CSE P 590 / CSE M 590 (Spring 2010)

# Computer Security and Privacy

Tadayoshi Kohno

Thanks to Dan Boneh, Dieter Gollmann, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

# Goals for Today

- Crypto
  - PKIs
  - Protocols
  - SSL
- Users (some more)
- Anonymity
- Research reading

- Lab 1 -- May 17
- HW 3 -- announced soon (due date not May 14)

# Public Key Cryptography

# Advantages of Public-Key Crypto

- Confidentiality without shared secrets
  - Very useful in open environments
  - No "chicken-and-egg" key establishment problem
    - With symmetric crypto, two parties must share a secret before they can exchange secret messages
    - Caveats to come
- Authentication without shared secrets
  - Use digital signatures to prove the origin of messages
- Reduce protection of information to protection of authenticity of public keys
  - No need to keep public keys secret, but must be sure that Alice's public key is <u>really</u> her true public key

# Disadvantages of Public-Key Crypto

- Calculations are 2-3 orders of magnitude slower
  - Modular exponentiation is an expensive computation
  - Typical usage: use public-key cryptography to establish a shared secret, then switch to symmetric crypto
    - E.g., IPsec, SSL, SSH, ...
- Keys are longer
  - 1024+ bits (RSA) rather than 128 bits (AES)
- Relies on unproven number-theoretic assumptions
  - What if factoring is easy?
    - Factoring is believed to be neither P, nor NP-complete
  - (Of course, symmetric crypto also rests on unproven assumptions)

# Exponentiation

- How to compute $M^x$ mod N?
- Say, x = 13
- Sums of power of 2, $x = 8+4+1 = 2^3+2^2+2^0$
- Can also write x in binary, e.g., x = 1101
- Can solve by repeated squaring
  - y = 1;
  - $y = y^2 * M \bmod N$ // $y = M$
  - $y = y^2 * M \bmod N$ // $y = M^2 * M = M^{2+1} = M^3$
  - $y = y^2 \bmod N$ // $y = (M^3)^2 = M^6$
  - $y = y^2 * M \bmod N$ // $y = (M^6)^2 * M = M^{12+1} = M^{13} = M^x$

# Timing attacks

Collect timings for exponentiation with a bunch of messages M1, M2, ... (e.g., RSA signing operations with a private exponent)

Assume (inductively) know $b_3$=1, $b_2$=1, guess $b_1$=1

| i | $b_i = 0$ | $b_i = 1$ | Comp | Meas |
|---|---|---|---|---|
| 3 | $y = y^2 \bmod N$ | $y = y^2 * M1 \bmod N$ | | |
| 2 | $y = y^2 \bmod N$ | $y = y^2 * M1 \bmod N$ | | |
| 1 | $y = y^2 \bmod N$ | $y = y^2 * M1 \bmod N$ | X1 secs | |
| 0 | $y = y^2 \bmod N$ | $y = y^2 * M1 \bmod N$ | | Y1 secs |

| i | $b_i = 0$ | $b_i = 1$ | Comp | Meas |
|---|---|---|---|---|
| 3 | $y = y^2 \bmod N$ | $y = y^2 * M2 \bmod N$ | | |
| 2 | $y = y^2 \bmod N$ | $y = y^2 * M2 \bmod N$ | | |
| 1 | $y = y^2 \bmod N$ | $y = y^2 * M2 \bmod N$ | X2 secs | |
| 0 | $y = y^2 \bmod N$ | $y = y^2 * M2 \bmod N$ | | Y2 secs |

# Timing attacks

- If $b_1 = 1$, then set of { Yj - Xj | j in {1,2, ..} } has distribution with "small" variance (due to time for final step, i=0)
  - "Guess" was correct when we computed X1, X2, ...
- If $b_1 = 0$, then set of { Yj - Xj | j in {1,2, ..} } has distribution with "large" variance (due to time for final step, i=0, and incorrect guess for $b_1$)
  - "Guess" was incorrect when we computed X1, X2, ...
  - So time computation wrong (Xj computed as large, but really small, ...)
- Strategy: Force user to sign large number of messages M1, M2, .... Record timings for signing.
- Iteratively learn bits of key by using above property.

# PKIs

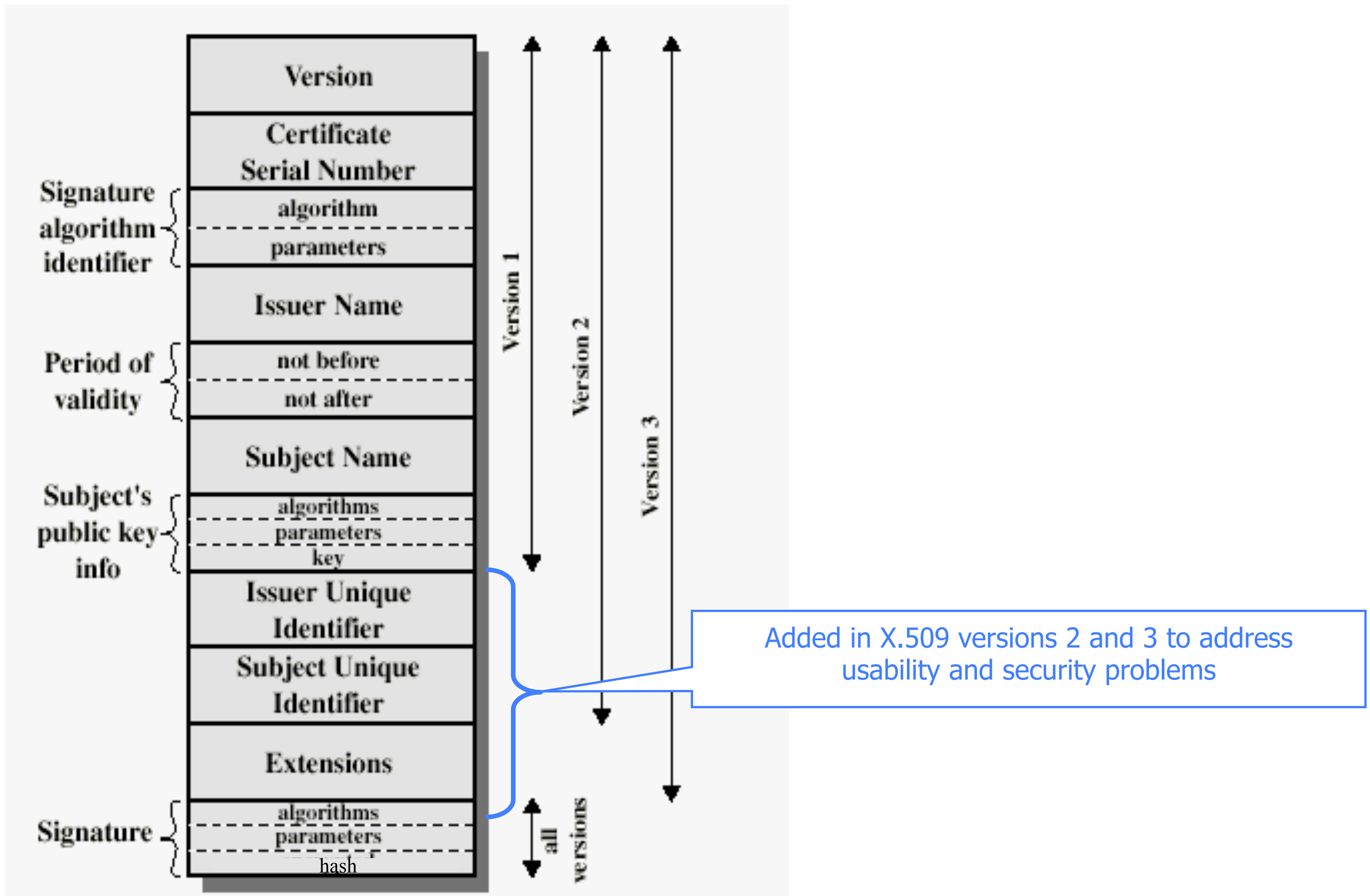
# PKI Overview

- Alice, Bob, Charlie, ..., trust Certificate Authority
- CA signs certificates binding Alice's identity with her public key:
  - Certificate = Alice, $PK_{Alice}$, ..., Sign($PK_{CA}$, "Alice, $PK_{Alice}$, ...")

# X.509 Authentication Service

- Internet standard (1988 onward)
- Specifies certificate format
  - X.509 certificates are used in IPSec and SSL/TLS
- Specifies certificate directory service
  - For retrieving other users' CA-certified public keys
- Specifies a set of authentication protocols
  - For proving identity using public-key signatures
- Does not specify crypto algorithms
  - Can use it with any digital signature scheme and hash function, but hashing is required before signing

# X.509 Certificate



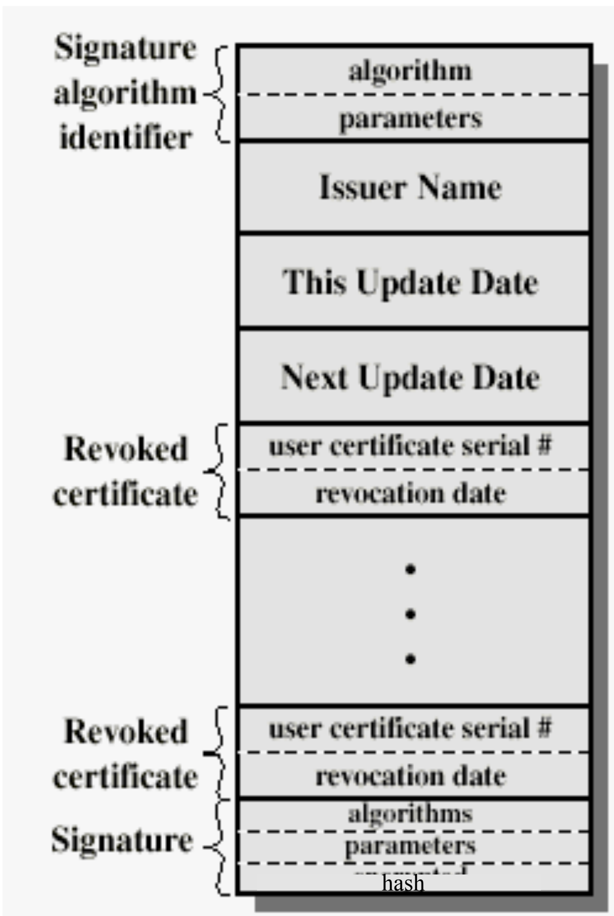
# Certificate Revocation

- Revocation is <u>very</u> important
- Many valid reasons to revoke a certificate
  - Private key corresponding to the certified public key has been compromised
  - User stopped paying his certification fee to this CA and CA no longer wishes to certify him
  - CA's private key has been compromised!
- Expiration is a form of revocation, too
  - Many deployed systems don't bother with revocation
  - Re-issuance of certificates is a big revenue source for certificate authorities

# Certificate Revocation Mechanisms

- Online revocation service
  - When a certificate is presented, recipient goes to a special online service to verify whether it is still valid
    - Like a merchant dialing up the credit card processor
- Certificate revocation list (CRL)
  - CA periodically issues a signed list of revoked certificates
    - Credit card companies used to issue thick books of canceled credit card numbers
  - Can issue a "delta CRL" containing only updates

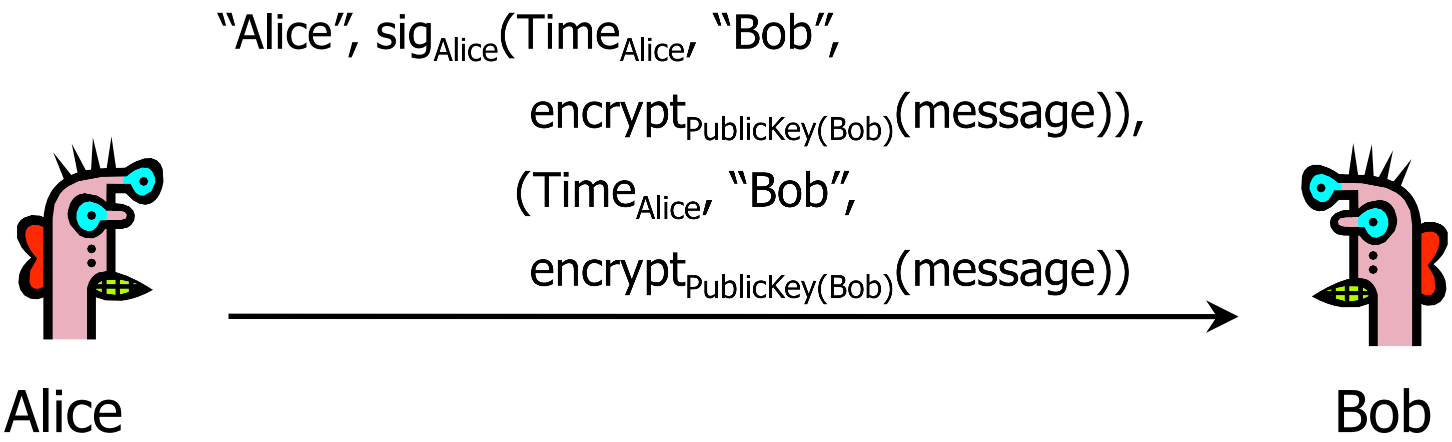
# X.509 Certificate Revocation List




Because certificate serial numbers must be unique within each CA, this is enough to identify the certificate

