

Distributed Systems

21. Authentication

Paul Krzyzanowski
pxk@cs.rutgers.edu

Security Goals

- **Authentication**
 - Ensure that users, machines, programs, and resources are properly identified
- **Confidentiality**
 - Prevent unauthorized access to data
- **Integrity**
 - Verify that data has not been compromised: deleted, modified, added
- **Availability**
 - Ensure that the system is accessible

Authentication

- Establish & verify identity
- Then decide whether to allow access to resources

Local authentication example: login

```

    graph TD
      A[Get authentication info] --> B[Validate]
      B --> C["setuid(user_id)  
setgid(group_id)"]
      C --> D[exec(login_shell)]
      A --- A1[get login name, password]
      B --- B1[Compare given password with stored password]
      C --- C1["Good? Then change user ID and group ID of process"]
      D --- D1[Replace the login process with the shell process]
      A --- L1[login process uid = root]
      D --- L2[login process uid = user's ID]
    
```

Local authentication example: login

```

    graph TD
      A[Get authentication info] --> B[Validate]
      B --> C["setuid(user_id)  
setgid(group_id)"]
      C --> D[exec(login_shell)]
      A --- A1[get login name, password]
      B --- B1[Compare given password with stored password]
      C --- C1["Good? Then change user ID and group ID of process"]
      D --- D1[Replace the login process with the shell process]
      A --- L1[login process uid = root]
      D --- L2[login process uid = user's ID]
      A --- I[Identification]
      B --- Au[Authentication]
      C --- AC[Access Control]
    
```

Identification vs. Authentication

- **Identification:**
 - Who are you?
 - User name, account number, ...
- **Authentication:**
 - Prove it!
 - Password, PIN, encrypt nonce, ...

Versus Authorization

Authorization defines access control

Once we know a user's identity:

- Allow/disallow request
- Operating system enforces system access based on user's credentials
- Network services usually run in another context
 - Network server may not know of the user
 - Application takes responsibility
- May contact an authorization server
 - Trusted third party that will grant credentials
 - Kerberos ticket granting service
 - RADIUS (centralized authentication/authorization)

Security



The Three A's

- Authentication
- Authorization
- Accounting

Authentication

Three factors:

- something you have *key, card*
 - can be stolen
- something you know *passwords*
 - can be guessed, shared, stolen
- something you are *biometrics*
 - costly, can be copied (sometimes)

Multi-Factor Authentication

Factors may be combined

- ATM machine: 2-factor authentication
 - ATM card *something you have*
 - PIN *something you know*

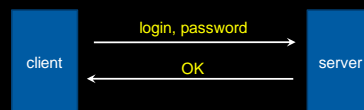
Password Authentication Protocol (PAP)

- Reusable passwords
- Server keeps a database of *username:password* mappings
- Prompt client/user for a login name & password
- To authenticate, use the login name as a key to look up the corresponding password in a database (file) to authenticate

```
if (supplied_password == retrieved_password)
    then user is authenticated
```

Authentication: PAP

Password Authentication Protocol



- Unencrypted, reusable passwords
- Insecure on an open network
- Also, password file must be protected from open access
 - But administrators can still see everyone's passwords

PAP: Reusable passwords

Problem #1: Access to the password file

What if the password file isn't sufficiently protected and an intruder gets hold of it? All passwords are now compromised!

Even if a trusted admin sees your password, this might also be your password on other systems.

Solution:

Store a hash of the password in a file

- given a file, you don't get the passwords
- have to resort to a dictionary or brute-force attack
- Password hashed with SHA-512 hashes (SHA-2)
- Salt can be used to guard against **dictionary attacks**

PAP: Reusable passwords

Problem #2: Network sniffing

Passwords can be stolen by observing a user's session in person or over a network:

- snoop on telnet, ftp, rlogin, rsh sessions
- Trojan horse
- social engineering
- brute-force or dictionary attacks

Solution:

Use **one-time passwords**

One-time passwords

Use a different password each time

- generate a list of passwords
- or:
- use an authentication card

S/key authentication

- One-time password scheme
- Produces a limited number of authentication sessions
- relies on one-way functions

