University of Pittsburgh

Applied Cryptography and Network Security
CS 1653

Summer 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Prof. Adam Lee's CS1653 slides.)

Announcements

- Homework 5 due this Friday @ 11:59 pm
- Project Phase 2 due this Friday @ 11:59 pm
- Homework 6 due Friday 7/7 @ 11:59 pm
- Homework 7 due this Friday 7/14 @ 11:59 pm
- Programming Assignment 1 posted tonight
  - Due on Friday 7/7
- Midterm Exam next Monday
  - Study guide on Canvas
  - Review session this Wednesday

Handshake Protocols

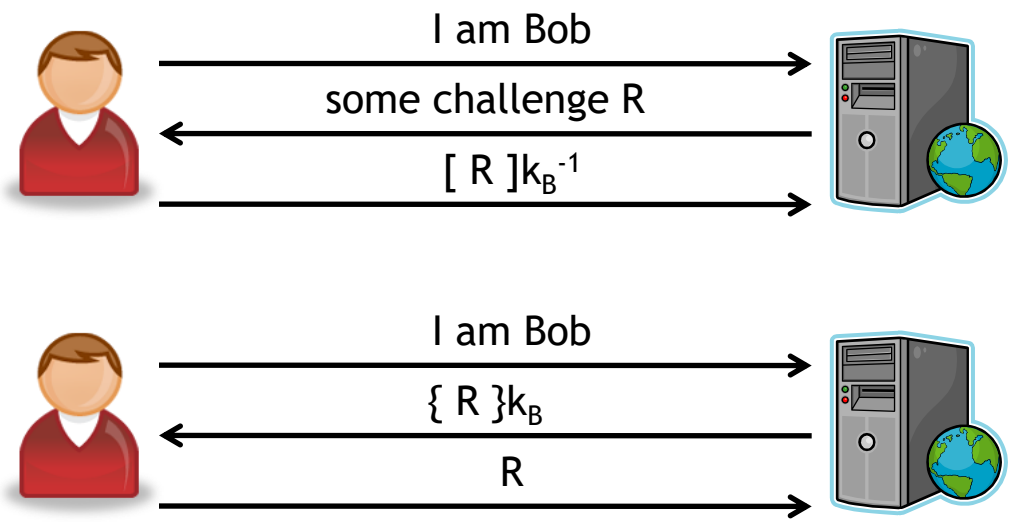
We'll start looking at four types of handshake protocols:

- Login-only protocols
- Mutual authentication protocols
- Integrity/encryption setup protocols
- Mediated authentication protocols

As we'll see, there is a lot of subtlety that goes into designing these types of protocols

Login-only protocols

These protocols can also be adapted to use public key cryptography



Why do these protocols work?

- Protocol 1: Only Bob can generate [R]k_B^{-1}
- Protocol 2: Only Bob can open { R }k_B

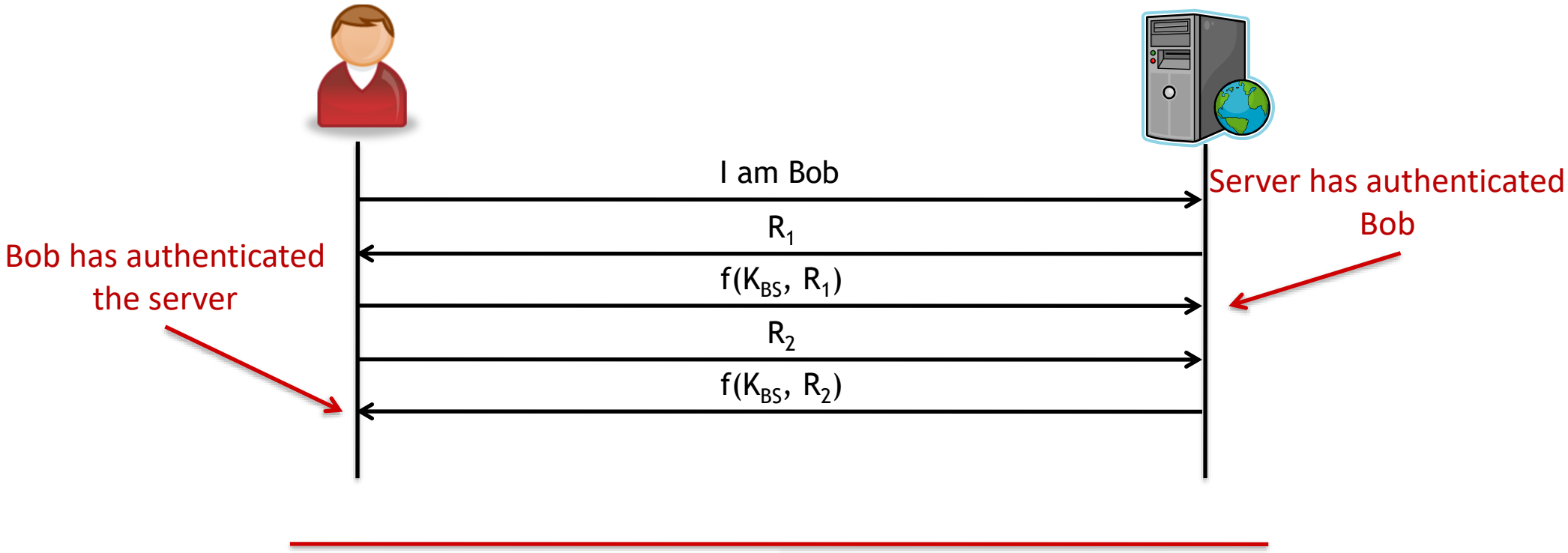
Interesting note: No more sensitive databases of user passwords or shared secrets!

What is the problem with these protocols?

Mutual Authentication

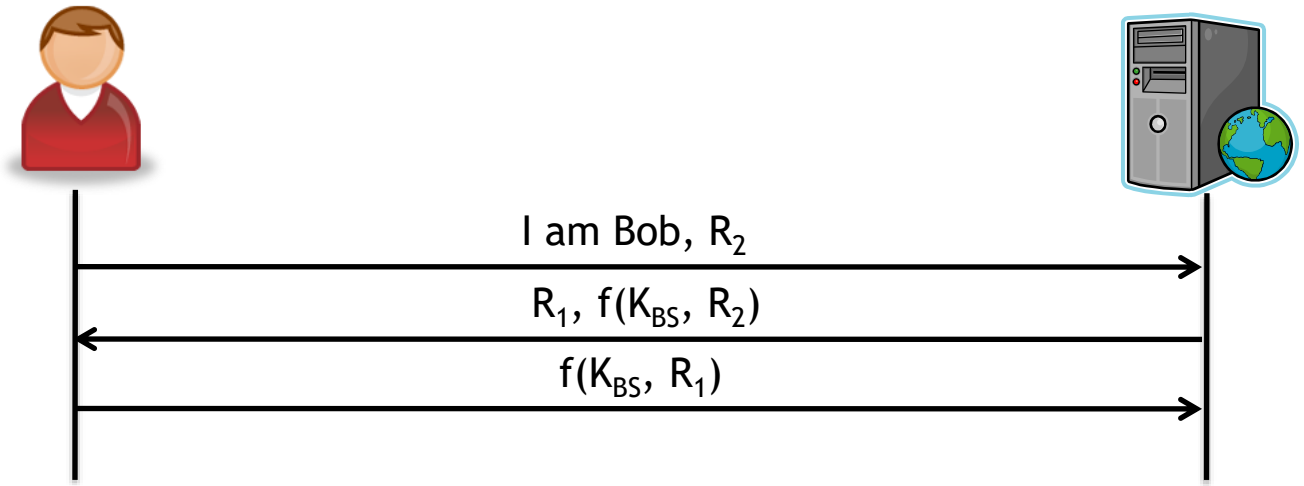
Often times, both participants in a protocol want to authenticate one another

One way to do this is to (essentially) run two invocations of our earlier protocols:



This seems like a lot of messages, doesn't it? Can't we optimize this thing somehow?

A stab at optimization…



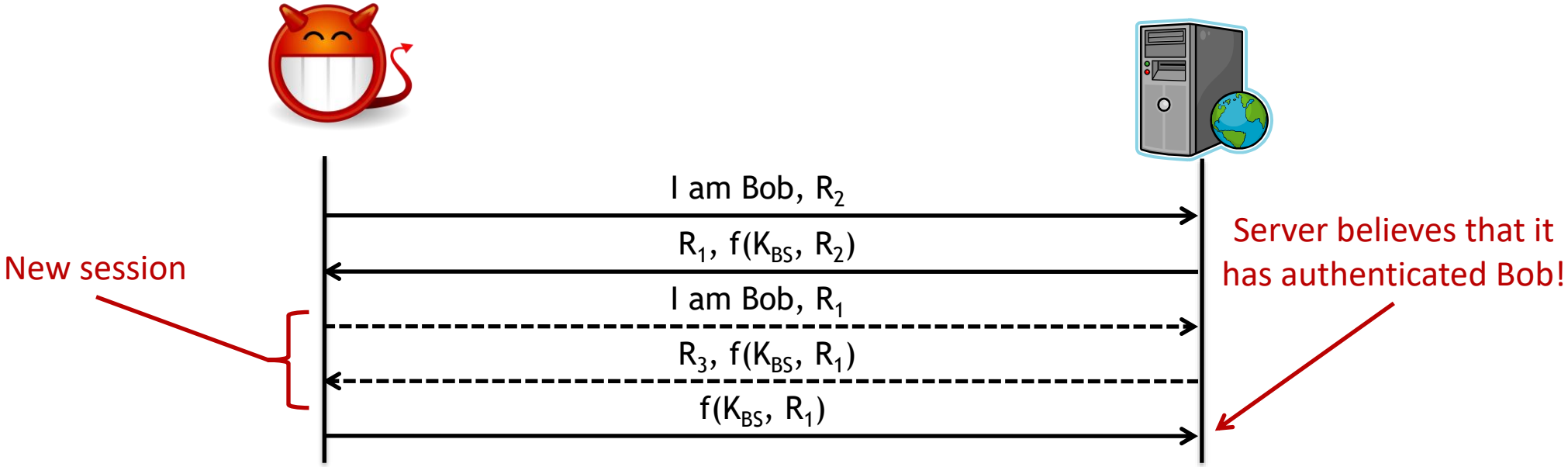
In theory:

- Bob has authenticated the server after message 2
- The server has authenticated Bob after message 3

In practice: This isn't actually the case!

This protocol is vulnerable to what is known as a reflection attack

This opens the door to what is known as a reflection attack



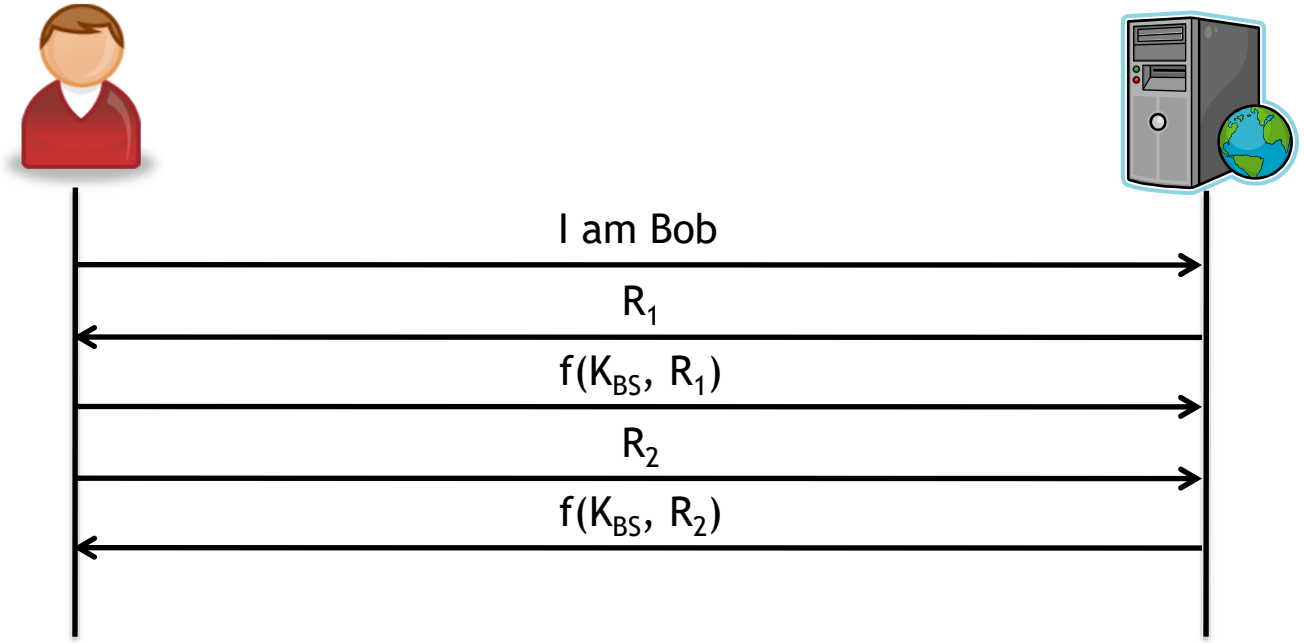
Why is this attack possible?

