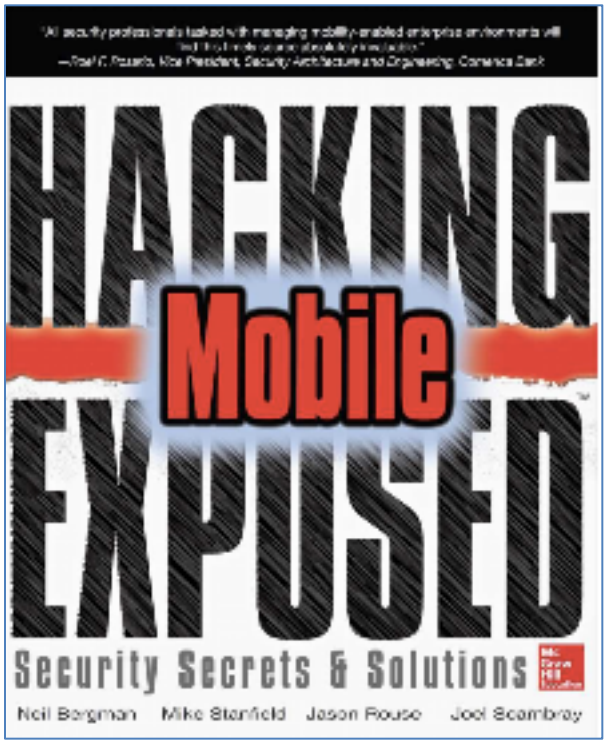
Ch 6: Mobile Services and Mobile Web Part 1



CNIT 128: Hacking Mobile Devices

Updated 2-22-17

Ch 6 Part 1
Through OAuth
Part 2 starts with SAML

Server-Side Technologies

- SQL (Structured Query Language)
  - Servers that manage databases
  - Contain SSNs, credit card numbers, sometimes passwords, etc.

SELECT * FROM sqlol.users	*Get all fields from the table "sql.users"*
SELECT * FROM sqlol.ssn	*Get all fields from the table "sql.ssn"*
SELECT name FROM sqlol.ssn	*Get field "name" from the table "sql.ssn"*

Server-Side Technologies

- SOAP (Simple Object Access Protocol)
  - XML-based middleware to exchange data between servers and clients
  - Can operate over any transport protocol such as HTTP, SMTP, TCP, UDP, or JMS (link Ch 6a)
    - Examples on next slides from link Ch 6l

Here is the SOAP request –

```
POST /Quotation HTTP/1.0
Host: www.xyz.org
Content-Type: text/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope" 

   <SOAP-ENV:Body xmlns:m="http://www.xyz.org/quotations" >

      <m:GetQuotation>
         <m:QuotationsName>MiscroSoft</m:QuotationsName>
      </m:GetQuotation>

   </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

A corresponding SOAP response looks like –

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope" S

   <SOAP-ENV:Body xmlns:m="http://www.xyz.org/quotation" >

      <m:GetQuotationResponse>
         <m:Quotation>Here is the quotation</m:Quotation>
      </m:GetQuotationResponse>

   </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

JSON v. XML

- From link Ch 6n

JSON

```
{
    name: 'Asahi Draft Beer',
    brewer: {
        name: 'Asahi',
        country: 'Japan'
    },
    calories: 41,
    alcohol:  '5.21'
}
```

XML

```
<beer>
    <name>Asahi Draft Beer</name>
    <brewer>
        <name>Asahi</name>
        <country>Japan</country>
    </brewer>
    <calories>41</calories>
    <alcohol>5.21</alcohol>
</beer>
```

Server-Side Technologies

- ReST (Representational State Transfer)
  - Uses HTTP to transfer data between machines
  - The World Wide Web can be viewed as a REST-based architecture (link Ch 6c)
  - Send data with PUT, get it with GET (link Ch 6n)

```
curl -X PUT \
  --anyauth --user username:password \
  -d '{"name": "Iced Mocha", "size": "Grandé", "tasty": true }' \
  'http://myhost:port/v1/documents?uri=/afternoon-drink.json'
```

```
curl -X GET \
  --anyauth --user username:password \
  'http://myhost:port/v1/documents?uri=/afternoon-drink.json'
```

Server-Side Technologies

- JSON (JavaScript Object Notation)
  - Lightweight data-interchange format
  - An alternative to XML (link Ch 6b)
  - Example from link Ch 6m

