Formal Security Analysis of the OpenID Financial-grade API

Daniel Fett, Pedram Hosseyni, Ralf Küsters

To appear at IEEE Security and Privacy 2019

2019-03-20
OpenID Financial-grade API
- Profile of OAuth 2.0 Authorization Framework

“Financial-grade”:
- “highly secured OAuth profile”
- “to be used in write access to financial data […] and other similar higher risk access”
- “higher risk use cases”

Such situations are extremely interesting for attackers ...
Some Recent Attacks ...

Article 1:

Forbes: 
Laughing All The Way To The Bank: Cybercriminals Targeting U.S. Financial Institutions

Article 2:

Financial Times:
North Korea Hackers Tried to Take $1.1 Billion in Bank Attacks

Article 3:

Business:
Mexico central bank says hackers siphoned $15 million from five companies
Objectives of our Work

► Create a Model of the Financial-grade API
  - Including: PKCE, mTLS, OAUTH, ...
► Capturing Security Goals and Assumptions
► Proof of Security Properties
Outline

• Model-based Approach
• Financial-grade API
  - Key Mechanisms
  - Attacker Model
• Security Properties
• Attacks Found through the Analysis
Outline

• Model-based Approach

• Financial-grade API
  - Key Mechanisms
  - Attacker Model

• Security Properties

• Attacks Found through the Analysis
Previously Analyzed Protocols

**WIM**

- Mozilla BrowserID
  - Discovered severe attacks against authentication
  - After fixes: Proof of authentication
  - Special feature privacy: broken beyond repair
  - [S&P2014] [ESORICS2015]

- SPRESSO
  - Designed from scratch
  - First formalized in **WIM**, then implemented
  - First SSO with proven privacy and security

- OAuth 2.0
  - Found several new attacks
  - Developed fixes and implementation guidelines
  - Proof of security

- OpenID Connect
  - Including extensions
  - Developed best practices against known attacks
  - Proof of security

- [CCS2015]
- [CCS2016]
- [CSF2017]
Our Model-Based Approach

- **Foundation:** Formal description of the web
  - Generic web infrastructure model
  - Application model built from source code or specification
  - Application-specific model
  - Security properties
  - Formal Proofs of Properties
  - Attacks
  - Fixes

Rinse and repeat until proof goes through.
Advantages

This approach can yield...

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

- Model-based Approach
- Financial-grade API
  - Key Mechanisms
  - Attacker Model
- Security Properties
- Attacks Found through the Analysis
OpenID Financial-grade API (FAPI)

► Profile of the OAuth 2.0 Authorization Framework
► Utilizes mechanisms of OpenID Connect
► Different Profiles
  - Read-Only Profile
    • Authorization Code Flow
  - Read-Write Profile
    • OIDC Hybrid Flow
    • Authorization Code Flow with JARM
OAuth 2.0 Authorization Code Mode

1. Authorization Request
2. Redirect Authorization Request + Authenticate
4. Redirect Authorization Response
5. send C
6. send Access Token AT
7. retrieve data using AT
Attacker Model (Read-Only Profile)

1. Authorization Request
2. Redirect Authorization Request + Authenticate
4. Redirect Authorization Response

App Client

Browser

Client

Authorization Server

Resource Server

5. send $C$
6. send Access Token $AT$
7. retrieve data using $AT$
Attacker Model (Read-Write Profile)

1. Authorization Request

2. Redirect Authorization Request + Authenticate


4. Redirect Authorization Response

5. send $C$

6. send Access Token $AT$

7. retrieve data using $AT$

Leakage

Browser

Client

Authorization Server

Resource Server

App Client

misconfigured Token Endpoint
FAPI: Key Mechanisms
New Defense Mechanisms

- Token Binding
  - Proof Key for Code Exchange (PKCE)
  - Signed Authorization Response (JARM)
- Improved Client Authentication
- Signed Authorization Request
Token Binding

- Two Methods:
  - OAuth 2.0 Token Binding
  - Mutual TLS

- Goal: Bind Authorization Code and Access Token to Client
Binding Access Tokens: Idea

