Authentication and Authorization

Identifying who you are and what you can do

Michael L Perry
qedcode.com
@michaellperry

pluralsight
hardcore dev and IT training

Passwords

- Reuse
- Dictionary words
- Not enough entropy

Password Attacks

- **Online attacks**
- **Phishing**
- **Social engineering**
- **Key loggers**

X.509 Certificates

No cryptographic solutions

Offline Password Attacks

- **Read only access to the password database**
  - SQL injection
  - Insider
  - Presume read-only
  - Assume it will happen

Passwords Stored in the Clear

Username	Password
jackharkness	doctor
sarajaneparker	doctor
rosetyler	b4dw01f!
amiliapond	r1v3rs0ng

- **Attacker immediately has access to all passwords**
- **Can send someone their password in email**
- **Hash the passwords**

Hashed Passwords

Username	Hashed Password
jackharkness	MIIBOQIBAAJBAKCCQtSbrS
sarajaneparker	MIIBOQIBAAJBAKCCQtSbrS
rosetyler	QdbtF2qNv7sQBHMvAwv4Ov
amiliapond	VJzH3Y439CnSw04IwbaYsR/H

- **If two people used the same password, they will have the same hash**
- **Offline dictionary attack**
- **Precomputed list of hashes for dictionary words**

Precomputed Hashes

Password	Password
god	H8v2SFLwbqIOYnpLjAxs1R
doctor	MIIBOQIBAAJBAKCCQtSbrS
love	RXAE1tZUi0Xi2G+lAiE
bacon	w4bjNc1UR9k9oJ0ITbDL0X

- **Dictionary words**

Rainbow Table

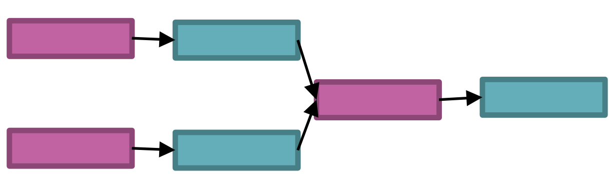
Hash: (text) -> (binary) e.g. SHA-1

Retry: (binary) -> (text) Not an inverse!

ae8c2 -> -> 0xE428F7 -> -> n3g72l -> -> 0x75D408 ->

-> db27 -> -> 0x49B91D0

0x75D408 -> -> db27 -> -> 0x49B91D0



Salt

- **Random input to hash function**
- **Thwarts precomputed attacks**
- **Need the salt to validate password**
- **Salt is not protected**

doctor tAoOcoa xe1ccArPVzDpwFpiT

Salted Hashed Passwords

Username	Hashed Password	Salt
jackharkness	xe1ccArPVzDpwFpiT	tAoOcoa
sarajaneparker	Ui0Xi2G+lAiEAydwr	VscW+jW
rosetyler	Nc1UR9k9oJ0ITbDL0X	GEH0t
amiliapond	LwbqIOYnpLjAxs1R	IgYJuR

- **Precomputed hash tables (rainbow tables) not effective**
- **Dictionary attacks are**
- **Need high entropy passwords**

Entropy

- From Information Theory
- Amount of information in a message
- Measured in bits

Computing Password Entropy

$$H = L \log_2 N$$

L is length of password
N is size of alphabet

Random Letters

vlwusgalfi

$L = 10$
$N = 26$

$$47 = 10 \log_2 26$$

Dictionary Words

troubador

N = Number of words

$20{,}000 < N < 1{,}000{,}000$

$$14 < \log_2 N < 20$$

Passphrase

correct horse battery staple

N = 20,000
L = 4

$$56 = 4 \log_2 20000$$

xkcd.com/936

* Some estimates based on a smaller lexicon

Computing Password Entropy

$$H = L \log_2 N$$

- **Common substitutions (`0`=`o`, `1`=`l`, `1`=`i`, etc)**
  - Log_2 of size of substitution dictionary times number of characters
- **Dictionary words**
  - Log_2 of size of dictionary times number of words
- **Capitalization**
  - Mostly caps or mostly lower?
  - 1 bit for each different capital not at the beginning of a word
- **Remaining characters**
  - Log_2 of size of alphabet times the number of characters

Minimum Allowable Entropy

- **Around 40 bits for most systems**

Username	Hashed Password	Salt
jackharkness	xe1ccArPVzDpwFpiT	tAoOcoa
sarajaneparker	Ui0Xi2G+lAiEAydwr	VscW+jW
rosetyler	Nc1UR9k9oJ0ITbDL0X	GEH0t
amiliapond	LwbqIOYnpLjAxs1R	IgYJuR

