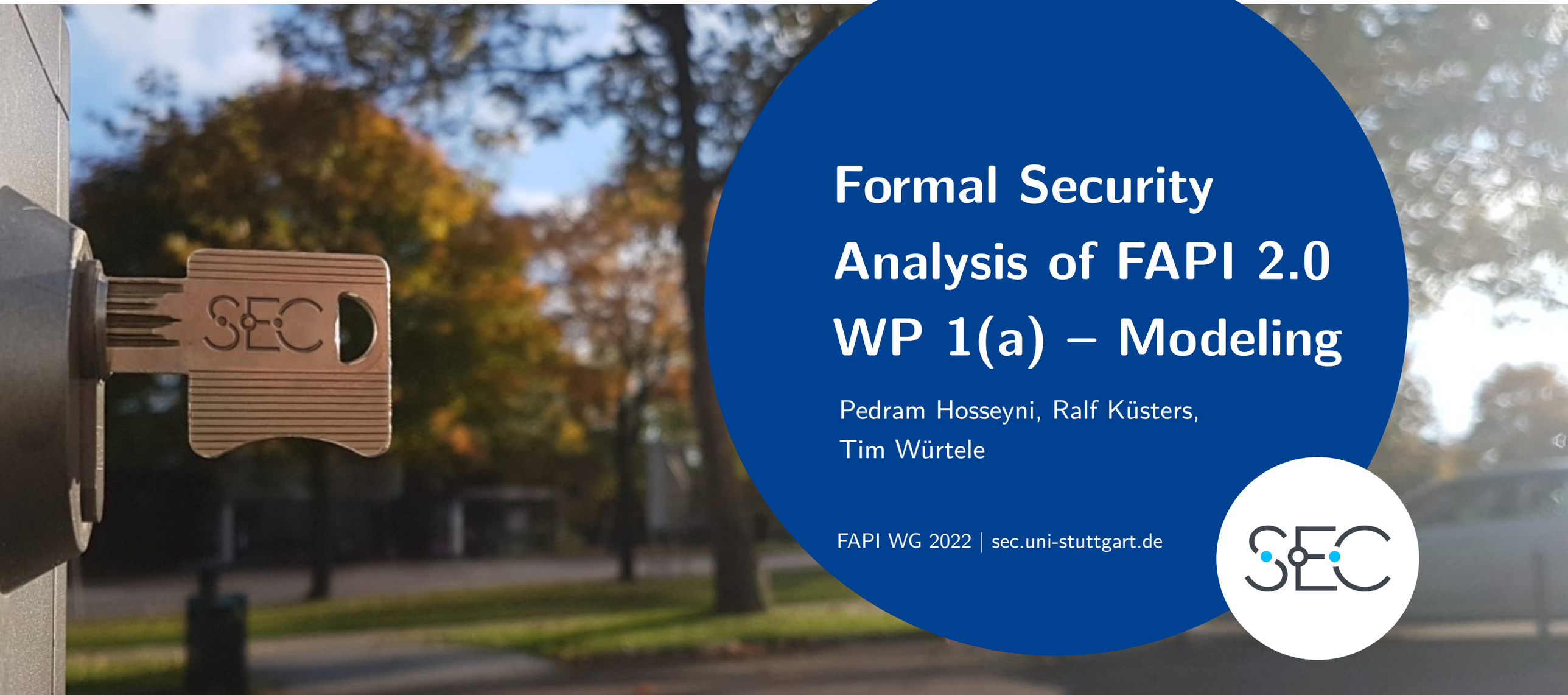
University of Stuttgart
Institute of
Information Security



Formal Security Analysis of FAPI 2.0 WP 1(a) – Modeling

Pedram Hosseyni, Ralf Küsters,
Tim Würtele

FAPI WG 2022 | sec.uni-stuttgart.de

SEC

Outline

- The Web Infrastructure Model
- Modeling FAPI 2.0 Security Profile
- Security Properties
- Modeling FAPI 2.0 Attacker Model
- Remarks & Discussion

