02239 Data Security
Protocol Security I

Sebastian Mödersheim

September 25, 2024

Overview of Problem Areas

Example: Alice wants to tell her bank to transfer 1000 Kr. to Bob.

- What are the involved goals?
  - ★ Authentication/Integrity
  - ★ Confidentiality/Privacy
- Involved Cryptographic Protocols: could be
  - ★ TLS
  - ★ The banking application
  - ★ Some login like MitID (also over TLS? Same session?)
- Implementation
  - ★ Crypto API
  - ★ All the non-crypto aspects, like parsing message formats.

Mathematical Abstraction

 ← 

- A clearly defined game
  - ★ "winnable" is a clearly defined
- Like in chess, it is still very complex for automated analysis
  - ★ astronomical or infinite size of search trees
  - ★ computers are sometimes better than humans at it...
- Mind the gap
  - ★ Be clear about the abstractions and assumptions made
  - ★ Separation of concerns

Protocol Security

"Logical Hacking" and Security Proofs

- What is an "attack"? (and what is not?)
- How can we automatically find attacks?
- How can we prove the security of a system?
  - ... not just with respect to currently known attacks, but against any attacks!
  - Is that even possible?
  - Can we do that even automatically?
- How can we build systems that are secure?

This requires a precise definitions of

- the systems in questions
- its goals
- the assumptions (in particular, the intruder)

Main Takeaways from Protocol Security

- You may easily overlook security problems if you are not precise about: what the system does, what the security goals are, and what the intruder can do.
- Even if you are precise on this, manual analysis can overlook problems. Tools can often find something you overlooked.
  - In particular one should by default assume that participants could be dishonest and all dishonest people work together.
- You can often easily avoid a lot of problems by being explicit in messages about who is communicating and what the message is supposed to say
- You can often easily make a system resilient against guessing attacks, even though users use weak passwords.

Today's Program

1. **Alice and Bob**

Alice and Bob

Alice and Bob notation

- aka Message Sequence Charts
  aka Protocol Narrations
- popular informal notation for protocols

```
A -> B: {NA,A}pk(B)
B -> A: {NA,NB}pk(A)
A -> B: {NB}pk(B)
```

AnB

- A formal language based on Alice and Bob notation
  - ★ Defining the roles of the protocol and their initial knowledge
    - ▶ Indirectly defining the intruder's initial knowledge
    - ▶ Indirectly defining how agents execute the protocol
  - ★ Defining the security goals of the protocol
- OFMC: Open-Source Fixedpoint Model-Checker.
  - ★ Automatically finding attacks in protocols
  - ★ AnB is one input language for OFMC.

Capture-the-flag challenges in AnB/OFMC: given a flawed protocol, find a fix that OFMC cannot attack anymore—without changing initial knowledge and goals of the protocol.

Example

Step-by-step development of a protocol for the following scenario:

- A (Alice) and B (Bob) want a secure connection with each other, but have no prior security relationship.
- There is a server s and both A and B each have with s a shared secret key $sk(A, s)$ and $sk(B, s)$, respectively.
- s should now help A and B establish a secure connection, i.e., a shared secret key KAB that they can use to communicate with each other.
- This only works if s is honest (why?)

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  $A: \quad A, B, s, sk(A, s)$;
  $B: \quad A, B, s, sk(B, s)$;
  $s: \quad A, B, s, sk(A, s), sk(B, s)$;
Actions :

$$
\begin{array}{llll}
A & \rightarrow & s & : & A, B \\
s & \rightarrow & A & : & KAB \\
A & \rightarrow & B & : & KAB
\end{array}
$$

Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- A, B are variables of type Agent: they can be instantiated with any agent name during the run of the protocol

First version

`Protocol` : *KeyExchange*
`Types` :
  `Agent` A, B, s;
  `Symmetric_key` KAB;
  `Function` sk;
`Knowledge` :
  $A: \quad A, B, s, sk(A, s);$
  $B: \quad A, B, s, sk(B, s);$
  $s: \quad A, B, s, sk(A, s), sk(B, s);$
`Actions` :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
`Goals` :
  KAB `secret between` A, B, s
  A `authenticates` s `on` KAB
  B `authenticates` s `on` KAB