Brute Force

Username	Hashed Password	Salt
jackharkness	xe1ccArPVzDpwFpiT	tAoOcoa
sarajaneparker	Ui0Xi2G+lAiEAydwr	VscW+jW
rosetyler	Nc1UR9k9oJ0ITbDL0X	GEH0t
amiliapond	LwbqIOYnpLjAxs1R	IgYJuR

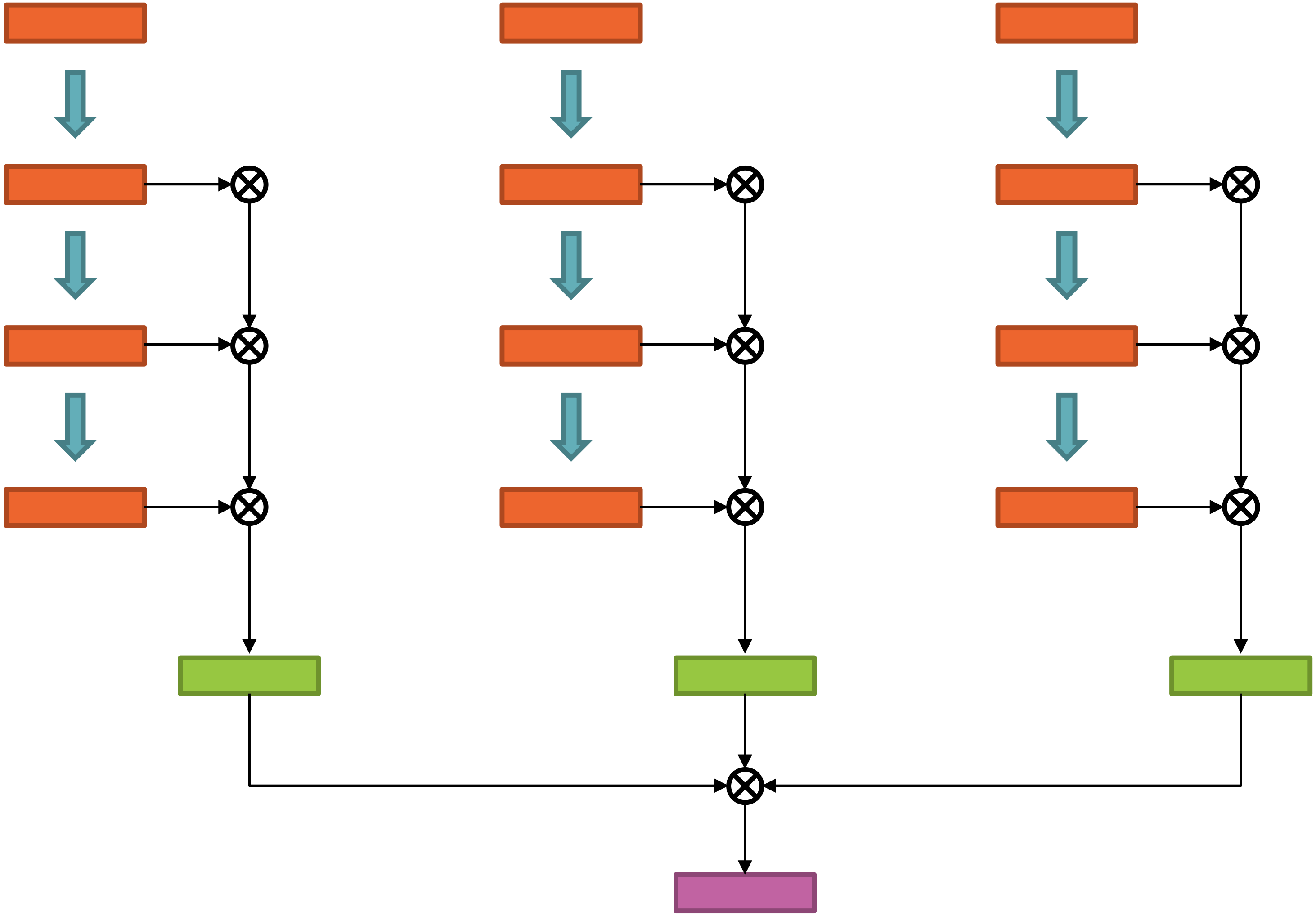
- **Try one password, one user at a time**
- **Make it take a long time**
- **Hash function is cheap**

