Applied Cryptography and Network Security

William Garrison
bill@cs.pitt.edu
6311 Sennott Square

Handshake Protocols

University of Pittsburgh

Today’s Topics

Today 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

Notation Review

Types of keys:

- K_{AB} : The secret key shared between A and B
- k_A : The public key belonging to A
- k_A^{-1} : The private key belonging to A

Types of cryptographic operations

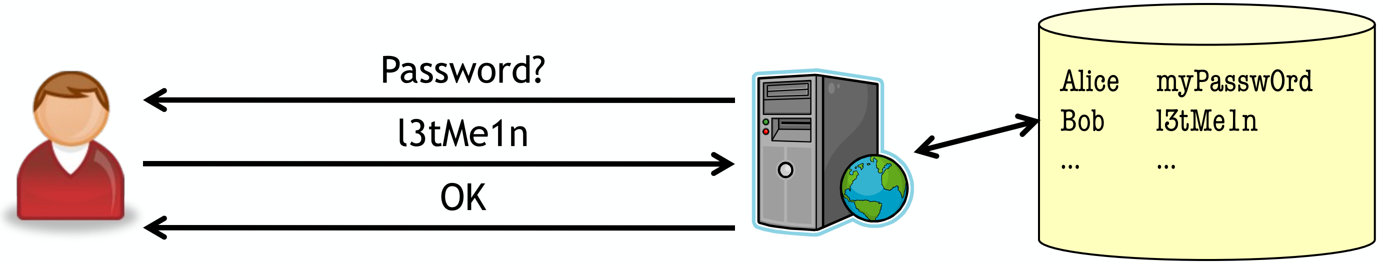
- $\{ M \}K_{AB}$: Message M encrypted with the secret key K_{AB}
- $\{ M \}k_A$: Message M encrypted using A’s public key
- $[M]k_A^{-1}$: Message M signed using A’s private key

Misc:

- R_A : A random number chosen by A
- $\{ M || R_A \}$: The concatenation of M and R_A

Login-only protocols

Login-only protocols are designed to authenticate the user prior to permitting system access



Password?

l3tMe1n

OK

Alice	myPassw0rd
Bob	l3tMe1n
…	…

This protocol operates under the assumptions that:

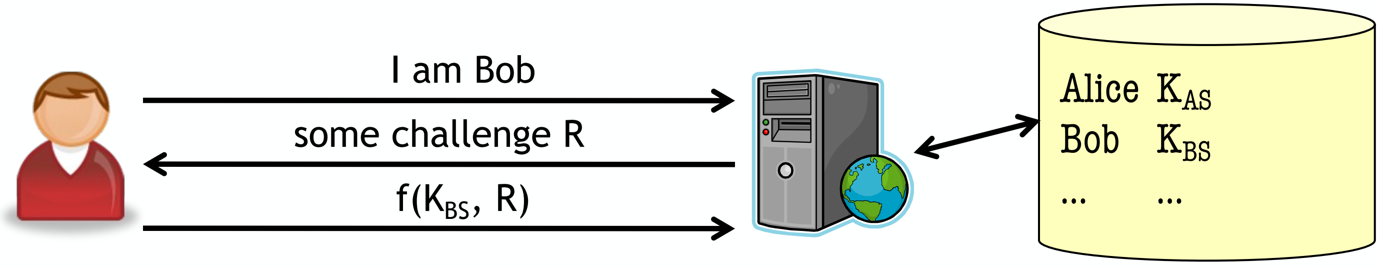
- Only the user knows their password
- The password database is private
- No one is listening on the communication channel

Question: Are these assumptions always valid? Always invalid?

- It depends!

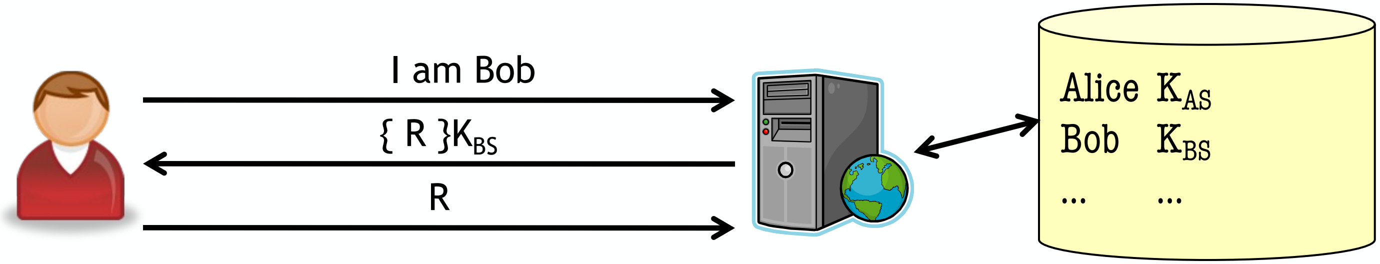
Fortunately, this botched protocol is easy to (partially) fix

It is natural to turn this basic protocol into a cryptographic challenge/response using a shared secret



Interesting notes:

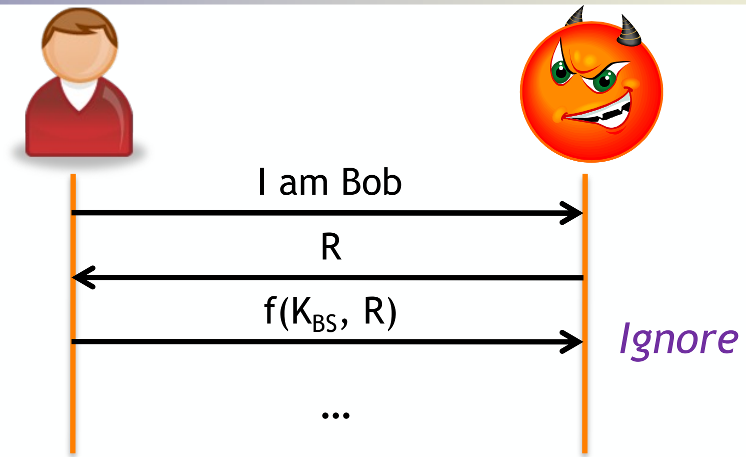
- The use of K_{BS} convinces the server that he is talking to Bob
- f can be either a two-way (encryption) or one-way (hash) function



Note: We must use a two-way (encryption) function (Why?)

