Computer Security

08. Integrity & key exchange

Paul Krzyzanowski
Rutgers University
Spring 2018

Message Integrity

McCarthy's Spy Puzzle (1958)

The setting:

- Two countries are at war
- One country sends spies to the other country
- To return safely, spies must give the border guards a password

Conditions

- Spies can be trusted
- Guards chat – information given to them may leak

McCarthy's Spy Puzzle

Challenge

- How can a border guard authenticate a person without knowing the password?
- Enemies cannot use the guard's knowledge to introduce their own spies

Solution to McCarthy's puzzle

- Michael Rabin, 1958
- Use a one-way function, $B = f(A)$
  - Guards get B
    - Enemy cannot compute A if they know A
  - Spies give A, guards compute f(A)
    - If the result is B, the password is correct.
- Example function:
  - Middle squares
    - Take a 100-digit number (A), and square it
    - Let B = middle 100 digits of 200-digit result

One-way functions

- Easy to compute in one direction
- Difficult to compute in the other

Examples:

Factoring:

$pq = N$	EASY	Basis for RSA
find p,q given N	DIFFICULT	

Discrete Log:

$a^b \bmod c = N$	EASY	Basis for Diffie-Hellman & Elliptic Curve
find b given a, c, N	DIFFICULT	

Example of a one-way function

Example with an 18 digit number

A = 289407349786637777

A^2 = 83756614110525308948445338203501729

Middle square, B = 110525308948445338

Given A, it is easy to compute B

Given B, it is difficult to compute A

"Difficult" = no known short-cuts; requires an exhaustive search

Cryptographic hash functions

Cryptographic hash functions

Also known as a digests or fingerprints

Properties

- Arbitrary length input → **fixed-length output**
- Deterministic: you always get the same hash for the same message
- **One-way function (pre-image resistance, or *hiding*)**
  - Given *H*, it should be difficult to find *M* such that *H=hash(M)*
- **Collision resistant**
  - Infeasible to find any two different strings that hash to the same value: Find *M, M'* such that *hash(M) = hash(M')*
- **Output should not give any information about any of the input**
  - Like cryptographic algorithms, relies on *diffusion*
- **Efficient**
  - Computing a hash function should be computationally efficient

Hash functions are the basis of authentication

- Not encryption
- Can help us to detect:
  - Masquerading:
    - Insertion of message from a fraudulent source
  - Content modification:
    - Changing the content of a message
  - Sequence modification:
    - Inserting, deleting, or rearranging parts of a message
  - Replay attacks:
    - Replaying valid sessions

SHA-1 Overview

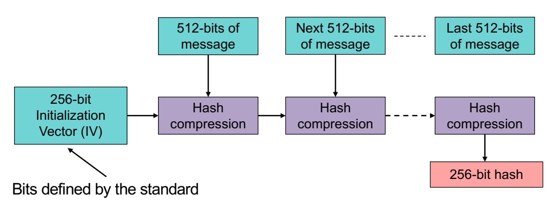
- Append the bit 1 to the message
- Pad message with 0 bits so its length = 448 mod 512
- Append length of message as a 64-bit big endian integer
- Initialize 5-word (160-bit) buffer to

```
a = 0x67452301  b = 0xefcdab89  c = 0x98badcfe
d = 0x10325476  e = 0xc3d2e1f0
```

- Process the message in 512-bit chunks
  - Expand the 16 32-bit words into 80 32-bit words via XORs & shifts
  - Iterate 80 times to create a hash for this chunk
    - Various sets of ANDs, ORs, shifts, and shifts
  - Add this hash chunk to the result so far

