Authentication

Shankar

April 11, 2017

Outline

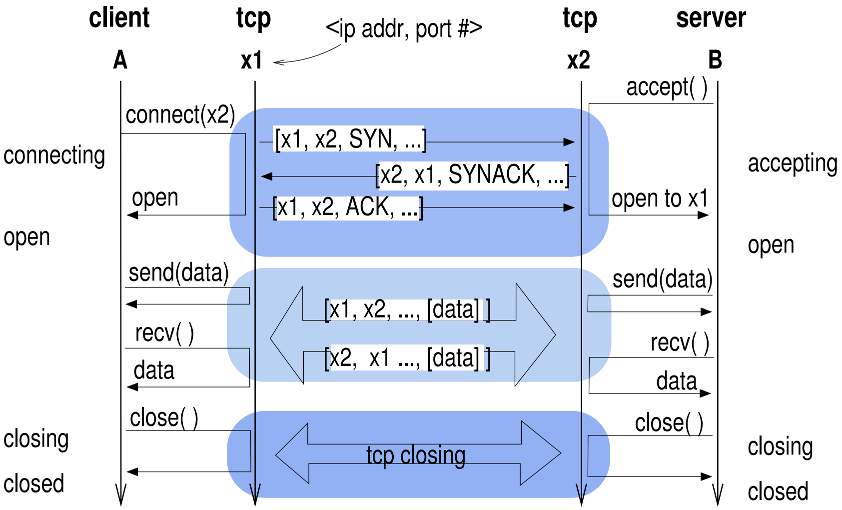
Authentication framework

- Principals are application clients and servers interacting over TCP/UDP in an insecure network (eg, Internet)
- Principals establish sessions, exchange data, close sessions

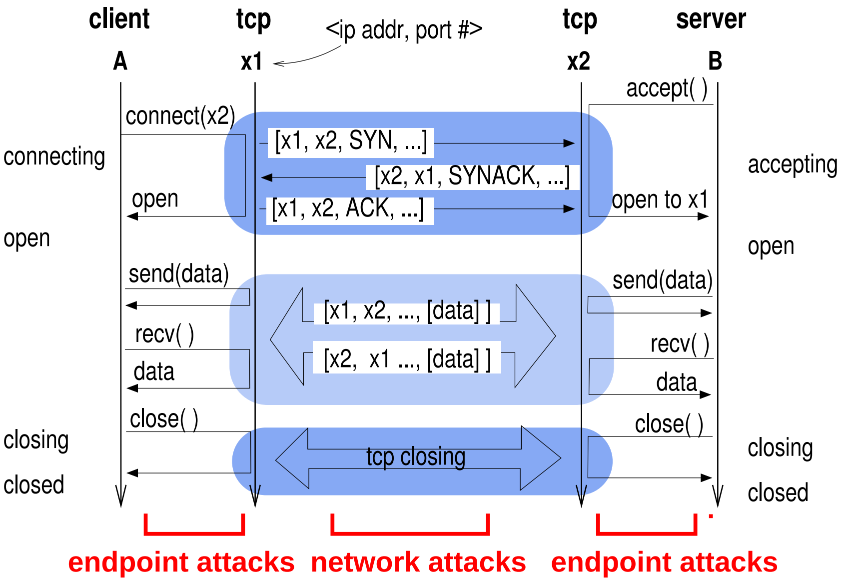
- Attacks
  - Network attacks: listen, intercept msgs, resend modifed msgs
  - Endpoint: malicious/compromised user

- Authentication goals:
  - Ensure that session peers are who they say they are
  - Establish session key(s) for data confidentiality/integrity
    - better to use temporary keys than long-term keys

