Bezpieczny wypoczynek

czyli uwierzytelnianie RESTa

Krzysztof Benedyczak
31.05.2017

Inspiracja

Prezentacja na 20. TJUGu

Bolesław Dawidowicz, Andrzej Goławski

Keycloak; Prosty sposób na bezpieczeństwo i uwierzytelnianie w nowoczesnych aplikacjach

Tokens: modern security standards are all about them... What is a token? [...] document with some payload Information about user. Tokens are signed.

JWT -- Simple! -- All you need to know to parse it in your app! ;)

VS

Thomas H. Ptáček
@tqbf

Obserwuj

Thinking about securing an API with JWT? First, punch yourself in the face. Then: just use a 256 bit random token, and a database.

BLOG

Critical vulnerabilities in JSON Web Token libraries

Tony Arcieri @bascule · 27 Jul 2015

.@codeslinger because people think it's a safe least common denominator solution, when it's anything but

Unfortunately, lately I've seen more and more people recommending to use JWT (JSON Web Tokens) for managing user sessions in their web applications. This is a terrible, *terrible* idea, and in this post, I'll explain why.

http://cryto.net/~joepie91/blog/2016/06/13/stop-using-jwt-for-sessions/

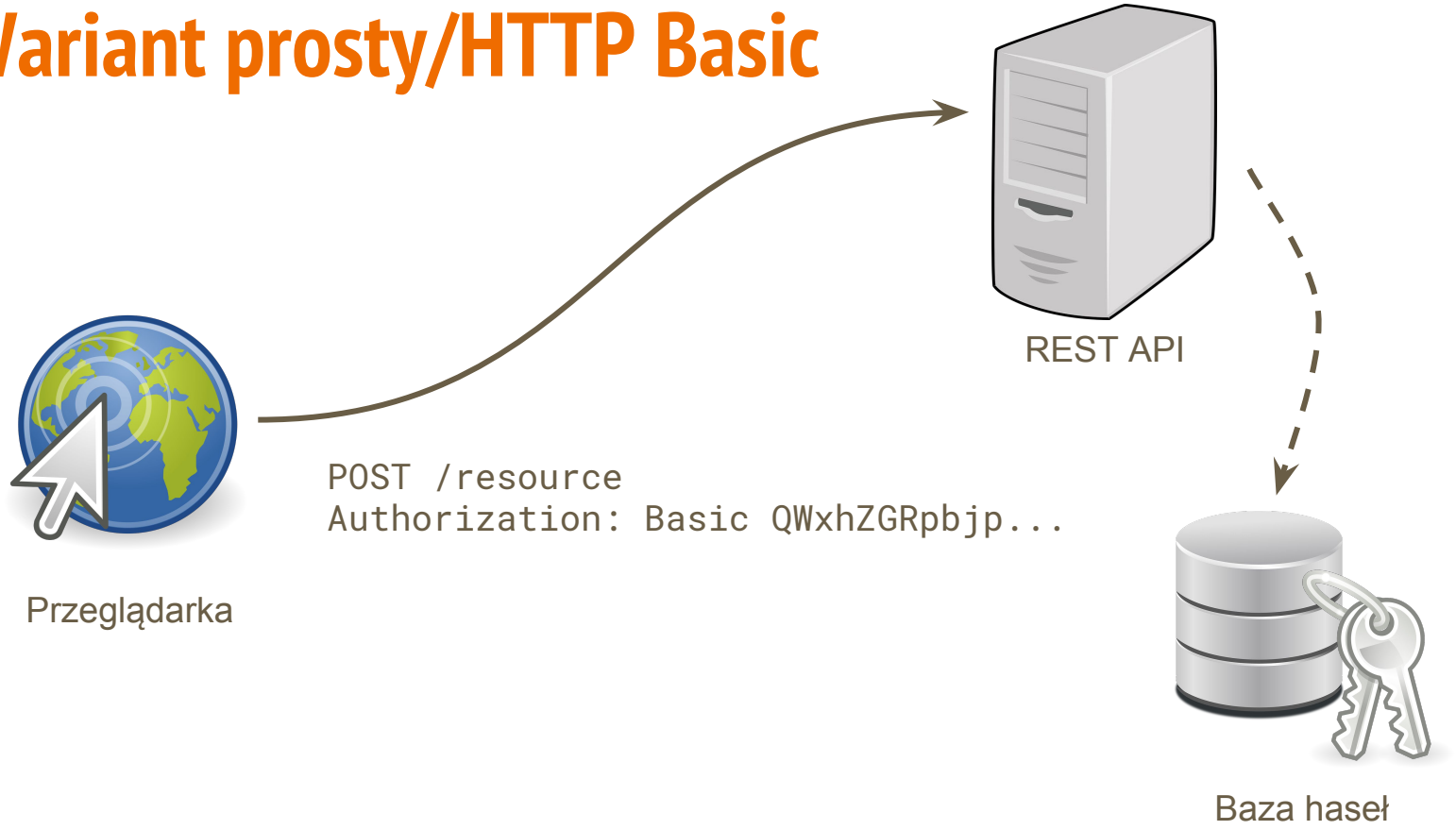
Zakres: przeglądarka + REST API

- Przypadek trudniejszy niż strona generowana przez serwlet
  - Mocniejsza separacja przeglądarki i serwera
  - Stanowość sesji logowania vs bezstanowość REST API
- Zazwyczaj Single Page Application + REST API
  - SPA to jeden z bardziej popularnych trendów, Angular itp
- *Użycie REST API bez przeglądarki jest prostsze, ale nie należy ślepo kopiować rozwiązań przeglądarkowych*