- Both parties do exactly the same thing
- No way to prevent "reflecting" challenges back

However, this weakness is easily avoided

- Use different keys in each direction
  - e.g., K_{BS} could be XORed with different constants in each direction
- Force challenges to encode direction of transmission
  - e.g., use (<sender name> || R) instead of R alone

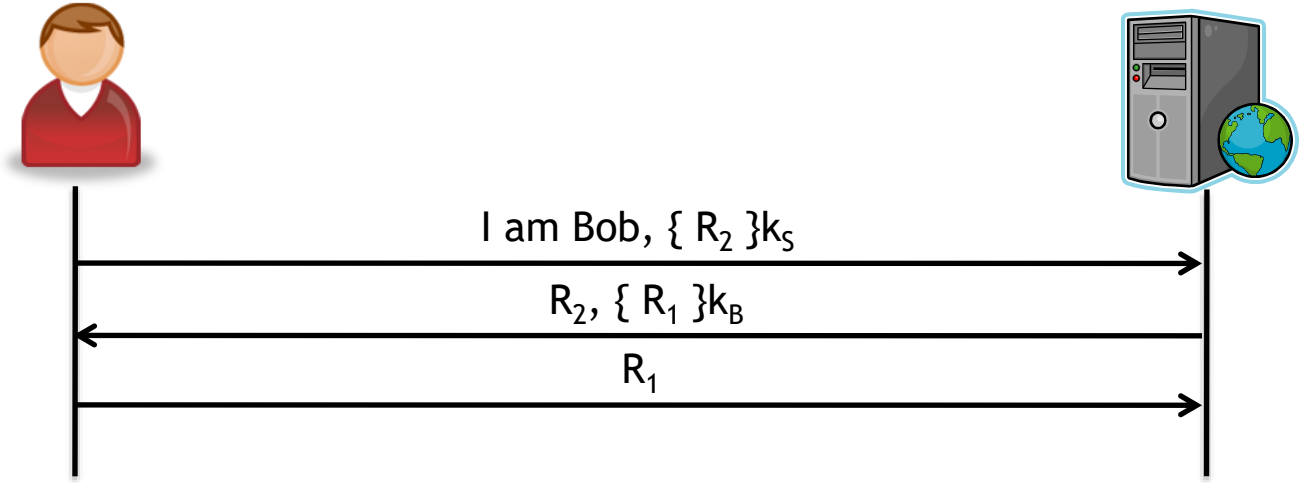
Why isn't the long protocol vulnerable to this attack?



Answer: Bob needs to prove his identity to the server prior to the server proving its identity to Bob!

Lesson: Be careful when "optimizing" security protocols…

Mutual authentication protocols can also be constructed using public key cryptography



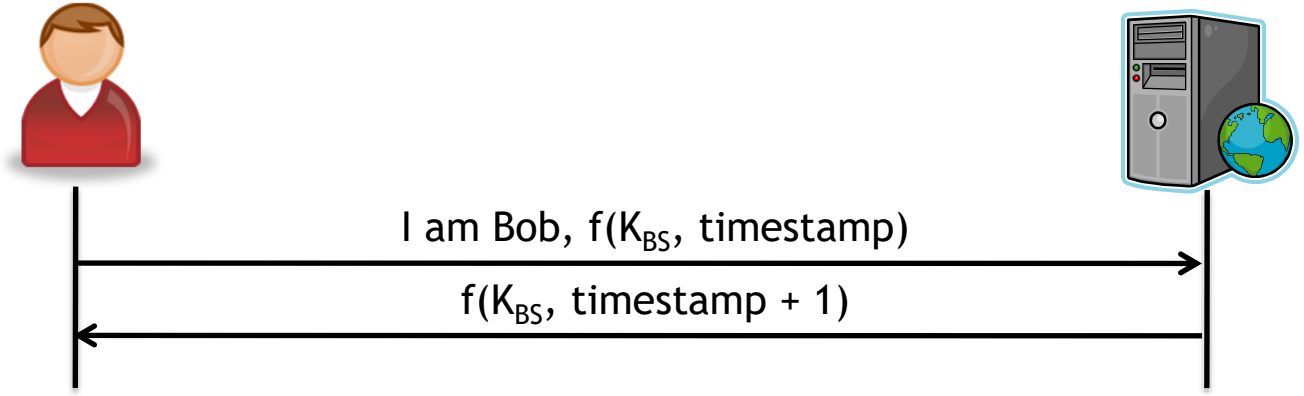
This looks a lot like our "optimized" secret key mutual authentication protocol, doesn't it?

Question: Does this protocol suffer from the reflection attack, too?

Note: This protocol assumes that Bob and the server know each others' public keys a priori. This could be done via:

- Acquisition from a trusted certificate authority
- Encrypt public keys with a symmetric key derived from a shared password
- Offline configuration (think SSH)

We can further reduce the number of messages by using timestamps instead of random challenges



This protocol is nice, as it fits within a two-message exchange

- Request/reply exchanges
- RPC invocations
- Etc.

Question: Why do we need timestamps for this to work?

- Freshness! (Provided that clocks are synchronized)

Question: Why does the server return f(K_{BS}, timestamp + 1)?

- Protection against replay attacks!

Integrity/Encryption Setup Protocols

Often times, authentication is just not enough…

After authenticating, it is often necessary to protect the integrity and/or confidentiality of the rest of the conversation

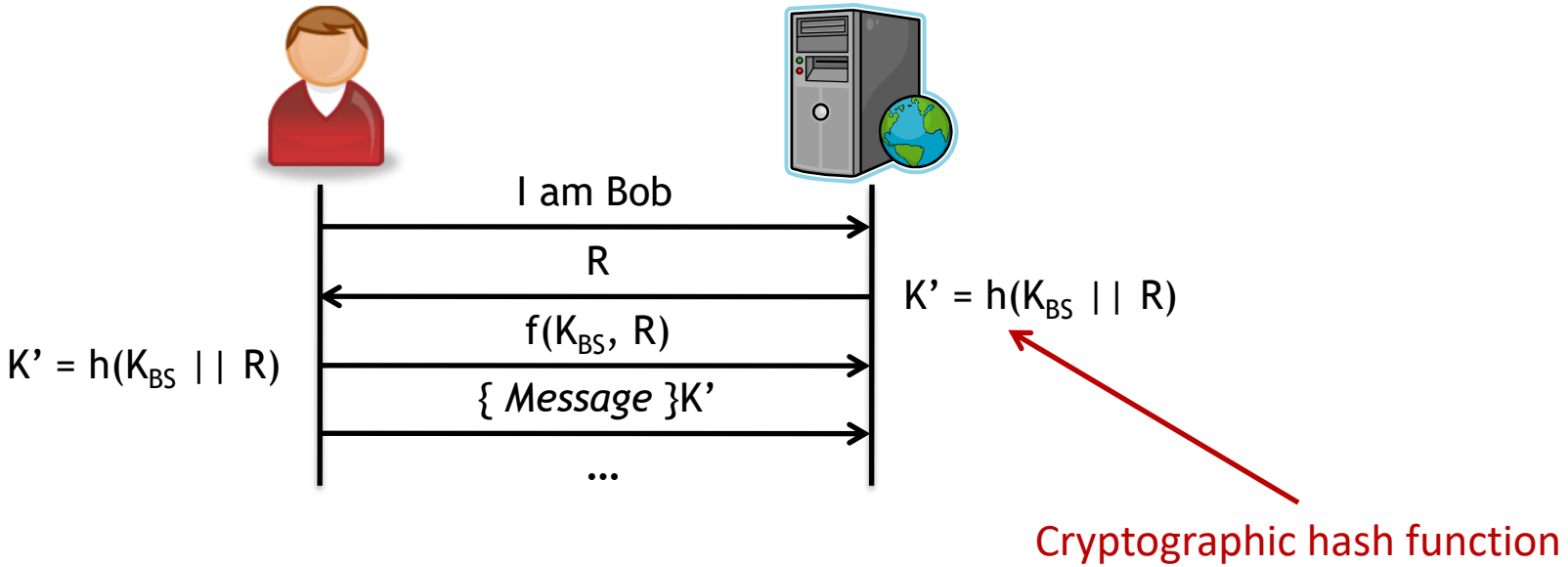
Example: Telnet versus SSH

But wait, don't all of the authentication protocols that we've talked about require shared keys anyway?!?

It turns out that generating fresh keys regularly is important

- Overuse can make long term secret keys easier to break
- Per-session keys limit replay/injection attacks to a single session
- Forward secrecy of individual sessions
- …

This turns out to be fairly simple to do!



Interesting notes:

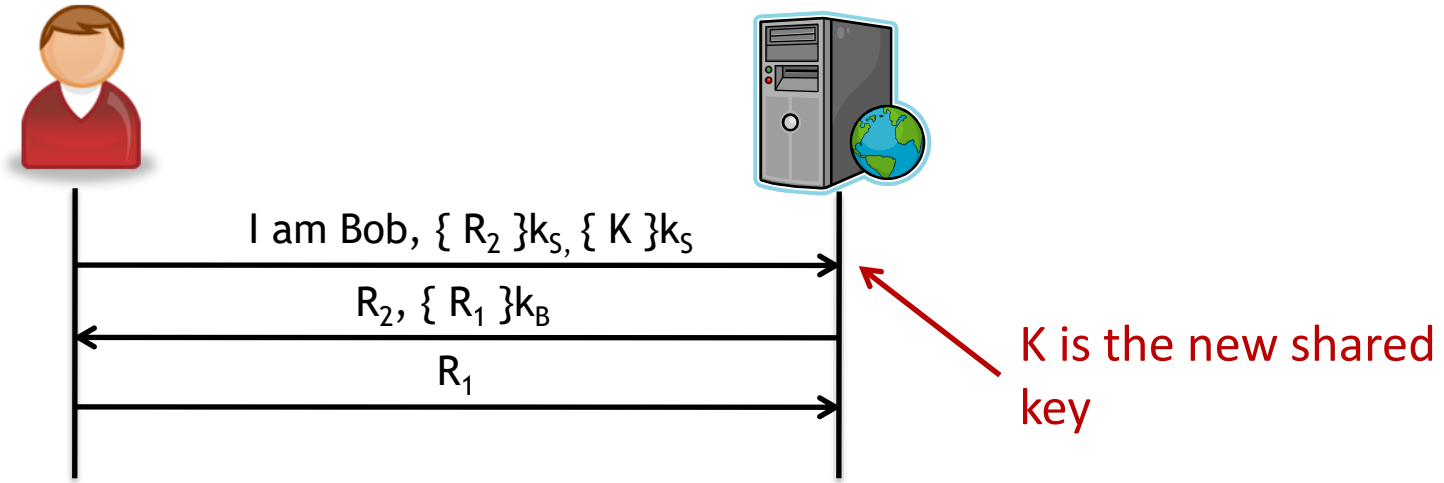
- This protocol does not add any messages to our authentication exchange!
- Even though R is visible to the adversary, K' cannot be guessed (Why?)
- If the session key K' is leaked, the long term secret K_{BS} is still safe (Why?)

Mutual authentication protocols can be adapted in a similar manner

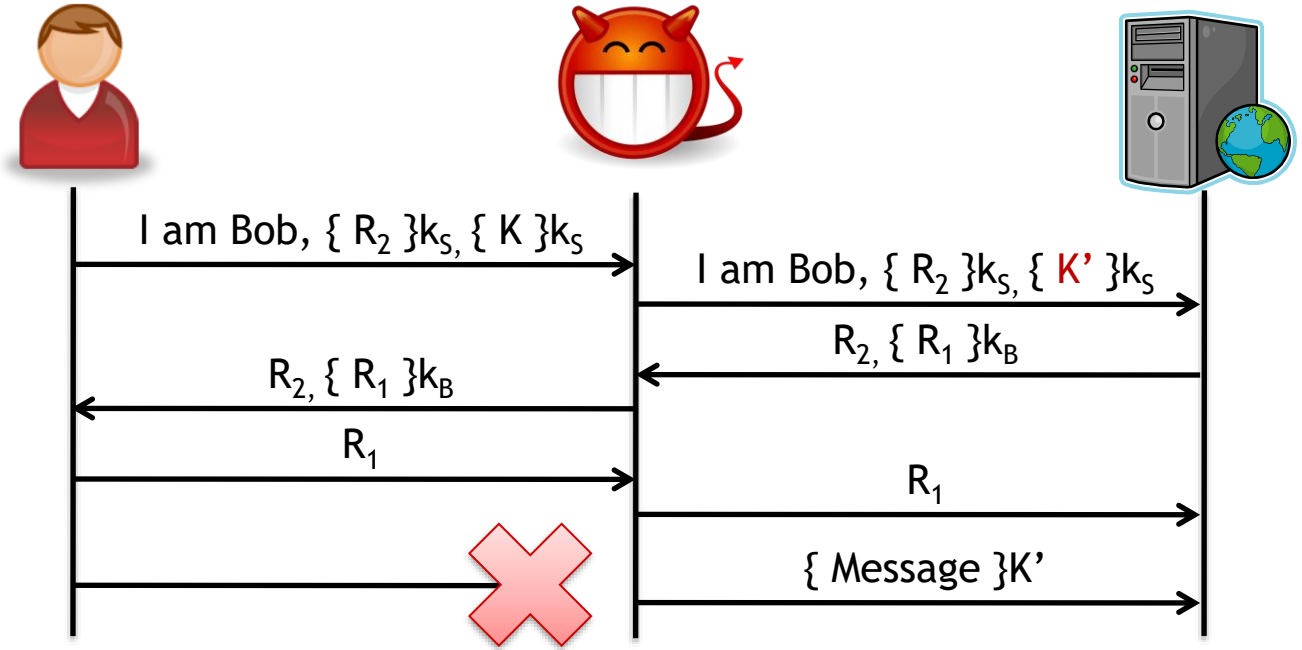
- E.g., $K' = h(K_{BS} || R_1 || R_2)$
- Note that order of Rs is important!

