Key Management

- Where do keys come from?
- More precisely, we have to distinguish between long-lived keys and *session keys*
- General solution: use long-lived key for authentication and to negotiate session key
- Many different ways to do this

Desired Properties

- Alice and Bob want to end up with a shared session key K, with the help of a key server S.
- They each want proof of the other's identity
- They want to be sure the key is *fresh*
- A fresh key is one that hasn't been used before, i.e., is not a replay

Why is Freshness Important?

- For stream ciphers, it’s crucial
- If too much traffic is encrypted with any key, it might help a cryptanalyst
- If too much traffic is encrypted with any one key, it’s a very tempting target for a cryptanalyst
- An old key may have somehow been compromised

Key Management for Symmetric Ciphers

- Simplest case: each pair of communicators has a shared key
- Doesn't scale.
- Besides, cryptographically unwise — each key is used too much
- Need a *Key Distribution Center* (KDC)

Needham-Schroeder Protocol (1978)

$$A \rightarrow S : \quad A, B, N_A \tag{1}$$

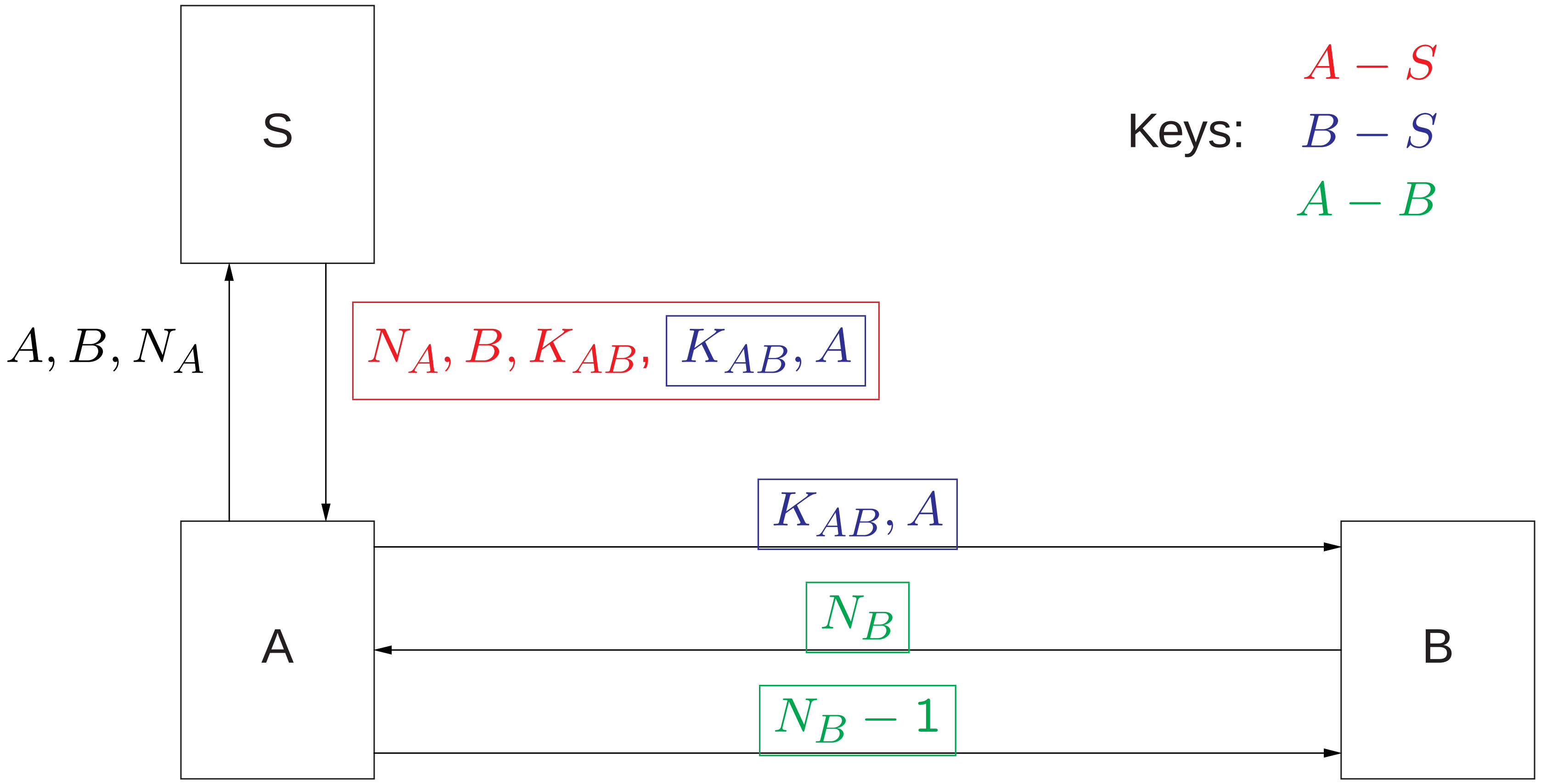
$$S \rightarrow A : \quad \{N_A, B, K_{AB}, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}} \tag{2}$$

