SSH, SSL, and IPsec: wtf?

Eric Rescorla
RTFM, Inc.
`ekr@rtfm.com`

What are we trying to accomplish?

- Alice, Bob want to talk to each other
- But they're worried about attack
  - How do you know you're talking to the right person?
  - How do you know people can't listen to your conversation
  - How do you know people can't change your conversation?
- We want to build a system that protects against these attacks

Terminology Dump 1: Attacker Capabilities

Passive Attacker doesn't send anything.

Active Attacker is allowed to send traffic.

On-path Attacker is on the communications path between A and B.

- Sees all traffic
- Can seamlessly impersonate either side

Off-path Attacker is not on communications path between A and B

- Can't see traffic between A and B.
- Can sometimes send traffic as either (subject to address filtering).

Terminology Dump 2: Security Properties

Confidentiality Information being transmitted is kept secret from attackers

Data Origin Authentication Receivers can determine the origin of traffic.

Message Integrity Tampering of traffic can be detected.

Third-party Verifiability A party not involved in the initial communication can verify what happened. (Often misleadingly called *non-repudiation*)

A simple problem: remote authentication

- You're a Web server
  - X connects to you claiming to be Alice
  - How can you tell?
- Assumptions:
  - All you have is the network traffic
    - Can send messages to X
    - Receive X's response
  - Attackers can forge but not view, intercept, or modify traffic
  - You have some prior relationship with Alice

Remote authentication: basic ideas

- Alice needs to be able to do something others can't do
  - Generally, compute some function
    - But why can't X do that?
- How do we break the symmetry?
  - Give Alice more resources
  - *Give Alice some secret*

One-sided authentication with shared secrets

- Assume Alice and Bob share a secret S_{ab}
  - Alice needs to prove possession of S_{ab}
  - (Assume Alice authenticates Bob some other way)
- Simple approach:
  - Bob and Alice both store S_{ab}
  - Alice sends Bob S_{ab}
  - Bob does `memcmp()`.

Problems with the previous scheme

Snooping. an attacker who is on-path can capture the password and *replay* it

Hijacking. an attacker can wait for you to exchange the password and then take over the connection

One-way authentication. how does Alice authenticate Bob?

Fixing snooping

- Alice doesn't send S_{ab} over the wire
  - Instead she computes some function f
  - And sends $f(S_{ab})$
- What properties does f need?

  1st Preimage Resistant hard to compute S_{ab} from $f(S_{ab})$

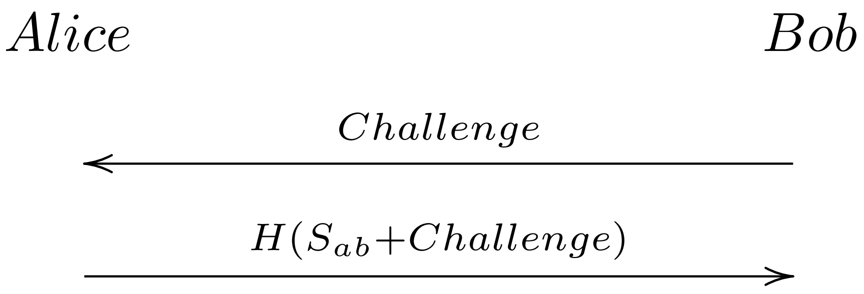
  2nd Preimage Resistant hard to find S' st $f(S') = f(S_{ab})$
- Luckily, we have such functions

Cryptographic hash functions

- Basic idea: one-way function (also called *message digests*)
  - Take an arbitrary length bit string m and reduce it to 100-200 (b) bits
  - $H(m) = h$
- Hash functions are preimage resistant
  - Takes approximately 2^b operations to find m given h
- Hash functions are collision resistant
  - Takes approximately $2^{b/2}$ operations to find m, m' st. $H(m) = H(m')$
- Popular algorithms: MD5, SHA-1, SHA-256

Challenge-Response

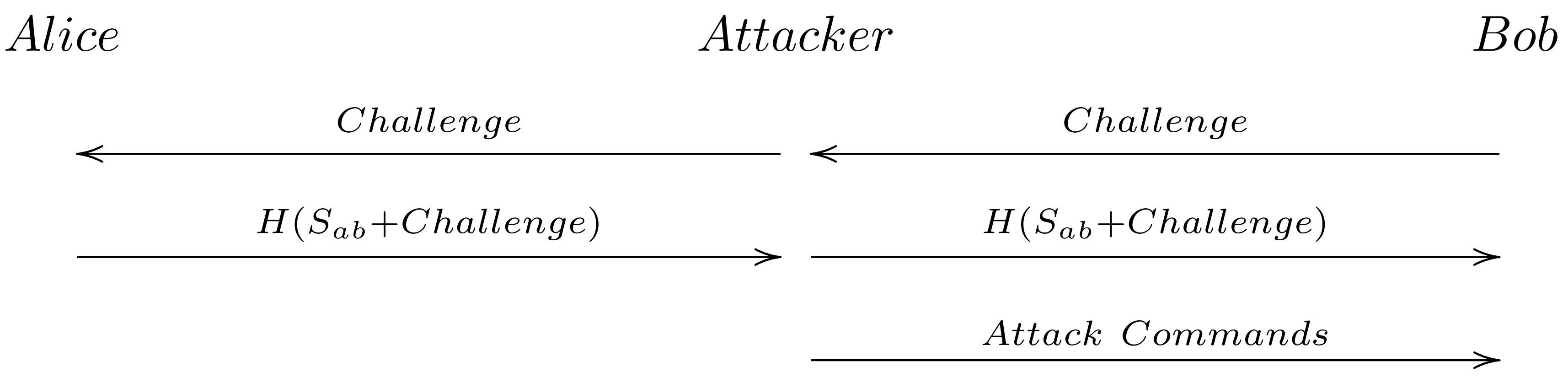
- So, Alice just sends $H(S_{ab})$, right?
  - Wrong
  - This becomes the new secret
  - So we still have a replay attack problem
- Bob needs to force Alice to compute a new function each time



- Challenge needs to be unique for every exchange
  - Does *not* need to be unpredictable

Why mutual authentication?

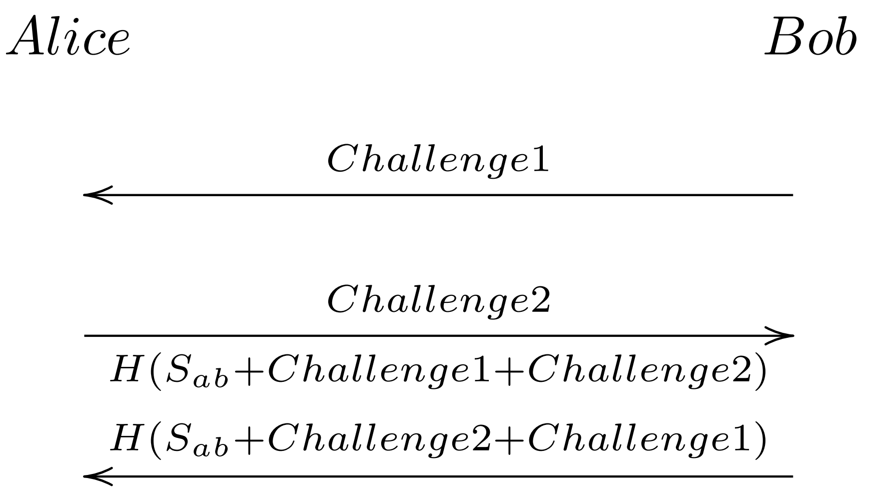
- We assumed that Alice was talking to Bob
  - But how does Alice know that?
  - She can't trust the network
  - What if she's connecting to the attacker



- Alice has just logged in for the attacker
  - He can issue any commands he wants (oops!)

