UNIVERSITY OF AMSTERDAM

Jaap van Ginkel

Security of Systems and Networks

October 14, 2024 Kerberos SSH TLS

Kerberos

- In Greek mythology, Kerberos is 3-headed dog that guards entrance to Hades
  - "Wouldn't it make more sense to guard the exit?"
- In security, Kerberos is an authentication system based on symmetric key crypto
  - Originated at MIT
  - Based on work by Needham and Schroeder
  - Relies on a **trusted third party (TTP)**

Motivation for Kerberos

- Authentication using public keys
  - N users $\Rightarrow$ N key pairs
- Authentication using symmetric keys
  - N users requires about N^2 keys
- Symmetric key case **does not scale!**
- Kerberos based on symmetric keys but only requires N keys for N users
  - But must rely on TTP
  - Advantage is that no PKI is required

Kerberos KDC

- Kerberos **Key Distribution Center** or **KDC**
  - Acts as a TTP
  - TTP must not be compromised!
  - KDC shares symmetric key K_A with Alice, key K_B with Bob, key K_C with Carol, etc.
  - Master key K_{KDC} known only to KDC
  - KDC enables authentication and session keys
  - Keys for confidentiality and integrity
  - In practice, the crypto algorithm used is DES

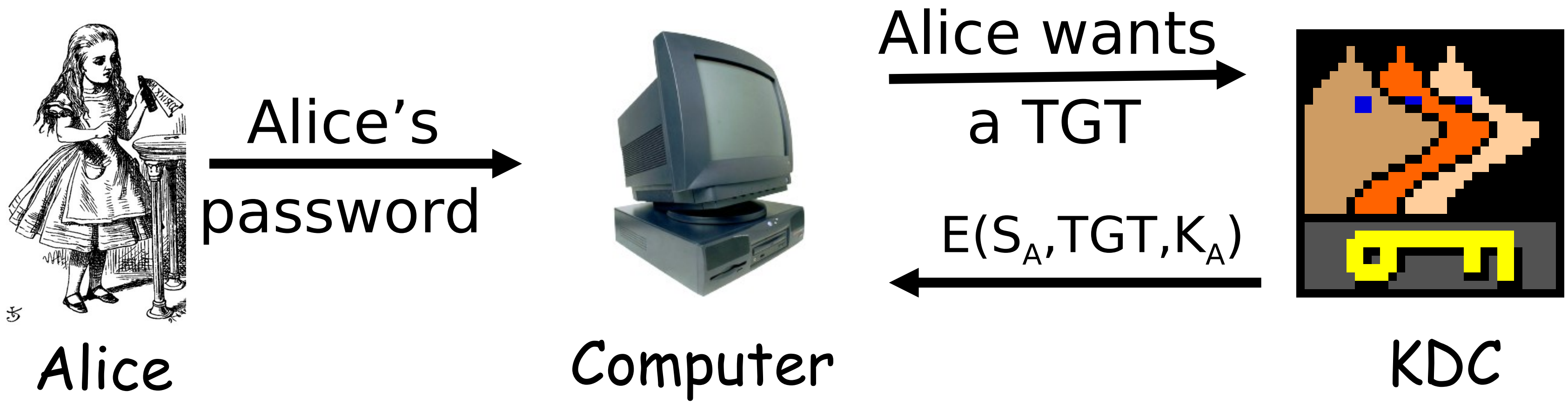
Kerberos Tickets

- KDC issues a **ticket** containing info needed to access a network resource
- KDC also issues **ticket-granting tickets** or **TGTs** that are used to obtain tickets
- Each TGT contains
  - Session key
  - User's ID
  - Expiration time
- Every TGT is encrypted with K_{KDC}
  - TGT can only be read by the KDC