$$A \rightarrow B : \quad \{K_{AB}, A\}_{K_{BS}} \tag{3}$$

$$B \rightarrow A : \quad \{N_B\}_{K_{AB}} \tag{4}$$

$$A \rightarrow B : \quad \{N_B - 1\}_{K_{AB}} \tag{5}$$

Needham-Schroeder Protocol



Explaining Needham-Schroeder

(1) Alice sends S her identity, plus a random *nonce*

(2) S's response is encrypted in K_{AS}, which guarantees its authenticity. It includes a new random session key K_{AB}, plus a sealed package for Bob

(3) Alice sends the sealed package to Bob. Bob knows it's authentic, because it's encrypted with K_{BS}

(4) Bob sends his own random nonce to Alice, encrypted with the session key

(5) Alice proves that she could read the nonce

Cryptographic Protocol Design is Hard

- Bob never proved his identity to Alice
- If K_{AB} is ever compromised, the attacker can impersonate Alice forever
- Denning and Sacco proposed a fix for this problem in 1981.
- In 1994, Needham found a flaw in their fix.
- In 1995, a new flaw was found in the public key version of the original Needham-Schroeder protocol — in modern notation, that protocol is only 3 messages.
- Cryptographic protocol design is hard. . .

Revisiting Diffie-Hellman

- A few days ago, we discussed the Diffie-Hellman algorithm, as a way to generate session keys without prearrangement
- I (deliberately) omitted something: the protocol is unauthenticated
- That is, Alice doesn't know if she's talking to Bob or someone else

Attacking DH Exponential Key Exchange

Suppose we have a man-in-the-middle between Alice and Bob. . .

$$
\begin{aligned}
A \rightarrow M &: \quad g^x \bmod p \\
M \rightarrow B &: \quad g^z \bmod p \\
B \rightarrow M &: \quad g^y \bmod p \\
M \rightarrow A &: \quad g^{z'} \bmod p
\end{aligned}
$$

Alice and M share a key $g^{xz} \bmod p$; Bob and M share a key $g^{yz'} \bmod p$.

When Alice sends a message towards Bob, M decrypts it, reads it and perhaps modifies it, re-encrypts it, and sends it to Bob.

Diffie-Hellman key exchange provides no authentication — and if Alice or Bob sent a password, M would read that, too.

Man-in-the-Middle Attacks

- An attacker who does more than just listen to communications
- Sits in the middle of a channel and relays messages back and forth
- Of course, the messages aren't always relayed intact. . .

Authenticating Diffie-Hellman

- Alice and Bob — and perhaps M — engage in a Diffie-Hellman exchange.
- Bob digitally signs a hash of the exchanged exponentials, and transmits it; Alice does the same.
- M can't tamper with digitally-signed messages, so they have to arrive intact
- If there's an attacker, Alice and Bob realize that the signed key doesn't match their own key, so they know there's something wrong.
- (Station-to-station protocol)

Other Cryptographic Protocols

- Cryptographic protocols allow us to do many strange things, such as signing a message you can't see
- Too many to discuss in this class; here are a few small examples

Coin Flips

- How do you flip a coin on the Internet, without a trusted third party?
- Alice picks a random number x, and sends $H(x)$ to Bob, where H is a cryptographic hash function.
- Bob guesses if x is even or odd, and sends his guess to Alice.
- If Bob's guess is right, the result is heads; if he's wrong, the result is tails.
- Alice discloses x. Both sides can verify the result. Alice can't cheat, because she can't find an x' such that $H(x) = H(x')$.
- Note: this protocol is crucially dependent on the lack of correlation between the parity of x and the values of $H(x)$, or Bob can cheat.

