Protocols II

Computer Security Lecture 13

Mike Just[1]

School of Informatics
University of Edinburgh

1 March 2010

[1] Based on original lecture notes by David Aspinall

Outline

Outline

Introduction

- Previously examined some simple protocols

Introduction

- Previously examined some simple protocols
  - Simple authentication using passwords, and shared keys

Introduction

- Previously examined some simple protocols
  - Simple authentication using passwords, and shared keys
  - Challenge response with shared (symmetric) keys

Introduction

- Previously examined some simple protocols
  - Simple authentication using passwords, and shared keys
  - Challenge response with shared (symmetric) keys
  - Use of nonces

Introduction

- Previously examined some simple protocols
  - Simple authentication using passwords, and shared keys
  - Challenge response with shared (symmetric) keys
  - Use of nonces
- This lecture expands and extends these concepts

Introduction

- Previously examined some simple protocols
  - Simple authentication using passwords, and shared keys
  - Challenge response with shared (symmetric) keys
  - Use of nonces
- This lecture expands and extends these concepts
  - Mutual authentication

Introduction

- Previously examined some simple protocols
  - Simple authentication using passwords, and shared keys
  - Challenge response with shared (symmetric) keys
  - Use of nonces
- This lecture expands and extends these concepts
  - Mutual authentication
  - Challenge response with public keys

Introduction

- Previously examined some simple protocols
  - Simple authentication using passwords, and shared keys
  - Challenge response with shared (symmetric) keys
  - Use of nonces
- This lecture expands and extends these concepts
  - Mutual authentication
  - Challenge response with public keys
  - Authentication *and* key establishment

Introduction

- Previously examined some simple protocols
  - Simple authentication using passwords, and shared keys
  - Challenge response with shared (symmetric) keys
  - Use of nonces
- This lecture expands and extends these concepts
  - Mutual authentication
  - Challenge response with public keys
  - Authentication *and* key establishment
  - Digital certificates

Introduction

- Previously examined some simple protocols
  - Simple authentication using passwords, and shared keys
  - Challenge response with shared (symmetric) keys
  - Use of nonces
- This lecture expands and extends these concepts
  - Mutual authentication
  - Challenge response with public keys
  - Authentication *and* key establishment
  - Digital certificates
  - More fun with nonces

Outline

Reminder: shared-key unilateral authentication

- Minimal protocol using a **random number**:

Message 1. $S \to A$: N_s
Message 2. $A \to S$: $\{N_s, S\}_{K_{as}}$

Reminder: shared-key unilateral authentication

- Minimal protocol using a **random number**:

  Message 1. $S \rightarrow A$: N_s
  Message 2. $A \rightarrow S$: $\{N_s, S\}_{K_{as}}$

- Minimal protocol using **timestamps**; the "challenge" is implicit:

  Message 1. $A \rightarrow S$: $\{T_a, S\}_{K_{as}}$

Reminder: shared-key unilateral authentication

- Minimal protocol using a **random number**:

$$\begin{array}{lll} \text{Message 1.} & S \rightarrow A: & N_s \\ \text{Message 2.} & A \rightarrow S: & \{N_s, S\}_{K_{as}} \end{array}$$

- Minimal protocol using **timestamps**; the "challenge" is implicit:

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow S: & \{T_a, S\}_{K_{as}} \end{array}$$

- Nonces prevent replay of *old messages*

Reminder: shared-key unilateral authentication

- Minimal protocol using a **random number**:

$$\begin{array}{lll} \text{Message 1.} & S \rightarrow A: & N_s \\ \text{Message 2.} & A \rightarrow S: & \{N_s, S\}_{K_{as}} \end{array}$$

- Minimal protocol using **timestamps**; the "challenge" is implicit:

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow S: & \{T_a, S\}_{K_{as}} \end{array}$$

- Nonces prevent replay of *old messages*
- S is included inside the encrypted package to foil a *reflection attack* (impersonation of S to A).