- A, B are variables of type `Agent`: they can be instantiated with any agent name during the run of the protocol
  - ... including the intruder i
  - The intruder can thus play the role of A or B

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :

$$
\begin{array}{llll}
A & \rightarrow & s & : A, B \\
s & \rightarrow & A & : KAB \\
A & \rightarrow & B & : KAB
\end{array}
$$

Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- A, B are variables of type Agent: they can be instantiated with any agent name during the run of the protocol
  - ... including the intruder i
  - The intruder can thus play the role of A or B
- s is a constant of type Agent: there is only one agent called s who will play in all sessions

First version

```
Protocol : KeyExchange
Types :
    Agent A, B, s;
    Symmetric_key KAB;
    Function sk;
Knowledge :
    A :  A, B, s, sk(A, s);
    B :  A, B, s, sk(B, s);
    s :  A, B, s, sk(A, s), sk(B, s);
Actions :
    A -> s : A, B
    s -> A : KAB
    A -> B : KAB
Goals :
    KAB secret between A, B, s
    A authenticates s on KAB
    B authenticates s on KAB
```

- A, B are variables of type `Agent`: they can be instantiated with any agent name during the run of the protocol
  - ... including the intruder i
  - The intruder can thus play the role of A or B
- s is a constant of type `Agent`: there is only one agent called s who will play in all sessions
  - the intruder cannot play the role of s
  - s is thus a trusted third party

First version

`Protocol` : *KeyExchange*
`Types` :
  `Agent` A, B, s;
  `Symmetric_key` KAB;
  `Function` sk;
`Knowledge` :
  $A: \quad A, B, s, sk(A, s)$;
  $B: \quad A, B, s, sk(B, s)$;
  $s: \quad A, B, s, sk(A, s), sk(B, s)$;
`Actions` :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
`Goals` :
  KAB `secret between` A, B, s
  A `authenticates` s `on` KAB
  B `authenticates` s `on` KAB

- KAB is a variable of type symmetric key.
  - ★ The value will be freshly created during the protocol run.

First version

`Protocol` : *KeyExchange*
`Types` :
  `Agent` A, B, s;
  `Symmetric_key` KAB;
  `Function` sk;
`Knowledge` :
  $A: \quad A, B, s, sk(A, s)$;
  $B: \quad A, B, s, sk(B, s)$;
  $s: \quad A, B, s, sk(A, s), sk(B, s)$;
`Actions` :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
`Goals` :
  KAB `secret between` A, B, s
  A `authenticates` s `on` KAB
  B `authenticates` s `on` KAB

- *KAB* is a variable of type symmetric key.
  - ★ The value will be freshly created during the protocol run.
- *sk* is a user-defined function. We use it to model shared secret keys of two agents that are fixed before the protocol run.

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- It is necessary to specify an initial knowledge for every role of the protocol.
  - ★ It determines how agents send and receive messages

First version

`Protocol` : *KeyExchange*
`Types` :
  `Agent` A, B, s;
  `Symmetric_key` KAB;
  `Function` sk;
`Knowledge` :
  $A: \quad A, B, s, sk(A, s)$;
  $B: \quad A, B, s, sk(B, s)$;
  $s: \quad A, B, s, sk(A, s), sk(B, s)$;
`Actions` :

$$\begin{array}{lllll} A & \rightarrow & s & : & A, B \\ s & \rightarrow & A & : & KAB \\ A & \rightarrow & B & : & KAB \end{array}$$

`Goals` :
  KAB `secret between` A, B, s
  A `authenticates` s `on` KAB
  B `authenticates` s `on` KAB

- It is necessary to specify an initial knowledge for every role of the protocol.
  - ★ It determines how agents send and receive messages
- Typically everybody knows all agent names.