Adding mutual authentication

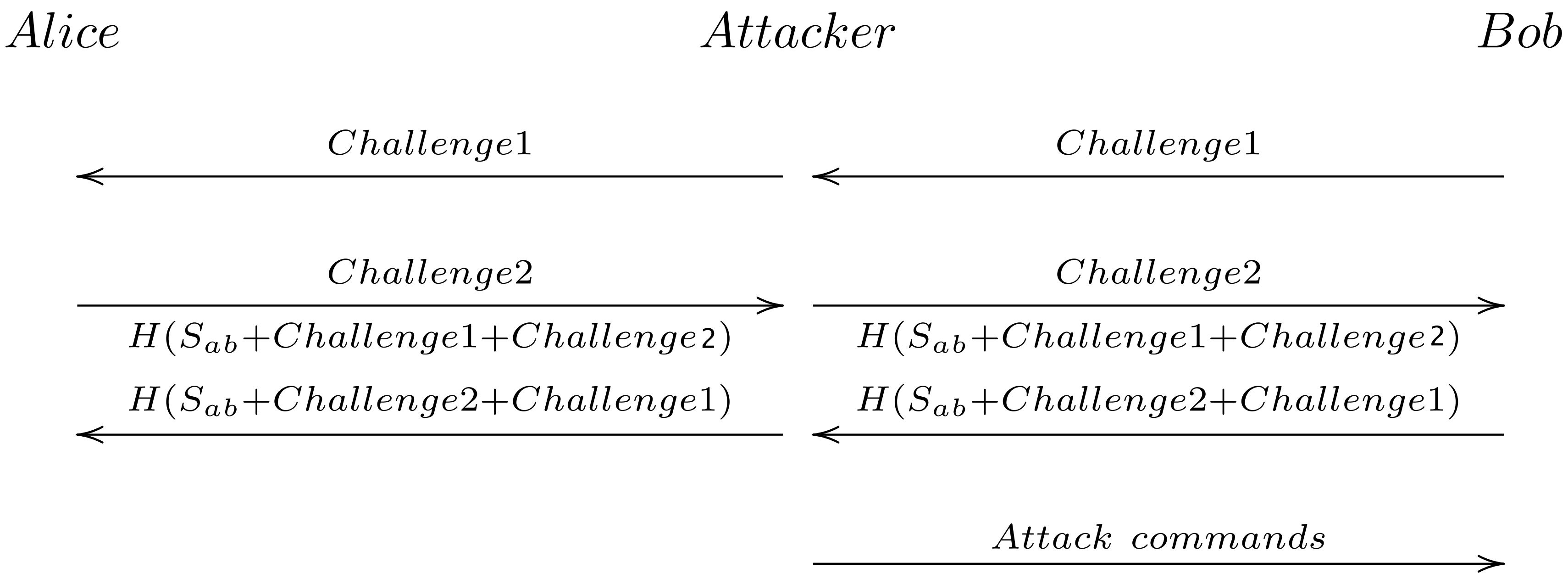
- We already know how to authenticate Alice
  - Now we need to authenticate Bob
  - Just reverse the procedure


$Alice$
Bob
$Challenge1$
$Challenge2$
$H(S_{ab}+Challenge1+Challenge2)$
$H(S_{ab}+Challenge2+Challenge1)$

- Each side needs to control its own challenges
  - Otherwise we have replay issues again

Hijacking

- This protocol still has a hijacking problem

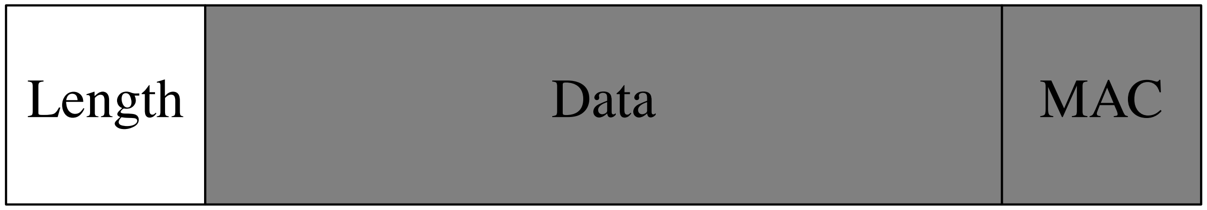


Alice
Attacker
Bob
$Challenge1$
$Challenge1$
$Challenge2$
$Challenge2$
$H(S_{ab}+Challenge1+Challenge2)$
$H(S_{ab}+Challenge1+Challenge2)$
$H(S_{ab}+Challenge2+Challenge1)$
$H(S_{ab}+Challenge2+Challenge1)$
$Attack\ commands$

- We need to authenticate the data
  - Not just the initial handshake

Authenticating data

- Break the data into records
  - Attach a *message authentication code* (MAC) to each record
  - Receiver verifies MACs on record



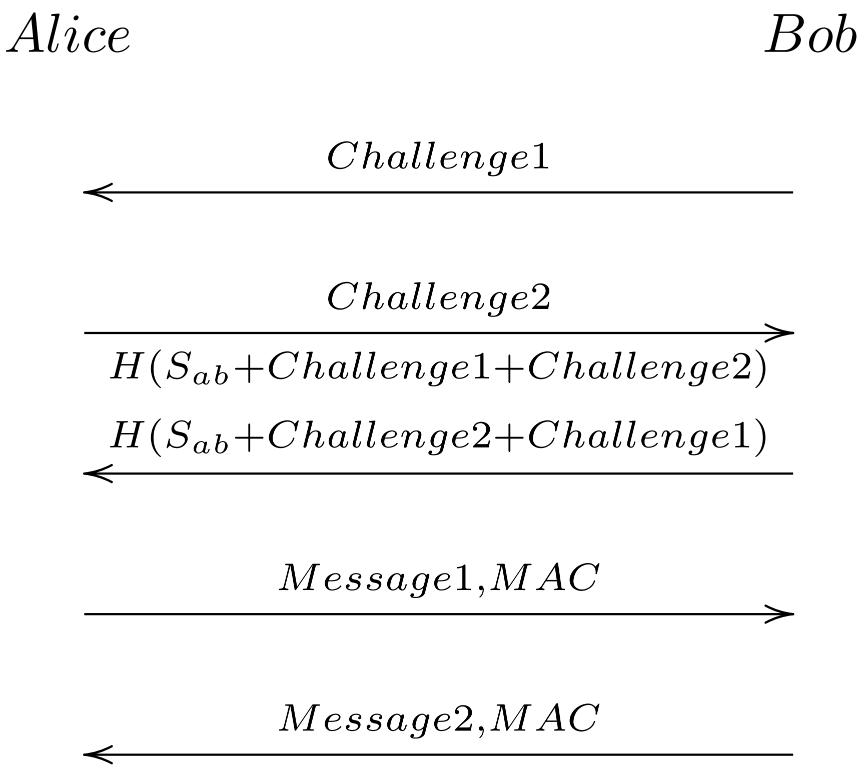
A message authentication code? Dude, wait, what?

- What's a MAC?
  - A one-way function of the key and some data
  - $F(k, data) = x$
    - x is short (80-200 bits)
    - Hard to compute x without k
    - Hard to compute $data$ even with k, x
- This sounds kinda like a hash
  - MACs are usually built from hashes
    - World's simplest MAC: $H(k + data)$ (this has problems)
- Popular MACs: HMAC

Where does the key come from?