See https://www.saylor.org/site/wp-content/uploads/2012/07/SHA-1-1.pdf

SHA-2 Overview



Popular hash functions

- MD5 *R.I.P.*
  - 128 bits (rarely used now since weaknesses were found)
- SHA-1
  - 160 bits – was widely used: checksum in Git & torrents
  - Google demonstrated a *collision attack* in Feb 2017
    … Google had to run >9 quintillion SHA-1 computations to complete the attack
    … but already being phased out since weaknesses were found earlier
  - Used for message integrity in GitHub
- SHA-2 – *believed to be secure*
  - Designed by the NSA; published by NIST
  - SHA-224, SHA-256, SHA-384, SHA-512
    - e.g., Linux passwords used MD5 and now SHA-512
  - SHA-256 used in bitcoin
- SHA-3 – *believed to be secure*
  - 256 & 512 bit

Linux commands

- sha1sum: create a SHA-1 hash

```
echo "hello, world!" | sha1sum
e91ba0972b9055187fa2efa8b5c156f487a8293a  -
```

- md5sum: create an MD5 hash

```
echo "hello, world!" | md5sum
910c8bc73110b0cd1bc5d2bcae782511  -
```


Collisions: The Birthday Paradox

How many people need to be in a room such that the probability that two people will have the same birthday is > 0.5?

Your guess before you took a probability course: 183

- This is true to the question of "how many people need to be in a room for the probability that someone else will have the same birthday as Alice?"

Answer: 23

$$p(n) = 1 - \frac{n! \cdot \binom{365}{n}}{365^n}$$

Approximate solution for # people required to have a 0.5 chance of a shared birthday, where m = # days in a year $n \approx \sqrt{2 \times m \times 0.5}$

The Birthday Paradox: Implications

- Searching for a collision with a pre-image (known message) is *A LOT* harder than searching for two messages that have the same hash
- Strength of a hash function is approximately ½ (# bits)
  - 256-bit hash function has a strength of approximately 128 bits

Message Integrity

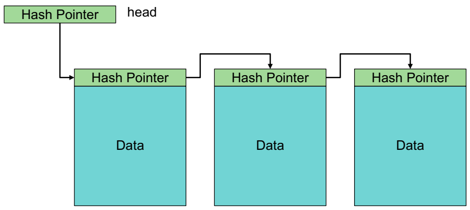
How do we detect that a message has been tampered?

- A cryptographic hash acts as a checksum
  $H(M) \neq H(M')$
- Associate a hash with a message
  – we're not encrypting the message
  – we're concerned with *integrity*, not *confidentiality*

Hash Pointers

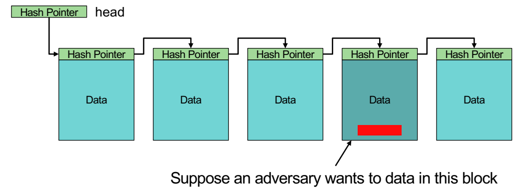
- We can use hash pointers instead of pointers in data structures
- Hash pointer = { pointer, *hash*(data) }
- This allows us to verify that the information we're pointing to has not changed