First version

$\texttt{Protocol}: \mathit{KeyExchange}$
$\texttt{Types}:$
$\quad \texttt{Agent}\ A, B, s;$
$\quad \texttt{Symmetric\_key}\ KAB;$
$\quad \texttt{Function}\ sk;$
$\texttt{Knowledge}:$
$\quad A: \quad A, B, s, sk(A, s);$
$\quad B: \quad A, B, s, sk(B, s);$
$\quad s: \quad A, B, s, sk(A, s), sk(B, s);$
$\texttt{Actions}:$

$$
\begin{array}{lllll}
A & \rightarrow & s & : & A, B \\
s & \rightarrow & A & : & KAB \\
A & \rightarrow & B & : & KAB
\end{array}
$$

$\texttt{Goals}:$
$\quad KAB\ \texttt{secret between}\ A, B, s$
$\quad A\ \texttt{authenticates}\ s\ \texttt{on}\ KAB$
$\quad B\ \texttt{authenticates}\ s\ \texttt{on}\ KAB$

- It is necessary to specify an initial knowledge for every role of the protocol.
  - ★ It determines how agents send and receive messages
- Typically everybody knows all agent names.
- A knows a secret key with the server: $sk(A, s)$
- B knows a secret key with the server: $sk(B, s)$
- s knows both $sk(A, s)$ and $sk(B, s)$

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :

$$\begin{array}{llll} A & \to & s & : & A, B \\ s & \to & A & : & KAB \\ A & \to & B & : & KAB \end{array}$$

Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- The idea of the protocol is to establish a fresh secret key KAB between A and B
  - ★ A and B initially do not have any key material with each other
  - ★ but both have a shared key with trusted third party s that can be used for establishing KAB.
- Question: why would this be impossible if we had an untrusted S instead of s?

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  $A: \quad A, B, s, sk(A, s)$;
  $B: \quad A, B, s, sk(B, s)$;
  $s: \quad A, B, s, sk(A, s), sk(B, s)$;
Actions :

$$
\begin{array}{lllll}
A & \rightarrow & s & : & A, B \\
s & \rightarrow & A & : & KAB \\
A & \rightarrow & B & : & KAB
\end{array}
$$

Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- The knowledge section also determines the initial knowledge of the intruder:
  - ★ Say $A = i$ and $B = b$ for agent i in role A and honest b in role B.
  - ★ Then the intruder gets the knowledge of A under this instantiation: $i, b, s, sk(i, s)$
  - ★ The intruder thus also has a shared secret key with s!
  - ★ That's only fair: the intruder should know enough to play a protocol role as a normal user.

First version

$\texttt{Protocol}$: *KeyExchange*
$\texttt{Types}$:
  $\texttt{Agent}$ A, B, s;
  $\texttt{Symmetric\_key}$ KAB;
  $\texttt{Function}$ sk;
$\texttt{Knowledge}$:
  $A: \quad A, B, s, sk(A, s)$;
  $B: \quad A, B, s, sk(B, s)$;
  $s: \quad A, B, s, sk(A, s), sk(B, s)$;
$\texttt{Actions}$:

$$
\begin{array}{lclcl}
A & \rightarrow & s & : & A, B \\
s & \rightarrow & A & : & KAB \\
A & \rightarrow & B & : & KAB
\end{array}
$$

$\texttt{Goals}$:
  KAB $\texttt{secret between}$ A, B, s
  A $\texttt{authenticates}$ s $\texttt{on}$ KAB
  B $\texttt{authenticates}$ s $\texttt{on}$ KAB

- The protocol starts by A contacting s stating the names of A and B
- Without crypto, there is no reliable information about senders and receivers.
- The intruder may intercept messages sent by honest agents, and insert arbitrary messages as if coming from any agent.
- A and B are not IP addresses, but unique identifiers (think domain name or user name/CPR).
- All agent names as public for now. Privacy: later lecture.