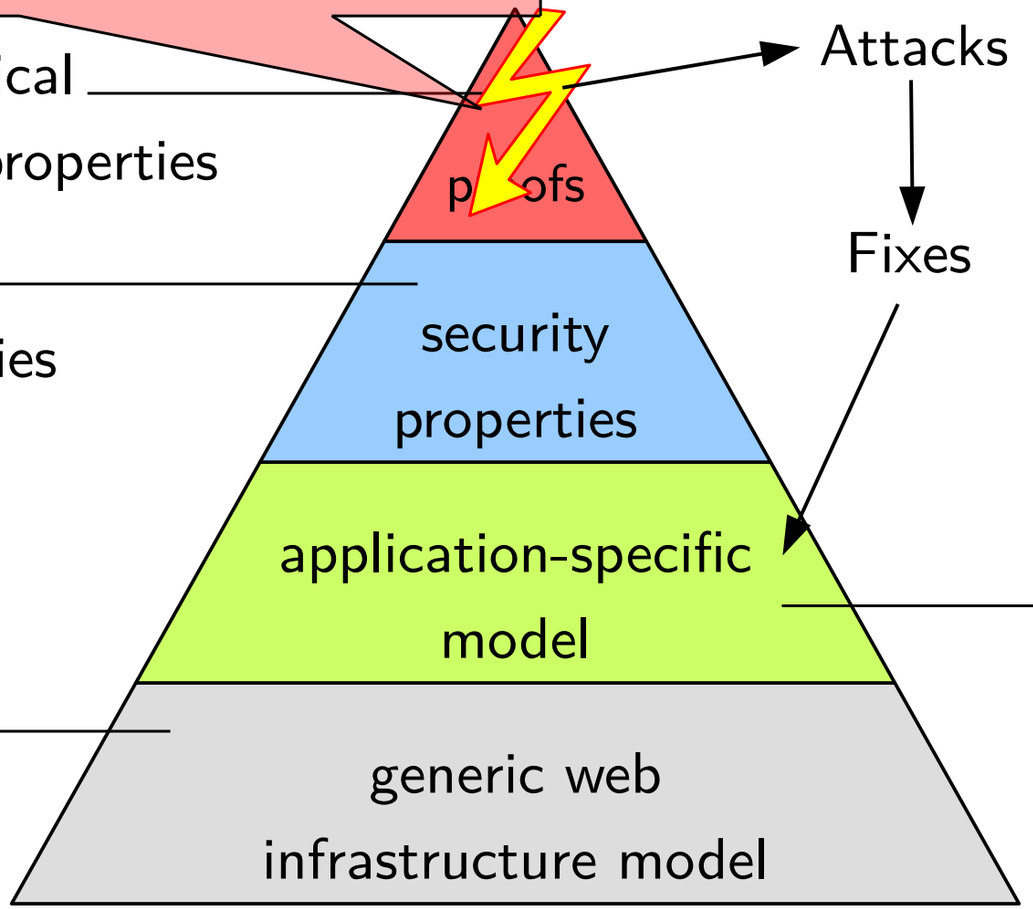
The Web Infrastructure Model (WIM)

The Web Infrastructure Model *WIM*

Mathematical proofs of properties

Precise formal security properties

Foundation: Formal description of the web



Attacks

Fixes

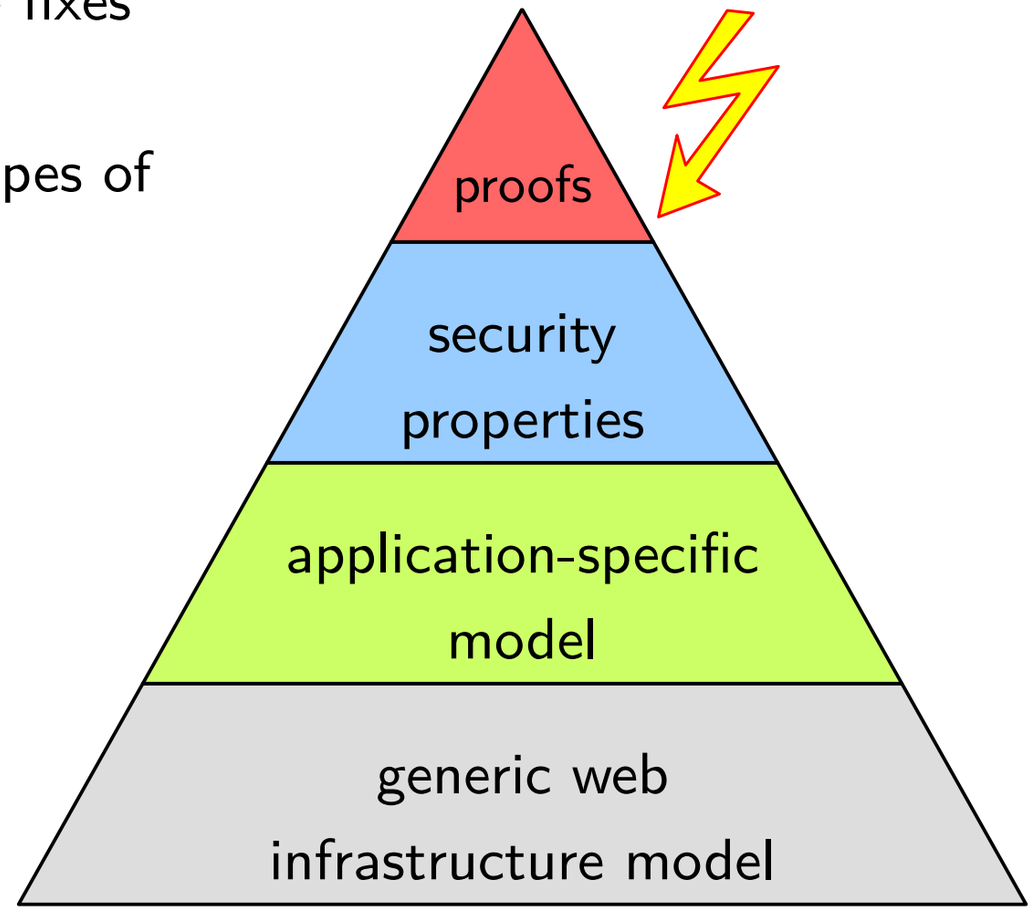
Repeat process until proof goes through.

Application model built (manually) from source code or specification

The Web Infrastructure Model *WIM*

This approach can yield...