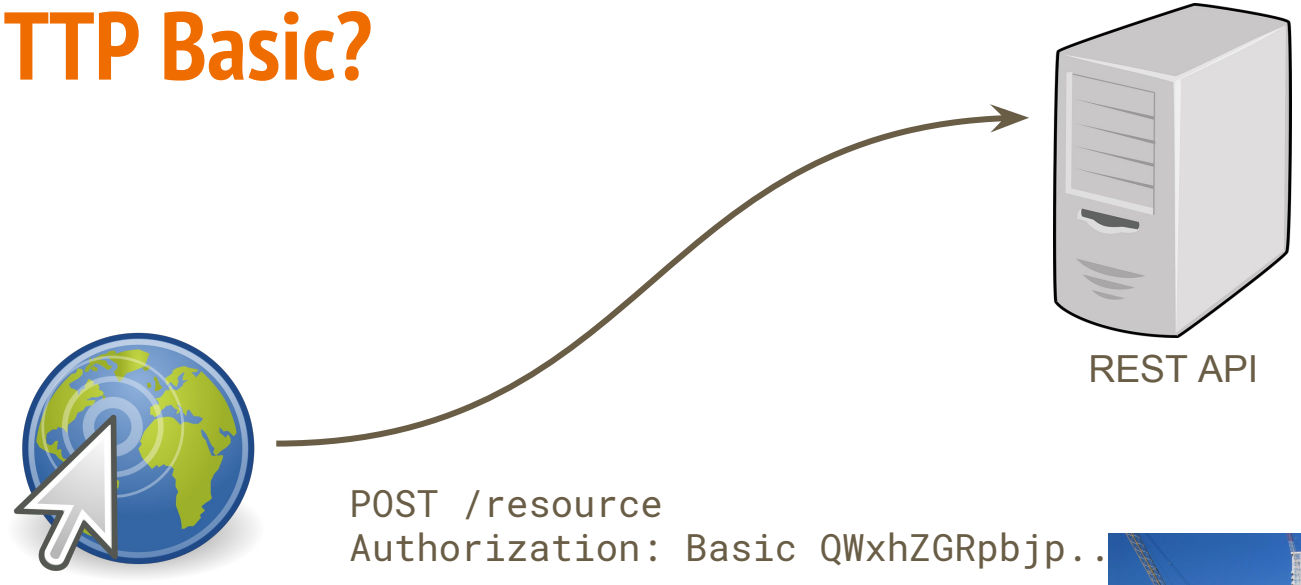
Ale czy to w ogóle jest trudne?

Spróbujmy najprościej!

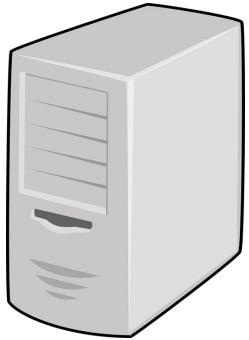
Wariant prosty/HTTP Basic



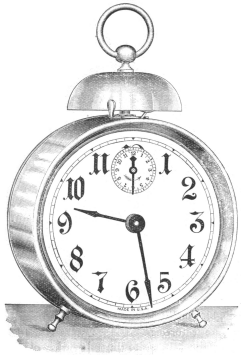
HTTP Basic?


REST API
POST /resource
Authorization: Basic QWxhZGRpbjp..
Przeglądarka

HTTP Basic?



REST API

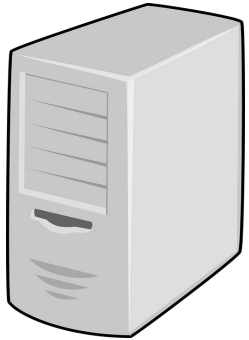


Przeglądarka

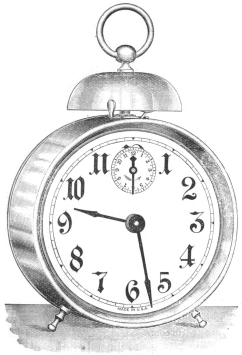
```
POST /resource
Authorization: Basic QWxhZGRpbjp..
```

HTTP Basic?

Przeglądarka



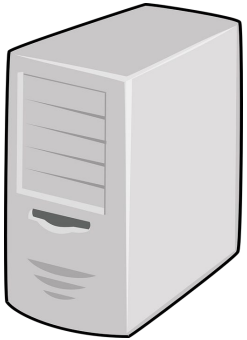
REST API



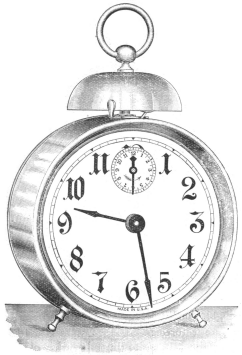
```
POST /resource
Authorization: Basic QWxhZGRpbjp..
```

HTTP Basic?

Przeglądarka



REST API



```
POST /resource
Authorization: Basic QWxhZGRpbjp..
```

XSS

- Cross-site scripting (XSS) pojawia się gdy umożliwiamy atakującemu na umieszczenie własnego kodu JS na stronie naszej aplikacji, tak że wykona go przeglądarka użytkownika otwierającego stronę aplikacji.
  - Umożliwiamy wyświetlanie komentarzy z podstawowymi tagami HTML:
    ```
    <IMG SRC=/ onerror="document.write('this page is hacked ;-)')"></img>
    ```
  - Zwracamy input użytkownika w przypadku gdy zawiera niedozwoloną zawartość:
    ```
    <span>Niedozwolona zawartość: ${maliciousContent} </span>
    ```
  - ... atakujący wysyła mejla ofierze zachęcając do wejścia na:
    ```
    <a href="https://our.blog.com?search=%3CSCRIPT%20
    SRC%3Dhttp%3A%2F%2Fevil.net.com%2Fxss.js%3E%3C%2FSCRIPT%3E%0A">
    our.blog.com</a>
    ```