First version

`Protocol` : *KeyExchange*
`Types` :
  `Agent` A, B, s;
  `Symmetric_key` KAB;
  `Function` sk;
`Knowledge` :
  $A: \quad A, B, s, sk(A, s)$;
  $B: \quad A, B, s, sk(B, s)$;
  $s: \quad A, B, s, sk(A, s), sk(B, s)$;
`Actions` :

$$\begin{array}{lllll} A & \rightarrow & s & : & A, B \\ s & \rightarrow & A & : & KAB \\ A & \rightarrow & B & : & KAB \end{array}$$

`Goals` :
  KAB `secret between` A, B, s
  A `authenticates` s `on` KAB
  B `authenticates` s `on` KAB

- The server generates a fresh shared key KAB for A and B.
  - The entity first using a non-agent variable is the creator.
- Here, KAB is sent in clear text to A. This obviously is not secure in an intruder-controlled network.
- In the last step A forwards the key to B (also in clear...)
- The server cannot directly send the key to both A and B, because a message can only have one recipient who has to be the sender of the next message

First version

`Protocol` : *KeyExchange*
`Types` :
  `Agent` A, B, s;
  `Symmetric_key` KAB;
  `Function` sk;
`Knowledge` :
  A : $A, B, s, sk(A, s)$;
  B : B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
`Actions` :

$$
\begin{array}{llll}
A & \rightarrow & s & : & A, B \\
s & \rightarrow & A & : & KAB \\
A & \rightarrow & B & : & KAB
\end{array}
$$

`Goals` :
  KAB `secret between` A, B, s
  A `authenticates` s `on` B, KAB
  B `authenticates` s `on` A, KAB

- The secrecy goal: only A, B, and s may know the key.
- The authentication goals: later

First version

```
Protocol : KeyExchange
Types :
    Agent A, B, s;
    Symmetric_key KAB;
    Function sk;
Knowledge :
    A :  A, B, s, sk(A, s);
    B :  A, B, s, sk(B, s);
    s :  A, B, s, sk(A, s), sk(B, s);
Actions :
    A → s : A, B
    s → A : KAB
    A → B : KAB
Goals :
    KAB secret between A, B, s
    A authenticates s on B, KAB
    B authenticates s on A, KAB
```

Running OFMC we get an attack:

```
SUMMARY:
  ATTACK_FOUND
GOAL:
  secrets

ATTACK TRACE:
i -> (s,1): x32,x31
(s,1) -> i: KAB(1)
i can produce secret KAB(1)

secret leaked: KAB(1)
```

First: try to associate attack steps with protocol steps

First version

$$\begin{array}{llll} A & \to & s & : & A, B \\ s & \to & A & : & KAB \\ A & \to & B & : & KAB \end{array}$$

```
i -> (s,1): x32,x31
(s,1) -> i: KAB(1)
i can produce secret KAB(1)
```

- OFMC uses internal variables like `x32` and `x31` for things the intruder can arbitrarily choose.
  - ★ Here, the intruder can choose any agent names for A and B
- `KAB(1)` means a fresh key that was generated by an honest agent – the number (1) is to make it unique.
- `(s,1)` means server in session 1 (sometimes an attack may involve several sessions/runs of the protocol)
- `i` is the intruder

1. Here the intruder contacts the server s posing as some agent `x32` (role A) who wants to talk to `x31` (role B).
2. The server generates a new key `KAB(1)` for `x32` and `x31` and sends it.
3. The intruder sees this key, violating secrecy.

How to Encrypt this?

```
A->s: A,B
s->A: {| KAB |}sk(A,s)
A->B: {| KAB |}sk(B,s)
```

```
ofmc: Protocol not executable:
At the following state of the knowledge:
...one cannot compose the
following message:
{|KAB|}sk(B,s)
sk(B,s)
|sk
```

- $\{\!|KAB|\!\}_{sk(A,s)}$ means symmetric encryption of KAB with key $sk(A,s)$.
- The server can do that, knowing $sk(A,s)$.
- However A cannot produce $\{\!|KAB|\!\}_{sk(B,s)}$ for B.
- OFMC rejects this specification since A cannot generate a message that the protocol tells her to send.
  - ★ In the error message you can see what OFMC tried: the message $\{\!|KAB|\!\}_{sk(B,s)}$ is not known to A, and neither is $sk(B,s)$ nor the entire function sk.