Kerberized Login

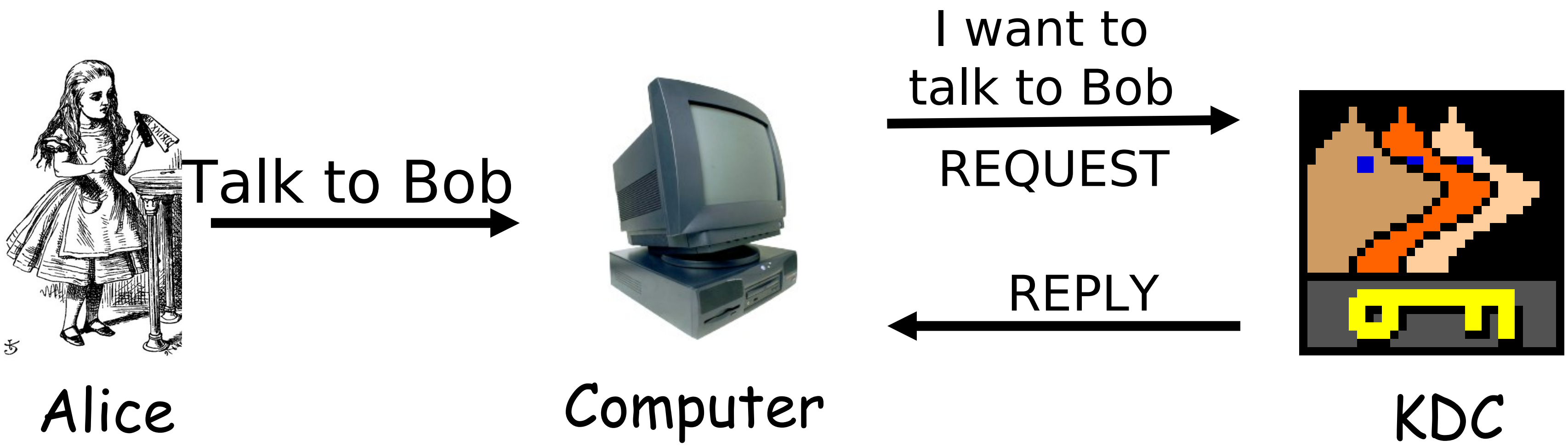
- Alice enters her password
- Alice's workstation
  - Derives K_A from Alice's password
  - Uses K_A to get TGT for Alice from the KDC
- Alice can then use her TGT (credentials) to securely access network resources
- **Plus:** Security is transparent to Alice
- **Minus:** KDC must be secure — it's trusted!

Kerberized Login



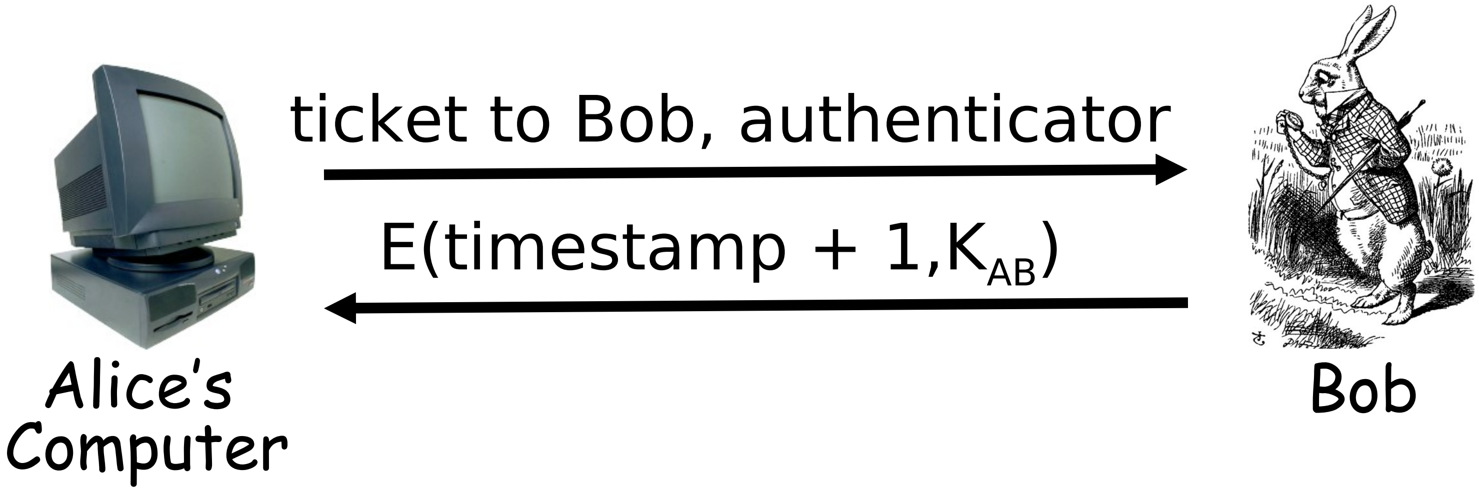
- Key K_A derived from Alice's password
- KDC creates session key S_A
- Workstation decrypts S_A, TGT, forgets K_A
- TGT = E("Alice",S_A, K_{KDC})

Alice Requests Ticket to Bob



- REQUEST = (TGT, authenticator) *where* authenticator = E(timestamp,S_A)
- REPLY = E("Bob",K_{AB},ticket to Bob, S_A)
- ticket to Bob = E("Alice",K_{AB},K_B)
- KDC gets S_A from TGT to verify timestamp

Alice Uses Ticket to Bob



- ticket to Bob = E("Alice",K_{AB}, K_B)
- authenticator = E(timestamp, K_{AB})
- Bob decrypts "ticket to Bob" to get K_{AB} which he then uses to verify timestamp

Kerberos

- Session key S_A used for authentication
- Can also be used for confidentiality/integrity
- Timestamps used for mutual authentication
- Recall that timestamps reduce number of messages
  - Acts like a nonce that is known to both sides
  - Note: **time** is a security-critical parameter!

