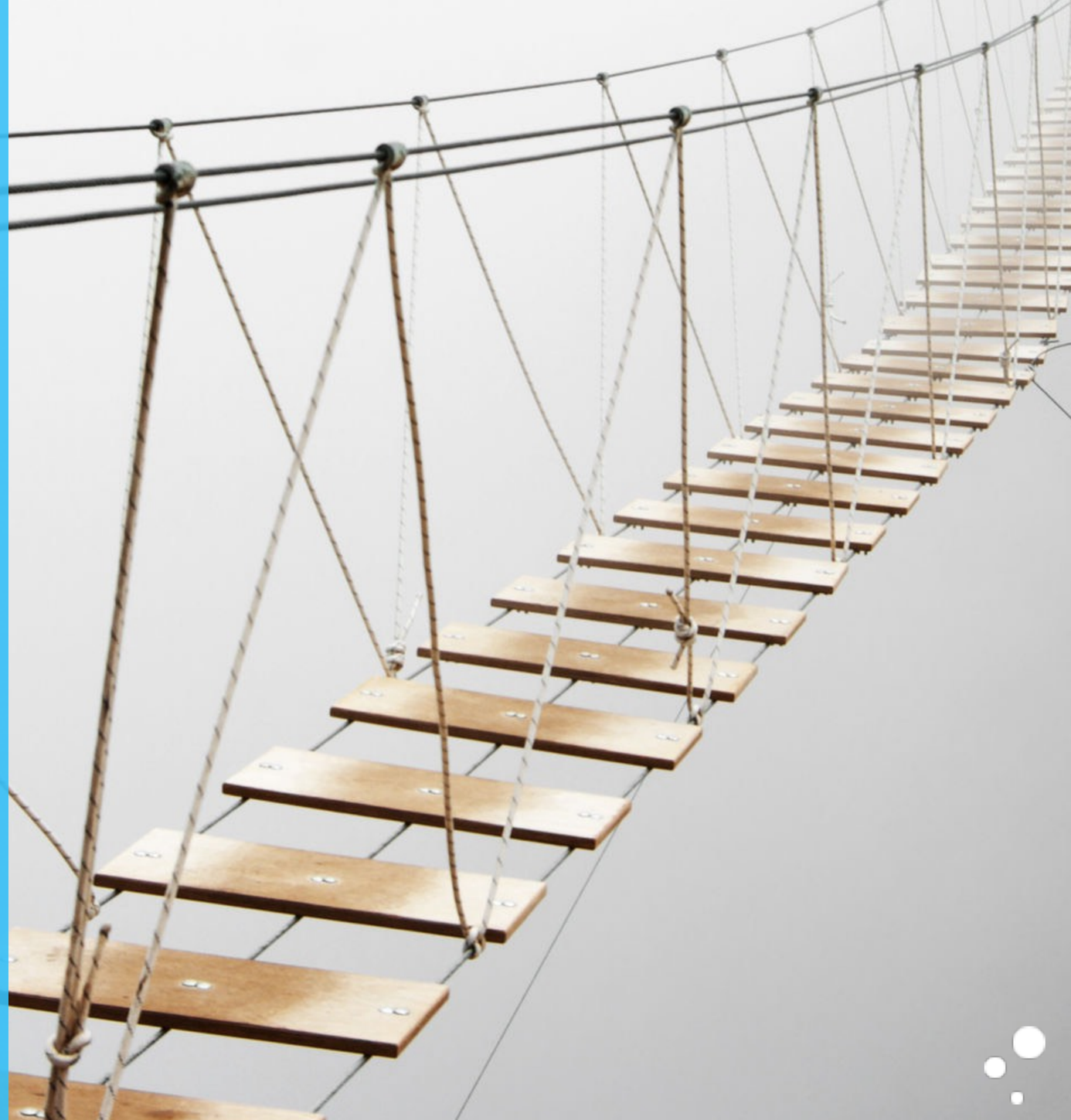
PentesterLab

JWT jku&x5u = ❤️

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

Crypto 101

Signature vs Encryption

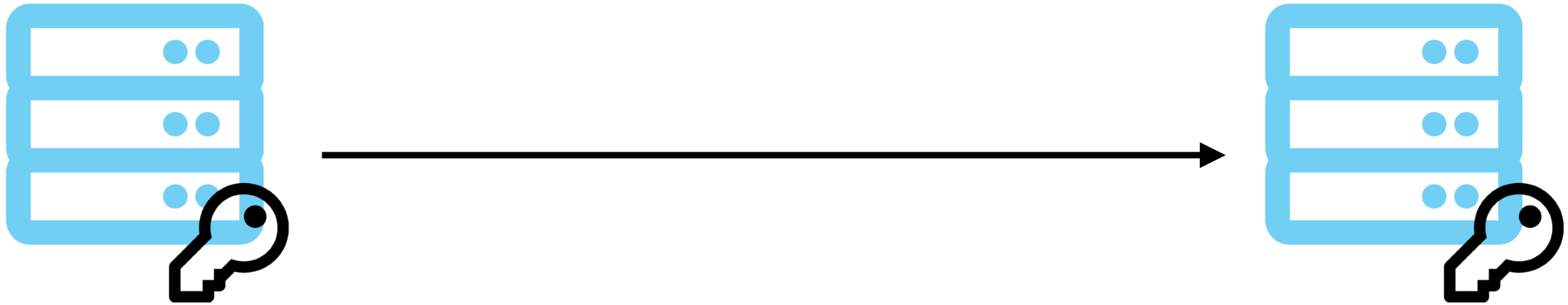
Encryption gives you **confidentiality**

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

.

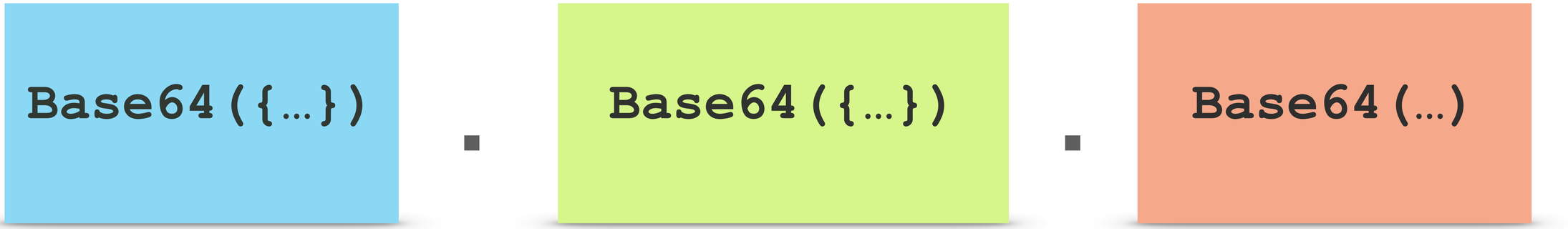
FSfvCBAwypJ4abF6
jFLmR7JgZhkW674
Z8dIdAIRyt1E

```
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

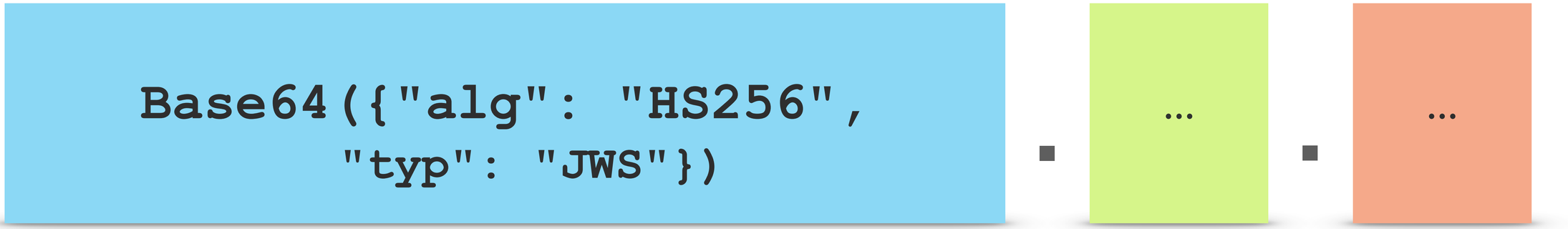
Urlsafe base64 encoding without padding:

```
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

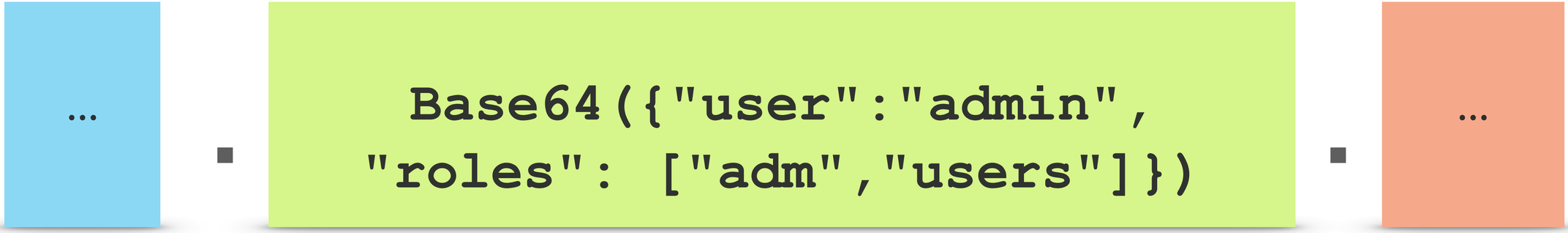
A lot of different algorithms are supported*:

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

* https://jwt.io/ covers most

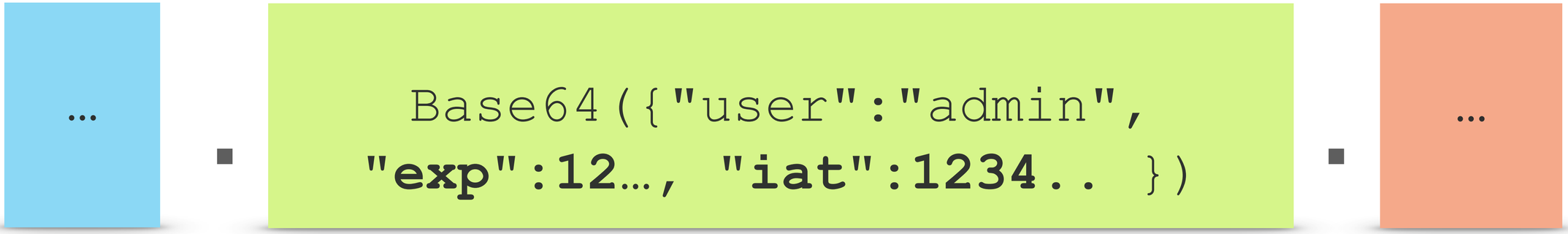
The JWT Format: payload

The payload may contain literally anything:



The JWT Format: payload

The payload **may** contain registered claims:



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

Classic JWT attacks

- None algorithm
- Trivial secret
- Algorithm confusion
- Injection in the kid parameter
- CVE-2018-0114
-

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)

The JWT Format: jku&x5u

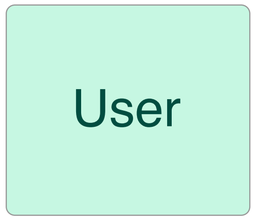
The JWT Format: jwk

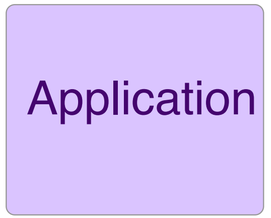
```
{
  "keys": [
    {
      "kty": "RSA",
      "use": "sig",
      "kid": "pentesterlab",
      "n": "oTtAXRgdJ6Pu0jr3hK3opCF5uqKWKbm4Kkq...vTF0FGw",
      "e": "AQAB",
      "alg": "RS256"
    }
  ]
}
```

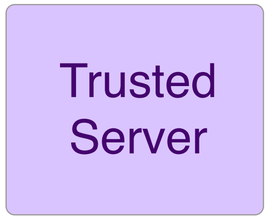
The JWT Format: x5c

```
{
  "keys": [
    {
      "kty": "RSA",
      "use": "sig",
      "kid": "pentesterlab",
      "x5c": "MIIDWDCCAkACCQCnE....fpye27SQbC2fBxebsek=",
      "alg": "RS256"
    }
  ]
}
```