Reminder: shared-key unilateral authentication

- Minimal protocol using a **random number**:

  Message 1. $S \rightarrow A$: N_s
  Message 2. $A \rightarrow S$: $\{N_s, S\}_{K_{as}}$

- Minimal protocol using **timestamps**; the "challenge" is implicit:

  Message 1. $A \rightarrow S$: $\{T_a, S\}_{K_{as}}$

- Nonces prevent replay of *old messages*
- S is included inside the encrypted package to foil a *reflection attack* (impersonation of S to A).
- Also, encrypting random strings can be risky: to prevent a **chosen-text attack** on the encryption scheme in the first case, A may include another random number in the encrypted package.

Shared-key mutual authentication

- This protocol achieves *mutual authentication* using shared keys and nonces:

$$\begin{array}{lll} \text{Message 1.} & S \to A: & N_s \\ \text{Message 2.} & A \to S: & \{N_s, N_a, S\}_{K_{as}} \\ \text{Message 3.} & S \to A: & \{N_a, N_s\}_{K_{as}} \end{array}$$

Shared-key mutual authentication

- This protocol achieves *mutual authentication* using shared keys and nonces:

$$\begin{array}{lll} \text{Message 1.} & S \rightarrow A: & N_s \\ \text{Message 2.} & A \rightarrow S: & \{N_s, N_a, S\}_{K_{as}} \\ \text{Message 3.} & S \rightarrow A: & \{N_a, N_s\}_{K_{as}} \end{array}$$

- The second nonce N_a in message 2 serves both as a challenge for message 3 and to prevent chosen-text attacks. On receiving message 2, S checks N_s was the nonce he issued in message 1, and that his name S is included in the encrypted package. He also recovers N_a to send in message 3.

Shared-key mutual authentication

- This protocol achieves *mutual authentication* using shared keys and nonces:

Message 1. $S \rightarrow A$: N_s
Message 2. $A \rightarrow S$: $\{N_s, N_a, S\}_{K_{as}}$
Message 3. $S \rightarrow A$: $\{N_a, N_s\}_{K_{as}}$

- The second nonce N_a in message 2 serves both as a challenge for message 3 and to prevent chosen-text attacks. On receiving message 2, S checks N_s was the nonce he issued in message 1, and that his name S is included in the encrypted package. He also recovers N_a to send in message 3.
- Mutual authentication may be obtained by running unilateral authentication twice, but that achieves something slightly weaker: the two authentications are not logically linked by the protocol (TOCTOU).

Outline

Challenge-response with PK decryption

- Designing public-key based protocols is also subtle. For example, it's important not to use a key-pair used for authentication for other purposes, since combining usages can compromise security.
- First PK approach: Alice demonstrates knowledge of a private key by **decrypting a challenge**.

Message 1. $S \to A$: $h(N_s), S, \{N_s, S\}_{K_a}$
Message 2. $A \to S$: N_s

Challenge-response with PK decryption

- Designing public-key based protocols is also subtle. For example, it’s important not to use a key-pair used for authentication for other purposes, since combining usages can compromise security.
- First PK approach: Alice demonstrates knowledge of a private key by **decrypting a challenge**.

$$
\begin{array}{lll}
\text{Message 1.} & S \rightarrow A: & h(N_s), S, \{ N_s, S \}_{K_a} \\
\text{Message 2.} & A \rightarrow S: & N_s
\end{array}
$$

  - Server Sam invents a nonce N_s, and challenges Alice to discover it.

Challenge-response with PK decryption

- Designing public-key based protocols is also subtle. For example, it's important not to use a key-pair used for authentication for other purposes, since combining usages can compromise security.
- First PK approach: Alice demonstrates knowledge of a private key by **decrypting a challenge**.

Message 1. $S \rightarrow A$: $h(N_s), S, \{N_s, S\}_{K_a}$
Message 2. $A \rightarrow S$: N_s

  - Server Sam invents a nonce N_s, and challenges Alice to discover it.
  - He sends a packet containing the nonce encrypted with her public key K_a and a *witness* $h(N_s)$, where h is a one-way hash function, which prevents chosen-text attacks.