S/key authentication

Authenticate Alice for 100 logins

- pick random number, R
- using a one-way function, $f(x)$:

$$\begin{aligned} X_1 &= f(R) \\ X_2 &= f(X_1) = f(f(R)) \\ X_3 &= f(X_2) = f(f(f(R))) \\ &\dots \\ X_{100} &= f(X_{99}) = f(\dots f(f(f(R)))\dots) \end{aligned}$$

give this list
to Alice

- then compute:

$$x_{101} = f(X_{100}) = f(\dots f(f(f(R)))\dots)$$

S/key authentication

Authenticate Alice for 100 logins

store x_{101} in a password file or database record associated with Alice

alice: x_{101}

S/key authentication

Alice presents the *last* number on her list:

Alice to host: { "alice", x_{100} }

Host computes $f(x_{100})$ and compares it with the value in the database

```

if ( $x_{100}$  provided by alice) = passwd("alice")
  replace  $x_{101}$  in db with  $x_{100}$  provided by alice
  return success
else
  fail
    
```

next time: Alice presents x_{99}
 if someone sees x_{100} there is no way to generate x_{99} .

Two-factor authentication with an authenticator card

Challenge/response authentication

- user is provided with a challenge number from host
- enter challenge number to challenge/response unit
- enter PIN
- get response: $f(\text{PIN}, \text{challenge})$
- transcribe response back to host

- host maintains PIN
 - computes the same function
 - compares data
- rely on **one-way function** (usually a hash function)

Authentication: CHAP

Challenge-Handshake Authentication Protocol

```

graph LR
    Client[client] -- challenge --> Server[server]
    Server -- hash(challenge, secret) --> Client
    Client -- OK --> Server
    
```

The challenge is a *nonce*. We create a hash of the nonce and the secret.

CHAP authentication

an eavesdropper does not see K

Authentication: MS-CHAP

Microsoft's Challenge-Handshake Authentication Protocol

```

graph LR
    Client[client] -- challenge: 16-byte random # --> Server[server]
    Server -- hash(user name, password, password_challenge, challenge) --> Client
    Client -- password_challenge: 16-byte random # --> Server
    Server -- OK --> Client
    
```

The same as CHAP – we're just hashing more things in the response

SecurID card

Something you know (PIN) Something you have (passcode from card)

1. Enter PIN
2. Press \diamond
3. Card computes password
4. Read password & enter

Password: 354982

SecurID card

- from RSA; SASL mechanism: RFC 2808
- two-factor authentication based on:
 - shared secret key (seed)
 - stored on authentication card
 - shared personal ID – PIN
 - known by user

SecurID (SASL) authentication: server side

- Look up user's PIN and seed associated with the token
- Get the time of day
 - Server stores relative accuracy of clock in that SecurID card
 - historic pattern of drift
 - adds or subtracts offset to determine what the clock chip on the SecurID card believes is its current time
- passcode is a cryptographic hash of seed, PIN, and time
 - server computes $f(\text{seed}, \text{PIN}, \text{time})$
- Server compares results with data sent by client

SecurID

- An intruder (sniffing the network) does not have the information to generate the password for future logins
 - Needs the seed number (from the card), the algorithm (inside the card & server), and the PIN (from the user)
- An intruder who steals the card cannot log in
 - Needs a PIN (the benefit of 2-factor authentication)
- An intruder who sees your PIN cannot log in
 - Needs the card (the benefit of 2-factor authentication)
- But...
 - Vulnerable to man-in-the-middle attacks
 - Attacker acts as application server
 - User does not have a chance to authenticate server

Combined authentication and key exchange

Wide-mouth frog

- arbitrated protocol – Trent (3rd party) has all the keys
- symmetric encryption for transmitting a session key

Wide-mouth frog

Trent:

- looks up key corresponding to sender ("alice")
- decrypts remainder of message using Alice's key
- validates timestamp (this is a new message)
- extracts destination ("bob")
- looks up Bob's key

Wide-mouth frog

Alice → Trent → Bob

Message from Alice: "alice", $E_A(T_A, \text{"bob"}, K)$

Message from Trent: $E_B(T_T, \text{"alice"}, K)$

session key
source
time stamp – prevent replay attacks

Trent:

- creates a new message
- new timestamp
- identify source of the session key
- encrypt the message for Bob
- send to Bob

