UNIVERSITÄT SIEGEN

Security of Online Social Networks

Authentication II

Lehrstuhl IT-Sicherheitsmanagment

Universität Siegen

April 26, 2012

Overview Lesson 03

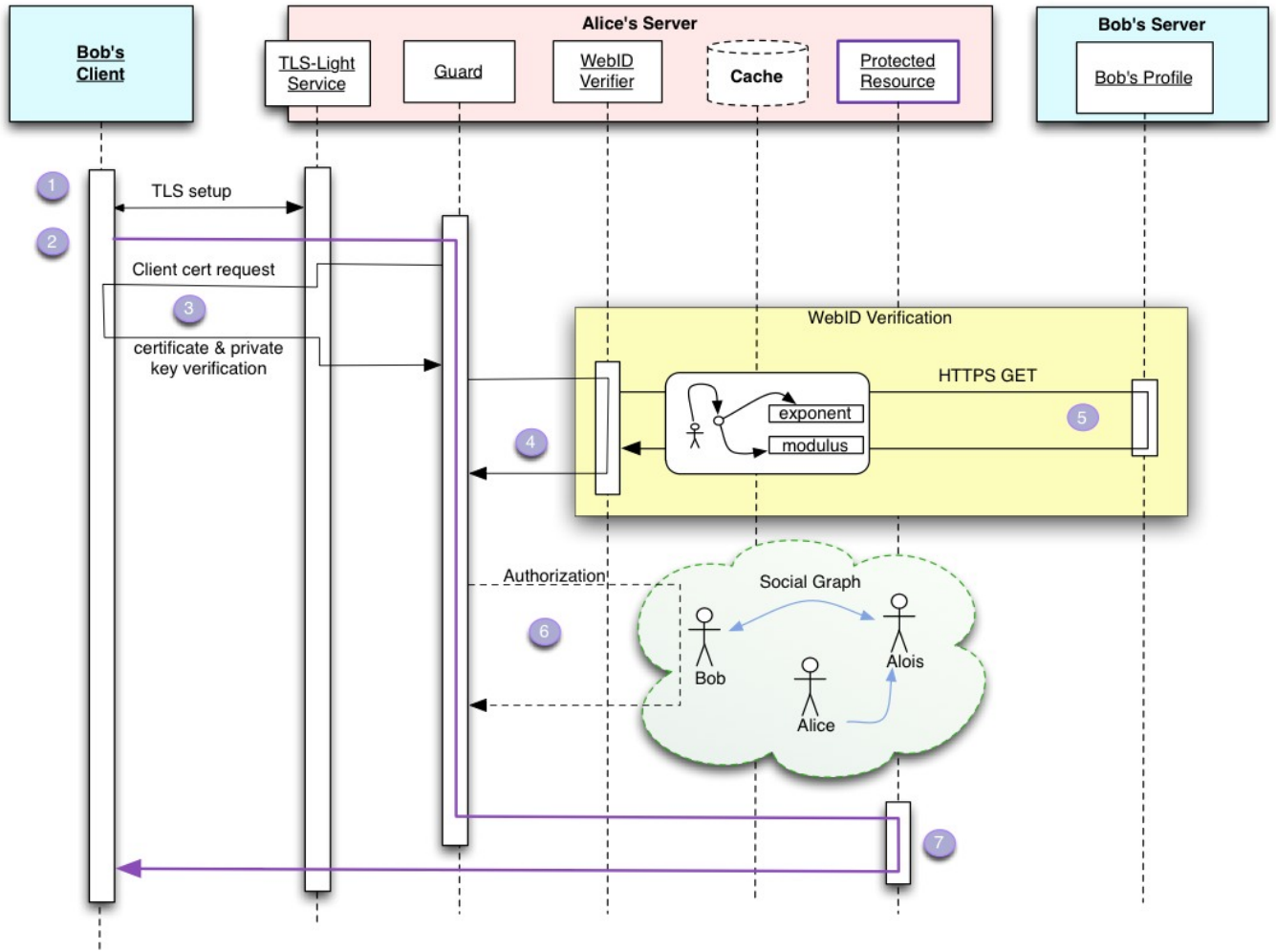
WebID

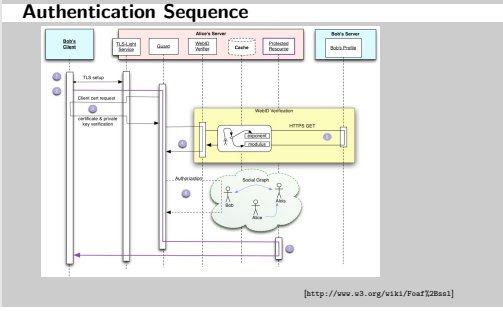
OpenID

OAuth

WebID II

Authentication Sequence

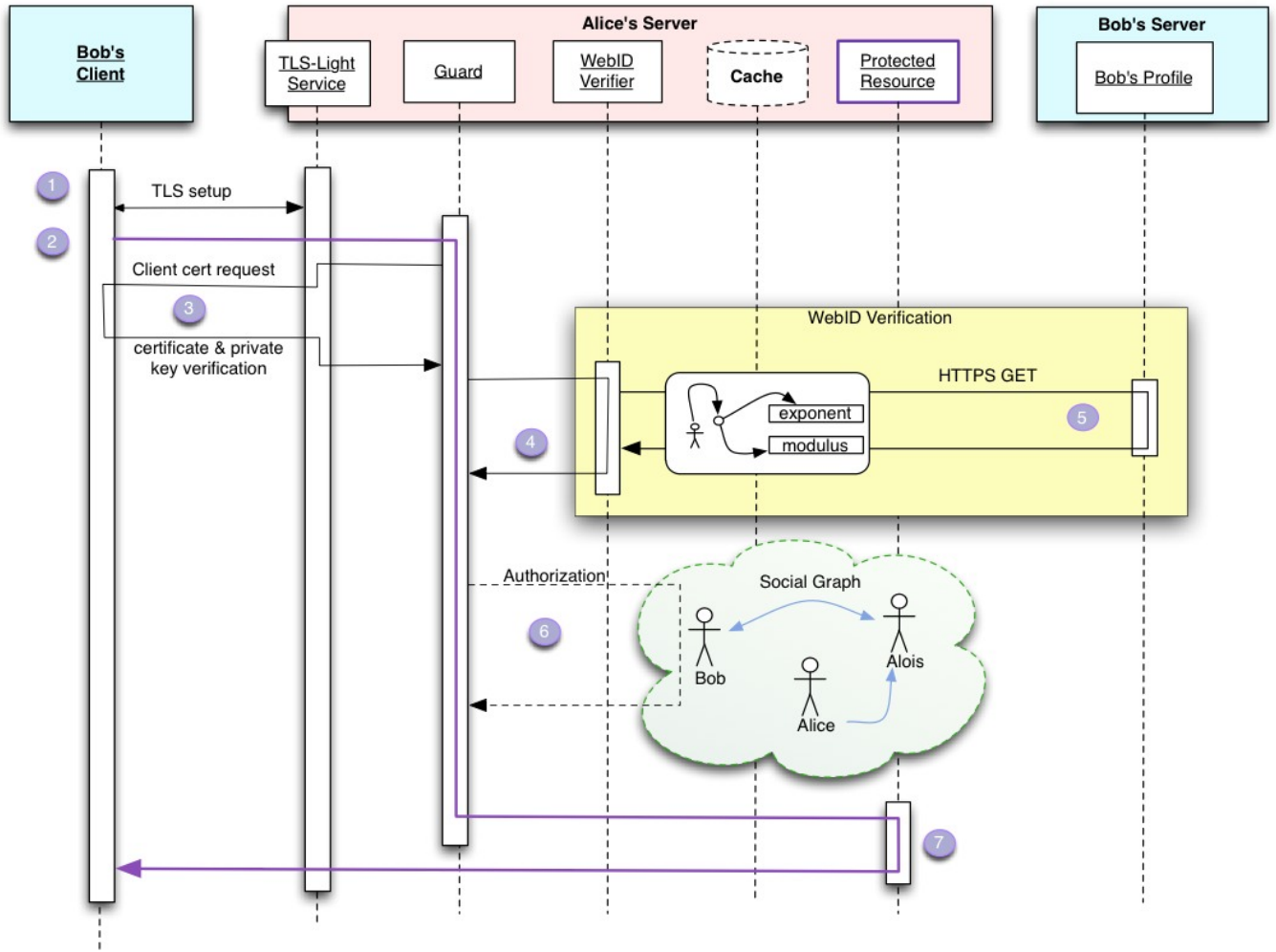


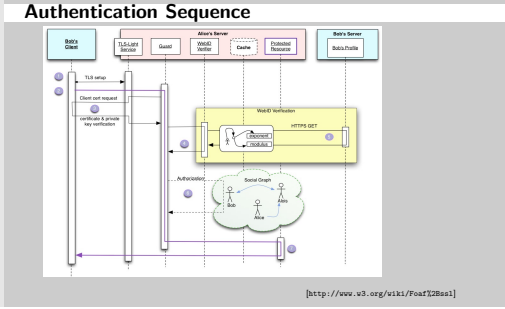


1. Bob's Client must open a TLS [RFC5246] connection with the server which authenticates itself using well known TLS mechanisms. This may be done as the first part of an HTTPS connection [HTTP-TLS].
2. Once the Transport Layer Security [TLS] has been set up, the application protocol exchange can start. If the protocol is HTTP then the client can request an HTTP GET, PUT, POST, DELETE, ... action on a resource as detailed by [HTTP11]. The Guard can then intercept that request and by checking some access control rules determine if the client needs authentication. We will consider the case here where the client does need to be authenticated.

3. The Guard must requests the client to authenticate itself using public key cryptography by signing a token with its private key and have the Client send its Certificate. This has been carefully defined in the TLS protocol and can be summarised by the following steps:
   - 3.1 The guard requests of the TLS agent that it make a Certificate Request to the client. The TLS layer does this. Because the WebID protocol does not rely on Certificate Authorities to verify the contents of the Certificate, the TLS Agent can ask for any Certificate from the Client. More details in Requesting the Client Certificate
   - 3.2 The Client asks Bob to choose a certificate if the choice has not been automated. We will assume that Bob does choose a WebID Certificate and sends it to the client.
   - 3.3 The TLS Agent must verify that the client is indeed in possession of the private key. What is important here is that the TLS Agent need not know the Issuer of the Certificate, or need not have any trust relation with the Issuer. Indeed if the TLS Layer could verify the signature of the Issuer and trusted the statements it signed, then step 4 and 5 would not be needed - other than perhaps as a way to verify that the key was still valid.
   - 3.4 The WebID Certificate is then passed on to the Guard with the proviso that the WebIDs still needs to be verified.