- new attacks and respective fixes
- strong security guarantees excluding even unknown types of attacks



Prior WIM Analyses

- Mozilla BrowserID [SP2014, ESORICS2015]
- SPRESSO [CCS2015]
- OAuth 2.0 [CCS2016]
- OpenID Connect 1.0 [CSF2017]
- OpenID Financial Grade API 1.0 (FAPI) [SP2019]
- W3C Web Payments [SP2022]

Modeling FAPI 2.0 ~~Baseline~~ Security Profile

Modeled Components

- The usual OAuth & OIDC specs, e.g., RFC 6749, (parts of) RFC 6750, OIDC Core 1.0, ...
- Authorization Server Metadata (RFC 8414, OIDD)
- Pushed Authorization Requests (RFC 9126)
- PKCE (RFC 7636)
- Token Introspection (RFC 7662)
- Mutual TLS for OAuth (RFC 8705)
- Authorization Server Issuer Identification (RFC 9207)
- DPoP (Draft 08)
- Plus many related specs, such as: HTTP Basic Authentication (RFC 7617), JWK (RFC 7517), JWT (RFC 7523), OAuth Security BCP, ...

Modeling Pushed Authorization Requests

Algorithm 9 Relation of AS R^{as} – Processing HTTPS Requests

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2:     if m.path ≡ /.well-known/openid-configuration ∨
       ↪ m.path ≡ /.well-known/oauth-authorization-server then   → We model both OIDD and RFC 8414.
3:         let metaData := [issuer: ⟨URL, S, m.host, ε, ⟨⟩, ⊥⟩]
4:         let metaData[auth_ep] := ⟨URL, S, m.host, /auth, ⟨⟩, ⊥⟩
5:         let metaData[token_ep] := ⟨URL, S, m.host, /token, ⟨⟩, ⊥⟩
6:         let metaData[par_ep] := ⟨URL, S, m.host, /par, ⟨⟩, ⊥⟩
7:         let metaData[introspec_ep] := ⟨URL, S, m.host, /introspect, ⟨⟩, ⊥⟩
8:         let metaData[jwks_uri] := ⟨URL, S, m.host, /jwks, ⟨⟩, ⊥⟩
9:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, metaData⟩, k)
10:        stop ⟨⟨f, a, m'⟩⟩, s'
11:    else if m.path ≡ /jwks then
12:        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, pub(s'.jwk)⟩, k)
13:        stop ⟨⟨f, a, m'⟩⟩, s'
14:    else if m.path ≡ /auth then   → Authorization endpoint: Reply with login page.
15:        if m.method ≡ GET then
```

```
53:    else if m.path ≡ /par ∧ m.method ≡ POST then   → Pushed Authorization Request
54:        if m.body[response_type] ≢ code ∨ m.body[code_challenge_method] ≢ S256 then
55:            stop
56:        let authnResult := AUTHENTICATE_CLIENT(m, s')   → Stops in case of errors/failed authentication
57:        let clientId := authnResult.1
58:        let s' := authnResult.2
```

Modeling Pushed Authorization Requests

53: **else if** $m.\texttt{path} \equiv /\texttt{par} \wedge m.\texttt{method} \equiv \texttt{POST}$ **then** → Pushed Authorization Request
54: **if** $m.\texttt{body}[\texttt{response\_type}] \not\equiv \texttt{code} \vee m.\texttt{body}[\texttt{code\_challenge\_method}] \not\equiv \texttt{S256}$ **then**
55: **stop**
56: **let** $authnResult :=$ NTICATE_CLIENT(m, s') → Stops in case of errors/failed authentication
57: **let** $clientId :=$ a
58: **let** s'
59: **let** m
60: **if** cli
61: **s** body[client_id]
62: **let** r o send redirect_uri with each request
63: **if** re
64: **s**
65: **if** re
66: **s**
67: **let** $codeChallenge :=$ $m.\texttt{body}[\texttt{code\_challenge}]$ → PKCE challenge
68: **if** $codeChallenge \equiv \langle\rangle$ **then**
69: **stop** → Missing PKCE challenge
70: **let** $requestUri := \nu_4$ → Choose random URI
71: **let** $authzRecord := [\texttt{client\_id}: clientId]$
72: **let** $authzRecord[\texttt{state}] := m.\texttt{body}[\texttt{state}]$
73: **let** $authzRecord[\texttt{scope}] := m.\texttt{body}[\texttt{scope}]$
74: **if** $\texttt{nonce} \in m.\texttt{body}$ **then**
75: **let** $authzRecord[\texttt{nonce}] := m.\texttt{body}[\texttt{nonce}]$
76: **let** $authzRecord[\texttt{redirect\_uri}] := redirectUri$
77: **let** $authzRecord[\texttt{code\_challenge}] := codeChallenge$
78: **let** $s'.\texttt{authorizationRequests}[requestUri] := authzRecord$ → Store data linked to $requestUri$
79: **let** $m' := \texttt{enc}_\texttt{s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 201, \langle\rangle, [\texttt{request\_uri}: requestUri]\rangle, k)$
80: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'

"Authorization servers shall support the authorization code grant [...] described in [RFC6749]"

"Authorization servers shall reject requests using the resource owner password credentials grant or the implicit grant described in [RFC6749] or the hybrid flow as described in [OIDC]"

Modeling Pushed Authorization Requests