Kerberos Questions

- When Alice logs in, KDC sends $E(S_A,TGT,K_A)$ where $TGT = E(\text{"Alice"},S_A,K_{KDC})$

  Q: Why is TGT encrypted with K_A?

  A: Extra work and no added security!
- In Alice's Kerberized login to Bob, why can Alice remain anonymous?
- Why is "ticket to Bob" sent to Alice?
- Where is replay prevention in Kerberos?

Kerberos Alternatives

- Could have Alice's workstation remember password and use that for authentication
  - Then no KDC required
  - But hard to protect password on workstation
  - Scaling problem
- Could have KDC remember session key instead of putting it in a TGT
  - Then no need for TGTs
  - But **stateless** KDC is big feature of Kerberos

Kerberos Keys

- In Kerberos, K_A = h(Alice's password)
- Could instead generate random K_A and
  - Compute K_h = h(Alice's password)
  - And workstation stores $E(K_A, K_h)$
- Then K_A need not change (on workstation or KDC) when Alice changes her password
- But $E(K_A, K_h)$ subject to password guessing
- This alternative approach is often used in applications (but not in Kerberos)

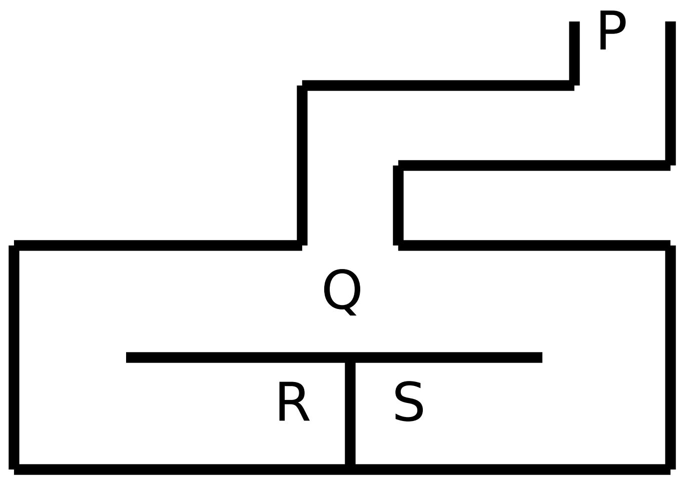
Zero Knowledge Proofs

Zero Knowledge Proof (ZKP)

- Alice wants to prove that she knows a secret without revealing **any** info about it
- Bob must verify that Alice knows secret
  - Even though he gains no info about the secret
- Process is probabilistic
  - Bob can verify that Alice knows the secret to an arbitrarily high probability
- An "interactive proof system"

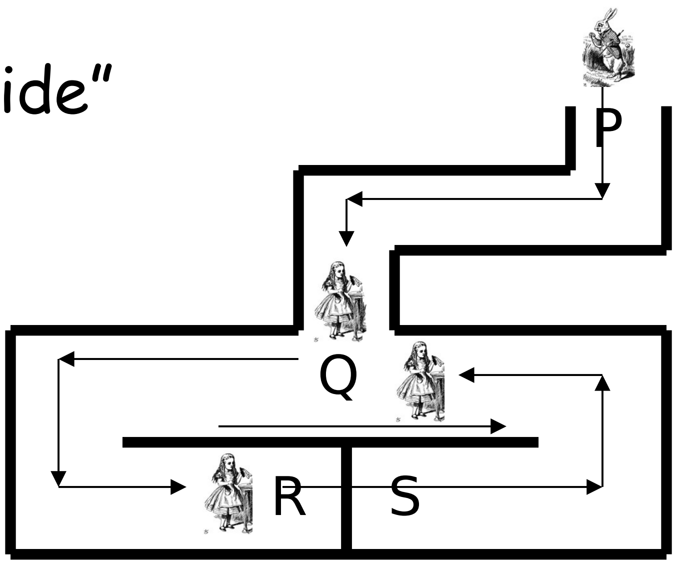
Bob's Cave

- Alice claims to know secret phrase to open path between R and S ("open sarsparilla")
- Can she convince Bob that she knows the secret without revealing phrase?