- We want a key that's unique to this connection
  - And tied to both sides
  - Get it from the challenge-response handshake
- First attempt: $K = H(S_{ab} + Challenge1 + Challenge2)$
  - But now the key is the same in both directions
  - And the same as the challenge response!
  - Allows *reflection* attacks
- Second attempt
  - $K_{a \to b} = H(S_{ab} + "AB" + Challenge1 + Challenge2)$
  - $K_{b \to a} = H(S_{ab} + "BA" + Challenge1 + Challenge2)$

World's simplest security protocol

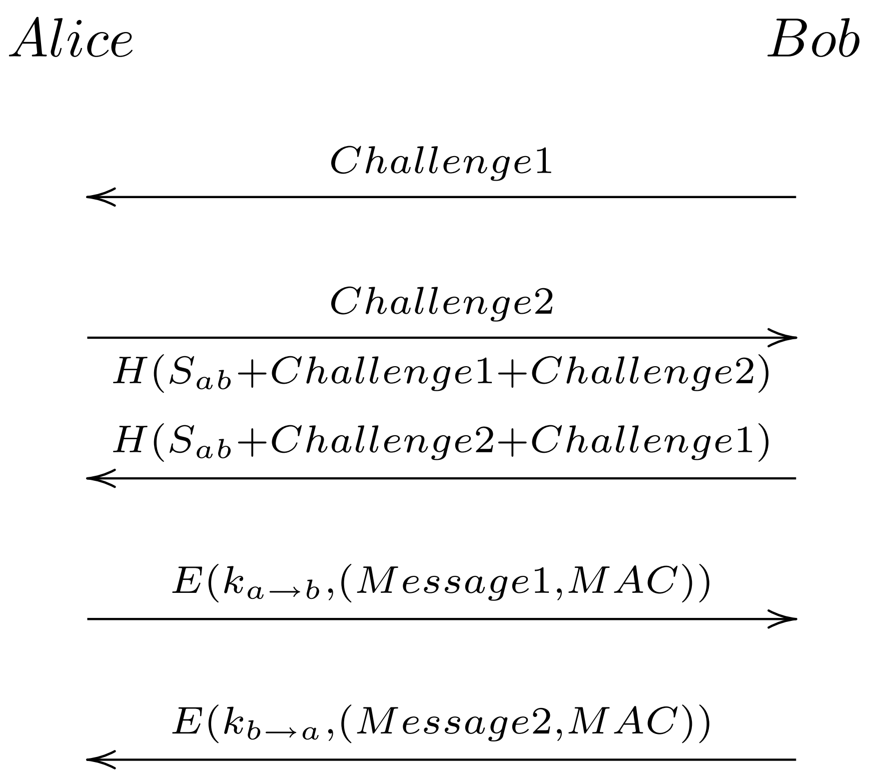

$Alice$
Bob
$Challenge1$
$Challenge2$
$H(S_{ab}+Challenge1+Challenge2)$
$H(S_{ab}+Challenge2+Challenge1)$
$Message1, MAC$
$Message2, MAC$

- Each side knows who the other is
- All messages are authenticated
  - But they're not confidential
  - So don't send any secret information

Symmetric Encryption

- We have two functions E, D st.
  - $E(k, Plaintext) = Ciphertext$
  - $D(k, Ciphertext) = Plaintext$
  - These are easy to compute
  - Either function is hard to compute without k
- Popular encryption algorithms: DES, 3DES, AES, RC4

A (mostly) complete channel security protocol


$Alice$
Bob
$Challenge1$
$Challenge2$
$H(S_{ab}+Challenge1+Challenge2)$
$H(S_{ab}+Challenge2+Challenge1)$
$E(k_{a \rightarrow b},(Message1,MAC))$
$E(k_{b \rightarrow a},(Message2,MAC))$

Rather than encrypting the MAC, we should encrypt the message and MAC the ciphertext

- Each side knows who the other is
- All messages are authenticated
- *All messages are confidential*

So, we're done, right?

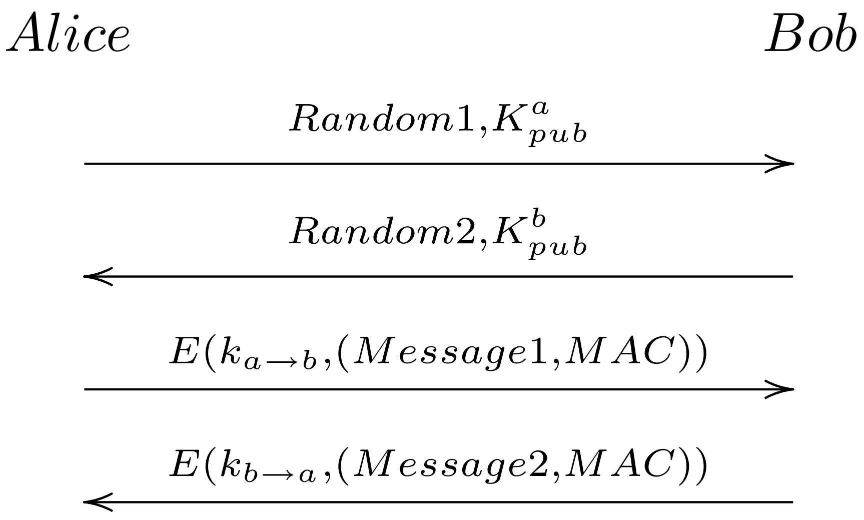
- How do Alice and Bob get S_{ab}?
- Some out of band channel
  - Send a letter—do you trust USPS?
  - Meet in person—airplane tickets are expensive
  - Guys with briefcases handcuffed to their wrists?
- All of these are pretty inconvenient
  - We can do better

Diffie-Hellman Key Agreement

- Each side has two keys ("public" and "private")
  - You publish the public key but the private key is secret
  - $F(K^a_{pub}, K^b_{priv}) = F(K^b_{pub}, K^a_{priv}) = ZZ$
  - You need at least one private key to compute ZZ
- This is crypto rocket science–but you don't need to understand how it works

Not actually true. Diffie-Hellman is not that complicated and you do need to understand how it works!

Using Diffie-Hellman


$Alice$
Bob
$Random1, K^a_{pub}$
$Random2, K^b_{pub}$
$E(k_{a \rightarrow b}, (Message1, MAC))$
$E(k_{b \rightarrow a}, (Message2, MAC))$

- Each side sends its public key
- The other side combines its private key with the other side's public key to compute ZZ
- The traffic keys are generated from ZZ

We need four different keys:
— Encryption keys Alice -> Bob and Bob -> Alice;
— MAC keys Alice -> Bob and Bob -> Alice

Man-in-the-middle attack

$$\begin{tikzcd}
Alice & Attacker & Bob \\
{} \arrow[r, "{Random1,K^{a}_{pub}}"] & {} \arrow[r, "{Random1,K^{A}_{pub}}"] & {} \\
{} & {} \arrow[l, "{Random2,K^{A}_{pub}}"'] & {} \arrow[l, "{Random2,K^{b}_{pub}}"'] \\
{} \arrow[r, "{E(k_{a \rightarrow A},(Message1,MAC))}"] & {} \arrow[r, "{E(k_{A \rightarrow b},(Message1,MAC))}"] & {} \\
{} & {} \arrow[l, "{E(k_{A \rightarrow a},(Message2,MAC))}"'] & {} \arrow[l, "{E(k_{b \rightarrow A},(Message2,MAC))}"']
\end{tikzcd}$$

- Each side thinks it's talking to the other
  - This is what happens when you don't authenticate
- Alice and Bob need some way to authenticate each other's public keys

Digital Signatures

- Remember MACs?
- There's a public key version of this
  - "Sign" with K_{priv}
  - "Verify" with K_{pub}
- A signed message can only be generated by someone who has the private key
- Popular algorithms: RSA, DSA, ECDSA

Public key distribution

- Public key cryptography is one piece of the puzzle
  - But only one piece
- I can verify a signature came from a given key
  - But where do I get that key from?
- We could have a global directory
  - Obvious scaling problems here
- What if I could give you a credential vouching for your public key?