```
53:  else if m.path ≡ /par ∧ m.method ≡ POST then    → Pushed Authorization Request
54:      if m.body[response_type] ≢ code ∨ m.body[code_challenge_method] ≢ S256 then
55:          stop
56:      let authnResult := AUTHENTICATE_CLIENT(m, s′)    → Stops in case of errors/failed authentication
57:      let clientId := authnResult.1
58:      let s′ := authnResult.2
59:      let mtlsInfo := authnResult.3
60:      if clientId ≢ m.body[client_id] then
61:          stop    → Key used in client authentication is not registered for m.body[client_id]
62:      let redirectUri := m.body[redirect_uri]    → Clients are required to send redirect_uri with each request
63:      if redirectUri ≡ ⟨⟩ then
64:          stop
65:      if redirectUri.protocol ≢ S then
66:          stop
67:      let codeChallenge := m.body[code_challenge]    → PKCE challenge
68:      if codeChallenge ≡ ⟨⟩ then
69:          stop    → Missing PKCE challenge
70:      let requestUri := ν4    → Choose random URI
71:      let authzRecord := [client_id: clientId]
72:      let authzRecord[state] := m.body[state]
73:      let authzRecord[scope] := m.body[scope]
74:      if nonce ∈ m.body then
75:          let authzRecord[nonce] := m.body[nonce]
76:      let authzRecord[redirect_uri] := redirectUri
77:      let authzRecord[code_challenge] := codeChallenge
78:      let s′.authorizationRequests[requestUri] := authzRecord    → Store data linked to requestUri
79:      let m′ := enc_s(⟨HTTPResp, m.nonce, 201, ⟨⟩, [request_uri: requestUri]⟩, k)
80:      stop ⟨⟨f, a, m′⟩⟩, s′
```

"Authorization servers shall authenticate clients using one of the following methods:

- MTLS as specified in [...] [RFC8705]
- DPoP as described in [I-D.ietf-oauth-dpop]"

Modeling Pushed Authorization Requests

```
53:  else if m.path ≡ /par ∧ m.method ≡ POST then    → Pushed Authorization Request
54:      if m.body[response_type] ≢ code ∨ m.body[code_challenge_method] ≢ S256 then
55:          stop
56:      let authnResult := AUTHENTICATE_CL                                  hentication
57:      let clientId := authnResult.1
58:      let s′ := authnResult.2
59:      let mtlsInfo := authnResult.3
60:      if clientId ≢ m.body[client_id] then
61:          stop     → Key used in client authentication
62:      let redirectUri := m.body[redirect_uri]    → Clients are required to send redirect_uri with each request
63:      if redirectUri ≡ ⟨⟩ then
64:          stop
65:      if redirectUri.protocol ≢ S then
66:          stop
67:      let codeChallenge := m.body[code_challenge]    → PKCE challenge
68:      if codeChallenge ≡ ⟨⟩ then
69:          stop     → Missing PKCE challenge
70:      let requestUri := ν4    → Choose random URI
71:      let authzRecord := [client_id: clientId]
72:      let authzRecord[state] := m.body[state]
73:      let authzRecord[scope] := m.body[scope]
74:      if nonce ∈ m.body then
75:          let authzRecord[nonce] := m.body[nonce]
76:      let authzRecord[redirect_uri] := redirectUri
77:      let authzRecord[code_challenge] := codeChallenge
78:      let s′.authorizationRequests[requestUri] := authzRecord    → Store data linked to requestUri
79:      let m′ := enc_s(⟨HTTPResp, m.nonce, 201, ⟨⟩, [request_uri: requestUri]⟩, k)
80:      stop ⟨⟨f, a, m′⟩⟩, s′
```

"The pushed authorization request endpoint is an HTTP API at the authorization server that accepts HTTP POST requests" – RFC 9126

Modeling Pushed Authorization Requests

53: **else if** $m.\texttt{path} \equiv \texttt{/par} \wedge m.\texttt{method} \equiv \texttt{POST}$ **then** → Pushed Authorization Request
54: **if** $m.\texttt{body}[\texttt{response\_type}] \not\equiv \texttt{code} \vee m.\texttt{body}[\texttt{code\_challenge\_method}] \not\equiv \texttt{S256}$ **then**
55: **stop**
56: **let** $authnResult$:= AUTHENTICATE_CLIENT(m, s') → Stops in case of err authentication
57: **let** $clientId := authnResult.1$
58: **let** $s' := authnResult.2$
59: **let** $mtlsInfo := authnResult.3$
60: **if** $clientId \not\equiv m.\texttt{body}[\texttt{client\_id}]$ **then**
61: **stop** → Key used in client authentication is not registered for $m.bo$
62: **let** $redirectUri := m.\texttt{body}[\texttt{redirect\_uri}]$ → Clients are required to s
63: **if** $redirectUri \equiv \langle\rangle$ **then**
64: **stop**
65: **if** $redirectUri.\texttt{protocol} \not\equiv \texttt{S}$ **then**
66: **stop**
67: **let** $codeChallenge := m.\texttt{body}[\texttt{code\_challenge}]$ PKCE challenge
68: **if** $codeChallenge \equiv \langle\rangle$ **then**
69: **stop** → Missing PKCE challenge
70: **let** $requestUri := \nu_4$ → Choose random URI
71: **let** $authzRecord := [\texttt{client\_id}: clientId]$
72: **let** $authzRecord[\texttt{state}] := m.\texttt{body}[\texttt{state}]$
73: **let** $authzRecord[\texttt{scope}] := m.\texttt{body}[\texttt{scope}]$
74: **if** $\texttt{nonce} \in m.\texttt{body}$ **then**
75: **let** $authzRecord[\texttt{nonce}] := m.\texttt{body}[\texttt{nonce}]$
76: **let** $authzRecord[\texttt{redirect\_uri}] := redirectUri$
77: **let** $authzRecord[\texttt{code\_challenge}] := codeChallenge$
78: **let** $s'.\texttt{authorizationRequests}[requestUri] := authzRecord$ → Store data linked to $requestUri$
79: **let** $m' := \texttt{enc}_\texttt{s}(\langle\texttt{HTTPResp}, m.\texttt{nonce}, 201, \langle\rangle, [\texttt{request\_uri}: requestUri]\rangle, k)$
80: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'

"Authorization servers shall require PKCE [RFC7636] with S256 as the code challenge method"

Modeling Pushed Authorization Requests

