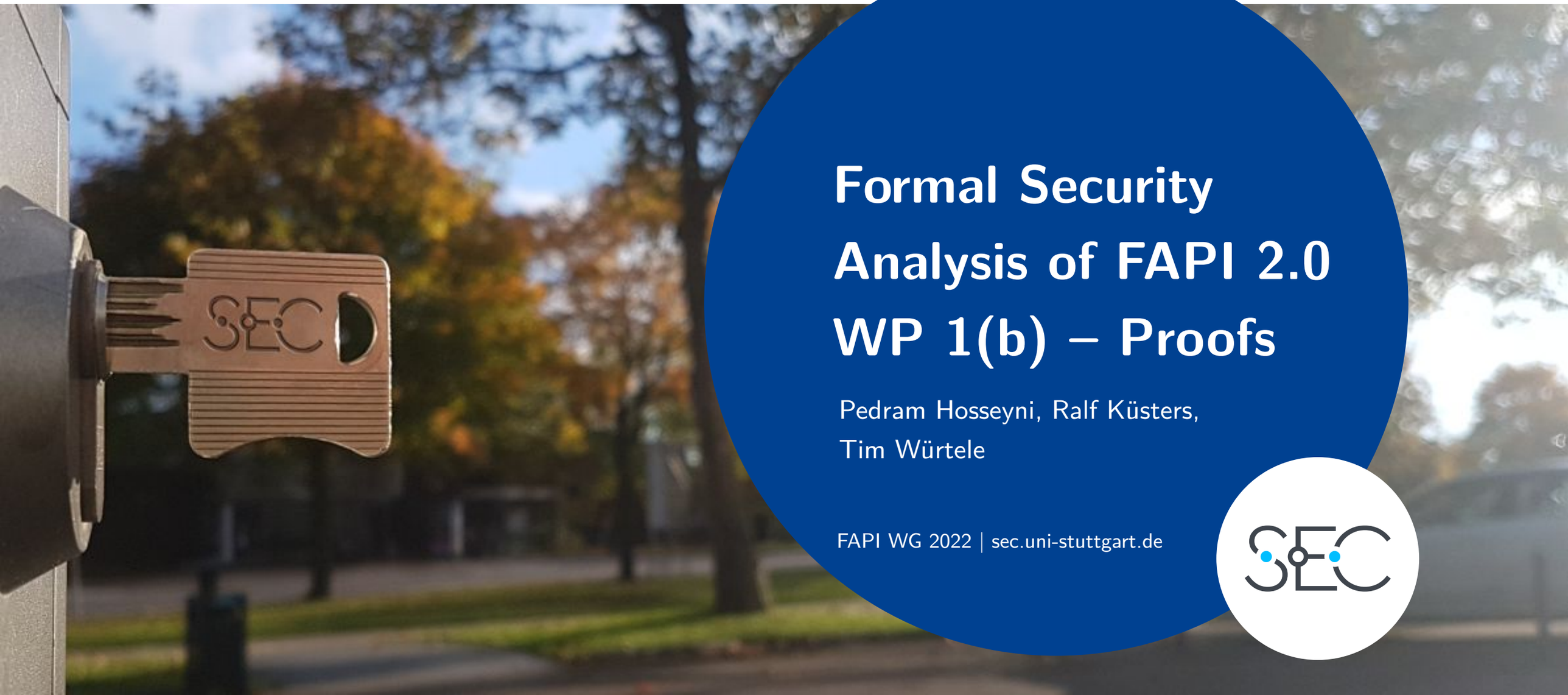
University of Stuttgart
Institute of
Information Security