Challenge-response with PK decryption

- Designing public-key based protocols is also subtle. For example, it's important not to use a key-pair used for authentication for other purposes, since combining usages can compromise security.
- First PK approach: Alice demonstrates knowledge of a private key by **decrypting a challenge**.

  Message 1. $S \rightarrow A$: $h(N_s), S, \{ N_s, S \}_{K_a}$
  Message 2. $A \rightarrow S$: N_s

  - Server Sam invents a nonce N_s, and challenges Alice to discover it.
  - He sends a packet containing the nonce encrypted with her public key K_a and a *witness* $h(N_s)$, where h is a one-way hash function, which prevents chosen-text attacks.
  - Alice decrypts, and responds with N_s only if the hash and name both match. When Sam sees his nonce N_s returned, Alice is authenticated.

Challenge-response with digital signatures

- Alice demonstrates knowledge of her signature private key by **signing a challenge**.

  Message 1. $S \rightarrow A$: N_s
  Message 2. $A \rightarrow S$: $N_a, S,\ \mathbf{S}_A(N_a, N_s, S)$

- Server Sam sends a nonce N_s. Alice replies with a message containing her own nonce N_a, the name S, and the signature for a message with both nonces and the name. She constructs the signature using her private signing function $\mathbf{S}_A$.

Challenge-response with digital signatures

- Alice demonstrates knowledge of her signature private key by **signing a challenge**.

Message 1. $S \to A$: N_s
Message 2. $A \to S$: $N_a, S, \mathbf{S}_A(N_a, N_s, S)$

- Server Sam sends a nonce N_s. Alice replies with a message containing her own nonce N_a, the name S, and the signature for a message with both nonces and the name. She constructs the signature using her private signing function $\mathbf{S}_A$.
- If the signature verifies for the plaintext N_a, N_s, S, he considers Alice authenticated.
- In both this case, and the previous slide, we assume that Sam already has the (correct) public verification function $\mathbf{V}_A$ to check Alice's signatures (wait for discussion of digital certificates).

Outline

Dealing with keys

- So far: protocols for **authentication**, assuming that any keys were securely distributed. But how?

Dealing with keys

- So far: protocols for **authentication**, assuming that any keys were securely distributed. But how?
- Many protocols have been designed for **key-exchange**. Key exchange usually establishes short-term *session keys*, which encrypt individual conversations, usually with conventional crypto.

Dealing with keys

- So far: protocols for **authentication**, assuming that any keys were securely distributed. But how?
- Many protocols have been designed for **key-exchange**. Key exchange usually establishes short-term *session keys*, which encrypt individual conversations, usually with conventional crypto.
- A new key for each conversation is good practice: if a particular key is used for a long time (or a lot of data), there is more opportunity for attack.

Dealing with keys

- So far: protocols for **authentication**, assuming that any keys were securely distributed. But how?
- Many protocols have been designed for **key-exchange**. Key exchange usually establishes short-term *session keys*, which encrypt individual conversations, usually with conventional crypto.
- A new key for each conversation is good practice: if a particular key is used for a long time (or a lot of data), there is more opportunity for attack.
- Many protocols combine **authentication and key-exchange**.

Dealing with keys . . .

- With symmetric cryptography

Dealing with keys . . .

- With symmetric cryptography
  - Use a Trusted Third Party (TTP), or

Dealing with keys . . .

- With symmetric cryptography
  - Use a Trusted Third Party (TTP), or
  - Can assume that users have fixed long-term keys which are used to exchange shorter-term *session keys*

Dealing with keys . . .

- With symmetric cryptography
  - Use a Trusted Third Party (TTP), or
  - Can assume that users have fixed long-term keys which are used to exchange shorter-term *session keys*
- For public-key cryptography, using the genuine public key is crucial (an example of data-origin authentication). Digital signatures and TTPs come together to create **digital certificates** which can be used to securely distribute public keys.

Dealing with keys . . .