Wide-mouth frog

Alice → Trent → Bob

Message from Alice: "alice", $E_A(T_A, \text{"bob"}, K)$

Message from Trent: $E_B(T_T, \text{"alice"}, K)$

session key
source
time stamp – prevent replay attacks

Bob:

- decrypts message
- validates timestamp
- extracts sender ("alice")
- extracts session key, K

Wide-mouth frog

Alice → Bob

Message: $E_K(M)$

Since Bob and Alice have the session key, they can communicate securely using the key

Kerberos

Kerberos

- Authentication service developed by MIT
 - project Athena 1983-1988
- Trusted third party
- Symmetric cryptography
- Passwords not sent in clear text
 - assumes only the network can be compromised

Kerberos

Users and services authenticate themselves to each other

To access a service:

- user presents a ticket issued by the Kerberos authentication server
- service examines the ticket to verify the identity of the user

Kerberos is a **trusted third party**

- Knows all (users and services) passwords
- Responsible for
 - Authentication: validating an identity
 - Authorization: deciding whether someone can access a service
 - Key exchange: giving both parties an encryption key (securely)

Kerberos

- User *Alice* wants to communicate with a service *Bob*
- Both Alice and Bob have keys
- Step 1:
 - Alice authenticates with Kerberos server
 - Gets *session key* and *sealed envelope*
- Step 2:
 - Alice gives Bob a session key (securely)
 - Convinces Bob that she also got the session key from Kerberos

Authenticate, get permission

"I want to talk to Bob"

if Alice is allowed to talk to Bob, generate session key, S

{ "Bob's server", S }_A

Alice decrypts this:

- gets ID of "Bob's server"
- gets session key
- knows message came from AS

TICKET sealed envelope

{ "Alice", S }_B

eh? (Alice can't read this!)

Send key

Alice encrypts a timestamp with session key

{ "Alice", S }_B, T_S

sealed envelope

Bob decrypts envelope:

- Envelope was created by Kerberos on request from Alice
- Gets session key

Decrypts time stamp

- Validates time window
- Prevent replay attacks

Authenticate recipient

Encrypt Alice's timestamp in return message

{ "Bob's Server", T }_S

Alice validates timestamp

{ Messages }_S

Alice & Bob communicate by encrypting data with S

Kerberos key usage

- Every time a user wants to access a service
 - User's password (key) must be used each time (in decoding message from Kerberos)
- Possible solution:
 - Cache the password (key)
 - Not a good idea
- Another solution:
 - Split Kerberos server into
 - Authentication Server + Ticket Granting Server

Ticket Granting Service (TGS)

TGS + AS = KDC (Kerberos Key Distribution Center)

- Before accessing any service, user requests a ticket to contact the TGS
- Anytime a user wants a service
 - Request a ticket from TGS
 - Reply is encrypted with session key from AS for use with TGS
- TGS works like a temporary ID

Using Kerberos

`$ kinit`
 Password: *enter password*
 ask AS for permission (session key) to access TGS
 Alice gets:

{"TGS", S}_A

{"Alice", S}_{TGS}

Compute key (A) from password to decrypt session key S and get TGS ID.
You now have a ticket to access the Ticket Granting Service

Using Kerberos

`$ rlogin somehost`
 rlogin uses TGS Ticket to request a ticket for the *rlogin* service on *somehost*

Alice sends session key, S, to TGS

rlogin → {"Alice", S}_{TGS}, T_S → TGS

Alice receives session key for rlogin service & ticket to pass to rlogin service

← {"rlogin@somehost", S'_S ← TGS

← {"Alice", S'_R ← TGS

session key for rlogin

ticket for rlogin server on somehost

Radius

RADIUS

- Remote Authentication Dial In User Service
- Centralized access control & authentication for network services
- RADIUS: network service that keeps track of access policies

User

→

Network Service

→

RADIUS server

Resource request(resource, user access credentials)

- Access credentials may be PAP, CHAP, others.
- Shared secret + MD5 hashes used for passwords
- Responses from RADIUS:
 - Accept
 - Challenge (need more info)
 - Reject

Public Key Authentication

Public key authentication

Demonstrate we can encrypt or decrypt a *nonce*

- Alice wants to authenticate herself to Bob:
- Bob: generates nonce, S
 - presents it to Alice
- Alice: encrypts S with her private key (sign it) and send to Bob