1. Authorization Request

2. Redirect Authorization Request + Authenticate

3. Authorization Response with Authorization Code \( C \)

4. Redirect Authorization Response

5. send \( C \)

6. send Access Token \( AT \)

7. retrieve data using \( AT \)

Proof of Possession

Accept only after successful PoP

Remember Public Key
FAPI: Attacker Model
Attacker Model

► Read-Only Profile:
  - Authorization Response leaks
  - Authorization Request leaks

► Read-Write Profile
  - Token Endpoint controlled by Attacker
  - Access Tokens leaks

As of 23-10-2018, (including JARM)
Attacker Model: More Details

► Read-Only Profile:
  - Authorization Response leaks
  - Authorization Request leaks

► Read-Write Profile
  - Token Endpoint controlled by Attacker
  - Access Tokens leaks

Part 1: Read-Only API Security Profile
5.2.2 Authorization server:
7. shall require RFC7636 with S256 as the code challenge method;
Attacker Model: More Details

- **Read-Only Profile:**
  - Authorization Response leaks
  - Authorization Request leaks

- **Read-Write Profile**
  - Token Endpoint controlled by Attacker
  - Access Tokens leaks

Figure: https://tools.ietf.org/html/rfc7636
Attacker Model: More Details

► Read-Only Profile:
  - Authorization Response leaks
  - Authorization Request leaks

► Read-Write Profile
  - Token Endpoint controlled by Attacker
  - Access Tokens leaks

Part 1: Read-Only API Security Profile
5.2.2 Authorization server:
7. shall require RFC7636 with S256 as the code challenge method;
Attacker Model: More Details

► Read-Only Profile:
  - Authorization Response leaks
  - Authorization Request leaks

► Read-Write Profile
  - Token Endpoint controlled by Attacker
  - Access Tokens leaks

4b. A more sophisticated attack scenario allows the attacker to observe requests (in addition to responses) to the authorization endpoint. [...] This was caused by leaking http log information in the OS. To mitigate this, "code_challenge_method" value must be set either to "S256" or a value defined by a cryptographically secure "code_challenge_method" extension.
Attacker Model: More Details

► Read-Only Profile:
  - Authorization Response leaks
  - Authorization Request leaks

► Read-Write Profile
  - Token Endpoint controlled by Attacker
  - Access Tokens leaks

8.3.2 Client credential and authorization code phishing at token endpoint

In this attack, the client developer is social engineered into believing that the token endpoint has changed to the URL that is controlled by the attacker.

As the result, the client sends the code and the client secret to the attacker, which will be replayed subsequently.

When the FAPI client uses MTLS or OAUTB, the authorization code is bound to the TLS channel, any phished client credentials and authorization codes submitted to the token endpoint cannot be used since the authorization code is bound to a particular TLS channel.
Attacker Model: More Details

► Read-Only Profile:
  - Authorization Response leaks
  - Authorization Request leaks

► Read-Write Profile
  - Token Endpoint controlled by Attacker
  - Access Tokens leaks

8.3.5 Access token phishing

When the FAPI client uses MTLS or OAUTB, the access token is bound to the TLS channel, it is access token phishing resistant as the phished access tokens cannot be used.
Security Definitions
FAPI: Security Definitions

► **Authentication**

Attacker cannot log in at client with honest identity

► **Authorization**

Attacker cannot access resources of honest identity

► **Session Integrity**

Honest user is logged in under their own account and using their own resources
Definition 17 (Authorization Property). We say that the FAPI web system with a network attacker $\mathcal{FAPI}^n$ is secure w.r.t. authorization iff for every run $\rho$ of $\mathcal{FAPI}^n$, every configuration $(S, E, N)$ in $\rho$, every authorization server $as \in AS$ that is honest in $S$ with $s^{as}_0.resource\_servers$ being domains of honest resource servers, every identity $id \in ID^{as}$ with $b = ownerOfID(id)$ being an honest browser in $S$, every client $c \in C$ that is honest in $S$ with client id $client\_Id$ issued to $c$ by $as$, every resource server $rs \in RS$ that is honest in $S$ such that $id \in s^{rs}_0.ids$, $s^{rs}_0.authServ \in dom(as)$ and with $dom_{rs} \in s^{as}_0.resource\_servers$ (with $dom_{rs} \in dom(rs)$), every access token $t$ associated with $c$, $as$ and $id$ and every resource access nonce $r \in s^{rs}_0.rNonce[id] \cup s^{rs}_0.wNonce[id]$ it holds true that:

If $r$ is contained in a response to a request $m$ sent to $rs$ with $t \equiv m.header[Authorization]$, then $r$ is not derivable from the attackers knowledge in $S$ (i.e., $r \not\in d_0(S(\text{attacker})))$. 
FAPI: Security Definitions

► Authentication
Attacker cannot log in at client with honest identity

► Authorization
Attacker cannot access resources of honest identity

► Session Integrity
Honest user is logged in under their own account and using their own resources
Attacks
Attacks Found Through Our Formal Analysis

► Cuckoo’s Token Attack
► Access Token Injection
► PKCE Chosen Challenge Attack
► Authorization Request Leak Attacks
Attacks Found Through Our Formal Analysis

► Cuckoo’s Token Attack
► Access Token Injection
► PKCE Chosen Challenge Attack
► Authorization Request Leak Attacks
Recap: Binding Access Tokens

1. Authorization Request

2. Redirect Authorization Request + Authenticate


4. Redirect Authorization Response

5. send $C$

6. send Access Token $AT$

7. retrieve data using $AT$

Remember Public Key

Accept only after successful PoP

Proof of Possession
Cuckoo’s Token Attack

1. Authorization Request
2. Redirect Authorization Request + Authenticate
4. Redirect Authorization Response
5. send $C$
6. send Access Token $AT$
7. retrieve data using $AT$

$AT$ is bound to Client

Read-Write Profile
Mitigation

1. Authorization Request

2. Redirect Authorization Request + Authenticate


4. Redirect Authorization Response

5. send $C$

6. send Access Token $AT$

7. retrieve data using $AT$ Include expected issuer of $AT$

Wrong AS → Stop

Read-Write Profile
Attacks Found Through Our Formal Analysis

- Cuckoo’s Token Attack
- Access Token Injection
- PKCE Chosen Challenge Attack
- Authorization Request Leak Attacks
Recap: Attacker Model

► Read-Only Profile:
  - Authorization Response leaks
  - Authorization Request leaks

► Read-Write Profile
  - Token Endpoint controlled by Attacker
  - Access Tokens leaks
Access Token Injection

1. Authorization Request

2. Redirect Authorization Request + Authenticate


4. Redirect Authorization Response

5. send $C$

6. send Access Token $AT$

7. retrieve data using $AT$

$AT_{\text{Read-Write Profile}}$ is bound to Client

Browser

Client

Authorization Server

Resource Server
Mitigation

1. Authorization Request

2. Redirect Authorization Request + Authenticate


4. Redirect Authorization Response

5. send $C$

6. send Access Token $AT$

Add at_hash to JARM-Response

Resource Server

Error: Wrong hash → Stop

Read-Write Profile

Client

Authorization Server

Browser

- Read-Write Profile
- Add at_hash to JARM-Response
- Or: add at_hash to second ID Token

Profile

Daniel Fett, Pedram Hosseyni, Ralf Küsters
Attacks Found Through Our Formal Analysis

- Cuckoo’s Token Attack
- Access Token Injection
- PKCE Chosen Challenge Attack
- Authorization Request Leak Attacks
Fixes and Security Proof

- Fixes proposed for all attacks

- Proved security
  - Authentication ✔
    - Attacker cannot log in at client with honest identity
  - Authorization ✔
    - Attacker cannot access resources of honest identity
  - Session Integrity ✔
    - Honest user is logged in under their own account and using their own resources

Only for Webserver
Clients using OAUTH
Conclusion

► First formal security analysis of the OpenID Financial-grade API
► Found several attack scenarios
► Suggested fixes
► Proved security under strong attacker model

Thanks!