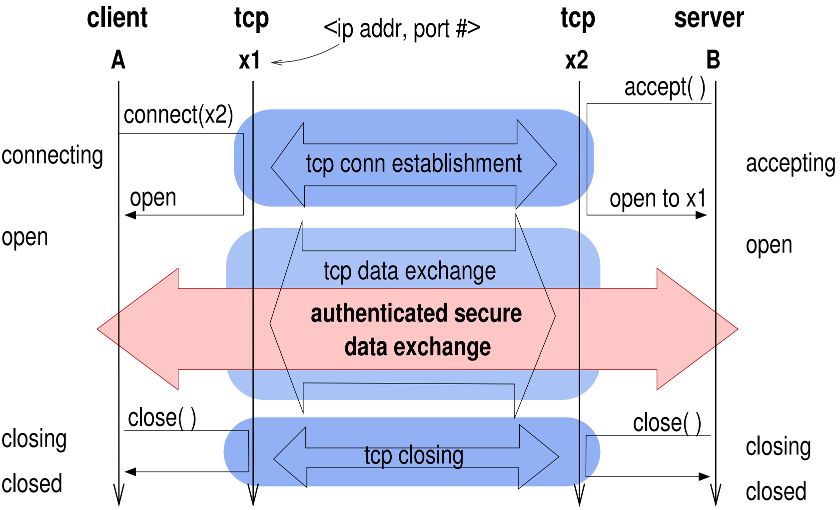
TCP-based session: without authentication



TCP-based session: attacks



TCP-based session: with authentication



Dictionary (aka password-guessing) attacks

- Weak secret (aka low-quality secret)
  - comes from a space small enough for a brute-force search
  - eg: passwords, and keys obtained from them
- Strong secret (aka high-quality secret): not weak
  - eg: key with 128 random bits
- Dictionary attacks (aka password-guessing attacks)
  - Given ciphertext from structured plaintext and weak key, decrypt with every possible key until structure appears
  - Online attack: interact with authenticator at every guess
    - Defense: limit number/frequency of attempts
  - Offline attack: interact with authenticator just once
    - Defense: don't expose relevant ciphertext

Conventions: Crypto

- Symmetric crypto
  - $E(key, msg)$: encrypt *msg* with *key* // includes any IV
  - $D(key, ctx)$: decrypt *ctx* with *key* // includes any IV
- Hash
  - $H(msg)$: hash of *msg* // eg, SHA-1
  - $H(key, msg)$: keyed-hash // eg, HMAC-SHA-1
- Asymmetric crypto // public-key pair $[sk, pk]$
  - $E_P(pk, msg)$: encrypt *msg* (with public key)
  - $D_P(sk, msg)$: decrypt *msg* (with secret key)
  - $Sgn(sk, msg)$: signature of *msg* (using secret key)
  - $Vfy(pk, msg, s)$: verify signature *s* of *msg* (with public key)

Conventions: Nonces

- Nonce: new value // new = never before seen
  - Can be predictable or random
  - Predictable: given one value, attacker can guess the next one
  - Random: not predictable // physical randomness, crypto output

Outline

A, B share symmetric key k

client A (key k for server B)	server B (has key k for user A)
send $[A, B, \text{conn}]$	
	rcv $[A, B, \text{conn}]$
	$c_B \leftarrow$ random // server challenge
	send $[B, A, c_B]$
rcv $[B, A, c_B]$	
$c_A \leftarrow$ random // client challenge	
$r_B \leftarrow E(k, c_B)$ // client response	
send $[c_A, r_B]$	rcv $[c_A, r_B]$
	if $(r_B \neq E(k, c_B))$ FAIL
	$r_A \leftarrow E(k, c_A)$ // server response
	send $[r_A]$
rcv $[r_A]$	session key $\leftarrow Func(c_A, c_B, k)$
if $(r_A \neq E(k, c_A))$ FAIL	
session key $\leftarrow Func(c_A, c_B, k)$	

A, B share symmetric key k (cont)

- Many variations of challenge/response
  - open challenge, encrypted response // $c_A \rightarrow E(k, c_A)$
  - encrypted challenge and response // $E(k, c_A) \rightarrow E(k, c_A + 1)$
- Offline dictionary attack if k is weak and
  - attacker can eavesdrop, or
  - attacker can attach to B's net address
- If client issues challenge first and k is weak,
  can do offline dictionary attack without attacking network
  - attacker sends challenge, gets response