Certificates

- Digital signatures let us do exactly that
- Create a central *certificate authority* (CA)
  - Alice proves her identity to the CA
  - The CA gives her a signed message *"Alice's public key is X"* (a certificate)
- Anyone can verify this certificate
  - As long as they have the public key of the CA
  - This key is compiled into the software
- Popular CAs: VeriSign, Thawte, GoDaddy

LetsEncrypt — free, easy to use

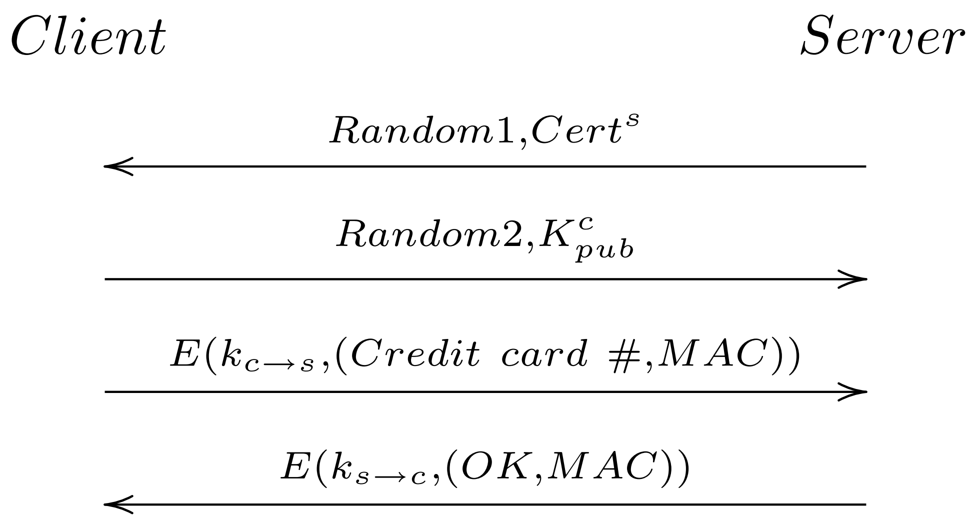
Diffie-Hellman with certificates

$$\begin{tikzcd}
Alice & & Bob \\
{} \arrow[rr, "Random1{,}Cert^a"] & & {} \\
{} & & {} \arrow[ll, "Random2{,}Cert^b"'] \\
{} \arrow[rr, "E(k_{a \to b}{,}(Message1{,}MAC))"] & & {} \\
{} & & {} \arrow[ll, "E(k_{b \to a}{,}(Message2{,}MAC))"']
\end{tikzcd}$$

- Certificates contain DH public keys
- Each side can authenticate the other
  - This is actually a bug
  - Certificates are too inconvenient for users to get
  - And the user doesn't always need to be authenticated
  - Or is authenticated some other way

One-way authentication with PKC

- One side (server) has a certificate
- The other side (client) makes up a random key pair


$Client$
$Server$
$Random1, Cert^s$
$Random2, K^c_{pub}$
$E(k_{c \to s}, (Credit\ card\ \#, MAC))$
$E(k_{s \to c}, (OK, MAC))$

- This authenticates the server but not the client
- We can do a similar trick with RSA
  - Encrypt with public key, decrypt with private key
- This is the main operational mode for SSL/TLS

Well, it was. Now, we're moving toward forward secrecy (next slide)

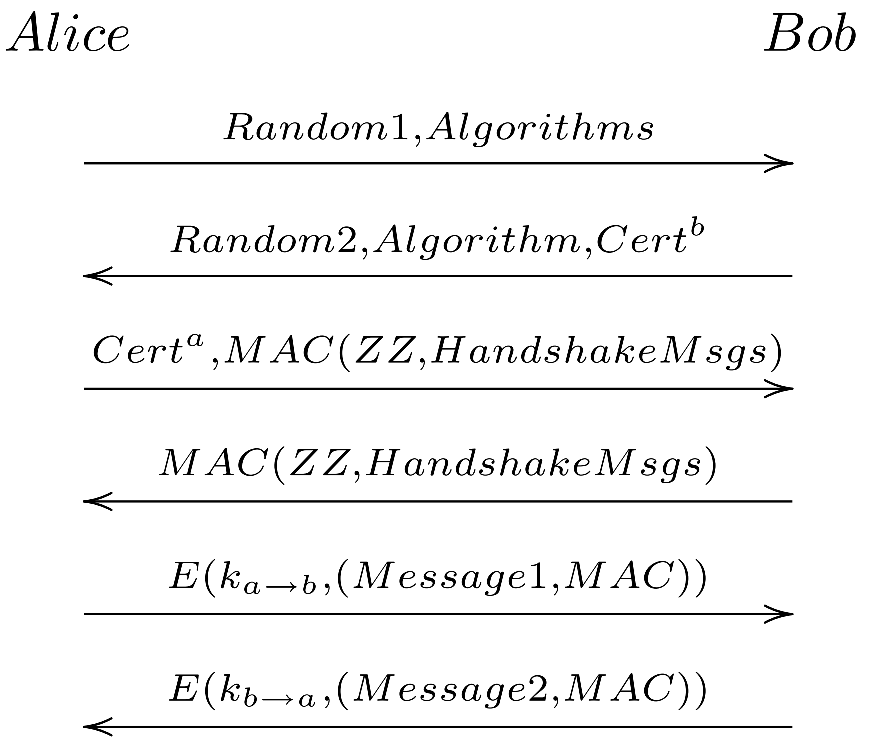
Perfect Forward Secrecy

- What happens if one side's computer is compromised?
  - Attacker gets private key
  - Can decode all communications by that side
- Fix: have certificates with signature keys (RSA, DSA)
  - Generate a random DH key for each handshake
  - Sign it with your signature key
- Compromise of private key doesn't affect past traffic
  - But you can MITM future connections
- This is the main operational mode for IPsec and TLS 1.3

Algorithm negotiation

- There are a lot of choices here
  - Who authenticates,
  - Public key algorithm
  - Digest algorithm
  - Encryption algorithm
- Each make sense in some scenarios
  - A good protocol is adaptable
- This means some kind of negotiation
  - This needs to be protected to prevent downgrade attacks

A complete channel security protocol


$Alice$
Bob
$Random1, Algorithms$
$Random2, Algorithm, Cert^b$
$Cert^a, MAC(ZZ, HandshakeMsgs)$
$MAC(ZZ, HandshakeMsgs)$
$E(k_{a \to b}, (Message1, MAC))$
$E(k_{b \to a}, (Message2, MAC))$

Again, we should MAC the ciphertext rather than encrypting the MAC

Secure Sockets Layer (SSL)

- Originally a Netscape proprietary protocol
- Target application: e-commerce
  - What people thought the Web was for in 1994
  - Objective: send my credit card to Amazon securely
- Basic principles (ca. 1994)
  - The server is authenticated (via certificate)
  - The client is unauthenticated
  - This should be easy to plug in to both sides

SSL/TLS History (1)