Formal Security Analysis of FAPI 2.0 WP 1(b) – Proofs

Pedram Hosseyni, Ralf Küsters,
Tim Würtele

FAPI WG 2022 | sec.uni-stuttgart.de

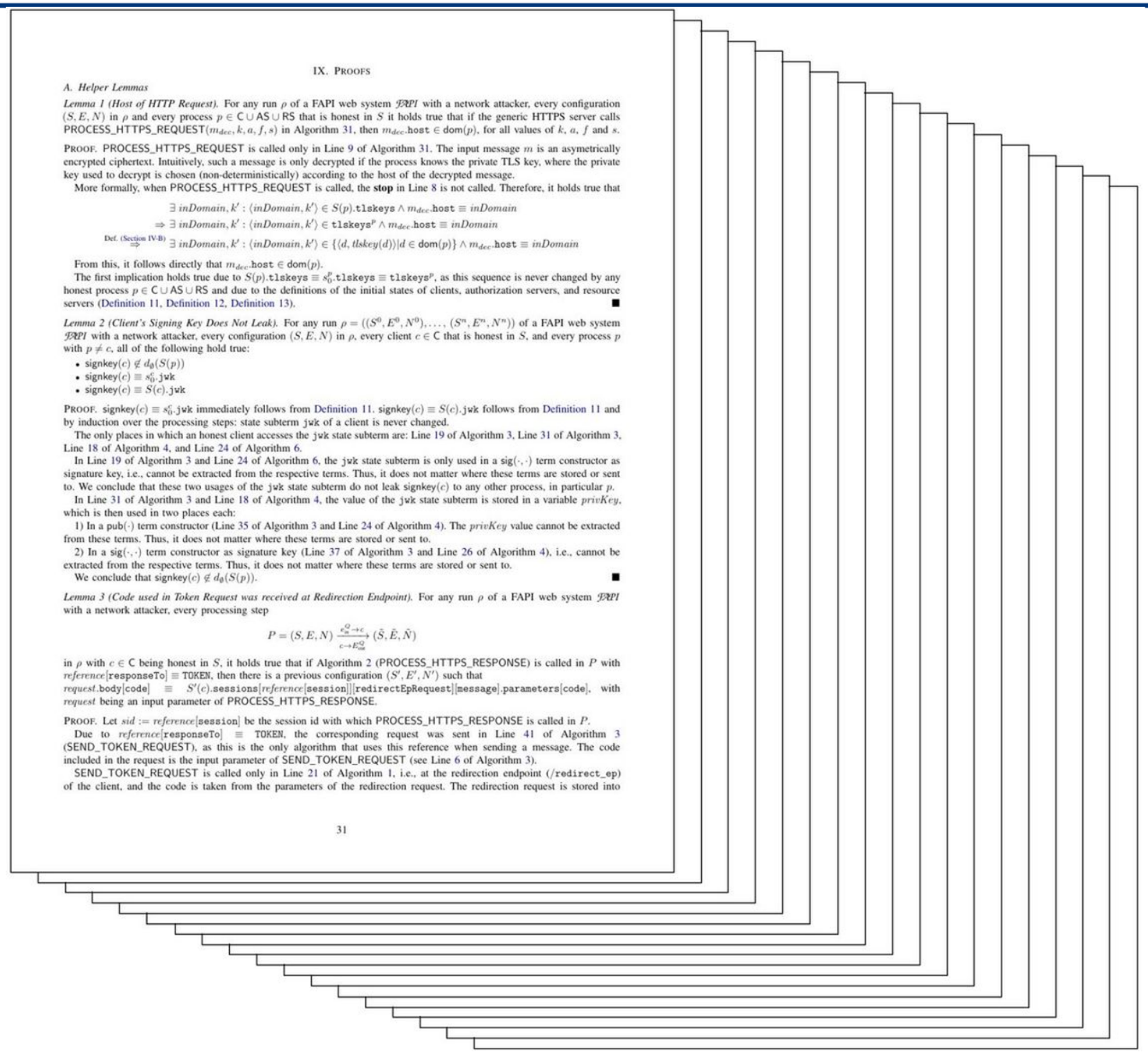
Outline

- Overview
- Summary of Results
- Security Properties
- Attacker Model: Changes
- Model: Differences to Specifications
- Conclusion

Overview

- Modeled FAPI 2.0 (based on the WIM)
- Formalized Security Goals from FAPI 2.0 Attacker Model
- (Tried to) Prove Formalized Security Properties
  - Under Attacker Assumptions from FAPI 2.0 Attacker Model

Proofs



IX. Proofs

A. Helper Lemmas

Lemma 1 (Host of HTTP Request). For any run ρ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ and every process $p \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$ that is honest in S it holds true that if the generic HTTPS server calls PROCESS_HTTPS_REQUEST(m_{dec}, k, a, f, s) in Algorithm 31, then $m_{dec}.\texttt{host} \in \mathsf{dom}(p)$, for all values of k, a, f and s.