Password Based Key Derivation Function (PBKDF)

- **Intended for deriving a symmetric key and IV from a password**
  - Used in openssl when encrypting an RSA key with AES-256
  - Can be used to generate salted hash
- **Slow down an offline attack**
- **Key stretching**

PBKDF

PBKDF



Iteration Count

- Should be at least 1,000
- Try 10,000
- Use the maximum number of iterations that your performance requirements can tolerate

PBKDF in Java

```
SecretKeyFactory f = SecretKeyFactory.getInstance(
    "PBKDF2WithHmacSHA1");

KeySpec ks = new PBEKeySpec(
    password, salt, 10000, 128);

SecretKey s = f.generateSecret(ks);

Key k = new SecretKeySpec(s.getEncoded(), "AES");
```

PBKDF in .NET

```
string hash = Crypto.HashPassword(password);
    // SHA-1
    // 128-bit salt
    // 256-bit subkey
    // 1000 iterations

    // Base-64 encoded hash
    // Salt is machine key
```

- Fixed number of iterations
- Fixed hashing algorithm
- Same salt for all users

PBKDF in .NET

```
var d = new Rfc2898DeriveBytes(
    password, salt, 10000);

byte[] hash = d.GetBytes(32);
    // SHA-1
```

- **Fixed hashing algorithm**

Progressive Salted Hashed Passwords

Username	Hashed Password	Salt	AlgID
jackharkness	xe1ccArPVzDpwFpiT	tAoOcoa	1
sarajaneparker	Ui0Xi2G+lAiEAydwr	VscW+jW	1
rosetyler	Nc1UR9k9oJ0ITbDL0X	GEH0t	1
amiliapond	LwbqIOYnpLjAxs1R	IgYJuR	2

- **Foreign key**
- **Hash algorithm, number of iterations**
- **Rehash as user logs in to migrate**
- **Any system can validate passwords**
- **Algorithm is also available to attackers**
  - Obfuscation is not the goal

Federation

- Remove the responsibility of identity from applications
- Separate authentication from authorization
- Based on trust

Factory Example

Philip

Stacy

Ralph

This is Michael

-- Philip

This person is a machinist

-- Stacy

- **Prove identity only to Philip**
- **Tell job function only to Stacy**
- **Ralph can focus on the job**

Separation of Responsibilities

- **Authentication**
  - Who you are
  - Philip
- **Authorization**
  - What you can do
  - Stacy
- **Application**
  - Getting the job done
  - Ralph

Federation Roles

- **Identity Provider (IP) (Philip)**
  - Performs authentication
  - Centralized identity management
- **Secure Token Service (STS) (Stacy)**
  - Performs authorization
  - Single repository of roles and responsibilities
- **Relying Party (RP) (Ralph)**
  - Consumes the tokens and acts upon those claims
  - Focus on business logic