Hash Pointers: Linked Lists

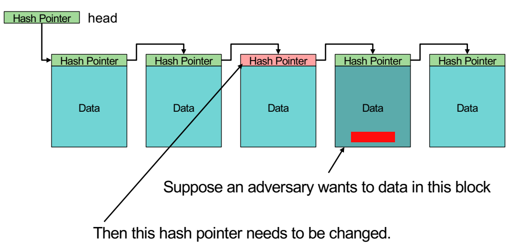


- Add new data blocks to the end of the list
- Tamper Evident Log = blockchain

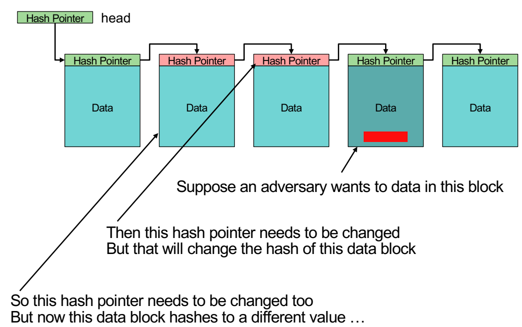
Tamper detection



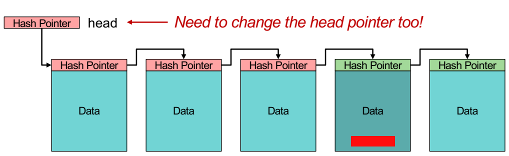
Tamper detection



Tamper detection



Tamper detection

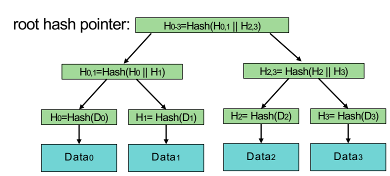


The adversary will have to change all hash pointers back to the root.

If we're holding on to the head of the list so an adversary cannot modify it, then we will be able to detect tampering.

Merkle Trees: Binary trees with hash pointers

Merkle Tree Hash pointer = { left_subtree, right_subtree, *hash*(left || right) }



- Another tamper-resistant structure
- Only need to examine $O(\log_2 n)$ hashes to validate data

a || *b* means *a* concatenated with *b*

Tamperproof Integrity: Message Authentication Codes and Digital Signatures

Message Integrity: MACs

- We rely on hashes to assert the integrity of messages
- An attacker can create a new message & a new hash and replace $H(M)$ with $H(M')$
- So let's create a checksum that relies on a key for validation

Message Authentication Code (MAC)

Hash-based MAC

We can create a MAC from a cryptographic hash function

HMAC = Hash-based Message Authentication Code

$HMAC(m, k) = H((opad \oplus k) \,||\, H(ipad \oplus k) \,||\, m))$

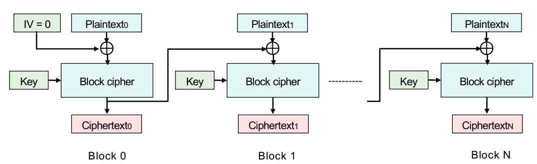
Where

- H = cryptographic hash function
- $opad$ = outer padding 0x5c5c5c5c … (01011100…)
- $ipad$ = inner padding 0x36363636… (00110110…)
- k = secret key
- m = message
- $\oplus$ = XOR, $||$ = concatenation

See RFC 2104

Block cipher based MAC: CBC-MAC

- Cipher block chaining assures that every encrypted block is a function of all previous blocks
- CBC MAC uses a zero initialization vector

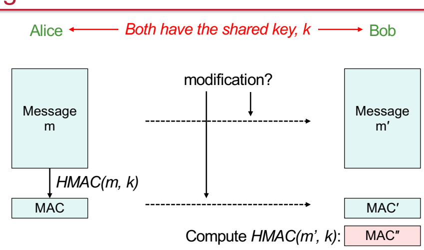


MAC = final ciphertext block – others are discarded

Examples: AES-CBC-MAC, DES-MAC

Don't use the same key for the MAC as for encrypting the message

Using a MAC



1. Bob receives the Message m' and a MAC.
2. Knowing the key, k, he generates a MAC for the message: MAC'' = HMAC(m', k)
3. If MAC' = MAC'', he's convinced that the message has not been modified

Digital Signatures

- MACs rely on a shared key
  - Anyone with the key can modify and re-sign a message
- Digital signature properties
  - Only you can sign a message but anyone can validate it
  - You cannot cut and paste the signature from one message to another
  - An adversary cannot forge a signature
    - Even after inspecting an arbitrary number of signed messages

Digital Signature Primitives

1. Key generation
   { secret_key, verification_key } := **gen_keys**(key_size)
2. Signing
   signature := **sign**(message, secret_key)
3. Validation
   Isvalid := **verify**(verification_key, message, signature)

We sign *hash(message)* instead of the *message*
- We'd like the signature to be a small, fixed size
- We trust hashes to be collision-free
- We can use a signature in a hash pointer
  - This will protect the integrity of the entire data structure