We can also derive session keys using public-key authentication protocols

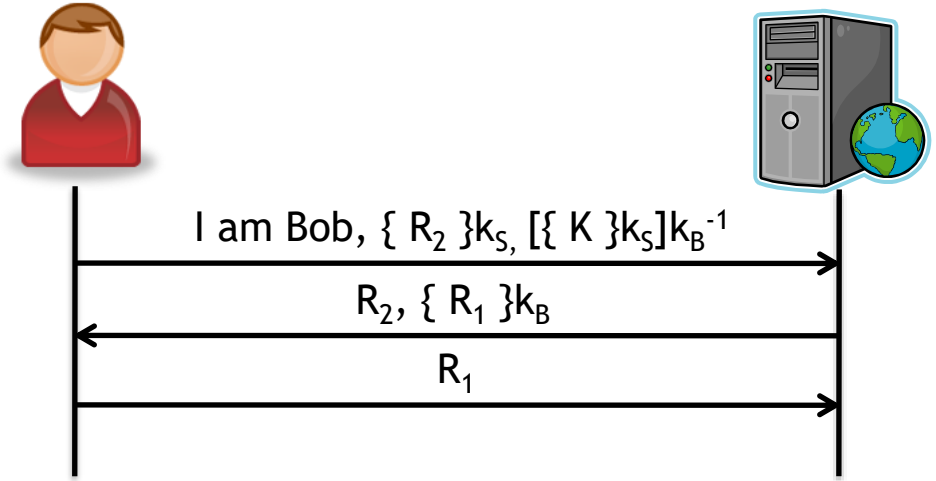
Initial attempt:



Unfortunately, this protocol can be hijacked!



Digital signatures can help us fix this problem!



Why does this work?

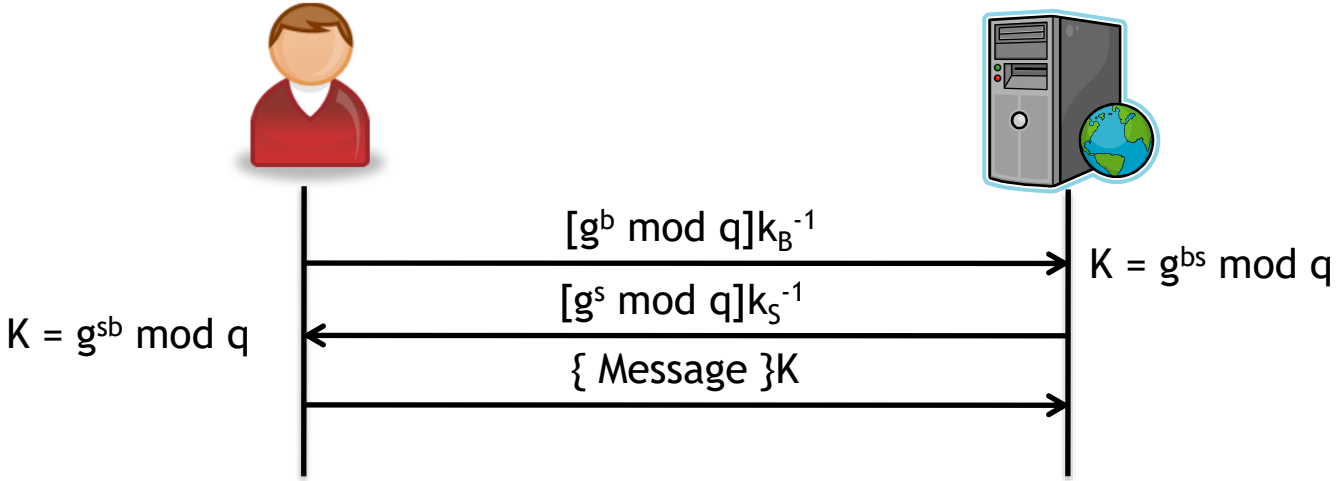
- The signature ensures that the key was actually generated by Bob
- This provides a binding between the authentication protocol and the key exchange protocol

However, there are still issues with this protocol…

Assume that the server is eventually compromised

- This means that the adversary learns k_S^{-1}
- If the adversary recorded the above exchange, K can be recovered!

A signed Diffie-Hellman key exchange prevents this problem



As we learned earlier, the Diffie-Hellman exchange allows Bob and the server to agree on a shared secret key over a public channel

Even if both parties are later compromised (i.e., k_B^{-1} and k_S^{-1} are revealed) data encrypted with K is safe! (Why?)

Question: Why are the digital signatures needed?

What happens if we need to generate more than one shared key?

The previous protocols only provide us with a single shared key. How can we derive multiple keys?

One method is to run these protocols multiple times

- This is expensive for the participants
- Fortunately, this is unnecessary!

Case study: Key derivation in SSH

- The SSH protocol uses a Diffie-Hellman exchange, and computes
  - A shared key K
  - An exchange hash value H
- Client to server encryption key: h(K || H || “C” || session_id)
- Client to server integrity key: h(K || H || “E” || session_id)

Question: Why is the above safe to do?

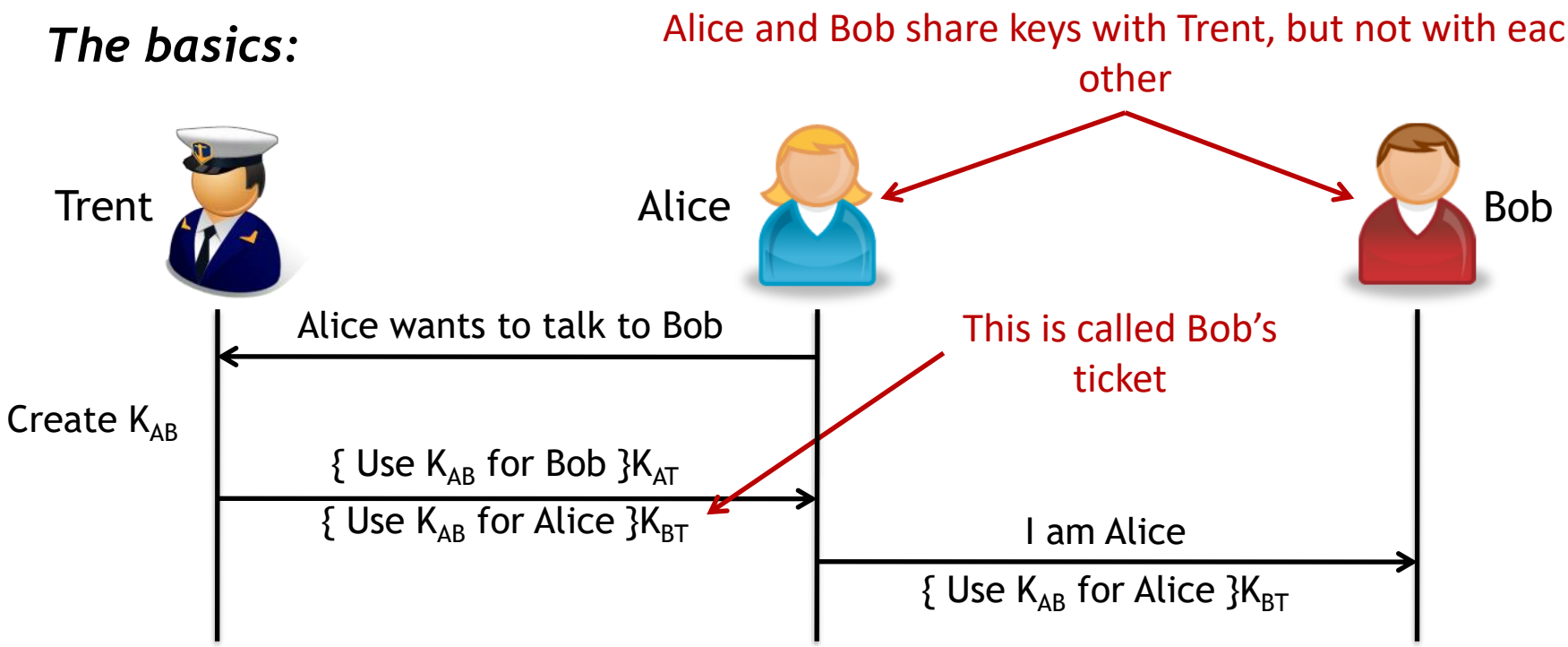
Mediated Authentication Protocols

What if we like the speed of symmetric key cryptography, but not the key management headaches?

Specifically, how can we let two users securely establish a shared key?

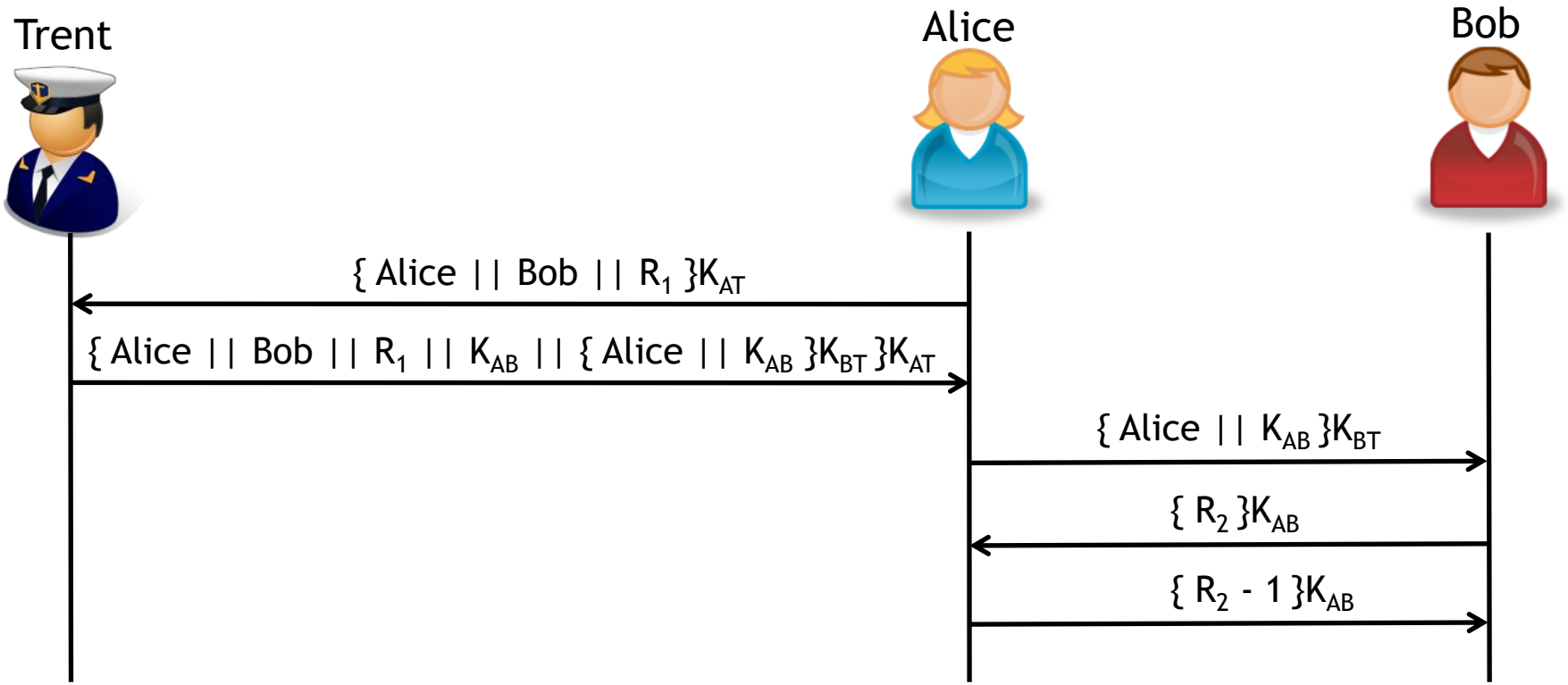
Mediated authentication protocols make use of a trusted mediator, or key distribution center (KDC), to make this possible!