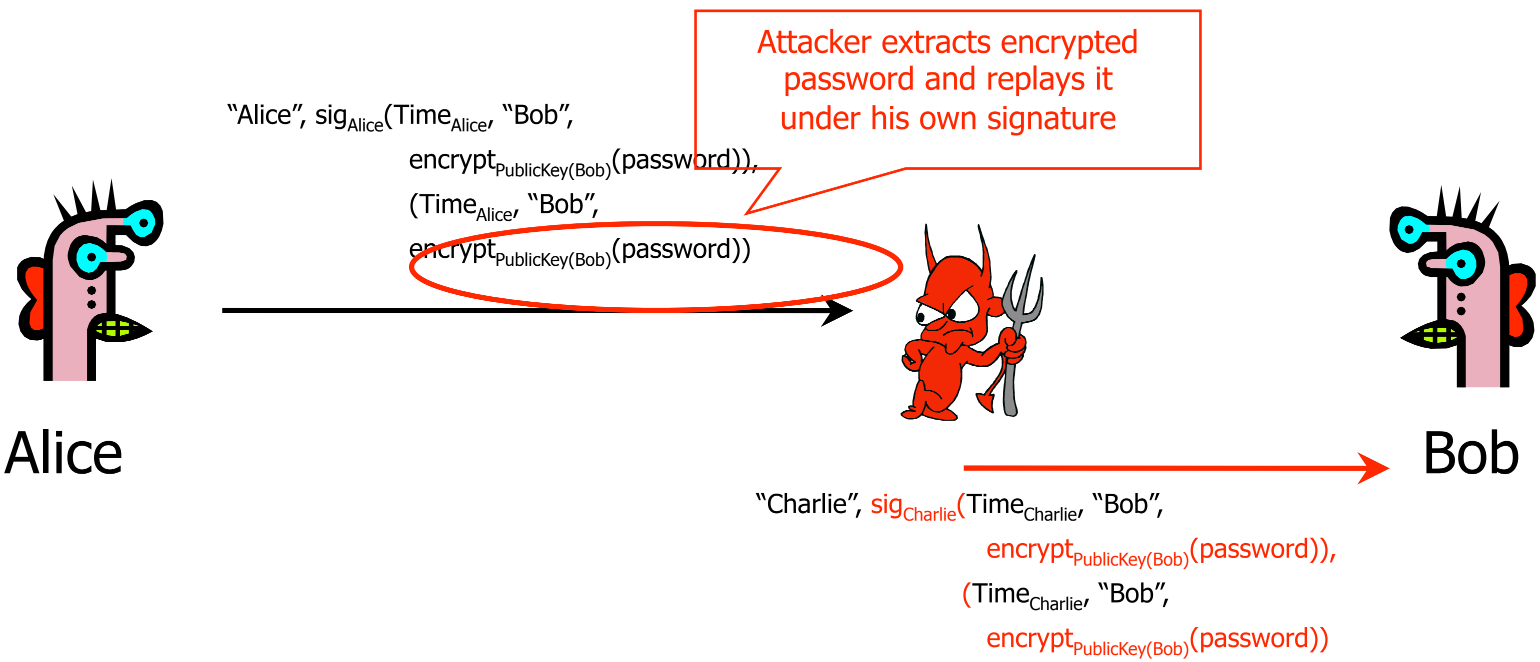
# Some Protocols

# X.509 Version 1



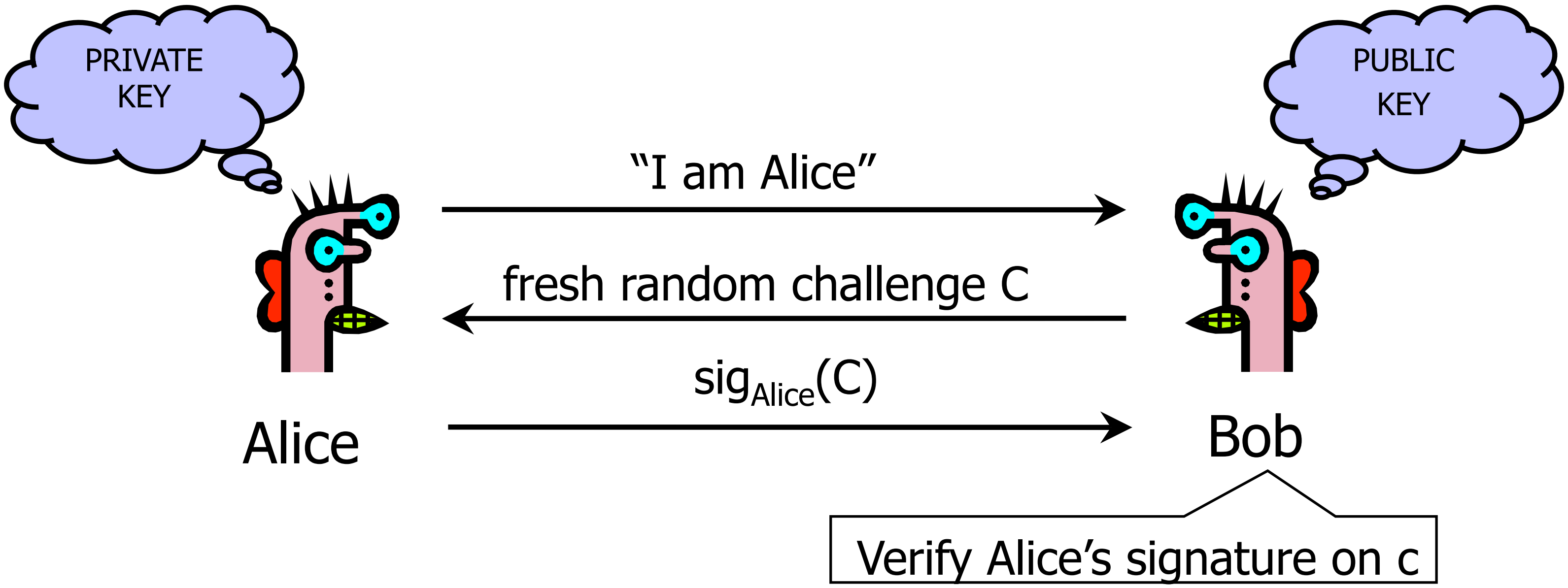

- Encrypt, then sign
  - Goal: achieve both confidentiality and authentication
  - E.g., encrypted, signed password for access control (for next slide: assume one password for whole system)
- Does this work?

# Attack on X.509 Version 1




- Receiving encrypted password under signature does <u>not</u> mean that the sender actually knows the password!

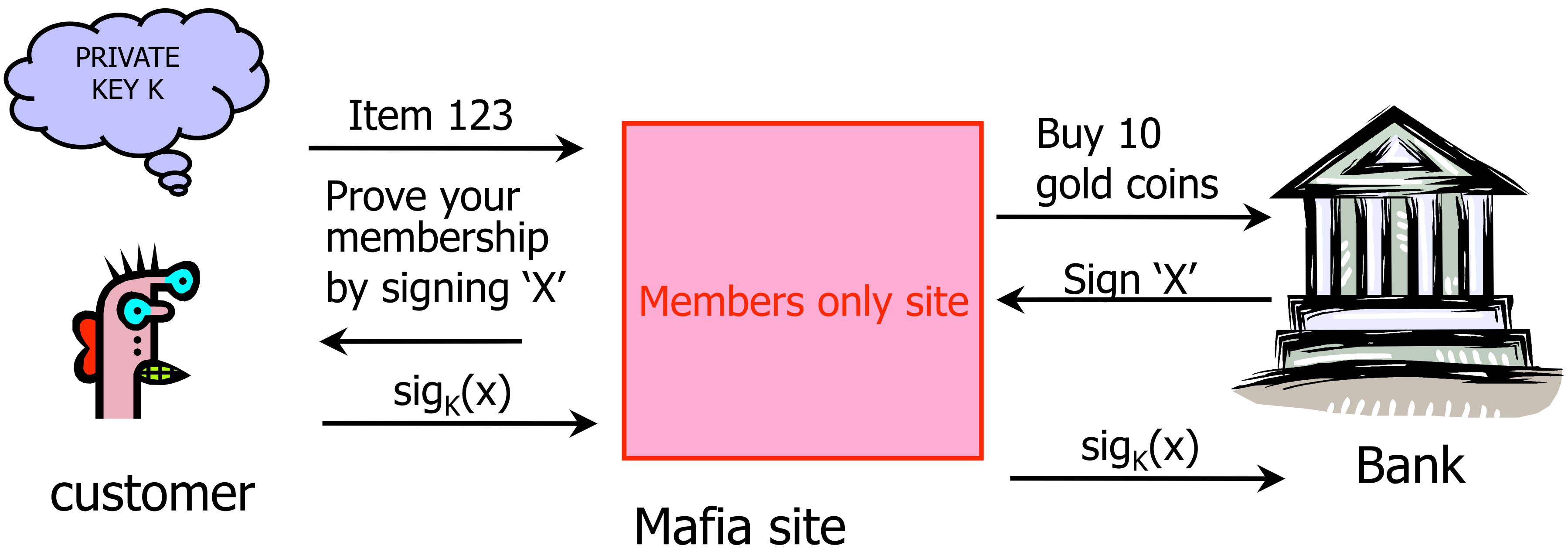
# Authentication with Public Keys




1. Only Alice can create a valid signature
2. Signature is on a fresh, unpredictable challenge

Potential problem: Alice will sign <u>anything</u>

# Mafia-in-the-Middle Attack [from Anderson's book]




One key recommendation: Don't use same public key / secret key pair for multiple applications. (Or make sure messages have different formats across applications.)

# Secure Sessions

- Secure sessions are among the most important applications in network security
  - Enable us to talk securely on an insecure network
- Goal: secure bi-directional communication channel between two parties
  - The channel must provide confidentiality
    - Third party cannot read messages on the channel
  - The channel must provide authentication
    - Each party must be sure who the other party is
  - Other desirable properties: integrity, protection against denial of service, anonymity against eavesdroppers

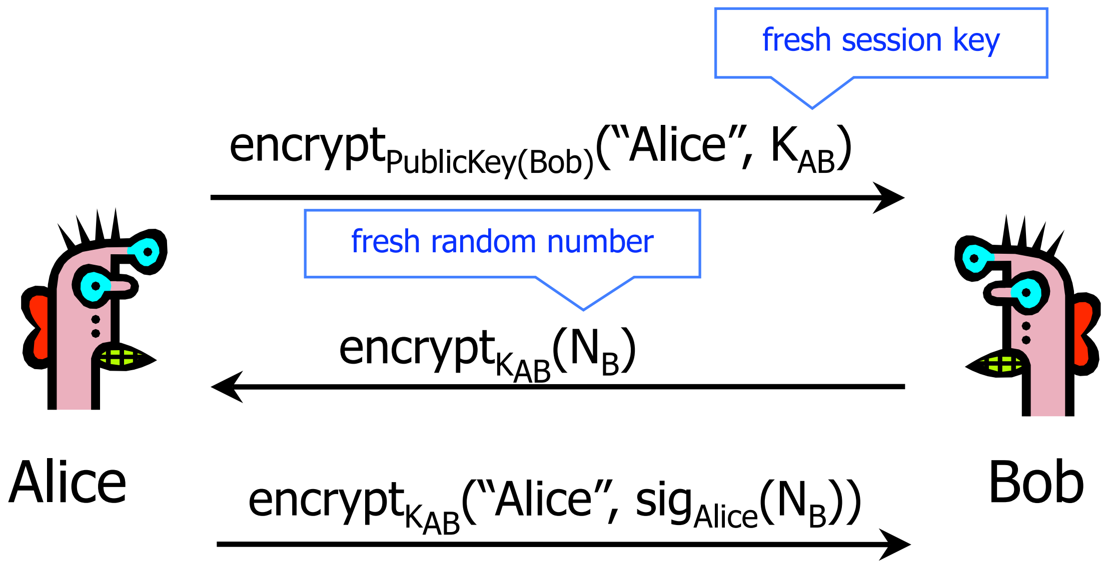
# Key Establishment Protocols

- Common implementation of secure sessions:
  - Establish a secret key known only to two parties
  - Then use block ciphers for confidentiality, HMAC for authentication, and so on
- Challenge: how to establish a secret key
  - Using only public information?
  - Even if the two parties share a long-term secret, a fresh key should be created for each session
    - Long-term secrets are valuable; want to use them as sparingly as possible to limit exposure and the damage if the key is compromised
- (Background: For N parties, there are N choose 2 = N*(N-1)/2 pairs of parties.)

# Key Establishment Techniques

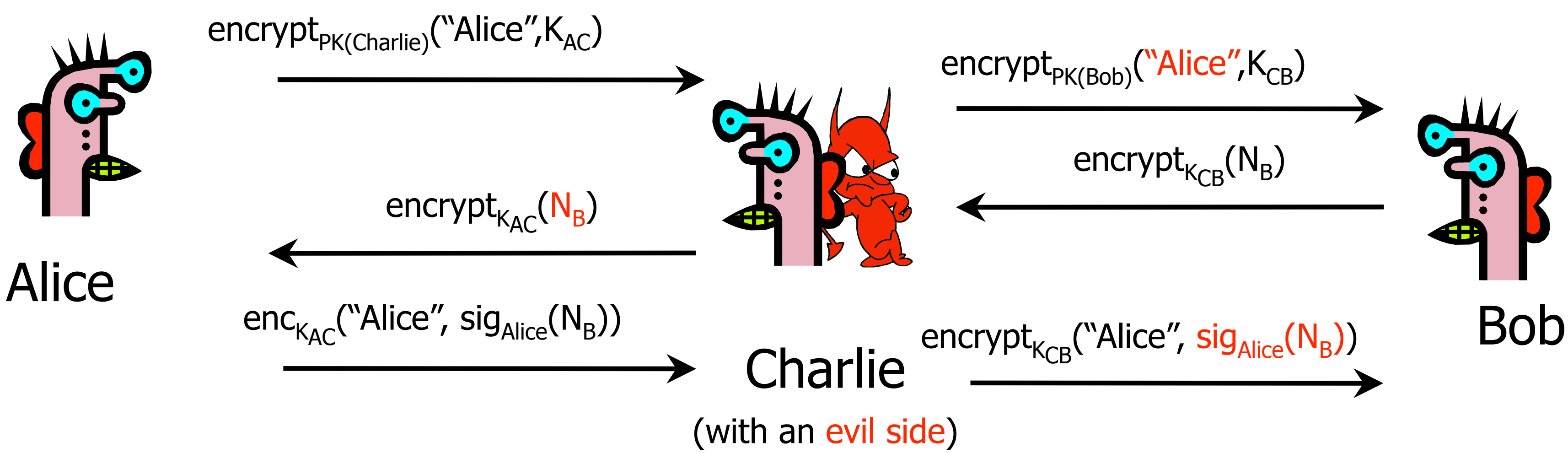
- Use a trusted key distribution center (KDC)
  - Every party shares a pairwise secret key with KDC
  - KDC creates a new random session key and then distributes it, encrypted under the pairwise keys
    - Example: Kerberos
- Use public-key cryptography
  - Diffie-Hellman authenticated with signatures
    - Example: IKE (Internet Key Exchange)
  - One party creates a random key, sends it encrypted under the other party's public key
    - Example: TLS (Transport Layer Security)