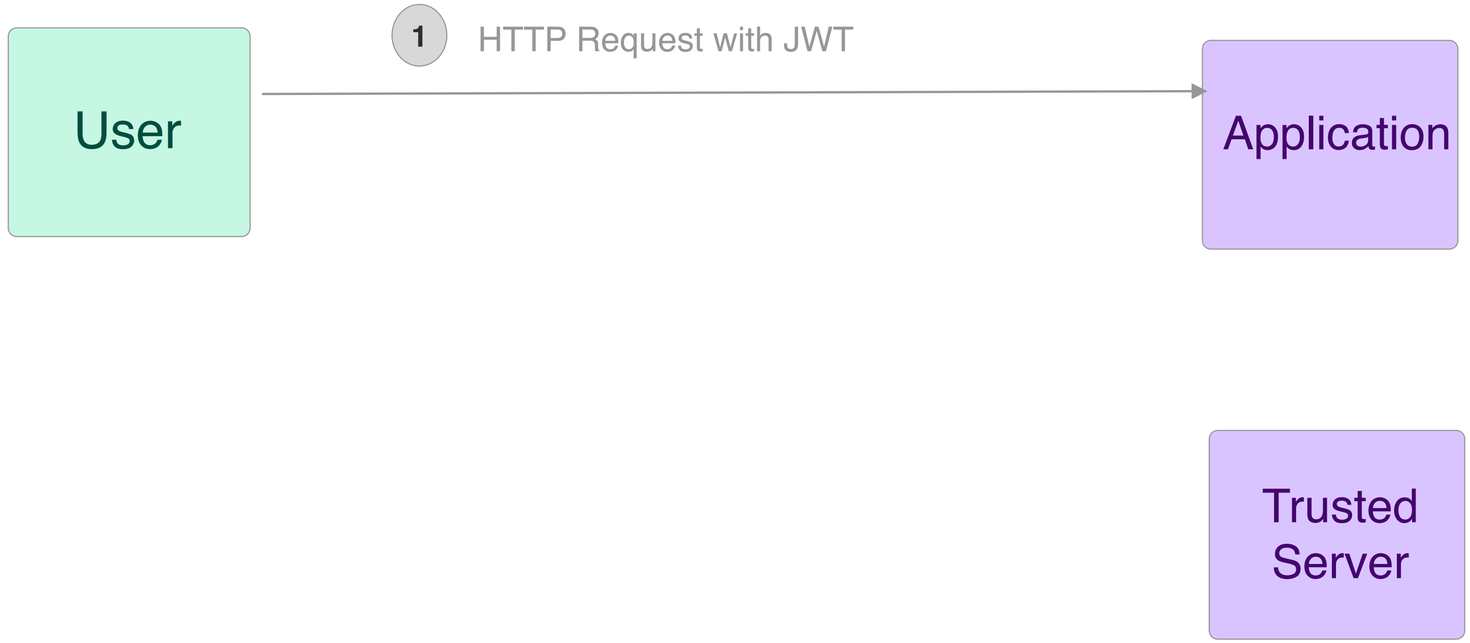
jku and x5u



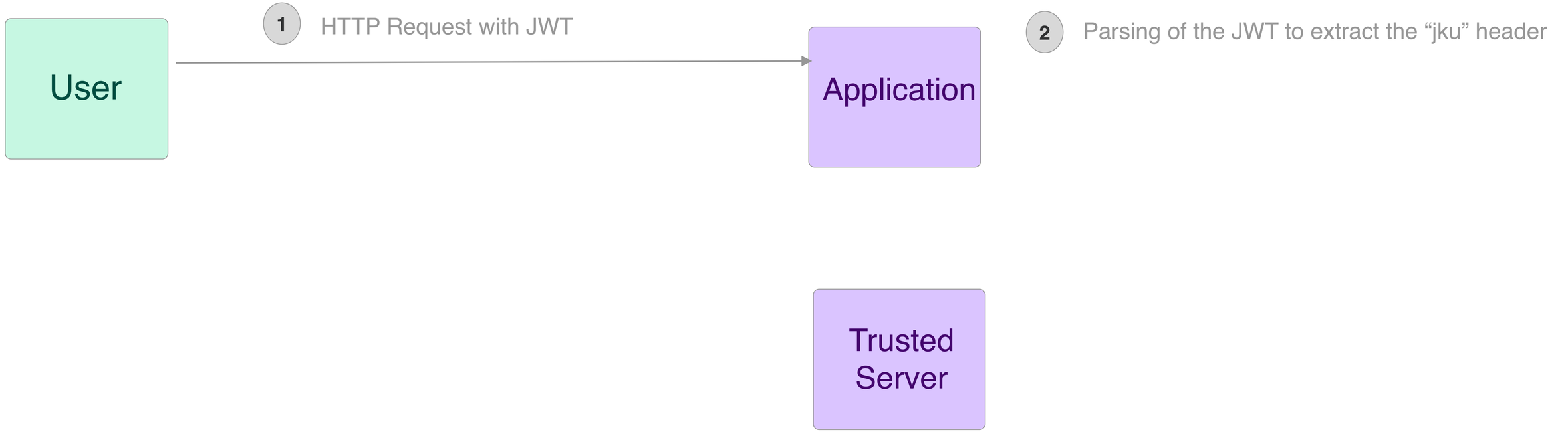




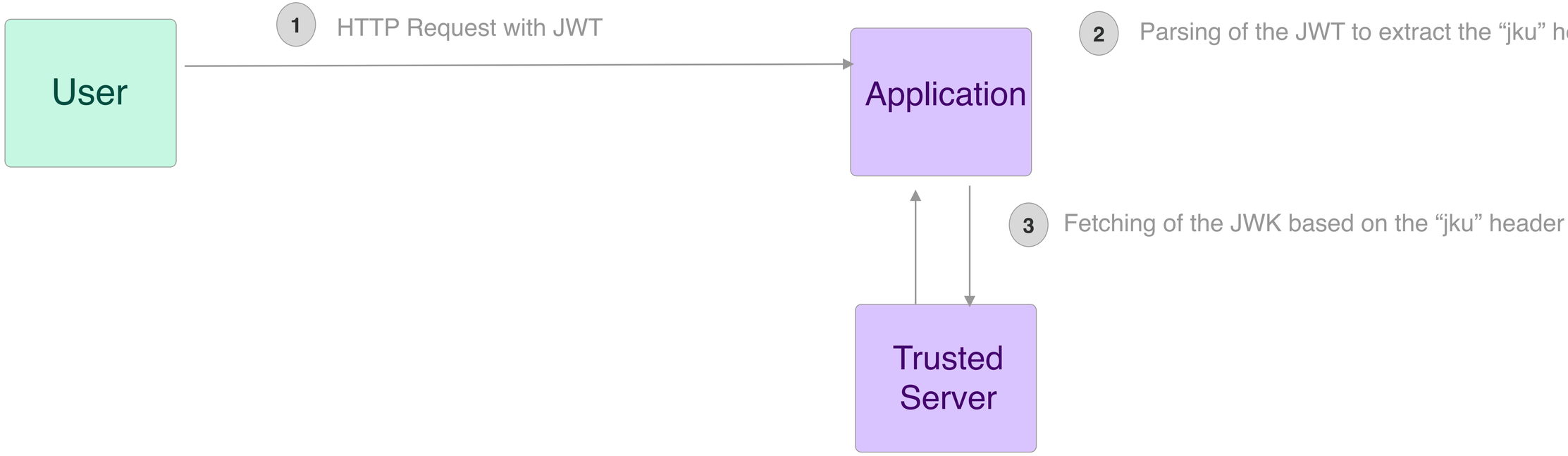
jku and x5u



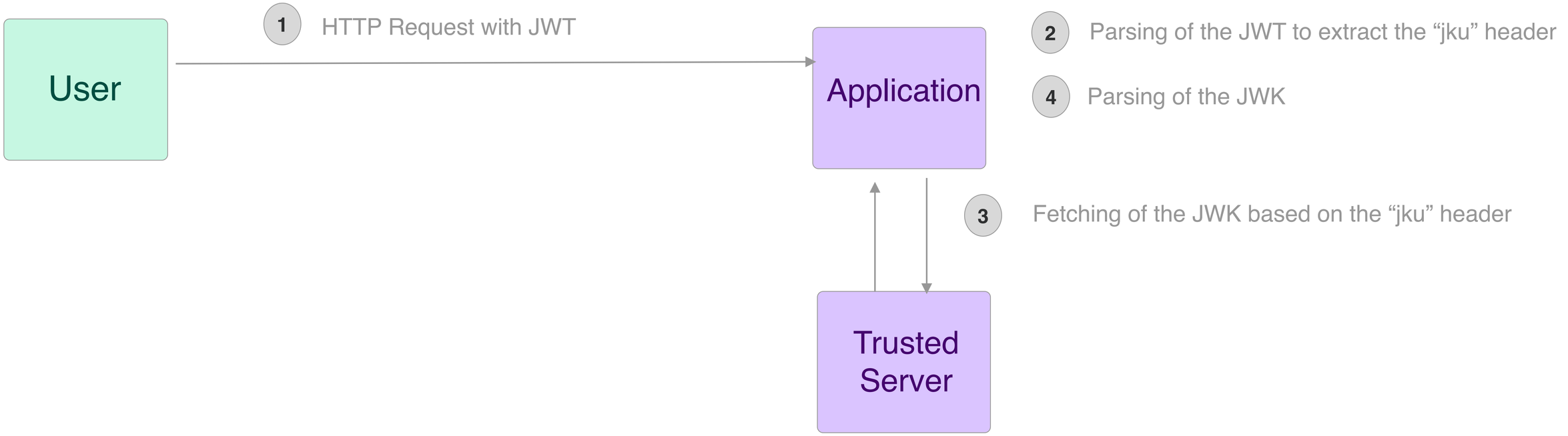
jku and x5u


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

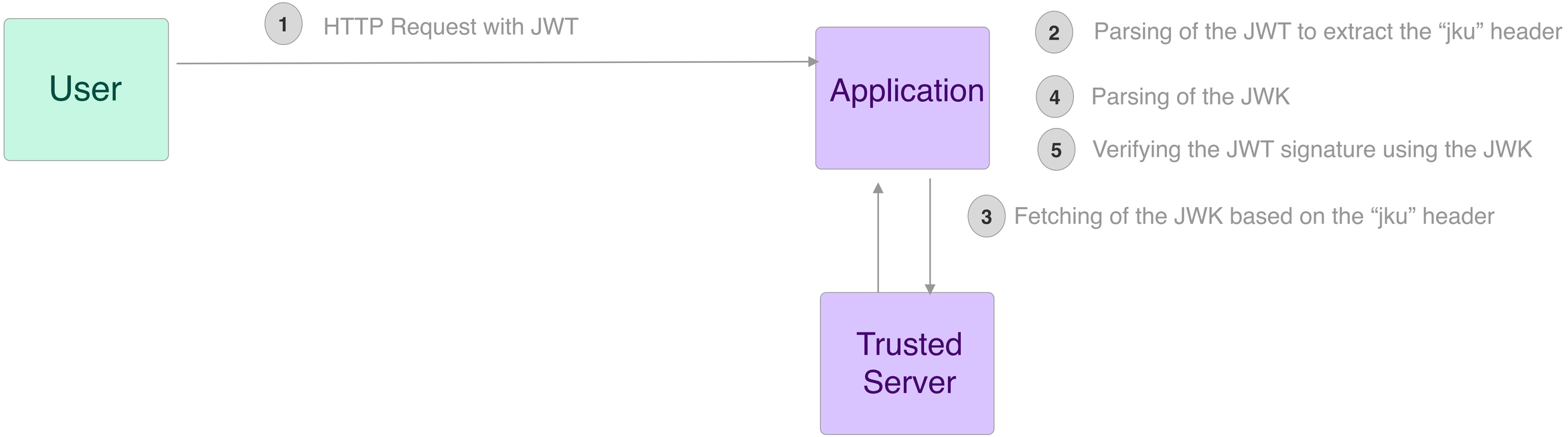
jku and x5u