Second Version

```
A->s: A,B
s->A: {| KAB |}sk(A,s),
      {| KAB |}sk(B,s)
A->B: {| KAB |}sk(B,s)
```

```
GOAL:
  weak_auth

i -> (s,1): x32,x401
(s,1) -> i: {|KAB(1)|}_(sk(x32,s)),
            {|KAB(1)|}_(sk(x401,s))
i -> (x401,1): {|KAB(1)|}_(sk(x401,s))
```

- In the second version, *s* generates both encrypted messages.
  - ★ *A* cannot decrypt the second one, but she can forward it to *B*.
- This is now a meaningful specification, but OFMC finds an attack:

Second Version

```
A->s: A,B
s->A: {| KAB |}sk(A,s),
      {| KAB |}sk(B,s)
A->B: {| KAB |}sk(B,s)
```

```
GOAL:
  weak_auth

i -> (s,1): x32,x401
(s,1) -> i: {|KAB(1)|}_(sk(x32,s)),
            {|KAB(1)|}_(sk(x401,s))
i -> (x401,1): {|KAB(1)|}_(sk(x401,s))
```

- In the second version, *s* generates both encrypted messages.
  - ★ *A* cannot decrypt the second one, but she can forward it to *B*.
- This is now a meaningful specification, but OFMC finds an attack:
  - ★ The intruder again chooses two agent names, and the server generates encrypted keys for them.
  - ★ The intruder forwards the part for `x401` as required in the protocol.

Second Version

```
A->s: A,B
s->A: {| KAB |}sk(A,s),
      {| KAB |}sk(B,s)
A->B: {| KAB |}sk(B,s)
```

```
GOAL:
  weak_auth

i -> (s,1): x32,x401
(s,1) -> i: {|KAB(1)|}_(sk(x32,s)),
            {|KAB(1)|}_(sk(x401,s))
i -> (x401,1): {|KAB(1)|}_(sk(x401,s))
```

- In the second version, *s* generates both encrypted messages.
  - ★ *A* cannot decrypt the second one, but she can forward it to *B*.
- This is now a meaningful specification, but OFMC finds an attack:
  - ★ The intruder again chooses two agent names, and the server generates encrypted keys for them.
  - ★ The intruder forwards the part for x401 as required in the protocol.
  - ★ So how does this represent an attack?

Second Version

```
                                GOAL: weak_auth
A->s: A,B                       ATTACK TRACE:
s->A: {| KAB |}sk(A,s),         i -> (s,1): x32,x401
      {| KAB |}sk(B,s)          (s,1) -> i: {|KAB(1)|}_(sk(x32,s)),
A->B: A,B,                                  {|KAB(1)|}_(sk(x401,s))
      {| KAB |}sk(B,s)          i -> (x401,1): x30,x401,
                                               {|KAB(1)|}_(sk(x401,s))
```

- Adding the agent names in clear text to the messages allows to see what's going wrong:
  - ★ To s, the intruder claims to be `x32`
  - ★ To B (`x401`), the intruder claims to be `x30`
- Thus there is confusion between B and s as to who A is
  This violates the goal

```
B authenticates s on A,KAB;
```

Second Version

```
A->s: A,B
s->A: {| KAB |}sk(A,s),
      {| KAB |}sk(B,s)
A->B: A,B,
      {| KAB |}sk(B,s)
```

```
GOAL: weak_auth
ATTACK TRACE:
i -> (s,1): x32,x401
(s,1) -> i: {|KAB(1)|}_(sk(x32,s)),
            {|KAB(1)|}_(sk(x401,s))
i -> (x401,1): x30,x401,
               {|KAB(1)|}_(sk(x401,s))
```

- Adding the agent names in clear text to the messages allows to see what's going wrong:
  - ★ To s, the intruder claims to be `x32`
  - ★ To B (`x401`), the intruder claims to be `x30`
- Thus there is confusion between B and s as to who A is
  This violates the goal

  ```
  B authenticates s on A,KAB;
  ```

- Suppose `x32=i`, then the intruder can see `KAB(1)` while B thinks he shares `KAB(1)` with `x30`.