# Early Version of SSL (Simplified)



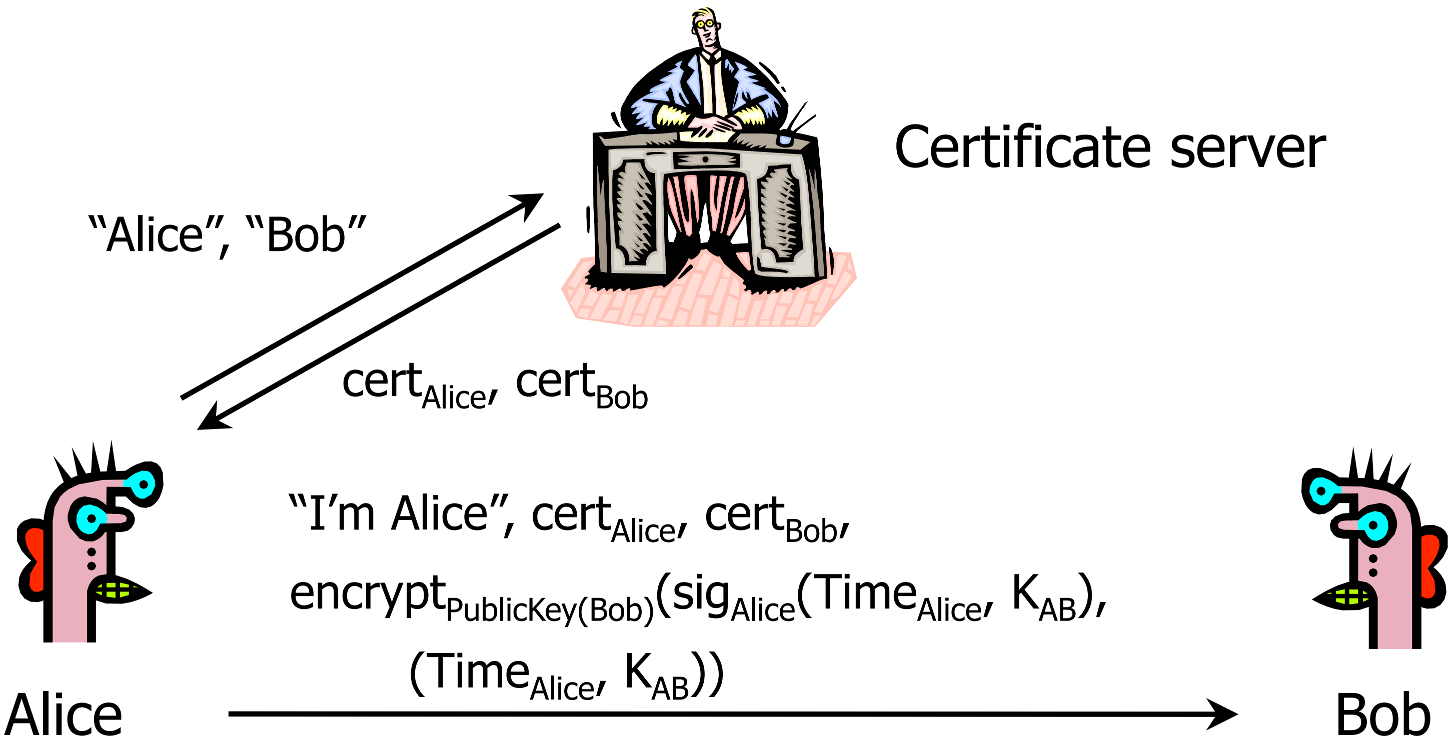

- Bob's reasoning: I must be talking to Alice because…
  - Whoever signed $N_B$ knows Alice's private key… Only Alice knows her private key… Alice must have signed $N_B$… $N_B$ is fresh and random and I sent it encrypted under $K_{AB}$… Alice could have learned $N_B$ only if she knows $K_{AB}$… She must be the person who sent me $K_{AB}$ in the first message...

# Breaking Early SSL



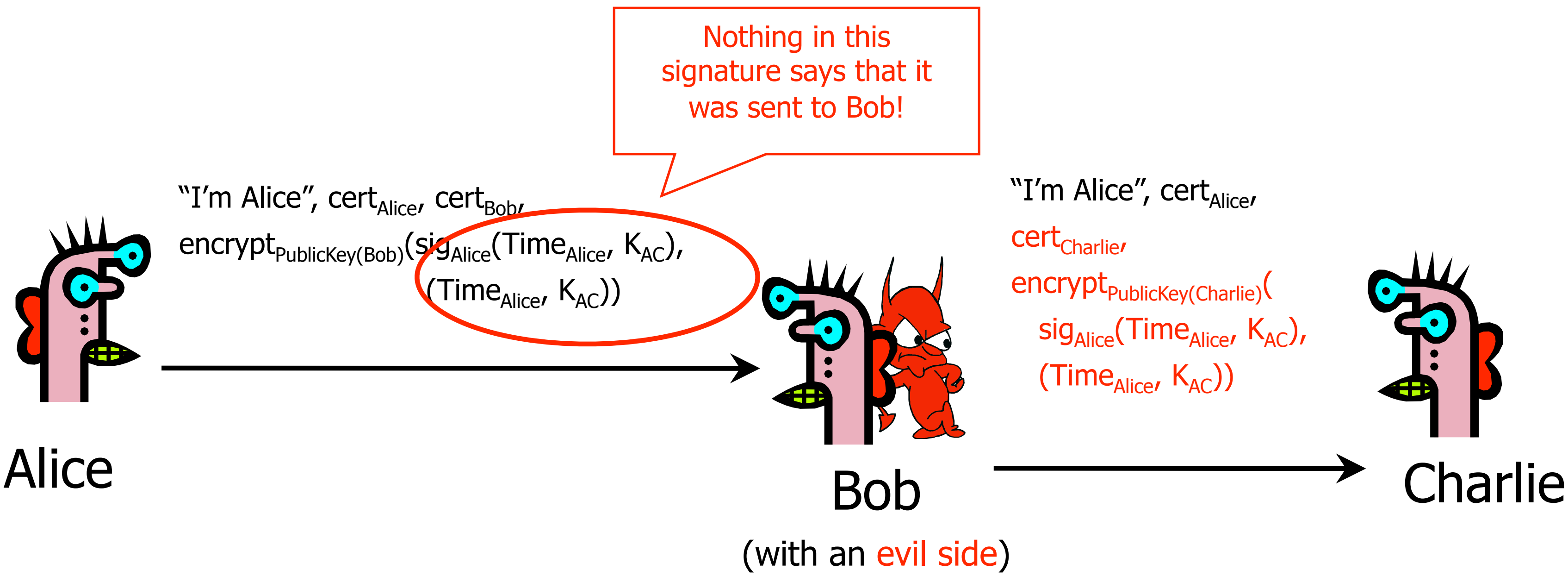

- Charlie uses his legitimate conversation with Alice to impersonate Alice to Bob
  - Information signed by Alice is not sufficiently explicit

# Denning-Sacco Protocol



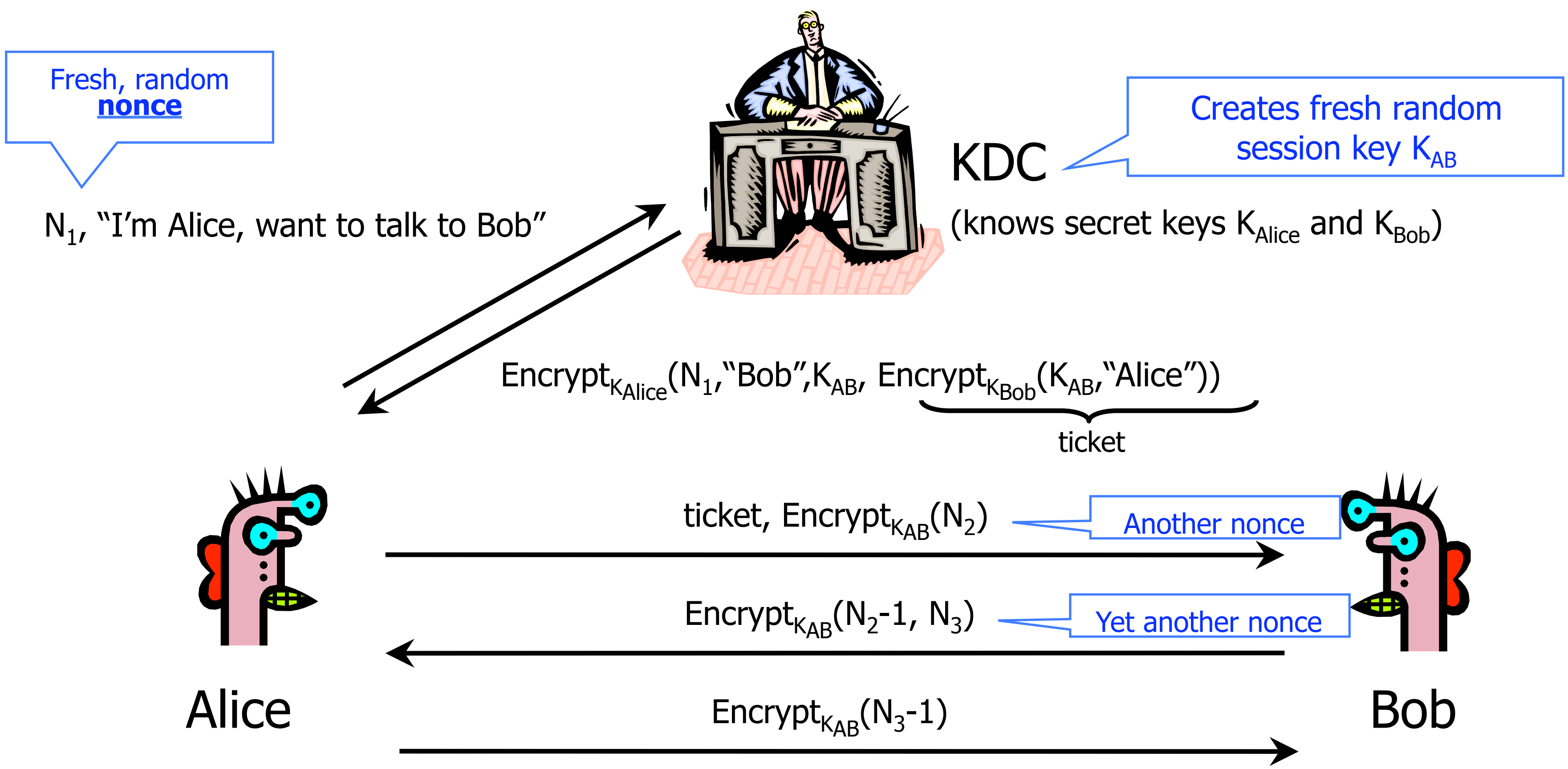

- Goal: establish a new shared key $K_{AB}$ with the help of a trusted certificate service

# Attack on Denning-Sacco



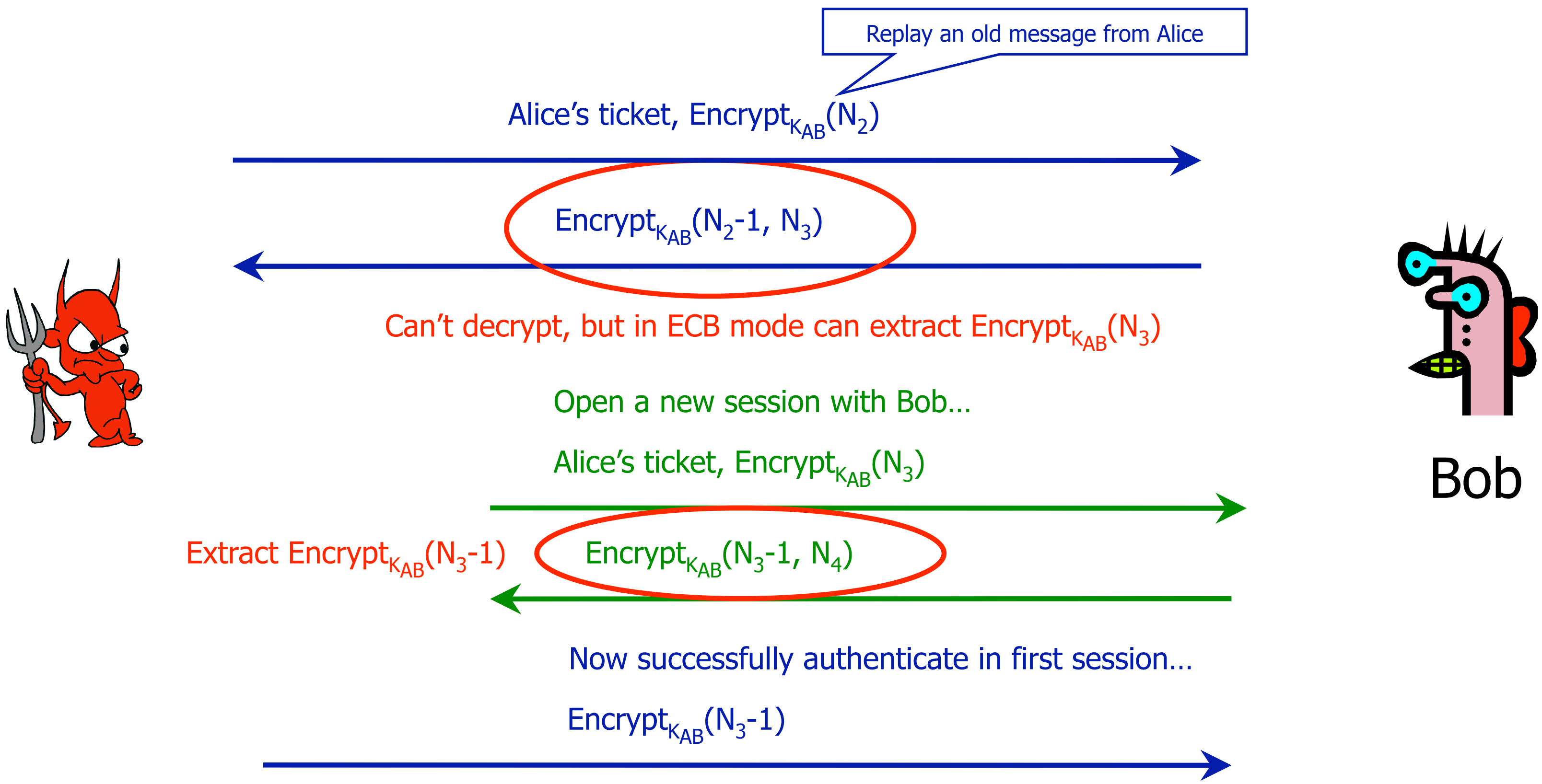

- Alice's signature is insufficiently explicit
  - Does not say to whom and why it was sent
- Alice's signature can be used to impersonate her