Authentication Sequence

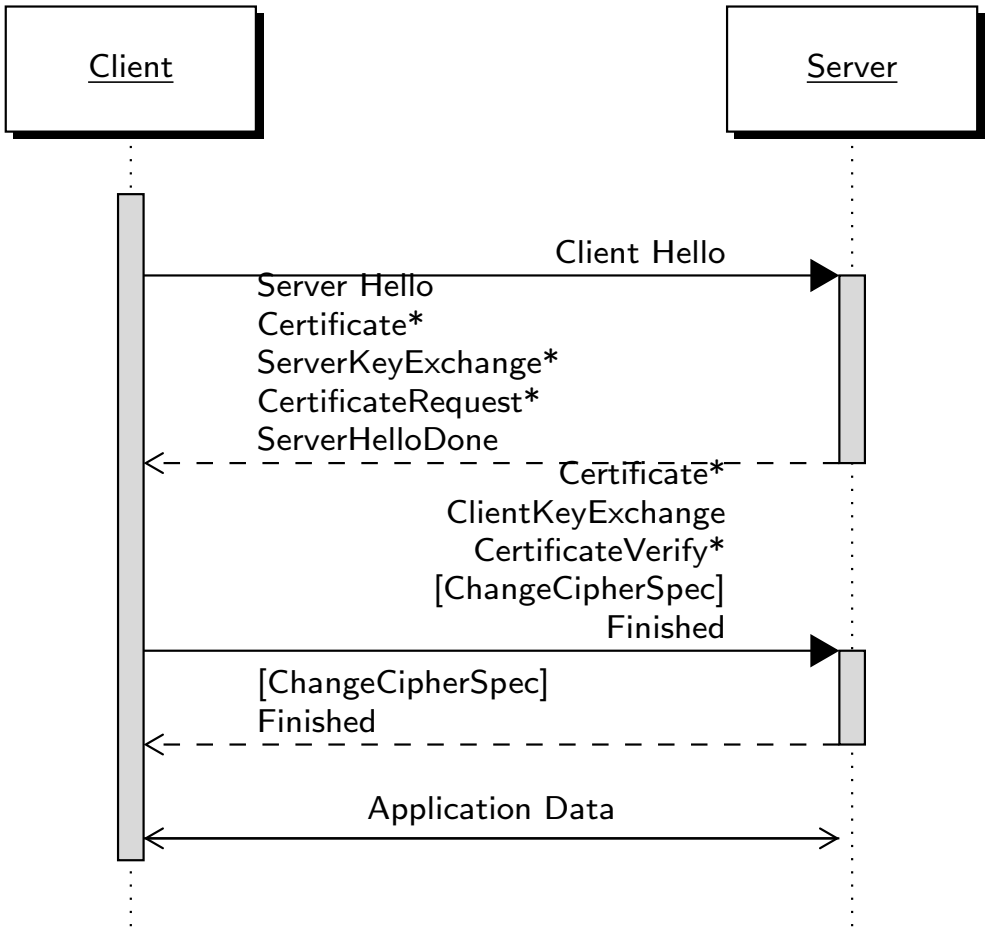


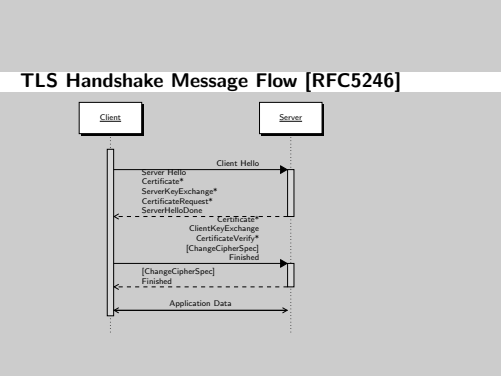


4. The Guard then must ask the Verification Agent to verify that the WebIDs do identify the agent who knows the given public key.

5. The WebID is verified by looking up the definition of the URL at its canonical location. This can be done by dereferencing it. The Verification Agent must extract the public key and all the URI entries contained in the Subject Alternative Name extension of the WebID Certificate. A WebID Certificate may contain multiple URI entries which are considered claimed WebIDs at this point, since they have not been verified. The Verification Agent may verify as many or as few WebIDs it has time for. It may do it in parallel and asynchronously. However that is done, a claimed WebIDs can only be considered verified if the following steps have been accomplished successfully:
   - 5.1 If the WebID Verifier does not have an up to date version of the WebID profile in the cache, then it must dereference the WebID using the canonical method for dereferencing a URL of that scheme. For an https://... WebID this would be done using the [HTTP-TLS] protocol.
   - 5.2 The returned representation is then transformed into an RDF graph as specified in Processing the WebID Profile
   - 5.3 That graph is then queried as explained in Querying the Graph. If the query succeeds, then that WebID is verified.
6. With the set of verified WebIDs the Guard can then check its access control rules using information from the web and other information available to it, to verify if the referent of the WebID is indeed allowed access to the protected resource. The exact nature of those Access Control Rules is left for another specification. Suffice it to say that it can be something as simple as a lookup in a table.
7. If access is granted, then the guard can pass on the request to the protected resource, which can then interact unimpeded with the client.

TLS Handshake Message Flow [RFC5246]





1. Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.
2. Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
3. Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