```
{
  "items": [
    {
      "key": "First",
      "value": 100
    },{
      "key": "Second",
      "value": false
    },{
      "key": "Last",
      "value": "Mixed"
    }
  ],
  "obj": {
    "number": 1.2345e-6,
    "enabled": true
  },
  "message": "Strings have to be in double-quotes."
}
```

Server-Side Vulnerabilities

- Expose far more data than client-side vulnerabilities
- Larger attack surface than client
  - Server runs services, some for clients, others for business logic, internal interfaces, databases, partner interfaces, etc.

General Web Service Security Guidelines: OWASP Top Ten Mobile Risks 2016

Link Ch 6k

M1 - Improper Platform Usage	This category covers misuse of a platform feature or failure to use platform security controls. It might include Android intents, platform permissions, misuse of TouchID, the Keychain, or some other security control that is part of the mobile operating system. There are several ways that mobile apps can experience this risk.
M2 - Insecure Data Storage	This new category is a combination of M2 + M4 from Mobile Top Ten 2014. This covers insecure data storage and unintended data leakage.
M3 - Insecure Communication	This covers poor handshaking, incorrect SSL versions, weak negotiation, cleartext communication of sensitive assets, etc.
M4 - Insecure Authentication	This category captures notions of authenticating the end user or bad session management. This can include: • Failing to identify the user at all when that should be required • Failure to maintain the user's identity when it is required • Weaknesses in session management
M5 - Insufficient Cryptography	The code applies cryptography to a sensitive information asset. However, the cryptography is insufficient in some way. Note that anything and everything related to TLS or SSL goes in M3. Also, if the app fails to use cryptography at all when it should, that probably belongs in M2. This category is for issues where cryptography was attempted, but it wasn't done correctly.

M6 - Insecure Authorization	This is a category to capture any failures in authorization (e.g., authorization decisions in the client side, forced browsing, etc.). It is distinct from authentication issues (e.g., device enrolment, user identification, etc.). If the app does not authenticate users at all in a situation where it should (e.g., granting anonymous access to some resource or service when authenticated and authorized access is required), then that is an authentication failure not an authorization failure.