```
53:  else if m.path ≡ /par ∧ m.method ≡ POST then    → Pushed Authorization Request
54:      if m.body[response_type] ≢ code ∨ m.body[code_challenge_method] ≢ S256 then
55:          stop
56:      let authnResult := AUTHENTICATE_CLIENT(m, s′)    → Sto
57:      let clientId := authnResult.1
58:      let s′ := authnResult.2
59:      let mtlsInfo := authnResult.3
60:      if clientId ≢ m.body[client_id] then
61:          stop    → Key used in client authentication is not re
62:      let redirectUri := m.body[redirect_uri]    → C  nts are requ
63:      if redirectUri ≡ ⟨⟩ then
64:          stop
65:      if redirectUri.protocol ≢ S then
66:          stop
67:      let codeChallenge := m.body[code_challenge]    → PKCE
68:      if codeChallenge ≡ ⟨⟩ then
69:          stop    → Missing PKCE challenge
70:      let requestUri := ν4    → Choose random URI
71:      let authzRecord := [client_id: clientId]
72:      let authzRecord[state] := m.body[state]
73:      let authzRecord[scope] := m.body[scope]
74:      if nonce ∈ m.body then
75:          let authzRecord[nonce] := m.body[nonce]
76:      let authzRecord[redirect_uri] := redirectUri
77:      let authzRecord[code_challenge] := codeChallenge
78:      let s′.authorizationRequests[requestUri] := authzRecord    → Store data linked to requestUri
79:      let m′ := enc_s(⟨HTTPResp, m.nonce, 201, ⟨⟩, [request_uri: requestUri]⟩, k)
80:      stop ⟨⟨f, a, m′⟩⟩, s′
```

"Authorization servers shall require the redirect_uri parameter in pushed authorization requests"
"clients [...] shall only offer TLS protected endpoints"

"The "request_uri" value [...] MUST contain some part generated using a cryptographically strong pseudorandom algorithm such that it is computationally infeasible to predict or guess a valid value." – RFC 9126

"The "request_uri" value MUST be bound to the client that posted the authorization request." – RFC 9126

Security Properties

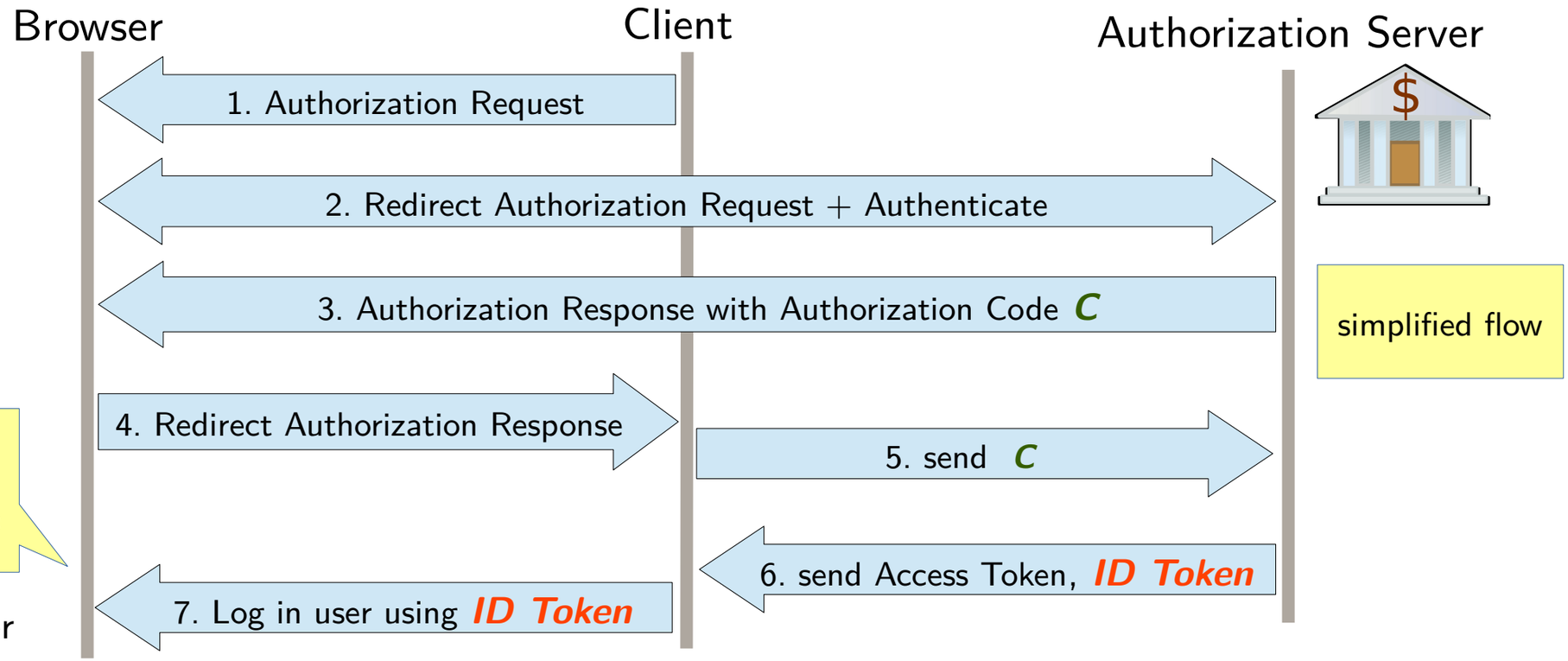
Security Goals

- FAPI 2.0 Attacker Model specifies three security goals
- Authorization: “no attacker can access resources belonging to a user”
- Authentication: “no attacker is able to log in at a client under the identity of a user”
- Session Integrity:
  - “no attacker is able to force a user to use resources of the attacker”
  - “no attacker is able to force a user to be logged in under the identity of the attacker”