Strong Password Protocols

- Suppose a user has to supply a key
- Users can’t remember long random strings; they can remember passwords
- Suppose we use some function $F(P)$, where P is the password
- The enemy intercepts $\{M\}_{F(P)}$ and guesses at the password to decrypt the message
- If M makes sense — if it has *verifiable plaintext* — the enemy knows the guess was correct and can read all traffic
- We need a scheme that prevents password-guessing

Encrypted Key Exchange (EKE)

- Alice and Bob prepare Diffie-Hellman exponentials $g^x \bmod p$ and $g^z \bmod p$
- D-H exponentials are (approximately) uniformly-distributed random numbers in $[0, p-1]$
- Alice and Bob then encrypt the exponentials with Alice's password and transmit them:

$$A \rightarrow B: \quad \{g^x \bmod p\}_P$$
$$B \rightarrow A: \quad \{g^y \bmod p\}_P$$

- If the attacker guesses wrong about P, he gets a random number
- If he guesses right, he gets a random-looking number
- The only way to tell is to solve the discrete log problem!

Kerberos

- Originally developed at MIT; now an essential part of Windows authentication infrastructure.
- Designed to authenticate users to servers
- Users must use their password as their initial key — and must not be forced to retype it constantly
- Based on Needham-Schroeder, with timestamps to limit key lifetime

“Kerberos” in Greek Mythology

Kerberos; also spelled Cerberus. *n*. The watch dog of Hades, whose duty it was to guard the entrance—against whom or what does not clearly appear; . . . it is known to have had three heads. . .

—Ambrose Bierce, The Enlarged Devil’s Dictionary

Design Goals

- Users only have passwords to authenticate themselves
- The network is completely insecure
- It's possible to protect the Kerberos server
- The workstations have not been tampered with (dubious!)

Resources Protected

- Workstation login
- Network access to home directory
- Printer
- IM system
- Remote login
- Anything else that requires authentication

Principals

- A Kerberos entity is known as a *principal*
- Could be a user or a system service
- Principal names are triples: $\langle$*primary name*, *instance*, *realm*$\rangle$
- Examples: username@some.domain.name, somehost/lpr@other.domain
- The *realm* identifies the Kerberos server