A, B have each other's public key

client A (has $[sk_A, pk_A]$, pk_B)	server B (has $[sk_B, pk_B]$, pk_A)
$c_A \leftarrow$ random // challenge	
send $[A, B, \text{conn}, E_P(pk_B, c_A)]$	rcv $[A, B, \text{conn}, y_A]$
	$c_A \leftarrow D_P(sk_B, y_A)$
	$c_B \leftarrow$ random // challenge
rcv $[B, A, y_B]$	send $[B, A, E_P(pk_A, [c_B, c_A])]$ // resp
$[c_B, r_A] \leftarrow D_P(sk_A, y_B)$	
if $(r_A \neq c_A)$ FAIL	
send $[E_P(pk_B, c_B)]$ // response	
session key $\leftarrow Func(c_A, c_B)$	rcv $[y_B]$
	$r_B \leftarrow D_P(sk_B, y_B)$
	if $(r_B \neq c_B)$ FAIL
	session key $\leftarrow Func(c_A, c_B)$

- Safe from dictionary attack // asymmetric keys always strong

A has pk_B; A, B share symm key k

client A (has k, pk_B)	server B (has $[sk_B, pk_B]$, k)
$c_A \leftarrow$ random // challenge	
send $[A, B, \text{conn}, E_P(pk_B, c_A)]$	rcv $[A, B, \text{conn}, y_A]$
	$r_A \leftarrow D_P(sk_B, y_A)$ // response
	$c_B \leftarrow$ random // challenge
rcv $[B, A, c_B, r_A]$	send $[B, A, c_B, r_A)]$ // plaintext
if $(r_A \neq c_A)$ FAIL	
$r_B \leftarrow E(k, c_B)$ // response	
send $[E_P(pk_B, r_B)]$	
session key $\leftarrow Func(c_A, c_B, k)$	rcv $[y_B]$
	$r_B \leftarrow D_P(sk_B, y_B)$
	if $(D(k, r_B) \neq c_B)$ FAIL
	session key $\leftarrow Func(c_A, c_B, k)$

- Warning: the above session key is weak if k is weak // Why?
- Better to use DH to get a strong session key

Authenticated Diffie-Helman

- Authenticated DH: incorporate a pre-shared key into DH
- If A and B share a symmetric key k, here are two ways
  1. Encrypt DH public keys with k
     - A sends $E(k,\ g^{S_A} \text{ mod-}p)$
     - B sends $E(k,\ g^{S_B} \text{ mod-}p)$
     - shared key: $g^{S_A \cdot S_B} \text{ mod-}p$
  2. Do usual DH, then exchange *keyed*-hashes of DH key.
- Secure against dictionary attack even if k is weak!
- If A and B have each other's public key, here are two ways
  1. Encrypt DH quantities with receiver's public key
  2. Sign DH quantities with sender's private key

Session Keys

- Should differ from long-term key used for authentication
  - to avoid long-term key "wearing out" (offline crypto attack)
- Should be forgotten after session ends
- Should be unique for each session
  - if compromised, only affects data sent in that session
  - can give to untrusted software // delegation

- Delegation
  - A, B share key k
  - A wants C to access B on A's behalf

- Two solutions to delegation
  1. A gives C the shared key k // terrible!
  2. A gives C a ticket: $E(k,\ [\textit{allowed ops}, \textit{expiry time},\ \ldots])$

Outline

Identifying a human

- What we know
  - password, date-of-birth, address, etc
  - Cons: exposed when used
- What physical object we hold
  - badges, keys, smart card (with strong crypto)
  - Cons: object must be difficult to forge, tamper, reverse engineer
- What physical property we have (biometrics)
  - fingerprint, face, iris
  - Cons: not hard to forge
- Others: where we are, how we react,, where we travel, etc
  - Cons: easy to forge
- Typically use a combination of methods
  - eg: password, browser fingerprint, location, ...