- With symmetric cryptography
  - Use a Trusted Third Party (TTP), or
  - Can assume that users have fixed long-term keys which are used to exchange shorter-term *session keys*
- For public-key cryptography, using the genuine public key is crucial (an example of data-origin authentication). Digital signatures and TTPs come together to create **digital certificates** which can be used to securely distribute public keys.
  - Though in this case, the TTP can be *off-line* from the key exchange protocol (as opposed to on-line, and actively participating with the symmetric case), and is only required to preserve authenticity, and not confidentiality of key material

Key-exchange using symmetric crypto

- Usual setting: a TTP, the *Key Distribution Centre* (KDC), with whom each principal shares a key.

Key-exchange using symmetric crypto

- Usual setting: a TTP, the *Key Distribution Centre* (KDC), with whom each principal shares a key.
- Method: Alice tells the KDC (Sam) she'd like to talk to Bob. Sam generates a new session key, and encrypts two copies of it: one with Alice's key and one with Bob's key. He sends both copies to Alice. Alice decrypts her copy, sends Bob his copy, and then they can communicate securely.

Key-exchange using symmetric crypto

- Usual setting: a TTP, the *Key Distribution Centre* (KDC), with whom each principal shares a key.
- Method: Alice tells the KDC (Sam) she'd like to talk to Bob. Sam generates a new session key, and encrypts two copies of it: one with Alice's key and one with Bob's key. He sends both copies to Alice. Alice decrypts her copy, sends Bob his copy, and then they can communicate securely.
- A particular protocol:

  Message 1. $A \to S: \quad A, B$
  Message 2. $S \to A: \quad \{K_{ab}, T\}_{K_{as}}, \{K_{ab}, T\}_{K_{bs}}$
  Message 3. $A \to B: \quad \{K_{ab}, T\}_{K_{bs}}$

  Here the session key has a timestamp T to indicate its creation time.

Key-exchange using PKC

- **Hybrid cryptography** combines PK crypto for exchanging a session key with conventional crypto to communicate using the session key. Two reasons: (1) PK algorithms are slow, so bad for lots of data; (2) PK cryptosystems are vulnerable to chosen-plaintext attacks (since E_e is public).
- Assume that Alice can generate good session keys, and that she already has Bob's public key K_b. Then she can send him the session key K_{ab} and a message M encrypted with it, in one go (for extra protection, she could sign and date the message):

$$A \to B : \quad \{ K_{ab} \}_{K_b}, \{ M \}_{K_{ab}}$$

- This requires just a single message (not interactive), so it works on a *store-and-forward* network (e.g., email), or for offline storage.
- Next: how can we be sure that Alice has the right public key?

Digital Certificates

- A **digital certificate** bundles a public key and/or signature verification function with identification data; the bundle is signed by a trusted **certification authority** (CA) who verified the data.

Digital Certificates

- A **digital certificate** bundles a public key and/or signature verification function with identification data; the bundle is signed by a trusted **certification authority** (CA) who verified the data.
- E.g., a certificate for Bob may take the form:

$$C_B = M, \mathbf{S}_{CA}(M) \qquad \text{where } M = (T_s, T_e, B, \mathbf{V}_B)$$

  where $\mathbf{S}_{CA}$ is the CA's signing function; T_s, T_e are start-time and end-time of validity.

Digital Certificates

- A **digital certificate** bundles a public key and/or signature verification function with identification data; the bundle is signed by a trusted **certification authority** (CA) who verified the data.
- E.g., a certificate for Bob may take the form:

$$C_B = M, \mathbf{S}_{CA}(M) \qquad \text{where } M = (T_s, T_e, B, \mathbf{V}_B)$$

  where $\mathbf{S}_{CA}$ is the CA's signing function; T_s, T_e are start-time and end-time of validity.
- Compared with a secure directory of public keys, this protects against MITM attacks; only the CA's verification function needs to be distributed securely. (But the CA's private signing key becomes a critical vulnerability.)