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

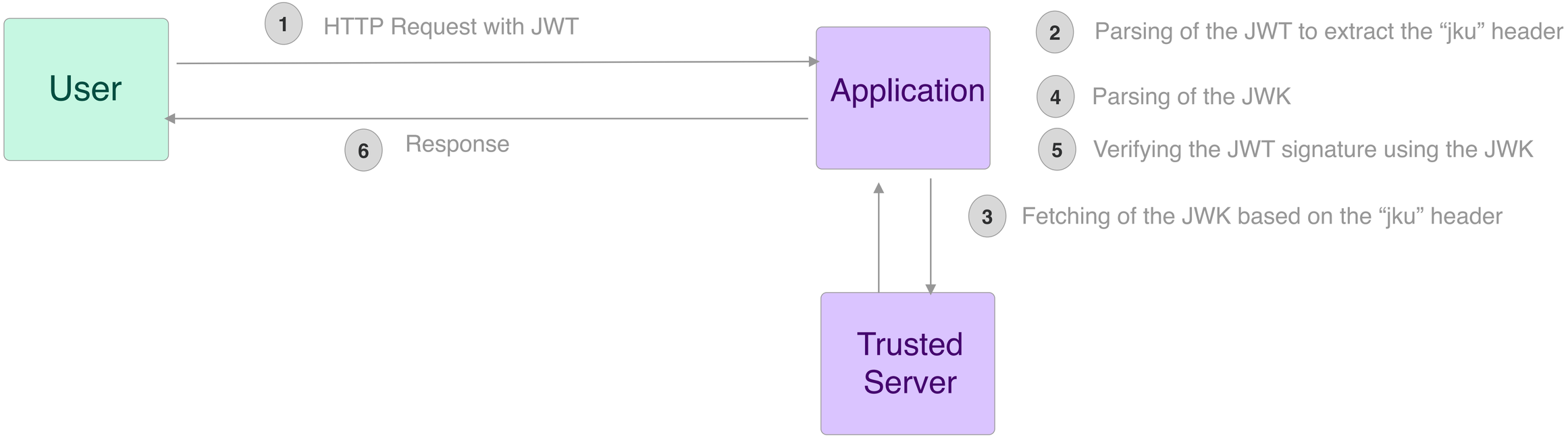
jku and x5u


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

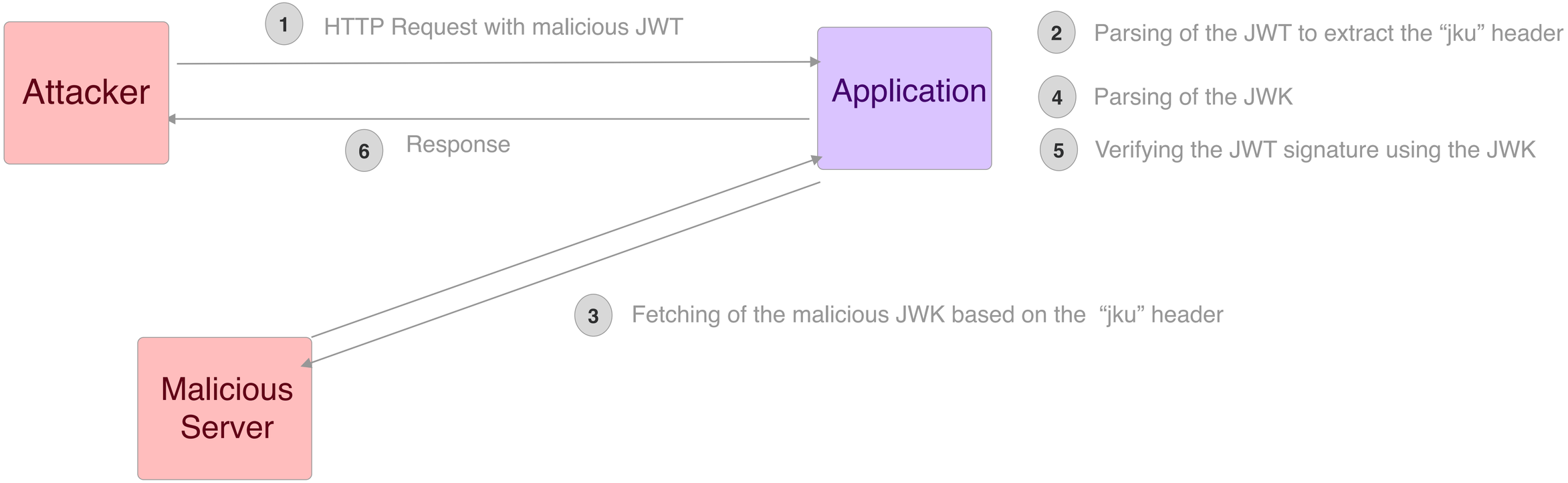
jku and x5u


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

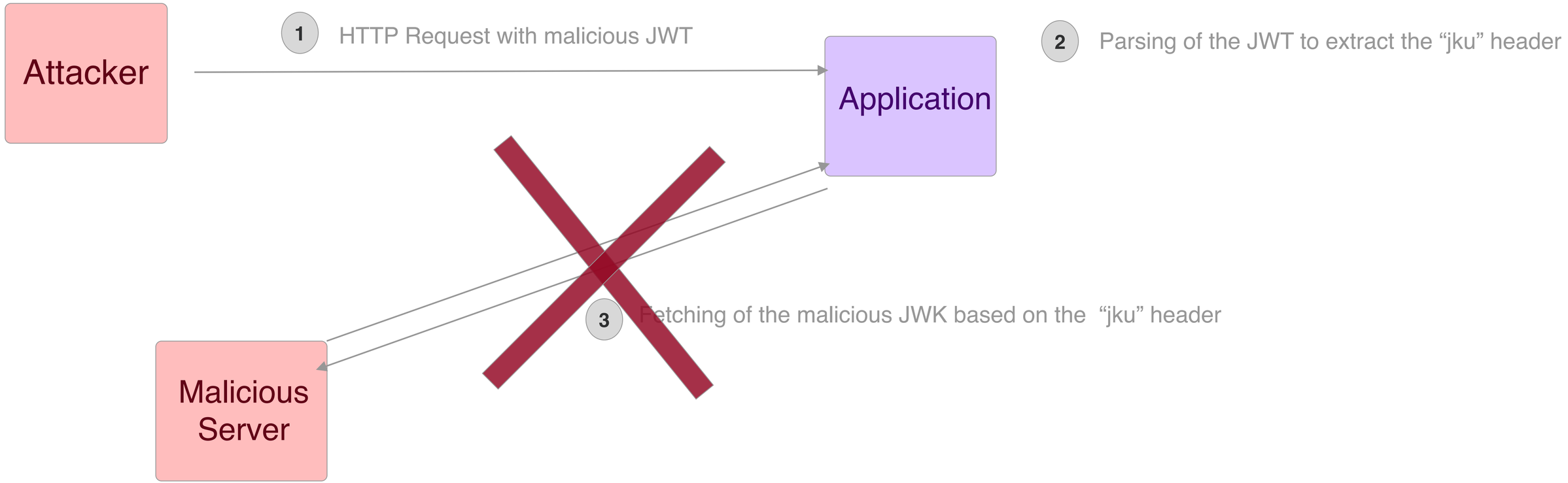