Bob's Cave

- Bob: "Alice come out on S side"
- Alice (quietly): "Open sarsparilla"
- If Alice does not know secret…
- …then Alice could come out from the correct side with probability 1/2
- If Bob repeats this n times, then Alice (who does not know secret) can only fool Bob with probability $1/2^n$



SSH

- Creates a "secure tunnel"
- Insecure command sent thru SSH tunnel are then secure
- SSH used with things like rlogin
  - Why is rlogin insecure without SSH?
  - Why is rlogin secure with SSH?
- SSH is a relatively simple protocol

SSH

- SSH authentication can be based on:
  - Public keys, or
  - Digital certificates, or
  - Passwords
- Here, we consider ***certificate*** mode
  - Other modes, see homework problems
- We consider slightly simplified SSH…

Simplified SSH

Alice

Alice, CP, R_A →

← CS, R_B

g^a mod p →

← g^b mod p, certificate_B, S_B

$E(\text{Alice}, \text{certificate}_A, S_A, K)$ →

Bob

- CP = "crypto proposed", and CS = "crypto selected"
- $H = h(\text{Alice},\text{Bob},\text{CP},\text{CS},R_A,R_B,g^a \bmod p,g^b \bmod p,g^{ab} \bmod p)$
- $S_B = [H]_{\text{Bob}}$
- $S_A = [H, \text{Alice}, \text{certificate}_A]_{\text{Alice}}$
- $K = g^{ab} \bmod p$

MiM Attack on SSH?

Alice

Alice, R_A →

← R_B

g^a mod p →

← g^t mod p, $cert_B$, S_B

E(Alice,$cert_A$,S_A,K) →

Trudy

Alice, R_A →

← R_B

g^t mod p →

← g^b mod p, $cert_B$, S_B

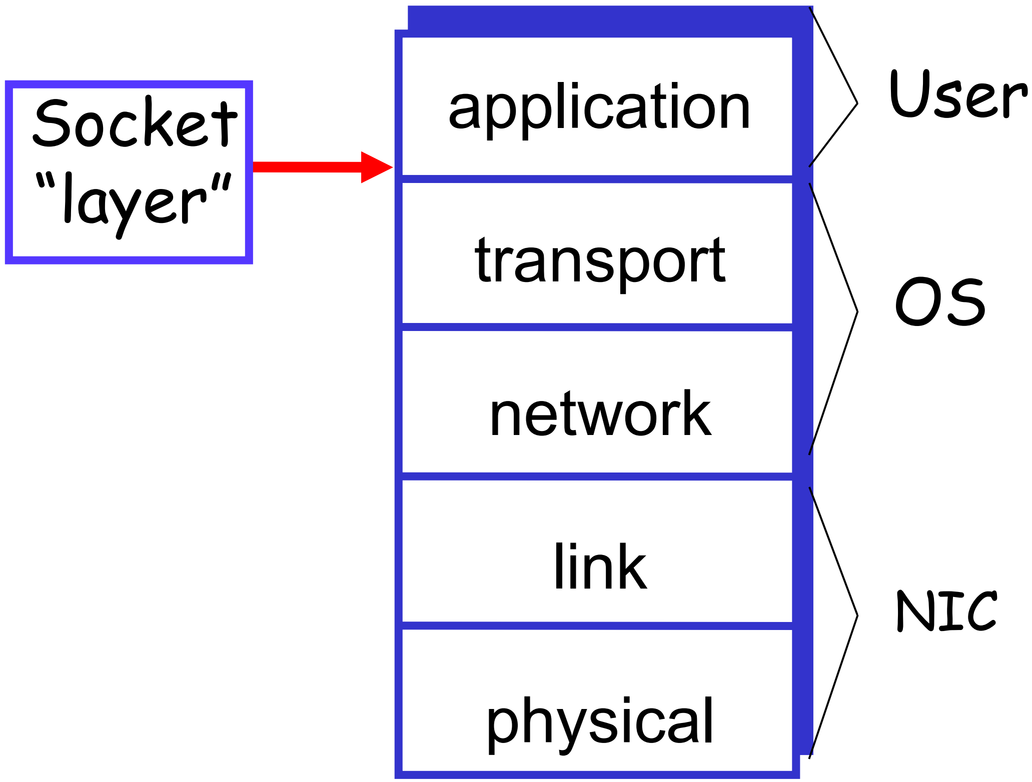
E(Alice,$cert_A$,S_A,K) →

Bob

- Where does this attack fail?
- Alice computes:
  - H_a = h(Alice,Bob,CP,CS,R_A,R_B,g^a mod p,g^t mod p,g^{at} mod p)
- But Bob signs:
  - H_b = h(Alice,Bob,CP,CS,R_A,R_B,g^t mod p,g^b mod p,g^{bt} mod p)