Unfortunately, these protocols are subject to a variety of attacks…

Since Bob does not authenticate the server, anyone can pose as the server!



I am Bob

R

f(K_{BS}, R)

Ignore

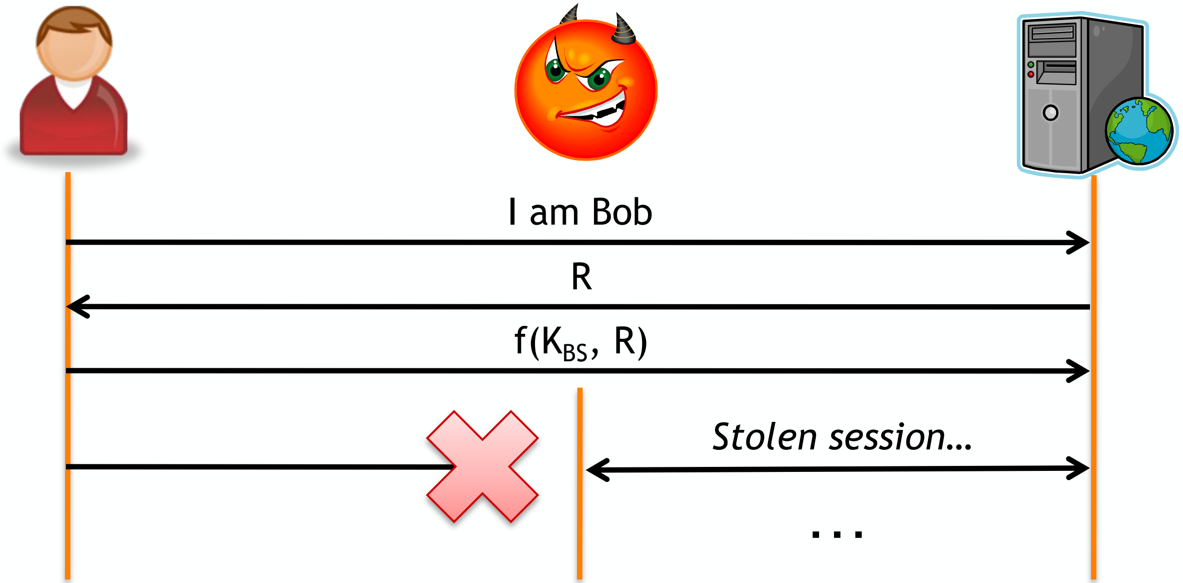
…

Question: Why might this be a problem?

If the attacker has any knowledge of the server, he can:

- Steal Bob's password(s) as he logs in to, e.g., his email
- Use f(K_{BS}, R) to act as an attacker in the middle
- If K_{BS} is derived from a password, f(K_{BS}, R) can be used to launch an offline password guessing attack
  - How?

An attacker capable of blocking traffic from Bob can hijack Bob's session after login



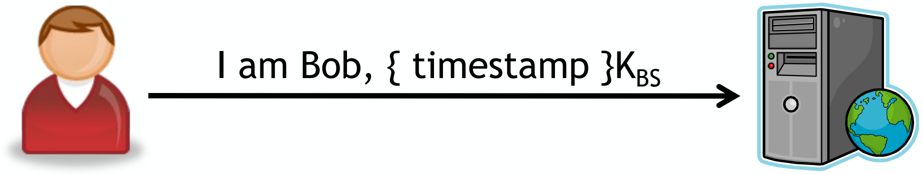
Who could this attacker be?

- Malicious router
- Peer on same network
- …

The protocol ends after the user is authenticated!
Need authentication to be linked to later steps

Question: Why is this attack possible?

What if we only have one message in the protocol to work with?



I am Bob, { timestamp }K_{BS}

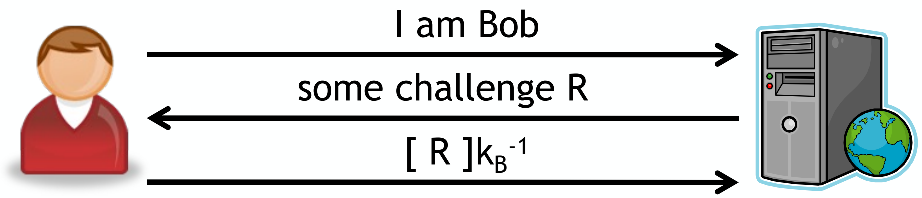
This protocol has several strong points:

- Easily replaces protocols that rely on simply sending a cleartext password
- Only requires one message, not three
- The server does not need to maintain volatile state (e.g., previously used Rs)

Sadly, it also has some weaknesses:

- Bob and the server need synchronized clocks!
- Attackers snooping on the wire can reuse Bob's encrypted timestamp to log into other servers within an acceptable window of time
- If an attacker can convince the server to roll back its clock, old encrypted timestamps can be reused!

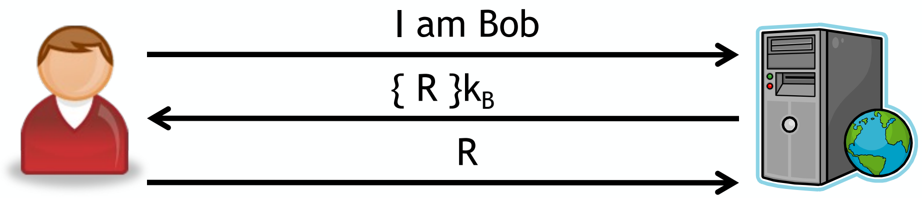
These protocols can also be adapted to use public key cryptography



I am Bob

some challenge R

$[R]k_B^{-1}$



I am Bob

$\{ R \}k_B$

R

Why do these protocols work?