A random bunch of bits

Public key authentication

- **Bob:**
look up "alice" in a database of public keys
 - decrypt the message from Alice using Alice's public key
 - If the result is S, then it was Alice!
- Bob is convinced.
- For **mutual authentication**, Alice has to present Bob with a nonce that Bob will encrypt with his private key and return

Public key authentication

- Public key authentication relies on binding identity to a public key
 - How do you know it really is Alice's public key?
- One option:
get keys from a trusted source
- Problem: requires always going to the source
 - cannot pass keys around
- Another option: **sign the public key**
– **digital certificate**

X.509 Certificates

ISO introduced a set of authentication protocols: X.509

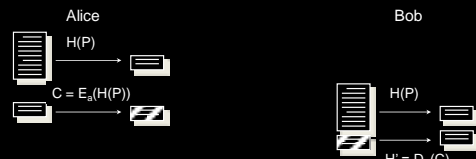
Structure for public key **certificates**:

version	serial #	algorithm, params	issuer	validity time
distinguished name			public key (alg, params, key)	signature of CA

Trusted **Certification Authority** issues a **signed** certificate

Reminder: What's a digital signature?

Hash of a message encrypted with the signer's private key



X.509 certificates

When you get a certificate

- Verify its signature:
 - hash contents of certificate data
 - Decrypt CA's signature with CA's public key

Obtain CA's public key (certificate) from trusted source

- Certification authorities are organized in a hierarchy
- A CA certificate may be signed by a CA above it
 - **certificate chaining**

Certificates prevent someone from using a phony public key to masquerade as another person

... if you trust the CA

List of built-in trusted root certificates in iOS 5

- Prefectura Association For JP/KI, BridgeCA
- Entrust.net Certification Authority (2048)
- A-Trust-Qual-02
- A-Trust-Qual-01
- A-Trust-Qual-03
- AOL Time Warner Root Certification Authority 1
- AOL Time Warner Root Certification Authority 2
- Japanese Government, ApplicationCA
- AddTrust Class 1 CA Root
- AddTrust External CA Root
- AddTrust Public CA Root
- AddTrust Qualified CA Root
- AffirmTrust Premium ECC
- AffirmTrust Premium
- AffirmTrust Networking
- AffirmTrust Commercial
- America Online Root Certification Authority 1
- America Online Root Certification Authority 2
- IGP/NL, Application CA G2
- Apple Root CA
- Apple Root Certificate Authority
- Admin-Root-CA
- Admin-CA-GO-T01
- Baltimore CyberTrust Root
- Buypass Class 2 CA 1
- Buypass Class 3 CA 1
- VeriSign Class 1 Public Primary Certification Authority - G3
- VeriSign Class 2 Public Primary Certification Authority - G3
- VeriSign Class 3 Public Primary Certification Authority - G3
- VeriSign Class 4 Public Primary Certification Authority - G3
- China Internet Network Information Center EV Certificates Root
- COMODO Certification Authority
- Certigna
- Changheva Telecom Co., Ltd., ePKI Root Certification Authority
- VeriSign, Inc. Class 1 Public Primary C1 Certification Authority
- VeriSign, Inc. Class 2 Public Primary Certification Authority
- VeriSign, Inc. Class 3 Public Primary Certification Authority
- VeriSign, Inc. Class 4 Public Primary Certification Authority
- AAA Certificate Services
- Secure Certificate Services
- Trusted Certificate Services
- DST Root CA X4
- Deutsche Telekom Root CA 2
- DigCert Assured ID Root CA
- DigCert Global Root CA
- DigCert High Assurance EV Root CA
- DnD CLASS 3 Root CA
- DnD Root CA 2