Passwords

- Setting a password
  - A chooses a password that is hard to guess // how hard?
  - A shares it securely with B, which stores it
- Logging in
  - A provides B the password; B checks it
  - A is authenticated iff match
  - If no match: B may delay next login attempt to A
- Recovering a forgotten password
  - Falling back to some other form authentication
    - pre-specified email or phone
    - visit office with physical id

What do strong passwords get us

- Strong password
  - Hard to guess; includes symbols, mixed case, etc
  - Dictionary attack doable, but more work than weak pwd
- Online dictionary attack
  - Defense: limit on number of wrong logins
  - Targted victim: strong pwd doesn't help
  - Any victim (stop at first success): strong pwd helps
- Offline dictionary attack
  - Targeted victim: strong pwd doesn't help (unless very strong)
  - Any victim: strong pwd helps (if many others have weak pwds)

Outline

Storing passwords at server

- Assume an attacker that has access to server filesystem
- Attempt 1: store [usr, pwd] pairs in plaintext file: worthless
- Attempt 2: store [usr, pwd] pairs in encrypted file
  - worthless if encryption key is also in filesystem
- Attempt 3: store hashes of passwords
  - store [usr, h] pairs in plaintext file, where $h = H(pwd)$
  - when A logs in with pwd, check if $H(pwd) = h$
  - Good: pwd is never in filesystem, only briefly in memory
  - Bad: vulnerable to dictionary attack
    - attacker precomputes $\{H(p_i)\}$ for candidate pwds $p_1, p_2, \ldots$
    - checks each $H(p_i)$ against the h's of all users

Storing passwords at server (cont)

- Attempt 4: store hashes of salted passwords
  - salt is a random nonce, different for each user
  - store $[usr, salt, h]$ triples, where $h = H(salt \| pwd)$
  - when A logs in with pwd, check if $H(salt \| pwd) = h$
  - Dictionary attack still doable but more work
    - candidate hashes $\{H(p_i)\}$ cannot be precomputed
    - each candidate hash applies only to one user

- Attempt 5: store k-fold hashes of salted passwords
  - store $[usr, salt, h]$ triples, where $h = H^k(salt \| pwd)$
    - $H^k(x) \; = \; H(H(\cdots H(x) \cdots))$ k times // slow hash
  - Dictionary attack still doable but work increases k times

Outline

Scaling to N users

- Naive approach using symmetric keys
  - Every principal shares a key with every other principal
  - Not scalable
    - N^2 storage at each principal
    - N cost for adding/removing principal
- Naive approach using asymmetric keys has similar problems
- Symmetric-key solution: key distribution center (KDC)
- Asymmetric-key solution: certification authority (CA)
- Brings up new attacks involving no-longer-valid master keys
  - a TOCTOU vulnerability
- Domain: set of principals covered by one KDC or CA

Outline

Domain with a KDC

- KDC is a special principal in the domain (= network usually)
- Every other principal z shares a **master key**, say k_z, with KDC
- A-B session: A gets [session key, **ticket** for B] from KDC

client A (has k_A)	KDC (has k_A, k_B)	server B (has k_B)
send $[A, B]$ to KDC	rcv $[A, B]$ $S \leftarrow$ *random* // session key $t_A \leftarrow E(k_A, [A, B, S])$ $t_B \leftarrow E(k_B, [A, B, S])$ send $[t_A, t_B]$ to A	
rcv $[t_A, t_B]$ $\cdot, \cdot, S \leftarrow D(k_A, t_A)$ send $[A, B, t_B]$		
		rcv $[A, B, t_B]$ $\cdot, \cdot, S \leftarrow D(k_B, t_B)$