Third Version

```
A->s: A,B
s->A: {| B,KAB |}sk(A,s),
      {| A,KAB |}sk(B,s)
A->B: {| A,KAB |}sk(B,s)
```

```
GOAL: weak_auth
ATTACK TRACE:
i -> (s,1): x401,x30
(s,1) -> i: {|x30,KAB(1)|}_(sk(x401,s)),
            {|x401,KAB(1)|}_(sk(x30,s))
i -> (x401,1): {|x30,KAB(1)|}_(sk(x401,s)
```

- Third version adds the name of the other party to the encrypted message.
- There is an attack, but it is a bit hard to see what is wrong.
- Let us replace the variables in the attack trace with concrete agent names *a* and *b*.

Third Version

```
A->s: A,B
s->A: {| B,KAB |}sk(A,s),
      {| A,KAB |}sk(B,s)
A->B: {| A,KAB |}sk(B,s)
```

```
GOAL: weak_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|b,KAB(1)|}_(sk(a,s)),
            {|a,KAB(1)|}_(sk(b,s))
i -> (a,1): {|b,KAB(1)|}_(sk(a,s))
```

- Third version adds the name of the other party to the encrypted message.
- From s's point of view: role A is played by a, role B by b.

Third Version

```
A->s: A,B
s->A: {| B,KAB |}sk(A,s),
      {| A,KAB |}sk(B,s)
A->B: {| A,KAB |}sk(B,s)
```

```
GOAL: weak_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|b,KAB(1)|}_(sk(a,s)),
            {|a,KAB(1)|}_(sk(b,s))
i -> (a,1): {|b,KAB(1)|}_(sk(a,s))
```

- Third version adds the name of the other party to the encrypted message.
- From s's point of view: role A is played by a, role B by b.
- From a's point of view: role A is played by b, role B is played by a. This violates again the authentication goal between B and s.

Third Version

```
A->s: A,B
s->A: {| B,KAB |}sk(A,s),
      {| A,KAB |}sk(B,s)
A->B: {| A,KAB |}sk(B,s)
```

```
GOAL: weak_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|b,KAB(1)|}_(sk(a,s)),
            {|a,KAB(1)|}_(sk(b,s))
i -> (a,1): {|b,KAB(1)|}_(sk(a,s))
```

- Third version adds the name of the other party to the encrypted message.
- From s's point of view: role A is played by a, role B by b.
- From a's point of view: role A is played by b, role B is played by a. This violates again the authentication goal between B and s.
- In many scenarios, it is a serious problem if the intruder can confuse agents about the role they play.

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

```
GOAL: strong_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|a,b,KAB(1)|}_(sk(a,s)),
            {|a,b,KAB(1)|}_(sk(b,s))
i -> (b,1): a,b,{|a,b,KAB(1)|}_(sk(b,s))
i -> (b,2): a,b,{|a,b,KAB(1)|}_(sk(b,s))
```

- Fourth version: in all encrypted messages we write both A and B–the ordering avoids the confusion.
  - ★ Alternative: have to tags `init` and `resp` to make clear which one is the initiator A and who is the responder B.

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

```
GOAL: strong_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|a,b,KAB(1)|}_(sk(a,s)),
            {|a,b,KAB(1)|}_(sk(b,s))
i -> (b,1): a,b,{|a,b,KAB(1)|}_(sk(b,s))
i -> (b,2): a,b,{|a,b,KAB(1)|}_(sk(b,s))
```

- Fourth version: in all encrypted messages we write both A and B–the ordering avoids the confusion.
  - ★ Alternative: have to tags `init` and `resp` to make clear which one is the initiator A and who is the responder B.
- In the attack, the intruder sends the last message a second time to b.
  - ★ For b, this is a completely new protocol run—note $(b, 1)$ vs. $(b, 2)$
  - ★ This is a replay attack: b is made to accept something a second time that was actually only said once by s.