Proof. PROCESS_HTTPS_REQUEST is called only in Line 9 of Algorithm 31. The input message m is an asymetrically encrypted ciphertext. Intuitively, such a message is only decrypted if the process knows the private TLS key, where the private key used to decrypt is chosen (non-deterministically) according to the host of the decrypted message.

More formally, when PROCESS_HTTPS_REQUEST is called, the **stop** in Line 8 is not called. Therefore, it holds true that

$$\begin{aligned} &\exists\, inDomain, k' : \langle inDomain, k' \rangle \in S(p).\texttt{tlskeys} \wedge m_{dec}.\texttt{host} \equiv inDomain \\ \Rightarrow\ &\exists\, inDomain, k' : \langle inDomain, k' \rangle \in \texttt{tlskeys}^p \wedge m_{dec}.\texttt{host} \equiv inDomain \\ \overset{\text{Def. (Section IV-B)}}{\Rightarrow}\ &\exists\, inDomain, k' : \langle inDomain, k' \rangle \in \{\langle d, tlskey(d)\rangle | d \in \mathsf{dom}(p)\} \wedge m_{dec}.\texttt{host} \equiv inDomain \end{aligned}$$

From this, it follows directly that $m_{dec}.\texttt{host} \in \mathsf{dom}(p)$.

The first implication holds true due to $S(p).\texttt{tlskeys} \equiv s_0^p.\texttt{tlskeys} \equiv \texttt{tlskeys}^p$, as this sequence is never changed by any honest process $p \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$ and due to the definitions of the initial states of clients, authorization servers, and resource servers (Definition 11, Definition 12, Definition 13). ■

Lemma 2 (Client's Signing Key Does Not Leak). For any run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every client $c \in \mathsf{C}$ that is honest in S, and every process p with $p \neq c$, all of the following hold true:

- $\mathsf{signkey}(c) \notin d_\emptyset(S(p))$
- $\mathsf{signkey}(c) \equiv s_0^c.\texttt{jwk}$
- $\mathsf{signkey}(c) \equiv S(c).\texttt{jwk}$

Proof. $\mathsf{signkey}(c) \equiv s_0^c.\texttt{jwk}$ immediately follows from Definition 11. $\mathsf{signkey}(c) \equiv S(c).\texttt{jwk}$ follows from Definition 11 and by induction over the processing steps: state subterm jwk of a client is never changed.

The only places in which an honest client accesses the jwk state subterm are: Line 19 of Algorithm 3, Line 31 of Algorithm 3, Line 18 of Algorithm 4, and Line 24 of Algorithm 6.

In Line 19 of Algorithm 3 and Line 24 of Algorithm 6, the jwk state subterm is only used in a $\mathsf{sig}(\cdot,\cdot)$ term constructor as signature key, i.e., cannot be extracted from the respective terms. Thus, it does not matter where these terms are stored or sent to. We conclude that these two usages of the jwk state subterm do not leak $\mathsf{signkey}(c)$ to any other process, in particular p.

In Line 31 of Algorithm 3 and Line 18 of Algorithm 4, the value of the jwk state subterm is stored in a variable $privKey$, which is then used in two places each:

1) In a $\mathsf{pub}(\cdot)$ term constructor (Line 35 of Algorithm 3 and Line 24 of Algorithm 4). The $privKey$ value cannot be extracted from these terms. Thus, it does not matter where these terms are stored or sent to.

2) In a $\mathsf{sig}(\cdot,\cdot)$ term constructor as signature key (Line 37 of Algorithm 3 and Line 26 of Algorithm 4), i.e., cannot be extracted from the respective terms. Thus, it does not matter where these terms are stored or sent to.