<http://support.apple.com/ab/HT5012>

List of built-in trusted root certificates in iOS 5

- EBG Elektronik Sertifika Hizmet
- ECA Root CA
- Echowoor Root CA2
- Entrust Root Certification Authority
- Entrust.net Secure Server Certification Authority
- Equifax, Equifax Secure Certificate Authority
- Equifax Secure Global eBusiness CA-1
- Equifax Secure eBusiness CA11
- Equifax Secure, Equifax Secure eBusiness CA-2
- Jaur-SK
- Common Policy
- Federal Common Policy CA
- Autoridad de Certificacion Fimaprofessional CIF A62634068
- Go Daddy Class 2 Certification Authority
- GTE CyberTrust Global Root
- GeoTrust Global CA
- GlobalSign
- GlobalSign Root CA
- GlobalSign
- Go Daddy Root Certificate Authority - G2
- Hongkong Post Root CA 1
- DST Root CA X3
- DST ACES CA X6
- tizenpe.com
- IZENPE S.A. - CIF A-01337200-RMenc,Vitoria-Gasteiz
- SecureSign RootCA11
- KMD-CA Kvalificeret Person
- KMD-CA Servermail-infoca@kmd-ca.dk
- Japanese Government MPHPT Certification Authority
- NetLock Arany (Class Gold)
- Network Solutions Certificate Authority
- VeriSign, Inc., Class 1 Public Primary Certification Authority
- VeriSign, Inc., Class 2 Public Primary Certification Authority
- VeriSign, Inc., Class 3 Public Primary Certification Authority
- Certum Trusted Network CA
- Chambers of Commerce Root
- Global Chambersign Root
- RSA Security Inc, RSA Security 2048 V3
- Visa eCommerce Root
- SECOM Trust.net, Security Communication RootCA1
- SECOM Trust Systems GO
- Starfield Technologies, Inc. Class 2 Certification Authority
- Sonera Class1 CA
- Sonera Class2 CA
- Starfield Root Certificate Authority - G2
- Starfield Services Root Certificate Authority - G2
- SwissSign Gold CA - G2

<http://support.apple.com/kb/HT5012>

List of built-in trusted root certificates in iOS 5

- SwissSign Platinum CA - G2
- SwissSign Silver CA - G2
- TDC OCES CA
- TDC Internet, TDC Internet Root CA
- Thawte Personal Basic CA
- Thawte Personal Freemail CA
- Thawte Personal Premium CA
- Thawte Premium Server CA
- Thawte Server CA
- Thawte, Inc., Class 3 Public Primary Certification Authority
- Trusts Limited, Trusts FPS Root CA
- Secure Global CA
- SecureTrust CA
- Getbze - Kocastli
- LDA Global Root
- USA Root
- UTM-USERFirst-Client Authentication and Email
- UTM-USERFirst-Hardware
- UTM-USERFirst-Network Applications
- UTM-USERFirst-Object
- UTM - DATACorp SGC
- Certum CA
- VAS Latvijas Pasts SSI(RCA)
- ValCert
- VeriSign Class 3 Public Primary Certification Authority - G5
- VeriSign, Inc., Class 1 Public Primary Certification Authority
- VeriSign, Inc., Class 2 Public Primary Certification Authority
- VeriSign, Inc., Class 3 Public Primary Certification Authority
- OISTE WiSeKey Global Root CA CA
- WellSecure Public Root Certificate Authority
- XRamp Global Certification Authority
- A-CERT ADVANCED
- CertNomis
- AC RatwC3vADz CerticvC3vA1mara S.A.
- Belgium Root CA
- Belgium Root CA2
- Class 2 Primary CA
- Class Root CA 2048
- CNMG ROOT
- CA Disp
- Entrust.net Certification Authority (2048)
- NetLock Expressz (Class C) Tanusihanykiado
- FNMT, FNMT Clase 2 CA
- GeoTrust Primary Certification Authority
- GlobalSign Root CA

<http://support.apple.com/kb/HT5012>

List of built-in trusted root certificates in iOS 5

- Barcelona, IPS Internet publishing Services s.l.
- KISA RootCA 1
- KISA RootCA 3
- NetLock Kozjegyzoi (Class A) Tanusihanykiado
- NetLock Minoskelt Kozjegyzoi (Class GA) Tanusihanykiado
- Thawte Personal Basic CA
- Thawte Personal Freemail CA
- Thawte Personal Premium CA
- VRK Gov. Root CA
- QuoVadis Root Certification Authority
- QuoVadis Root CA 2
- QuoVadis Root CA 3
- Thawte Consulting co, Certification Services Division
- Thawte Consulting co, Certification Services Division
- Staat der Nederlanden Root CA - G2
- Staat der Nederlanden Root CA
- StartCom Certification Authority
- Swisscom Root CA 1
- SwissSign CA (RSA IK May 6 1999 18:00:58)
- TC TrustCenter Universal CA I
- TC TrustCenter Universal CA II
- TC TrustCenter Class 2 CA II
- TC TrustCenter Class 3 CA II
- TC TrustCenter Class 4 CA II
- thawte, Inc., Certification Services Division
- thawte Primary Root CA - G2
- TC TrustCenter Universal CA III
- ORKTRUST Elektronik Sertifika Hizmet
- TWCA Root Certification Authority
- NetLock Uzleti (Class B) Tanusihanykiado
- Wells Fargo Root Certificate Authority