Popular Digital Signature Algorithms

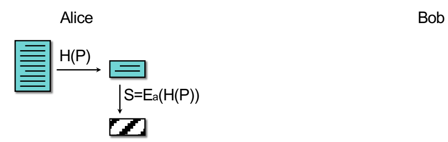
- DSA: Digital Signature Algorithm
  - NIST standard
  - Uses SHA-1 or SHA-2 hash
  - Key pair based on difficulty of computing discrete logarithms
- ECDSA: Elliptic Curve Digital Signature Algorithm
  - Variant of DSA that uses elliptic curve cryptography
  - Used in bitcoin
- RSA cryptography
  - $E_{pri\_key}(H(M))$, $D_{pub\_key}(H(M))$

Note: not Diffie-Hellman, which is only a key exchange algorithm

Digital signatures: public key cryptography

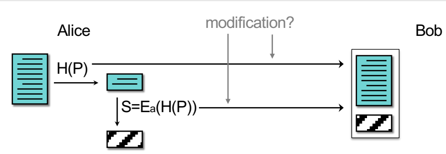
Alice generates a hash of the message

Digital signatures: public key cryptography



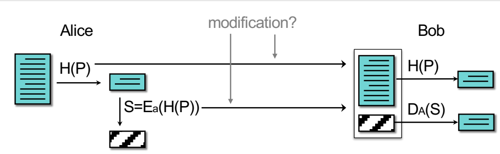
Alice encrypts the hash with her private key
This is her **signature.**

Digital signatures: public key cryptography



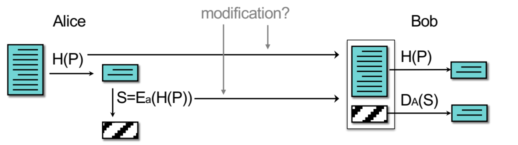
Alice sends Bob the message & the encrypted hash

Digital signatures: public key cryptography



1. Bob decrypts the hash using Alice's public key
2. Bob computes the hash of the message sent by Alice

Digital signatures: public key cryptography

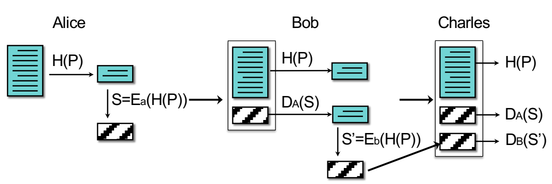


If the hashes match, the signature is valid
– the encrypted hash *must* have been generated by Alice

Digital signatures & non-repudiation

- Digital signatures provide non-repudiation
  – Only Alice could have created the signature because only Alice has her private key
- Proof of integrity
  – The hash assures us that the original message has not been modified
  – The encryption of the hash assures us that an attacker could not have re-created the hash

Digital signatures: multiple signers



Charles:
- Generates a hash of the message, H(P)
- Decrypts Alice's signature with Alice's public key
  - Validates the signature: $D_A(S) \stackrel{?}{=} H(P)$
- Decrypts Bob's signature with Bob's public key
  - Validates the signature: $D_B(S) \stackrel{?}{=} H(P)$

Covert AND authenticated messaging

If we want to keep the message secret
– combine encryption with a digital signature

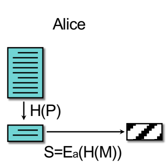
Use a session key:

– Pick a random key, *K*, to encrypt the message with a symmetric algorithm

– encrypt *K* with the public key of each recipient

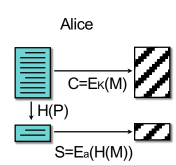
– for signing, encrypt the hash of the message with sender's private key

Covert and authenticated messaging



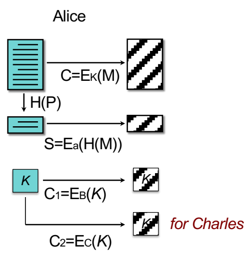
Alice generates a digital signature by encrypting the message with her private key