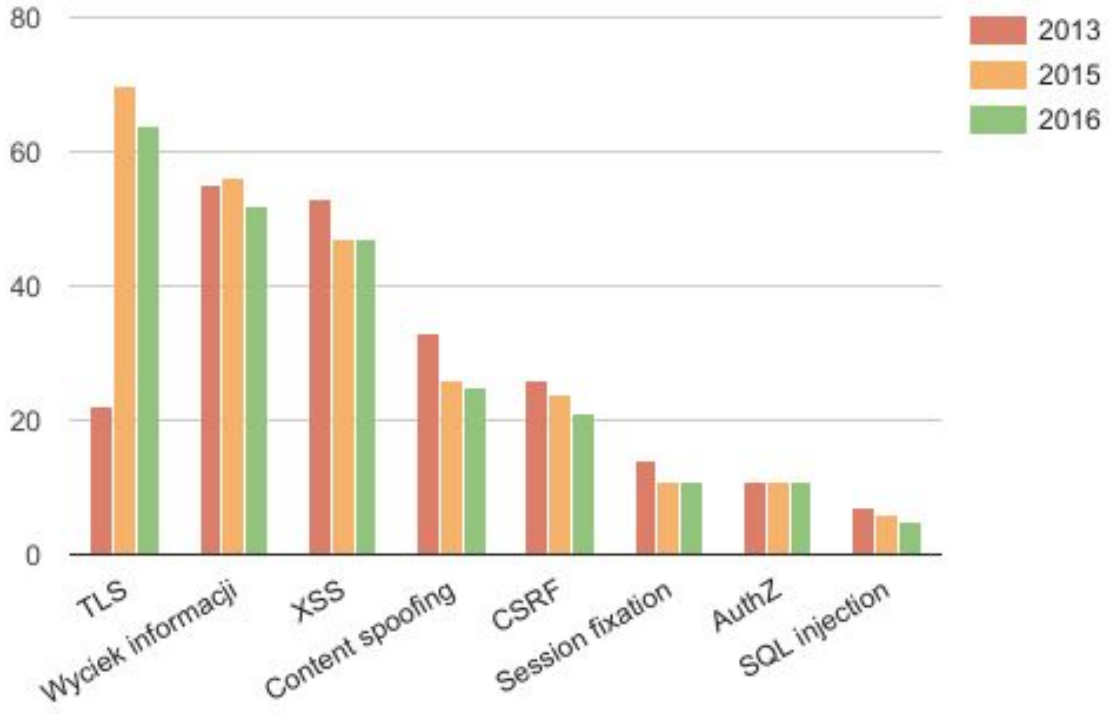
XSS

XSS ma dostęp do LocalStorage i zwykłych ciastek. Jest SOP.

XSS raczej się nam trafi.

Trzeba być uważnym. Cały czas.

Obrona: procesowanie wyjścia.



Prawdopodobieństwo wystąpienia przynajmniej jednej podatności

Za: Website Security Statistics Report 2016, 2015, 2013, WhiteHat Security

Czego nas uczy ten przypadek?

- Hasła nie wolno przechowywać - musi być użyte tylko raz do uwierzytelnienia
- Dane wrażliwe w przeglądarce nie mogą być dostępne z poziomu JS
  - Zabezpieczenie się przed XSS jest krytyczne
- Samodzielne stworzenie mechanizmu przechowywania haseł jest bardzo trudne i czasochłonne.
  - Warto rozważyć użycie usługi/komponentu zewnętrznego/gotowego
  - Warto mieć możliwość stosowania MFA
- Powinniśmy mieć mechanizm wygaszania nieaktywnej sesji
  - Czyli wsparcie użytkownika który się nie wylogował