Socket layer

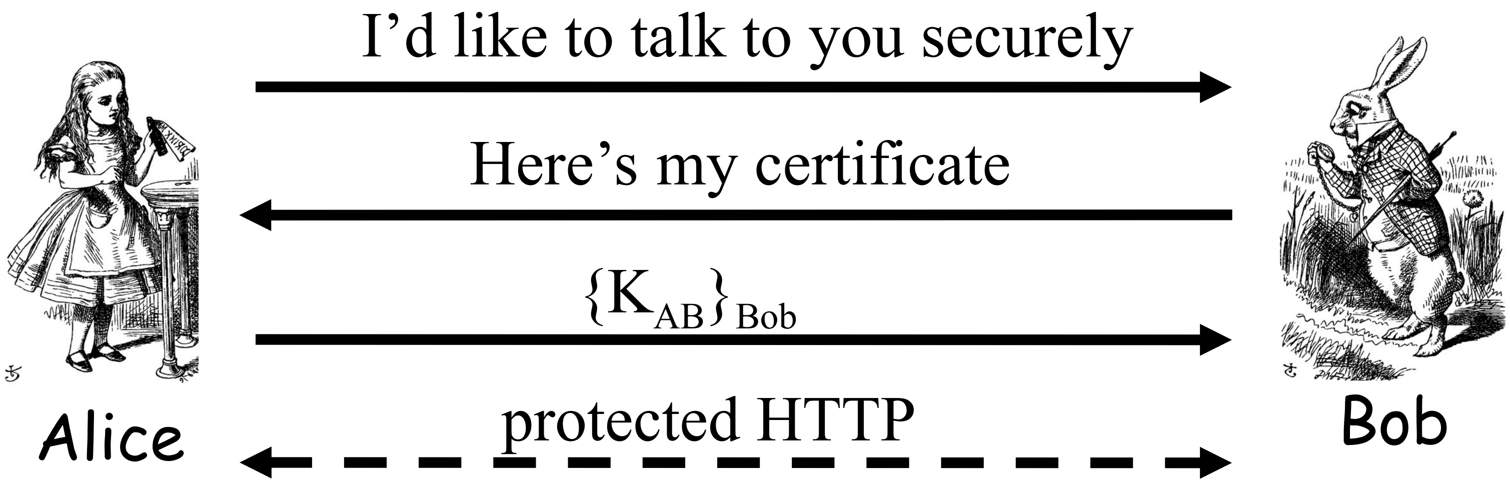
- "Socket layer" lives between application and transport layers
- SSL usually lies between HTTP and TCP



What is SSL?

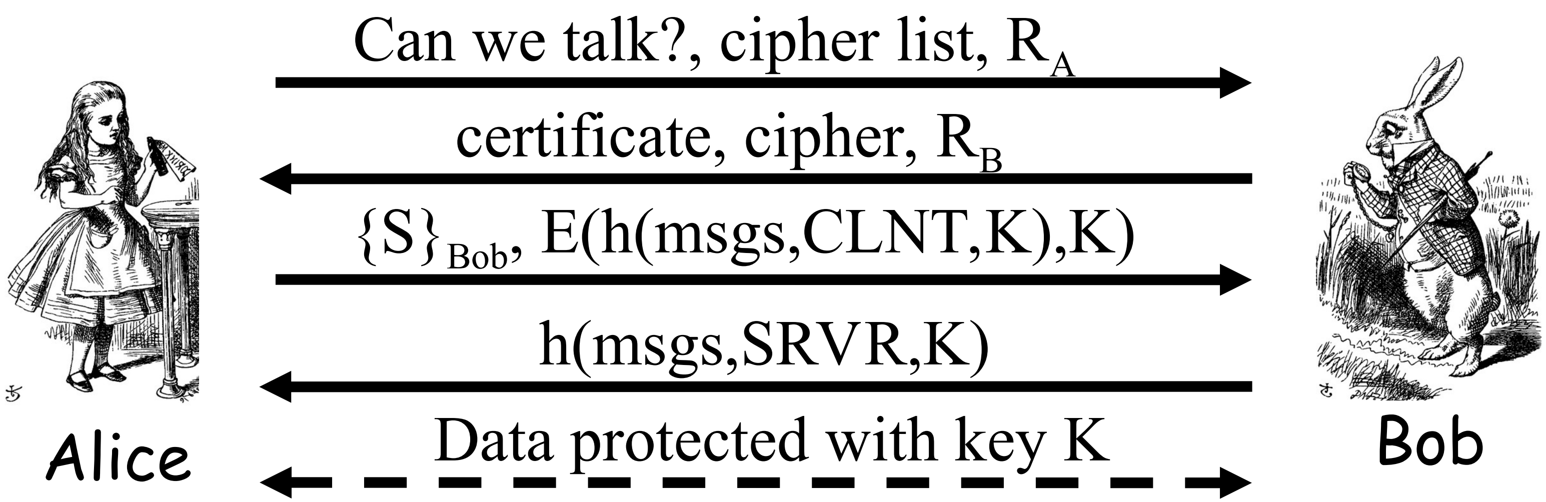
- SSL is the protocol used for most secure transactions over the Internet
- For example, if you want to buy a book at amazon.com…
  - You want to be sure you are dealing with Amazon (**authentication**)
  - Your credit card information must be protected in transit (**confidentiality** and/or **integrity**)
  - As long as you have money, Amazon doesn't care who you are (authentication need not be mutual)