# Private-Key Needham-Schroeder



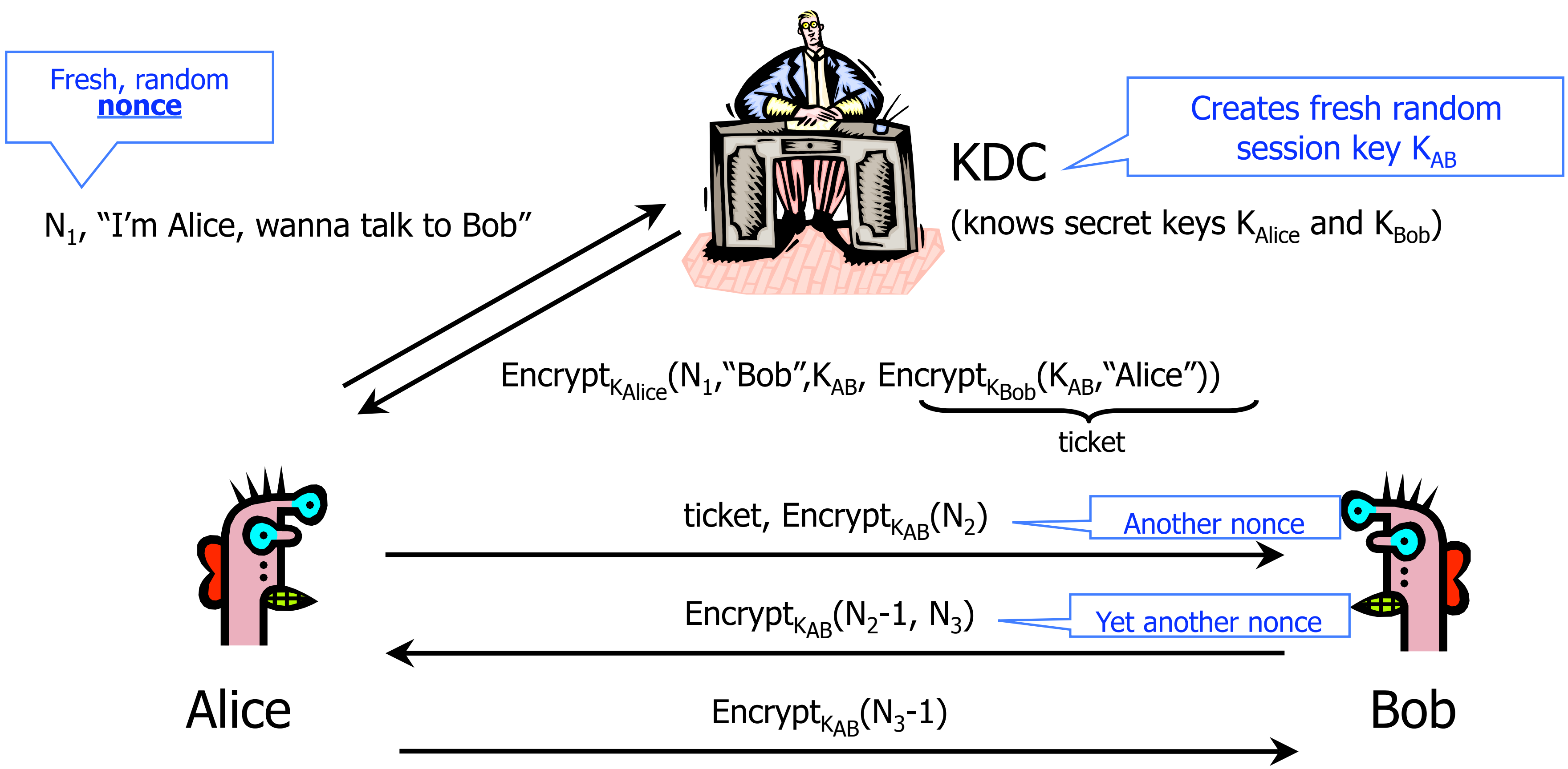
# Reflection Attack

- Suppose symmetric encryption is in ECB/CBC mode…
  - (Easier to see with ECB mode, so assume that)



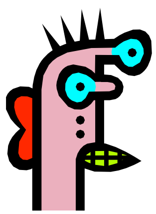

(Recall: Differences between encryption and authenticated encryption.)

# Private-Key Needham-Schroeder




- Another issue: If learn $K_{AB}$ after session completes, then can re-use. (Solution: timestamps, nonces.)

# Public-Key Needham-Schroeder



Alice

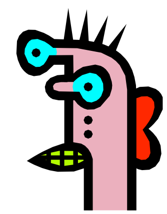
Alice's nonce

$\text{Encrypt}_{\text{PublicKey(Bob)}}(\text{"Alice"}, N_A)$ →

Bob's nonce

← $\text{Encrypt}_{\text{PublicKey(Alice)}}(N_A, N_B)$

$\text{Encrypt}_{\text{PublicKey(Bob)}}(N_B)$ →



Bob

Create new key from $N_A$ and $N_B$, e.g., $N_A \oplus N_B$

Alice's reasoning:

- The only person who could know $N_A$ is the person who decrypted 1st message
- Only Bob can decrypt message encrypted with Bob's public key
- Therefore, Bob is on the other end of the line

**Bob is authenticated!**

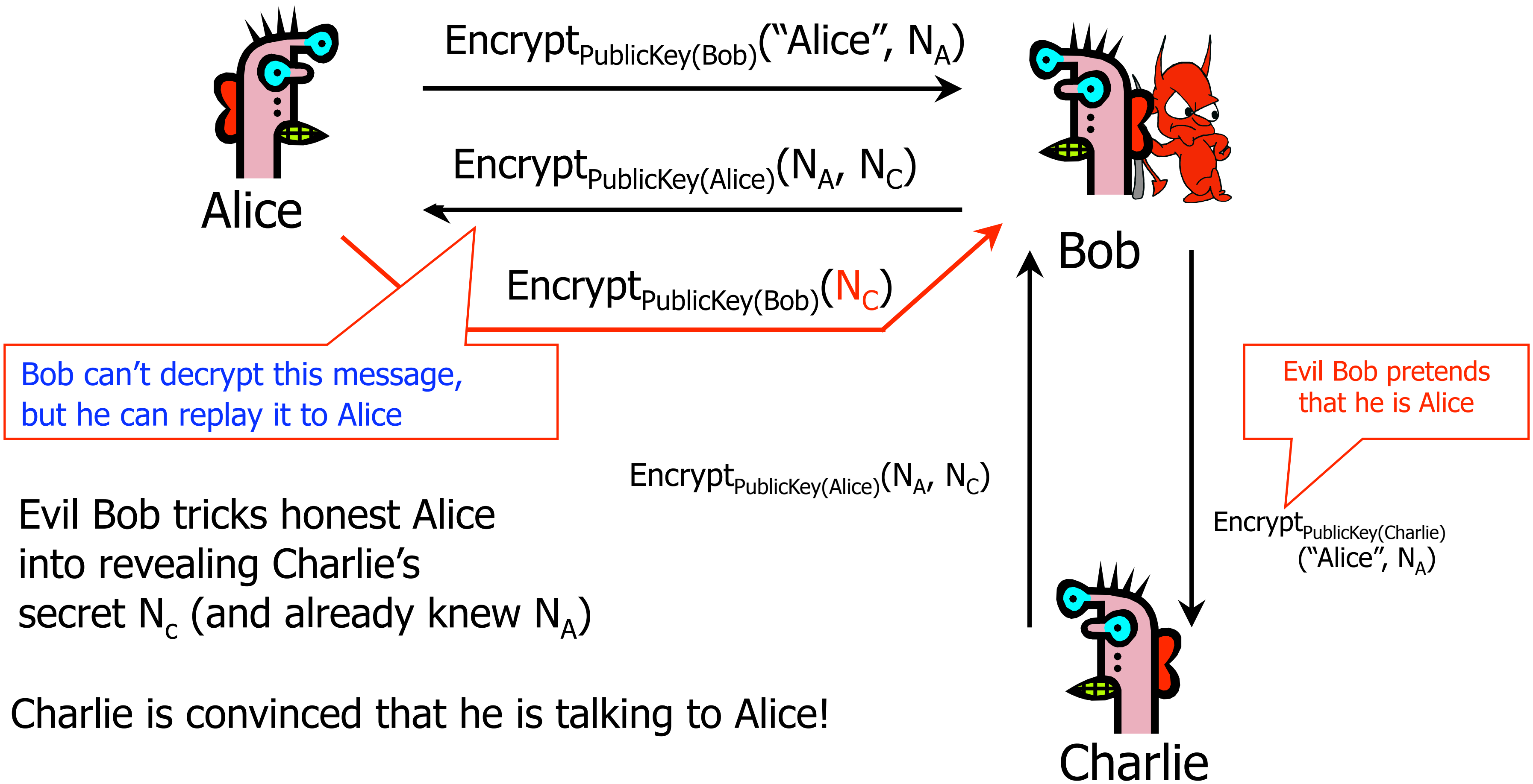
Bob's reasoning:

- The only way to learn $N_B$ is to decrypt 2nd message
- Only Alice can decrypt 2nd message
- Therefore, Alice is on the other end

**Alice is authenticated!**

# Attack on Needham-Schroeder

[published by Gavin Lowe]




Evil Bob tricks honest Alice into revealing Charlie's secret $N_c$ (and already knew $N_A$)

Charlie is convinced that he is talking to Alice!

# Lessons of Needham-Schroeder

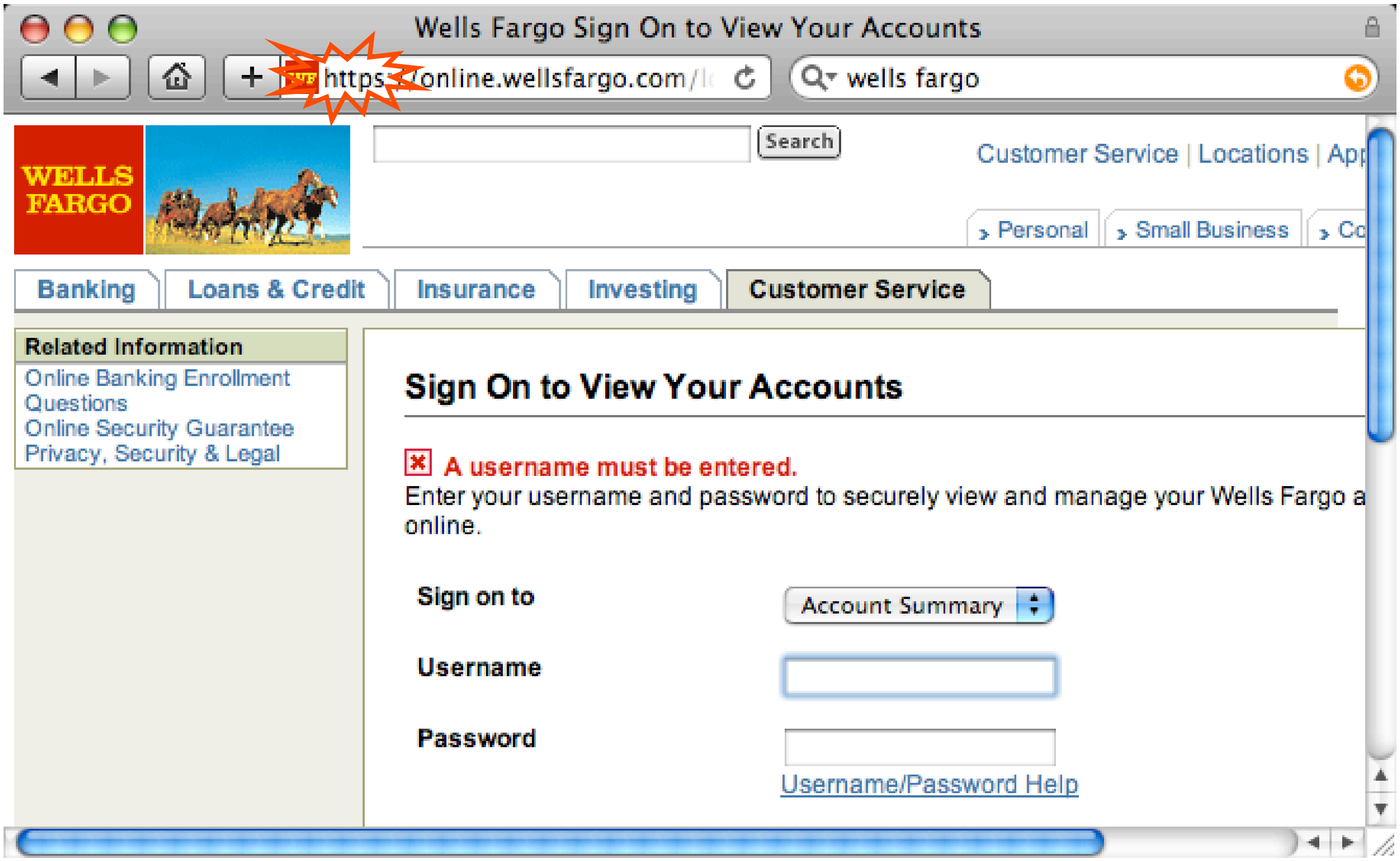
- This is yet another example of design challenges
  - Alice is correct that Bob must have decrypted $Encrypt_{PublicKey(Bob)}$(“Alice”, $N_A$), but this does not mean that $Encrypt_{PublicKey(Alice)}(N_A, N_B)$ came from Bob
- It is important to realize limitations of protocols
  - The attack requires that Alice willingly talk to attacker
    - Attacker uses a legitimate conversation with Alice to impersonate Alice to Charlie