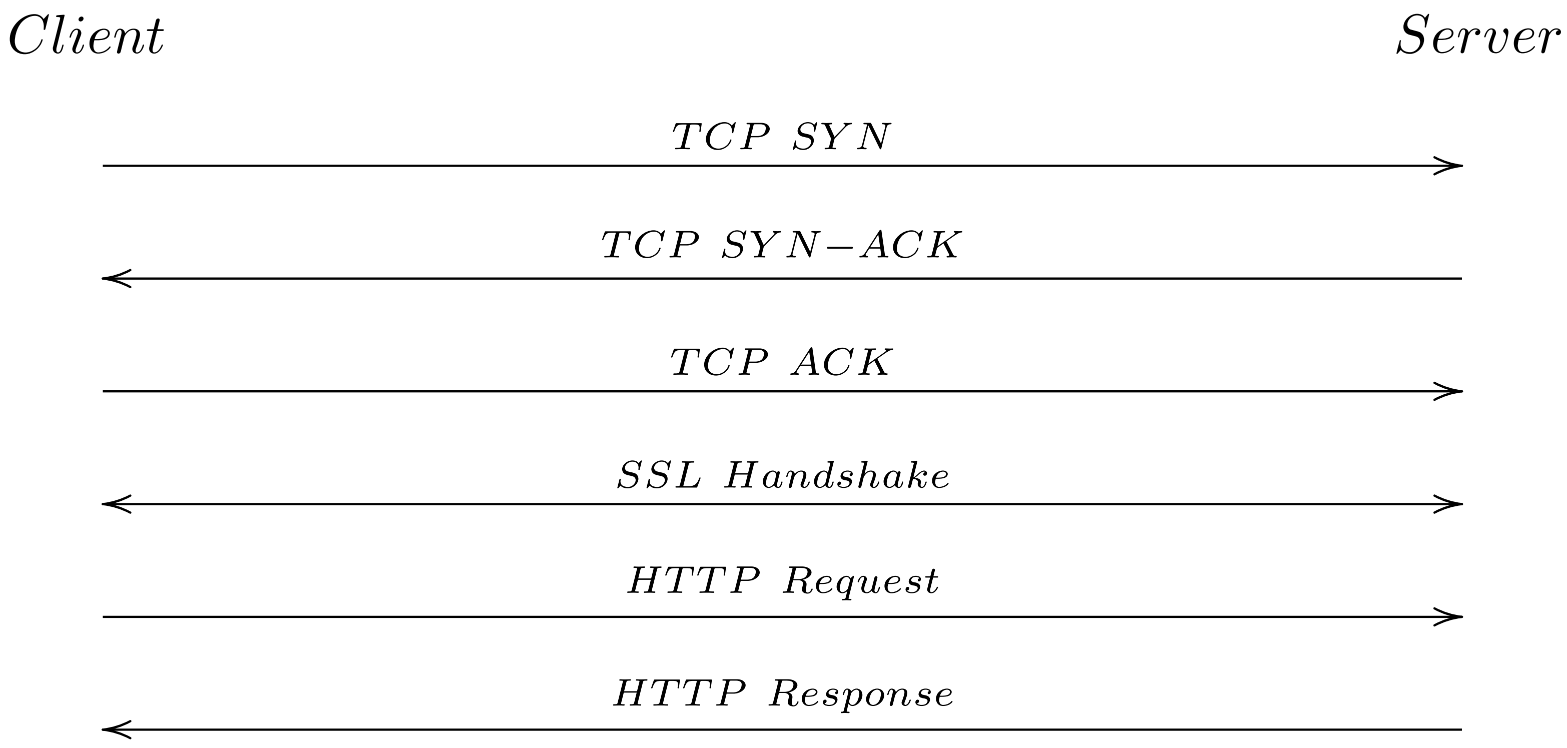
- SSLv1 (never released)
  - Designed by Kipp Hickman
  - Severe security flaws (immediately obvious to anyone who knew crypto)
- SSLv2
  - Hickman again (after being beaten up by others)
  - Modest security flaws (truncation attacks, downgrade)
  - Very widely deployed
- SSLv3
  - Freier, Karlton, Kocher
  - Fixes the above problems

SSL/TLS History (2)

- Transport Layer Security (TLS) 1.0 (RFC 2246)
  - First standardized version of SSL
  - Modest improvements to key derivation
- TLS 1.1 (RFC 4346)
  - Fixes for modest security flaws
- TLS 1.2 (RFC 5246)
  - Flexibility for hash functions (thanks Dr. Wang!)
- As you can see, this is in maintenance mode

TLS 1.3 is in progress, major changes:
— No RSA key exchange (for forward secrecy);
— authenticated encryption modes;
— 0 RTT handshakes

HTTP over SSL (HTTPS)



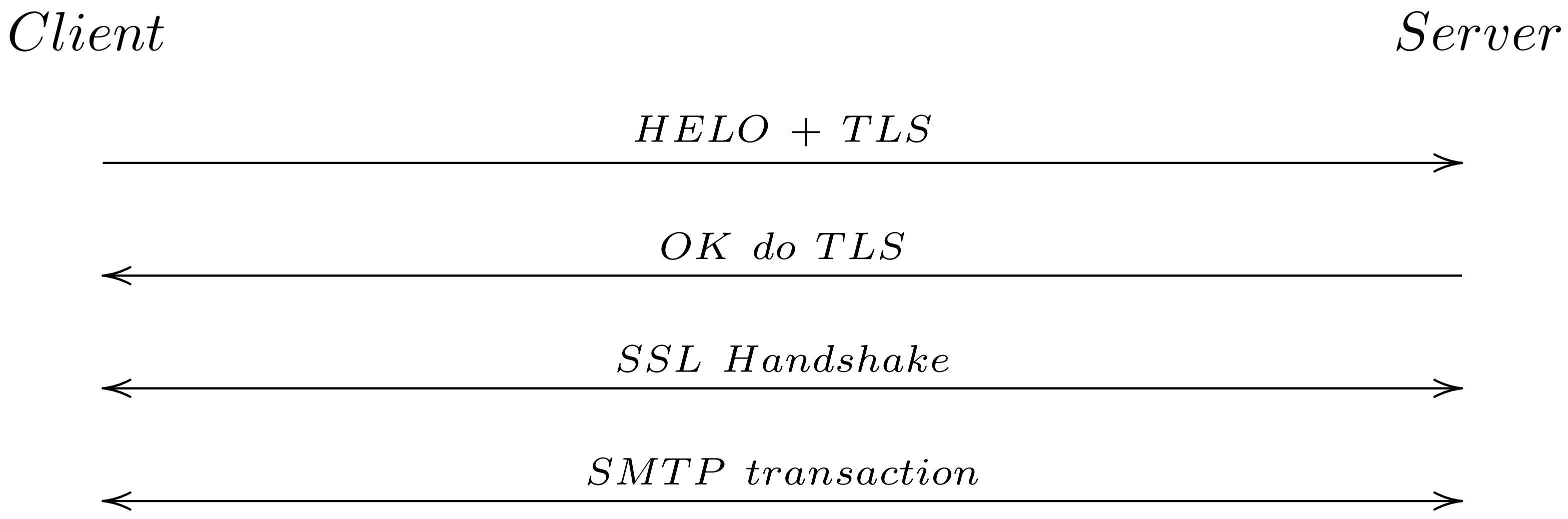
- The client *knows* that the server expects HTTPS
  - It's in the URL `https://www.example.com/`
  - It's on a separate port
- The server's certificate has its domain name (`www.example.com`)

SSL Session Resumption

- Asymmetric (private key) operations are expensive
  - And HTTPS tends to involve a lot of SSL/TCP connections
- Caching pays off here
  - Each handshake establishes a *session*
  - Clients can *resume* the session with the same keying material
  - Thus skipping the key exchange

Upward Negotiation

- What if the client and server don't know each other's capabilities
  - Would be nice to discover them
  - And automatically upgrade to TLS
- Example: SMTP



- Of course, this allows downgrade attacks

DoS Attacks on SSL/TLS

- Resource consumption
  - Public key operations are expensive
    - Client can force the server to do a lot of them
    - But not blindly (TCP handshake)
  - State on the server side
- SSL/TLS connection runs over TCP
  - TCP connections are easy to DoS
  - SSL/TLS can't protect you from this
  - Needs to be at a lower layer

Datagram TLS (RFC 4347)

- TLS requires a reliable channel
  - The handshake is in sequence
  - The data records depend on each other
  - In practice this means TCP
- What about unreliable channels?
  - DTLS is a slight modification of TLS
  - Reliability for the handshake
  - Record independence
- More DoS resistance (more on this later)