Digital Certificates

- A **digital certificate** bundles a public key and/or signature verification function with identification data; the bundle is signed by a trusted **certification authority** (CA) who verified the data.
- E.g., a certificate for Bob may take the form:

  $$C_B = M, \mathbf{S}_{CA}(M) \qquad \text{where } M = (T_s, T_e, B, \mathbf{V}_B)$$

  where $\mathbf{S}_{CA}$ is the CA's signing function; T_s, T_e are start-time and end-time of validity.
- Compared with a secure directory of public keys, this protects against MITM attacks; only the CA's verification function needs to be distributed securely. (But the CA's private signing key becomes a critical vulnerability.)
- X.509 uses this model. Each certificate is signed by one CA, and there is a chain of certificates until a *root* or **self-signed** certificate is reached. Common root certificates are built into web browsers.

Key-exchange using certificates

Here is a way to exchange a session key using certificates. First, Alice asks the server Sam for her certificate and Bob's certificate. Then she generates a session key K_{ab} and a timestamp T_a to send to Bob. She signs these with her signing function $\mathbf{S}_A$, encrypts them with Bob's public key, and sends them to him together with the certificates.

Message 1. $A \rightarrow S$: A, B
Message 2. $S \rightarrow A$: C_a, C_b
Message 3. $A \rightarrow B$: $C_a, C_b, \{K_{ab}, T_a, \mathbf{S}_A(K_{ab}, T_a)\}_{K_b}$

Key-exchange using certificates

Here is a way to exchange a session key using certificates. First, Alice asks the server Sam for her certificate and Bob's certificate. Then she generates a session key K_{ab} and a timestamp T_a to send to Bob. She signs these with her signing function $\mathbf{S}_A$, encrypts them with Bob's public key, and sends them to him together with the certificates.

Message 1. $A \rightarrow S$: A, B
Message 2. $S \rightarrow A$: C_a, C_b
Message 3. $A \rightarrow B$: $C_a, C_b, \{K_{ab}, T_a, \mathbf{S}_A(K_{ab}, T_a)\}_{K_b}$

Denning and Sacco proposed this key-exchange protocol in 1981.

Key-exchange using certificates

Here is a way to exchange a session key using certificates. First, Alice asks the server Sam for her certificate and Bob's certificate. Then she generates a session key K_{ab} and a timestamp T_a to send to Bob. She signs these with her signing function $\mathbf{S}_A$, encrypts them with Bob's public key, and sends them to him together with the certificates.

Message 1. $A \to S$: A, B
Message 2. $S \to A$: C_a, C_b
Message 3. $A \to B$: $C_a, C_b, \{K_{ab}, T_a, \mathbf{S}_A(K_{ab}, T_a)\}_{K_b}$

Denning and Sacco proposed this key-exchange protocol in 1981. In 1994, Abadi and Needham pointed out a fatal flaw. That a serious flaw went unnoticed for so long in such a simple protocol shows quite how delicate protocol design is, and suggests that **formal analysis** is called for.

Key-exchange using certificates

Here is a way to exchange a session key using certificates. First, Alice asks the server Sam for her certificate and Bob's certificate. Then she generates a session key K_{ab} and a timestamp T_a to send to Bob. She signs these with her signing function $\mathbf{S}_A$, encrypts them with Bob's public key, and sends them to him together with the certificates.

Message 1. $A \to S$: A, B
Message 2. $S \to A$: C_a, C_b
Message 3. $A \to B$: $C_a, C_b, \{K_{ab}, T_a, \mathbf{S}_A(K_{ab}, T_a)\}_{K_b}$

Denning and Sacco proposed this key-exchange protocol in 1981. In 1994, Abadi and Needham pointed out a fatal flaw. That a serious flaw went unnoticed for so long in such a simple protocol shows quite how delicate protocol design is, and suggests that **formal analysis** is called for. Can you guess what the flaw is?

Key-exchange using certificates

Here is a way to exchange a session key using certificates. First, Alice asks the server Sam for her certificate and Bob's certificate. Then she generates a session key K_{ab} and a timestamp T_a to send to Bob. She signs these with her signing function $\mathbf{S}_A$, encrypts them with Bob's public key, and sends them to him together with the certificates.