M7 - Client Code Quality	This was the "Security Decisions Via Untrusted Inputs", one of our lesser-used categories. This would be the catch-all for code-level implementation problems in the mobile client. That's distinct from server-side coding mistakes. This would capture things like buffer overflows, format string vulnerabilities, and various other code-level mistakes where the solution is to rewrite some code that's running on the mobile device.
M8 - Code Tampering	This category covers binary patching, local resource modification, method hooking, method swizzling, and dynamic memory modification. Once the application is delivered to the mobile device, the code and data resources are resident there. An attacker can either directly modify the code, change the contents of memory dynamically, change or replace the system APIs that the application uses, or modify the application's data and resources. This can provide the attacker a direct method of subverting the intended use of the software for personal or monetary gain.
M9 - Reverse Engineering	This category includes analysis of the final core binary to determine its source code, libraries, algorithms, and other assets. Software such as IDA Pro, Hopper, otool, and other binary inspection tools give the attacker insight into the inner workings of the application. This may be used to exploit other nascent vulnerabilities in the application, as well as revealing information about back end servers, cryptographic constants and ciphers, and intellectual property.
M10 - Extraneous Functionality	Often, developers include hidden backdoor functionality or other internal development security controls that are not intended to be released into a production environment. For example, a developer may accidentally include a password as a comment in a hybrid app. Another example includes disabling of 2-factor authentication during testing.

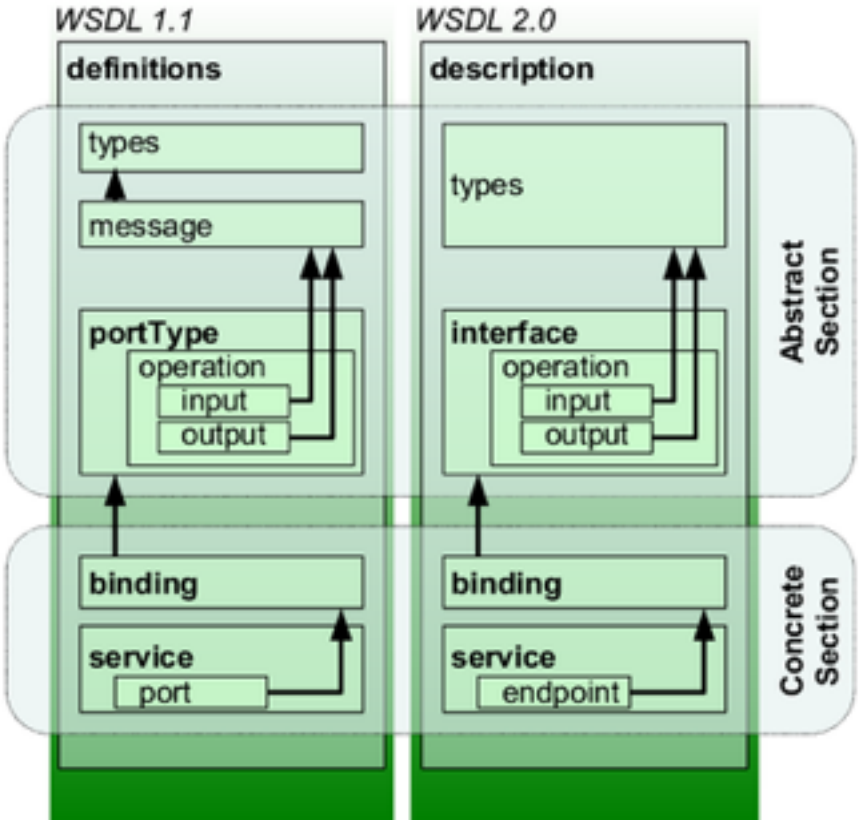
Attacks Against XML-Based Web Services

Security Audit of XML-Based Web Service

- Identify web service endpoints
  - By examining source code of client
  - Or examining Web traffic while client runs
- Craft legitimate web service requests
  - For all endpoints and operations
- Vulnerability Discovery
  - By altering structure of contents of XML documents sent to the web service endpoints

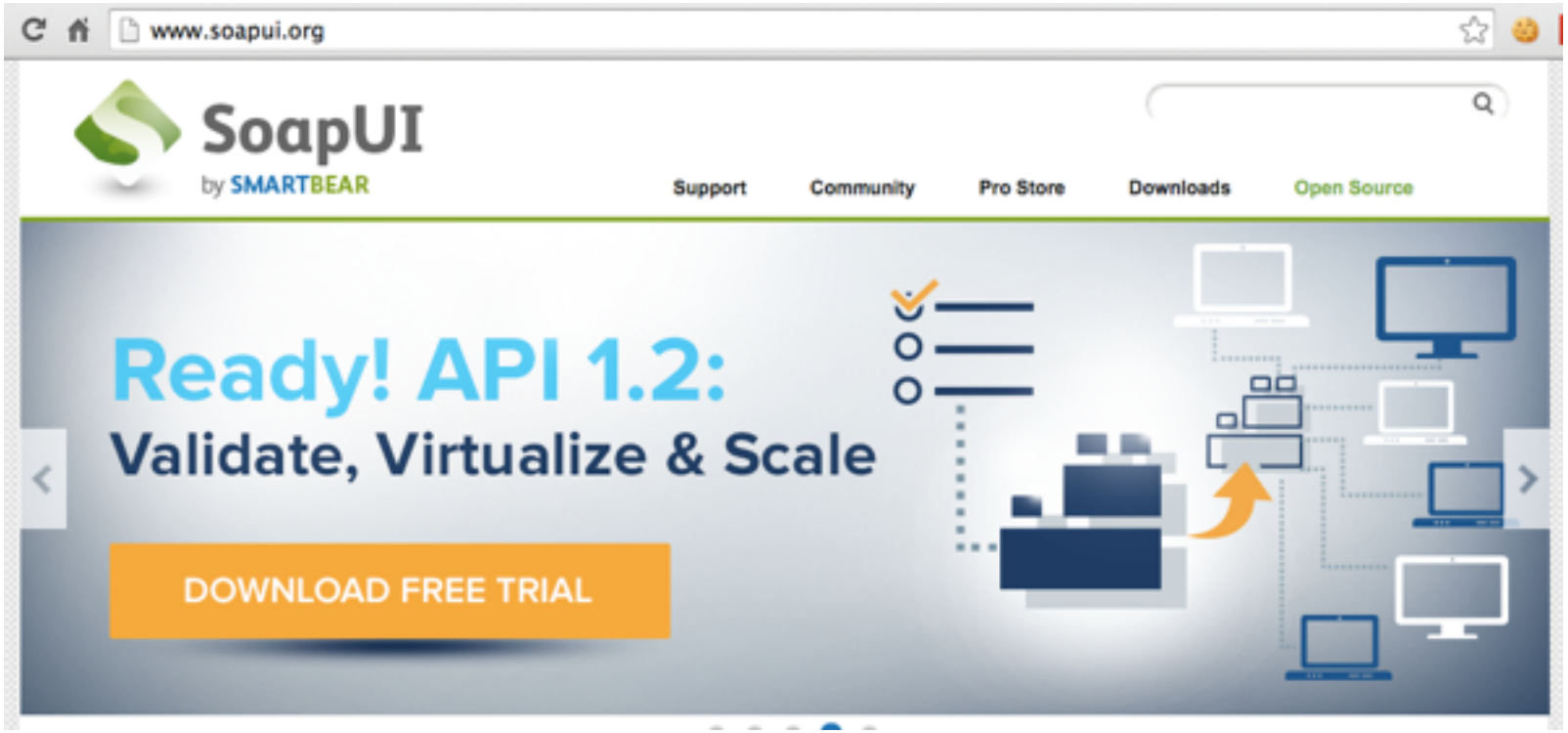
Web Services Description Language (WSDL)

- An XML-based interface description language
  - Used to describe functionality offered by a Web service
  - Image from Wikipedia (link Ch 6f)



SoapUI

- SoapUI can build a set of base test cases given a URL to an identified WDSL
  - Link Ch 6g



XML Injection Example

- Client sends