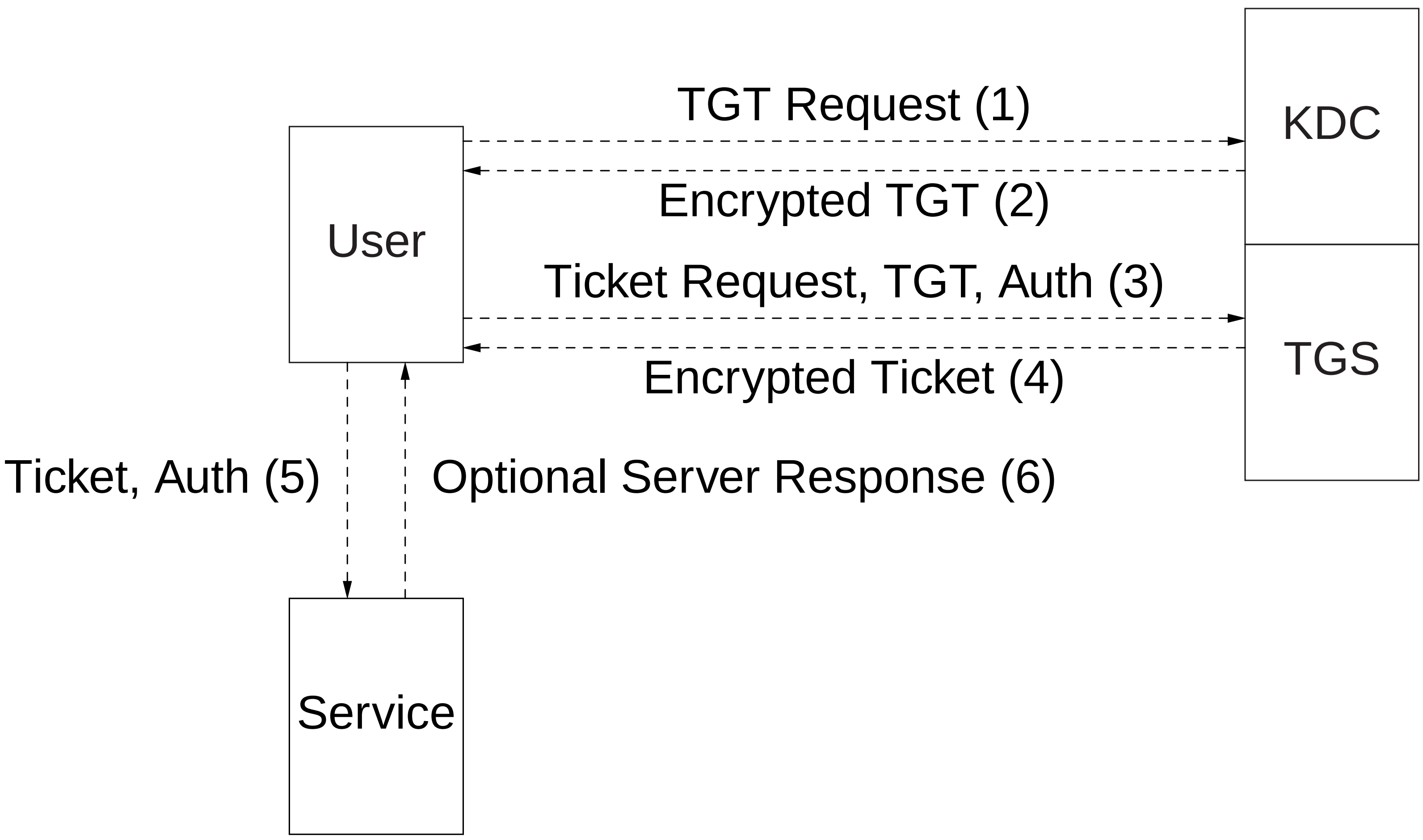
How Kerberos Works

- Users present *tickets* — cryptographically sealed messages with session keys and identities — to obtain a service.
- Use Needham-Schroeder (with password as Alice's key) to get a *Ticket-Granting Ticket* (TGT); this ticket (and the associated key) are retained for future use during its lifetime.
- Use the TGT (and TGT's key) in a Needham-Schroeder dialog to obtain keys for each actual service

Shared Secrets

- Everyone shares a secret with the Kerberos KDC
- For users, this is their password (actually, a key derived from the password)
- The KDC is assumed to be secure and trustworthy; anything it says can be believed

Kerberos Data Flow



Getting a Ticket-Granting Ticket (TGT)

- The user sends its principal name to the Kerberos KDC
- The KDC responds with

$$\{K_{c,tgs}, \{T_{c,tgs}\}_{K_{tgs}}\}_{K_c}$$

- That is, it contains a session key $K_{c,tgs}$ and a TGT encrypted with a key known only to the KDC
- The ticket contains

$$\{tgs, c, \textit{addr}, \textit{timestamp}, \textit{lifetime}, K_{c,s}\}_{K_{tgs}}$$

- It has the service name (tgs), the principal's name, its IP address, the validity period, and the session key $K_{c,tgs}$ sent to the client
- K_c is the user's password, known to the user and the KDC

Who Knows What Now?

- The user and the KDC know K_c; the user use it to decrypt $\{K_{c,tgs}\}_{K_c}$ and recover $K_{c,tgs}$
- Only the KDC knows K_{tgs}; therefore, anything encrypted with that key could only have been created by the KDC
- The user will use $K_{c,tgs}$ plus the ticket-granting ticket to obtain more credentials

Using the TGT

- The client uses the TGT to obtain tickets for other services
- To get a ticket for service s — say, email access — it sends s (email), the ticket, and an *authenticator* to the KDC
- The KDC uses this information to construct a service ticket

Authenticators

- Authenticators prove two things: that the client knows $K_{c,s}$, and that the ticket is fresh
- An authenticator for a service s contains

$$\{c, \textit{addr}, \textit{timestamp}\}_{K_{c,s}}$$

- That is, it contains the client name and IP address, plus the current time, encrypted in the key associated with that ticket
- For a ticket-granting ticket, s is the tgs

