

Operating Systems Design

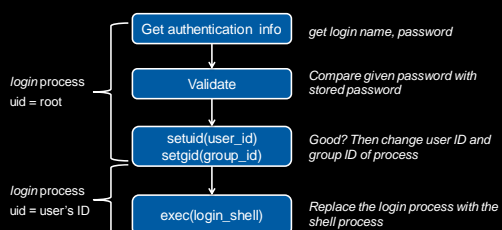
19. Authentication

Paul Krzyzanowski
pxk@cs.rutgers.edu

Authentication

- For a user/process:
 - Establish & verify identity
 - Then decide whether to allow access to resources
- For a file or data stream:
 - Validate that the integrity of the data; that it has not been modified by anyone other than the author
 - E.g., digital signature

Authentication example: login



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)

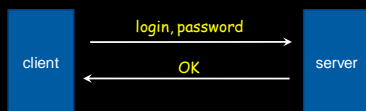
- Reusable passwords
- Server keeps a database of *username:password* mappings
- Prompt client/user for a login name & password
- To authenticate, use the login name as a key to look up the corresponding password in a database (file) to authenticate

```

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

Authentication: PAP

Password Authentication Protocol



- Unencrypted, reusable passwords
- Insecure on an open network
- Password file must be protected from access

PAP: Reusable passwords

One problem: what if the password file isn't sufficiently protected and an intruder gets hold of it, he gets all the passwords!

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

Enhancement:

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
- Unix approach
 - Password encrypted with 3DES hashes; then MD5 hashes; now SHA512 hashes
 - Salt used to guard against **dictionary attacks**

PAP: Reusable passwords

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

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, $f(x)$:

$$\begin{aligned}
 x_1 &= f(R) \\
 x_2 &= f(x_1) = f(f(R)) \\
 x_3 &= f(x_2) = f(f(f(R))) \\
 &\dots \\
 x_{100} &= f(x_{99}) = f(\dots f(f(f(R)))\dots)
 \end{aligned}$$

- 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

Challenge-Response authentication

	network	
Alice		host
"alice"	→ "alice" →	look up alice's key, K
		generate random challenge number C
$R' = f(K, C)$	← C ←	$R = f(K, C)$
		$R = R' ?$
	← R' ←	
	← "welcome" ←	
	⏟	
	an eavesdropper does not see K	

Authentication: CHAP

Challenge-Handshake Authentication Protocol

```

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

Has shared secret Has shared secret

The challenge-response scheme in a slightly different form.

The challenge is a *nonce*. Instead of encrypting the nonce with a shared secret key, we create a hash of the nonce and the secret.

Authentication: MS-CHAP

Microsoft's Challenge-Handshake Authentication Protocol


```

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

Has user's password Has user's password

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

SecurID card



Username: paul

Password: 1234032848

PIN passcode from card

Something you know Something you have

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
 - DB 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)
- 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

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 deciding whether someone can access a service
 - **Authorization**

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

eh? (Alice can't read this!)

← {"Alice", S}_S →

TICKET
sealed envelope

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

← {"Bob's Server", T}_S →

Encrypt Alice's timestamp in return message

Alice validates timestamp

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
- 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

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

Transport Layer Security (TLS)

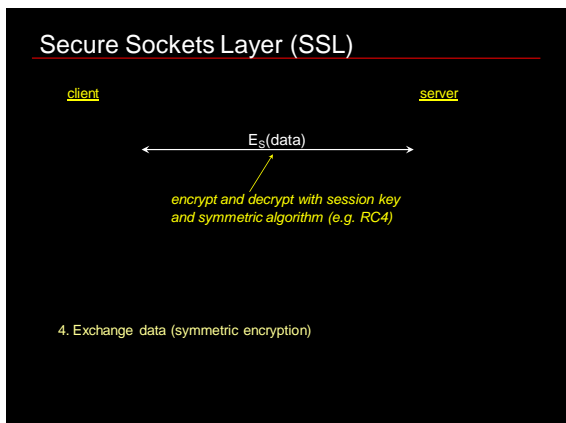
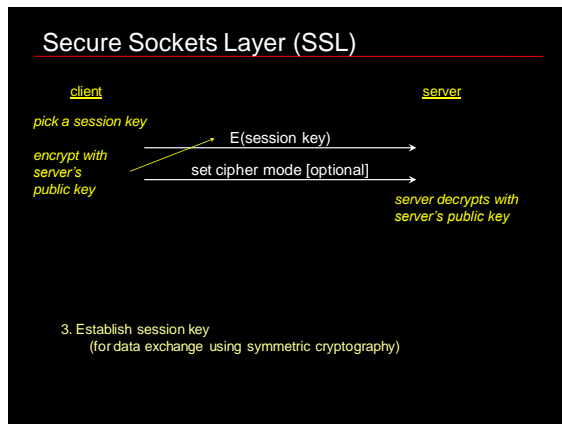
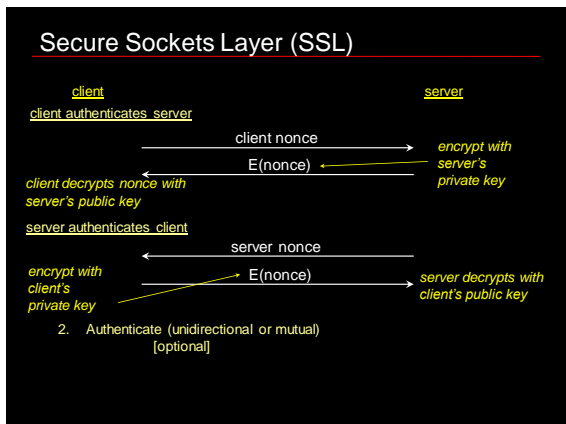
- **aka Secure Sockets Layer**
- Sits on top of TCP/IP
- Goal: provide an encrypted and possibly authenticated communication channel
 - Provides authentication via RSA and X.509 certificates
 - Encryption of communication session via a symmetric cipher
- **Hybrid cryptosystem:**
 - Public key for authentication;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

Secure Sockets Layer (SSL)