Covert and authenticated messaging



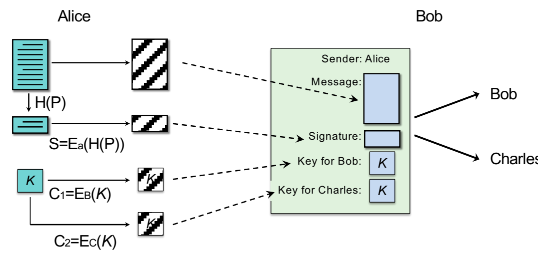
Alice picks a random key, *K*, and encrypts the message *P* with it using a symmetric cipher

Covert and authenticated messaging



Alice encrypts the session key for each recipient of this message using their public keys

Covert and authenticated messaging



The aggregate message is sent to Bob & Charles

Note: we do not have forward secrecy by doing this

Public Keys as Identities

- A public signature verification key can be treated as an identity
  - Only the owner of the corresponding private key will be able to create the signature
- New identities can be created by generating new random {private, public} key pairs
- **Anonymous** – no identity management
  - A user is known by a random-looking public key
  - Anybody can create a new identity at any time
  - Anybody can create as many identities as they want
  - A user can throw away an identity when it is no longer needed
  - Example: Bitcoin identity = hash(public key)

Certificates: Identity Binding

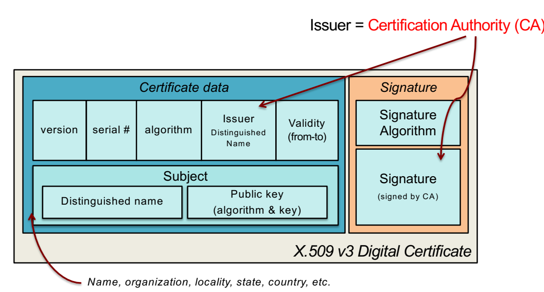
Identity Binding

- How does Alice know Bob's public key is really his?
- Get it from a trusted server?
  - What if the enemy tampers with the server?
  - Or intercepts Alice's query to the server (or the reply)?
  - What set of public keys does the server manage?
  - How do you find it in a trustworthy manner?
- Another option
  - Have a trusted party sign Bob's public key
  - Once signed, it is tamper-proof

X.509 Certificates

ISO introduced a set of authentication protocols

X.509: Structure for public key certificates:



X.509 certificates

To validate a certificate

Verify its signature:

1. Hash contents of certificate data
2. Decrypt CA's signature with CA's public key

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

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

…if you trust the CA

Certification Authorities (CAs)

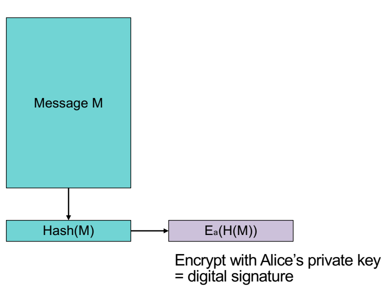
- How do you know the public key of the CA?
  - You can get it from another certificate!
  - This is called **certificate chaining**
  - But trust has to start somewhere: you need a public key you can trust (probably sitting inside a certificate) – this is the **root CA**
    - Apple's keychain is pre-loaded with hundreds of CA certificates
    - Windows stores them in the Certificate Store and makes them accessible via the Microsoft Management Console (mmc)
    - Android stores them in Credential Storage
- Can you trust a CA?
  - Maybe…
    check their reputation and read their *Certification Practice Statement*
  - Even trustworthy ones might get hacked (e.g., VeriSign in 2010)

Key revocation

- Used to invalidate certificates before expiration time
  - Usually because of a compromised key
  - Or policy changes (e.g., someone leaves a company)
- Certificate revocation list (CRL)
  - Lists certificates that are revoked
  - Only certificate issuer can revoke a certificate
- Problems
  - Need to make sure that the entity issuing the revocation is authorized to do this
  - Revocation information may not circulate quickly enough
    - Dependent on dissemination mechanisms, network delays & infrastructure
    - Some systems may not have been coded to process revocations