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

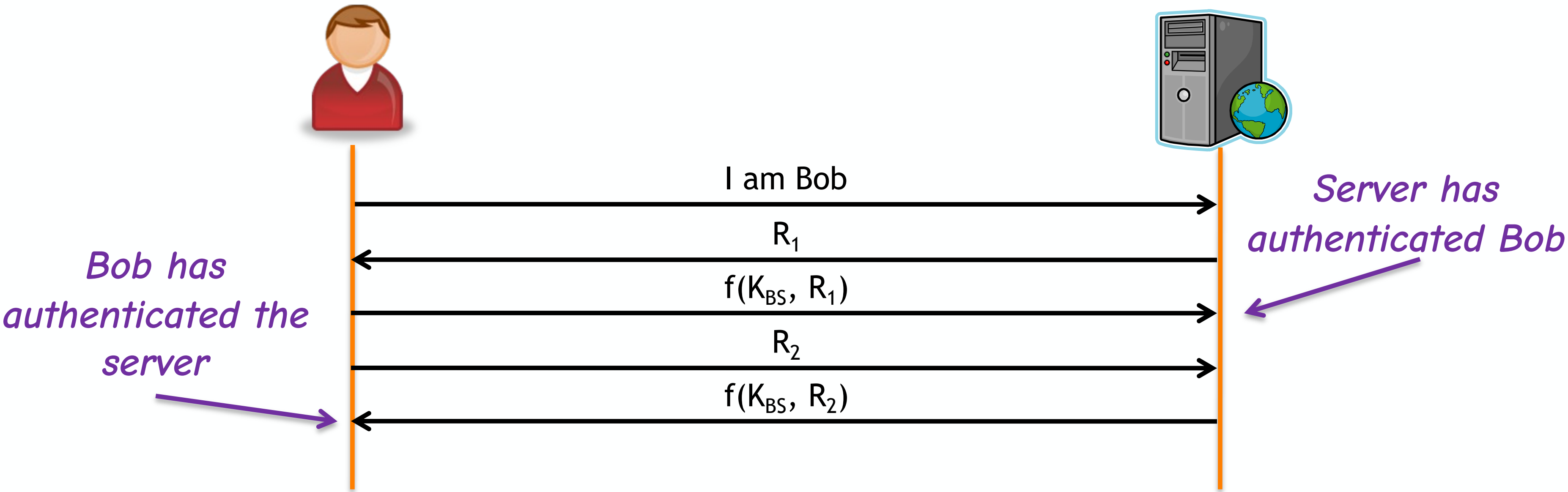
Interesting note: No more sensitive databases of user passwords or shared secrets!

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:



I am Bob

R_1

$f(K_{BS}, R_1)$

R_2

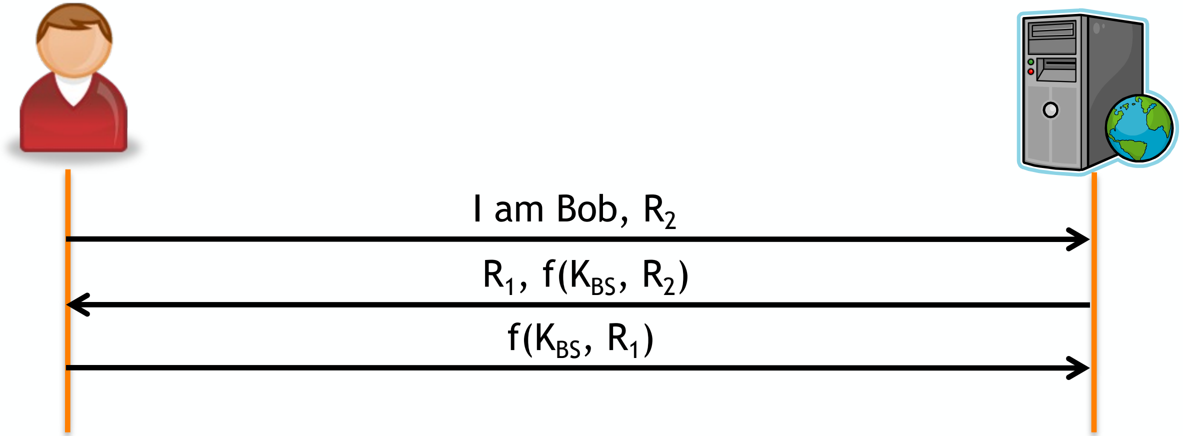
$f(K_{BS}, R_2)$

Server has authenticated Bob

Bob has authenticated the server

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

A stab at optimization…



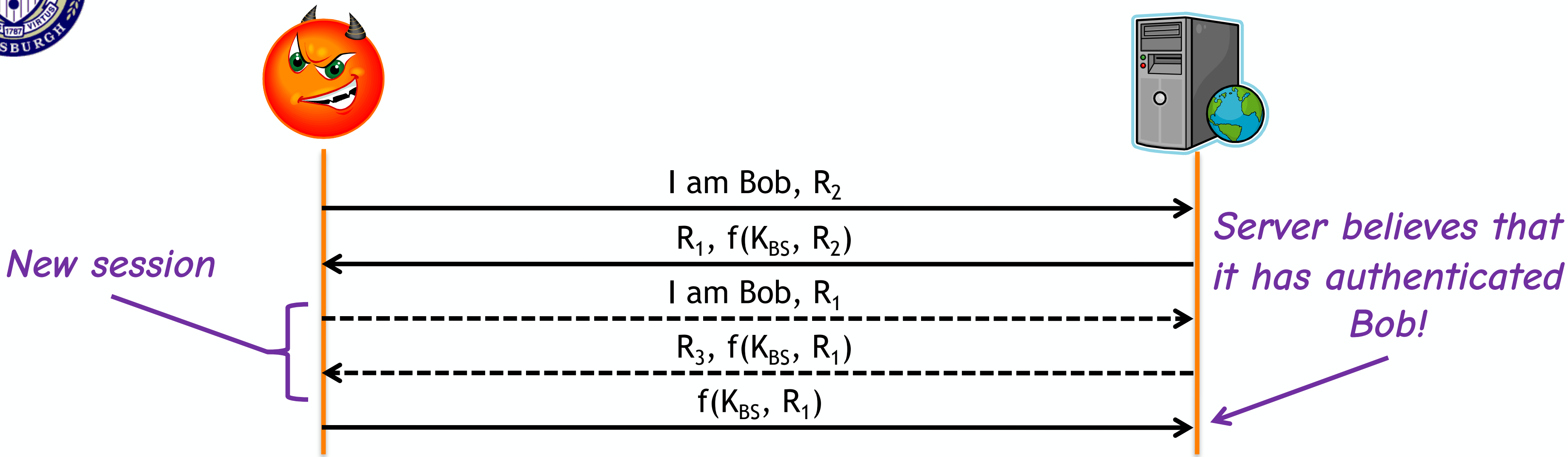
In theory:

- 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


I am Bob, R_2
R_1, f(K_{BS}, R_2)
I am Bob, R_1
R_3, f(K_{BS}, R_1)
f(K_{BS}, R_1)
New session
Server believes that it has authenticated Bob!