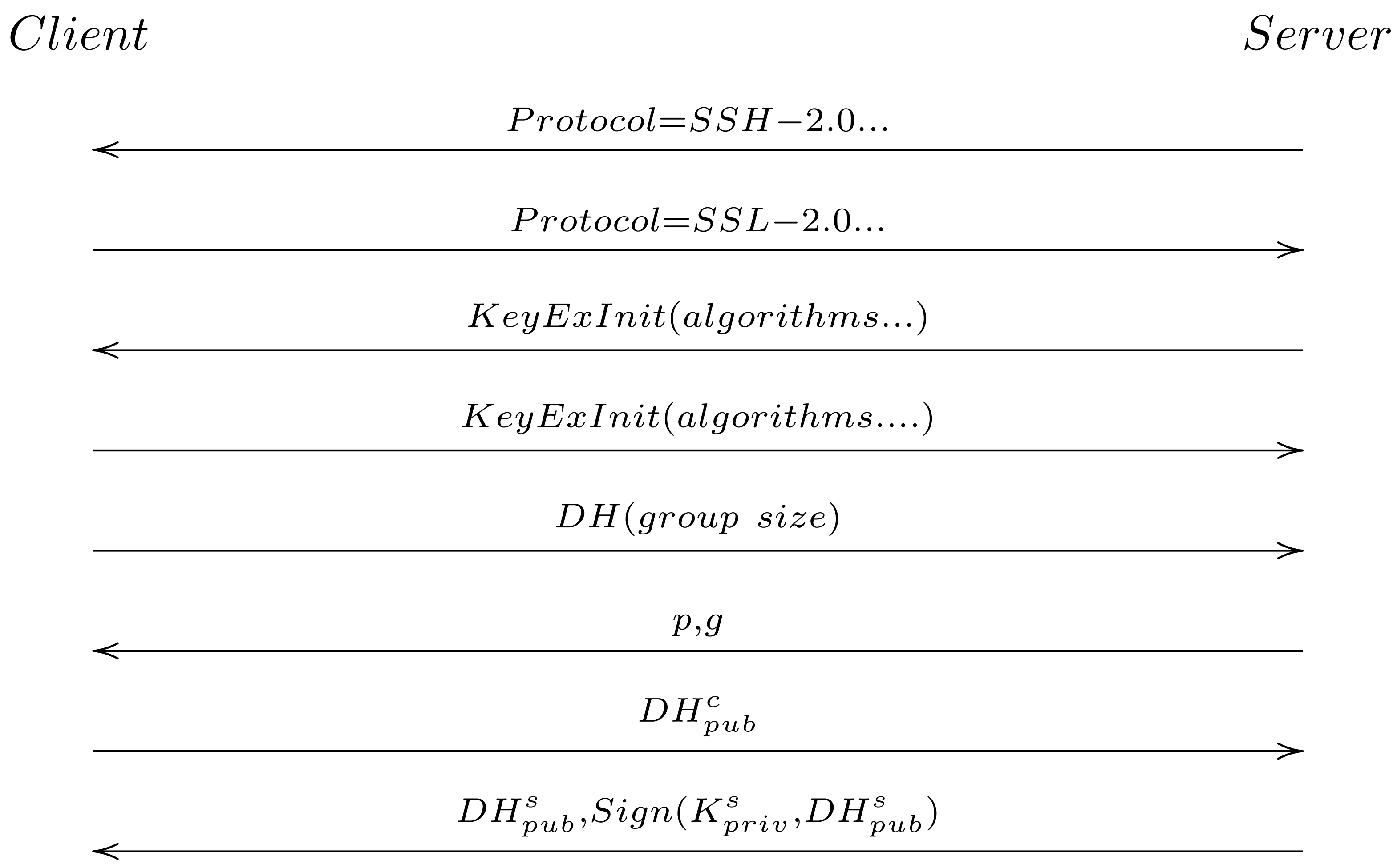
Secure Shell (SSH)

- Originally designed by Tatu Ylonen
  - Replacement for rsh
  - Now the standard tool for secure remote login
  - A lot of authentication mechanisms
- Other features
  - Remote X
  - File transfer
  - Port forwarding
- Original version was seriously broken
  - Later standardized versions are better
  - Transport protocol looks a lot like TLS

SSH leap of faith authentication

- No certificates–server just has a raw public key
  - The server provides the key when the client connects
  - The client stores the server's key on first connection
  - Any changes in the key are an error
- The key can be authenticated out of band
  - The server operator tells the client the key fingerprint (hash) over the phone
  - But only the most paranoid people do this
- This was considered insanity at the time
  - Now it's considered clever

SSL Key Exchange Protocol

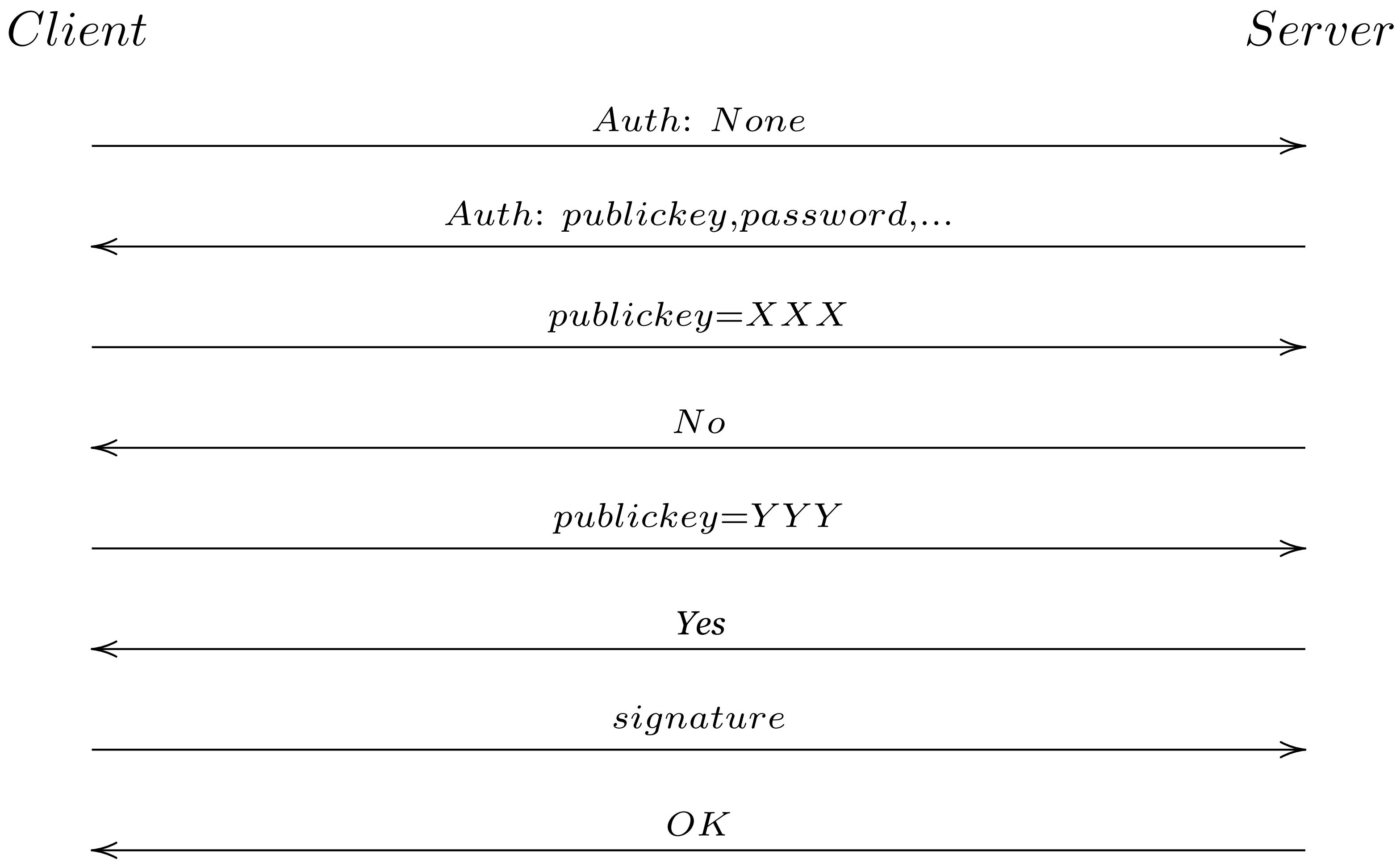


$Client$
$Server$
$Protocol{=}SSH{-}2.0...$
$Protocol{=}SSL{-}2.0...$
$KeyExInit(algorithms...)$
$KeyExInit(algorithms....)$
$DH(group\ size)$
p,g
DH^c_{pub}
$DH^s_{pub},Sign(K^s_{priv},DH^s_{pub})$