FAPI 2.0 Attacker Model: https://bitbucket.org/openid/fapi/src/209f58afbd41fb20ab3ed65ca4e2f67ffd5dda77/FAPI_2_0_Attacker_Model.md

Example: Authentication

"no attacker is able to log in at a client under the identity of a user"



simplified flow

in the model: client sets a session identifier cookie

→ Formalized property: The attacker cannot get a session identifier cookie for a session of an honest user

Definition 11 (Authentication Property). We say that the FAPI web system with a network attacker $\mathcal{FAPI}$ *is secure w.r.t. authentication* iff for every run ρ of $\mathcal{FAPI}$, every configuration (S, E, N) in ρ, every $c \in \mathsf{C}$ that is honest in S, every identity $id \in \mathsf{ID}$ with $as = \mathsf{governor}(id)$ being an honest AS and with $b = \mathsf{ownerOfID}(id)$ being an honest browser in S, every service session identified by some nonce n for id at c, n is not derivable from the attackers knowledge in S (i.e., $n \notin d_{\emptyset}(S(\mathsf{attacker}))$).

Modeling FAPI 2.0 Attacker Model

FAPI 2.0 Attacker Model

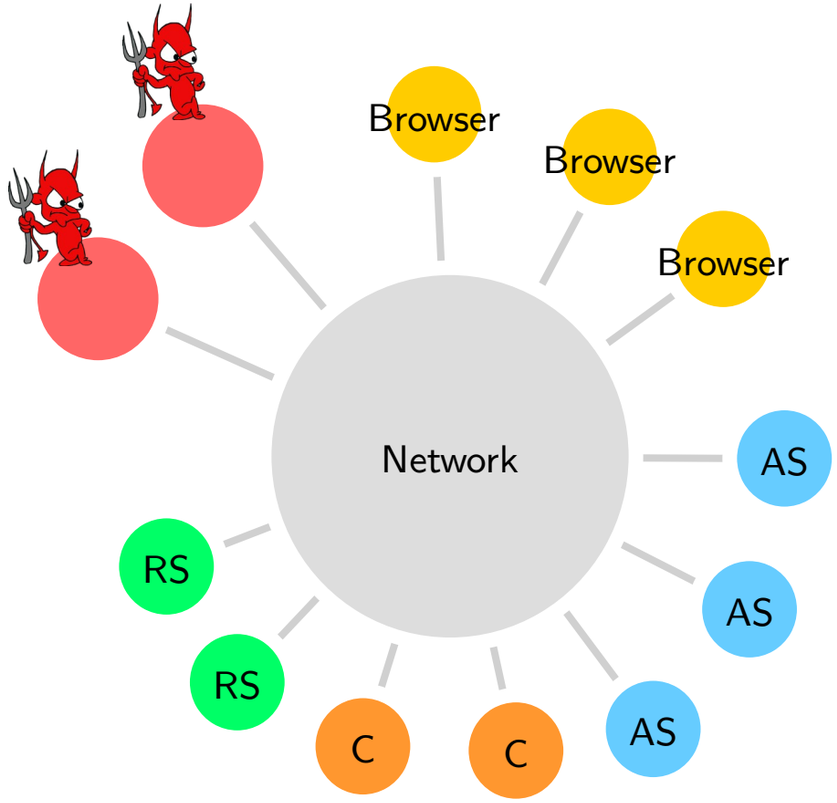
- FAPI 2.0 aims to fulfill the security goals in the presence of a strong attacker

  → "for arbitrary combinations of the following attackers"

- **A1, A1a, A2: Web and classical network attackers**
  - We aim to prove the security properties for network attackers

    → network attacker subsumes the web attackers,

    i.e.: if the properties are true for network attackers, they are also true for any web attackers



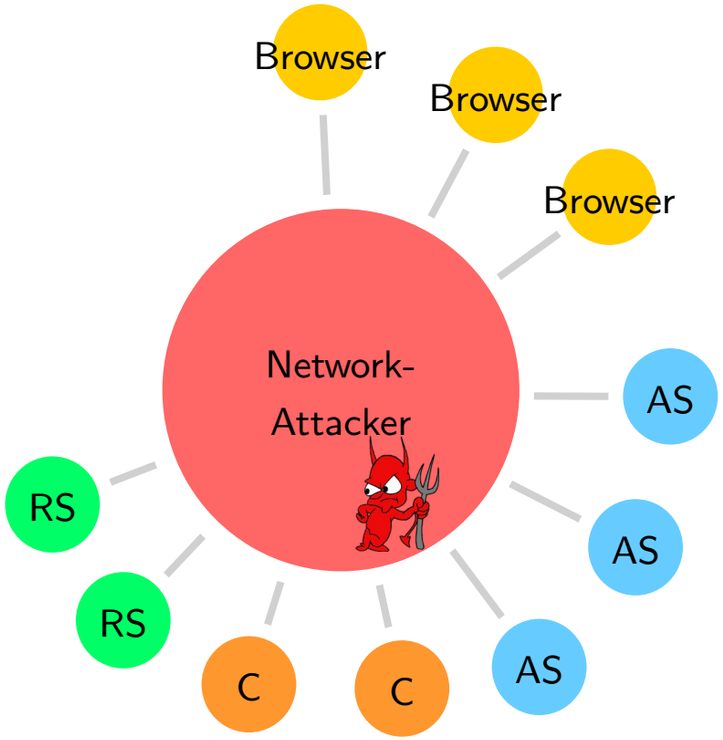
FAPI 2.0 Attacker Model: https://bitbucket.org/openid/fapi/src/209f58afbd41fb20ab3ed65ca4e2f67ffd5dda77/FAPI_2_0_Attacker_Model.md

FAPI 2.0 Attacker Model

- FAPI 2.0 aims to fulfill the security goals in the presence of a strong attacker

  → "for arbitrary combinations of the following attackers"

- **A1, A1a, A2: Web and classical network attackers**
  - We aim to prove the security properties
    for network attackers
    → network attacker subsumes the web attackers,
    i.e.: if the properties are true for network attackers,
    they are also true for any web attackers



FAPI 2.0 Attacker Model: https://bitbucket.org/openid/fapi/src/209f58afbd41fb20ab3ed65ca4e2f67ffd5dda77/FAPI_2_0_Attacker_Model.md

FAPI 2.0 Attacker Model

- **A3a Attacker: Authorization Request Leakage**
  - Assumption: Authorization request might leak to attacker

After receiving the PAR response, the client creates the authorization request.