The basics:



Note: This protocol is incomplete, as it does not authenticate Alice and Bob to one another. However, it is a good place to start…

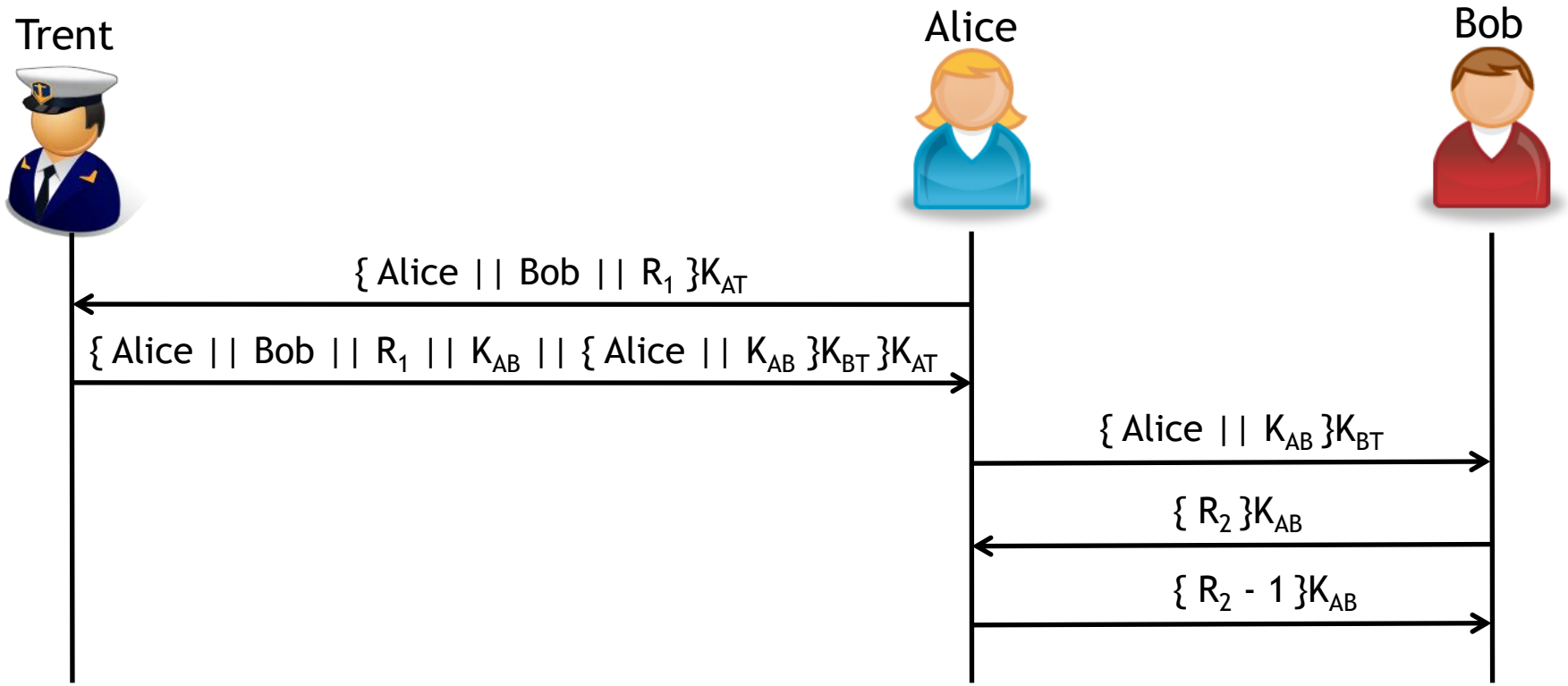
The Needham-Schroeder protocol is a well-known mediated authentication protocol



Note: R_1 and R_2 are called nonces

- Must be generated at random (unpredictable)
- Cannot be used in more than one protocol execution

Why does Needham-Schroeder work?



After message 2 Alice

- Knows that this message is fresh
- Knows that the session key is to be shared with Bob

After message 3, Bob knows that he has a shared key with Alice

After message 5, Bob knows that this key is fresh (Why?)

The Needham-Schroeder protocol assumes that all keys remain secret

Assume that Eve intercepts the message { Alice || K_{AB} }K_{BT} and later learns the session key K_{AB}

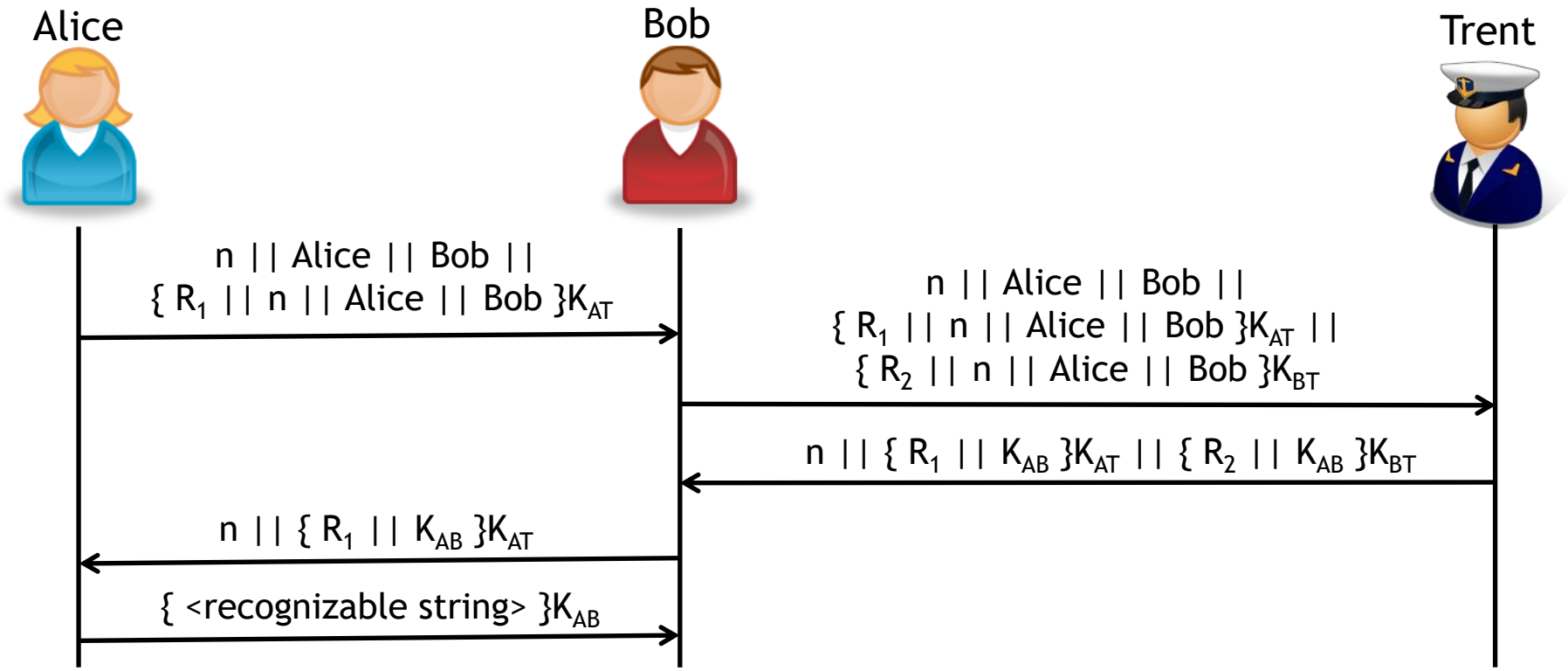
Question: *Why might a session key become compromised?*

Eve can now launch a replay attack!

- Eve can replay the message { Alice || K_{AB} }K_{BT}
- She can intercept Bob's response { R_3 } K_{AB} to Alice
- Since Eve knows K_{AB}, she can decrypt this message and reply { R_3 - 1 } K_{AB}

How can we defend against this type of attack?

The Ottway-Rees protocol prevents this attack



Properties of this protocol:

- After message 3, Bob knows that he has a fresh session key to share with Alice that was generated by Trent
- After message 4, Alice knows that she has a fresh session key to share with Bob that was generated by Trent
- After message 5, Bob knows that Alice received the shared key

Why doesn't a compromised session key subvert this protocol?

Assume that Eve:

- Records the message $n \,||\, \{ R_1 \,||\, K_{AB} \}K_{AT} \,||\, \{ R_2 \,||\, K_{AB} \}K_{BT}$
- Breaks the key K_{AB}

Now, say that Eve tries to forge a version of message 4 to Alice

- $n \,||\, \{ R_1 \,||\, K_{AB} \}K_{AT}$

If Alice does not have an ongoing exchange with Bob, this forgery will fail

- Why? No initial state saved

If Alice does have an ongoing exchange with Bob

- The forgery will fail if the number n does not match
- If n does match, the forgery will fail because Alice will be using a different nonce R_1

Summary So Far …

So far, we've learned about four types of handshake protocols:

- Login only protocols
- Mutual authentication protocols
- Integrity/encryption setup protocols
- Mediated authentication protocols

These protocols are very simple, but very sensitive to changes

Understanding the types of attacks that these protocols can be subjected to is a very important facet of designing secure networked systems

Next: Strong password protocols

Strong Password Protocols

Now, we'll focus on strong password protocols

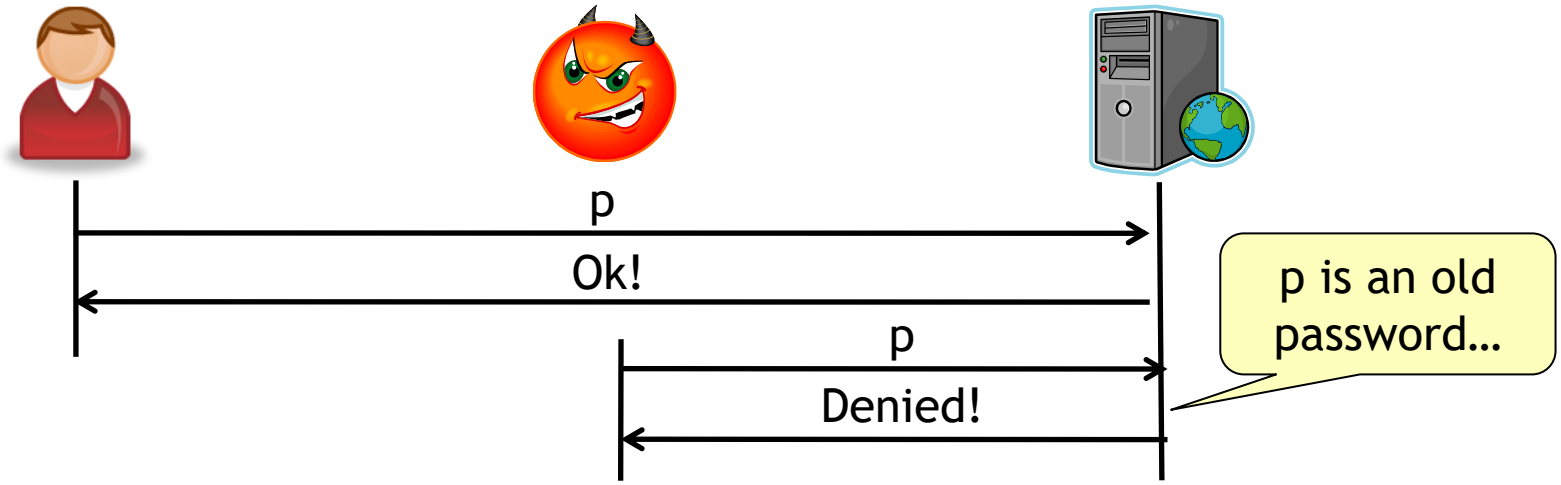
In particular, we'll look at

- Lamport's hash-based one-time password scheme
- Encrypted Key Exchange (EKE)
- Secure Remote Password (SRP)
- Secure credential download protocols

As we'll see, these protocols allow us to leverage weak passwords into strong cryptographic protocols

One problem with password-based systems is that if the password is ever observed, it is compromised

In a one-time password scheme, passwords are invalidated after use



Clearly, this prevents impersonation attempts by a passive adversary

However, these systems come at a cost

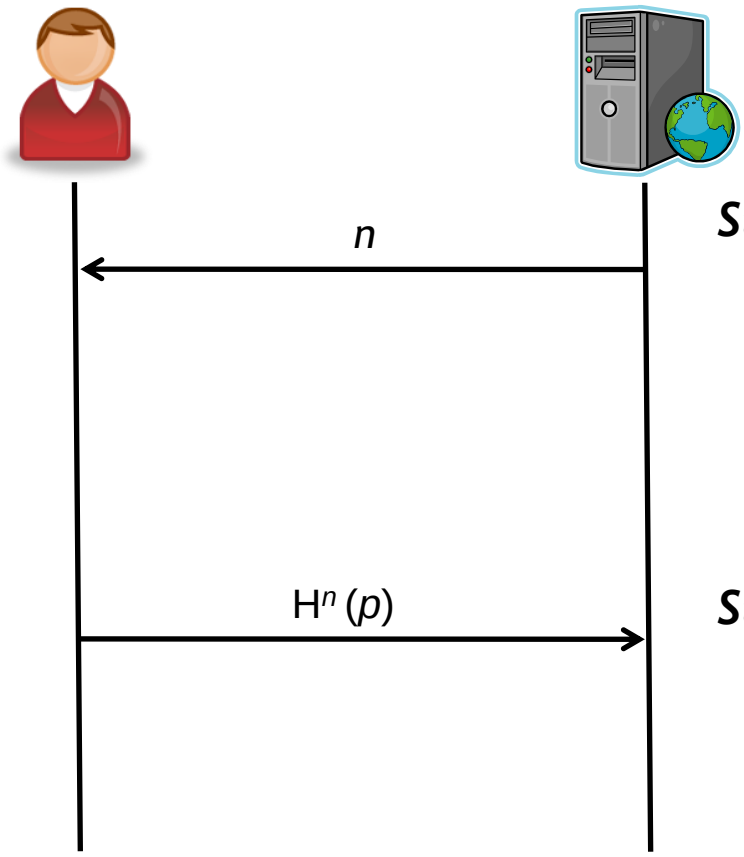
- Do you *really* expect users to memorize a list of passwords?
- Will this require that the server stores tons of state for each user?
- …

It turns out that these types of systems are actually quite easy to deploy!

Leslie Lamport developed a one-time password scheme that uses hash chains

Leslie Lamport, "Password Authentication with Insecure Communication," Communications of the ACM 24(11):770-772 November 1981.