```
<?xml version="1.0"?>
<ProductRequest>
   <Id>584654</Id>
</ProductRequest>
```

- Sever replies, echoing Id from client

```
<?xml version="1.0"?>
<ProductResponse>
   <Id>584654</Id>
   <Price>199.99</Price>
</ProductResponse>
```

XML Injection Example

- Client sends an Id of

```
584654</Id><Price>0.99</Price>
</ProductResponse>
<ProductResponse><Id>123
```

- Sever reply becomes

```
<?xml version="1.0"?>
<ProductResponse>
   <Id>584654</Id><Price>0.99</Price>
</ProductResponse>
<ProductResponse><Id>123
</Id>
   <Price>199.99</Price>
</ProductResponse>
```

Effect of XML Injection

- Depends on how server handles a strange response like that
- Most would accept the first XML portion with the modified price

XML Injection Countermeasures

- Input validation
  - Best done with whitelisting (allowing only known-good characters)
- Output encoding
  - Change "<" to "<"
- Use encoding functions from a trusted source, such as OWASP

XML Entity Expansion

- A Denial-of-Service (DoS) attack using XML entities that expand greatly at process time
- The example sends a 662-byte request that expands to 20 MB at the server
- Enough of these requests can stop a server by RAM exhaustion

XML Entity Expansion

```
<?xml version="1.0"?>
<!DOCTYPE root [ <ENTITY a1 "I've often
seen a cat without a grin..."> ]>
<someElement1><someElement2>&a1;
</someElement2></someElement1>
```

Expands to

```
<?xml version="1.0"?>
<someElement1><someElement2>
I've often seen a cat without a grin...</
someElement2></someElement1>
```

XML Entity Expansion Example

```
POST /SomeWebServiceEndpoint HTTP/1.1
Host: www.example.com
Content-Length: 662

<?xml version="1.0"?>
<!DOCTYPE root [
<ENTITY a1 "I've often seen a cat without a grin...">
<ENTITY a2 "&a1;&a1;"><ENTITY a3 "&a2;&a2;">
<ENTITY a4 "&a3;&a3;"><ENTITY a5 "&a4;&a4;">
...
<ENTITY a20 "&a19;&a19;">
 ]>
<someElement1><someElement2>&a20;</someElement2></
someElement1>
```

XML Entity Expansion Countermeasures

- Disable Document Type Definitions (DTDs) in the XML parser
- Set a limit on the depth of entity expansions in the parser
- Note: phones have XML parsers too, and can be attacked the same way
  - The iOS NSXMLParser parser is protected
  - But not Android's SAXParser

XML Entity Reference

- Abuse XML entities to acquire the contents of files on the Web server
- The example on the next page defines an external entity reference "fileContents" that points to the hosts file on Windows and uses it

XML Entity Reference Example

```
POST /SomeWebServiceEndpoint HTTP/1.1
Host: www.example.com
Content-Length: 196