```
21:  else if reference[responseTo] ≡ PAR then
22:      let requestUri := m.body[request_uri]
23:      let clientId := session[client_id]
24:      let request := s'.sessions[sessionId][startRequest]
25:      let authEndpoint := oauthConfigCache[issuer][auth_ep]
26:      let authEndpoint.parameters := [client_id: clientId, request_uri: requestUri]
27:      let headers := [Location: authEndpoint, ReferrerPolicy: origin]
28:      let headers[Set-Cookie] := [⟨__Host, sessionId⟩: ⟨sessionId, ⊤, ⊤, ⊤⟩]
29:      let response := enc_s(⟨HTTPResp, request[message].nonce, 303, headers, ⟨⟩⟩, request[key])
30:      let leak := ⟨LEAK, authEndpoint⟩    → We assume that the authorization request leaks to the attacker, see [5] and Section VI
31:      let leakAddress ← IPs
32:      stop ⟨⟨request[sender], request[receiver], response⟩, ⟨leakAddress, request[receiver], leak⟩⟩, s'
```

  - The client model always leaks the authorization request
  - Goal: prove that the security properties are true even in this case

FAPI 2.0 Attacker Model

- **A3b Attacker: Authorization Response Leakage**
  - Authorization server model: always leaks response (similar to authorization request)
- **A5 Attacker: Token Endpoint Configuration**

14: **let** $misconfiguredTEp \leftarrow \{\top, \bot\}$
15: **if** $misconfiguredTEp \equiv \bot$ **then**
16: **let** $tokenEndpoint := oauthConfig[\texttt{token\_ep}]$
17: **else** $\rightarrow$ Choose wrong token endpoint.
18: **let** $host \leftarrow$ Doms
19: **let** $path \leftarrow \mathbb{S}$
20: **let** $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
21: **let** $tokenEndpoint := \langle \texttt{URL}, \texttt{S}, host, path, parameters, \bot \rangle$

Goal: Prove properties for all possible choices