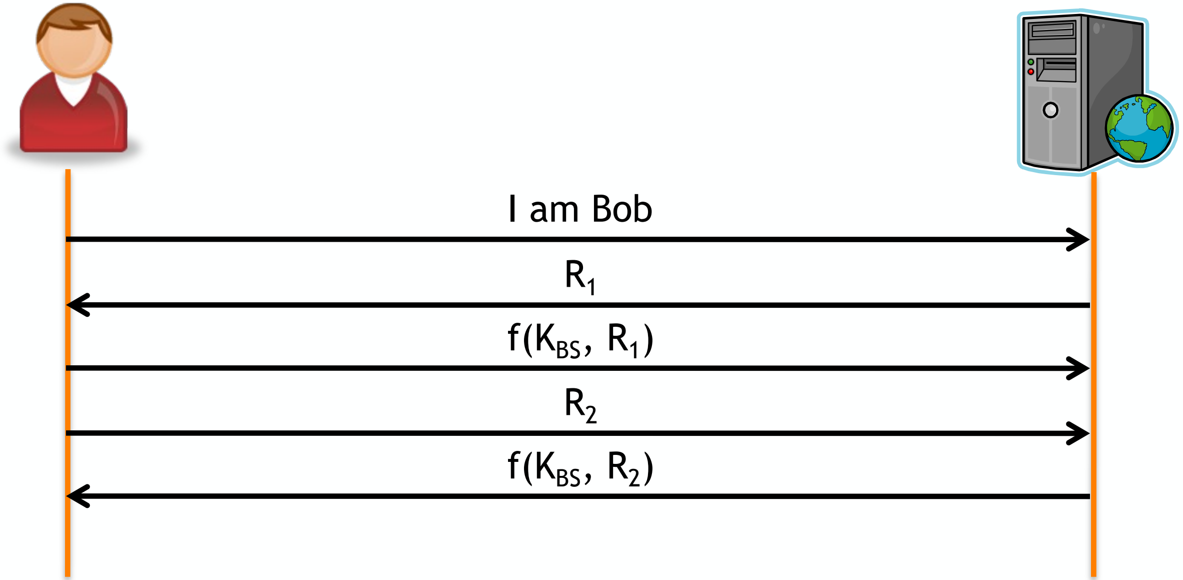
Why is this attack possible?

- 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

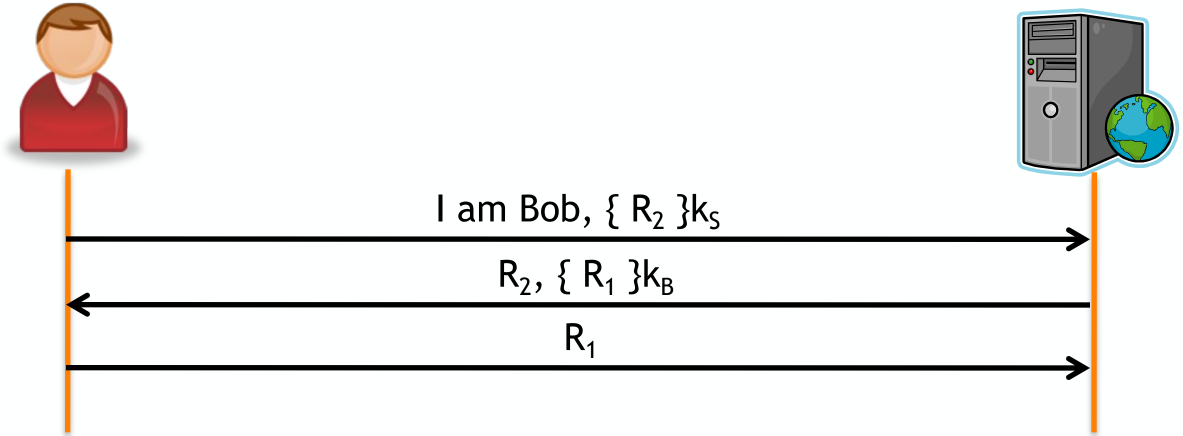
Why isn't the long protocol vulnerable to this attack?



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



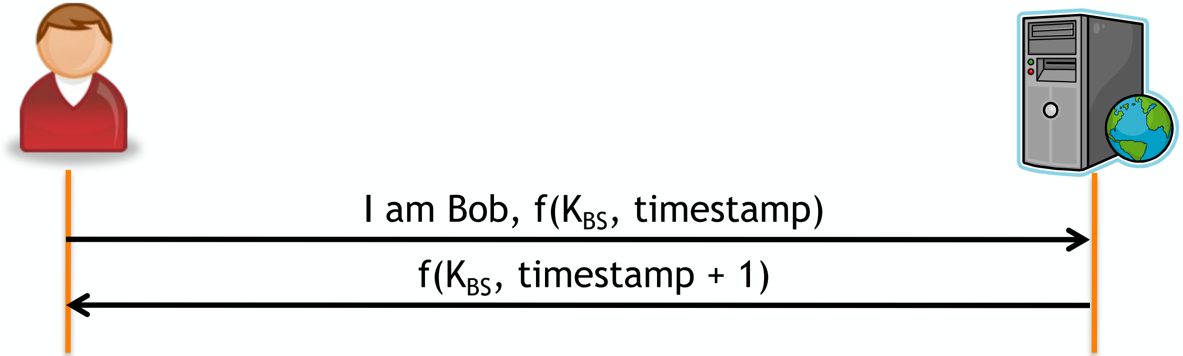
This looks a lot like our "optimized" secret key mutual authentication protocol, doesn't it?