Simple SSL-like Protocol



- Is Alice sure she's talking to Bob?
- Is Bob sure he's talking to Alice?

Simplified SSL Protocol



- S is **pre-master secret**
- $K = h(S,R_A,R_B)$
- msgs = all previous messages
- CLNT and SRVR are constants

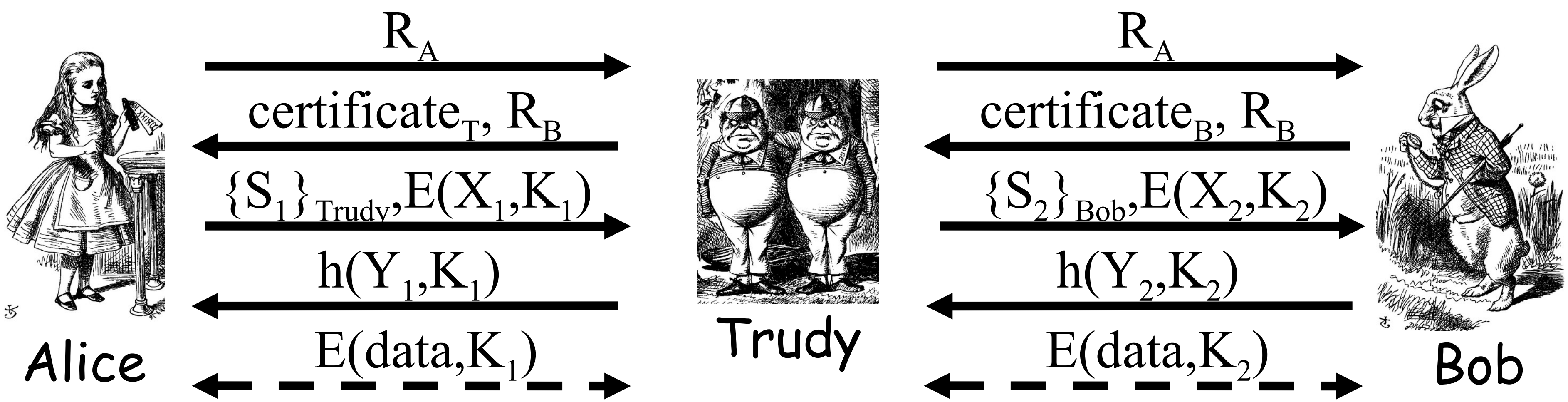
SSL Keys

- 6 "keys" derived from $K = \text{hash}(S,R_A,R_B)$
  - 2 encryption keys: send and receive
  - 2 integrity keys: send and receive
  - 2 IVs: send and receive
  - Why different keys in each direction?
- **Q:** Why is $h(\text{msgs},\text{CLNT},K)$ encrypted (and integrity protected)?
- **A:** It adds no security…