  - Client model: for each token request, decides to use the correct endpoint (from the AS metadata) or a non-deterministically chosen URL

FAPI 2.0 Attacker Model

- **A7 Attacker: Resource Request and Response Leakage**
  - Client model leaks resource request (including access token)
  - Response leakage: only for TLS protected message (ciphertext)
- **A8 Attacker: Resource Response Tampering**
  - Only for TLS protected message (ciphertext)

as discussed in first meeting

also as discussed in first meeting

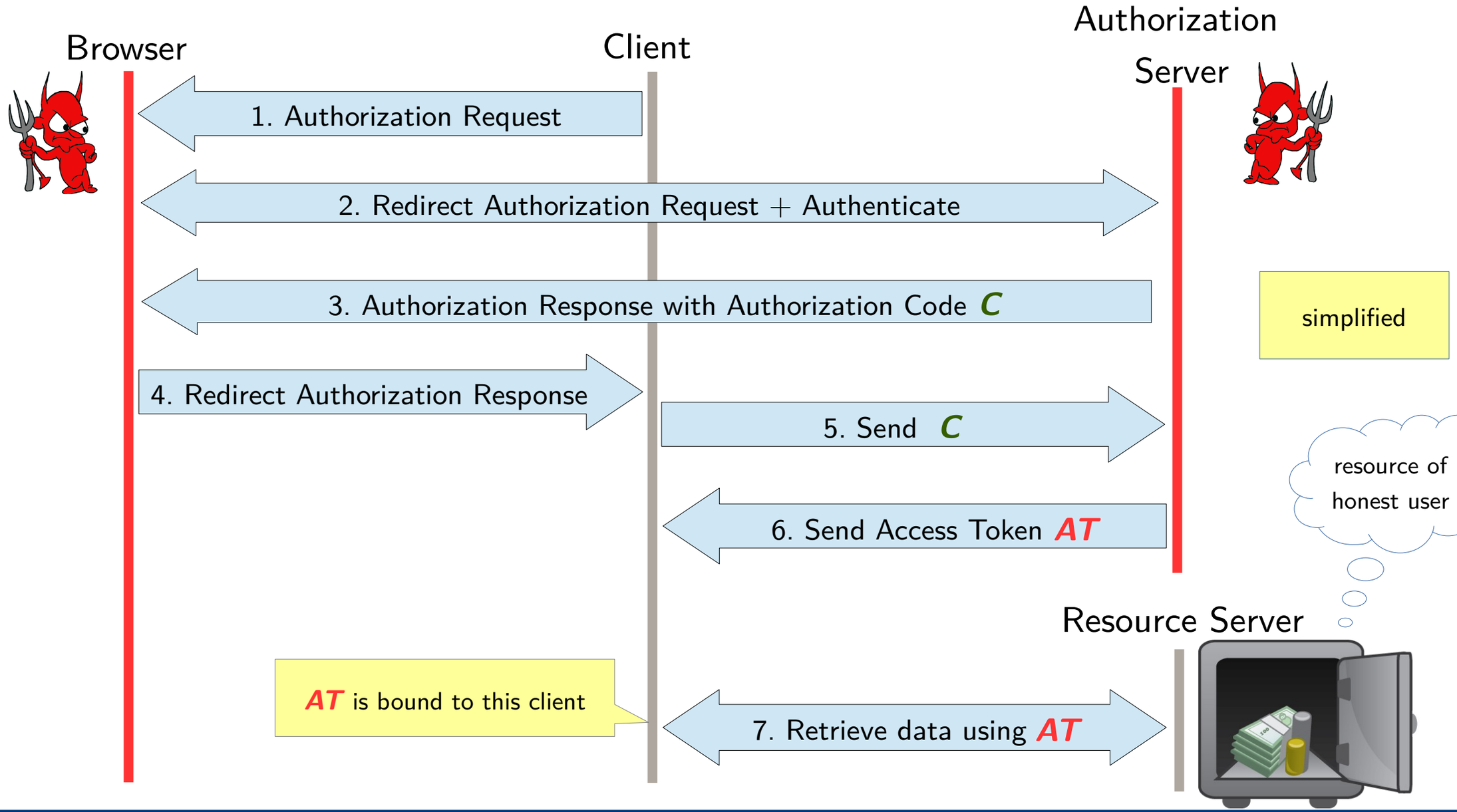
Remarks

Remarks

- Some open questions (see also Section III of the report)
- In particular: Attacks found during our FAPI 1.0 analysis still seem to be possible
  - Cuckoo's Token Attack
  - Access Token Injection Attack
  - Authorization Request Leak Attack

Cuckoo's Token Attack

Attacker model: ***AT*** leakage possible
(A7 – resource request leakage)



Authorization Request Leak Attacks

Attacker model: AuthZ request leakage possible (A3a attacker)

Browser

Client

Authorization Server

Auth. Request

2. Authorization Request

1. PAR (including *PKCE-CC*)

3. Redirect Authorization Request + Attacker Authenticates

(*C*, *PKCE-CC*)

4. Authorization Response with Authorization Code *C*

CSRF

5. Redirect Authorization Response

6. Send *C* (+ *PKCE-CV*)

PKCE: successful verification

7. Send Access Token *AT*

Associated with attacker

[draft-ietf-oauth-security-topics-19]: "Clients that have ensured that the authorization server supports PKCE [RFC7636] MAY rely the CSRF protection provided by PKCE."

Resource Server

8. Retrieve data using *AT*

Discussion

Discussion

- A7 attacker contradicts authorization goal ✅
- A8 attacker contradicts session integrity (for authorization) goal ✅
- Attacks discovered during previous FAPI 1.0 analysis