- Above is incomplete: eg, vulnerable to replay of S

Trust model, pros, cons

- Trust the KDC to not
  - issue weak keys, reuse keys, read msgs, impersonate others, etc
  - go offline
- Advantages of KDC
  - Adding new principal D: one interaction between D and KDC
  - Revocation of principal D: deactivate D's master key at KDC
- Disadvantages of KDC
  - KDC compromise makes the entire network vulnerable.
  - KDC failure means no new sessions can be started.
  - KDC can be a performance bottleneck.
- Replicating KDC fixes the last two disadvantages, but then need to protect replicas and keep them in sync
  - if master key changes, need to handle tickets issued with old key

Cross-domain session

- A's KDC is X
- B's KDC is Y
- X, Y share key k_{XY}
- A: send $[A, B.Y]$ to X
- X: generate session key k_{AY} // for A–Y session
  $t_{XA} \leftarrow E(k_{AX}, [A, Y, k_{AY}])$ // k_{AX}: A-X key
  $t_{XY} \leftarrow E(k_{XY}, [A, Y, k_{AY}])$ // k_{XY}: X-Y key
  send $[t_{XA}, t_{XY}]$ to A
- A: extract k_{AY} from t_{XA}; send $[A.X, B, t_{XY}]$ to Y
- Y: extract k_{AY} from t_{XY}
  generate session key k_{AB} // for A-B session
  $t_{YA} \leftarrow E(k_{AY}, [A, Y, k_{AB}])$
  $t_{YB} \leftarrow E(k_{BY}, [A, Y, k_{AB}])$ // k_{BY}: B-Y key
  send $[t_{YA}, t_{YB}]$ to A
- A: extract k_{AB} from t_{YA}; send $[A, B, t_{YB}]$ to Y
- B: extract k_{AB} from t_{YB} // A, B now share k_{AB}

Cross-domains session $A.X_1\text{-}X_2\text{-}\cdots\text{-}B.X_N$

- A gets [session key k_{A,X_2}, ticket t_{X_1,X_2}] from X_1
- A gets [session key k_{A,X_3}, ticket t_{X_2,X_3}] from X_2
- $\cdots$
- A gets [session key $k_{A,B}$, ticket $t_{X_N,B}$] from X_N
- A sends [ticket $t_{X_N,B}$] to B

- Better: A passes along the sequence of KDCs traversed, so that B sees the entire KDC-chain rather than just X_N

Kerberos: Unix KDC implemenation

- Kerberos 1
- Kerberos 2
- Commonly used in enterprise-level networks
- Handles
  - Changing master keys
  - Tickets: long-lived, post-dated, delegation, etc
  - Handles variety of crypto, hw architecture, etc
  - Compensates for weak keys (human users)
  - X-servers
  - Cross-domains authentication
  - lots more

Outline

Overview

Authentication basics

Authenticating humans

Storing passwords at servers

Scaling to many users and domains

- KDC: Key Distribution Center
- CA: Certification Authority

Domain with a CA – 1

- Every principal z has a public-key pair $[sk_Z, pk_Z]$
  - except some human principals may use passwords

- CA is a special principal, say with id X
- CA is trusted to create correct certificates
- CA issues a certificate for every z: $[z, pk_Z, \textit{expiry time}, \cdots, \textit{sgn}]$
  - *sgn*: CA's signature of the certificate // using sk_X
  - certificate is typically long-lived // eg, months, years
- CA can revoke z's certificate before expiry if needed
  - eg: sk_Z has become exposed, z leaves the domain, etc
- Every z has CA's public key
  - so z can verify certificates and their status (revoked or not)

Domain with a CA – 2

- To acquire y's public key
  - get y's certificate and and verify // using pk_X
  - get certificate's status and verify
  - can get these from anywhere // eg, y, a server, CA

- CA makes certificate status info available in two ways
  - Periodically issues a certificate revocation list (CRL)
    - list of all revoked unexpired certificates, signed by CA
    - unexpired certificate valid if it's not in a recent-enough CRL
  - On demand: issues a certificate's status (CS)
    - Online Certificate Status Protocol (OCSP)
    - CA (or its agent) must be online and responsive