Processing the Ticket Request

- The KDC decrypts the ticket to recover $K_{c,tgs}$
- It uses that to decrypt the authenticator
- It verifies the IP address and the timestamp (permissible clock skew is typically a few minutes)
- If everything matches, it knows that the request came from the real client, since only it would have access to the $K_{c,tgs}$ that was in the ticket
- It then sends a service ticket back to the client

Service Tickets

- Service tickets are almost identical to ticket-granting tickets
- The differences is that they have the name of a different service — say, “email” — rather than the ticket-granting service
- They’re encrypted in a key shared by the KDC and the service

Using Service Tickets

- The client sends the service ticket and an authenticator to the serivce
- The service decrypts the ticket, using its own key
- The service knows it's genuine, because only the KDC knows the key used to produce it
- The service verifies that the ticket is for it and not some other service
- It uses the enclosed key to decrypt and verify the authenticator
- The net result is that the service knows the client's principal name, extracted from the ticket

Authentication, Not Authorization

- Kerberos is an *authentication* service
- It does not (usually) provide authorization
- The services know a genuine name for the client, vouched for by the KDC
- They then make their own authorization decision based on this name

Bidirectional Authentication

- Sometimes, the client wants to be sure of the server's identity
- It asks the server to prove that it, too, knows the session key
- The server replies with $\{\textit{timestamp} + 1\}_{K_{c,s}}$ using the same timestamp as was in the authenticator

Ticket Lifetime

- TGTs typically last about 8–12 hours — the length of a login session
- Service tickets can be long- or short-lived, but don't outlive the TGT
- Live tickets are cached by the client
- When service tickets expire, they're automatically and transparently renewed

Inter-Realm Tickets

- A ticket from one realm can't be used in another, since a KDC in one realm doesn't share secrets with services in another realm
- Realms can issue tickets to each other
- A client can ask its KDC for a TGT to another realm's KDC
- The remote realm trusts the user's KDC to vouch for the user's identity
- It then issues serivce tickets *with the original realm's name* for the principal, not its own realm name
- As always, services use the principal name for authorization decisions

Putting Authorization into Tickets

- Under certain circumstances, tickets can contain authorization information known or supplied to the KDC
- Windows KDCs use this, to centralize authorization data
- (As a result, Windows and open source Kerberos KDCs don't interoperate well. . .)
- Users can supply some authorization data, too, to restrict what other services do with *proxy tickets*

Proxy Tickets

- Suppose a client wants to print a file
- The print spooler doesn't want to copy the user's file; that's expensive
- The user obtains a *proxy ticket* granting the print spooler access to its files
- The print spooler uses that ticket to read the user's file

Restricting the Print Spooler

- The client doesn't want the spooler to have access to all of its files
- It lists the appropriate file names in the proxy ticket request; the KDC puts that list of names into the proxy ticket
- When the print spooler presents the proxy ticket to a file server, it will only be given those files
- Note: the file server must verify that the client has access to those files!

Kerberizing Applications

- Replace (or supplement) existing authentication mechanisms with something that uses Kerberos
- Add authorization check
- If necessary (and it probably is, these days), change all network I/O to use the Kerberos session key to encrypt and authenticate all messages

Limitations of Kerberos

- Ticket cache security
- Password-guessing
- Subverted login command

Ticket Cache Security

- Where are cached tickets stored?
- Often in `/tmp` — is the OS protection good enough?
- Less of an issue on single-user workstations; often a threat on multi-user machines
- Note: `/tmp` needs to be a local disk, and not something mounted via NFS. . .

Password-Guessing

- Kerberos tickets have verifiable plaintext
- An attacker can run password-guessing programs on intercepted ticket-granting tickets
- (Mike Merritt and I invented EKE while studying this problem with Kerberos.)
- Kerberos uses *passphrases* instead of *passwords*
- Does this make guessing harder? No one knows

It's Worse Than That

- On many Kerberos systems, anyone can ask the KDC for a TGT
- There's no need to eavesdrop to get them — you can get all the TGTs you want over the Internet!
- Solution: *preauthentication*
- The initial request includes a timestamp encrypted with K_c
- It's still verifiable plaintext, but collecting TGTs becomes harder again

Subverting Login

- No great solutions!
- Keystroke loggers are a real threat today
- Some theoretical work on secure network booting
- Perhaps use the Trusted Computing mechanisms to protect passphrase entry? Unclear it it will really help