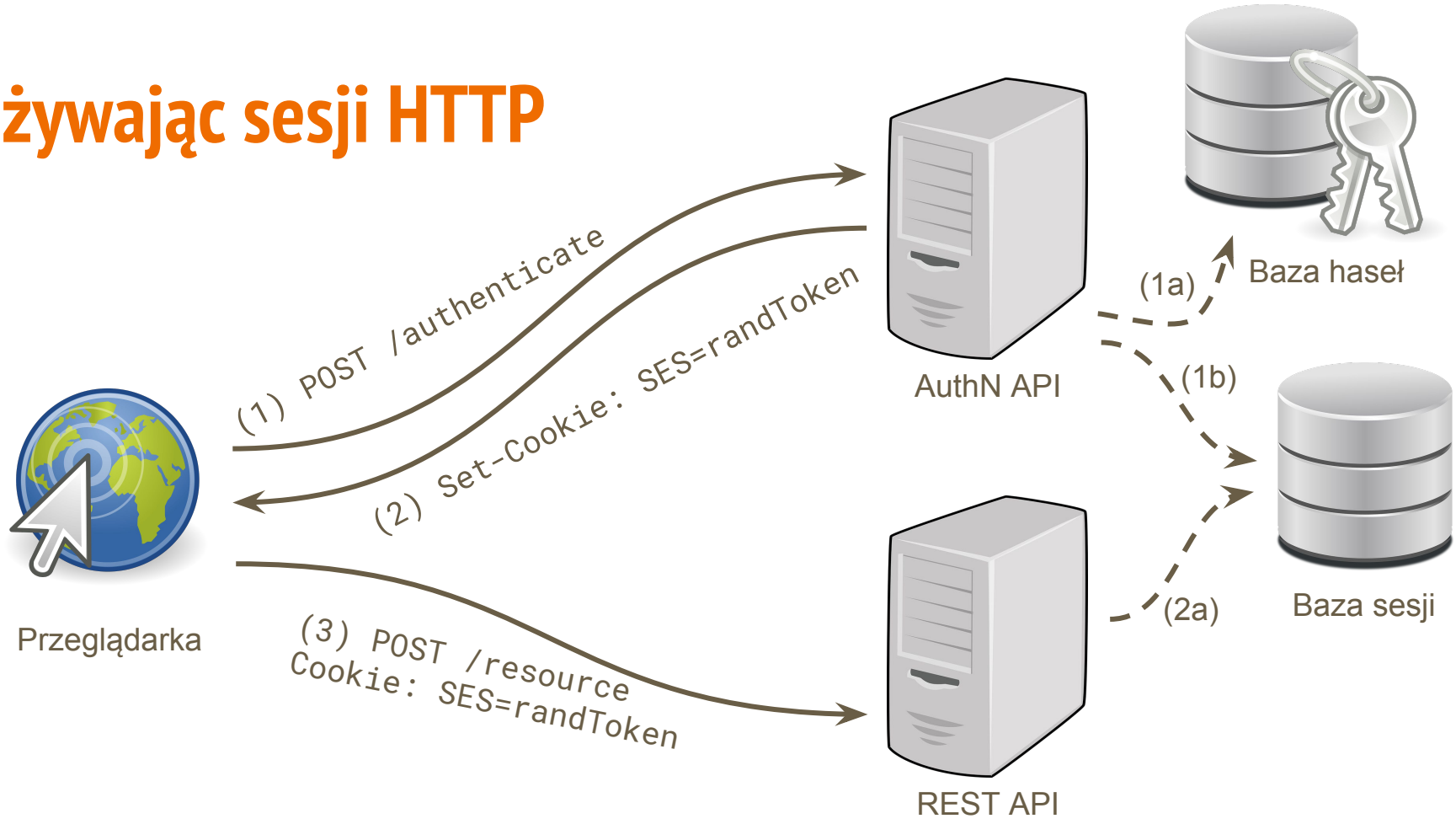
Podejście naiwne jest złe

Odwołajmy się więc do sprawdzonych rozwiązań

Rozwiązanie tradycyjne

- Hasło może być sprawdzane tylko jeden raz: wydzielamy punkt logowania
- W konsekwencji musimy mieć metodę przekazania informacji o zalogowaniu do innych zasobów
- Rozwiązanie tradycyjne to sesja HTTP
- Muszą ja współdzielić wszystkie usługi

Używając sesji HTTP



O czym należy pamiętać przy sesjach?

- Ciasteczko sesji HttpOnly
  - Nie można wykraść sesji w przypadku udanego XSS
- Bezwzględnie duży i losowy token sesji
  - Zazwyczaj domyślna implementacja z serwera aplikacji
- Mogą się pojawić problemy z domenami dla ciasteczek
  - Dokładnie przemyśleć strukturę domen aplikacji
  - I odpowiednio dobrać parametr Domain ciasteczka (lub z niego zrezygnować)
- Zaimplementować zabezpieczenia przed CSRF

CSRF?

- Aplikacja firmy kurierskiej B2B umożliwia przekierowanie nadanej paczki za pomocą żądania POST do REST API
  - Uwierzytelnianie ciastkiem z tokenem sesji
- Atakujący zachęca użytkowników do wejścia na spreparowaną stronę w jego kontroli.
- Strona zawiera niewidoczny formularz, przy wejściu na stronę jej kod automatycznie go wysyła.
  - Formularz generuje żądanie POST przekierowujące paczkę na fizyczny adres złodzieja.
- Przeglądarka załącza automatycznie ciasteczko przypisane do domeny aplikacji firmy kurierskiej.
- Jeśli tylko użytkownik się nie wylogował...

CSRF - zabezpieczenia

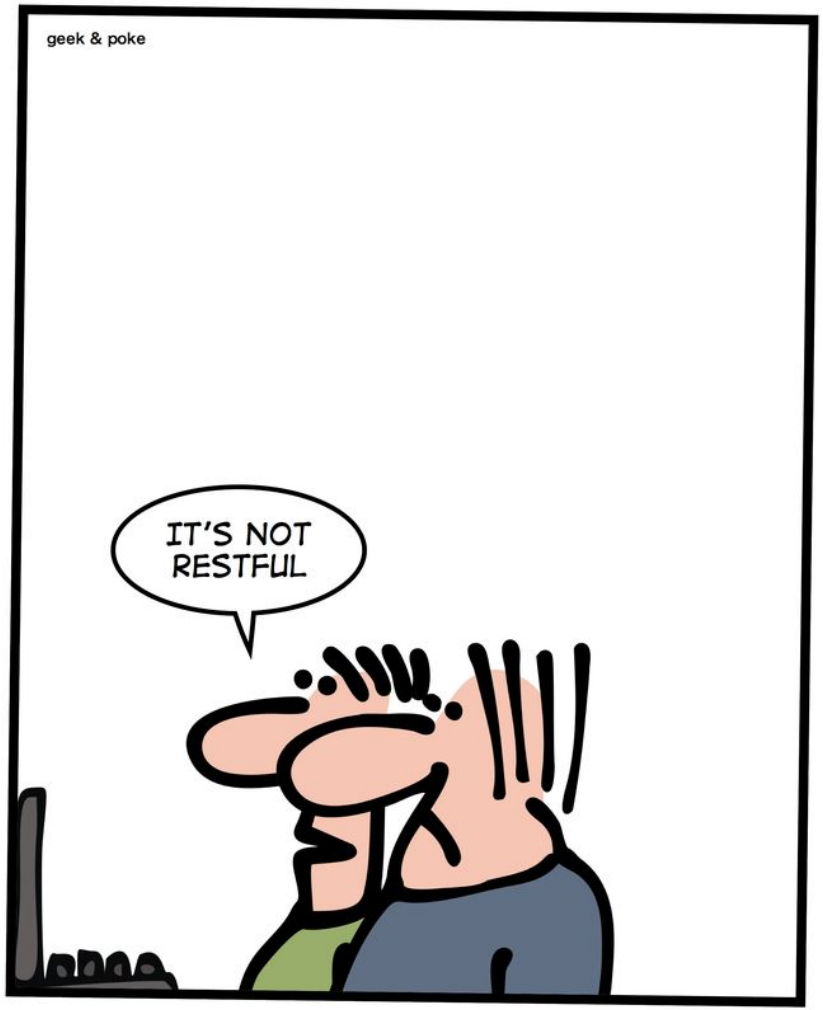
- Walka z CSRF sprowadza się do wymuszenia aktywacji kontroli CORS w przeglądarce.
- Najprostsze zabezpieczenie przed CSRFem to dodanie (i sprawdzanie po stronie serwera) niestandardowego nagłówka HTTP. Np.: `X-Requested-With: XMLHttpRequest`
  - Serwer odrzuca żądania bez tego nagłówka.
- Warto także po stronie serwera kontrolować nagłówki Origin i Referer a także sprawdzać content type requestów.
- Są też inne metody jeśli nie robimy aplikacji stricte JS, np. double submit cookie.