```

sequenceDiagram
    participant client
    participant server
    client->>server: hello(version, protocol)
    server-->>client: hello(version, protocol)
    server-->>client: certificate (or public key)
    client-->>server: hello done
    server-->>client: certificate (or none)
    
```

1. Establish protocol, version, cipher suite, compression mechanism, exchange certificates (or send public key)



- ### 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
 - Public key + symmetric cryptography
 - Authentication
 - Nonce + encryption
 - Message authentication codes
 - Hashes
 - Digital signature
 - Hash + encryption

Security approaches in the OS (more)

Defense from malicious software

- **Access privileges**
 - Don't run as administrator
 - Warning: network services don't run with the privileges of the user requesting them – they are extra vulnerable
- **Signed software**
 - Validate the integrity of the software you install
 - Optionally, validate when running it
- **Personal firewall**
 - Intercept & explicitly allow/deny applications access to the network
 - Personal firewalls are application-aware

Code Integrity: signed software

- **Per-page signatures**
 - Check hashes for every page upon loading (demand paging)
 - OS X & Vista/Windows 7:
 - OS X: `codesign` command
 - Windows 7: `signwizard` GUI
 - XP/Vista/Windows 7: Microsoft Authenticode
 - Hashes stored in system catalog or signed & embedded in the file
 - OS X
 - Hashes & certificate chain stored in file

Code signing: Microsoft Authenticode

- A format for signing executable code (dll, exe, cab, ocx, class files)
- **Software publisher:**
 - Generate a public/private key pair
 - Get a digital certificate: VeriSign class 3 Commercial Software Publisher's certificate
 - Generate a hash of the code to create a fixed-length digest
 - Encrypt the hash with your private key
 - Combine digest & certificate into a Signature Block
 - Embed Signature Block in executable
- **Recipient:**
 - Call `WinVerifyTrust` function to validate:
 - Validate certificate, decrypt digest, compare with hash of downloaded code

Vista/Win 7 code integrity checks


- Implemented as a file system driver
 - Works with demand paging from executable
 - Check hashes for every page as the page is loaded
- Hashes in system catalog or embedded in file along with X.509 certificate.
- Check integrity of boot process
 - Kernel code must be signed or it won't load
 - Drivers shipped with Windows must be certified or contain a certificate from Microsoft

Sandboxing

- Restrict what code is allowed to do
 - Lets you work with software you don't fully trust
- Java Virtual Machine
 1. **Bytecode verifier:** verifies Java bytecode before it is run
 - Disallow pointer arithmetic
 - Automatic garbage collection
 - Array bounds checking
 - Null reference checking
 2. **Class loader:** determines if an object is allowed to add classes
 - Ensures key parts of the runtime environment are not overwritten
 - Runtime data areas (stacks, bytecodes, heap) are randomly laid out
 3. **Security manager**
 - Defines the boundaries of the sandbox (file, net, native, etc. access)
 - Consulted before any access to a resource is allowed

Dealing with application security

- **Isolation & memory safety**
 - Rely on operating system; MMU, address space layout randomization
 - Compiler for stack canaries
- **Code auditing**
 - If possible: but need access to code & skilled staff
- **Access control checking at interfaces**
 - E.g., Java security manager
- **Code signing**
 - E.g., Authenticode
- **Runtime, load-time code verification**
 - Sandboxing: Java bytecode verifier, class loader
 - Microsoft CLR



The End