Message 1. $A \to S: \quad A, B$
Message 2. $S \to A: \quad C_a, C_b$
Message 3. $A \to B: \quad C_a, C_b, \{K_{ab}, T_a, \mathbf{S}_A(K_{ab}, T_a)\}_{K_b}$

Denning and Sacco proposed this key-exchange protocol in 1981. In 1994, Abadi and Needham pointed out a fatal flaw. That a serious flaw went unnoticed for so long in such a simple protocol shows quite how delicate protocol design is, and suggests that **formal analysis** is called for. Can you guess what the flaw is? (Hint: consider signed packet)

Outline

Key-exchange with authentication

- It makes sense to combine key-exchange with authentication. A direct way to achieve this is by adding names to the (symmetric key) protocol for key exchange given earlier:

  Message 1. $A \rightarrow S$: A, B
  Message 2. $S \rightarrow A$: $\{A, B, K_{ab}, T\}_{K_{as}}, \{A, B, K_{ab}, T\}_{K_{bs}}$
  Message 3. $A \rightarrow B$: $\{A, B, K_{ab}, T\}_{K_{bs}}$

  Instead of encrypting just the key K_{ab} and timestamp T, Sam sends a package containing the names A, B as well. The names allow Alice and Bob to verify they're talking to the right people, providing authentication.

Key-exchange with authentication

- It makes sense to combine key-exchange with authentication. A direct way to achieve this is by adding names to the (symmetric key) protocol for key exchange given earlier:

  Message 1. $A \to S$: A, B
  Message 2. $S \to A$: $\{A, B, K_{ab}, T\}_{K_{as}}, \{A, B, K_{ab}, T\}_{K_{bs}}$
  Message 3. $A \to B$: $\{A, B, K_{ab}, T\}_{K_{bs}}$

  Instead of encrypting just the key K_{ab} and timestamp T, Sam sends a package containing the names A, B as well. The names allow Alice and Bob to verify they're talking to the right people, providing authentication.
- Protocols of this kind have been much studied, with variations using nonces instead of time stamps, changing the pattern of communications, or trying to optimise the communications.

Key-exchange with authentication

- It makes sense to combine key-exchange with authentication. A direct way to achieve this is by adding names to the (symmetric key) protocol for key exchange given earlier:

  Message 1. $A \rightarrow S: \quad A, B$
  Message 2. $S \rightarrow A: \quad \{A, B, K_{ab}, T\}_{K_{as}}, \{A, B, K_{ab}, T\}_{K_{bs}}$
  Message 3. $A \rightarrow B: \quad \{A, B, K_{ab}, T\}_{K_{bs}}$

  Instead of encrypting just the key K_{ab} and timestamp T, Sam sends a package containing the names A, B as well. The names allow Alice and Bob to verify they're talking to the right people, providing authentication.
- Protocols of this kind have been much studied, with variations using nonces instead of time stamps, changing the pattern of communications, or trying to optimise the communications.
- Some examples follow. . .

Wide-mouthed frog

- Probably the simplest symmetric key management protocol using a trusted server. Relies on synchronized clocks and the assumption that Alice can generate good keys.

Message 1. $A \rightarrow S$: $A, \{T_a, B, K_{ab}\}_{K_{as}}$
Message 2. $S \rightarrow B$: $\{T_s, A, K_{ab}\}_{K_{bs}}$

Wide-mouthed frog

- Probably the simplest symmetric key management protocol using a trusted server. Relies on synchronized clocks and the assumption that Alice can generate good keys.

Message 1. $A \rightarrow S$: $A, \{T_a, B, K_{ab}\}_{K_{as}}$
Message 2. $S \rightarrow B$: $\{T_s, A, K_{ab}\}_{K_{bs}}$

- In 1, Alice concatenates a timestamp, Bob's name, a random session key, and encrypts under her key shared with Sam.