jku and x5u


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

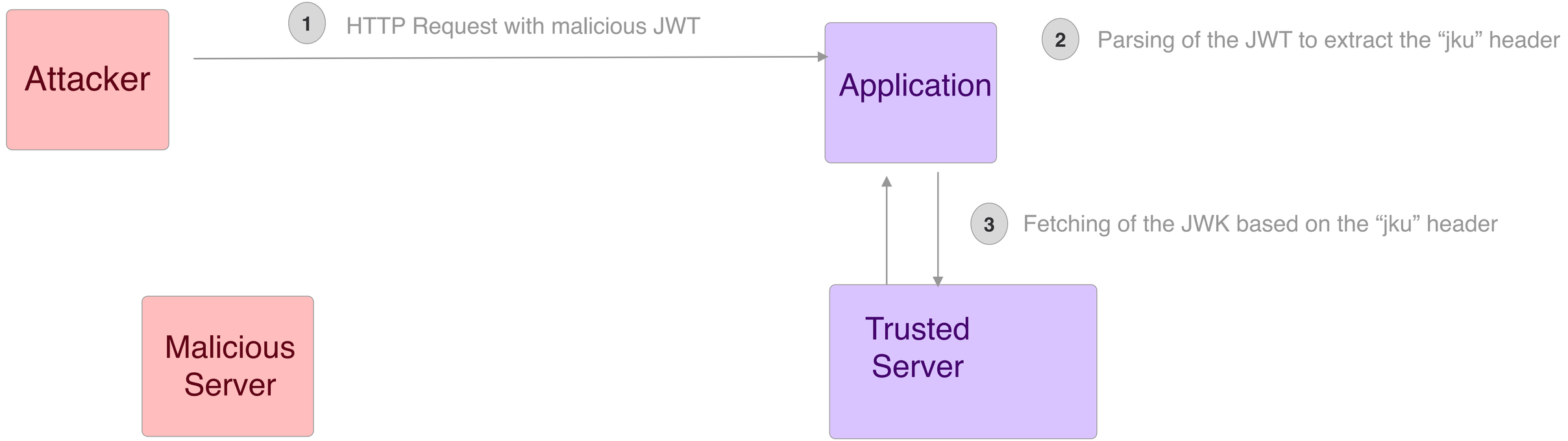
https://trusted => https://trusted@pentesterlab.com

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

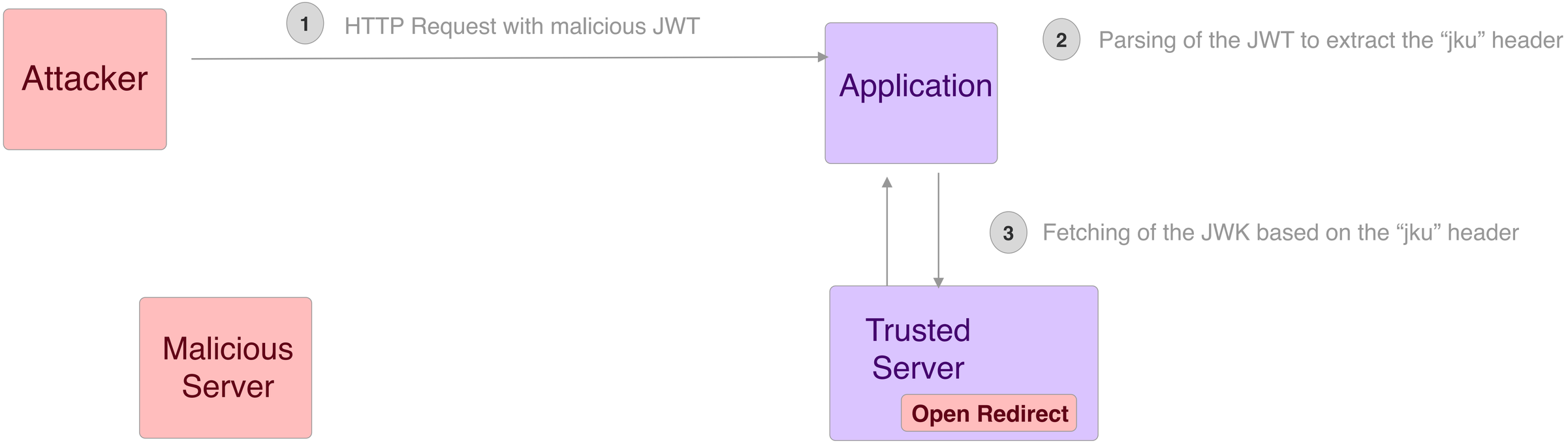
https://trusted/jwks/ => https://trusted/jwks/../open_redirect