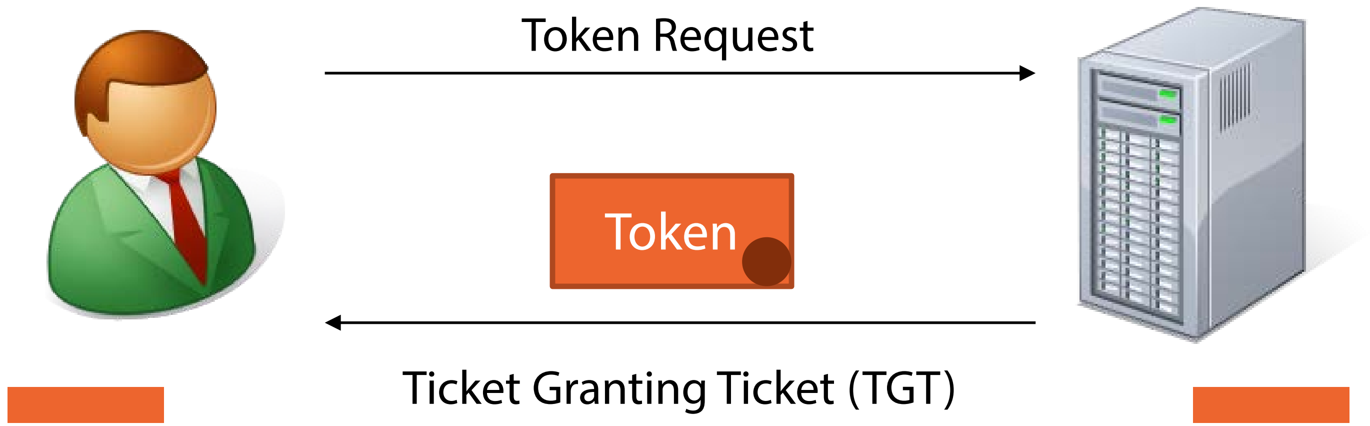
IP-STS

RP-STS

Kerberos

- Both authentication and authorization
- Used in many operating systems, including Windows, OS X, and some Linux distros

Kerberos



WS-Trust and WS-Federation

- **Defines a protocol and XML schemas for SOAP web services to exchange security tokens**
- **Active federation – client machine provides proof of identity**
  - Proof key
  - Client signs a message to prove that he is the holder of a key pair
- **Passive federation – browser redirects exchange tokens through cookies**
  - No proof key
  - Password-based authentication
  - Bearer token signed by STS and encrypted for a specific RP

Secure Assertion Markup Language (SAML)

- **XML**
- **Both authentication and authorization claims (assertions)**
- **Assertions are signed by STS**
- **Enveloped signature**
  - Signature has reference to its assertion, usually by ID

Enveloped Signature

```
<Envelope>
  <Header>
    <Assertion id="valid">
      <Signature>
        <Reference href="#valid" />
      </Signature>
    </Assertion>
  </Header>
  <Body>
    <!-- -->
  </Body>
</Envelope>
```

XML Signature Wrapping Attack

```xml
<Envelope>
  <Header>
    <Assertion id="invalid">
      <Signature>
        <Reference href="#valid" />
      </Signature>
    </Assertion>
  </Header>
  <Body>
    <Assertion id="valid" />
    <!-- -->
  </Body>
</Envelope>
```

Vulnerability

- Validate one assertion
- Use another
- Permutations
- Not all SAML stacks are vulnerable

OAuth

- Social applications
- Mash ups
- Auth stands for "Authorization"
- Delegate access to services
- The agent is authorized, not the user

