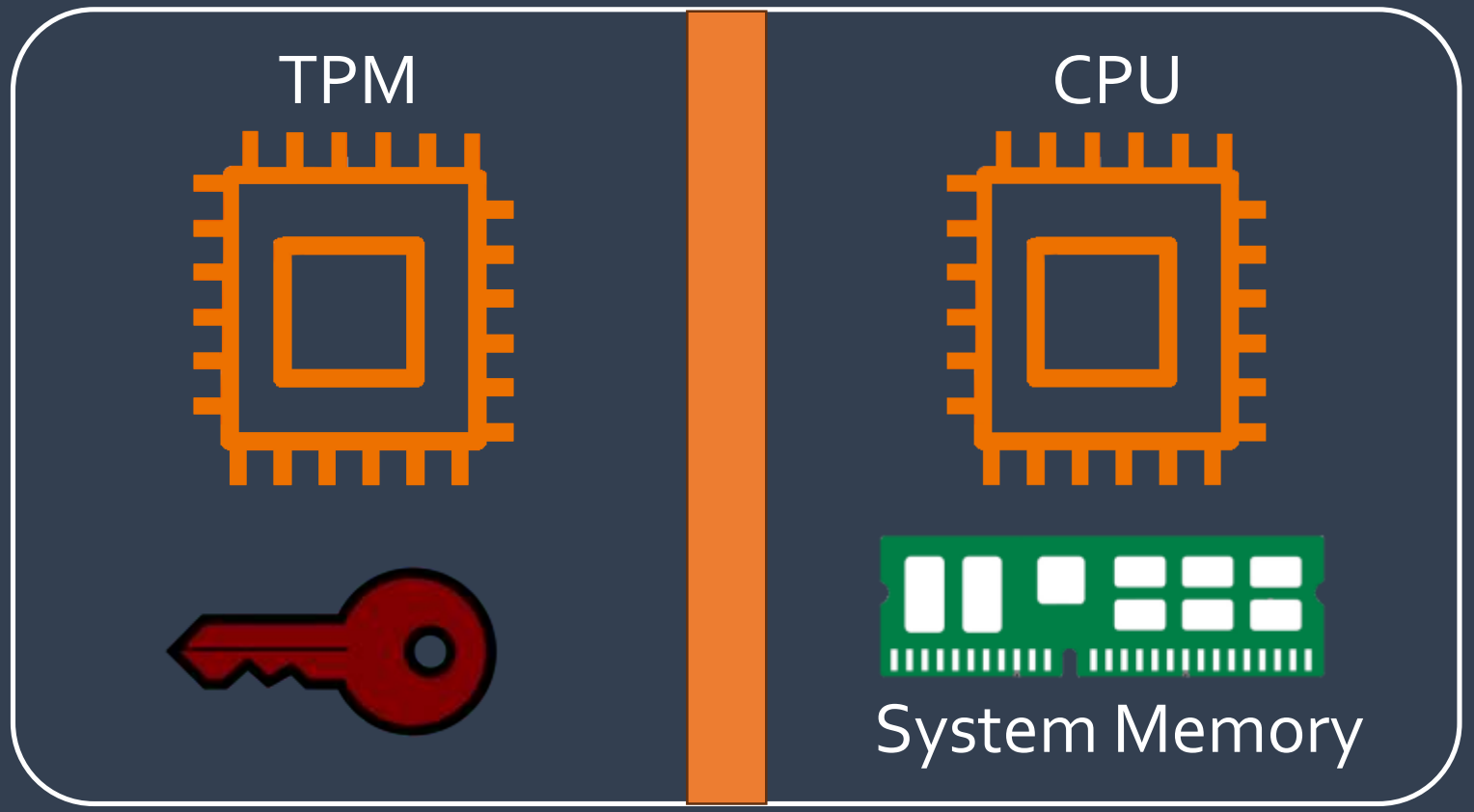


Nonce value is
signed with private key

Signed nonce is
verified with public key

Passkeys

"Passwordless" is the new hotness



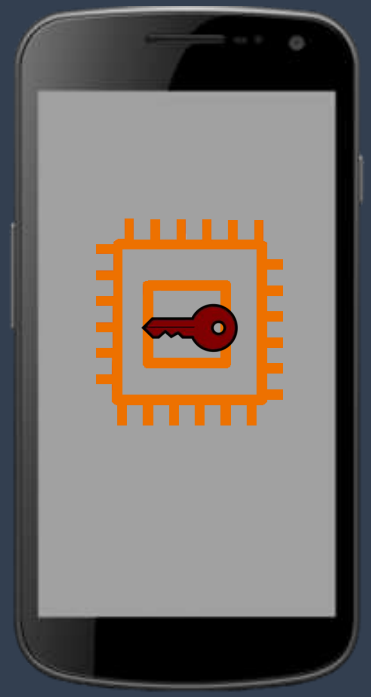
Trusted Platform Modules are dedicated chips that create and securely store cryptographic keys