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 / out-of-band configuration (consider 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
- …

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

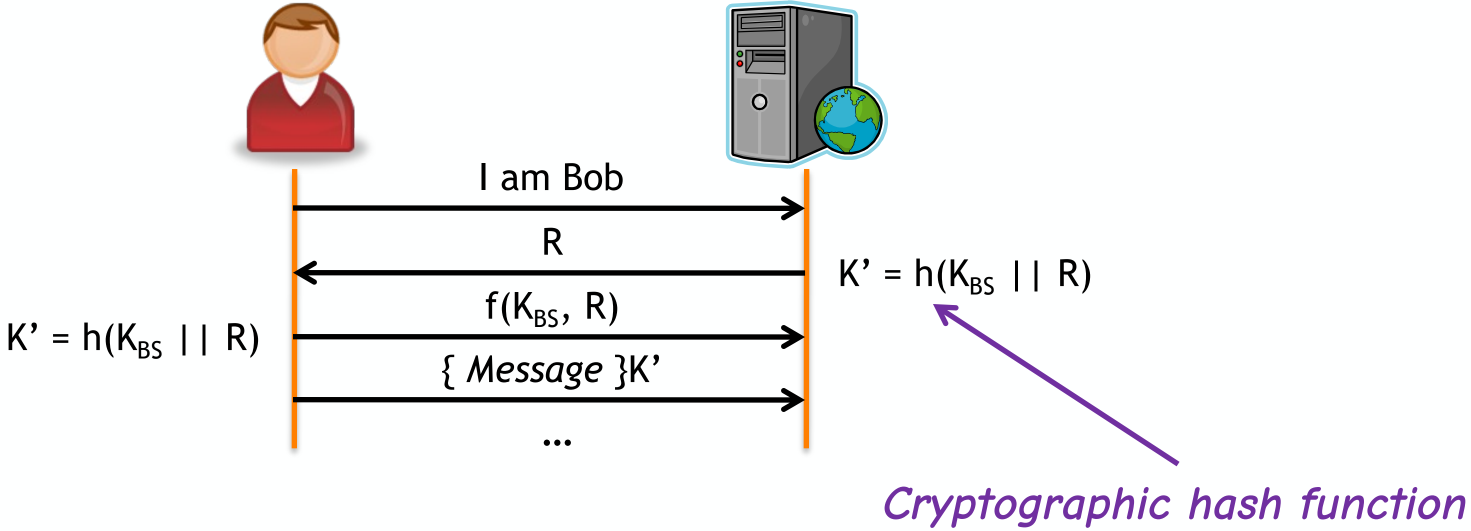
Example: Telnet versus SSH

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

Recall: 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!



I am Bob

R

$K' = h(K_{BS} \mid\mid R)$

$f(K_{BS}, R)$

$K' = h(K_{BS} \mid\mid R)$

{ *Message* }K'

…

Cryptographic hash function

Interesting notes:

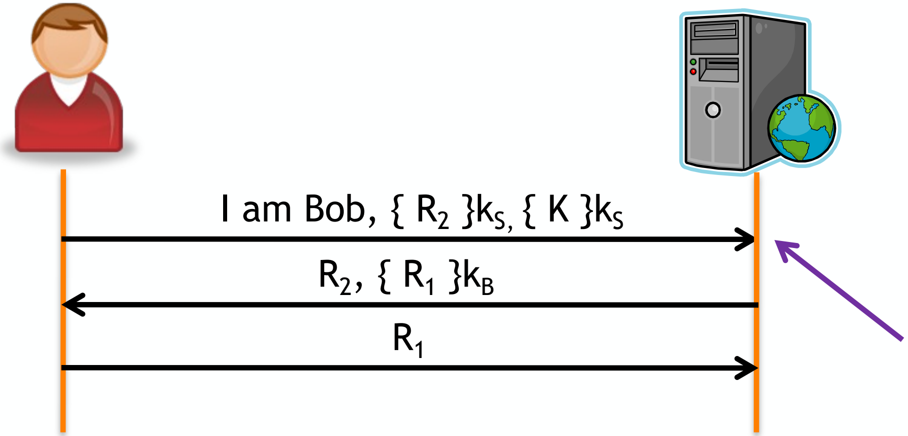
- This protocol does not add any messages to our authentication exchange!
- Even though R is visible to the adversary, K' cannot be guessed (Why?)
- If the session key K' is leaked, the long-term secret K_{BS} is still safe (Why?)

Mutual authentication protocols can be adapted in a similar manner

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

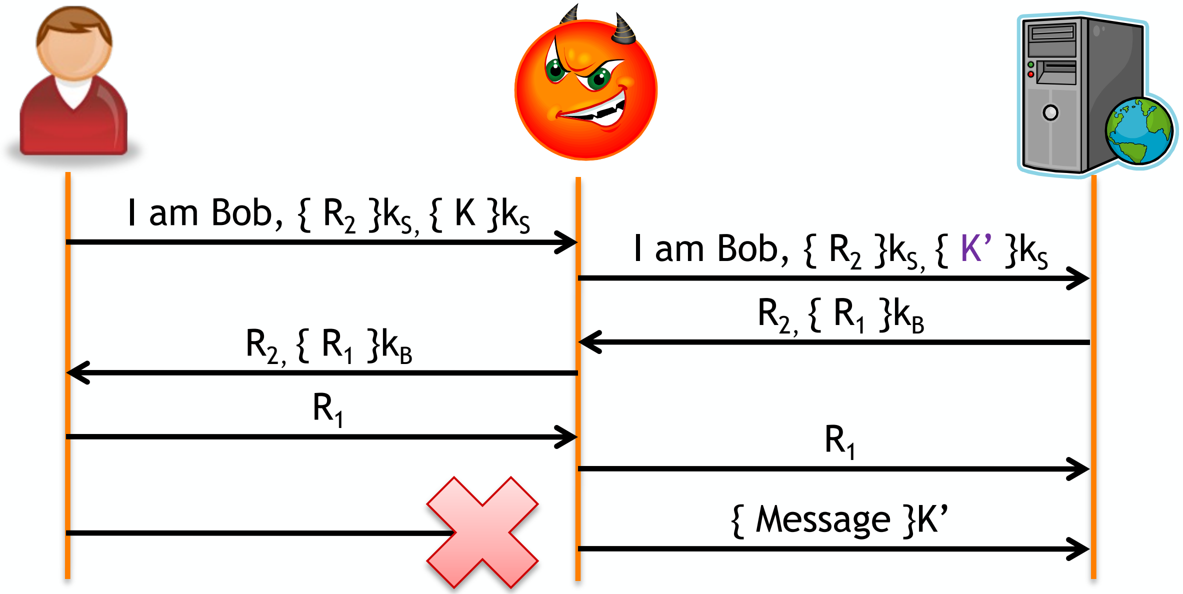
We can also derive session keys using public-key authentication protocols