SSH Client Authentication

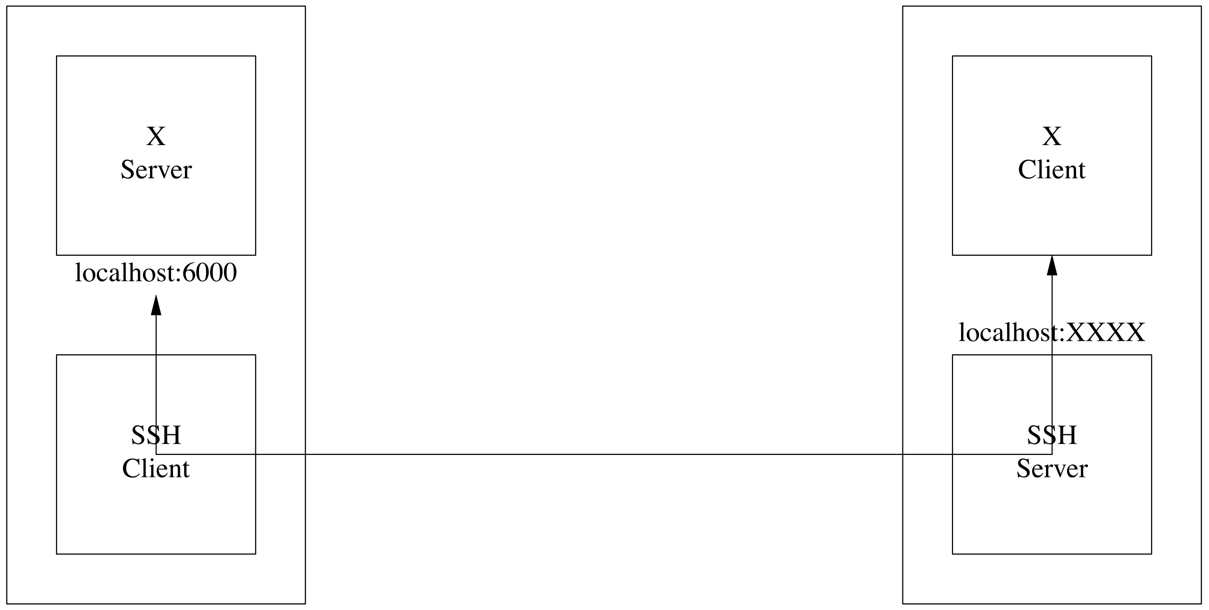
- Server is authenticated first
- Client is then authenticated
  - Raw password
  - Challenge-response
  - Public key
  - GSS-API
  - Kerberos
- Mechanisms are negotiated

SSH Client Authentication Protocol



Port Forwarding

- SSH provides a port forwarding feature
- Example: X11 remote



- SSH server does `setenv DISPLAY localhost:XXXX`
- Apps just automatically work

Secure Remote Shell

- SSH is backward compatible with rsh
  - So other applications can be securely remoted
  - Even without port forwarding
- Examples
  - CVS
  - rsync
  - dump/restore
- Apps don't need security, just remote access

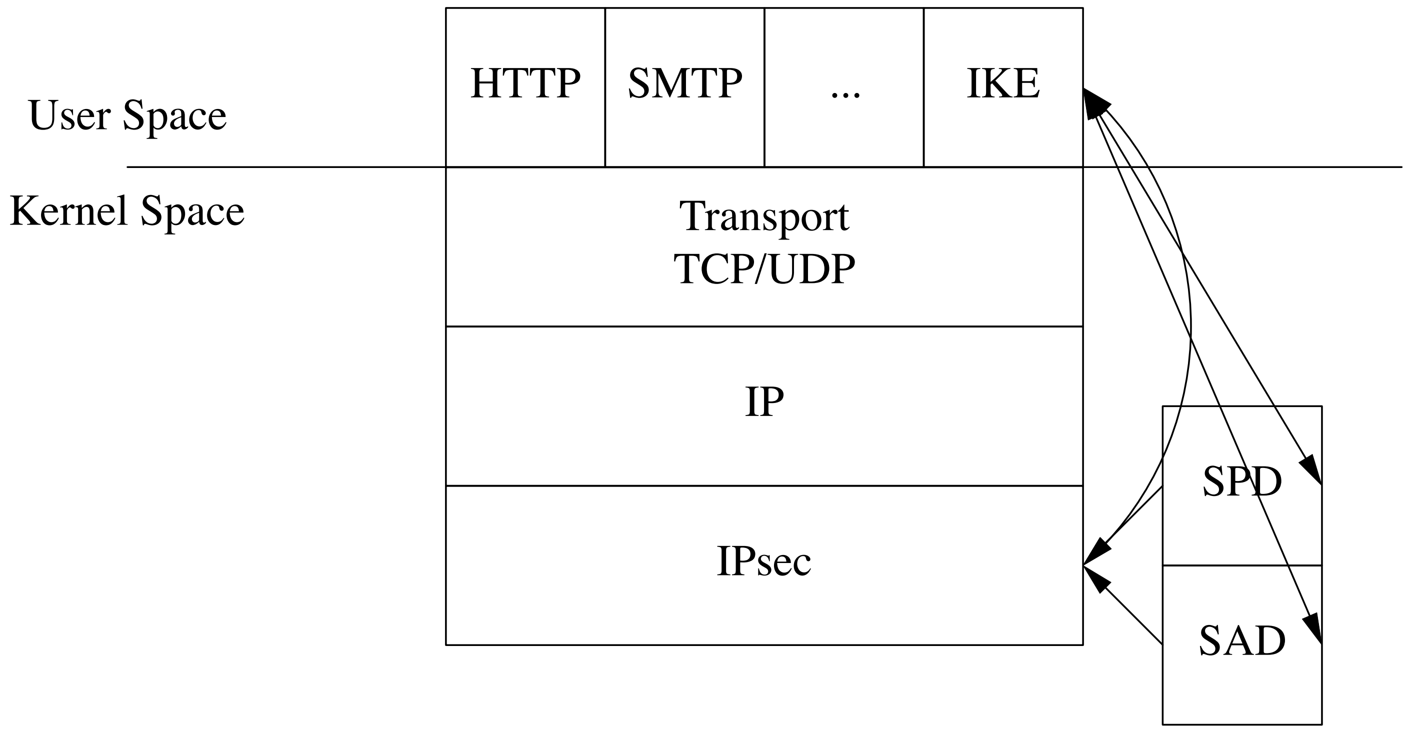
IPsec: IP Security

- Basic idea: secure IP datagrams
  - Instead of at application layer like TLS or SSH
- Why was this considered a good idea?
  - Secure all traffic, not just TCP/UDP
  - Automatically secure applications
    - Without any change to the application
  - Built-in-firewalling/access control