# SSL/TLS

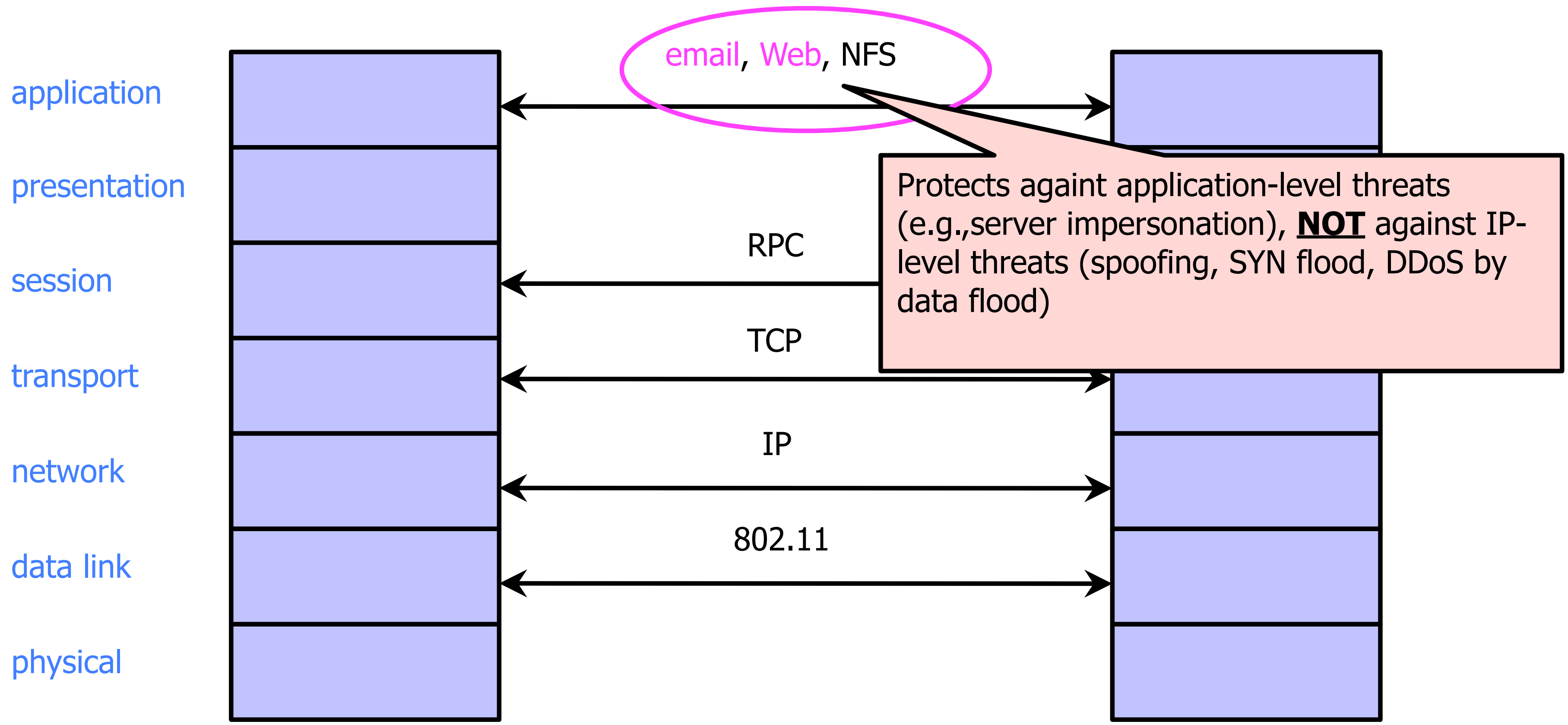
# What is SSL / TLS?

- Transport Layer Security (TLS) protocol, version 1.2
  - De facto standard for Internet security
  - "The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications"
  - In practice, used to protect information transmitted between browsers and Web servers (and mail readers and ...)
- Based on Secure Sockets Layers (SSL) protocol, version 3.0
  - Same protocol design, different algorithms
- Deployed in all(?) Web browsers

# SSL / TLS in the Real World



# Application-Level Protection



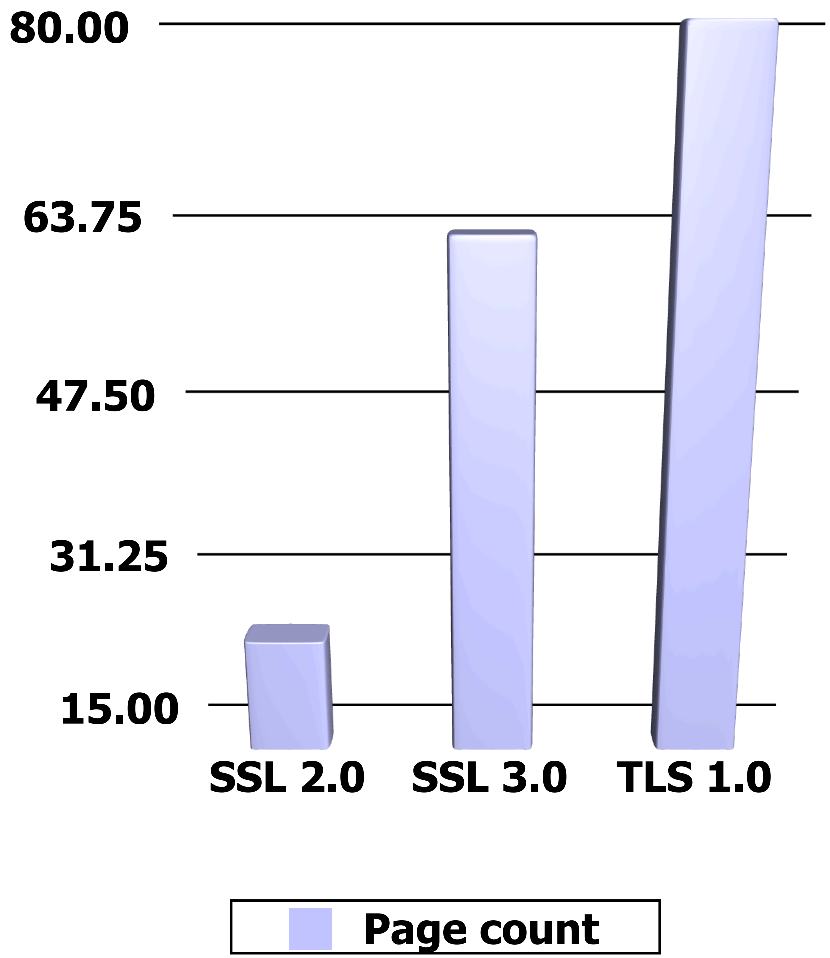
# History of the Protocol

- SSL 1.0
  - Internal Netscape design, early 1994?
- SSL 2.0
  - Published by Netscape, November 1994
  - Several weaknesses
- SSL 3.0
  - Designed by Netscape and Paul Kocher, November 1996
- TLS 1.0
  - Internet standard based on SSL 3.0, January 1999
  - Not interoperable with SSL 3.0
    - TLS uses HMAC instead of earlier MAC; can run on any port
- TLS 1.2
  - Remove dependencies to MD5 and SHA1

# "Request for Comments"

- Network protocols are usually disseminated in the form of an RFC
- TLS version 1.2 is described in RFC 5246
- Intended to be a self-contained definition of the protocol
  - Describes the protocol in sufficient detail for readers who will be implementing it and those who will be doing protocol analysis
  - Mixture of informal prose and pseudo-code

# Evolution of the SSL/TLS RFC




**104 pages for TLS 1.2**

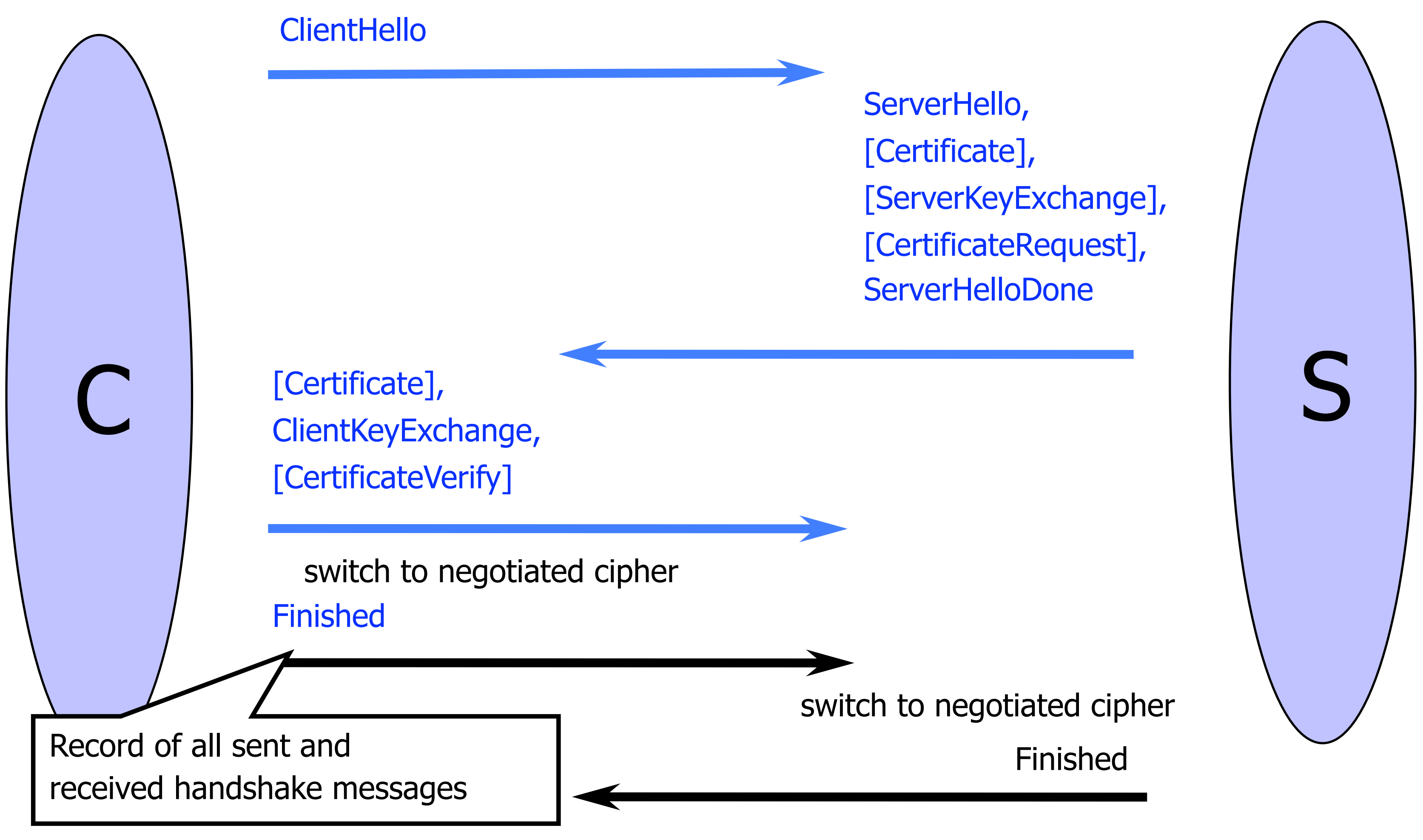
# TLS Basics

- TLS consists of two protocols
  - Familiar pattern for key exchange protocols
- Handshake protocol
  - Use public-key cryptography to establish a shared secret key between the client and the server
- Record protocol
  - Use the secret key established in the handshake protocol to protect communication between the client and the server
- We will focus on the handshake protocol

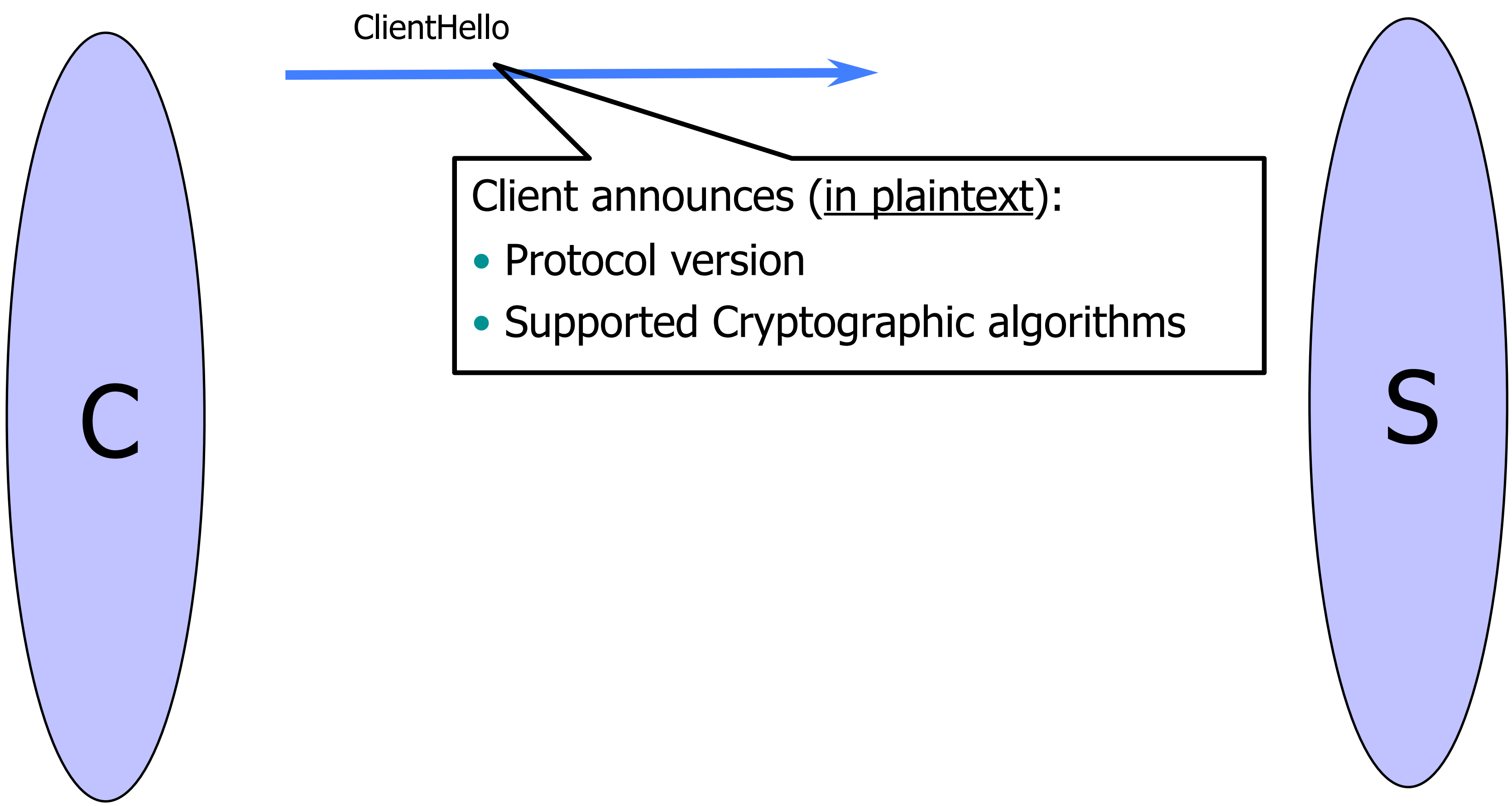
# TLS Handshake Protocol

- Two parties: client and server
- Negotiate version of the protocol and the set of cryptographic algorithms to be used
  - Interoperability between different implementations of the protocol
- Authenticate client and server (optional)
  - Use digital certificates to learn each other's public keys and verify each other's identity
- Use public keys to establish a shared secret