W wielu przypadkach sesja HTTP to właściwe podejście

...i można na nim poprzestać

- Jest świetne wsparcie dla sesji w kontenerach.
- Nie wpadamy w kryptografię!
- Sesja powinna być lekka
- Możemy ją współdzielić również przez własną bazę
  - Często Key-Value store wystarcza
  - Redis, Hazelcast

HOW TO INSULT A DEVELOPER



Rob Winch

Cryptography is hard - even for experts

It is hard to fight this feeling ... [but] ... you CAN make it scale and perform.

A jeśli jednak się uprzeć na bezstanowość?

Czyli gdzie współdzielona sesja się nie sprawdzi?

- A właśnie że robimy duży system multi-DC
- Ponieważ tworzymy system rozproszony geograficznie lub federacyjny
- Ponieważ chcemy użyć rozwiązania z półki (lub chmurki) zamiast implementować własną bazę haseł i użytkowników

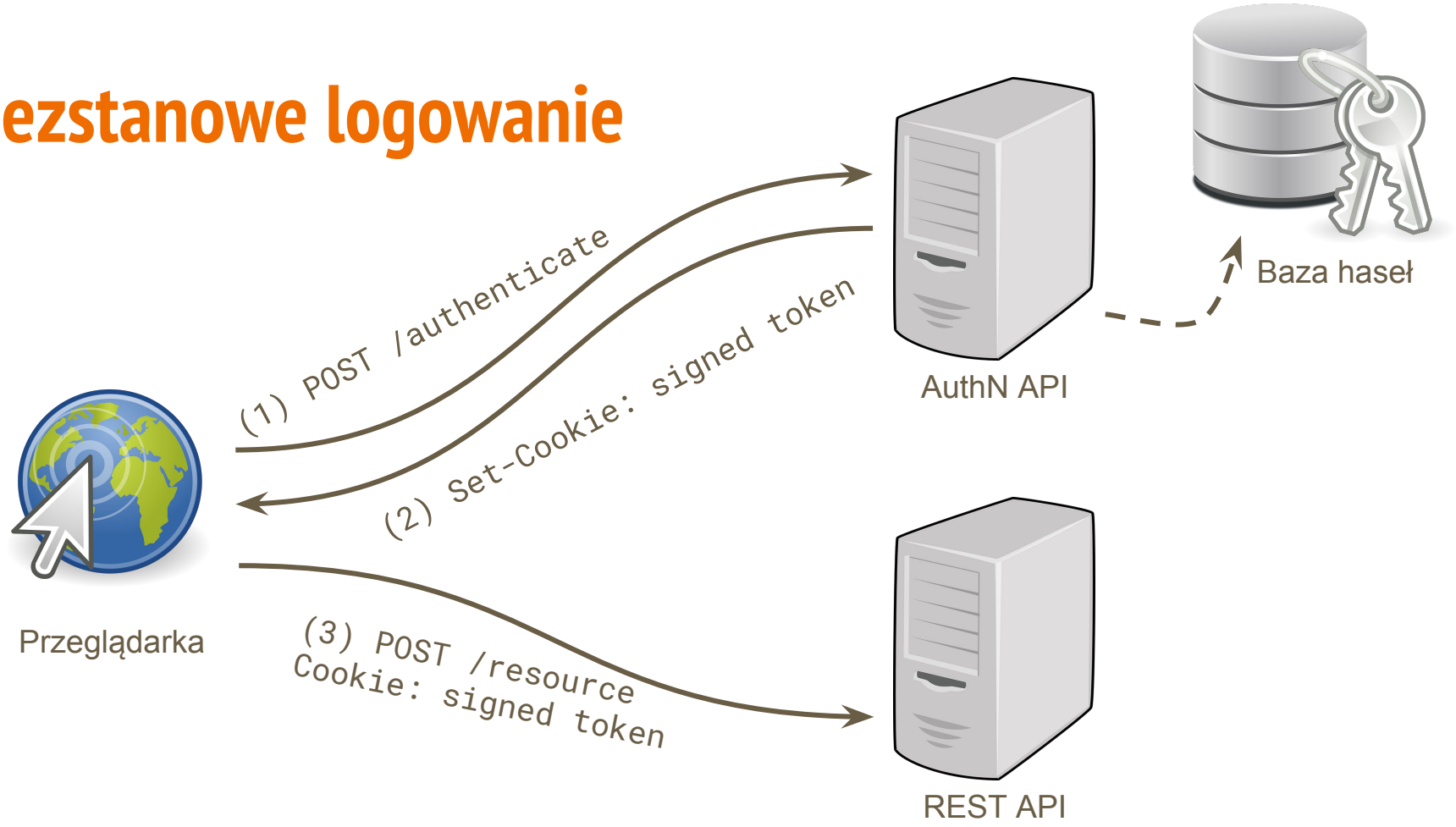
Nasze wymagania są duże

Potrzebujemy bezstanowości, zastosujemy więc podpisane tokeny

W pełni bezstanowe uwierzytelnianie

1. Komplet niezbędnych informacji o użytkowniku przekazany w tokenie
2. Token podpisany tak że każdy punkt REST API może samodzielnie sprawdzić jego autentyczność
3. Przeglądarka wysyła token z każdym żądaniem
4. Token zawiera czas ważności który ogranicza jego trwałość

Bezstanowe logowanie



Problemy czyli czas życia tokenu

- Token musi być ważny krótko: *maksymalnie* tyle na ile ustalamy dozwolony czas nieaktywności użytkownika
  - W praktyce trzeba brać pod uwagę jeszcze szybkość propagacji zmian danych zawartych w tokenie (np. roli)
- API logowania musi oferować punkt odświeżania tokenu
  - Tylko ważny token może być odświeżony
  - Odświeżenie wymienia stary token na nowy
  - Potrzebne górne ograniczenie (np. 24h).
- Klient (u nas SPA w przeglądarce) musi kontrolować aktywność użytkownika i odświeżać token przed jego wygaśnięciem
  - Ponieważ JS nie ma dostępu do tokena po jego otrzymaniu punkt uwierzytelniania musi podawać oddzielnie czas ważności

Jak zrealizować podpisany token?