OAuth

Client Key	Client Secret
hz91aXaKa	DZRWmPn9
mu0pNsng	i6wvllF

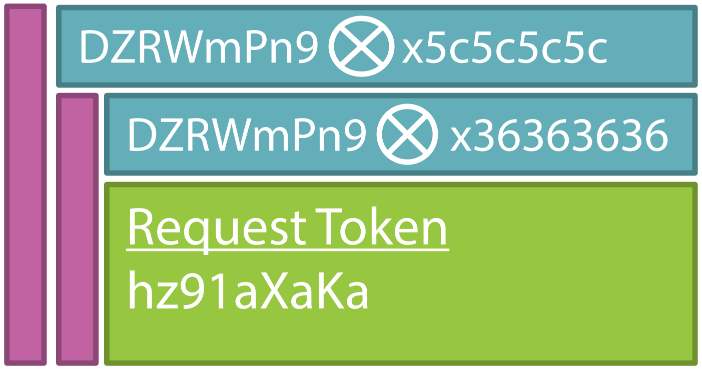
Service Provider

Registration

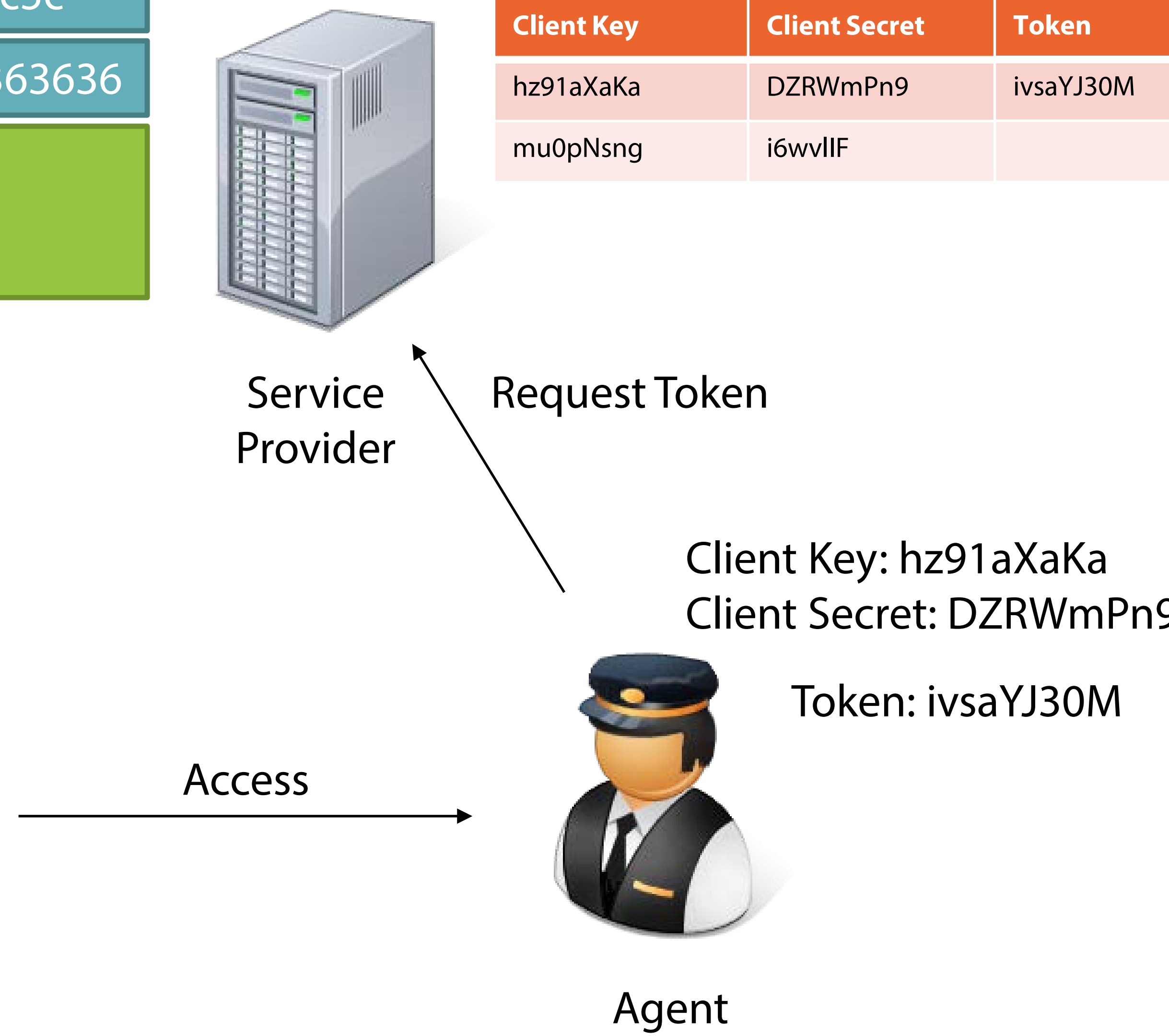
Client Key: hz91aXaKa
Client Secret: DZRWmPn9