Setup Phase



Step 1:

- Choose a large n (e.g., n = 1000)
- Send n to the user

Step 2:

- Choose a password p
- Compute $H^n(p)$
- Send $H^n(p)$ to server
- Store (p, n)

Step 3:

- Store $(n, H^n(p))$

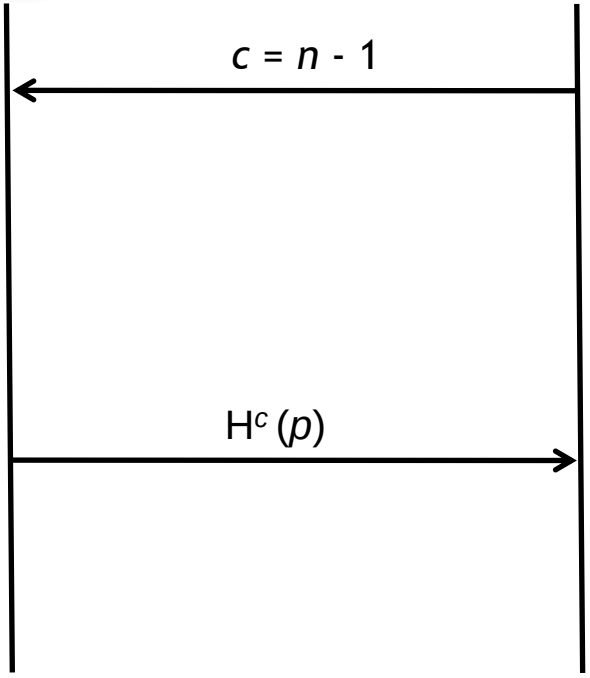
Notation

- H is a hash function
- $H^n(p)$ represents n applications of H to p
- E.g., $H^2(p) = H(H(p))$

Using Lamport's OTP Scheme

~~Storage: (p, n)~~

~~Storage: $(n, H^n(p))$~~



Step 1:

- Send c = n - 1 to user

Step 2:

- Verify that $c < n$
- Compute $H^c(p)$
- Send x = $H^c(p)$ to the server
- Store (p, c)

Step 3:

- Verify that H(x) = $H^n(p)$
- Store $(n - 1, H^{n-1}(p))$

Storage: (p, c)

Storage: $(n\text{-}1, H^{n-1}(p))$

Why is this scheme safe?

To prove the safety of these scheme, we need to show that knowing an old (challenge, response) pair does not help the attacker

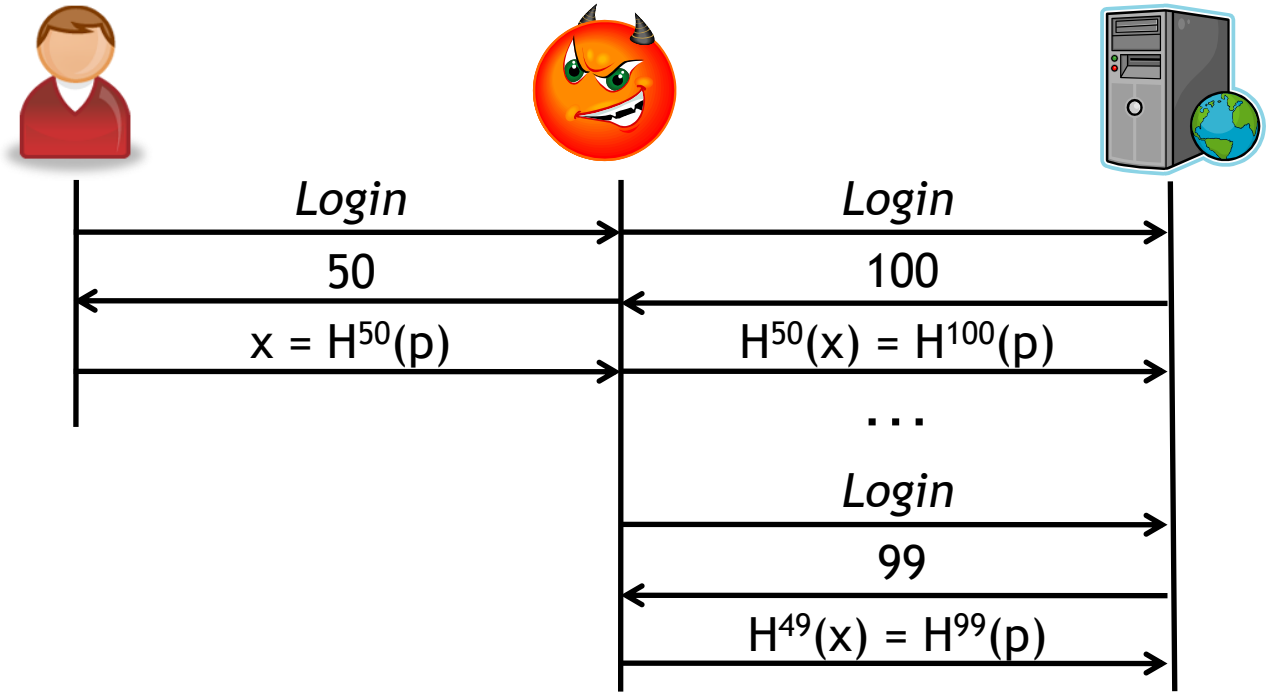
Attack 1: The adversary attempts to use an old (challenge, response)

- The user will never accept an old challenge
- The server will never request an old challenge (*n* decremented after use)

Attack 2: Derive the k^{th} password from the $k\text{-}1^{st}$ password

- Assume the $k\text{-}1^{st}$ password is $H^m(p)$
- This means that the k^{th} password will be $H^{m-1}(p)$
- To guess the k^{th} password, we need a value v such that $H(v) = H^m(p)$
  - That is, we need to find the preimage of $H^m(p)$
- The *preimage resistance* property of H means that this is infeasible

Lamport's scheme is not secure against an active attacker (man in the middle)



The adversary does not know p, but can impersonate Bob anyway!

Question: Can we simply require that challenges decrement by 1?

- What about packet loss?
- Failed login attempts by others?

In short, this system will only work if our deployment environment assumes that there are no active attackers

Strong Password Protocols

Strong password protocols are designed to prevent both passive and active attackers from gaining enough information to conduct an offline password cracking attempt

This class of protocols was first proposed by Bellovin and Merritt

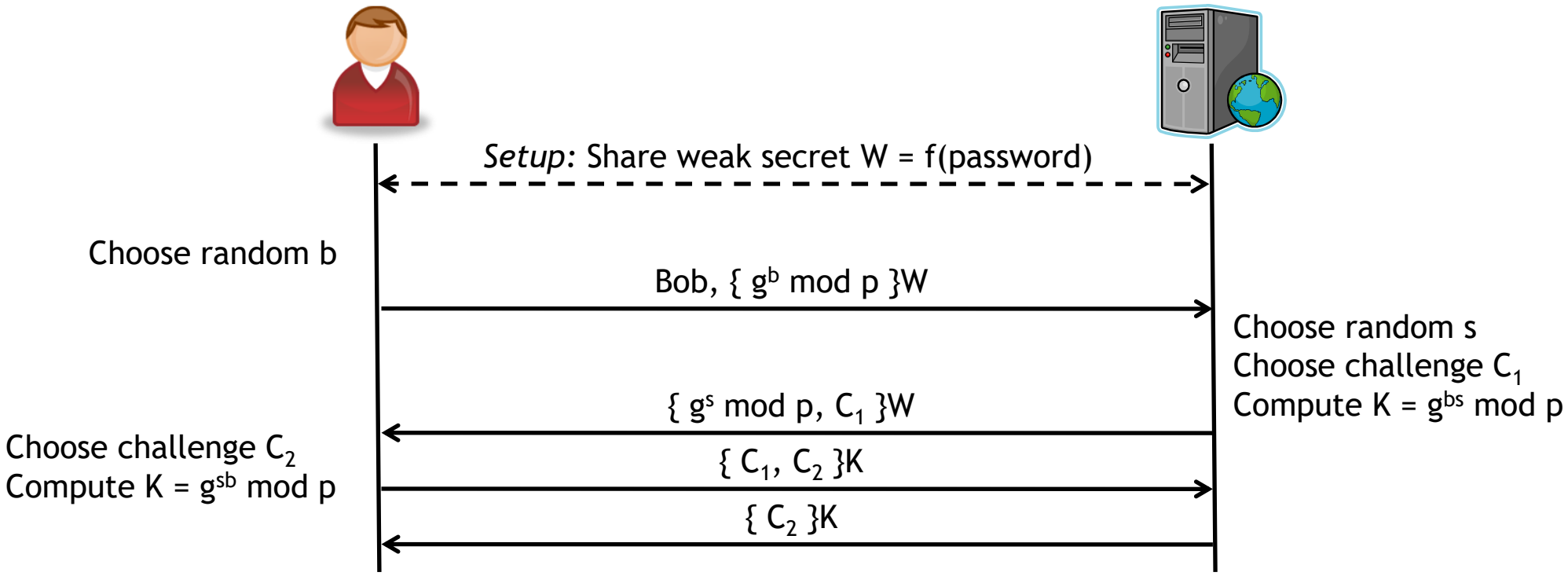
- Encrypted key exchange (EKE)

At a high level this protocol works as follows:

- Bob and the server share some weak secret (i.e., a password)
- Both parties carry out a Diffie-Hellman key exchange
  - Messages encrypted using the weak secret
- Mutual authentication occurs using the D-H key

This works because the whole exchange essentially looks random to outside observers!

EKE in Detail



How does this protocol prevent offline password guessing?

- Decrypting { g^b mod p }W using the wrong secret gives a randomized output
- Further, g^b mod p is essentially a random number mod p
- As a result, the result of properly decrypting { g^b mod p }W also looks randomized if b is unknown
- Result: There's no way to "check" whether W' = W for a password guess W'

Interestingly, our choice of modulus p can actually make it possible for adversaries to attack this protocol!

Observation: $g^b \bmod p < p$ by the definition of "mod"

If an adversary decrypts $\{ g^b \bmod p \}W$ using a guess W' and obtains a value greater than p, then W' is certainly not the correct secret

This could be a problem if p is slightly greater than a power of 2

Why? Assume p slightly bigger than 2^n

- The binary representation of p requires n+1 bits
- Since $2^{n+1} = 2 \times 2^n$, this bit field can hold (roughly) two times as many values as are actually needed by the protocol
- As such, a random decryption has (roughly) a 1 in 2 chance of being greater than the value p

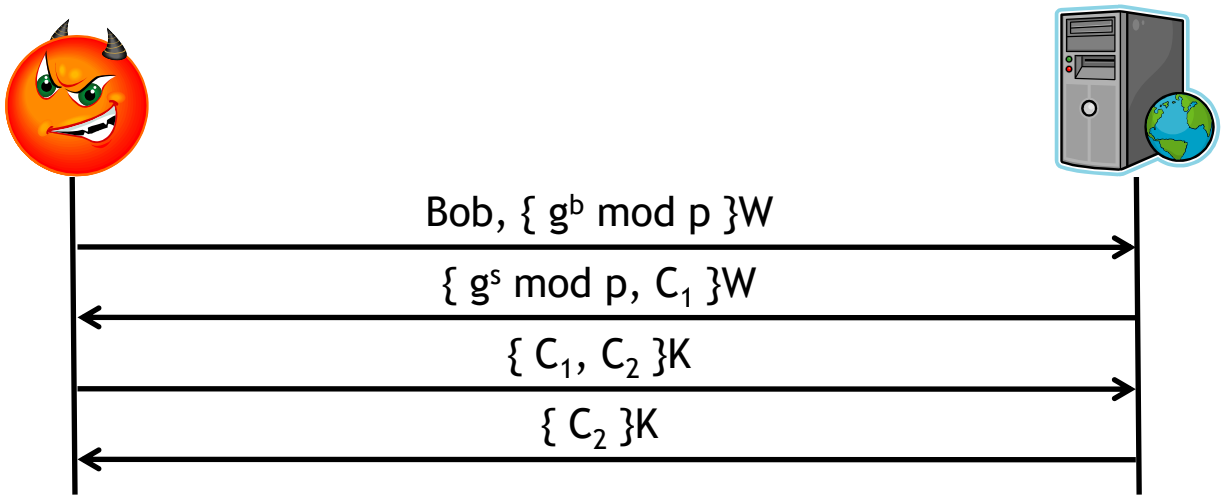
What's the fix? Choose a p that is slightly less than a power of 2!

What happens if the server is compromised?

For EKE to work, the server needs to store a list of ⟨user, W⟩ bindings

If the server is compromised, the adversary can impersonate any user!