4. Generate a master secret from the premaster secret and exchanged random values.
5. Provide security parameters to the record layer.
6. Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

[1]

TLS Certificate Request

- CertificateRequest
- immediately after ServerKeyExchange
- WebID: empty `certificate_authorities`

```
struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    SignatureAndHashAlgorithm
      supported_signature_algorithms<2^16-1>;
    DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

TLS Client Verify

- immediately after Client Certificate
- must have signing capabilities

```
struct {
  digitally-signed struct {
     opaque handshake_messages[handshake_messages_length];
  }
} CertificateVerify;
```

WebID other Authentication

If the Client does not send a certificate, because either it does not have one or it does not wish to send one, other authentication procedures can be pursued at the application layer with protocols such as OpenID, OAuth, BrowserID, etc...

Further Links

RDF

- Concepts `http://www.w3.org/TR/rdf-concepts`
- Primer `http://www.w3.org/TR/rdf-primer`

SPARQL

- `http://www.w3.org/TR/rdf-sparql-query`

Further Notation

Turtle

- http://www.w3.org/TeamSubmission/turtle/

```
[6] triples             ::= subject predicateObjectList
[7] predicateObjectList ::= verb objectList ( ';' verb objectList )* ( ';')?
[8] objectList          ::= object ( ',' object)*
[9] verb                ::= predicate | 'a'
```

N3

- http://www.w3.org/TeamSubmission/n3/

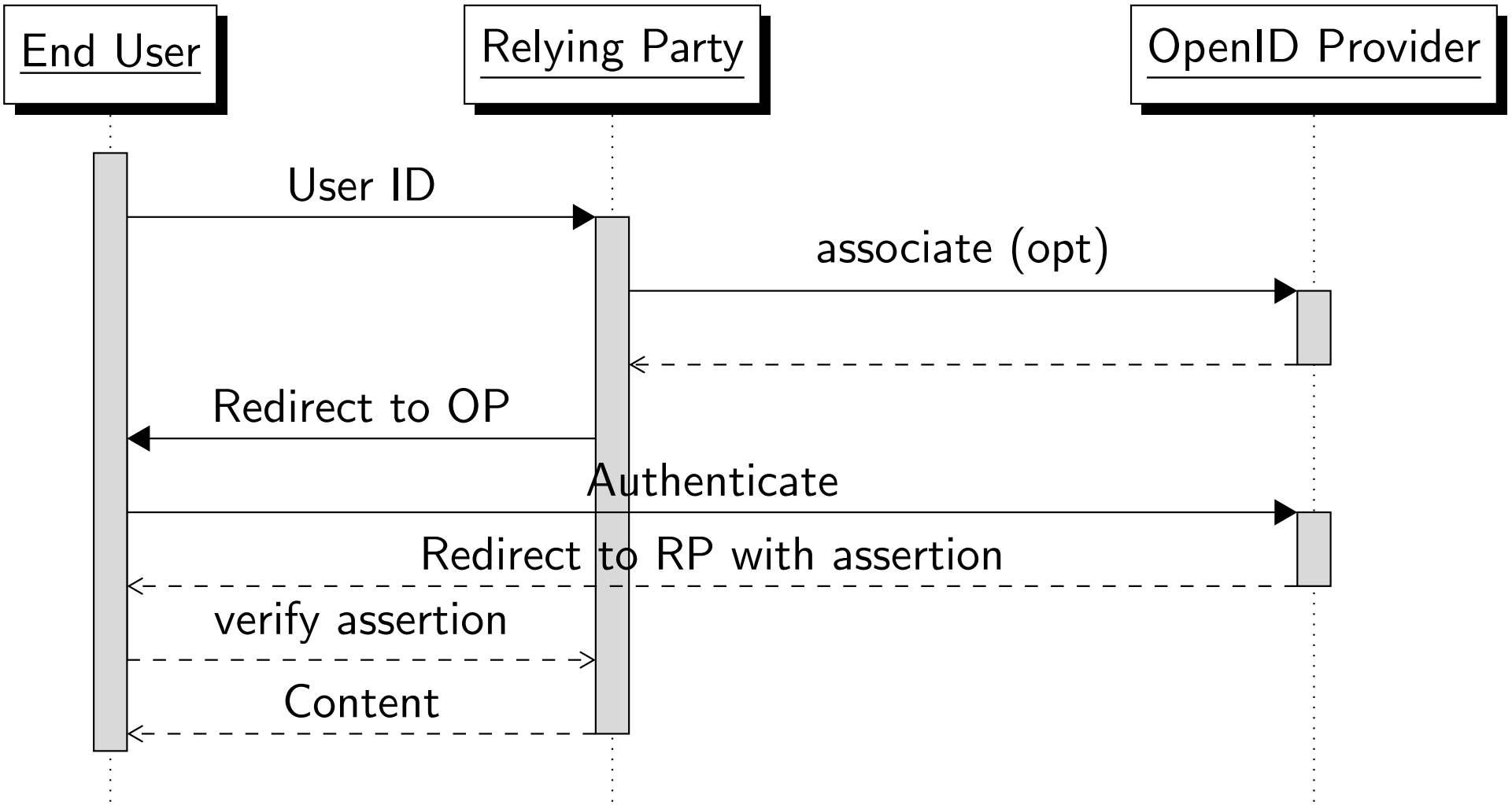

OpenID

OpenID Overview

- Federated Authentication
- OpenID Authentication 2.0 - Final: `http://openid.net`
- URI based ID
- Roles: End-User, Relying Party, OpenID Provider
- Relying Party learns attributes

next week

Protocol Overview



Protocol Messages I

- HTML 4.01 [2], UTF-8 Encoding
- "`key:value\n`"-Encoding
- HTTP: keys start with `openid.`
- HTTP-Request
  - `openid.ns:http://specs.openid.net/auth/2.0`
  - `openid.mode`
- Direct Communication
- Indirect Communication
  - HTTP Redirect or HTML Form Submission
  - ```
    HTTP/1.1 301 Moved Permanently
    Location: http://www.example.org/
    ```
    ```