<http://support.apple.com/kb/HT5012>

TLS

Transport Layer Security (TLS)

- aka **Secure Sockets Layer**
- Sits on top of TCP/IP
- Goal: provide an encrypted and possibly authenticated communication channel
 - Provides authentication via RSA and X.509 certificates
 - Encryption of communication session via a symmetric cipher
- **Hybrid cryptosystem:** (usually, but also supports Diffie-Hellman)
 - Public key for authentication
 - Symmetric for data communication
- Enables TCP services to engage in secure, authenticated transfers
 - http, telnet, ntp, ftp, smtp, ...

Transport Layer Security (TLS)

```

client → server: hello(version, random#, cipher suites)
server → client: hello(chosen version, random #, chosen cipher suites, session ID)
client → server: certificate (or public key)
client → server: hello done
    
```

1. Establish protocol, version, cipher suite, compression mechanism, get server certificate (or public key) [details depend on chosen cipher]

Transport Layer Security (TLS)

client → server

$E(\text{hash}(\text{messages}))$

encrypt with client's private key

2. Verify client (OPTIONAL)
 - a. Client hashes previous handshake messages
 - b. Encrypts with client's private key and sends to server
 - c. Server decrypts with client's public key and compares hashes

Transport Layer Security (TLS)

client → server

$E(\text{PreMasterSecret})$

encrypt with server's public key

Master Secret = $f(\text{PreMasterSecret}, \text{random numbers})$

3. Establish a shared key
 - a. Client generates random #: PreMasterSecret
 - b. Encrypts with server's public key and sends to server
 - c. Both sides generate the session key (Master Secret)

Transport Layer Security (TLS)

client → server

ChangeCipherSpec

Everything from the client will now be encrypted & authenticated

Finished: $E(\text{hash}(\text{messages}))$

Server decrypts (Master Secret) & validates client's encrypted hash

Server validates that the client has the key

4. Authenticate client (validate knowledge of the key)
 - a. Client hashes previous handshake messages
 - b. Encrypts with Master Secret & sends to server
 - c. Server hashes previous handshake messages
 - d. Decrypts client's message & compares hashes

Transport Layer Security (TLS)

server → client

ChangeCipherSpec

Everything from the client will now be encrypted & authenticated

Finished: $E(\text{hash}(\text{messages}))$

Client decrypts (Master Secret) & validates client's encrypted hash

Client validates that the client has the key

5. Authenticate server (validate knowledge of the key)
 - a. Server hashes previous handshake messages
 - b. Encrypts with Master Secret & sends to client
 - c. Decrypts client's message & compares hashes

Transport Layer Security (TLS)

client ↔ server

$E_s(\text{data})$

encrypt and decrypt with Master Secret and symmetric algorithm (e.g. RC4)

6. Exchange data (symmetric encryption)
 - Authenticated with hash (optional)
 - Encrypted (optional)

Transport Layer Security (TLS)

- Optimizing reconections: *abbreviated handshake*
 - Server sends a session ID during initial *hello* message
 - Client & server save negotiated parameters and *master secret*
 - Client can use the session ID when reconnecting
 - Clients and servers

OpenID

OpenID

- Designed to solve the problem of
 - Having to get an ID per website
 - Managing passwords per site
- Decentralized mechanism for single sign on
 - Access different services (sites) using the same identity
 - User chooses which OpenID provider to use
 - OpenID does not specify authentication form – up to provider
 - Web site never sees your password

OpenID: Step 1. Create an ID

- Choose an OpenID provider
 - E.g., pip.verisignlabs.com
- Create an account
 - Create your user name and password
 - Possibly choose authentication method

OpenID: Step 2. Visit a service

- Relying party = OpenID-enabled web site
- Instead of asking for a user name, password, the site asks for an OpenID name
 - E.g., pxk.pip.verisignlabs.com
- Web site
 - Converts OpenID Identity Provider to a URL (pip.verisignlabs.com)
 - Contacts the URL to get the real URL of the identity provider
 - Redirect the user's browser to the provider's site