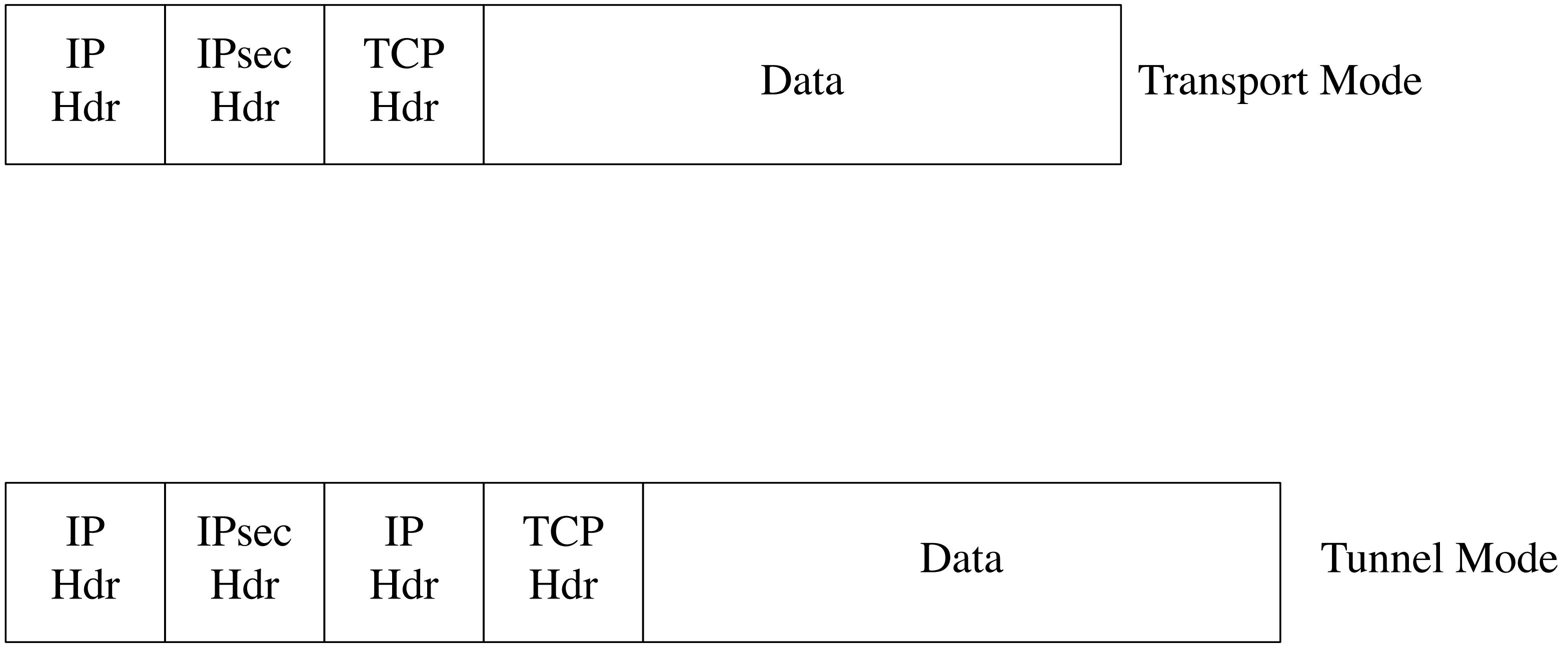
IPsec history

- Work started in 1992-1993
- General agreement on packet formats early on
  - Though confusion about integrity vs. authentication
- Key agreement was very controversial
  - Design issues
  - IPR issues
- First "proposed standards" published in 1998
  - Mishmash of IKE, ISAKMP, OAKLEY
- Complaints about clarity and complexity
  - IKEv2 approved in 2005

IPsec architecture

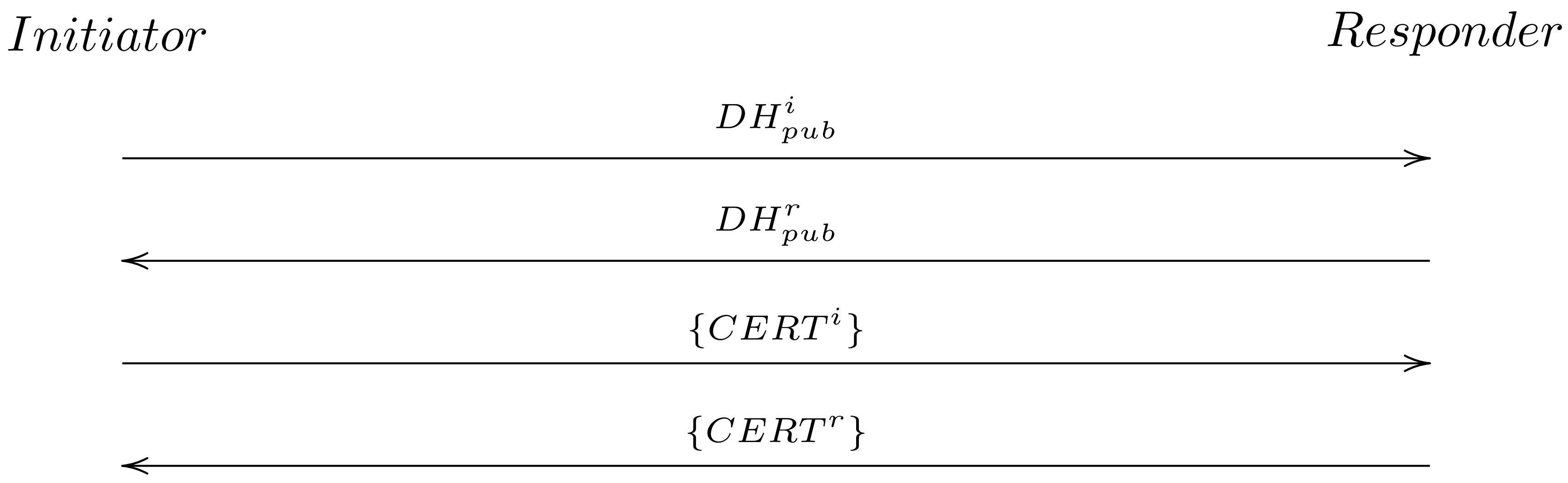


IPsec Packet Formats



IKE “Anonymity”

- The handshakes we’ve seen leak your identity to passive attackers
  - Arguably this is bad
  - IKE tries to stop this



- An active attacker can get the initiator’s identity

IKE DoS prevention

- Objective: prevent blind DoS attacks

$$\begin{tikzcd}
Initiator & & Responder \\
{} \arrow[rr, "DH^i_{pub}"] & & {} \\
{} & & {} \arrow[ll, "Ticket"'] \\
{} \arrow[rr, "{DH^i_{pub}, Ticket}"] & & {} \\
{} & & {} \arrow[ll, "DH^r_{pub}"'] \\
{} \arrow[rr, "\{CERT^i\}"] & & {} \\
{} & & {} \arrow[ll, "\{CERT^r\}"']
\end{tikzcd}$$

- Ticket has to be stateless

IPsec Status

- Many implementations
  - Windows, OS X, Linux, FreeBSD, IOS...
- Nearly all deployments are in VPN settings
- And peopel are cutting over to SSL/VPN
  - Semi-manual configuration
- This is not what was intended
- Widely regarded as a semi-failure

What was wrong with IPsec?

- Complexity
- Time to market
- Wrong design goals
- Hard to use

Final thoughts

- All of these protocols look strikingly alike
  - To some extent they were designed by the same people
  - But also there appear to only be so many ways to do this
- All have gone through multiple revisions
  - This is really hard to get right
  - Even when you ave experienced people
  - Don't invent your own
- Usage models matter
  - SSL/TLS and SSH got this right
  - IPsec did not