# Handshake Protocol Structure



# ClientHello



# ClientHello (RFC)

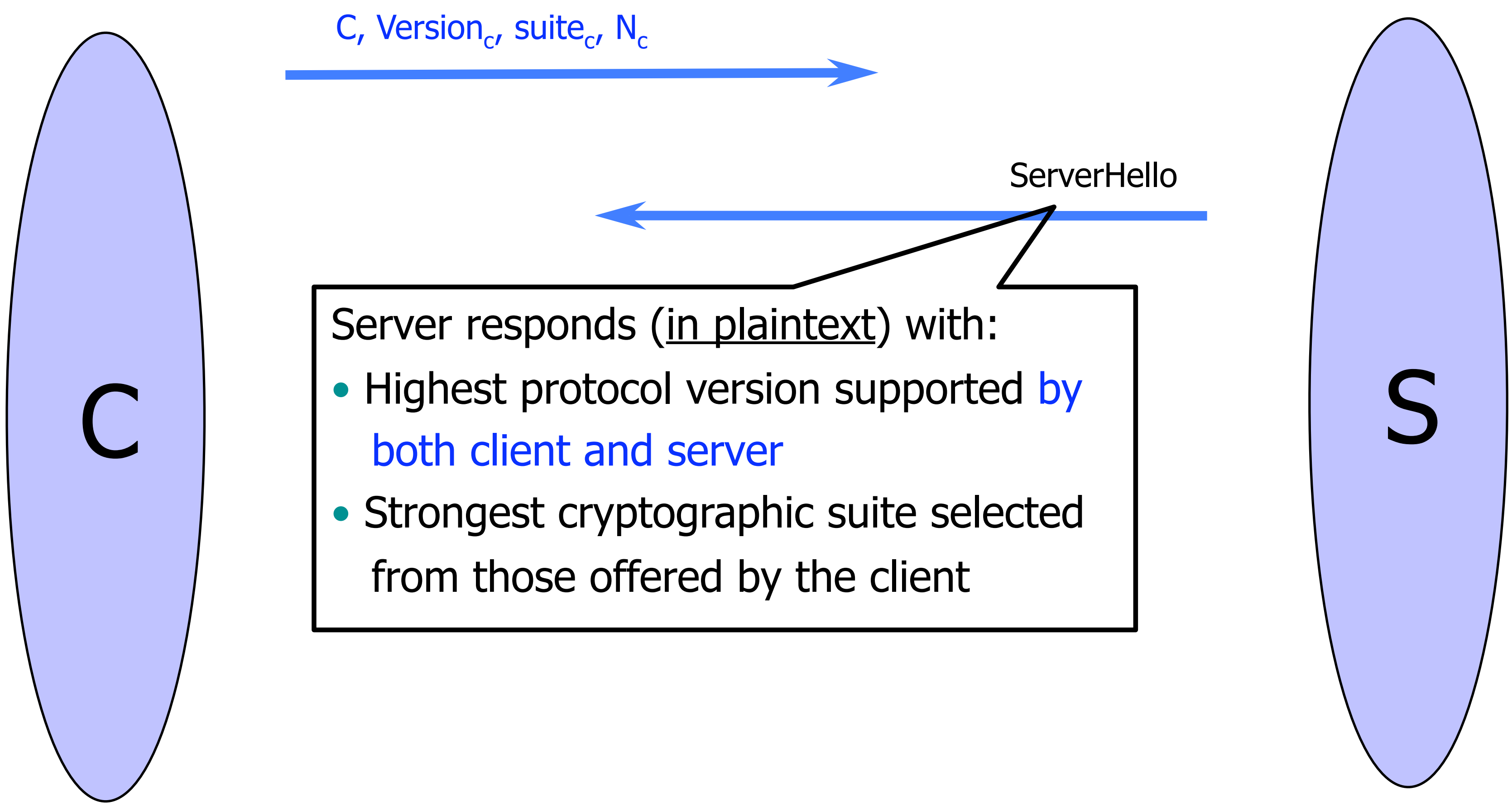
```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites;
    CompressionMethod compression_methods;
} ClientHello
```

Highest version of the protocol supported by the client

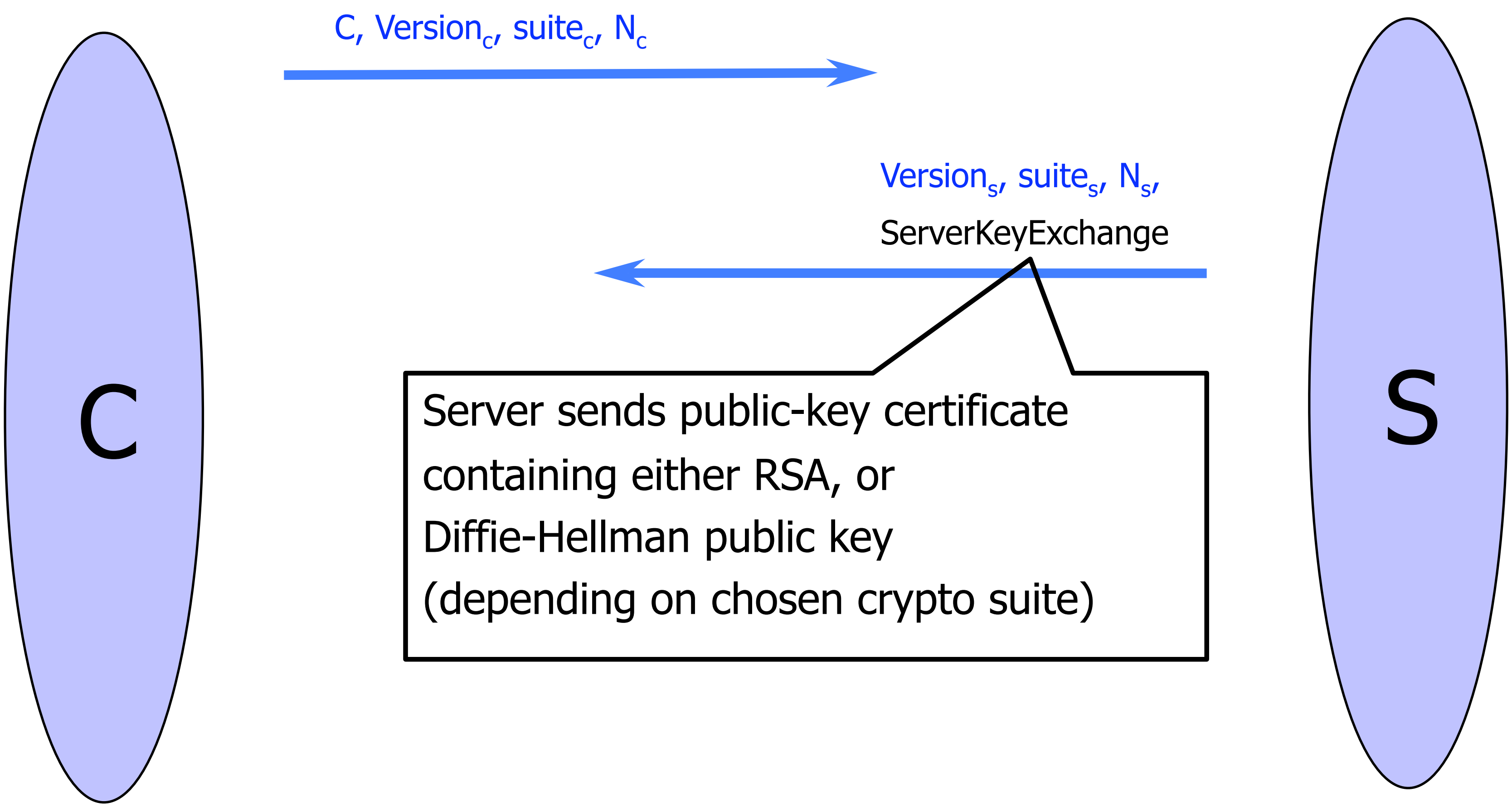
Session id (if the client wants to resume an old session)

Set of cryptographic algorithms supported by the client (e.g., RSA or Diffie-Hellman)