<?xml version="1.0"?>
<!DOCTYPE fileDocType = [
<ENTITY fileContents SYSTEM "C:\Windows
\System32\drivers\etc\hosts">
 ]>
<someElement1><someElement2>&fileContents;</
someElement2></someElement1>
```

XML Entity Reference

- If the XML parser supports DTDs with external entities
  - Many parsers do by default
  - The parser will fetch the host file and may display the file in the XML response to the attacker
  - It's limited only by file permissions
  - If the Web service runs as root, it can read any file

XML Entity Reference

- Can be used for DoS by
  - Requesting a special device file, or
  - Forcing the parser to make many HTTP requests to remote resources, exhausting the network connection pool

XML Entity Reference Countermeasures

- Disable DTDs altogether if you don't need them
- Allow DTDs that contain general entities, but
  - Prevent the processing of external entities
- Set up an EntityResolver object to limit access to a whitelist of resources

XML Entity Reference

- Can attack phones too
  - Android is vulnerable and the same countermeasures apply
  - The iOS NSXMLParser class does not handle external entities by default, but a developer can enable this dangerous functionality

Common Authentication and Authorization Frameworks

Authentication Issues

- Web apps typically authenticate with passwords
  - So do mobile apps
- Users don't want to type in the password every time they use an app
  - Storing user's credentials in plaintext is unwise

Credential Storage Options

- Secure Element (SE)
  - A special tamper-resistant hardware component
  - Not present in all phones, although some have one for NFC payment (link Ch 6h)
- Authorization Framework
  - Such as OAuth
  - First authenticates a user with a password