Code Integrity

Review: signed messages



We can sign code too

- Validate integrity of the code
  - If the signature matches, then the code has not been modified
- Enables
  - Distribution from untrusted sources
  - Distribution over untrusted channels
  - Detection of modifications by malware
- Signature = encrypted hash signed by trusted source
  - Does *not* validate the code is good … just where it comes from

Code Integrity: signed software

- Windows 7-10: Microsoft Authenticode
  - SignTool command
  - Hashes stored in system catalog or signed & embedded in the file
  - Microsoft-tested drivers are signed
- macOS
  - codesign command
  - Hashes & certificate chain stored in file
- Also Android & iOS

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
- Microsoft SmartScreen:
  - Manages reputation based on download history, popularity, anti-virus results
- Recipient:
  - Call *WinVerifyTrust* function to validate:
    - Validate certificate, decrypt digest, compare with hash of downloaded code

Per-page hashing

- Integrity check when program is first loaded
- Per-page signatures – improved performance
  - Check hashes for every page upon loading (demand paging)
- Per-page hashes can be disabled optionally on both Windows and macOS

Windows 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 stored 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

Key exchange algorithms

Notation

$Z \,||\, W$
- *Z* concatenated with *W*

$X \rightarrow Y: \{ Z \,||\, W \}\, k_{A,B}$
- *X* sends a message to *Y*
- The message is the concatenation of Z & W and is encrypted by key $k_{A,B}$, which is shared by users *A* & *B*

$X \rightarrow Y: \{ Z \}\, k_A \;||\; \{ W \}\, k_{A,Y}$
- X sends a message to Y
- The message is a concatenation of Z encrypted using A's key and W encrypted by a key shared by A and Y

r_1, r_2
- nonces – strings of random bits

Bootstrap problem

- How to Alice & Bob communicate securely?
- Alice cannot send a key to Bob in the clear
  - We assume an unsecure network
- We looked at two mechanisms:
  - Diffie-Hellman key exchange
  - Public key cryptography
- Let's examine the problem some more

Simple Protocol

Use a trusted third party – Trent – who has all the keys

Trent transmits a session key to Alice and Bob

Alice → Trent: { Request session key to Bob } k_A

Alice ← Trent: { ks } k_A || { ks } k_B

Alice → Bob: { ks } k_B

Alice ↔ Bob: { m } ks

Problems

- How does Bob know he is talking to Alice?
  - Trusted third party, Trent, has all the keys
  - Trent knows the request came from Alice since only he and Alice can have the key
  - Trent can authorize Alice's request
  - Bob gets a message (session key) encrypted with his key, which only Trent could have created
    - But Bob doesn't know who requested the session
    - Trent would have to add sender information to the message
- Vulnerable to replay attacks
  - Eve records the message from Alice to Bob and later replays it
  - Bob might think he's talking to Alice, reusing the same session key
- Protocols should provide authentication & defend against replay

Needham-Schroeder

Add *nonces* – random strings – avoid replay attacks

Alice → Trent: { Alice || Bob || r_1 }

Alice ← Trent: { Alice || Bob || r_1 || ks || { Alice || ks } k_B } k_A

Alice → Bob: { Alice || ks } k_B

Alice ← Bob: { r_2 } ks

Alice → Bob: { r_2 – 1 } ks

Needham-Shroeder

Add *nonces* – random strings – avoid replay attacks

Alice → Trent: { Alice || Bob || r_1 }

Alice ← Trent: { Alice || Bob || r_1 || ks || { Alice || ks } k_B } k_A

Alice → Bob: { Alice || ks } k_B

Alice ← Bob: { r_2 } ks

Alice → Bob: { r_2 – 1 } ks

Message must have been created by Trent & is a response to the first message (contains r_1). Use of r_1 ensures it's not a replay attack.