Passkeys

"Passwordless" is the new hotness

Android's
"Trusted Execution Environment"

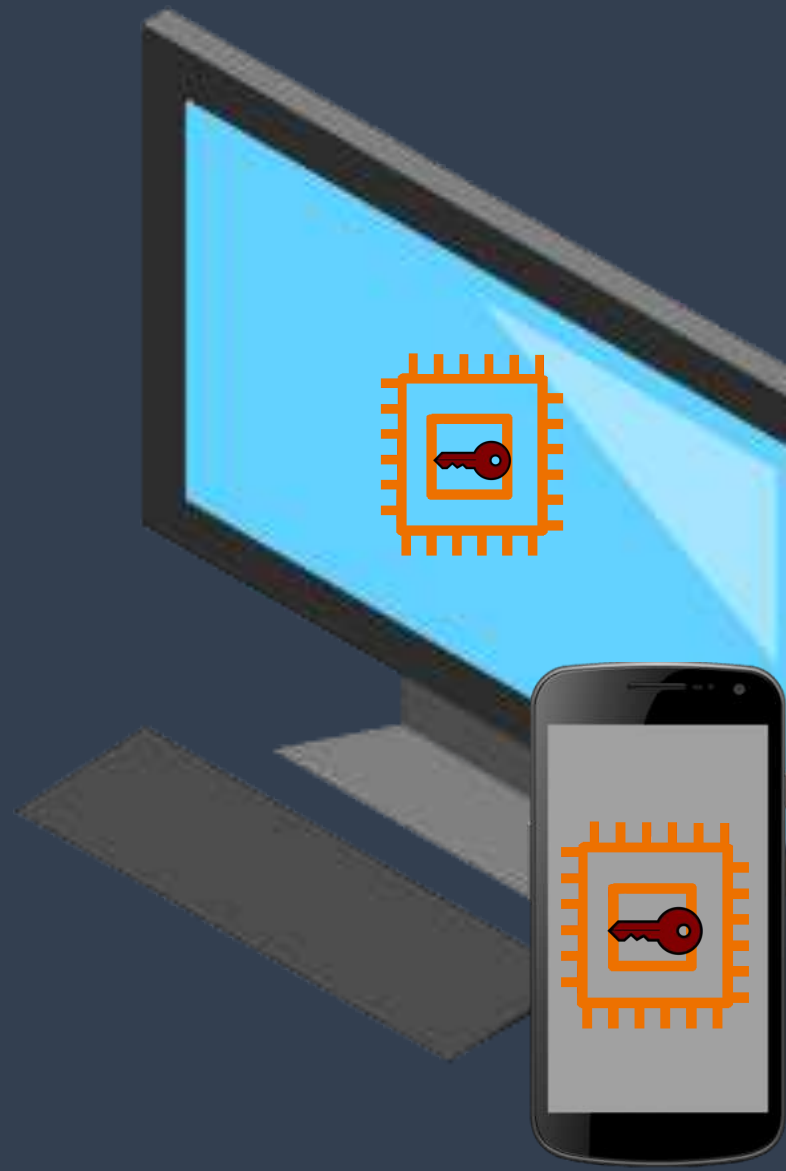
Apple's
"Secure Enclave"