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

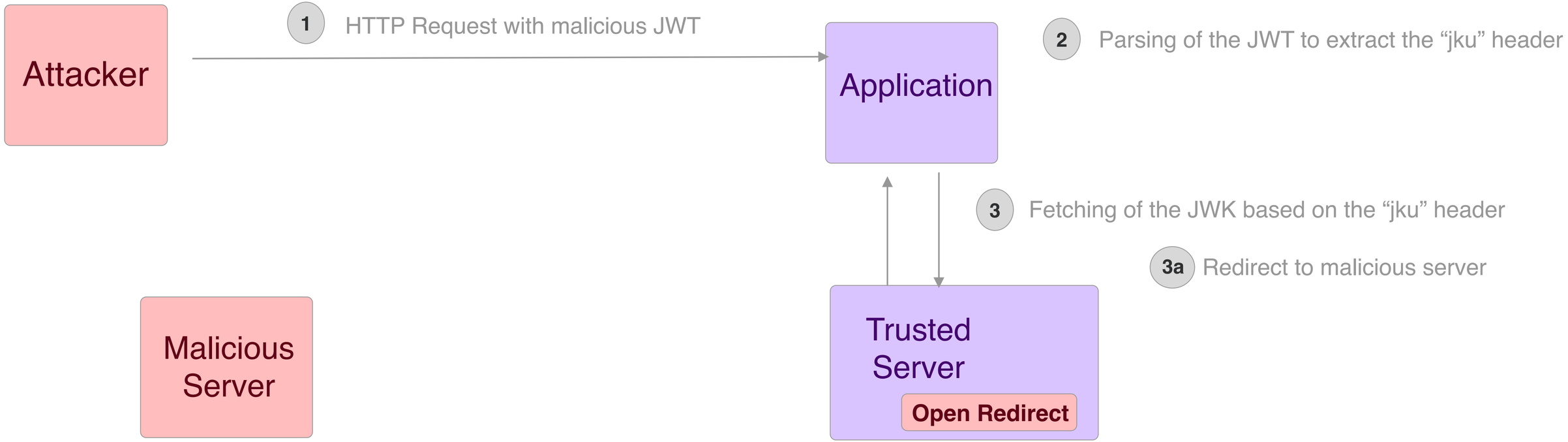
jku and Open Redirect



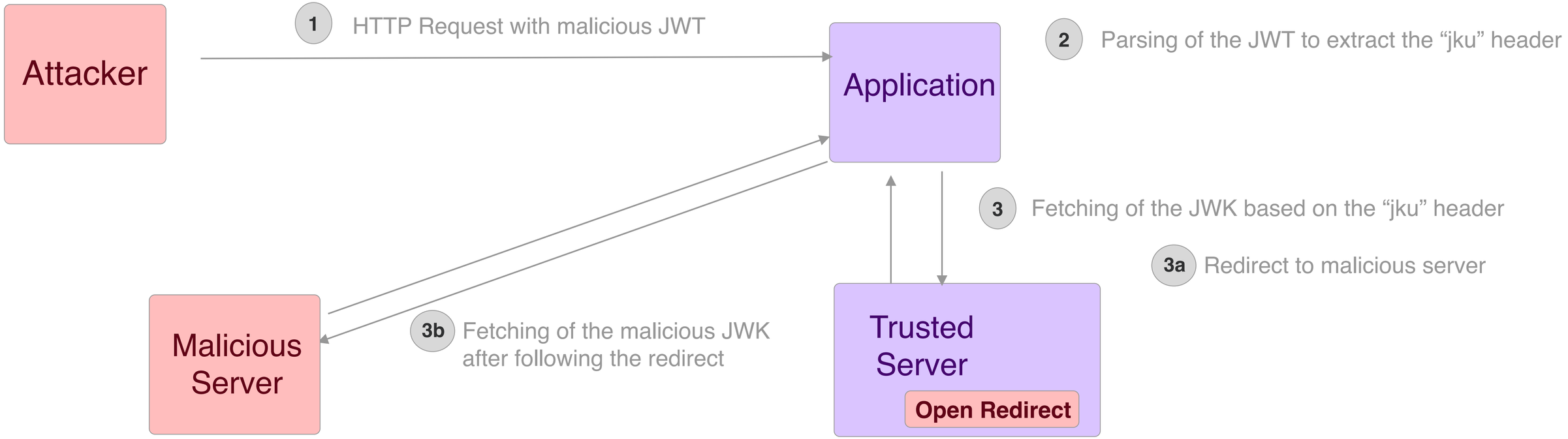
jku and Open Redirect



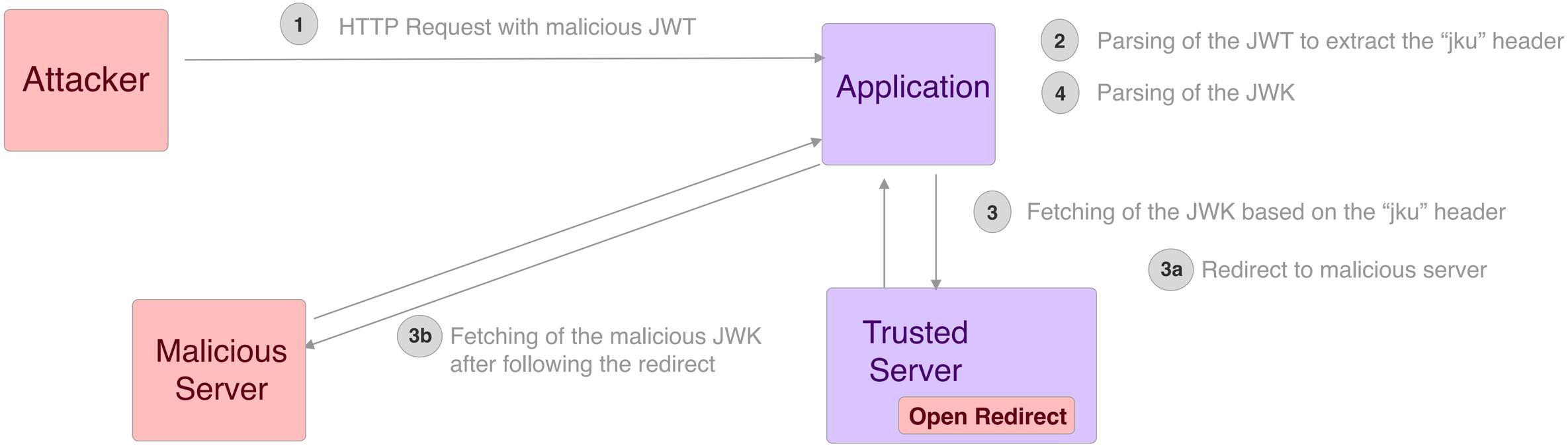
jku and Open Redirect



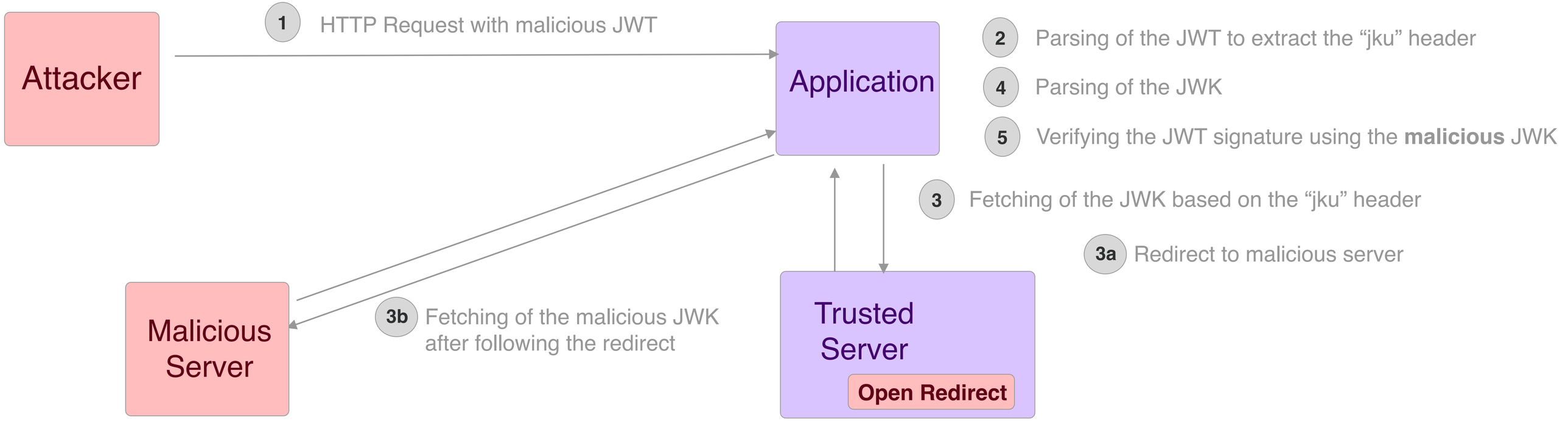
jku and Open Redirect



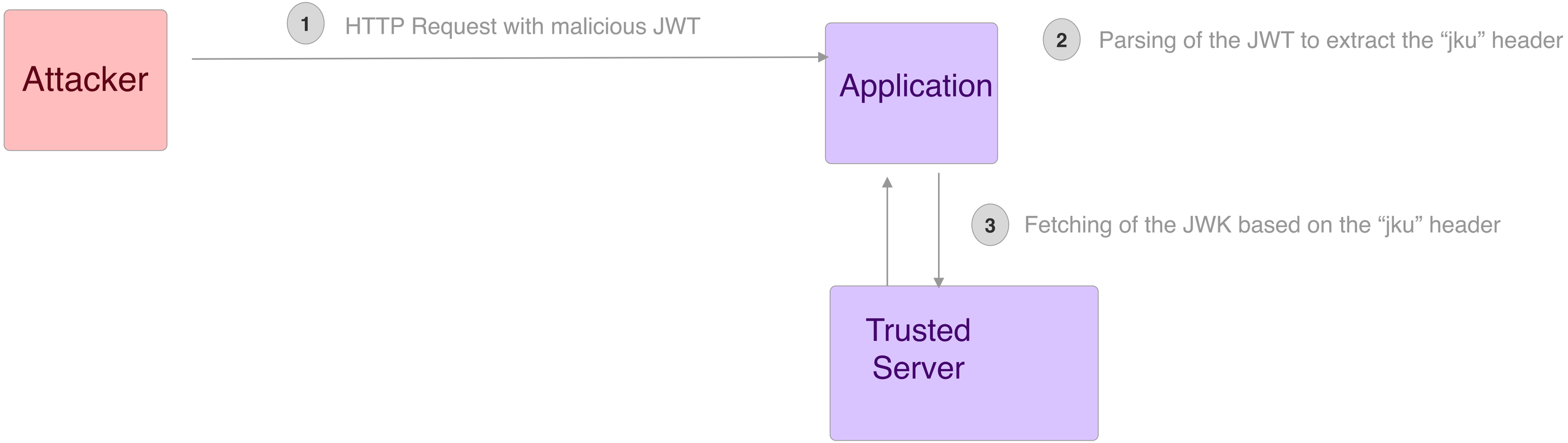
jku and Open Redirect



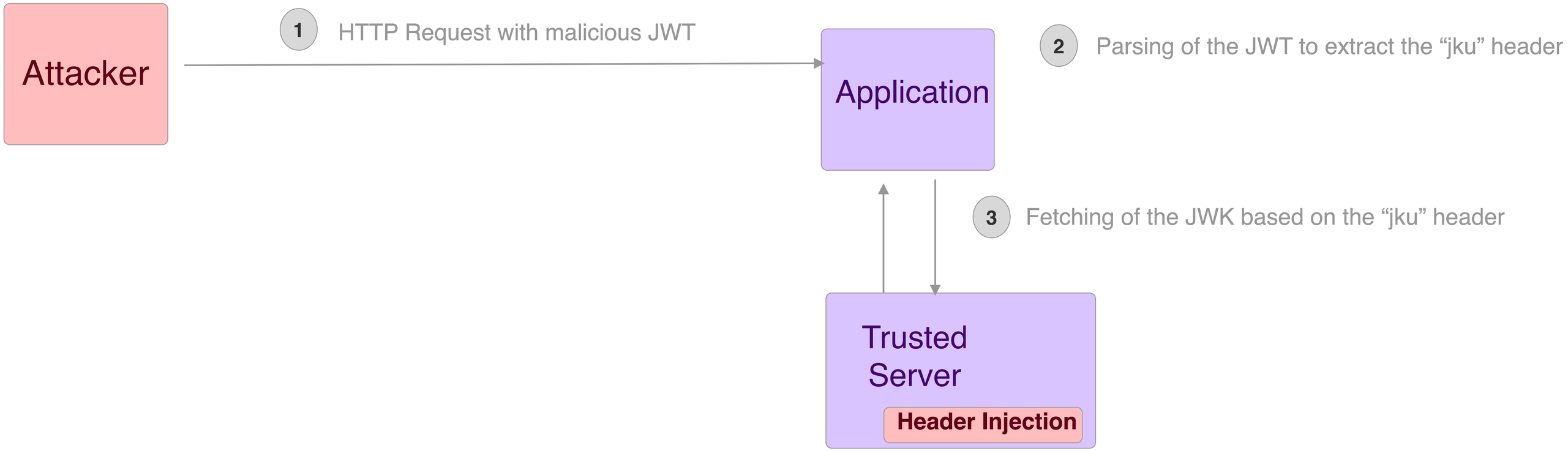
jku and Open Redirect