Protocol Messages II

- Direct Response: no openid-prefix
- Indirect Error Response:
  - `openid.ns:http://specs.openid.net/auth/2.0`
  - `openid.mode:error`
  - `openid.error:You did wrong!`
  - `openid.contact:admin@openid-serv.ex`
  - `openid.reference:SupportTicket=0xb54`

Associations

- establish shared secret RP $\leftrightarrow$ OP
- alternative: stateless mode (Section 11.4.2)
- application of association type
  - HMAC-SHA1, HMAC-SHA256

Request:

```
openid.ns:http://specs.openid.net/auth/2.0
openid.mode:associate
openid.assoc_type:HMAC-SHA256|HMAC-SHA256
openid.session_type:no-encryption|DH-SHA1|DH-SHA256
```

Diffie-Hellman Association

Request:

```
openid.dh_modulus:\x4f\x...
openid.dh_gen:\x02
openid.dh_consumer_public:\x7c\x23...
```

Response:

```
assoc_handle:255orlesscharacters
session_type:<see request>
assoc_type:<see request>
expires_in:1987200 (seconds, base10 ASCII)
```

Request Authentication

```
openid.ns:http://specs.openid.net/auth/2.0
openid.mode:checkid_setup
openid.claimed_id:someid@openid.org
openid.return_to:https://relying.com/login
```

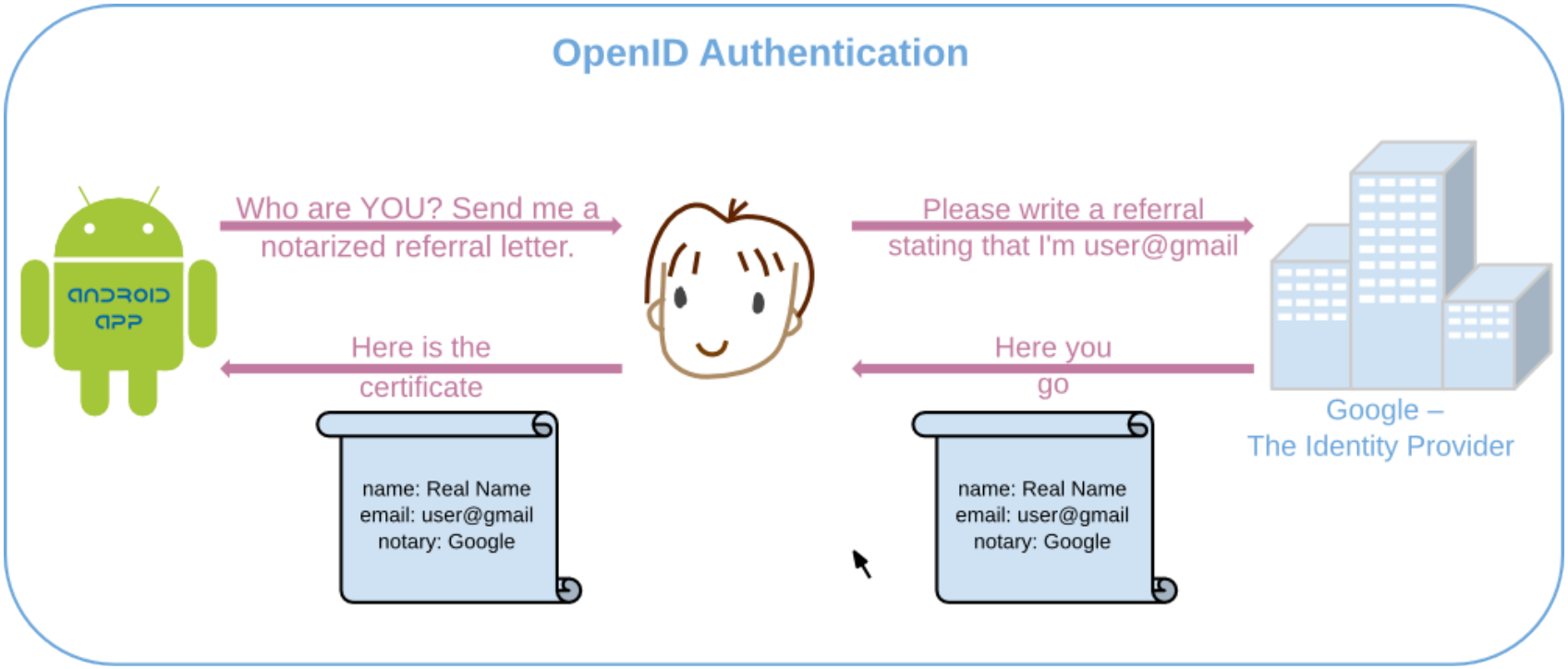
Signatures

- values of keys in `signed`:
- using association secret
- application of association type
  - HMAC-SHA1, HMAC-SHA256