We conclude that $\mathsf{signkey}(c) \notin d_\emptyset(S(p))$. ■

Lemma 3 (Code used in Token Request was received at Redirection Endpoint). For any run ρ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every processing step

$$P = (S, E, N) \xrightarrow[c \to E_{out}^Q]{s_{in}^Q \to c} (\hat{S}, \hat{E}, \hat{N})$$

in ρ with $c \in \mathsf{C}$ being honest in S, it holds true that if Algorithm 2 (PROCESS_HTTPS_RESPONSE) is called in P with $reference[\texttt{responseTo}] \equiv \texttt{TOKEN}$, then there is a previous configuration (S', E', N') such that $request.\texttt{body}[\texttt{code}] \equiv S'(c).\texttt{sessions}[reference[\texttt{session}]][\texttt{redirectEpRequest}][\texttt{message}].\texttt{parameters}[\texttt{code}]$, with $request$ being an input parameter of PROCESS_HTTPS_RESPONSE.

Proof. Let $sid := reference[\texttt{session}]$ be the session id with which PROCESS_HTTPS_RESPONSE is called in P.

Due to $reference[\texttt{responseTo}] \equiv \texttt{TOKEN}$, the corresponding request was sent in Line 41 of Algorithm 3 (SEND_TOKEN_REQUEST), as this is the only algorithm that uses this reference when sending a message. The code included in the request is the input parameter of SEND_TOKEN_REQUEST (see Line 6 of Algorithm 3).

SEND_TOKEN_REQUEST is called only in Line 21 of Algorithm 1, i.e., at the redirection endpoint (/redirect_ep) of the client, and the code is taken from the parameters of the redirection request. The redirection request is stored into

31

- Detailed, systematic proofs of formalized security properties
- Based on formal model of FAPI 2.0

Summary of Results

- Some of the original security goals are not met under original attacker assumptions
- Adapted goals and assumptions as agreed on with FAPI WG + minor change to protocol
- Found no violations of adapted goals under adapted assumptions

Is FAPI 2.0 Security Profile secure?

→ Within the limits of our model, FAPI 2.0 meets the formalized security properties.

Recap: Security Properties

Security Properties

- Authorization – "[...] no attacker can access protected resources other than his own."
- Authentication – "[...] no attacker is able to log in at a client under the identity of another user."
- Session Integrity
  - for Authorization – "[...] no attacker is able to force a user to use resources of the attacker."
  - for Authentication – "[...] no attacker is able to force a user to be logged in under the identity of the attacker."
  - Analysis: Only if authorization request does not leak[1] (i.e., w/o A3a Attacker)