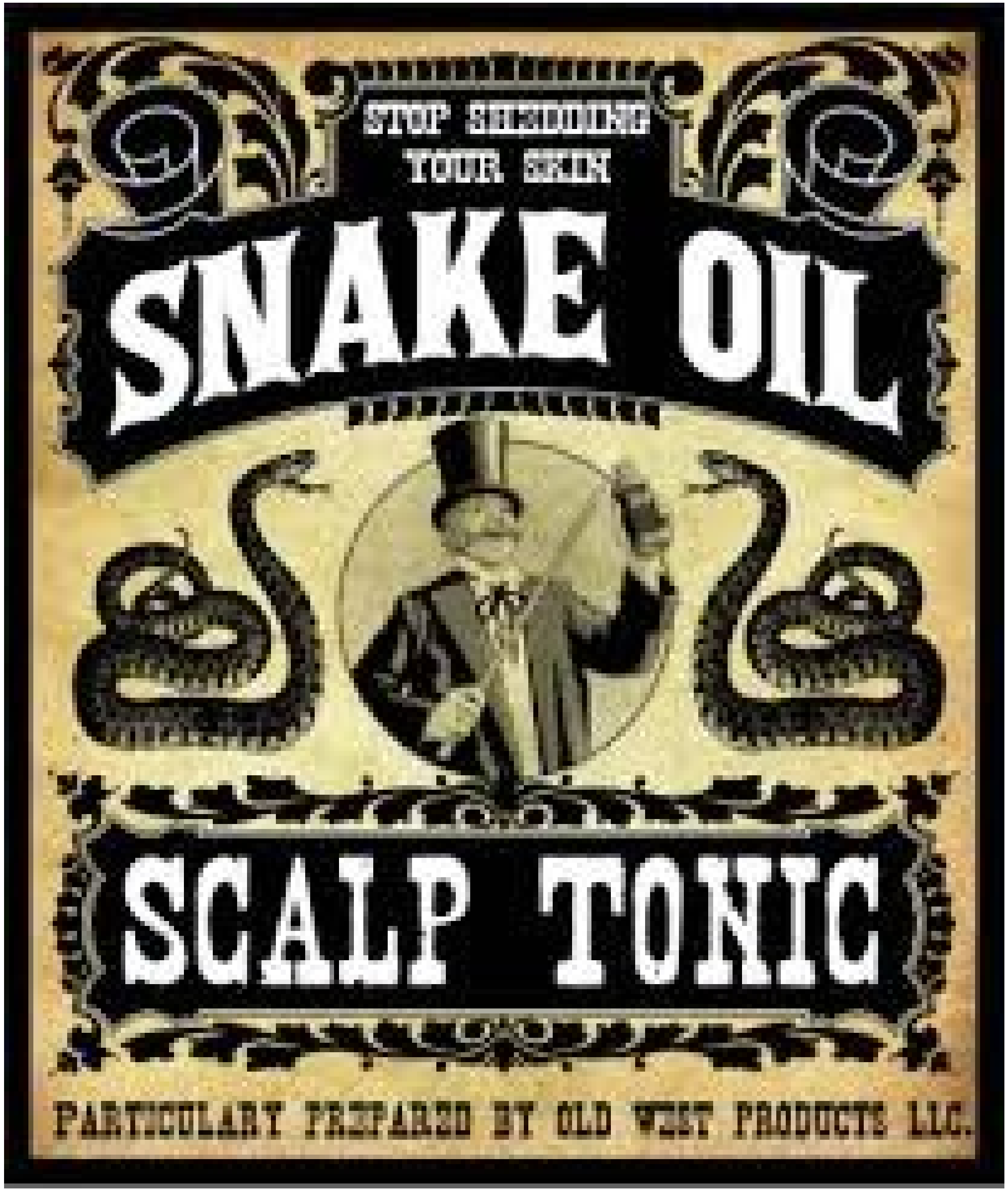
SSL Authentication

- Alice authenticates Bob, not vice-versa
  - How does client authenticate server?
  - Why does server not authenticate client?
- Mutual authentication is possible: Bob sends **certificate request** in message 2
  - This requires client to have certificate
  - If server wants to authenticate client, server could instead require (encrypted) password

SSL MiM Attack



- **Q:** What prevents this MiM attack?
- **A:** Bob's certificate must be signed by a certificate authority (such as Verisign)
- What does Web browser do if sig. not valid?
- What does user do if signature is not valid?


STOP SHEDDING
YOUR SKIN
SNAKE OIL
SCALP TONIC
PARTICULARY PREPARED BY OLD WEST PRODUCTS LLC.