Positive Assertion

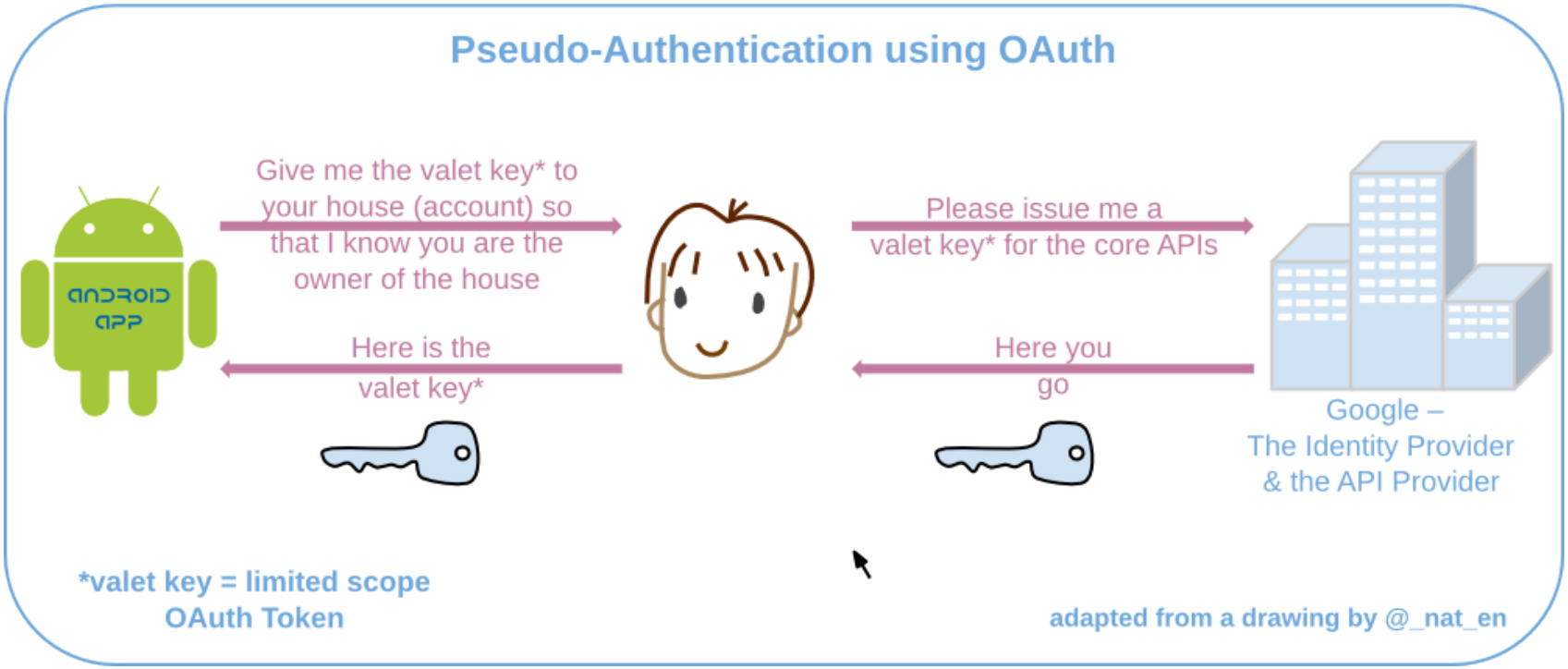
```
openid.ns:http://specs.openid.net/auth/2.0
openid.mode:id_res
openid.op_endpoint:
openid.return_to:https://relying.com/login
openid.response_nonce:2012-04-26T10:45:23SEC-OSN
openid.signed:return_to,op_endpoint,claimed_id,response_nor
openid.sig: MTY3MmFmMmZlMmE2NmIyMWMzNTQ1Cg==
```

OpenID Conclusion



[http://en.wikipedia.org/wiki/File:OpenIDvs.Pseudo-AuthenticationusingOAuth.svg]

OAuth vs. OpenID



[http://en.wikipedia.org/wiki/File:OpenIDvs.Pseudo-AuthenticationusingOAuth.svg]

OAuth

OAuth Overview

resource owner (User)

client (Consumer)

server (Service Provider)

client credentials

temporary credentials

token credentials

Objectives:

- Redirection-Based Authentication
- partial authorisation to (web)-resource
- no password disclosure to client

resource owner (User)
client (Consumer)
server (Service Provider)

client credentials
temporary credentials
token credentials

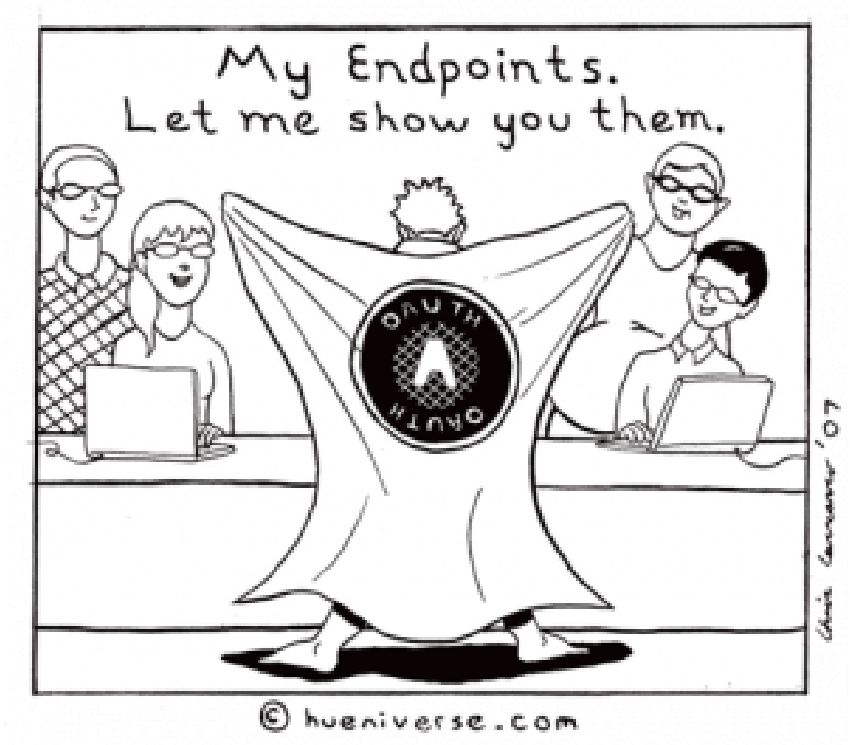
Objectives:

- Redirection-Based Authentication
- partial authorisation to (web)-resource
- no password disclosure to client

(Terms in parentheses are used in the original spec, other terminology is from the RFC.)