Initial attempt:

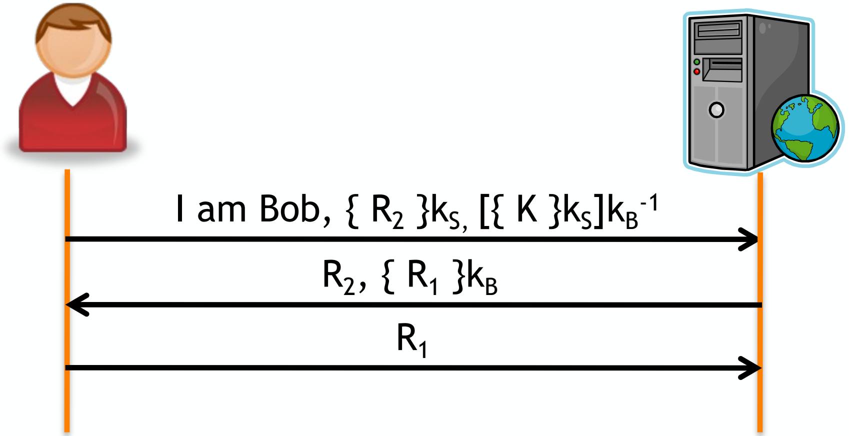


K is the new shared key

Unfortunately, this protocol can be hijacked!


I am Bob, $\{ R_2 \}k_S$, $\{ K \}k_S$
I am Bob, $\{ R_2 \}k_S$, $\{ K' \}k_S$
R_2, $\{ R_1 \}k_B$
R_2, $\{ R_1 \}k_B$
R_1
R_1
{ Message }K'

Digital signatures can help us fix this problem!



I am Bob, $\{ R_2 \}k_{S,}$ $[\{ K \}k_S]k_B^{-1}$

R_2, $\{ R_1 \}k_B$

R_1

Why does this work?

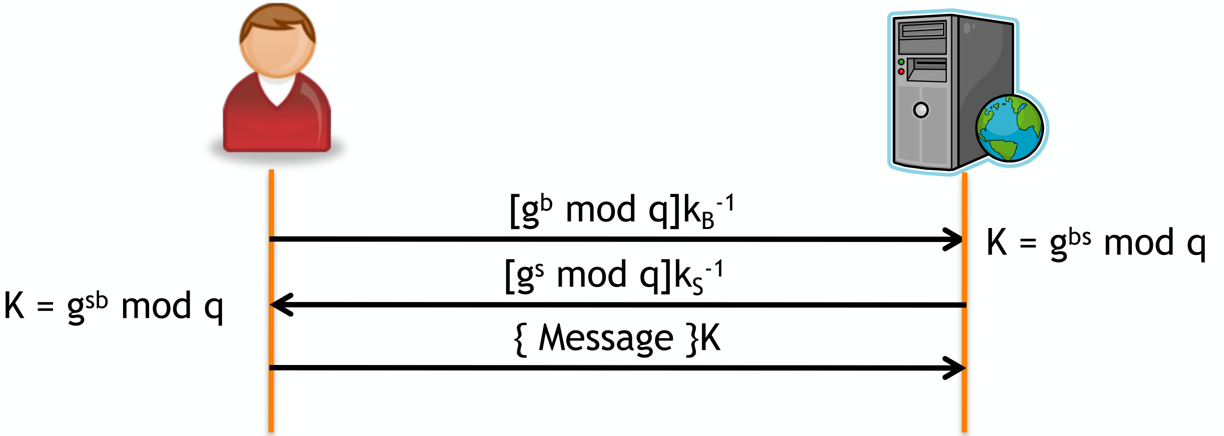
- The signature ensures that the key was actually generated by Bob
- This provides a binding between the authentication protocol and the key exchange protocol

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!
    - And then?

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?

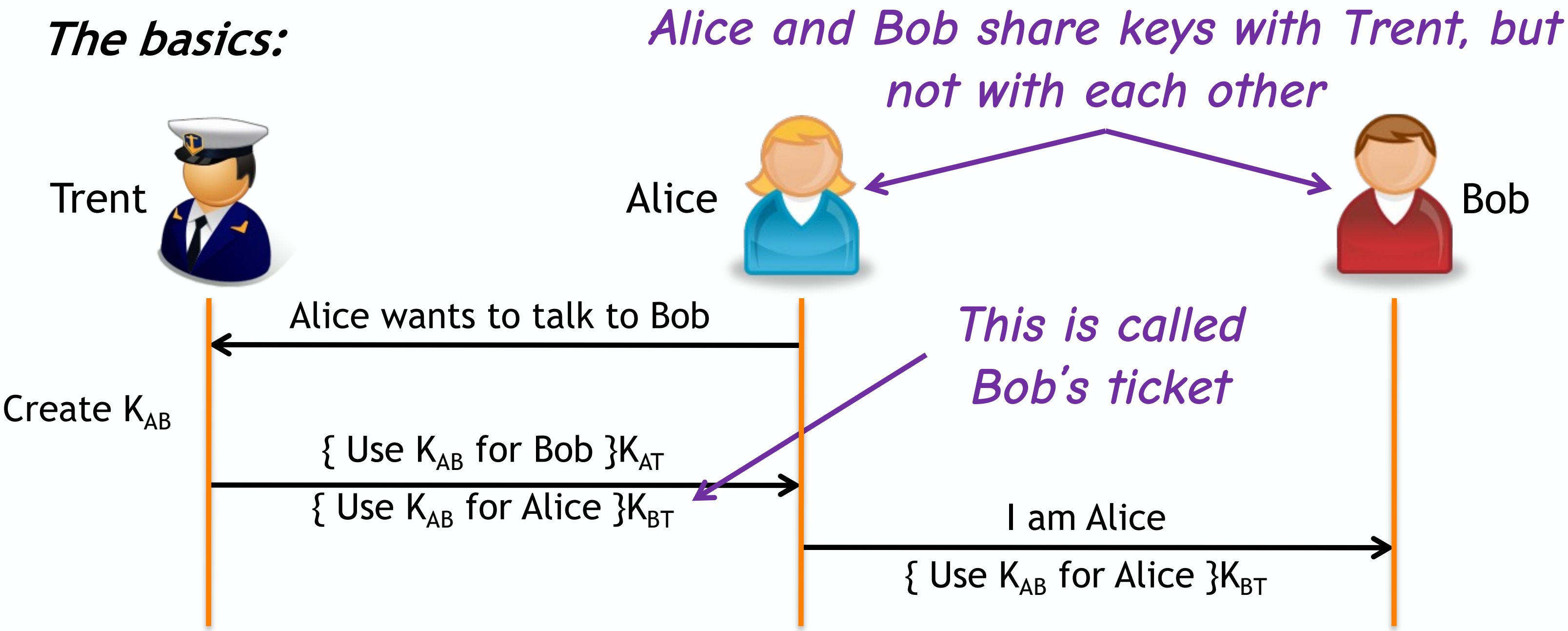
Mediated Authentication Protocols

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:



Alice and Bob share keys with Trent, but not with each other

Trent

Alice

Bob

Alice wants to talk to Bob

Create K_{AB}

{ Use K_{AB} for Bob }K_{AT}

{ Use K_{AB} for Alice }K_{BT}

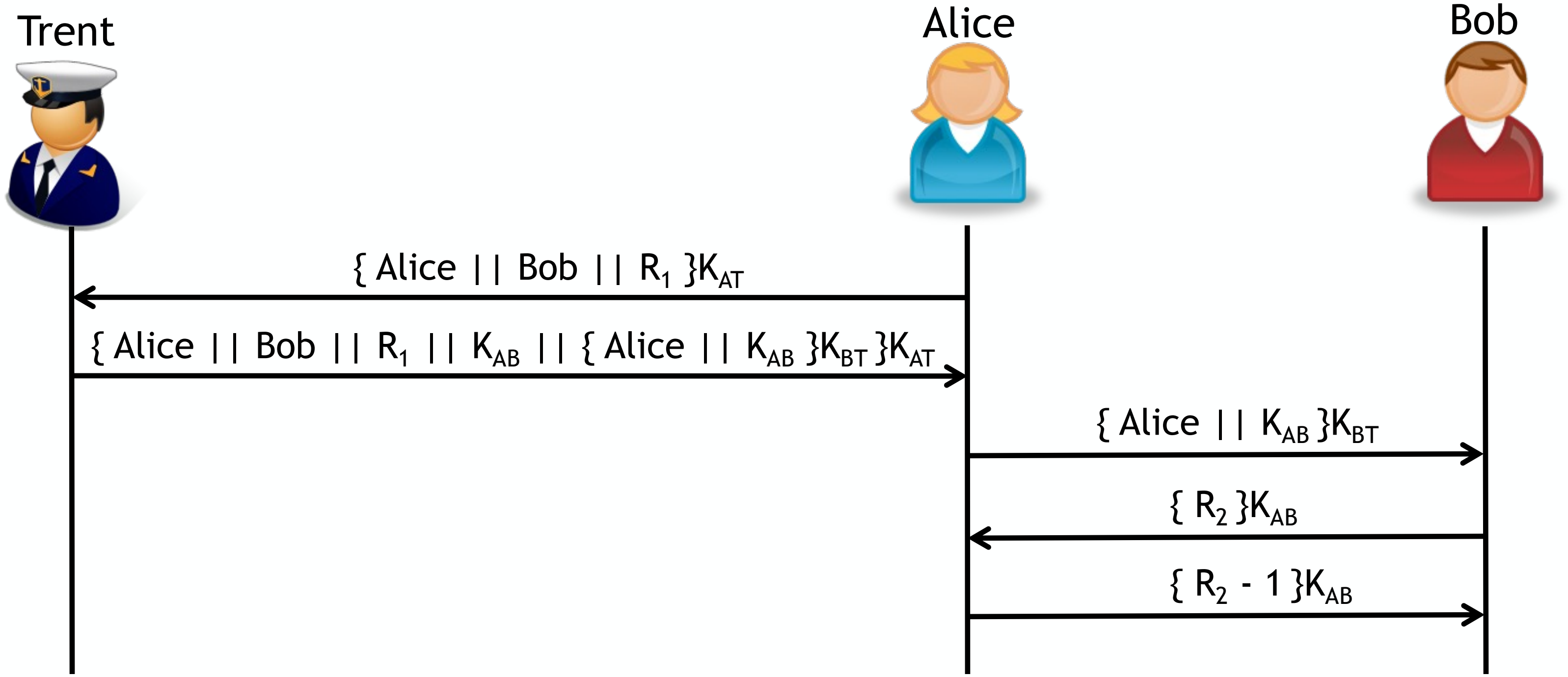
This is called Bob's ticket

I am Alice

{ Use K_{AB} for Alice }K_{BT}

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



Trent

Alice

Bob

{ Alice || Bob || R_1 }K_{AT}

{ Alice || Bob || R_1 || K_{AB} || { Alice || K_{AB} }K_{BT} }K_{AT}

{ Alice || K_{AB} }K_{BT}

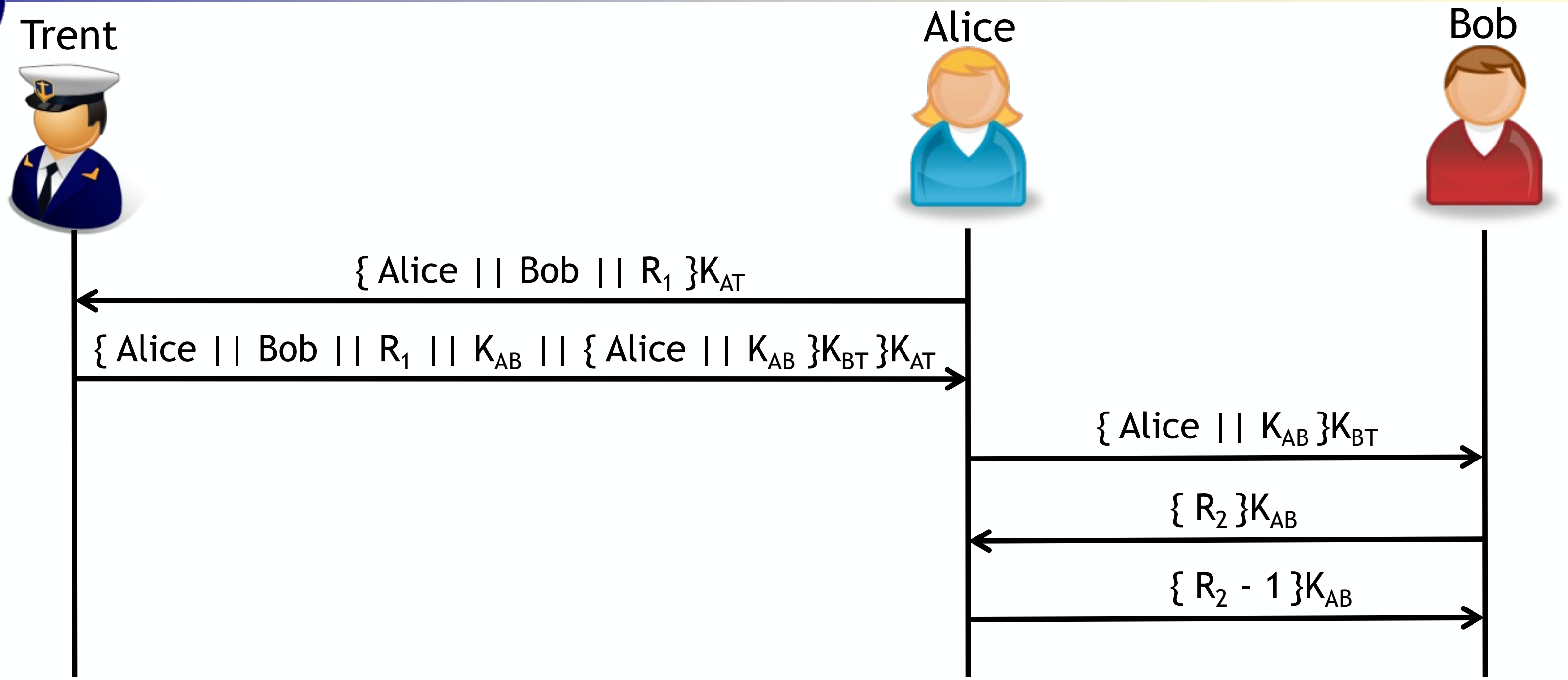
{ R_2 }K_{AB}

{ R_2 - 1 }K_{AB}

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?



Trent

Alice

Bob

{ Alice || Bob || R_1 }K_{AT}