OpenID: Step 3. Authenticate

- User sees the OpenID provider's authentication page
- Provider will ask for authentication info (typically password)
 - Choose duration: authenticate forever, limited time, one-time

OpenID: Step 4. Return to service

- If authentication is successful AND user accepts the request:
- Browser is redirected to a return page on the original (relying party's) site
- Redirect contains
 - User's credentials
 - Digital signature by OpenID Identity Provider

OpenID: Step 5. Service validates login

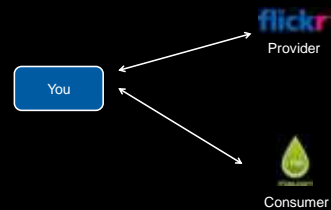
- Relying party needs to check validity of user's identity
- Dumb mode
 - Establishes secure session with Identity Provider
 - Requests authentication info about the user to compare it with the assertion received via the browser redirect.
 - If they match then the party is convinced that the user authenticated successfully with the provider
- Smart mode
 - In the initial redirect, the relying party sent an association request to the Identity Provider and established a shared secret key
 - Use secret key to decode the signature in the message
 - If the signature matches then, the party is convinced

OAuth

OAuth: Open Authorization

- Like OpenID in a couple of ways
 - Decentralized & open standard
 - Relies on browser redirects
- OpenID: framework for remote authentication
 - Have one login for multiple web sites
- OAuth: framework for service authorization
 - Allows you to authorize one website (consumer) to access data from another website (provider) – *in a controlled manner*
 - Designed for web services
 - Example: allow the Moo photo printing service to get your photos from your Flickr account
 - OpenID can still be used for logging into the sites

OAuth Entities



OAuth Controls

- Authorization
 - What parts of a site can be accessed?
E.g., download photos but not delete
- Authentication
 - Allow a service to log in to access the data
 - Logged-in user has a token to access data

Register a consumer application

- Service provider (e.g., Flickr):
 - Gets data about your application (name, creator, URL)
 - Assigns consumer a key & secret
 - Provides documentation of authorization URLs and methods

How does authorization take place?

- Consumer needs a *Request Token* from the Provider (e.g., moo.com needs a *request token* from flickr.com)
 - Consumer redirects user to service provider
 - User authenticates (e.g., via OpenID)
 - User authorizes requested access
 - Service Provider redirects back to consumer
 - Consumer now has an authorized *Request Token*
 - Consumer exchanges *Request Token* for *Access Token*
 - Message sent to service provider
 - Service Provider returns Access Token
 - Consumer makes requests from Provider using the Access Token

Key Points



- You still need to log into the Provider's OAuth service when redirected (or use OpenID)
- You specify approve the specific access that you are granting
- The Provider validates the requested access when it gets a token from the Consumer

Cryptographic toolbox

- Symmetric encryption
- Public key encryption
- One-way hash functions
- Random number generators
 - Used for nonces and session keys

Examples

- **Key exchange**
 - Public key cryptography
- **Key exchange + secure communication**
 - Random # + Public key + symmetric cryptography
- **Authentication**
 - Nonce (random #) + encryption
- **Message authentication codes**
 - Hashes
- **Digital signature**
 - Hash + encryption

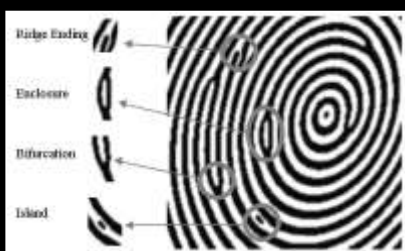
Biometric authentication

Biometrics

- Statistical pattern recognition
 - Thresholds
- Each biometric system has a characteristic ROC curve
 - (receiver operator characteristic, a legacy from radio electronics)

Biometrics: forms

- Fingerprints
 - identify minutia



The diagram shows a fingerprint pattern with four types of minutia labeled on the left: Ridge Ending, Enclosure, Bifurcation, and Island. Arrows point from these labels to corresponding features in the fingerprint image on the right.