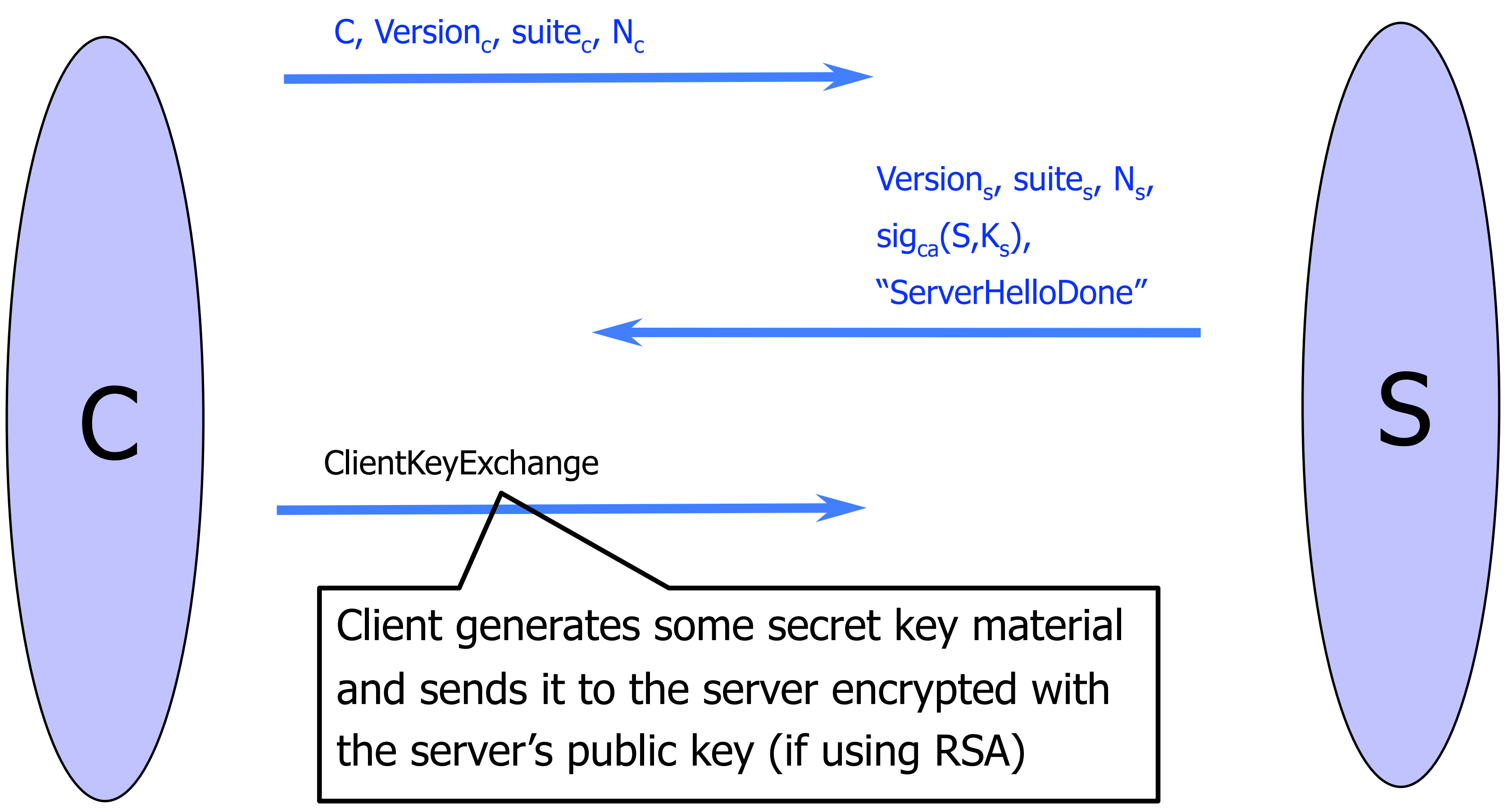
# ServerHello



# ServerKeyExchange



# ClientKeyExchange

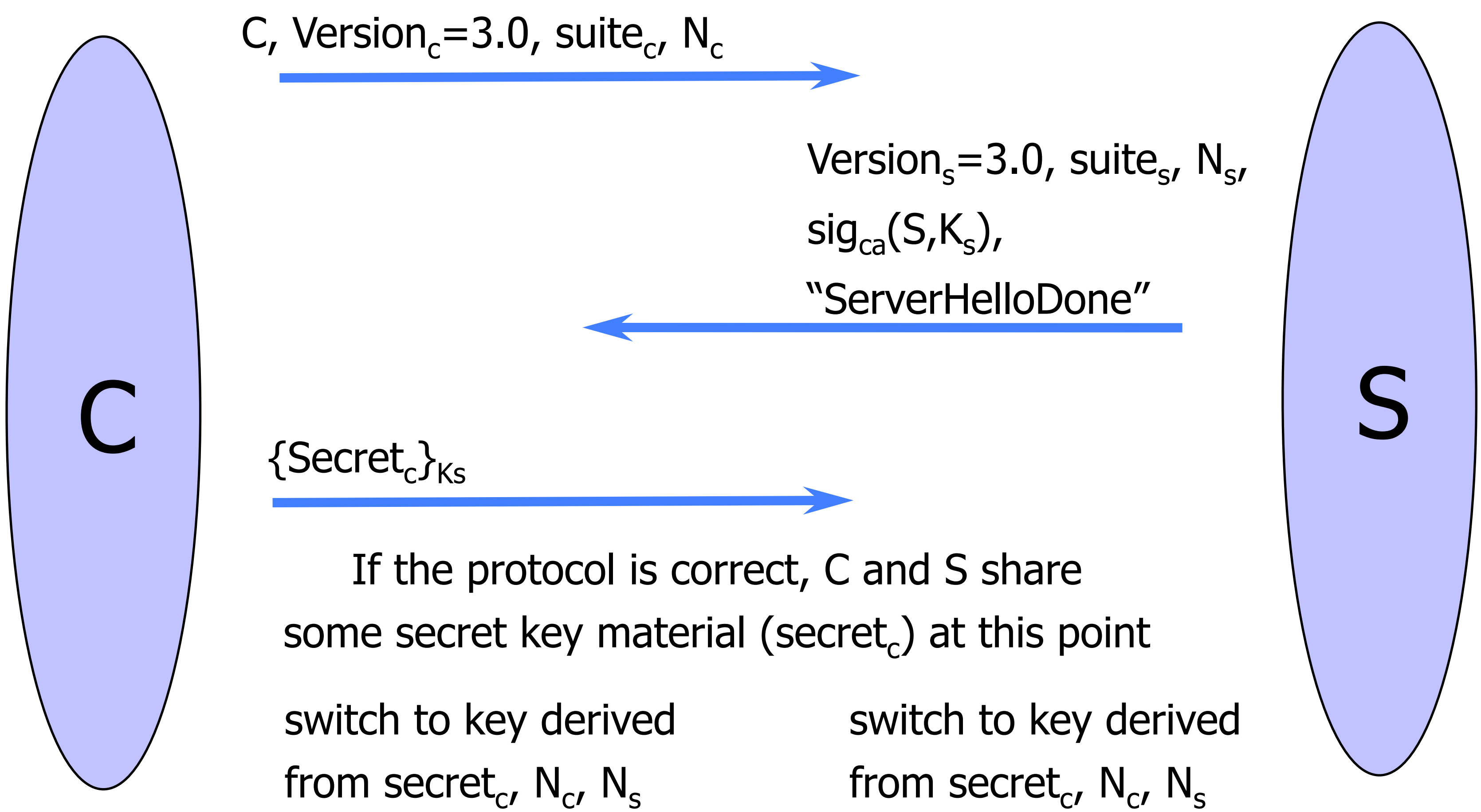


# ClientKeyExchange (RFC)

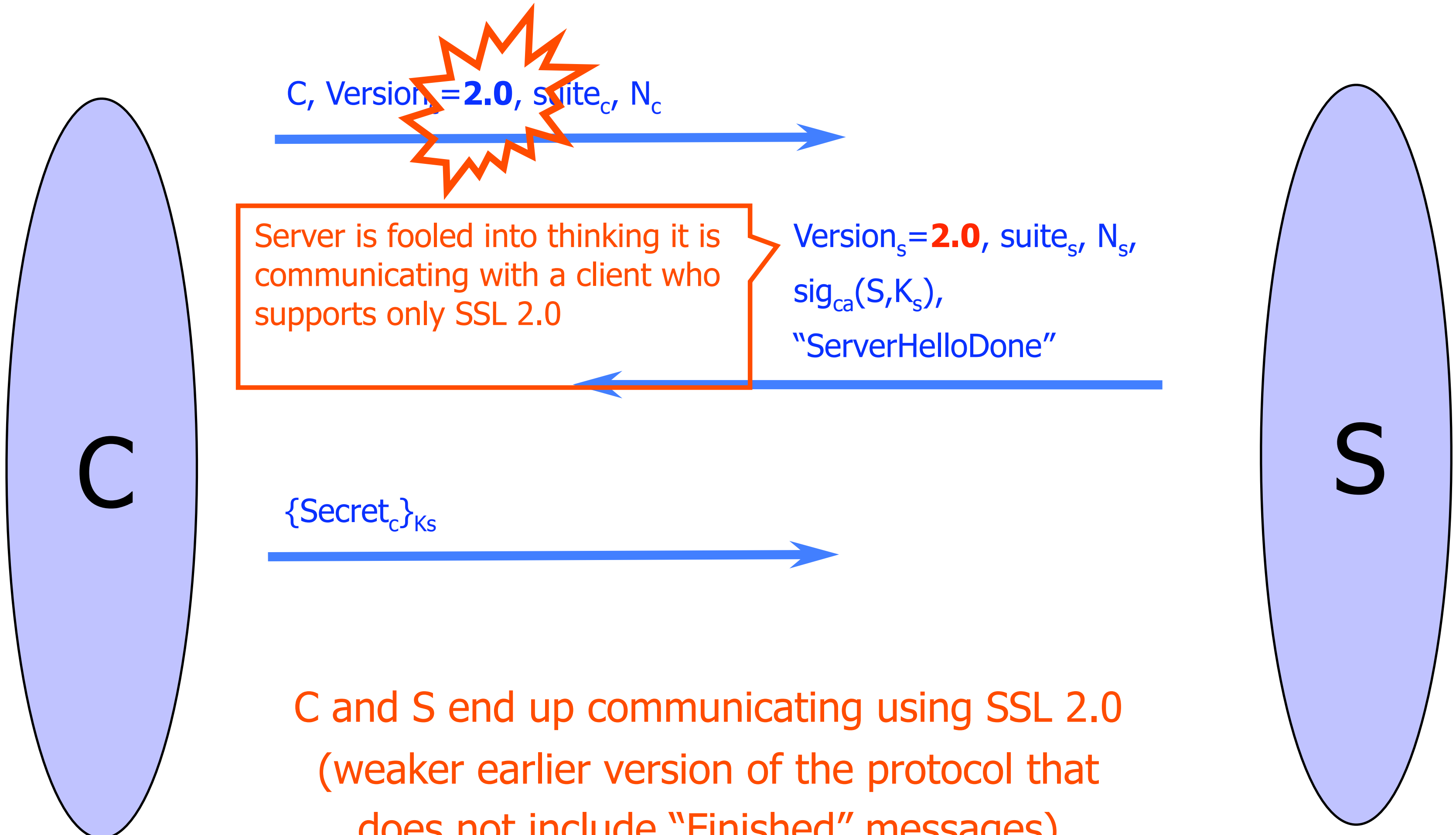
```
struct {
   select (KeyExchangeAlgorithm) {
      case rsa: EncryptedPreMasterSecret;
      case diffie_hellman: ClientDiffieHellmanPublic;
   } exchange_keys
} ClientKeyExchange
struct {

   ProtocolVersion client_version;
   opaque random[46];
} PreMasterSecret
```

Random bits from which symmetric keys will be derived (by hashing them with nonces)

# "Core" SSL 3.0 Handshake (Not TLS)



# Version Rollback Attack




C and S end up communicating using SSL 2.0 (weaker earlier version of the protocol that does not include "Finished" messages)

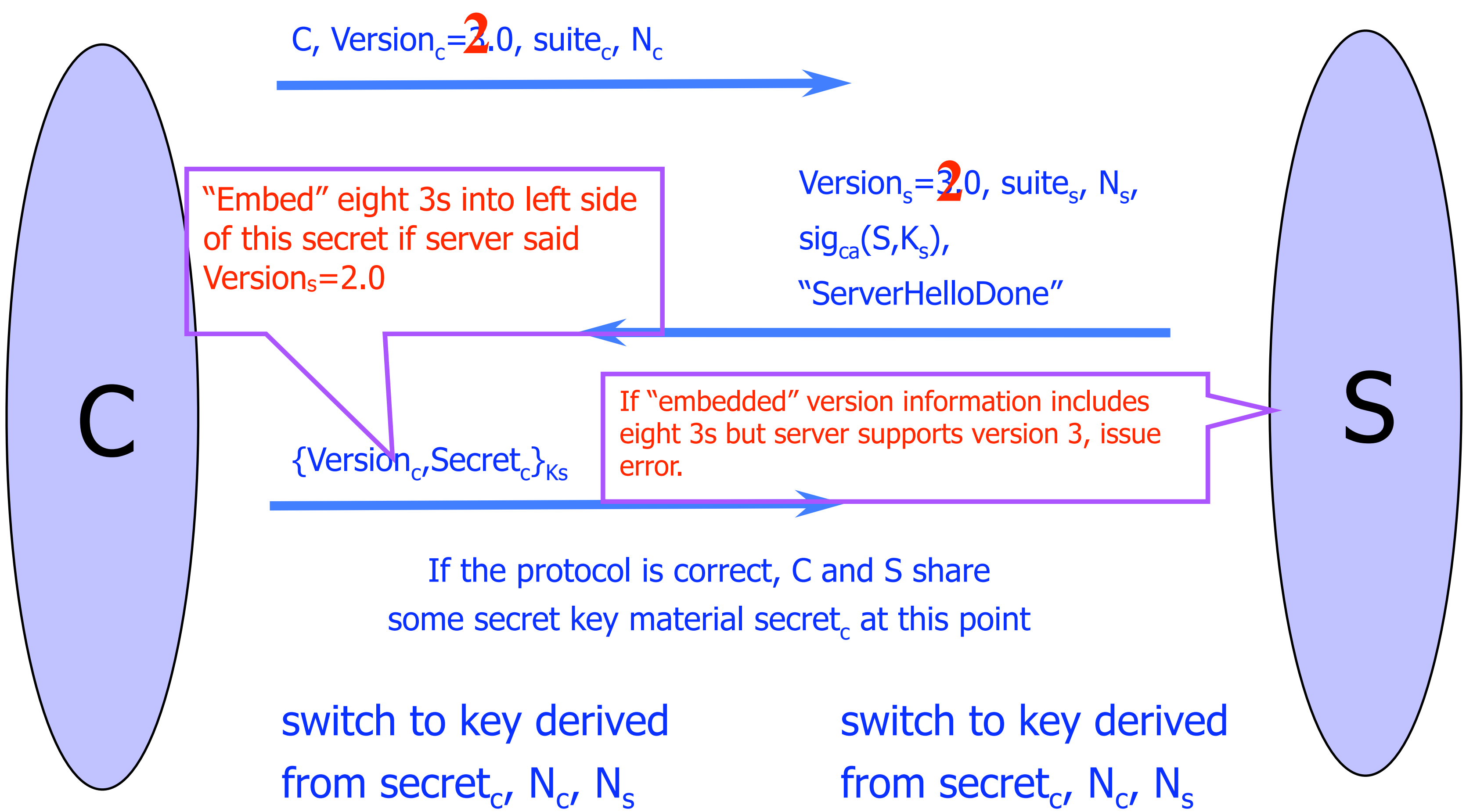
# SSL 2.0 Weaknesses (Fixed in 3.0)

- Cipher suite preferences are not authenticated
  - "Cipher suite rollback" attack is possible
- SSL 2.0 uses padding when computing MAC in block cipher modes, but padding length field is not authenticated
  - Attacker can delete bytes from the end of messages
- MAC uses only 40 bits in export mode
- No support for certificate chains or non-RSA algorithms, no handshake while session is open

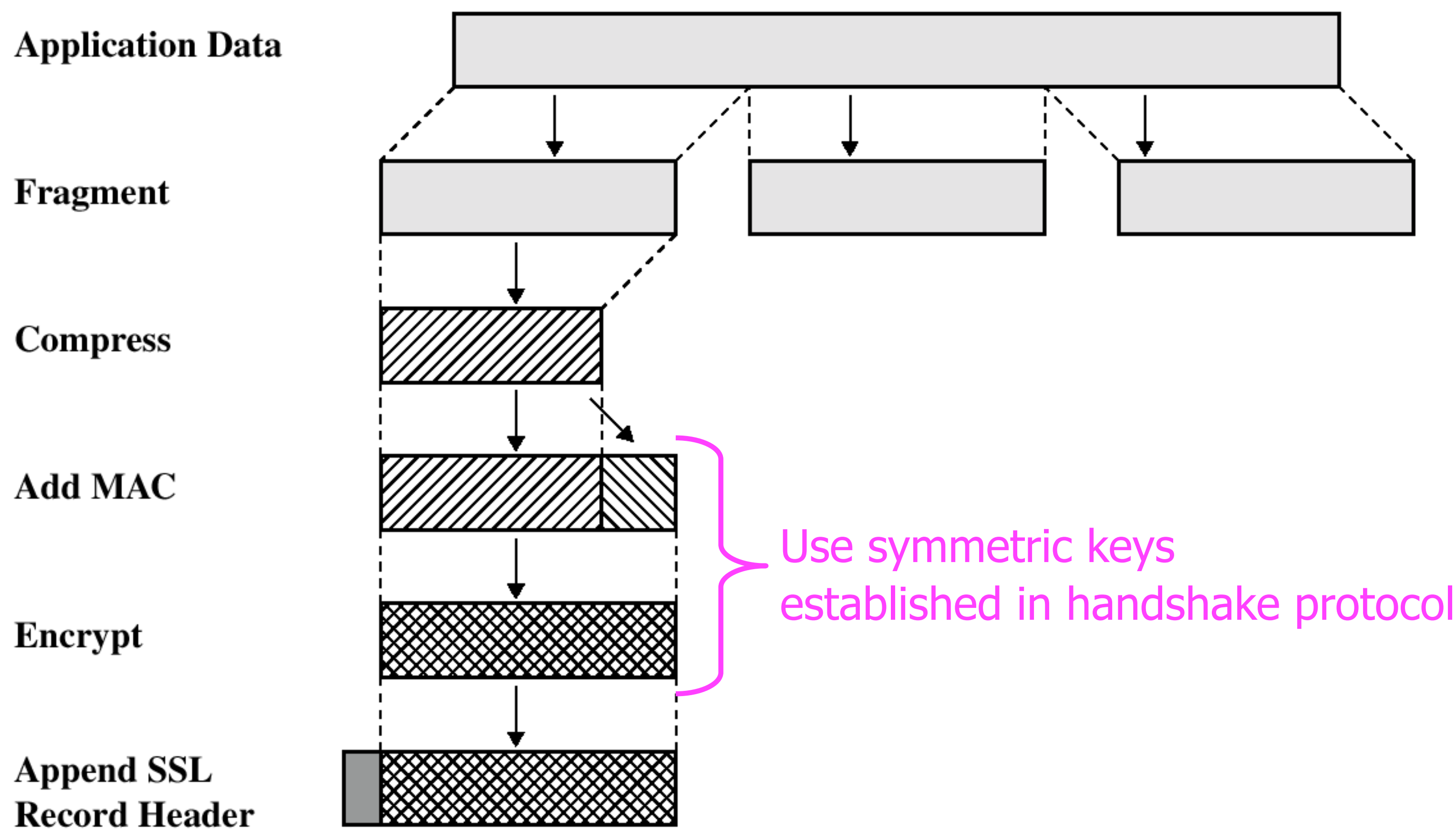
# Protocol Rollback Attacks

- Why do people release new versions of security protocols? Because the old version got broken!
- New version must be backward-compatible
  - Not everybody upgrades right away
- Attacker can fool someone into using the old, broken version and exploit known vulnerability
  - Similar: fool victim into using weak crypto algorithms
- Defense is hard: must authenticate version in early designs
- Many protocols had "version rollback" attacks
  - SSL, SSH, GSM (cell phones)