- Alice knows only Bob & Trent can read this and get the session key.
- Bob knows it's a request from Alice

- Bob now tries to find out if this is a replay attack
- If it is, Eve cannot decipher r_2

This is an **authentication** step: Bob asks Alice to prove she has ks

Denning-Sacco Modification

Needham-Schroeder is still vulnerable to a certain replay attack!

- We assume all keys are secret
- But suppose Eve can obtain the session key from an old message (she worked hard, got lucky, and cracked an earlier message)

Eve → Bob: { Alice || ks } k_B

Eve ← Bob: { r_2 } ks

Eve → Bob: { r_2 – 1 } ks

Bob sees this as a legitimate request approved by Trent. It was (but earlier)!

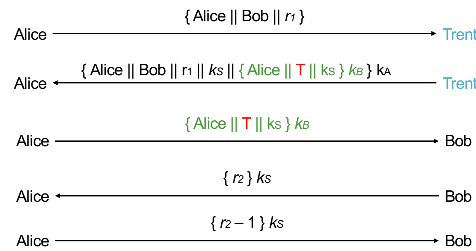
Eve the eavesdropper. She decrypted an old session key and is trying to get Bob to use it to think he's talking to Alice.

Solution

- Problem: replay in the third step of the protocol
  - Eve replays the message: { Alice || k_S } k_B
- Solution: use a time stamp *T* to detect replay attacks

Needham-Shroeder w/Denning-Sacco mods

Add nonces – random strings – AND a timestamp



Problem with timestamps

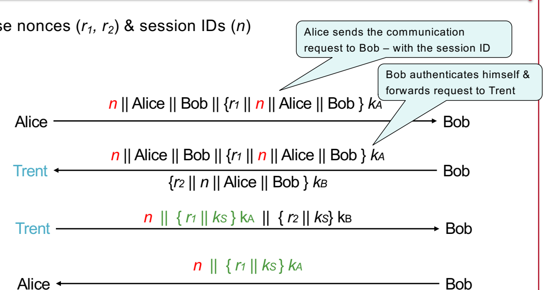
- Use of timestamps relies on synchronized clocks
  - Messages may be falsely accepted or falsely rejected because of bad time
- Time synchronization becomes an attack vector
  - Create fake NTP responses
  - Generate fake GPS signals

Otway-Rees Protocol: Session IDs

- Another way to correct the *third message replay* problem
- Instead of using timestamps
  - Use a random integer, *n*, that is associated with all messages in the key exchange

Otway-Rees Protocol

Use nonces (r_1, r_2) & session IDs (n)



Kerberos

Kerberos

- Authentication service developed by MIT
  - project Athena 1983-1988
- Uses a trusted third party & symmetric cryptography
- Based on Needham Schroeder with the Denning Sacco modification
- 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

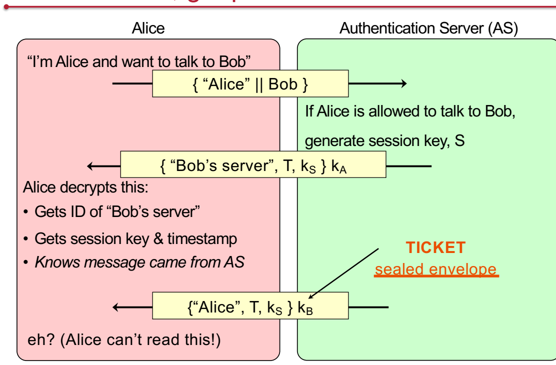
Kerberos is a trusted third party

- 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)