Passkeys

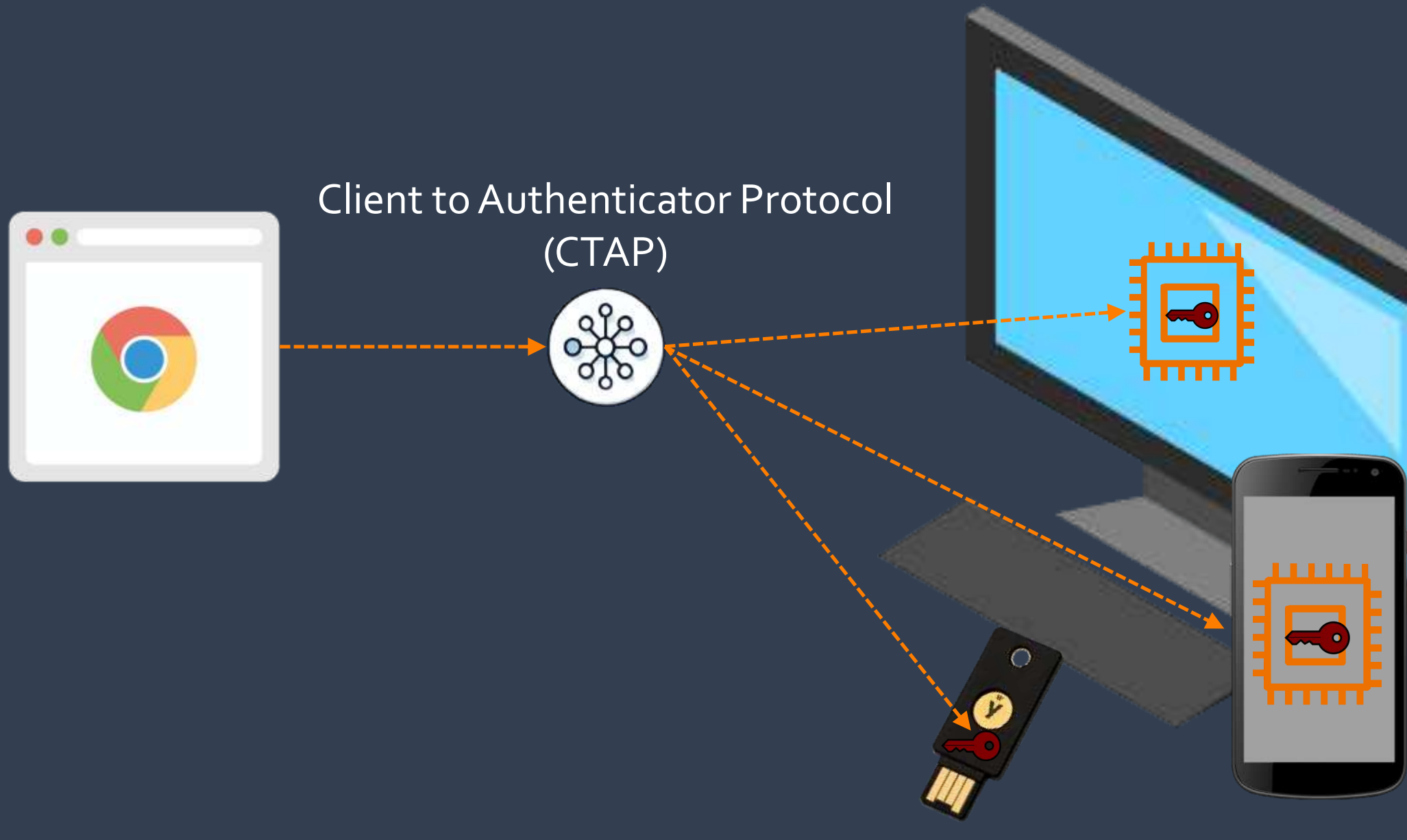
"Passwordless" is the new hotness

Passkeys can also be stored on physical devices like USB Security Keys



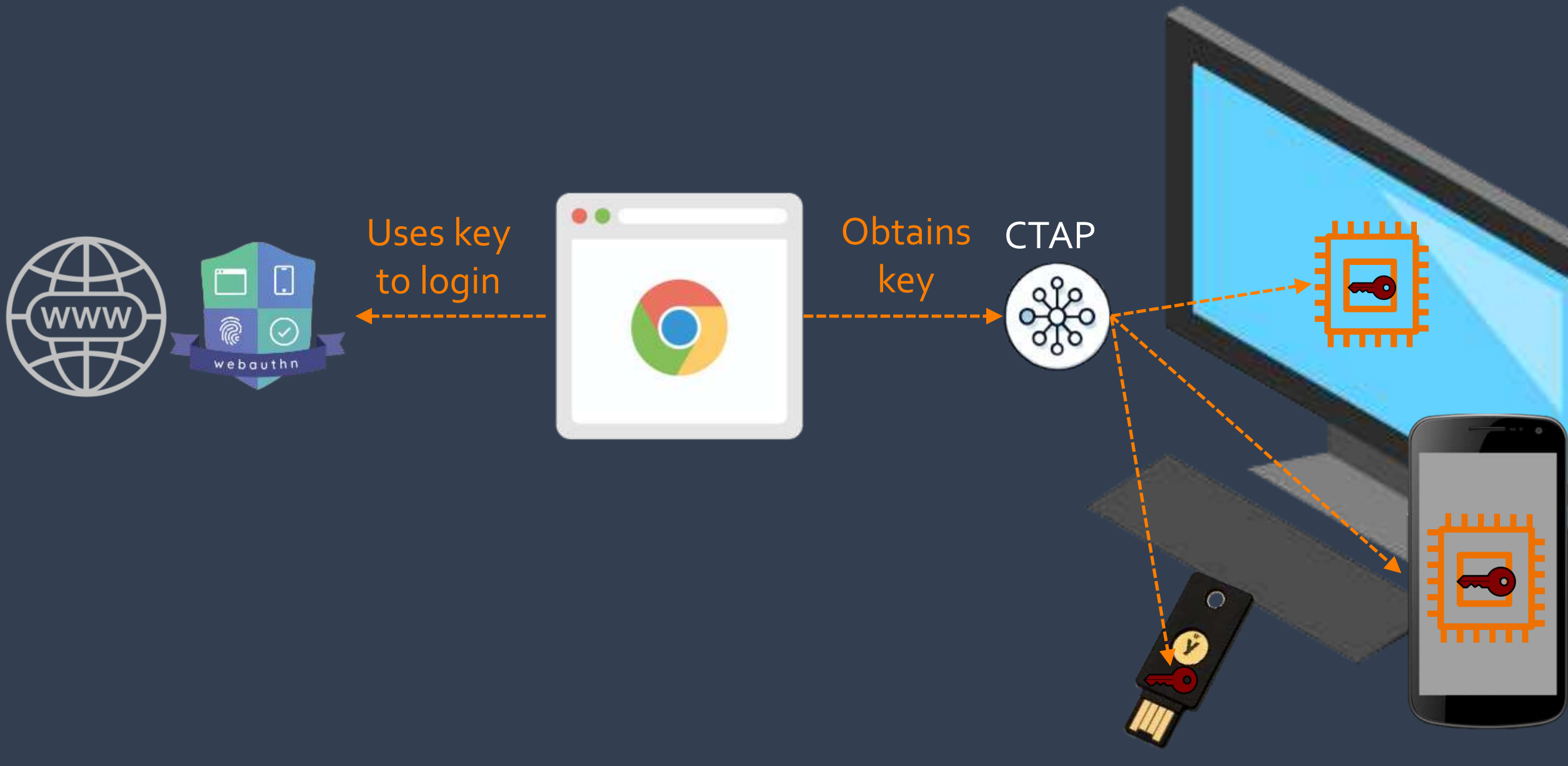
Passkeys

"Passwordless" is the new hotness



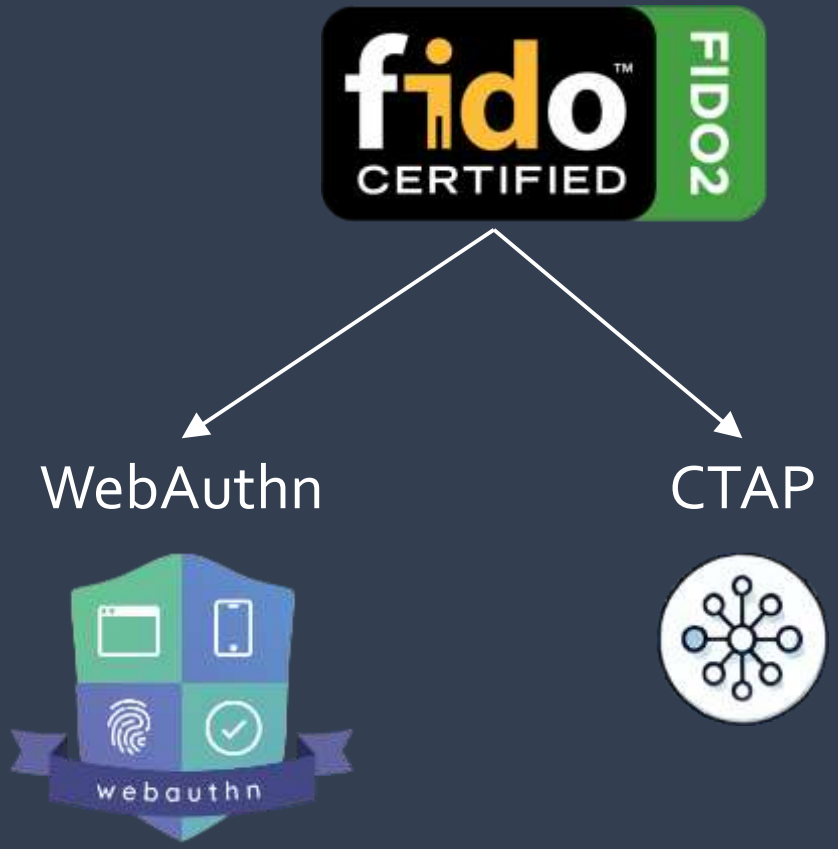
Passkeys

"Passwordless" is the new hotness



Passkeys

"Passwordless" is the new hotness



Agenda

Bit.ly/CodeMash25Auth

- Passwords
- Basic Auth
- API Keys as bearer tokens
- API Keys as cryptographic keys
- JSON Web Tokens
- SAML
- OpenID Connect
- OAuth
- Authenticator Apps
- Passkeys

Credential-based authentication

Claims-based identity

Authorization, not authentication

"Something you have"

@spetryjohnson seth@petry-johnson.com