Agent

OAuth



Client Key	Client Secret	Token
hz91aXaKa	DZRWmPn9	ivsaYJ30M
mu0pNsng	i6wvlIF	



OAuth

Client Key	Client Secret	Token
hz91aXaKa	DZRWmPn9	ivsaYJ30M
mu0pNsng	i6wvlIF	

Service
Provider

login?token=ivsaYJ30M

Client Key: hz91aXaKa
Client Secret: DZRWmPn9

Token: ivsaYJ30M

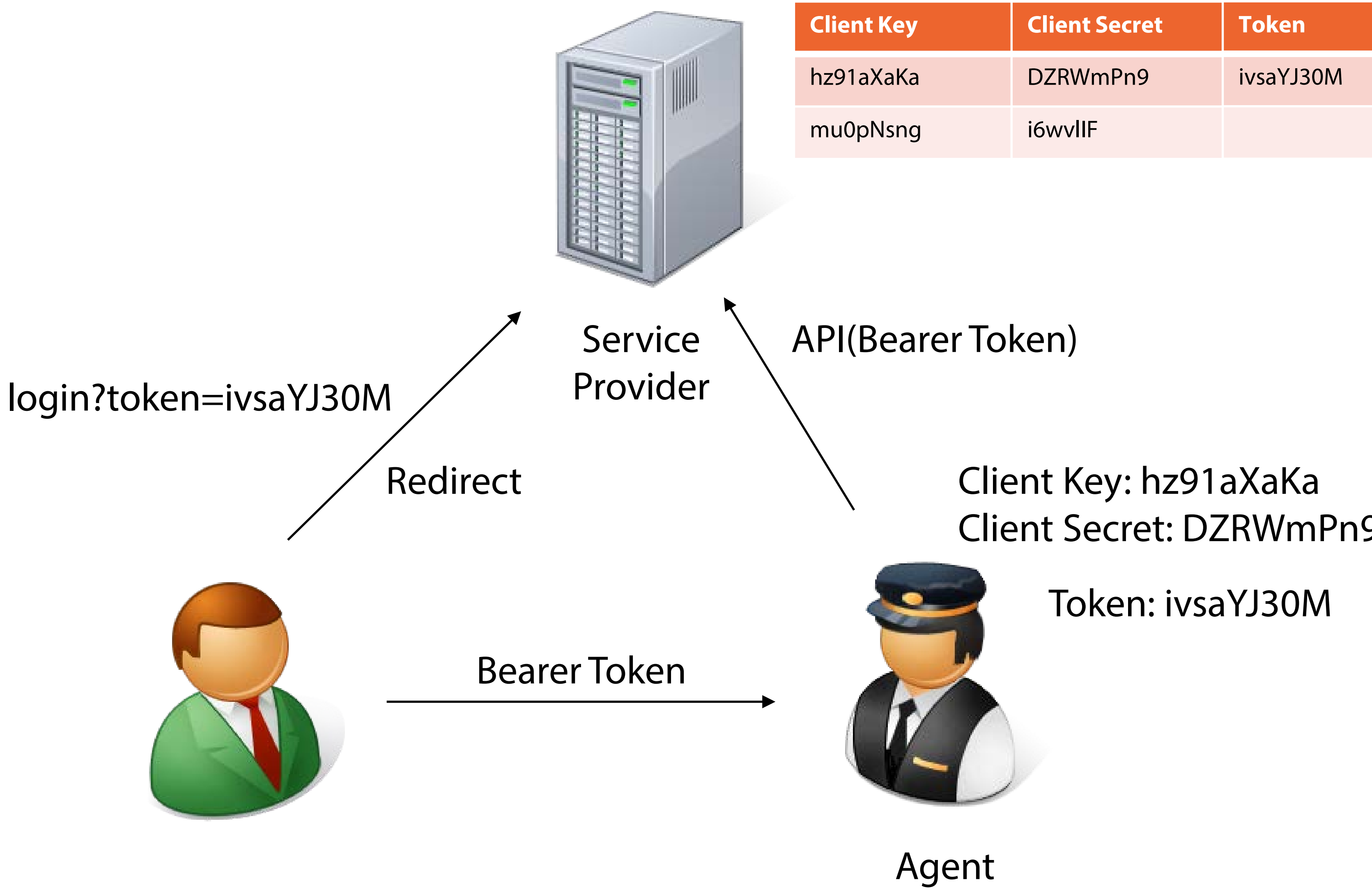
Access

Redirect

Agent

OAuth

Client Key	Client Secret	Token
hz91aXaKa	DZRWmPn9	ivsaYJ30M
mu0pNsng	i6wvlIF	



Cryptography in OAuth

- **Almost non-existent**
- **Token request is signed**
  - Not asymmetric
  - Shared secret

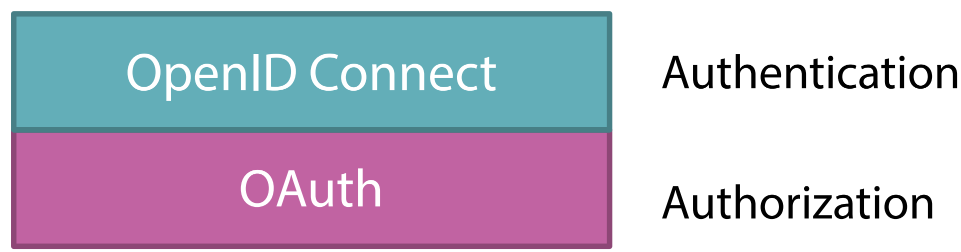
Mobile and Desktop Apps

- **No back end**
  - API calls from client
- **Client secret embedded in mobile app**
- **Can be easily decompiled**
- **No assurance**

OpenID Connect

- **Original OpenID protocol**
  - End user owns identity provider
  - Cumbersome
- **OpenID Connect**
  - Log in using Facebook, Twitter, Google, etc.