  - Creates a **token** to be stored on the phone
  - Less valuable than a password to an attacker

Recommendations

- Token can be made less dangerous
  - Set reasonable expiration dates
  - Restrict token's scope
  - Revoke tokens that are known to be compromised
- For financial apps
  - Don't store any token at all on client-side
  - Force the user to authenticate each time the app is used

OAuth 2
Open Authorization

- Popular
  - Used by Google, Facebook, Yahoo!, LinkedIn, and PayPal
- Allows one app to access protected resources in another app without knowing the user's credentials
  - Like Microsoft's Federated Identity Management

Main Actors in OAuth 2

- Resource owner
  - End-user with access to credentials, who owns the protected resources
- Resource server
  - Server hosting protected resources
  - Allows client access to the protected resources when provided a valid access token

Main Actors in OAuth 2

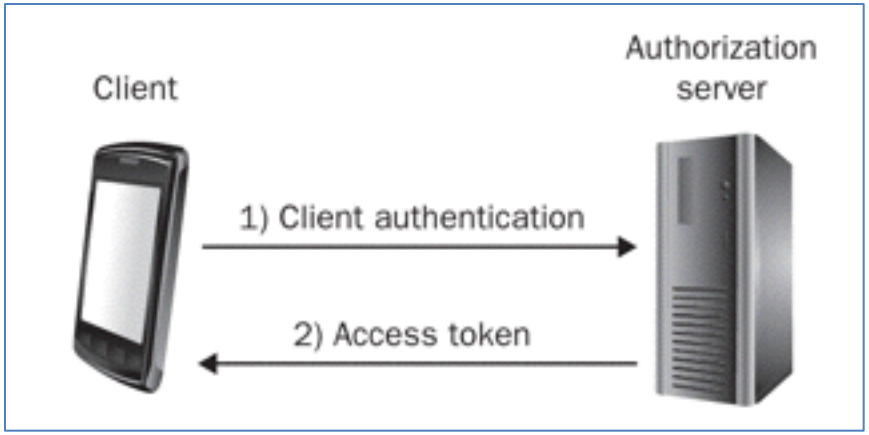
- Client
  - App seeking to access protected resources
  - Typically a mobile app or web app
- Authorization Server
  - Server that provides the client application with access tokens
    - After the resource owner has provided valid credentials

OAuth 2 has Four Grant Types

- Client Credentials Grant
- Resource Owner Password Credentials Grant
- Authorization Code Grant
- Implicit Grant

- "User Agent" in these diagrams is either your mobile browser or a WebView component embedded within the application

OAuth Client Credentials Grant

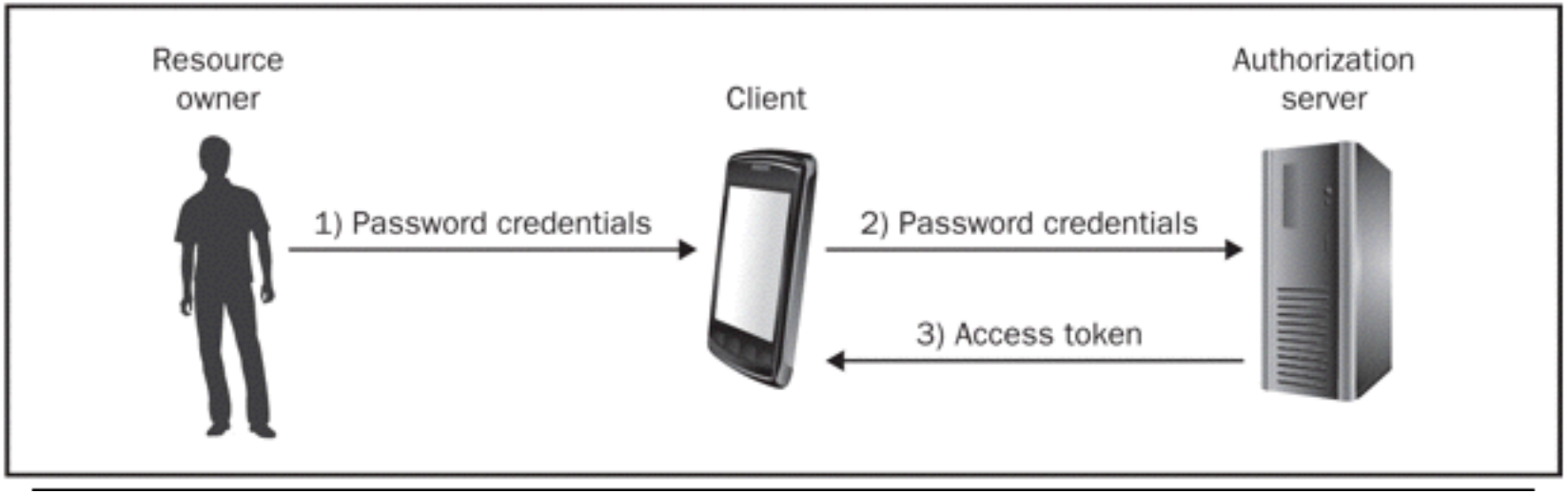


- Stores password on client
- Should only be used for confidential clients
  - That can maintain the confidentiality of their credentials
- Not usually appropriate for mobile devices because of device theft