- Najprościej - zalecane - za pomocą MAC, zazwyczaj HMAC
  - O ile wszyscy odbiorcy tokenu mogą łatwo współdzielić sekretny klucz
  - W przeciwnym przypadku podpis z kryptografii asymetrycznej
- MAC (Message Authentication Code) to kod pozwalający na jednoczesne:
  - Zapewnienie autentyczności pochodzenia stowarzyszonej wiadomości
  - Oraz zapewnienie jej integralności
- Najpopularniejsza realizacja to MAC bazowane na funkcji hash:

  SHA256(KEY | SHA256(KEY | MESSAGE))

  - Klucz musi być duży i losowy, zalecane 256bit
  - Nie należy używać “uproszczonych” wariantów, np. SHA256(KEY | MESSAGE)
- AE jako opcja: encrypt-then-MAC

To co jest nie tak z tym JWT?

- Podstawowy zarzut podnoszony przez ekspertów to wady samej specyfikacji, które wręcz promują tworzenie dziurawych implementacji.
  - Co niestety stało się faktem i to dosyć powszechnym.
- Struktura JWT:

Header:

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

Body:

```
{
  "sub": "123456",
  "name": "Toruń JUG",
  "role": "super-user"
}
```

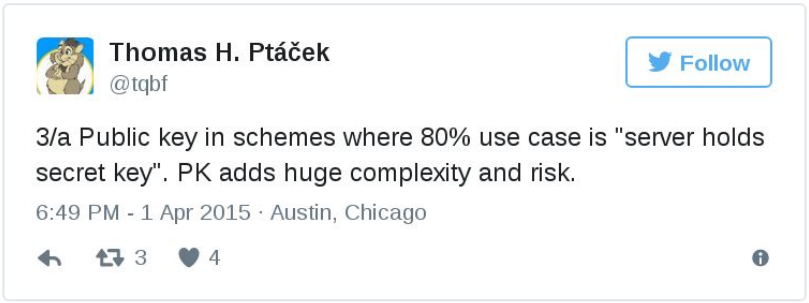
Signature:
zależna od wartości alg. Np. dla HS256:

HMAC-SHA256(secretKey, base64(header) + ”.” + base64(body))

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTYiLCJuYW1lIjoiVG9ydcWEIEpVRyIsInJvbGUiOiJzdXBlci11c2VyIn0.ZB7nHrU7RKE1926-Pa_JROaq_gA6uBRDszbLE1-1GRQ

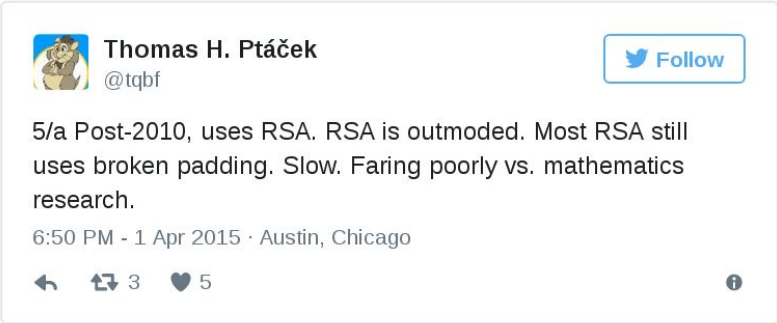

Thomas H. Ptáček
@tqbf
Follow
1/a Negotiation phases. Nobody has ever gotten this right. Invariably: downgrade attacks.
6:49 PM - 1 Apr 2015 · Austin, Chicago
4
7


Thomas H. Ptáček
@tqbf
Follow
2/a Selectable algorithms. Complexity, protocol joinery. Subtle distinctions between alg assumptions break.
6:49 PM - 1 Apr 2015 · Austin, Chicago
4
6


Thomas H. Ptáček
@tqbf
Follow
3/a Public key in schemes where 80% use case is "server holds secret key". PK adds huge complexity and risk.
6:49 PM - 1 Apr 2015 · Austin, Chicago
3
4

JWT


Thomas H. Ptáček
@tqbf
Follow
4/a Interchangeably uses MAC and public key signature. Not remotely the same concepts. PK sigs much trickier than MAC.
6:50 PM - 1 Apr 2015 · Austin, Chicago
3
4


Thomas H. Ptáček
@tqbf
Follow
5/a Post-2010, uses RSA. RSA is outmoded. Most RSA still uses broken padding. Slow. Faring poorly vs. mathematics research.
6:50 PM - 1 Apr 2015 · Austin, Chicago
3
5


Thomas H. Ptáček
@tqbf
Follow
6/a Static public keys of any sort used for anything but signing. Modern cryptosystems are forward secure.
6:50 PM - 1 Apr 2015 · Austin, Chicago
4
4

JWT: alg

Specyfikacja JWT nakłada obowiązek na kompatybilne implementacje by wspierać algorytm **none**.