Built on OAuth

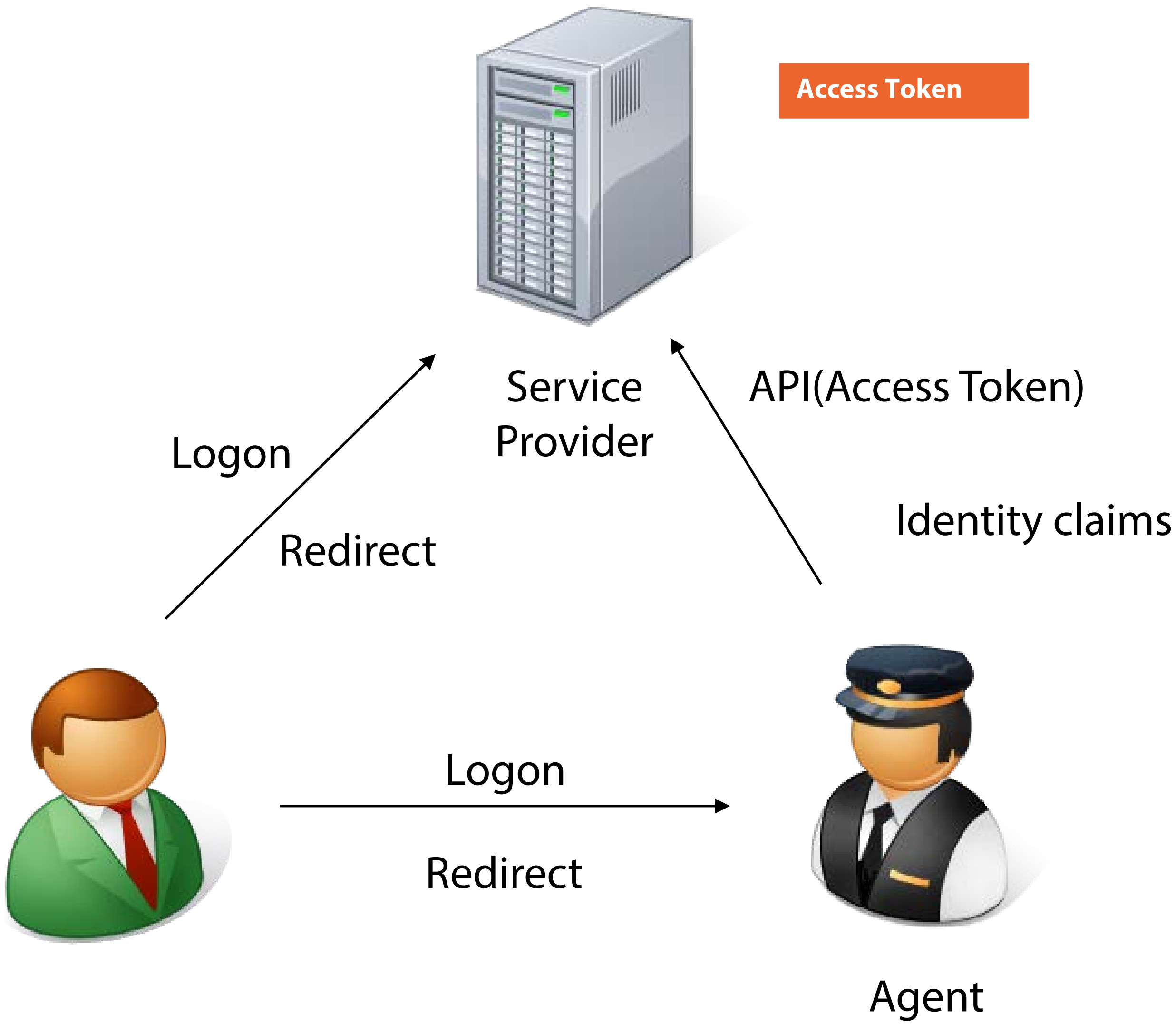


Authorization should *follow* authentication!

What OAuth is Really Doing

- **OAuth does not authorize the user**
  - It authorizes the app
- **OpenID grants authority for the app to know your identity**

OpenID Connect



OAuth and OpenID Connect

- Weakened cryptography
- Some assurance of identity of application
- Bearer token

Authentication and Authorization

- **Passwords**
  - Hash
  - Salt
  - Progressive rehashing
  - Password based key derivation function
- **Sign tokens**
  - Prove veracity of claims
  - Trust relationship
- **Weak cryptography**
  - Bearer tokens
  - Unprotected client secrets