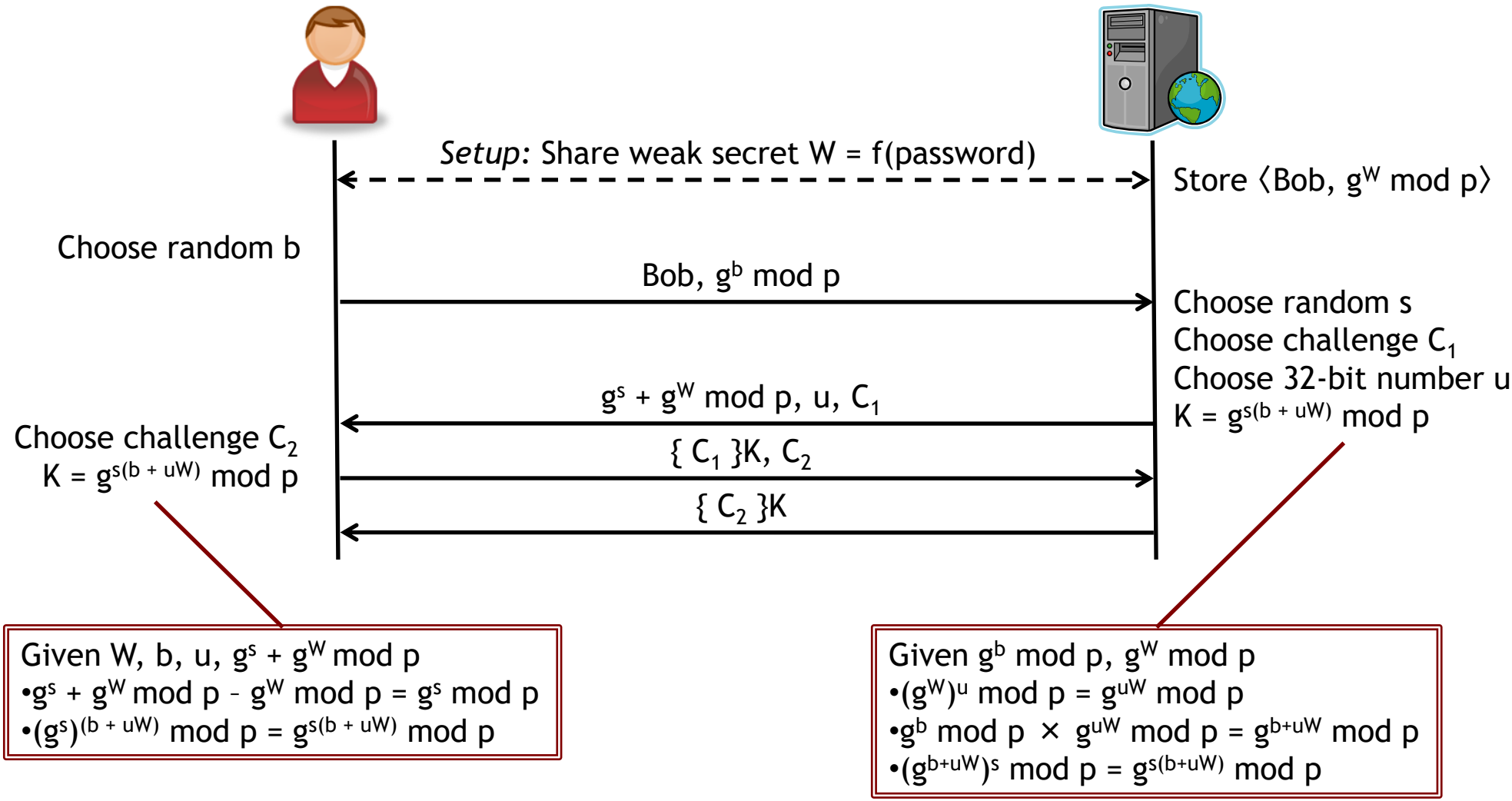
- The adversary doesn't know the password, but they don't need it
- W is all that is needed to authenticate!



Ideally, this shouldn't happen…

New property: Compromising the server should still require the adversary to launch a dictionary attack to recover W

The Secure Remote Password (SRP) protocol provides us with this assurance



Question: Why does SRP force the adversary to launch a dictionary attack?

Aren't passwords old technology? Why are we learning about this?

Asymmetric key cryptography seems like a much cooler solution… Can't we just use this for authentication?

For this to work, we need to manage public and private keys

- Public keys can be stored publicly, so this is no problem
- Where do we keep our private keys?

What if I need to use multiple machines? Replicating secrets is bad. Plus, I don't trust my administrators.

Private keys can be stored in a number of ways

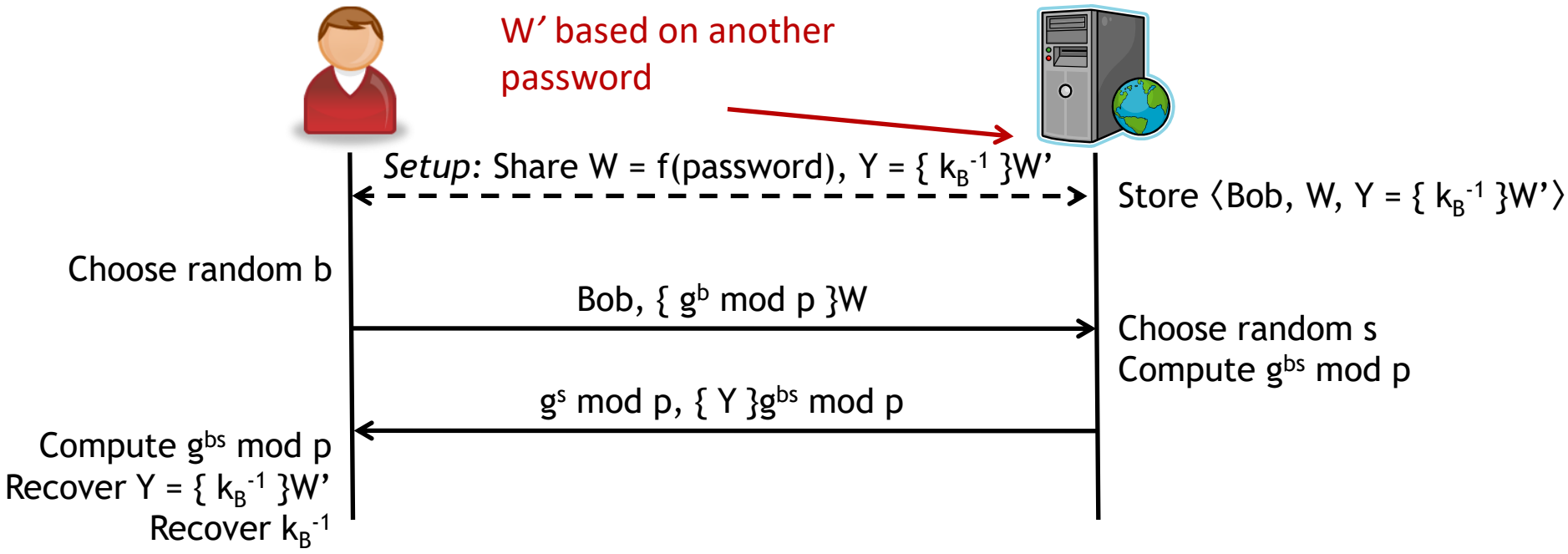
- On the local machine, protected by the file system's permission settings
- On a smart card or some other token
- On a trusted server

What if I lose or break my token?

What if the server is compromised? What if I don't trust my trusted server?

Strong password protocols can help us solve the private key storage conundrum in a robust fashion

We can safely store our private keys so that even if the server is compromised, the key is not leaked!



Note that the server never finds out whether Bob knew the password W

- Why? Bob never talks to the server after retrieving the encrypted key!

An attacker impersonating Bob cannot launch an offline attack against

- Message 1 commits the attacker to a single password guess
- If this guess is incorrect, the rest of the math fails to work!

Question: Why does Bob need to use two different passwords?

Summary So Far …

Although passwords are ancient technology, they are still widely used

So far we have discussed

- One-time password schemes that are resilient to eavesdropping attacks
- Strong password protocols that prevent offline password guessing attacks
- Hardened versions of these protocols that are also resilient to server compromise
- Secure credential retrieval protocols that allow us to use passwords to protect stronger cryptographic secrets like private keys

In the end, we'll probably never fully get rid of passwords ☹

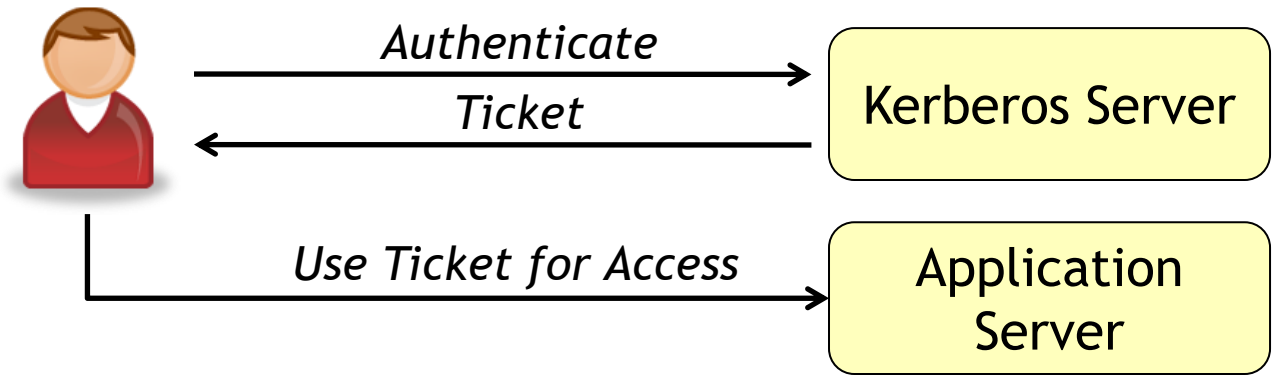
At least these protocols allow us to use passwords in a safer manner ☺

Next: Kerberos

Kerberos Overview

This is kind of like our term project, yeah?

Kerberos is a mediated authentication protocol



This protocol is based on the following assumptions:

- Server(s) used by Kerberos are highly secured (How?)
- Application servers are moderately secure, though may be compromised
- Client machines are untrusted

Kerberos uses secret key cryptography to allow users to authenticate to networked services from any location

Kerberos Design Goals

Main goal: Breaking into one host should not help the attacker compromise the overall security of the system

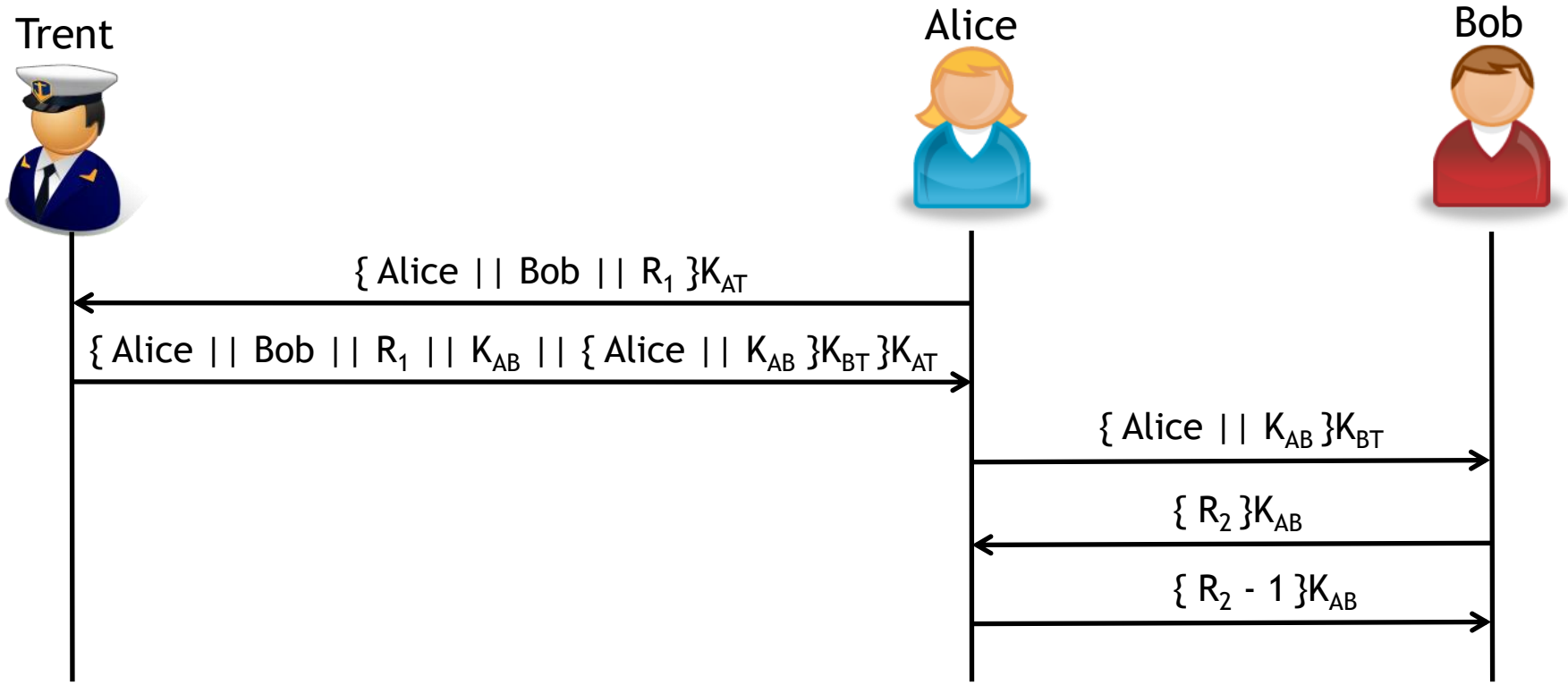
Client authenticator goals:

- Users cannot remember cryptographic keys, so keys should be derived from the user's password
- Passwords should not be sent in cleartext
- Passwords should not be stored on the server
- The client's password should be exposed as little as possible

The use of Kerberos should require only minimal modifications to existing applications

So, how does this work?