1. resource owner, the user that owns a web-resource and wishes to grant (temporary) access to that resource to the
2. client, who might be a secondary webservice, a mobile app or similar, that should do something with a users resource hosted at a
3. server, that hosts and controls the resource owner's data.

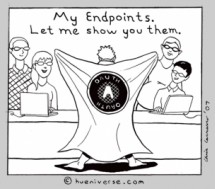
OAuth



(source: http://hueniverse.com/oauth/)

State 2012

- OAuth 1.0 RFC 5849 [3] Session Fixation Attack
- OAuth 2.0 Facebook deployed, Microsoft, Google experimental



(source: http://hueniverse.com/oauth/)

State 2012

- OAuth 1.0 RFC 5849 [3] Session Fixation Attack
- OAuth 2.0 Facebook deployed, Microsoft, Google experimental

OAuth provides a protocol to grant an application limited access for a limited time to a web-resource.Sources State 2011:

OAuth 1.0 Session Fixation `http://oauth.net/advisories/2009-1/`

OAuth 2.0

- Facebook `http://developers.facebook.com/docs/authentication/?_fb_noscript=1`
- Google `http://googlecode.blogspot.com/2011/03/making-auth-easier-oauth-20-for-google.html`
- Microsoft `http://windowsteamblog.com/windows_live/b/developer/archive/2011/05/04/announcing-support-for-oauth-2-0.aspx`

OAuth Phases

Preliminaries:

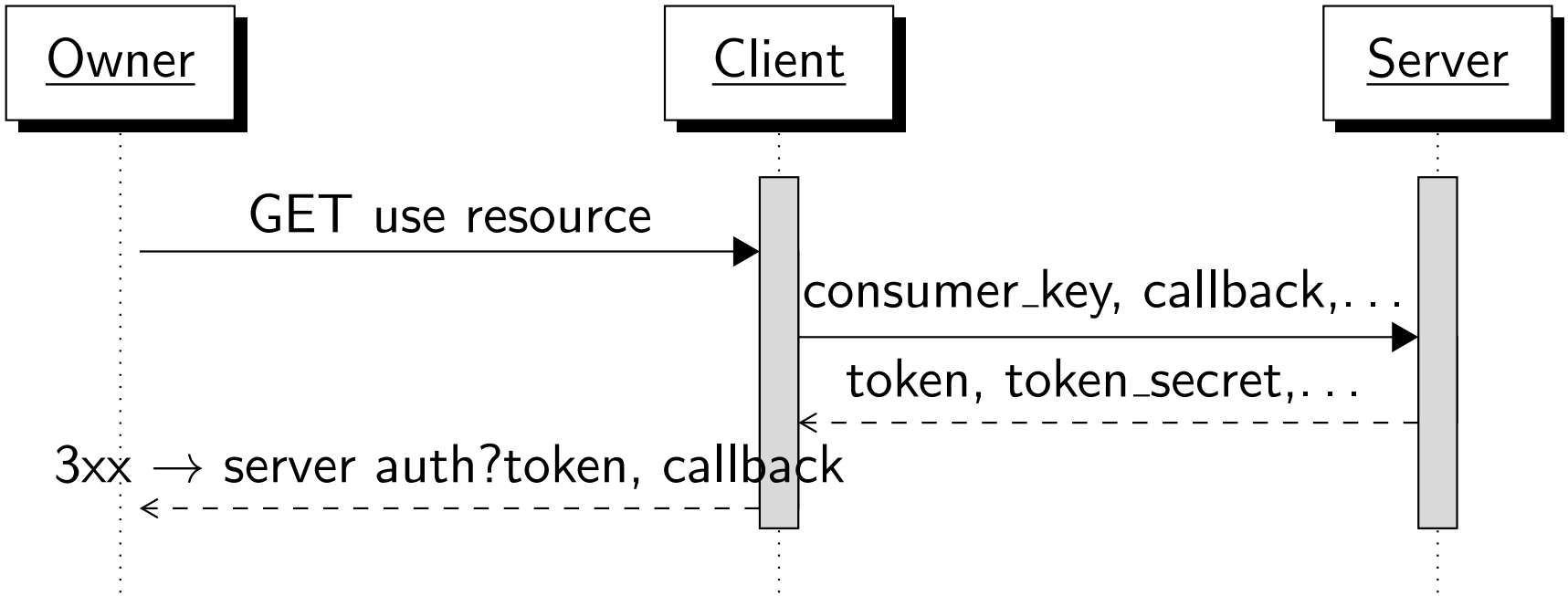
0. (Client Credentials)

Server Communication Endpoints:

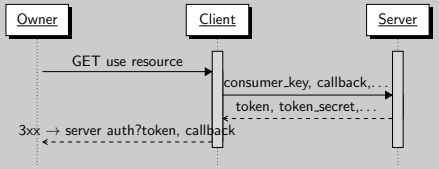
1. Temporary Credential Request
2. Resource Owner Authorisation
3. Token Request

1. *Client Credentials* are not part of the protocol, and are obtained beforehand to authenticate the client-software/instance.
2. The first step in the protocol is for the client to request a *temporary credential*. These are used for Resource Owner Authorisation and afterwards by the client to claim a token from the server.
3. The resource owner authorises the client by authenticating itself in a request to the server, including the clients temporary credential.
4. Using an previously authorised temporary credential, the client requests a token from the server for authentication of further requests.