Wide-mouthed frog

- Probably the simplest symmetric key management protocol using a trusted server. Relies on synchronized clocks and the assumption that Alice can generate good keys.

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow S: & A, \{T_a, B, K_{ab}\}_{K_{as}} \\ \text{Message 2.} & S \rightarrow B: & \{T_s, A, K_{ab}\}_{K_{bs}} \end{array}$$

- In 1, Alice concatenates a timestamp, Bob's name, a random session key, and encrypts under her key shared with Sam.
- Sam decrypts the message from Alice and checks that the message is timely. Then he concatenates a new timestamp, Alice's name, and the key. He encrypts this under the key he shares with Bob, and sends the package along to Bob in Message 2.

Wide-mouthed frog

- Probably the simplest symmetric key management protocol using a trusted server. Relies on synchronized clocks and the assumption that Alice can generate good keys.

Message 1. $A \rightarrow S$: $A, \{T_a, B, K_{ab}\}_{K_{as}}$
Message 2. $S \rightarrow B$: $\{T_s, A, K_{ab}\}_{K_{bs}}$

- In 1, Alice concatenates a timestamp, Bob's name, a random session key, and encrypts under her key shared with Sam.
- Sam decrypts the message from Alice and checks that the message is timely. Then he concatenates a new timestamp, Alice's name, and the key. He encrypts this under the key he shares with Bob, and sends the package along to Bob in Message 2.
- Bob decrypts this message, and checks that message contains a newer timestamp than any seen before. If so, he accepts the session key.

The Needham-Schroeder protocol (flawed)

A protocol using nonces and a server to generate keys.

Message 1. $A \to S: \quad A, B, N_a$

(1) A makes contact with the server

The Needham-Schroeder protocol (flawed)

A protocol using nonces and a server to generate keys.

Message 1. $A \rightarrow S: \quad A, B, N_a$
Message 2. $S \rightarrow A: \quad \{ N_a, B, K_{ab}, \{ K_{ab}, A \}_{K_{bs}} \}_{K_{as}}$

(1) A makes contact with the server who **(2)** provides the session key K_{ab} and a package for transmission to B containing the same key.

The Needham-Schroeder protocol (flawed)

A protocol using nonces and a server to generate keys.

Message 1. $A \to S$: A, B, N_a
Message 2. $S \to A$: $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
Message 3. $A \to B$: $\{K_{ab}, A\}_{K_{bs}}$
Message 4. $B \to A$: $\{N_b\}_{K_{ab}}$

(1) A makes contact with the server who **(2)** provides the session key K_{ab} and a package for transmission to B containing the same key. Then **(3)** A transmits the package to B, and B initiates a handshake with A **(4)** using N_b to prevent replay.

The Needham-Schroeder protocol (flawed)

A protocol using nonces and a server to generate keys.

Message 1. $A \to S$: A, B, N_a
Message 2. $S \to A$: $\{ N_a, B, K_{ab}, \{ K_{ab}, A \}_{K_{bs}} \}_{K_{as}}$
Message 3. $A \to B$: $\{ K_{ab}, A \}_{K_{bs}}$
Message 4. $B \to A$: $\{ N_b \}_{K_{ab}}$
Message 5. $A \to B$: $\{ N_b - 1 \}_{K_{ab}}$

(1) A makes contact with the server who **(2)** provides the session key K_{ab} and a package for transmission to B containing the same key. Then **(3)** A transmits the package to B, and B initiates a handshake with A **(4)** using N_b to prevent replay. The final message from A uses $N_b - 1$ to distinguish it from the previous message from B.

The Needham-Schroeder protocol (flawed)

A protocol using nonces and a server to generate keys.

Message 1. $A \to S$: A, B, N_a
Message 2. $S \to A$: $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
Message 3. $A \to B$: $\{K_{ab}, A\}_{K_{bs}}$
Message 4. $B \to A$: $\{N_b\}_{K_{ab}}$
Message 5. $A \to B$: $\{N_b - 1\}_{K_{ab}}$

(1) A makes contact with the server who **(2)** provides the session key K_{ab} and a package for transmission to B containing the same key. Then **(3)** A transmits the package to B, and B initiates a handshake with A **(4)** using N_b to prevent replay. The final message from A uses $N_b - 1$ to distinguish it from the previous message from B.
Can you guess what the flaw is?