Biometrics: forms

- Iris
 - Analyze pattern of spokes: excellent uniqueness, signal can be normalized for fast matching
- Retina scan
 - Excellent uniqueness but not popular for non-criminals
- Fingerprint
 - Reasonable uniqueness
- Hand geometry
 - Low guarantee of uniqueness: generally need 1:1 match
- Signature, Voice
 - Behavioral vs. physical system
 - Can change with demeanor, tend to have low recognition rates
- Facial geometry

Biometrics: desirable characteristics

- Robustness
 - Repeatable, not subject to large changes over time
 - Fingerprints & iris patterns are more robust than voice
- Distinctiveness
 - Differences in the pattern among population
 - Fingerprints: typically 40-60 distinct features
 - Iris: typically >250 distinct features
 - Hand geometry: ~1 in 100 people may have a hand with measurements close to yours.

Biometrics: desirable characteristics

Biometric	Robustness	Distinctiveness
Fingerprint	Moderate	High
Hand Geometry	Moderate	Low
Voice	Moderate	Low
Iris	High	High
Signature	Low	Moderate

Irises vs. Fingerprints

- Number of features measured:
 - High-end fingerprint systems: ~40-60 features
 - Iris systems: ~240 features
- Ease of data capture
 - More difficult to damage an iris
 - Feature capture more difficult for fingerprints:
 - Smudges, gloves, dryness, ...

Irises vs. Fingerprints

- False accept rates (FAR)
 - Fingerprints: ~ 1:100,000 (varies by vendor)
 - Iris: ~ 1:1.2 million
- Ease of searching
 - Fingerprints cannot be normalized
 - 1:many searches are difficult
 - Iris can be normalized to generate a unique IrisCode
 - 1:many searches much faster

Biometrics: desirable characteristics

- **Cooperative systems** (multi-factor)
 - User provides identity, such as name and/or PIN
- **Non-cooperative**
 - Users cannot be relied on to identify themselves
 - Need to search large portion of database
- **Overt vs. covert** identification
- **Habituated vs. non-habituated**
 - Do users regularly use (train) the system

Identification vs. Verification

- **Identification:** Who is this?
 - 1:many search
- **Verification:** Is this X?
 - Present a name, PIN, token
 - 1:1 (or 1:small #) search

Biometric: authentication process

0. Enrollment

- The user's entry in a database of biometric signals must be populated.
- Initial sensing and feature extraction
- May be repeated to ensure good feature extraction

Biometric: authentication process

1. Sensing

- User's characteristic must be presented to a sensor
- Output is a function of:
 - Biometric measure
 - The way it is presented
 - Technical characteristics of sensor

2. Signal Processing

- Feature extraction
- Extract the desired biometric pattern
 - remove noise and signal losses
 - discard qualities that are not distinctive/repeatable
 - Determine if feature is of "good quality"

Biometric: authentication process

3. Pattern matching

- Sample compared to original signal in database
- Closely matched patterns have "small distances" between them
- Distances will hardly ever be 0 (perfect match)

4. Decisions

- Decide if the match is close enough
- Trade-off:
 - ↓ false non-matches leads to ↑false matches

Detecting Humanness

Gestalt Psychology (1922-1923)

- Max Wertheimer, Kurt Koffka
- Laws of organization
 - Proximity
 - We tend to group things together that are close together in space
 - Similarity
 - We tend to group things together that are similar
 - Good Continuation
 - We tend to perceive things in good form
 - Closure
 - We tend to make our experience as complete as possible
 - Figure and Ground
 - We tend to organize our perceptions by distinguishing between a figure and a background

Gestalt Psychology



Gestalt Psychology

HELLO

Gestalt Psychology

HELLO


Authenticating humanness

- Battle the Bots
 - Create a test that is easy for humans but extremely difficult for computers
- CAPTCHA
 - Completely Automated Public Turing test to tell Computers and Humans Apart
 - Image Degradation
 - Exploit our limits in OCR technology
 - Leverages human Gestalt psychology: reconstruction
 - 2000: Yahoo! and Manuel Blum & team at CMU
 - EZ-Gimpy: one of 850 words
 - Henry Baird @ CMU & Monica Chew at UCB
 - BaffleText: generates a few words + random non-English words

Source: http://www.sitam.com/print_version.cfm?articleID=00053EA7-86E9-3F80-85758341487F0103
<http://tinyurl.com/dqz2f>

CAPTCHA





The End