# Version Check in SSL 3.0 (Approximate)



# SSL/TLS Record Protection



# Password Managers

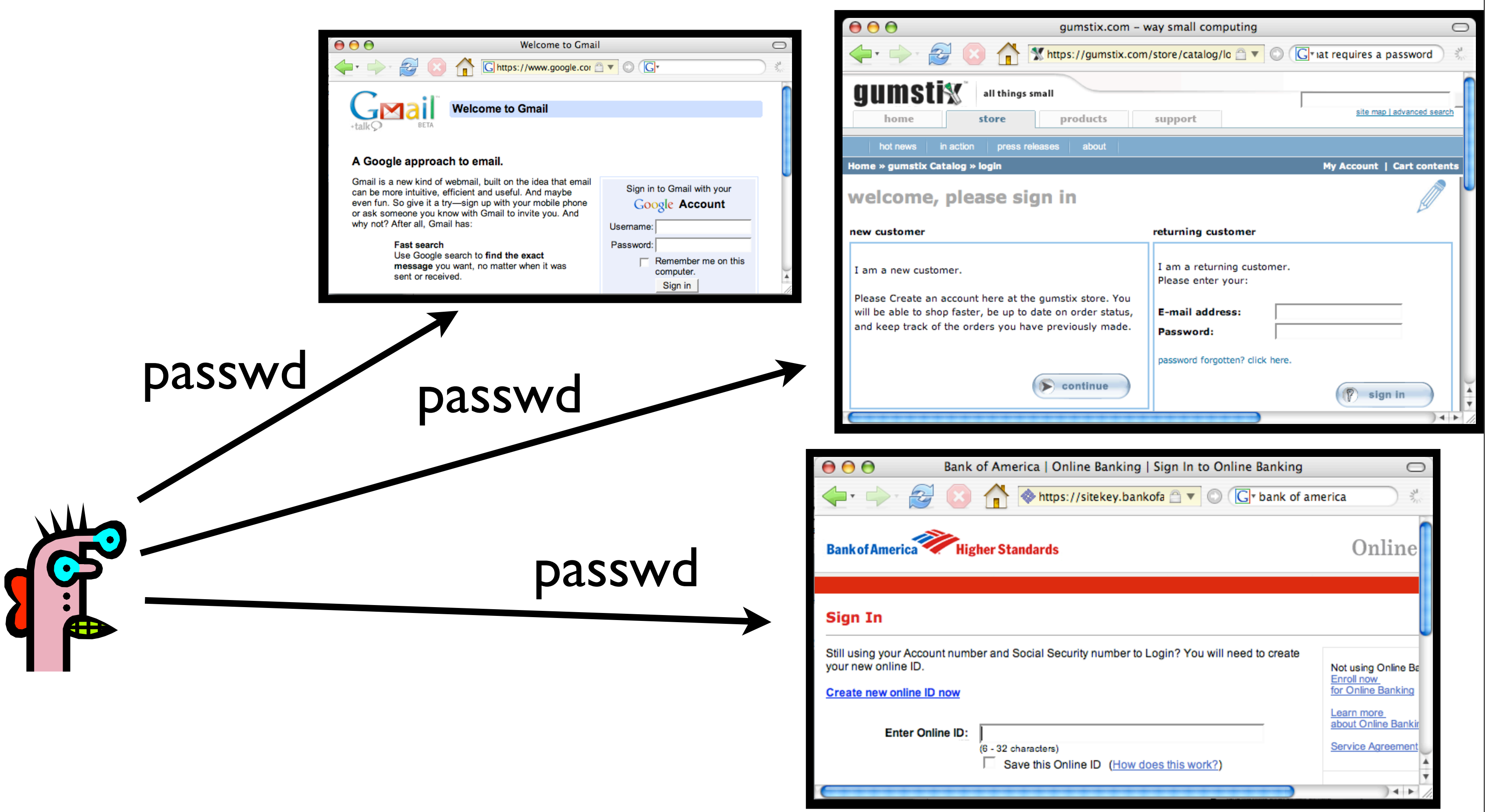
- Idea: Software application that will store and manage passwords for you.
  - You remember one password.
  - Each website sees a different password.
- Examples: PwdHash (Usenix Security 2005) and Password Multiplier (WWW 2005).

# Key ideas

- User remembers a single password
- Password managers
  - On input: (1) the user's single password and (2) information about the website
  - Compute: Strong, site-specific password
- Goal: Avoid problems with passwords

# The problem

Alice needs passwords for all the websites that she visits



# Possible solutions

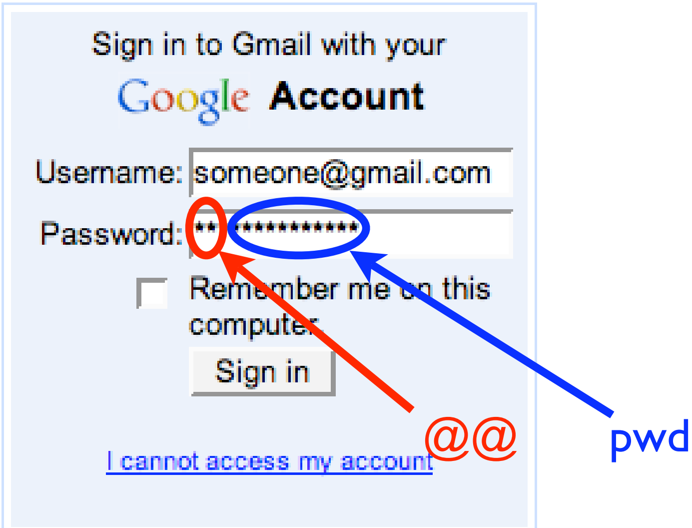
- Easy to remember: Use same password on all websites. Use "weak" password.
  - Poor security (don't share password between bank website and small website)
- More secure: Use different, strong passwords on all websites.
  - Hard to remember, unless write down.

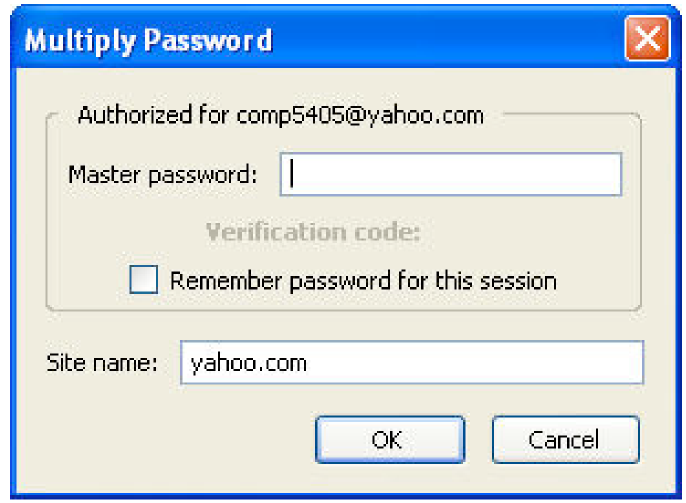
# Alternate solution: Password managers

- Password managers handle creating and "remembering" strong passwords
- Potentially:
  - Easier for users
  - More secure
- Examples:
  - PwdHash (Usenix Security 2005)
  - Password Multiplier (WWW 2005)

# PwdHash

# Password Multiplier




@@ in front of passwords to protect; or F2

sitePwd = Hash(pwd,domain)

Prevent phishing attacks

Active with Alt-P or double-click

sitePwd = Hash(usrname, pwd, domain)

Both solutions target simplicity and transparency.

# Usenix 2006: Usability testing

HCI is important!

- Are these programs usable? If not, what are the problems?
- Two main approaches for evaluating usability:
  - Usability inspection (no users)
    - Cognitive walk throughs
    - Heuristic evaluation
  - User study
    - Controlled experiments
    - Real usage

This paper stresses need to observe real users

[Chiasson, van Oorschot, Biddle]

# Study details

- 26 participants, across various backgrounds (4 technical)
- Five assigned tasks per plugin
- Data collection
  - Observational data (recording task outcomes, difficulties, misconceptions)
  - Questionnaire data (initial attitudes, opinions after tasks, post questionnaires)

[Chiasson, van Oorschot, Biddle]

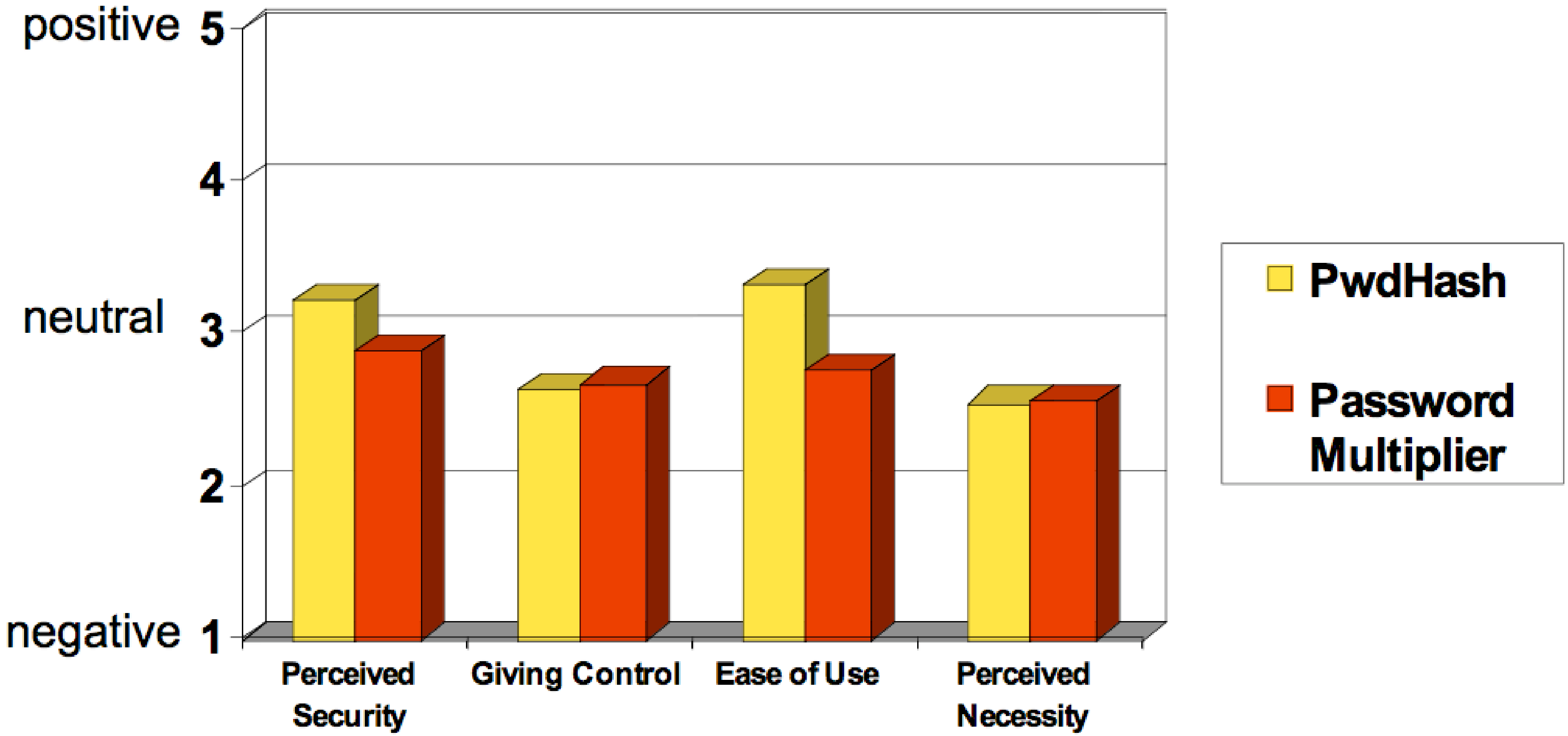
# Task completion results

| | Success | Potentially Causing Security Exposures | | | |
|---|---|---|---|---|---|
| | | Dangerous Success | Failures | | |
| | | | Failure | False Completion | Failed due to Previous |
| **PwdHash** | | | | | |
| Log In | 48% | 44% | 8% | 0% | N/A |
| Migrate Pwd | 42% | 35% | 11% | 11% | N/A |
| Remote Login | 27% | 42% | 31% | 0% | N/A |
| Update Pwd | 19% | 65% | 8% | 8% | N/A |
| Second Login | 52% | 28% | 4% | 0% | 16% |
| **Password Multiplier** | | | | | |
| Log In | 48% | 44% | 8% | 0% | N/A |
| Migrate Pwd | 16% | 32% | 28% | 20% | N/A |
| Remote Login | N/A | N/A | N/A | N/A | N/A |
| Update Pwd | 16% | 4% | 44% | 28% | N/A |
| Second Login | 16% | 4% | 16% | 0% | 16% |

http://www.scs.carleton.ca/~schiasso/Chiasson_UsenixSecurity2006_PwdManagers.ppt

[Chiasson, van Oorschot, Biddle]

# Questionnaire responses




http://www.scs.carleton.ca/~schiasso/Chiasson_UsenixSecurity2006_PwdManagers.ppt

# Problem: Transparency

- Unclear to users whether actions successful or not.
  - Should be obvious when plugin activated.
  - Should be obvious when password protected.
- Users feel that they should be able to know their own password.

# Problem:  Mental model

Users seemed to have misaligned mental models

- Not understand that one needs to put "@@" before *each* password to be protected.
- Think different passwords generated for each session.
- Think successful when were not.
- Not know to click in field before Alt-P.
- PwdHash: Think passwords unique to them.

HCI is important!

# When “nothing works”

- Tendency to try all passwords
  - A poor security choice.
  - May make the use of PwdHash or Password Multiplier *worse* than not using any password manager.
- Usability problem leads to security vulnerabilities.

# Facebook founder Mark Zuckerberg 'hacked into emails of rivals and journalists'

By MAIL FOREIGN SERVICE
Last updated at 2:09 AM on 06th March 2010

Co

Facebo
been a
accoun

The CE
social r
at least
of artic

As part
detailin
magaz
eviden

Business Insider claimed he then told a friend how he had hacked into the accounts of Crimson staff.

He allegedly told the friend that he used TheFacebook.com to search for members who said they were Crimson staff.

Then, he allegedly examined a report of failed logins to see if any of the Crimson members had ever entered an incorrect password into TheFacebook.com.

In the instances where they had, Business Insider claimed that Zuckerberg said he tried using those incorrect passwords to access the Crimson members' Harvard email accounts.

In two instances, the magazine claimed, he succeeded - and was able to read emails between Crimson staff discussing the possibility of writing an article on the accusations surrounding him.

'In other words,' Business Insider claimed, 'Mark appears to have used private login data from TheFacebook to hack into the separate email accounts of some TheFacebook users'.