Header:

```
{
  "alg": "none",
  "typ": "JWT"
}
```

Body:

```
{
  "sub": "123456",
  "name": "Toruń JUG",
  "role": "super-user"
}
```

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOiIxMjM0NTYiLCJuYW1lIjoiVG9ydcWEIEpV
RyIsInJvbGUiOiJzdXBlci11c2VyIn0.
```

JWT: alg mixup

Specyfikacja JWT podstawowe algorytmy: **RS256** i **HS256** - odpowiednio oparte na asymetrycznym RSA i symetrycznym HMAC.

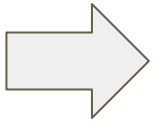
Załóżmy że serwer generuje tokeny przy użyciu RS256. Będzie podpisywał kluczem prywatnym a sprawdzał podpis tokenu kluczem publicznym.
Atakujący wysyła stworzony przez siebie token, używając klucza publicznego serwera do podpisu:

Header:

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

Body:

```
{
  "sub": "123456",
  "name": "Toruń JUG",
  "role": "super-user"
}
```



Jeśli serwer nie sprawdza ręcznie wartości alg:

```
verify(token, publicKey)
```

wynik jest łatwy do przewidzenia...

JWT?

JWT można używać jako kontenera podpisanych danych, jednak nadzwyczaj ostrożnie.

Dokładne i ścisłe sprawdzenie tego czy nagłówek jest oczekiwany jest kluczowe.

Bezpieczniej polegać na HS256.


The Quest is Over!
MONTY PYTHON'S
HOLY GRAIL
Ale
Tempered over burning witches
HOLY GRAIL
ALE
Tempered over burning witches

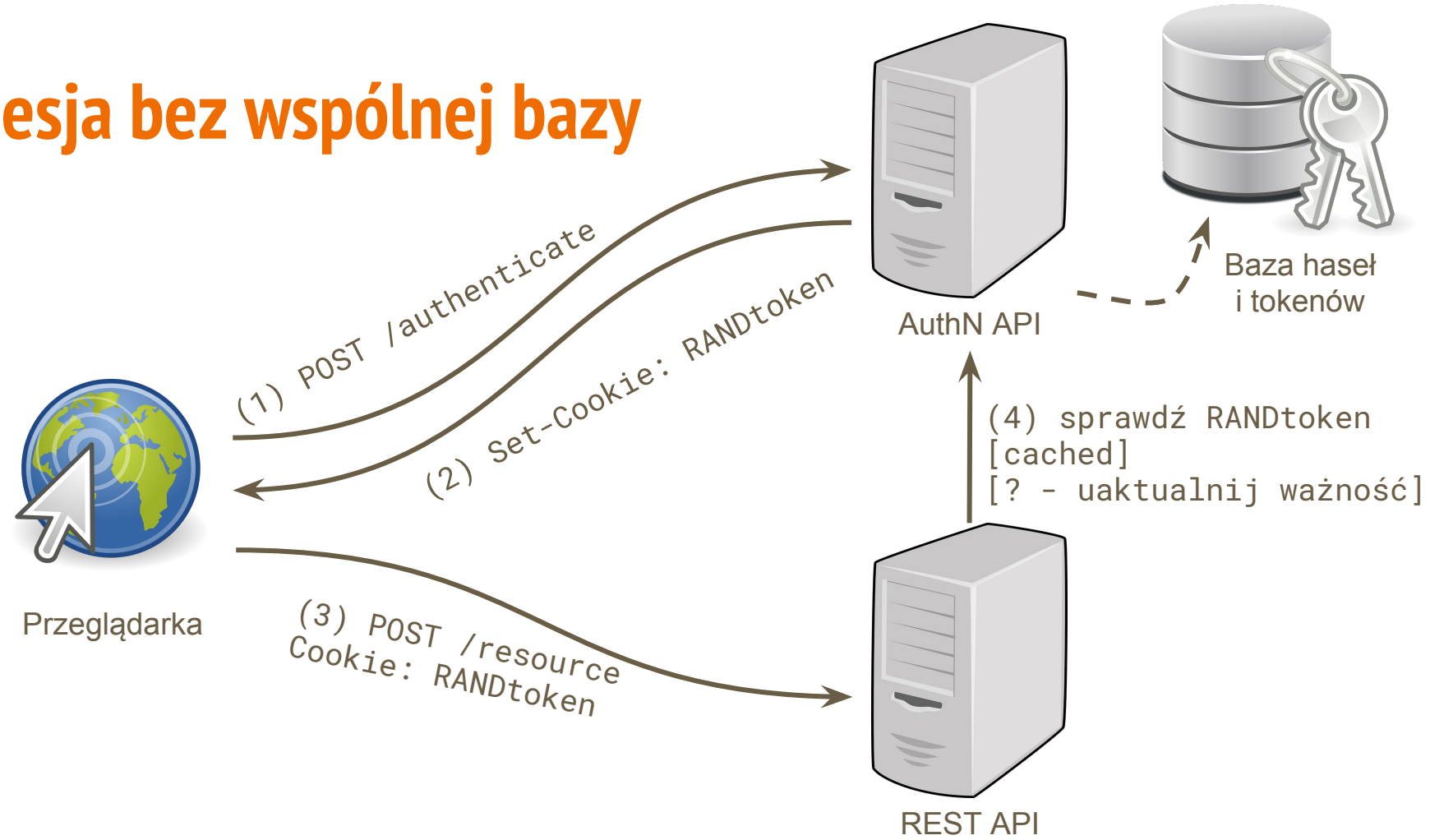
Problemy które pozostają

- Mamy skomplikowaną kryptografię - trzeba uważać i rozumieć co się robi
- Implementacja frontu się bardzo komplikuje
  - Łatwo o dziurawy kod
- Punkt logowania robi się też bardziej skomplikowany
- Zmiana atrybutów użytkownika wciąż jest propagowana z pewną latencją.
  - Można wystawiać token o bardzo krótkiej ważności (np. 30s) i zmuszać klienta do odświeżania go non-stop, ale daje to sporo bezsensownego obciążenia serwerów i nawet chwilowy reset usługi logowania spowoduje wylogowanie wszystkich użytkowników.

I tak źle i tak nie dobrze...

Czy można zrezygnować zarówno z kryptografii jak i z współdzielonej bazy sesji?

Sesja bez wspólnej bazy



Logowanie z losowym tokenem

Zalety

- Brak użycia kryptografii