Kerberos is based on the Needham-Schroeder secret key authentication protocol



After message 2, Alice

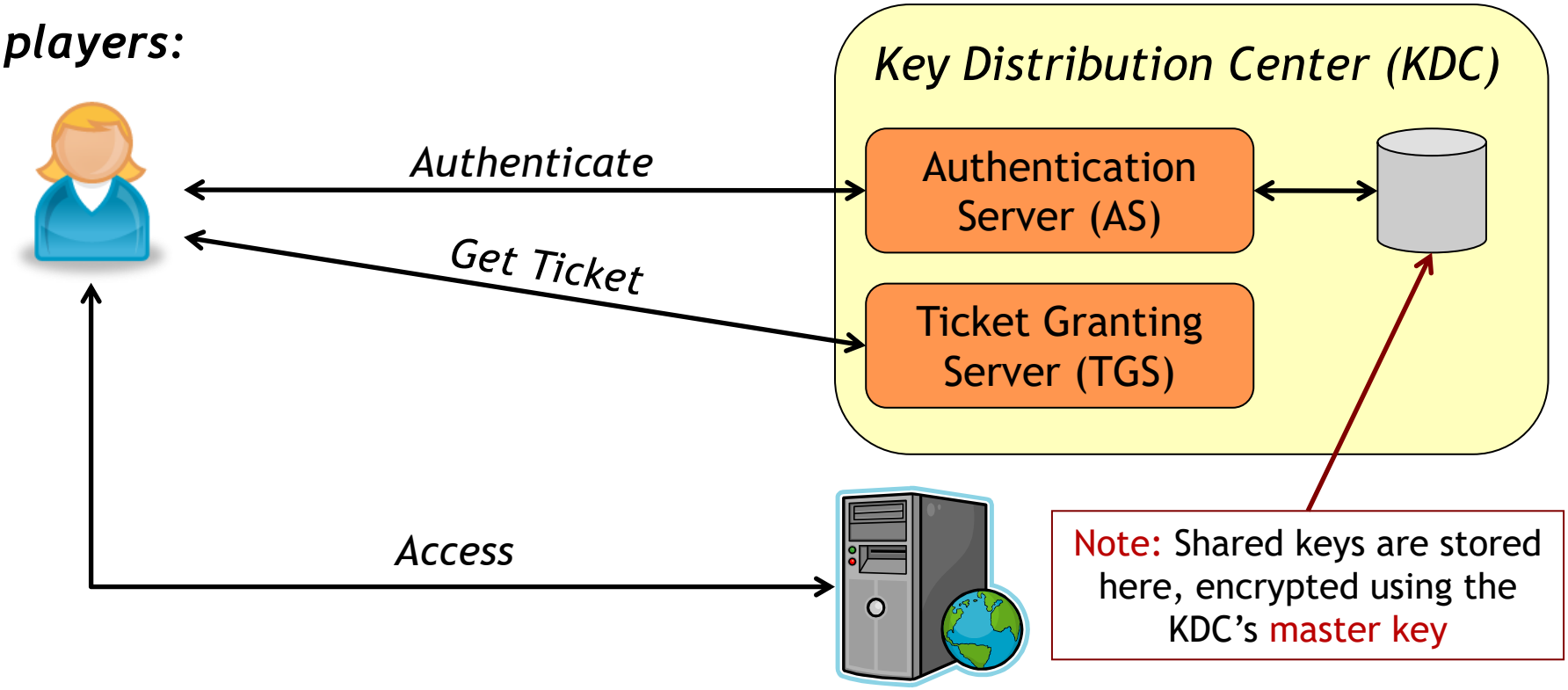
- Knows that this message is fresh
- Knows that the session key is to be shared with Bob

After message 3, Bob knows that he has a shared key with Alice

After message 5, Bob knows that this key is fresh

Kerberos v4: The Basics

The players:

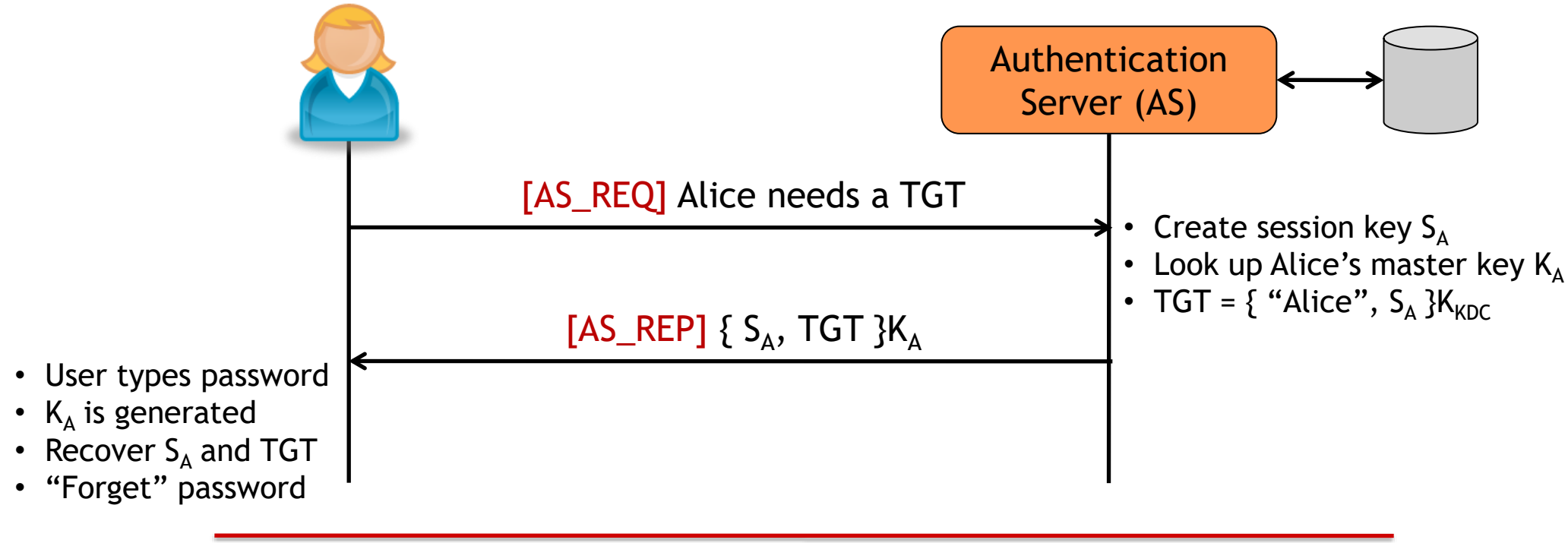


All principals in the system share a secret key with the AS

- User keys derived from their password
- Service keys are random cryptographic keys

The cryptographic algorithm used by Kerberos is (Triple) DES

Step 1: Obtaining a ticket-granting ticket (TGT)



The above process is used to initiate a login session

- Password used to initiate the session
- S_A is used for subsequent exchanges

Note that the user password is not needed until after the TGT is obtained

- Why is this a good thing?
- Why is this a bad thing?

What is the purpose of this process?

In a single session, a user may want to access many different machines

For example…

- Download a file from a secured web server
- SSH into another machine to compare results with experimental data
- Email a colleague to inquire about an oddity in the file
- FTP an updated file to the secured server
- …

By obtaining a single session key S_A, Alice only uses her password once!

- This minimizes the amount of time that the password is exposed
- If S_A is cracked, the password is still safe

Furthermore, the TGT frees the KDC from maintaining any state

- Recall that TGT = {"Alice", S_A }K_{KDC}
- No need to track S_A at the server, just ask Alice for her TGT

Obtaining a TGT: Message Detail

AS_REQ

# Bytes	Content
1	Version of Kerberos (`4`)
1	Message Type (`1`)
≤ 40	Alice's name
≤ 40	Alice's instance
≤ 40	Alice's realm
4	Alice's timestamp
1	Desired ticket lifetime
≤ 40	Service name (`krbtgt`)
≤ 40	Service instance

In multiples of 5 minutes (up to about 21.5 hours)

Helps Alice match request/reply pairs

Note: This message is sent unencrypted

Obtaining a TGT: Message Detail

AS_REP

Bytes	*Content*
1	Version of Kerberos (4)
1	Message type (2)
≤ 40	Alice's name
≤ 40	Alice's instance
≤ 40	Alice's realm
4	Alice's timestamp
1	Number of tickets (1)
4	Ticket expiration time
1	Alice's key version number
2	Credentials length
var	Credentials

Bytes	*Content*
8	S_A
≤ 40	TGS name
≤ 40	TGS instance
≤ 40	TGS realm
1	Ticket lifetime
1	TGS key version number
1	Length of ticket
var	Ticket
4	Timestamp
var	Padding of 0s

Note: The "Credentials" field is encrypted with the Alice's master key

Obtaining a TGT: Message Detail

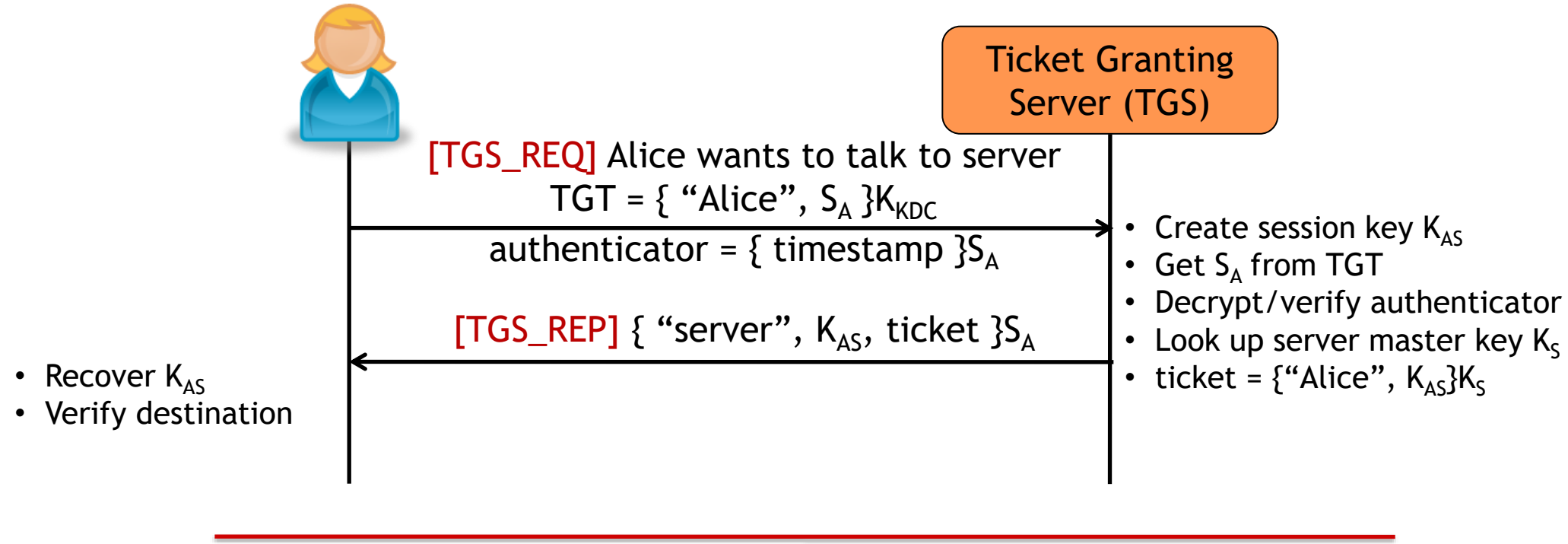
This is the credentials field…

Bytes	*Content*
8	S_A
≤ 40	TGS name
≤ 40	TGS instance
≤ 40	TGS realm
1	Ticket lifetime
1	TGS key version number
1	Length of ticket
var	Ticket
4	Timestamp
var	Padding of 0s

Bytes	*Content*
≤ 40	Alice's name
≤ 40	Alice's instance
≤ 40	Alice's realm
4	Alice's IP address
8	Session key S_A
1	Ticket lifetime
4	KDC timestamp
≤ 40	TGS name
≤ 40	TGS instance
var	Padding of 0s

Note: The "Ticket" field is encrypted with the TGS's master key

Step 2: Obtaining a service ticket



Interesting notes:

- Alice did not user her password to authenticate!
- The TGS did not need to maintain any state to verify Alice's identity

The authenticator attests to the freshness of the current exchange

- This means that Kerberos requires synchronized clocks (usually ~5 mins)
- Not actually needed in this exchange (Why?)

Obtaining a Service Ticket: Message Detail

TGS_REQ

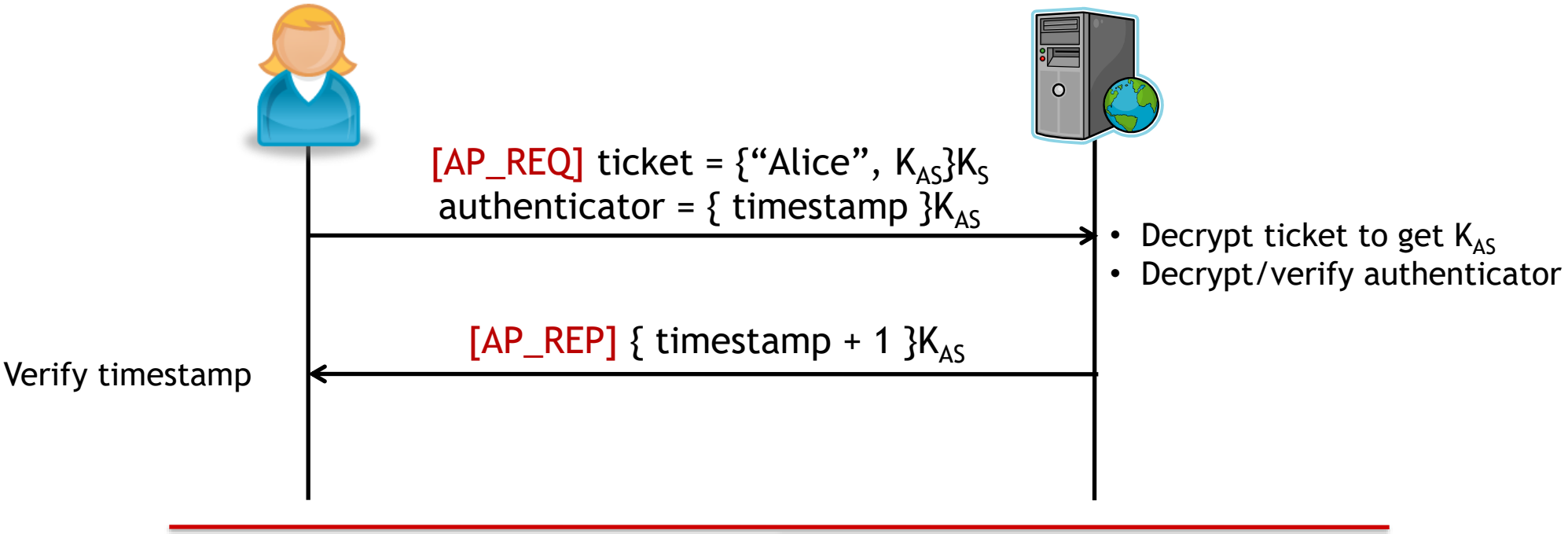
Bytes	*Content*
1	Version of Kerberos (4)
1	Message Type (3)
1	KDC key version number
≤ 40	KDC realm
1	Length of TGT
1	Length of authenticator
var	TGT
var	authenticator
4	Alice's timestamp
1	Desired ticket lifetime
≤ 40	Service name
≤ 40	Service instance

Copied from credentials field of AS_REP

Bytes	*Content*
≤ 40	Alice's name
≤ 40	Alice's instance
≤ 40	Alice's realm
4	checksum
1	5ms timestamp
4	Timestamp
var	999999999999999999999999 999999999999999999999999 999999999999999999999999 999999999999

Note: The TGS_REP message is the same format as AS_REP

Step 3: Using a service ticket



The AP_REQ message authenticates Alice to the server

- Only the KDC knows K_S, so the ticket for Alice is authentic
- If timestamp is recent, then this message is fresh and sent by Alice

The AP_REP message authenticates the server to Alice

- Only Alice, the server, and the KDC know K_{AS}
- If the timestamp is as expected, then this message is fresh

Question: How can we prevent replay attacks?