OAuth 1.0: Temporary Credential Request



OAuth 1.0: Temporary Credential Request

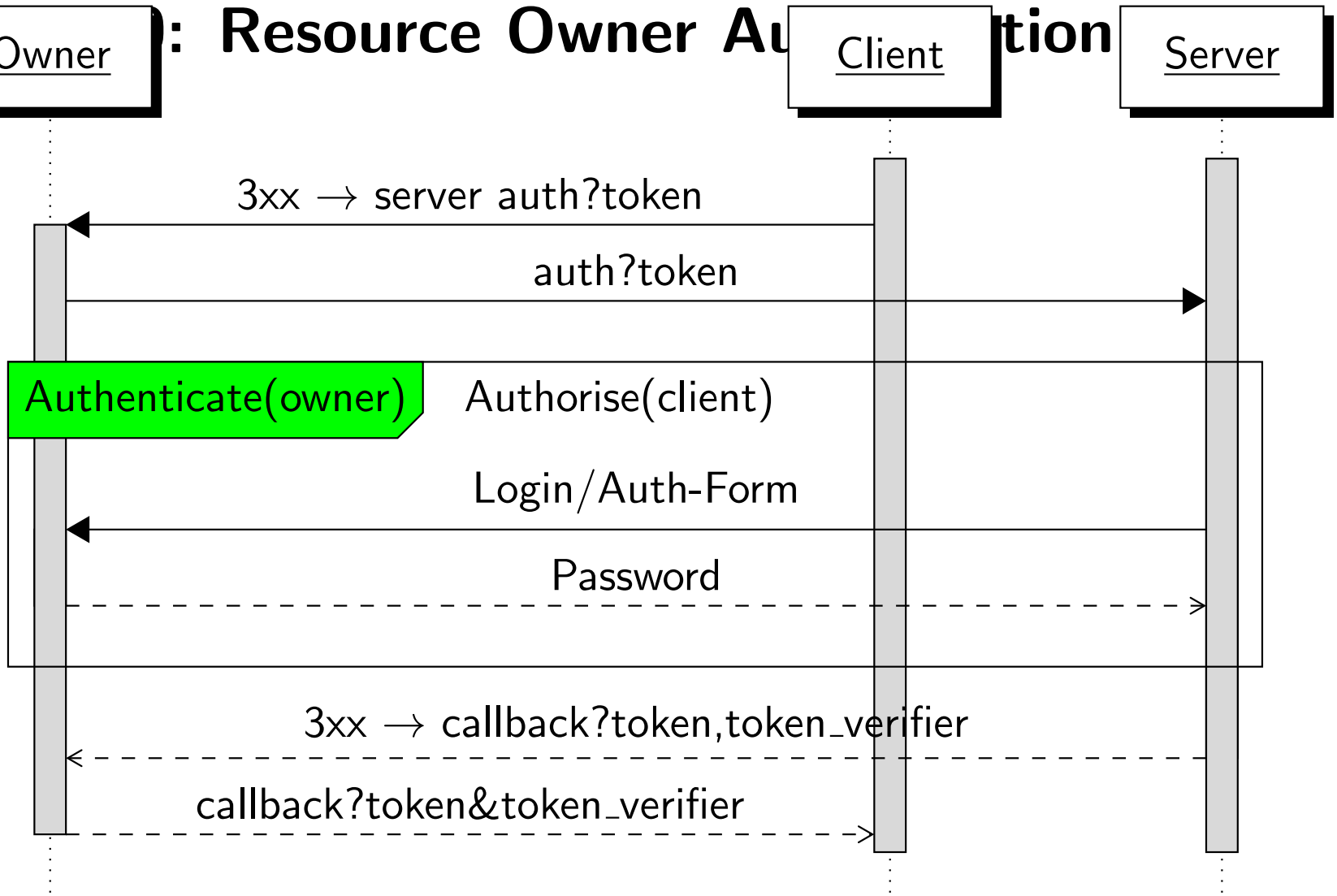


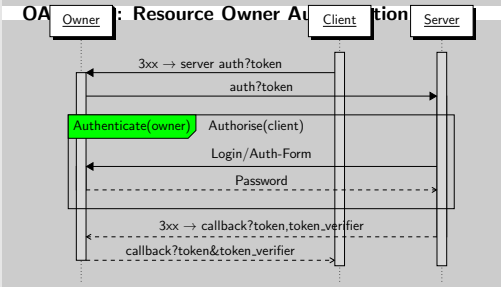
As the owner requests some action from the client, that requires access to resources on the server, the clients sends a request to the server, requesting temporary credentials (in form of a token). The client then replies to the owner with a redirection to the authorisation endpoint of the server containing the token (not the token_secret).

OAuth 1.0: Resource Owner Authorisation

The owner follows the clients redirection to the servers authorisation endpoint, using the token. There the owner authenticates himself to the server and explicitely authorises the client to access the given resource. The server then redirects the user to the client-callback previously set, providing a token_verifier. The owner calls (gets redirected) to the clients callback and thus issues the token_verifier to the client.