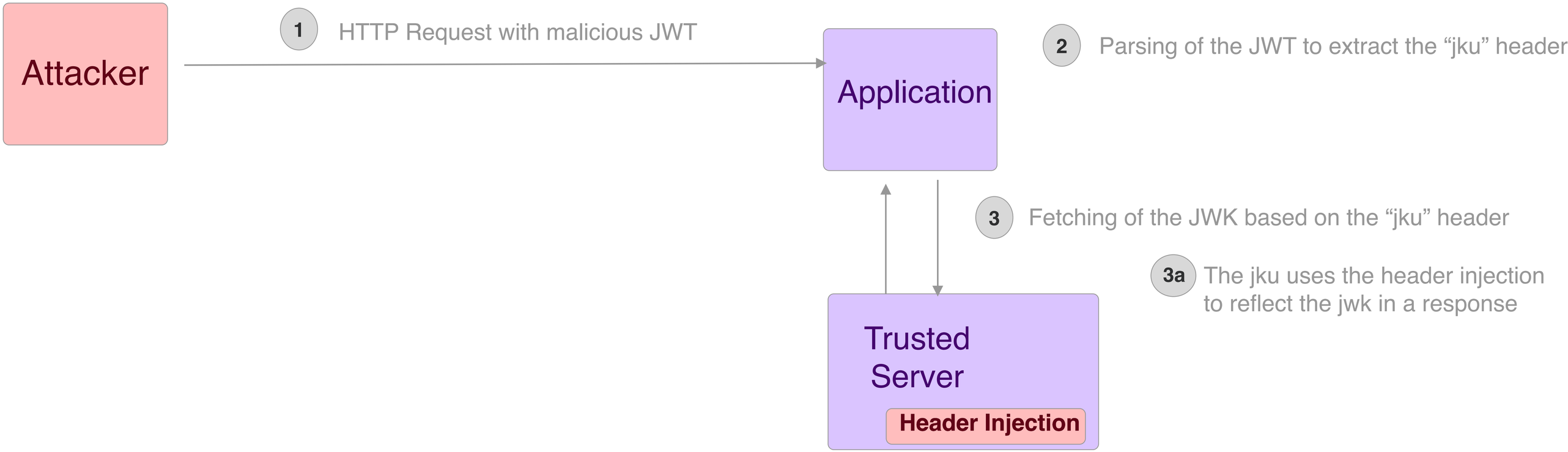
jku and Header Injection



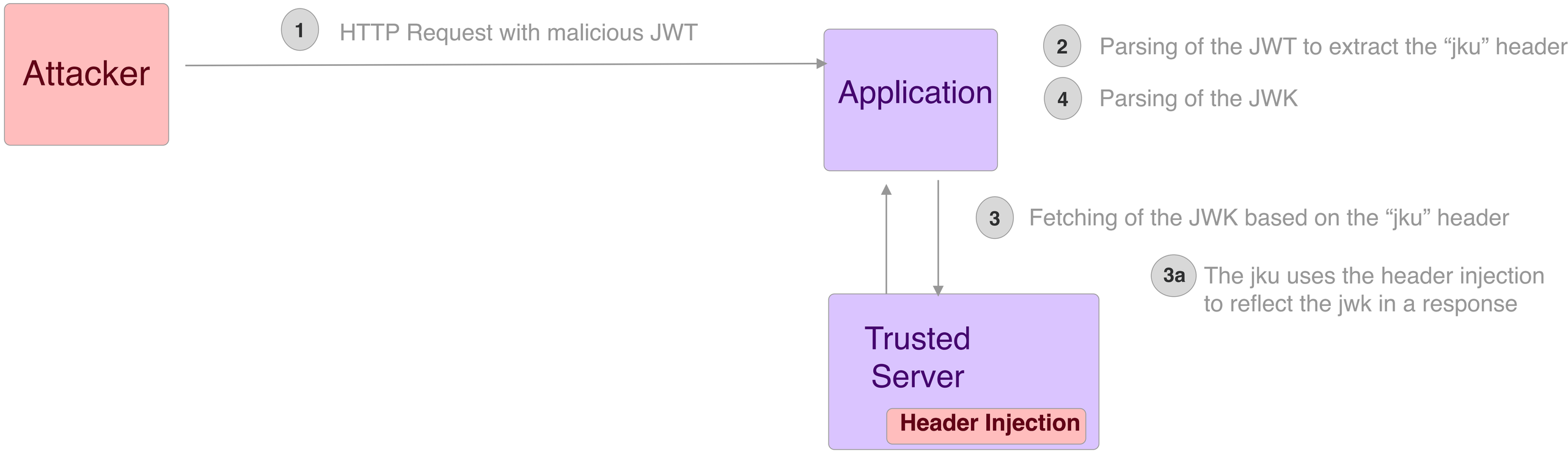
jku and Header Injection



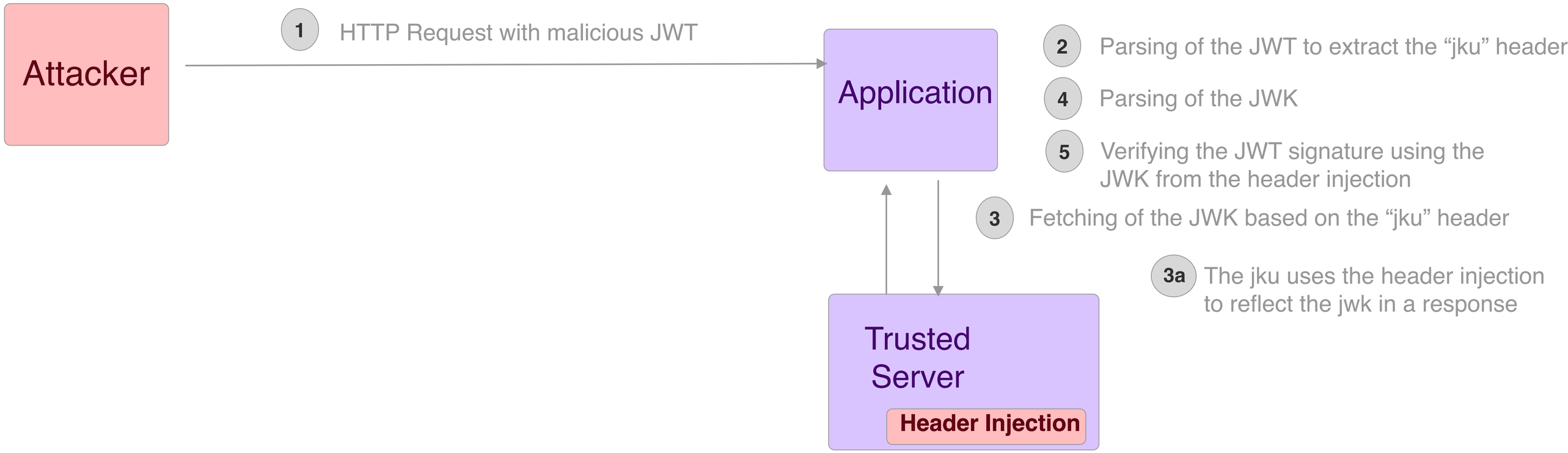
jku and Header Injection



jku and Header Injection



jku and Header Injection



Libraries: jku header injection - Exploitation

Exploitation:

- Find a Header Injection
- Use the Header Injection to return your JWK
- Add the Header Injection as jku
- Sign the token with your RSA key

jku and x5u: downgrade

- The RFC calls out enforcing TLS to avoid MITM

- Few implementations get it wrong:

  Enforcing when you set the value

  vs

  Enforcing when you fetch the key

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

Recommendations

- ✓ Test for x5u and jku
- ✓ Don't burn Open Redirect
- ✓ Read RFC
- ✓ Hack all the things!

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?

louis@pentesterlab.com / PentesterLab.com / @PentesterLab