OAuth Client Credentials Grant

- OK if mobile app has access to a Secure Element (SE)
  - But most mobile apps cannot interface with a SE
- This grant type should be avoided
  - Unless app takes additional steps to protect authentication info
  - Such as forcing user to enter a complex password every time the app launches
  - Password used to encrypt/decrypt authentication info

OAuth Resource Owner Password Credentials Grant Type

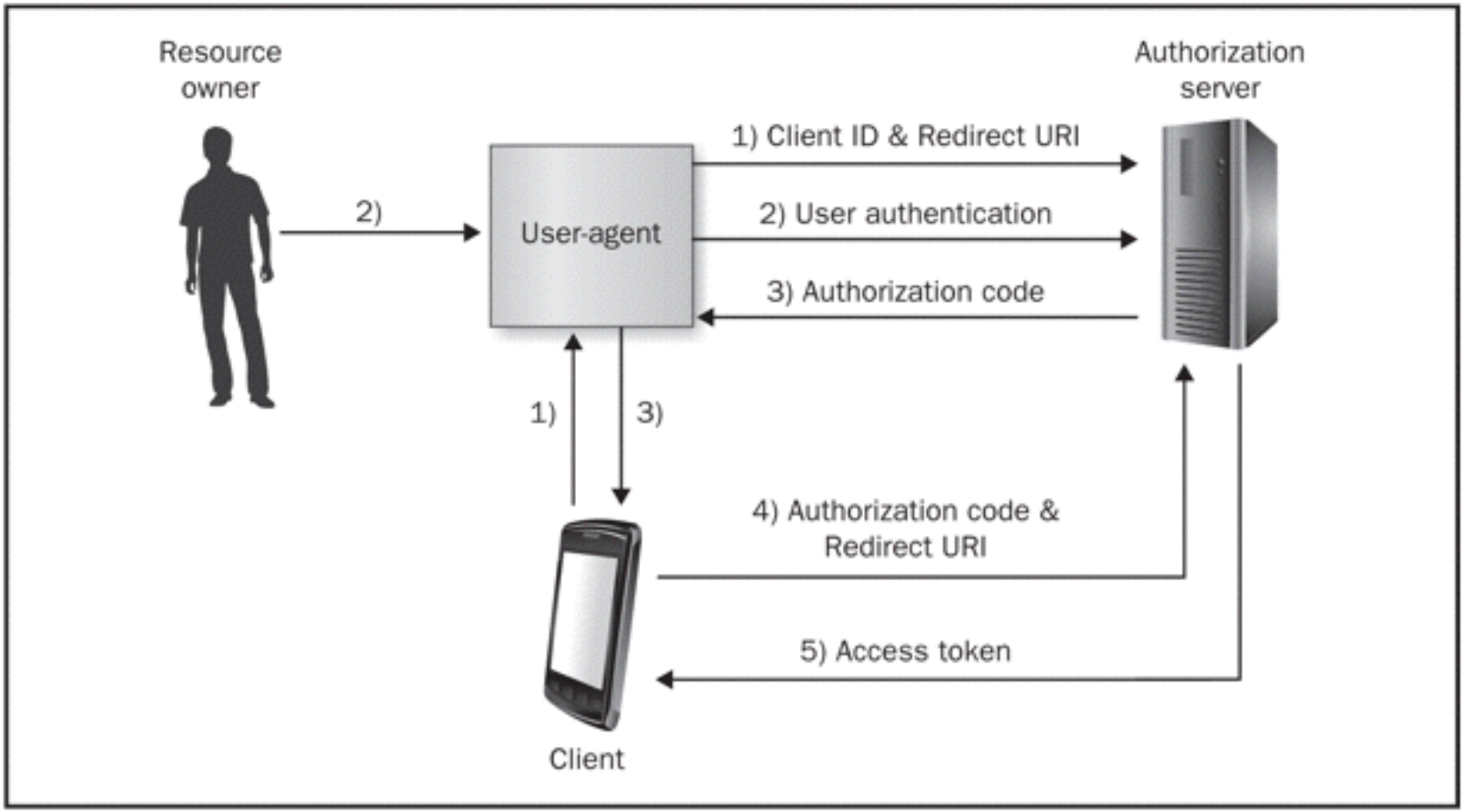


- App is trusted with credentials, but need not save them
- It can save the token instead

OAuth Resource Owner Password Credentials Grant Type

- OK if
  - Client app is trusted not to leak credentials to a third party
  - Same entity controls authorization server, resource server, and client app
- Better than storing credentials in plaintext on the mobile device and submitting them in every HTTP request

OAuth Authorization Code Grant Type



OAuth Authorization Code Grant Type

- 1. Client directs user-agent (browser or WebView component) to authorization endpoint
- Request includes
  - Client identifier
  - Requested scope
  - Local state
  - Redirection URI

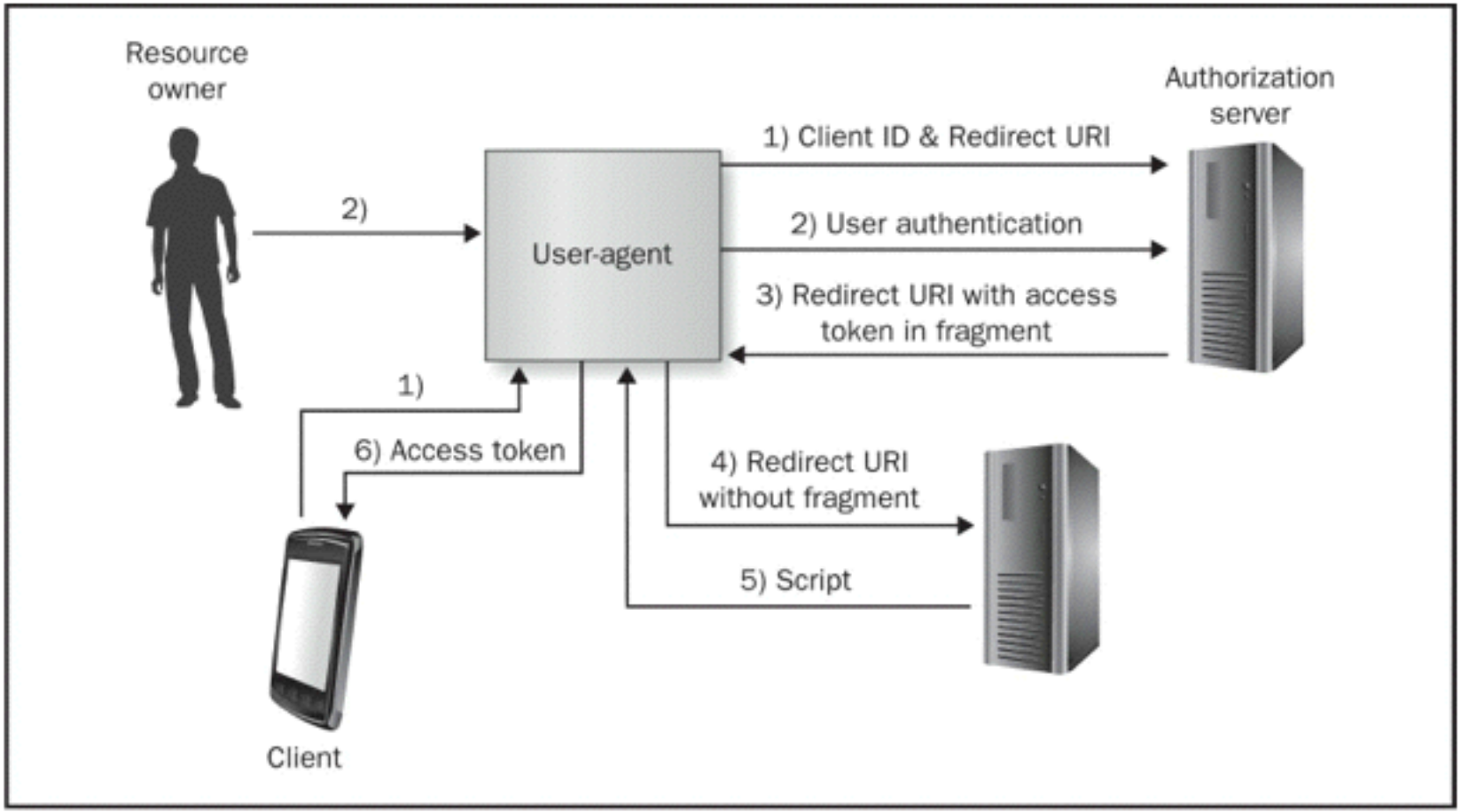
Using Mobile WebView to Steal Credentials

- In theory, client app cannot access resource owner's credentials
  - Because resource owner types credentials on the authorization server's web page
  - Via user-agent, typically a browser
- If a mobile app uses a WebView component instead of an external mobile browser
  - Client app can steal credentials with malicious JavaScript

URL Redirection Attacks

- The URI in steps 1 and 4 could be malicious, sending the client to a dangerous website
  - This could phish users, or steal tokens