Using a Service Ticket: Message Detail

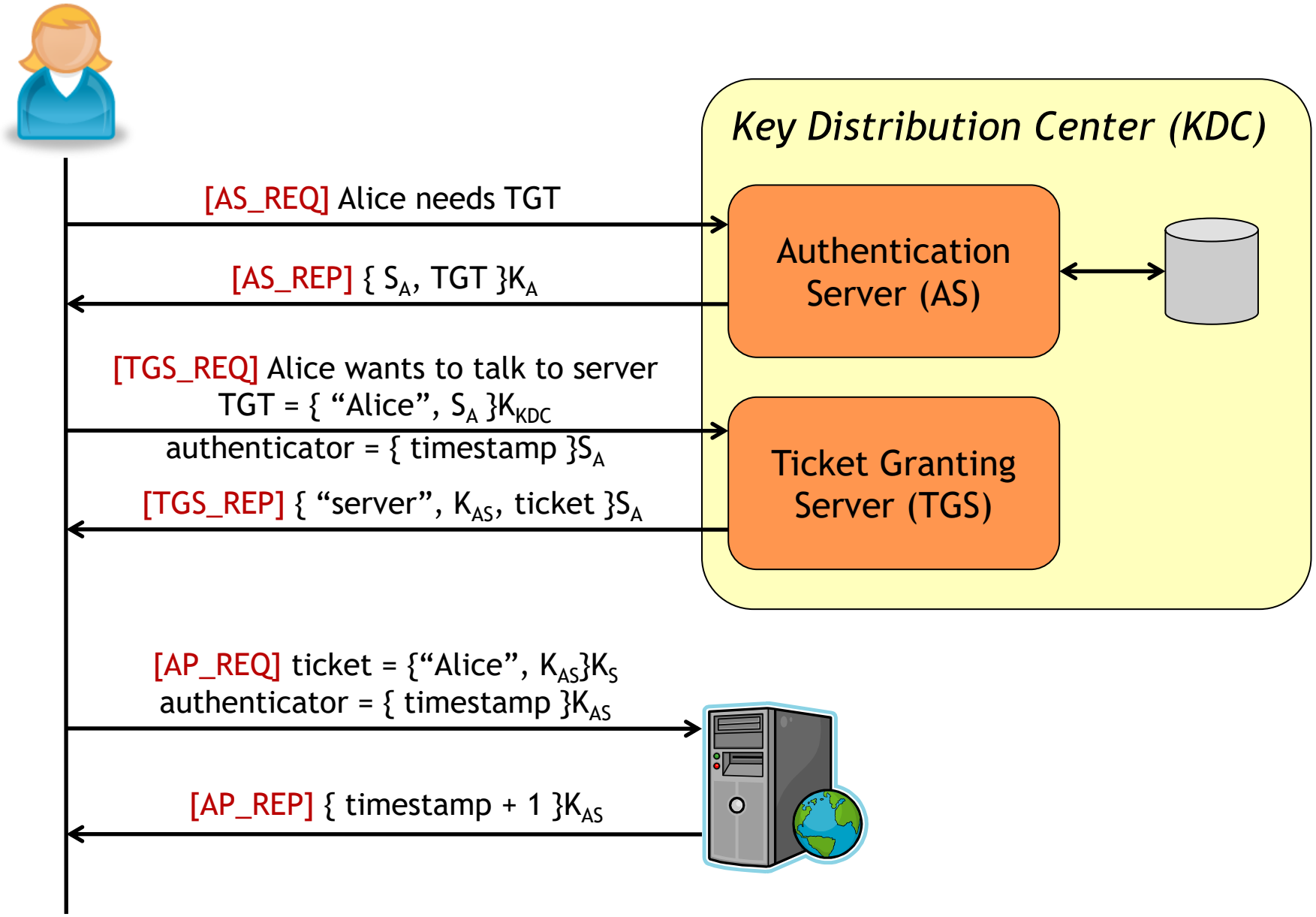
AP_REQ

Bytes	*Content*
1	Version of Kerberos (4)
1	Message Type (8)
1	Server's key version number
≤ 40	Server's realm
1	Length of ticket
1	Length of authenticator
var	ticket
var	authenticator

Copied from credentials field of TGS_REP

The ticket and authenticator follow the same format as in the TGS_REQ messages

Putting it all together…



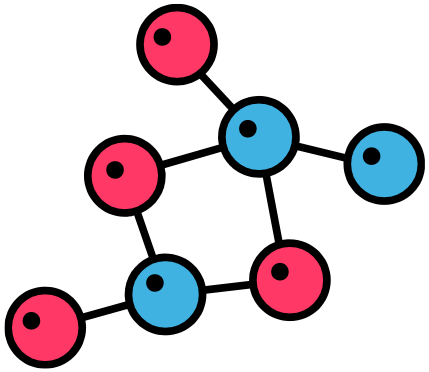
Is the assumption of a global KDC *really* realistic?

Problem: The KDC knows all keys!

- Probably not reasonable in mutually-distrustful organizations
- Scalability as number of users increases is poor
- Very valuable single point of attack

Problem: Reliability and fault tolerance

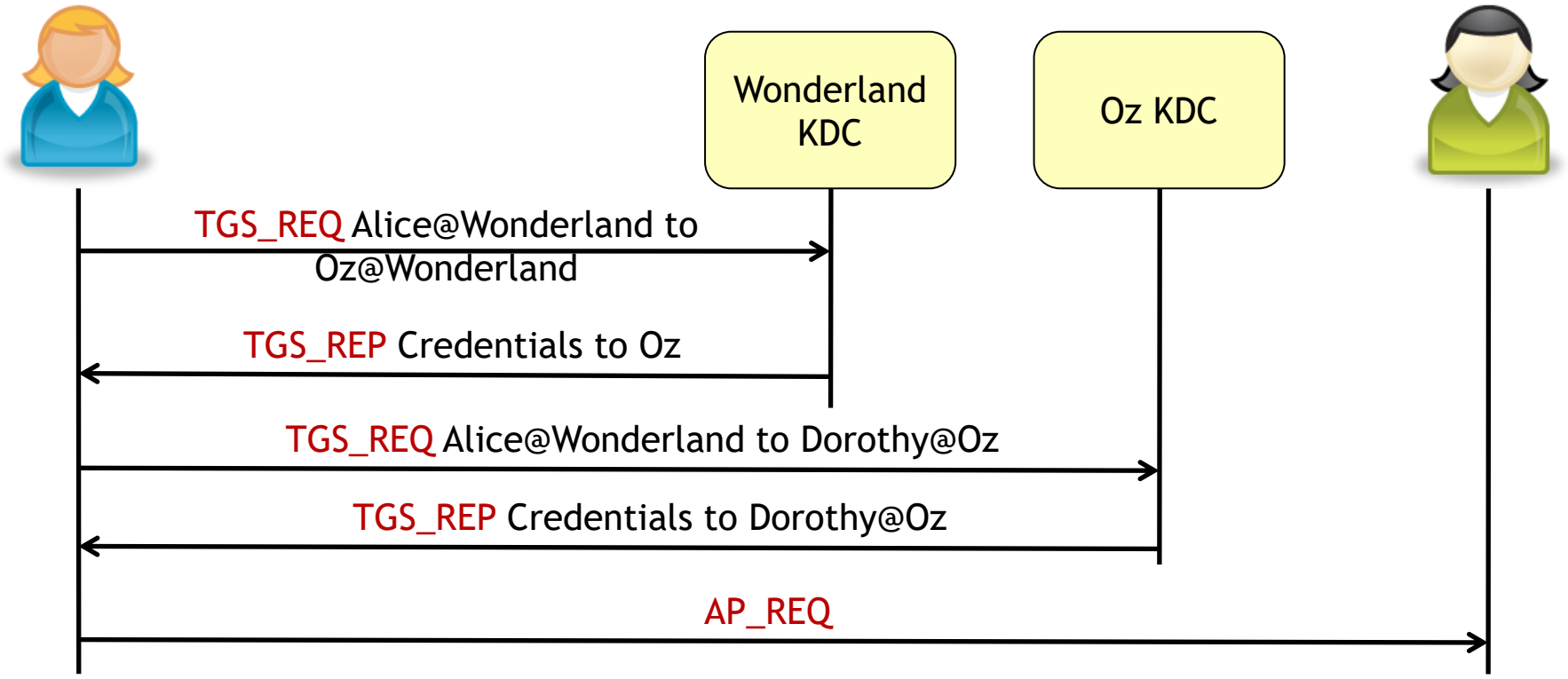
- A single KDC is a single point of failure
- If users cannot authenticate, they cannot work!
- Even if KDC is up all the time and everyone trusts it, it will probably not be able to serve all requests in a timely manner…



Kerberos is widely used, right? How does it address these problems?

Kerberos solves the untrusted KDC problem by allowing inter-realm authentication

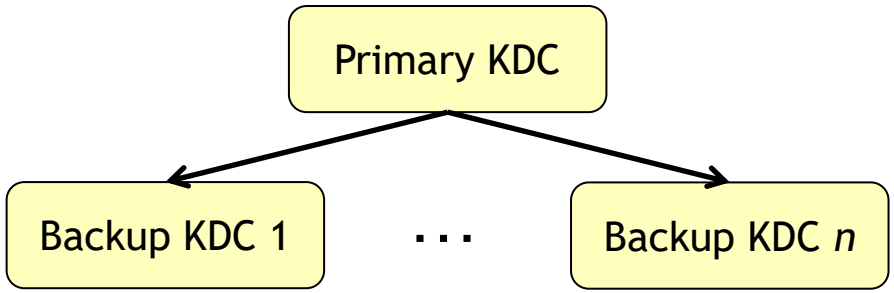
Example: Alice from the realm "Wonderland" wants to talk to Dorothy in the realm "Oz". Clearly, the Wonderland KDC knows nothing about principals managed by the Oz KDC. How do we proceed?



Note: For inter-realm authentication to work, the KDCs for each realm must agree to share keys a priori

Kerberos solves the failure and bottleneck problems by replicating the KDC's database

Goal: Any KDC should be able to service any client request



For this to work:

- The KDCs must all share the same master key
- The databases managed by each replica must be consistent with the primary

How can the replicated databases maintain consistency with the primary?

- Primary DB contents are periodically downloaded by the backups
- Backups are used exclusively for read only operations (Is this a problem?)

How is the DB protected during transmission?

- Confidentiality: Provided "for free" since DB is stored encrypted
- Integrity: Keyed hash using shared master key

Despite being a widely-deployed authentication solution, Kerberos v4 is far from perfect

Security

- Kerberos v4 is based on DES
- Integrity provided using non-standard techniques
- Does not support the use of other algorithms

Clocks and timestamps

- Maximum ticket lifetime is ~21.5 hours
- Cannot renew tickets
- Cannot obtain tickets in advance

Networks and naming

- 4 bytes used for network address
- What about IPv6??
- Names constrained to 40 characters

Kerberos v5 fixes these problems!

Supports extensible security suites. New algorithms can be added to the protocol as they are discovered. Standard techniques used for integrity protection.

Security

- Kerberos v4 is based on DES
- Integrity provided using non-standard techniques
- Does not support the use of other algorithms

ASN.1 used to encode names and addresses. Much more flexible.

Tickets have start and end times, can be renewed, and use a different timestamp format.

Clocks and timestamps

- Maximum ticket lifetime is ~21.5 hours
- Cannot renew tickets
- Cannot obtain tickets in advance

Networks and naming

- 4 bytes used for network address
- What about IPv6??
- Names constrained to 40 characters

Summary of Kerberos

Kerberos is a widely-used authentication paradigm based on the Needham-Schroeder authentication protocol

Authentication via Kerberos is a three-step process

1. Password-based authentication to the AS
2. TGT returned by the AS is used to request a service ticket from the TGS
3. Service ticket is used to mutually authenticate with a service

Inter-realm authentication allows users in different administrative domains to mutually authenticate

Many KDCs are replicated to prevent bottlenecks in the event of failure

Next time: Public key infrastructure (PKI) models