The Needham-Schroeder protocol (flawed)

A protocol using nonces and a server to generate keys.

Message 1. $A \to S$: A, B, N_a
Message 2. $S \to A$: $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
Message 3. $A \to B$: $\{K_{ab}, A\}_{K_{bs}}$
Message 4. $B \to A$: $\{N_b\}_{K_{ab}}$
Message 5. $A \to B$: $\{N_b - 1\}_{K_{ab}}$

(1) A makes contact with the server who **(2)** provides the session key K_{ab} and a package for transmission to B containing the same key. Then **(3)** A transmits the package to B, and B initiates a handshake with A **(4)** using N_b to prevent replay. The final message from A uses $N_b - 1$ to distinguish it from the previous message from B.

Can you guess what the flaw is? (Hint: consider if a session key K_{ab} is broken)

Kerberos (simplified V4)

- A repaired version of Needham-Schroeder, using synchronized clocks and trusted servers. Used in Windows 2000 (& DICE).

Kerberos (simplified V4)

- A repaired version of Needham-Schroeder, using synchronized clocks and trusted servers. Used in Windows 2000 (& DICE).
- Scenario: Alice wishes to access a resource B. First she must log in to the authentication server to get a *ticket-granting ticket* (TGT), which is encrypted with her secret key. It contains a session key K_{ab} for Alice to use with a *ticket-granting server* (TGS).

Kerberos (simplified V4)

- A repaired version of Needham-Schroeder, using synchronized clocks and trusted servers. Used in Windows 2000 (& DICE).
- Scenario: Alice wishes to access a resource B. First she must log in to the authentication server to get a *ticket-granting ticket* (TGT), which is encrypted with her secret key. It contains a session key K_{ab} for Alice to use with a *ticket-granting server* (TGS).
- Alice contacts TGS S and asks to access B. It grants her a *ticket* for using B with a limited duration. She passes this to B, with an *authenticator*. Optional: B replies for mutual authentication.

Message 1.	$A \to S$:	A, B
Message 2.	$S \to A$:	$\{T_s, L, K_{ab}, B, \{T_s, L, K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
Message 3.	$A \to B$:	$\{T_s, L, K_{ab}, A\}_{K_{bs}}, \{A, T_a\}_{K_{ab}}$
Message 4.	$B \to A$:	$\{T_a + 1\}_{K_{ab}}$

Here, T_s and T_a are timestamps, and L is a *lifetime*.

Outline

Protocols: summary

- **Weak authentication protocols** (e.g., traditional passwords). Stored *time-invariant* secrets or hashes of secrets. Added salt.
- **Strong authentication protocols**. Challenge-response with time-variant parameters (**nonces** or **timestamps**) to guarantee freshness and prevent **replay attacks**. Shared key and public key protocols, demonstrating knowledge of keys. Another kind of authentication protocol we haven't looked at (yet): **zero-knowledge** protocols, are based on demonstrating knowledge without giving way any further information, provably.
- **Key-exchange protocols**. Using shared keys, public keys, and digital signatures/certificates.
- **Key-exchange and authentication protocols**. Using shared keys, public keys. Well-known ones are Kerberos and Needham-Schroeder.

References

- Ross Anderson.
  Security Engineering: A Comprehensive Guide to Building Dependable Distributed Systems. 2nd Edition
  Wiley & Sons, 2008.

- Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone, editors.
  Handbook of Applied Cryptography ("HAC").
  CRC Press Series on Discrete Mathematics and Its Applications. CRC Press, 1997.
  Online version at `http://cacr.math.uwaterloo.ca/hac`.

Recommended Reading

Chapter 10 of HAC (10.1–10.3). Chapter 5 of Anderson.