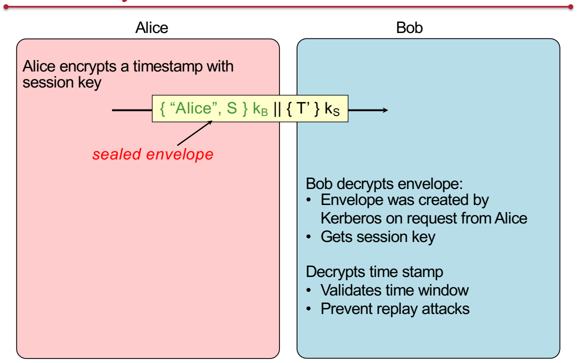
Kerberos

- 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 *ticket* (*sealed envelope*)
- Step 2:
  - Alice gives Bob the ticket, which contains the session key
  - Convinces Bob that she got the session key from Kerberos

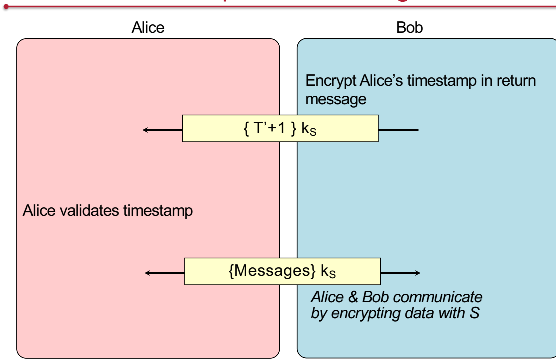
Authenticate, get permission



Send key



Authenticate recipient of message



Kerberos key usage

- Every time a user wants to access a service
  - User's password (key) must be used to decode the message from Kerberos
- We can avoid this by caching the password in a file
  - Not a good idea
- Another way: create a temporary password
  - We can cache this temporary password
  - Similar to a session key for Kerberos – to get access to other services
  - Split Kerberos server into
    Authentication Server + Ticket Granting Server

Ticket Granting Server (TGS)

- TGS works like a temporary ID
- User first requests access to the TGS
  - Contact Kerberos Authentication Server
    - Knows all users & their secret keys
    - User enters a password to do this
    - Gets back a ticket & session key to the TGS – these can be cached
- To access any service
  - Send a request to the TGS – encrypted with the TGS session key along with the ticket for the TGS
  - The ticket tells the TGS what your session key is
  - It responds with a session key & ticket for that service

Using Kerberos

$ kinit

Password: *enter password*

ask AS for permission (session key) to access TGS

Alice gets:

{"TGS", S } k_A	← *Session key*
{"Alice", S } k_{TGS}	← *TGS Ticket*
{ T } k_S	← *Encrypted timestamp*

Compute key (A) from password to decrypt session key S and get TGS ID.

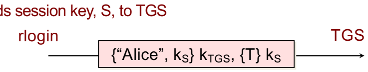
You now have a ticket to access the Ticket Granting Service

Using Kerberos

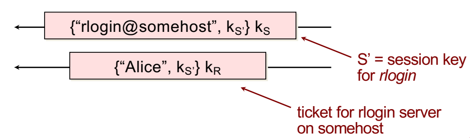
$ rlogin *somehost*

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

Alice sends session key, S, to TGS



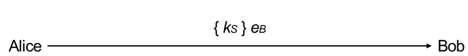
Alice receives session key for rlogin service & ticket to pass to rlogin service



Public Key Exchange

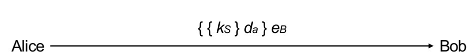
We did this

- Alice's & Bob's public keys known to all: e_A, e_B
- Alice & Bob's private keys are known only to the owner: d_a, d_b
- Simple protocol to send symmetric session key: k_S



Problem

- Vulnerable to forgery or replay
- Public keys are known to anyone
  - Bob has no assurance that Alice sent the message
- Fix: have Alice sign the session key



Key k_S encrypted with Alice's private key
Entire message encrypted with Alice's public key

Problem #2

- How do we know we have the right public keys?
- Send a certificate so Bob can verify it

Alice —— $\{ \{ k_S \} d_a \} e_B$, X ——→ Bob

Add Alice's certificate, which contains Alice's verifiable public key

The End