[1] https://bitbucket.org/openid/fapi/issues/534

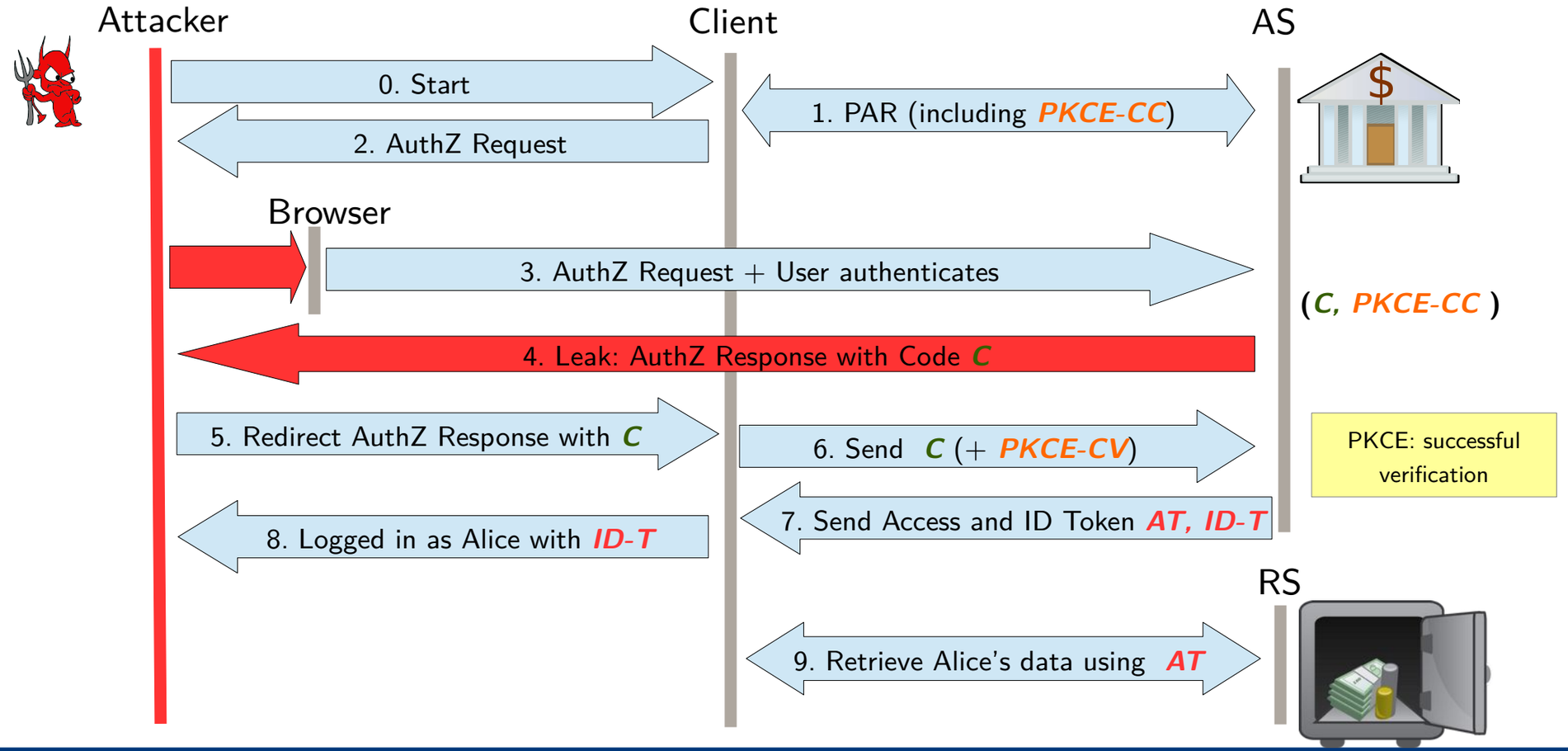
Attacker Model: Main Changes

A3b Attacker: Authorization Response Leakage

- Breaks Authorization and Authentication

attacker can access resources of honest user and log in at client as honest user

- Discussed with WG during FAPI Atlantic call on September 28th



A3b Attacker: Authorization Response Leakage

- Idea: Invalidate code at client
  - based on feedback, this cannot be implemented in realistic scenarios
- No other practical solution
- Resolution: remove the A3b Attacker[2]

> None of the existing mechanisms protect against such attacks. Implementers should ensure that Authorization Responses (in particular, the Authorization Code) cannot leak wherever possible, e.g., in the case of native apps.

[2] https://bitbucket.org/openid/fapi/pull-requests/377

A5 Attacker: Token Endpoint Misconfiguration

- A5 Attacker:[3]

  This attacker makes the client use a token endpoint that is not the one of the honest AS.

- A5 Attacker is effectively removed by requiring Server Metadata:[3]

> Important: This attacker is a model for misconfigured token endpoint URLs that were considered in FAPI 1.0. **Since the FAPI 2.0 Security Profile mandates that the token endpoint address is obtained from an authoritative source and via a protected channel, i.e., through OAuth Metadata obtained from the honest AS, this attacker is not relevant in FAPI 2.0.** The description here is kept for informative purposes only.