OA : Resource Owner Au tion



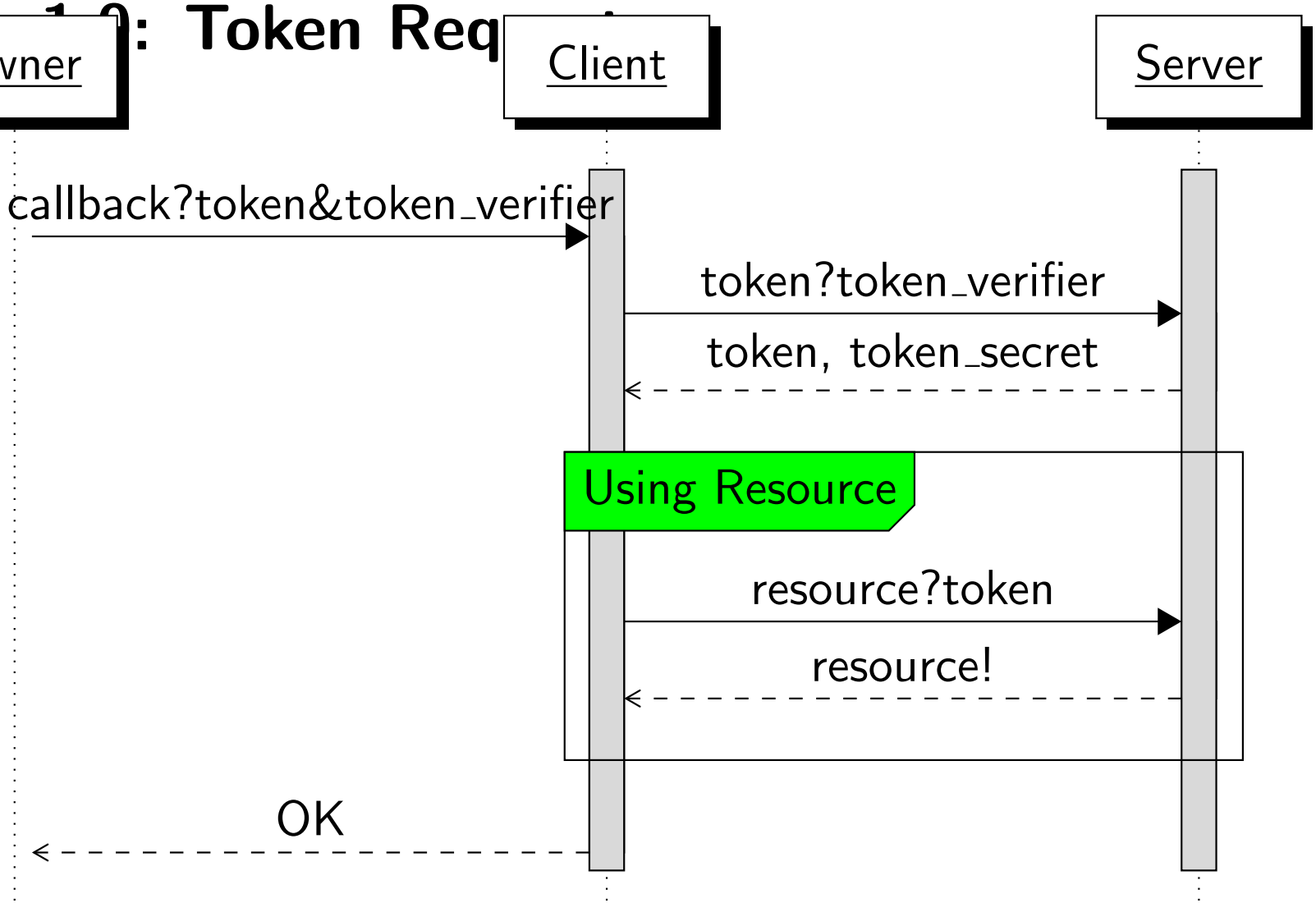


The owner follows the clients redirection to the servers authorisation endpoint, using the token. There the owner authenticates himself to the server and explicitely authorises the client to access the given resource. The server then redirects the user to the client-callback previously set, providing a token_verifier. The owner calls (gets redirected) to the clients callback and thus issues the token_verifier to the client.

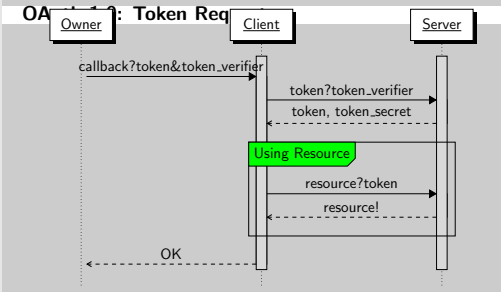
OAuth 1.0: Token Request

After the client received a token_verifier from the owner it requests an access token from the server, proving authorisation with the token_verifier. The server grants a new access token and the client can then request/use the resource. At the end it is nice to inform the owner of the result.

OA[illegible]: Token Req[illegible]


Owner
Client
Server
callback?token&token_verifier
token?token_verifier
token, token_secret
Using Resource
resource?token
resource!
OK

OAuth 1.0: Token Request



After the client received a token_verifier from the owner it requests an access token from the server, proving authorisation with the token_verifier. The server grants a new access token and the client can then request/use the resource. At the end it is nice to inform the owner of the result.

OAuth 1.0: Session Fixation

- Attacker uses (honest) client to get temp. credential
- Attacker does not follow authorisation redirect
- Attacker tricks resource owner to click redirect
- Owner authorises honest client at server
- Attacker uses saved temp. credential to request token
- Attacker uses token to access resource

(source: `http://oauth.net/advisories/2009-1/`)

OAuth 2.0

Why update?

- OAuth 1.0 too complex
- Scalability issues
- Incompatible to existing Auth. Schemes

(source http://hueniverse.com/2010/05/introducing-oauth-2-0/)

State:

- Almost stable IETF draft v2.22

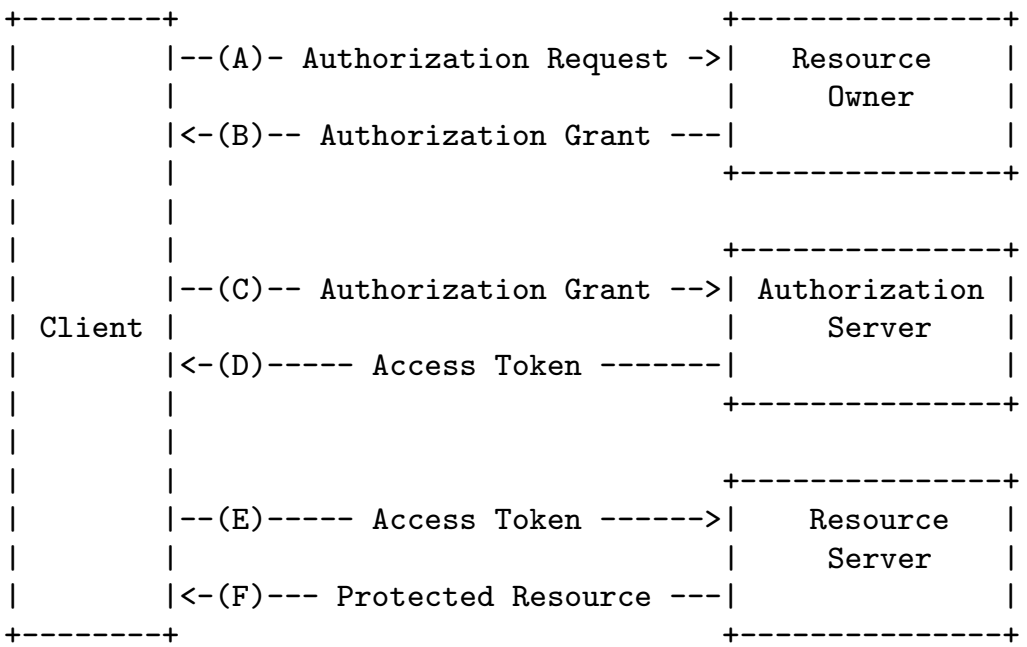
(see http://tools.ietf.org/html/draft-ietf-oauth-v2-22)

OAuth 2.0: Delta

- Role Separation: Authorization Server
- 6 Different Protocol Flows:
  - User-Agent,
  - Web-Server,
  - Device,
  - Username-Password,
  - Client Credentials,
  - Assertion (eg. SAML)
- Bearer Tokens
- Short-Lived Tokens/Long-Lived authorizations

(source `http://hueniverse.com/2010/05/introducing-oauth-2-0/`)

OAuth 2.0: Flow



(source: OAuth v2.22 (draft)

Literatur I


T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol*, Network Working Group Std., Rev. Version 1.2, 2008. [Online]. Available: http://tools.ietf.org/html/rfc5246


HTML 4.01 Specification, W3C W3C Recommendation, latest version.


The OAuth 1.0 Protocol, IETF Informational RFC 5849, April 2010.