SSL Sessions vs Connections

- SSL **session** is established as shown on previous slides
- SSL designed for use with HTTP 1.0
- HTTP 1.0 usually opens multiple simultaneous (parallel) **connections**
- SSL session establishment is costly
  - Due to public key operations
- SSL has an efficient protocol for opening new connections given an existing session

SSL Connection

Alice

session-ID, cipher list, R_A →

← session-ID, cipher, $R_{B,}$ h(msgs,SRVR,K)

h(msgs,CLNT,K) →

← Protected data →

Bob

- Assuming SSL **session** exists
- So S is already known to Alice and Bob
- Both sides must remember session-ID
- Again, $K = h(S,R_A,R_B)$
- **No public key operations!** (relies on known S)

SSL vs IPSec

- IPSec — discussed in next section
  - Lives at the network layer (part of the OS)
  - Has encryption, integrity, authentication, etc.
  - Is overly complex (including serious flaws)
- SSL (and IEEE standard known as TLS)
  - Lives at socket layer (part of user space)
  - Has encryption, integrity, authentication, etc.
  - Has a simpler specification

SSL vs IPSec

- IPSec implementation
  - Requires changes to OS, but no changes to applications
- SSL implementation
  - Requires changes to applications, but no changes to OS
- SSL built into Web application early on (Netscape)
- IPSec used in VPN applications (secure tunnel)
- Reluctance to retrofit applications for SSL
- Reluctance to use IPSec due to complexity and interoperability issues
- Result? **Internet less secure than it should be!**

Secure Network Programming API

- Early research efforts toward transport layer security included the Secure Network Programming (SNP) application programming interface (API), which in 1993 explored the approach of having a secure transport layer API closely resembling Berkeley sockets, to facilitate retrofitting preexisting network applications with security measures.[4]

SSL 1.0, 2.0 and 3.0

- Developed by Netscape engineers
- Phil Karlton, Alan Freier

- Version 1.0 never released

- Version 2.0 February 1995
- some (serious) security flaws

- Version 3.0 1996
- Complete redesign
- Basis for current TLS versions

-

TLS 1.0 (SSL 3.1)

- TLS 1.0 January 1999
- RFC 2246
- Based on SSL Version 3.0
- "the differences between this protocol and SSL 3.0 are not dramatic, but they are significant enough that TLS 1.0 and SSL 3.0 do not interoperate."
-

TLS 1.1 (SSL 3.2)

-
- RFC 4346 April 2006
- CBC attack protection
- Better IV
- Better padding
-
-

TLS 1.2 (SSL 3.3)

- RFC 5246 August 2008
- RFC 6176 March 2011

- MD5-SHA-1 replaced with SHA-256
- Downgrade protections
- Galois/Counter Mode (GCM)

TLS 1.3

- RFC 8446 in August 2018
- Breaks some TLS interception :-)
- Added
  - ChaCha20 stream cipher with the Poly1305 message authentication code
  - Ed25519 and Ed448 digital signature algorithms
  - x25519 and x448 key exchange protocols
- Removed
  - RSA key transport — Doesn't provide forward secrecy
  - CBC mode ciphers — Responsible for BEAST, and Lucky
  - RC4 stream cipher — Not secure for use in HTTPS
  - SHA-1 hash function — Deprecated in favor of SHA-2
  - Arbitrary Diffie-Hellman groups — CVE-2016-0701
  - Export ciphers — Responsible for FREAK and LogJam

Key exchange/agreement and authentication

Algorithm	SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3	Status
RSA	Yes	Yes	Yes	Yes	Yes	No	Defined for TLS 1.2 in RFCs
DH-RSA	No	Yes	Yes	Yes	Yes	No	
DHE-RSA (forward secrecy)	No	Yes	Yes	Yes	Yes	Yes	
ECDH-RSA	No	No	Yes	Yes	Yes	No	
ECDHE-RSA (forward secrecy)	No	No	Yes	Yes	Yes	Yes	
DH-DSS	No	Yes	Yes	Yes	Yes	No	
DHE-DSS (forward secrecy)	No	Yes	Yes	Yes	Yes	No[45]	
ECDH-ECDSA	No	No	Yes	Yes	Yes	No	
ECDHE-ECDSA (forward secrecy)	No	No	Yes	Yes	Yes	Yes	
PSK	No	No	Yes	Yes	Yes		
PSK-RSA	No	No	Yes	Yes	Yes		
DHE-PSK (forward secrecy)	No	No	Yes	Yes	Yes		
ECDHE-PSK (forward secrecy)	No	No	Yes	Yes	Yes		
SRP	No	No	Yes	Yes	Yes		
SRP-DSS	No	No	Yes	Yes	Yes		
SRP-RSA	No	No	Yes	Yes	Yes		
Kerberos	No	No	Yes	Yes	Yes		
DH-ANON (insecure)	No	Yes	Yes	Yes	Yes		
ECDH-ANON (insecure)	No	No	Yes	Yes	Yes		
GOST R 34.10-94 / 34.10-2001[46]	No	No	Yes	Yes	Yes		Proposed in RFC drafts

TLS1.3 Adoption

- Slowly in browsers
- Breaks Bluecoat like interception
- Not always enabled by default
- Adoption slow