- Kontrola danych użytkownika w pełni po stronie serwera
- Czas propagacji zmian atrybutów zależny tylko od ustawień cache'u
- Można łatwo zaimplementować kontrolę aktywności użytkownika i przedłużanie ważności tokenu
- Perfekcyjnie dopasowane do użycia zewnętrznych/gotowych usług logowania

Wady

- Potrzebna bardziej skomplikowana usługa logowania (niż w przypadku współdzielenia sesji)

No dobrze a OAuth? OIDC?

Kiedy zastosować?

OAuth i OpenID Connect

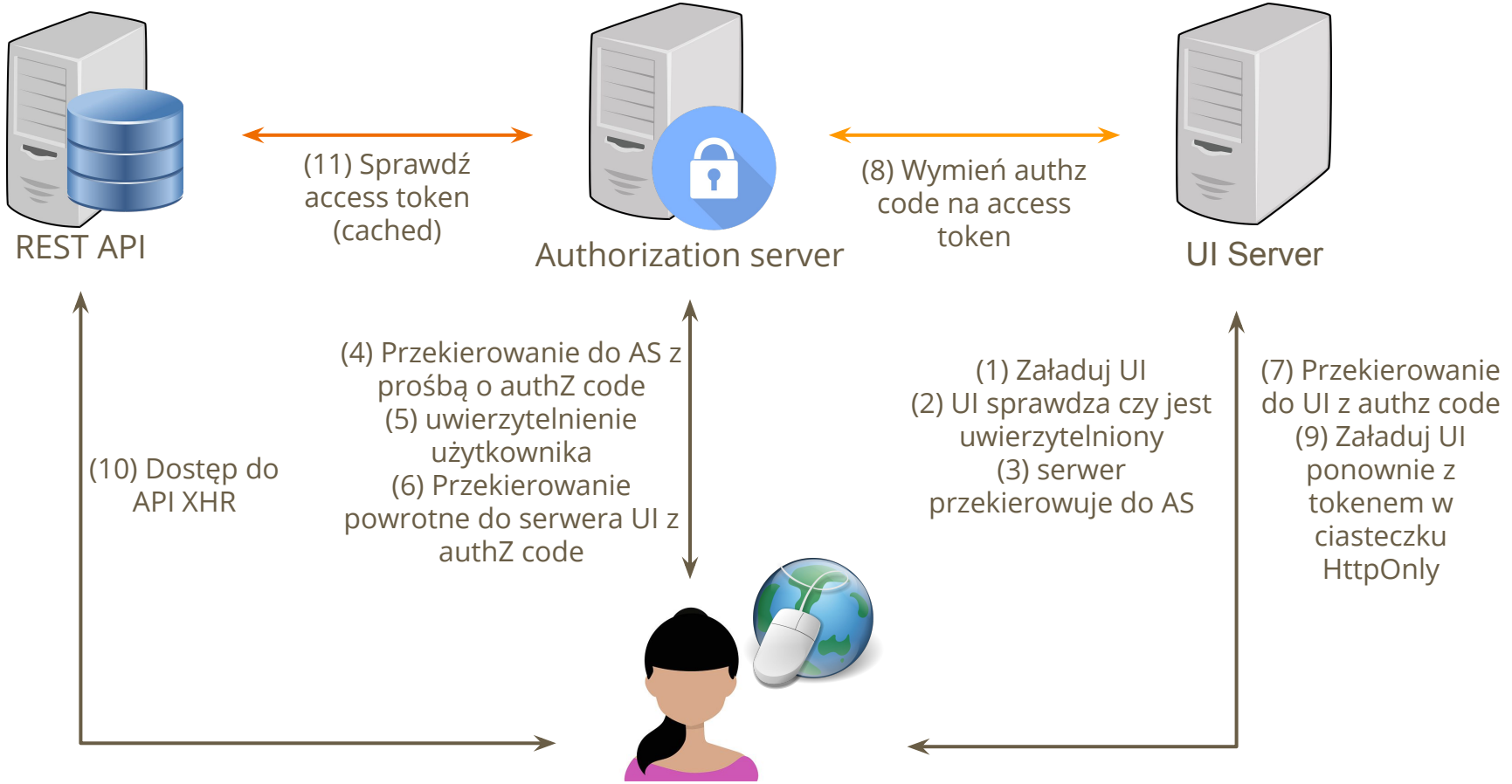
- Pamiętajmy: OAuth został zaprojektowany jako mechanizm delegacji, nie uwierzytelniania.
- Można go zastosować do uwierzytelniania ale będzie to nieco nienaturalne.
  - Niektóre elementy specyfikacji będą utrudniać życie (refresh)
  - Np. OAuth-owy Client to trochę UI w przeglądarce trochę serwer który to UI ładuje i pomaga ustawić ciastko. A to może być jeszcze pomieszane z Resource Providerem.
  - Wybór odpowiedniego wariantu może być skomplikowany
- OIDC to profil OAuth opierający się na JWT i podpisanym tokenie.
  - Standaryzuje przekazywanie informacji o użytkowniku
  - Dalej nie przybliża OAuth-a do użycia w przeglądarce.

OAuth guide

- Stosować jeśli używamy usługi uwierzytelniania (on-permise, cloud).
  - Ew. jeśli robimy własną, ale tu warto się 3 razy zastanowić czy należy - konkurencja rynku IAM/IRM/IdM/... jest ogromna.
- Jeśli chcemy użyć wariantu z losowym tokenem - czysty OAuth wystarczy
- Jeśli jednak token podpisany - wtedy OIDC
- Używając OAuth/OIDC praktycznie zawsze należy zastosować *authorization code grant* (*flow*).

OAuth - problemy

- Sprawdzenie access token nie jest zdefiniowane w bazowej specyfikacji OAuth2.
  - W przypadku JWT - nieistotne
  - Ostatnio wydano RFC 7662 OAuth 2.0 Token Introspection - wypełnia lukę
  - Praktycznie wszystkie usługi definiują własny mechanizm
- Odnowienie (refresh) w wydaniu OAuth jest kłopotliwe.
  - Zostało zaprojektowane na potrzeby delegacji
  - Może go dokonać w praktyce tylko serwer.



OAuth | authz code

Na wynos

- Uwierzytelnianie z przeglądarki tylko przez HttpOnly ciastko
- Konieczne zabezpieczenie przed CSRF