{ Alice || Bob || R_1 || K_{AB} || { Alice || K_{AB} }K_{BT} }K_{AT}

{ Alice || K_{AB} }K_{BT}

{ R_2 }K_{AB}

{ R_2 - 1 }K_{AB}

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}

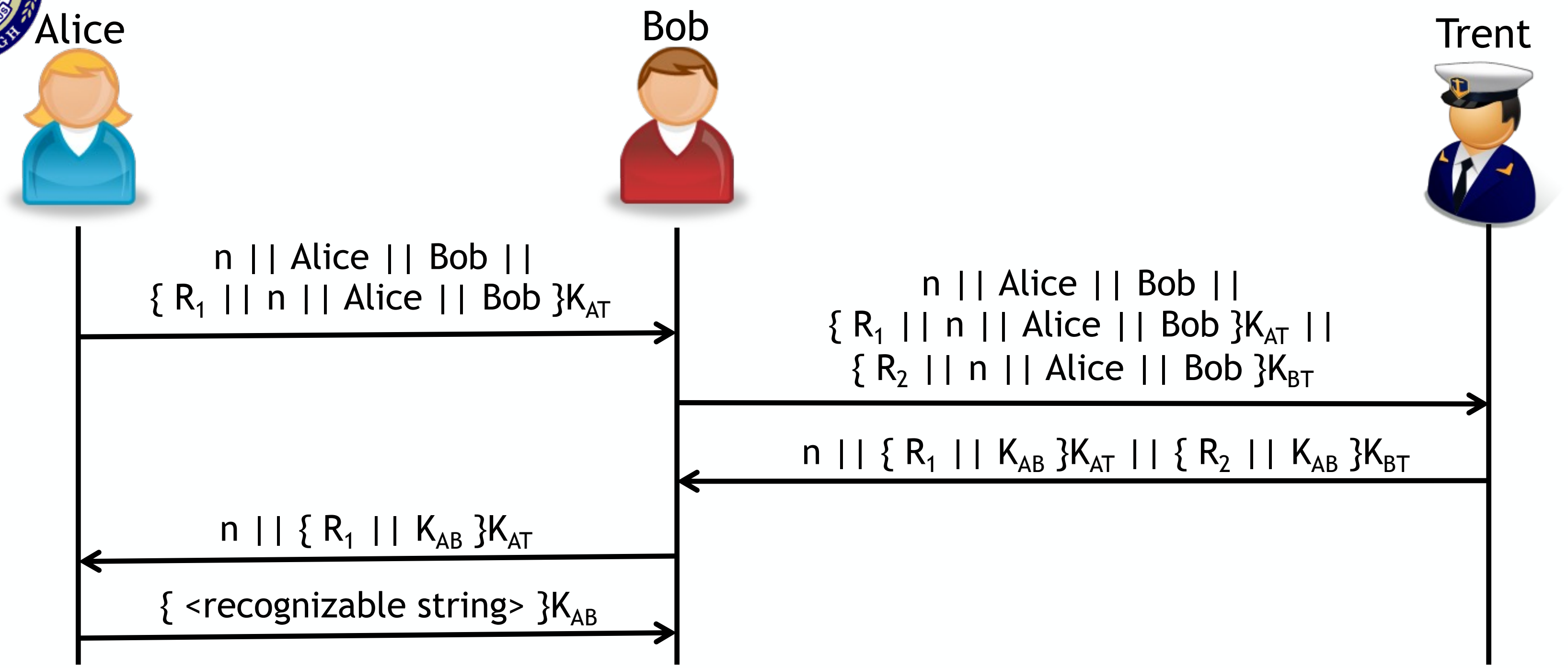
Question: *Why might a session key become compromised?*

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



Alice

Bob

Trent

n || Alice || Bob ||
{ R_1 || n || Alice || Bob }K_{AT}

n || Alice || Bob ||
{ R_1 || n || Alice || Bob }K_{AT} ||
{ R_2 || n || Alice || Bob }K_{BT}

n || { R_1 || K_{AB} }K_{AT} || { R_2 || K_{AB} }K_{BT}

n || { R_1 || K_{AB} }K_{AT}

{ <recognizable string> }K_{AB}

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

Conclusions

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