Certificates

- Certificate for Z issued by X
  - serial number // for CRL
  - issuer: X's name, address, ...
  - subject: Z's name, address, ...
  - subject's public-key: pk_Z
  - expiry time // long-lived: month, year, ...
  - certificate's capabilities // eg, can Z issue certificates?
  - ...
  - X's signature on above

CRLs and CSs

- CRL issued by X
  - issuer: X's name, address, ...
  - issue time // frequent: hourly, daily, ...
  - list of serial numbers of all revoked unexpired certificates
  - X's signature on above
- CRL is typically huge // burden on client

- Certificate status (CS) of a certificate issued by X
  - serial number of certificate
  - issuer: X's name, address, ...
  - issue time // should be recent
  - status: still valid or no longer valid // as of issue time
- OCSP takes time // burden on client
- OCSP stapling: server provides CS (and certificate) to client

Obtaining Z's public-key

- Do step 1 and either step 2 or step 3

1. Obtain a certificate for Z issued by X.
   Check that the certificate has not expired.
   Verify the certificate's signature. // using pk_X

2. Obtain a recent-enough CRL issued by X that does not contain the certificate's serial number.
   Verify the CRL's signature. // using pk_X

or

3. Obtain a recent-enough CS (certificate status) issued by X that indicates the certificate is still valid
   Verify the CS's signature. // using pk_X

Handling users without public keys

- Consider client A and server B, where
  - B has public key
  - A does not have a public key
  - A shares *pwd* with B

- A–B session establishment
  - A obtains B's public key // using standard procedure
  - A sends $E_P(pk_B, pwd)$ to B

Trust model, pros, cons

- Trust the CA to
  - correctly vet principals
  - be online to handle OCSP requests
  - CA is the trust root // its public key is not verified

- Advantages of CA (vs KDC)
  - CA can be offline // if separate OCSP server
  - CA does not participate in A–B session
  - CA cannot decrypt A–B session
    (but it can impersonate a principal via false certificate)
  - CA failure does not stop new sessions until certs expire

- Disadvantages
  - Timely revocation is expensive // sloppily done in Internet

Certificate chains: crossing domains

- How does A verify B's public key if
  - A has a certificate issued by CA X // $cert_{XA}$
  - B has a certificate issued by CA Y // $cert_{YB}$
- Solution: X issues a certificate for Y // $cert_{XY}$
  - A verifies pk_Y using $cert_{XY}$ and cs_{XY} // cs_{XY}: revocation info
  - A verifies pk_B using pk_Y, $cert_{YB}$, cs_{YB}
- $[cert_{XY}, cs_{XY}], [cert_{YB}, cs_{YB}]$ is a certificate chain
- Certificate chain: $[cert_1, cs_1], [cert_2, cs_2], \cdots, [cert_n, cs_n]$
  - $[cert_j, cs_j]$ verifies public-key of $cert_{j+1}$'s issuer
  - $cert_1$'s issuer is the anchor of the chain
  - $cert_n$'s subject is the target of the chain
  - A can use the chain if the anchor is a trust root of A

PKI: Public-Key Infrastructure

- PKI is hierarchical
- Top-level CAs
  - Verisign, Comodo, Thawte, etc
  - Their public keys are pre-configured in OS/browsers/...
- Mid-level CAs
  - Have certificates from top-level/mid-level CAs
  - Issue certificates
  - Reputable and not // certificates for $10
- Low-level CAs // individuals and small organizations
  - May not have certificates issued by others
  - May issue certificates for internal use, accepted on faith

Web of Trust

- Non-hierarchical PKI
  - pioneered by PGP
- Anyone can issue certificates for people they know
- Directed graph of certificate chains can have cycles
- How to decide whether to trust a certificate chain?
  - anchor and intermediates
  - length of chain // shorter is better
  - how many other chains end at the same target
  - ...
- How to decide whether to issue a certificate for someone?
  - reputation, appearance, ... ???