[3] https://bitbucket.org/openid/fapi/src/master/FAPI_2_0_Attacker_Model.md

Model: Differences to Specifications

Preventing Cuckoo's Token Attack[4]

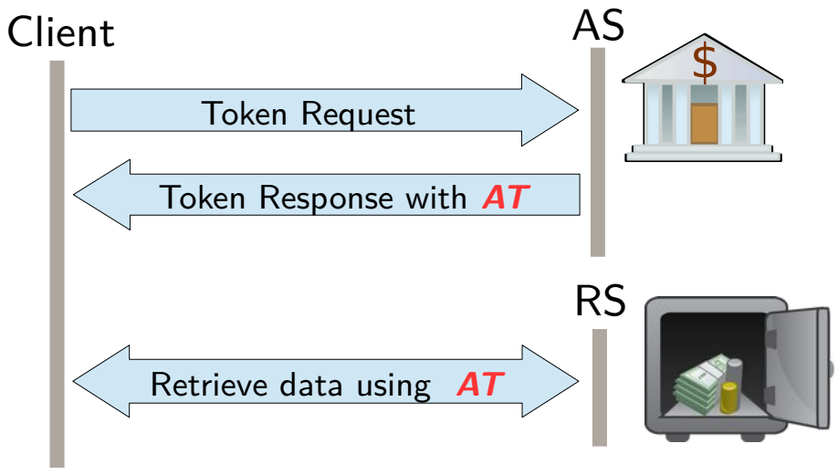
- Recap:
  - Sender-constrained access token leaks to attacker
  - Malicious AS replays access token to correct client
- Recommendations in the Security profile:

> The pre-conditions for this attack do not apply to many ecosystems and require a powerful attacker. In situations where the pre-conditions may be met, the possible mitigations include:
>
> 1. Clients using different DPoP keys or MTLS certificates at each authorization server
>
> 2. Clients sending the issuer identifier the access token was obtained from to the resource server, and requiring resource servers to verify the issuer matches the authorization server that originally issued the token (though there is no standardized method for clients to send the issuer to the resource server)
>
> 3. Reducing the time window for the attack by using short lived access tokens alongside refresh tokens

[4] https://bitbucket.org/openid/fapi/issues/525/decide-on-what-to-do-for-a-cuckoo-s-token

Preventing Cuckoo's Token Attack[4]

- None of the recommendations are required
  → in the analysis, each mitigation might "mask" some unknown attack
- Issue 525:[4] "`For the analysis, we need to find out what the least 'invasive' way is to avoid running into this attack`"
- Formal model: Assume that client checks whether the requested resource is managed by the AS that sent the AT



[4] https://bitbucket.org/openid/fapi/issues/525/decide-on-what-to-do-for-a-cuckoo-s-token

Preventing DPoP Proof Replay

- A7 Attacker: Leakage of resource request
- Replay of DPoP proofs possible → violates Authorization property
- Current specification: no effective mitigation
- Formal model:
  - Assume that there is an effective replay protection using DPoP nonces
  - Assume that RS invalidates the nonce (hence, also the DPoP proof)
  - Assume that resource request leaks after the request was processed by the RS[5]

Not mandated by DPoP specification

[5] https://bitbucket.org/openid/fapi/pull-requests/381/change-attacker-model-to-reflect-formal

Further Assumptions (Out of Scope of FAPI 2.0)

- CSRF Protection at Client
  - Interaction for initiating the protocol is out-of-scope
  - Formal model: done using a dedicated endpoint that start the flow upon receiving a request
  - Assumption: This endpoint is protected against CSRF

Otherwise, an attacker could initiate the flow in the background without the user noticing how the flow started

- Session Cookie between Browser and Client
  - In the formal model: client sets cookie at browser to establish a session
  - Assumption: Protection against session hijacking

Remark: ID Token Validation

- As agreed on: modelled client that does not check ID token signature, but relies on TLS connection information[6]
- Result: ID token signature verification was not necessary to prove security properties (within the model)

[6] https://bitbucket.org/openid/fapi/issues/522/optional-id-token-signature-validation-for

Conclusion

Conclusion

- Some of the original security goals are not met under original attacker assumptions
- Adapted goals and assumptions as agreed on with FAPI WG + minor change to protocol
- Found no violations of adapted goals under adapted assumptions

Is FAPI 2.0 Security Profile secure? 

→ Within the limits of our model, FAPI 2.0 meets the formalized security properties.