- URIs should be validated, and enforced to be equal in steps 1 and 4

OAuth Implicit Grant Type



https://samsclass.info/128/128_S15.shtml#projects

- The "#projects" is a **fragment**
- Not sent to server in HTTP request
  - Used only by the browser to display the specified potion of the page
    - Link Ch 6i

OAuth Implicit Grant Type

- Token not sent to server in step 4
- In step 5, server sends JavaScript to get the token
- Intermediate servers cannot see data stored in the fragment
- Fragment does not appear in an unencrypted form in client or web server logs
  - Limits some information leakage vulnerabilities

General OAuth Threats

Lack of TLS Enforcement

- OAuth does not support message-level confidentiality or integrity
- Must use TLS to prevent sniffing of
  - Authorization tokens
  - Refresh tokens
  - Access tokens
  - Resource owner credentials

Cross-Site Request Forgery (CSRF)

- Attacker can steal a token by tricking the user into visiting a malicious URL like

  <img src="http://www.example.com/oauth_endpoint?code=attacker_code">
- Will steal token
- Tokens can be re-used, unless optional "state" parameter is enabled in OAuth 2

Improper Storage of Sensitive Data

- Server-side has many tokens and credentials
- Must be secured against attack with cryptographic controls

Overly Scoped Access Tokens

- Scope: level of access a token grants
- Token may enable
  - Sending social networking messages on your behalf, or
  - Merely viewing portions of your social messaging profile
- Follow principle of least privilege

Lack of Token Expiration

- Tokens that don't expire and are overly scoped are almost as good as stealing credentials
  - Sometimes even better
  - A password reset may not cause old tokens to expire