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

```
GOAL: strong_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|a,b,KAB(1)|}_(sk(a,s)),
            {|a,b,KAB(1)|}_(sk(b,s))
i -> (b,1): a,b,{|a,b,KAB(1)|}_(sk(b,s))
i -> (b,2): a,b,{|a,b,KAB(1)|}_(sk(b,s))
```

- Fourth version: in all encrypted messages we write both A and B–the ordering avoids the confusion.
  - ★ Alternative: have to tags `init` and `resp` to make clear which one is the initiator A and who is the responder B.
- In the attack, the intruder sends the last message a second time to b.
  - ★ For b, this is a completely new protocol run—note $(b, 1)$ vs. $(b, 2)$
  - ★ This is a replay attack: b is made to accept something a second time that was actually only said once by s.
- Replay can often be exploited, for instance:
  - ★ a bank transfer that was ordered once is executed many times
  - ★ an agent is made to accept an old broken key

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

```
GOAL: strong_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|a,b,KAB(1)|}_(sk(a,s)),
            {|a,b,KAB(1)|}_(sk(b,s))
i -> (b,1): a,b,{|a,b,KAB(1)|}_(sk(b,s))
i -> (b,2): a,b,{|a,b,KAB(1)|}_(sk(b,s))
```

- Note `strong_auth` at `GOAL`: this appears in OFMC whenever the agreement on the names and data is correct, but something has been accepted more often than it was said (a replay attack).
- One can turn off the replay detection and just ask for the pure agreement by changing the goal to weak authentication:

```
A weakly authenticates s on B,KAB;
B weakly authenticates s on A,KAB;
```

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

- One can turn off the replay detection and just ask for the pure agreement by changing the goal to weak authentication:

  ```
  A weakly authenticates s on B,KAB;
  B weakly authenticates s on A,KAB;
  ```

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

- One can turn off the replay detection and just ask for the pure agreement by changing the goal to weak authentication:

  ```
  A weakly authenticates s on B,KAB;
  B weakly authenticates s on A,KAB;
  ```

- Then OFMC will output:

  ```
  Open-Source Fixedpoint Model-Checker version 2024
  Verified for 1 sessions
  Verified for 2 sessions
  ^C
  ```

- Here `^C` means that I pressed Control-C to stop, because it will go on forever when no attack is found, checking more and more sessions.
- For the purposes of this course it is fine to step after two sessions, and you can do this in OFMC directly with the option `--numSess 2`

Fifth Version

```
Number NA,NB;
...
B->A: NB
A->s: A,B,NA,NB
s->A: {|A,B,KAB,NA,NB|}sk(A,s),
      {|A,B,KAB,NA,NB|}sk(B,s)
A->B: {|A,B,KAB,NA,NB|}sk(B,s)
```

```
SUMMARY:
  NO_ATTACK_FOUND
```

- The best way to solve replay is to use challenge response:
  - Participants create a fresh random number like NA and NB.
  - They are included in encrypted messages to prove that the encryption is not older than the fresh numbers.

Fifth Version

```
Number NA,NB;
...
B->A: NB
A->s: A,B,NA,NB
s->A: {|A,B,KAB,NA,NB|}sk(A,s),
      {|A,B,KAB,NA,NB|}sk(B,s)
A->B: {|A,B,KAB,NA,NB|}sk(B,s)
```

```
SUMMARY:
  NO_ATTACK_FOUND
```

- The best way to solve replay is to use challenge response:
  - ★ Participants create a fresh random number like `NA` and `NB`.
  - ★ They are included in encrypted messages to prove that the encryption is not older than the fresh numbers.
  - ★ We are done. However there is a better way to do this using Diffie-Hellman.

Sixth Version

```
Protocol: KeyExchange
Types: Agent A,B,s;
       Number X,Y,g,Payload;
       Function sk;
Knowledge: A: A,B,s,sk(A,s),g;
           B: A,B,s,sk(B,s),g;
           s: A,B,s,sk(A,s),sk(B,s),g;
Actions:
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
Goals:
exp(exp(g,X),Y) secret between A,B;
Payload secret between A,B;
A authenticates B on exp(exp(g,X),Y);
B authenticates A on exp(exp(g,X),Y),Payload;
```

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

Diffie-Hellman:

- every agent generates a random X and Y
- they exchange $\exp(g, X) \mod p$ and $\exp(g, Y) \mod p$
  - ★ p is a large fixed prime number – we omit in OFMC
  - ★ g is a fixed generator of the group $\mathbb{Z}_p^\star$
  - ★ Both p and g are public
  - ★ we omit writing $\mod p$ in OFMC
- It is computationally hard to obtain X from $\exp(g, X) \mod p$
- However A and B have now a shared key $\exp(\exp(g, X), Y) \mod p = \exp(\exp(g, Y), X) \mod p$

Diffie-Hellman and ECDH

	Classic	
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	
Group Op.	${\times} : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	
Generator	$g \in \mathbb{Z}_p^\star$	
Secrets	$X, Y \in \{1, \ldots, p-1\}$	
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$	
Full key	$(g^X)^Y = (g^Y)^X$	

Diffie-Hellman and ECDH

	Classic	Elliptic Curve (ECDH)
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	Finite field $\mathbb{F}$ of order n
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	$+ : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ (not quite so intuitive...)
Generator	$g \in \mathbb{Z}_p^\star$	g on curve
Secrets	$X, Y \in \{1, \ldots, p-1\}$	$X, Y \in \{1, \ldots, n-1\}$
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$	$X \cdot g := \underbrace{g + \ldots + g}_{X \text{ times}}$ $Y \cdot g := \ldots$
Full key	$(g^X)^Y = (g^Y)^X$	$X \cdot Y \cdot g = Y \cdot X \cdot g$

Diffie-Hellman and ECDH

	Classic	Elliptic Curve (ECDH)
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	Finite field $\mathbb{F}$ of order n
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	$\times : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ (not quite so intuitive...)
Generator	$g \in \mathbb{Z}_p^\star$	g on curve
Secrets	$X, Y \in \{1, \ldots, p-1\}$	$X, Y \in \{1, \ldots, n-1\}$
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$
Full key	$(g^X)^Y = (g^Y)^X$	$(g^X)^Y = (g^Y)^X$

Trick: write $\times$ for the group operation also in ECDH.

Diffie-Hellman and ECDH

	Classic	Elliptic Curve (ECDH)
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	Finite field $\mathbb{F}$ of order n
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	$\times : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ (not quite so intuitive...)
Generator	$g \in \mathbb{Z}_p^\star$	g on curve
Secrets	$X, Y \in \{1, \ldots, p-1\}$	$X, Y \in \{1, \ldots, n-1\}$
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$
Full key	$(g^X)^Y = (g^Y)^X$	$(g^X)^Y = (g^Y)^X$
Typical size	thousand of bits	hundreds of bits

Trick: write $\times$ for the group operation also in ECDH.

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?
  - ★ Both A and B contribute something fresh to the key

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?
  - ★ Both A and B contribute something fresh to the key
  - ★ The trusted party s does not even get to know the key
    - ▶ An honest but curious s cannot read messages between A and B.

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?
  - ★ Both A and B contribute something fresh to the key
  - ★ The trusted party s does not even get to know the key
    - ▶ An honest but curious s cannot read messages between A and B.
  - ★ Perfect Forward Secrecy: The intruder cannot read `Payload` even when learning `sk(A,s)` and `sk(B,s)` after the exchange.

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?
  - ★ Both A and B contribute something fresh to the key
  - ★ The trusted party s does not even get to know the key
    - ▶ An honest but curious s cannot read messages between A and B.
  - ★ Perfect Forward Secrecy: The intruder cannot read `Payload` even when learning `sk(A,s)` and `sk(B,s)` after the exchange.
- Do we even need the trusted party s then?

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?
  - Both A and B contribute something fresh to the key
  - The trusted party s does not even get to know the key
    - An honest but curious s cannot read messages between A and B.
  - Perfect Forward Secrecy: The intruder cannot read `Payload` even when learning `sk(A,s)` and `sk(B,s)` after the exchange.
- Do we even need the trusted party s then? Yes!
  - `exp(g,X)` and `exp(g,Y)` are public
    - you may call them public keys (with X and Y the private keys)
  - but they need to be authenticated (like public keys):
    - that `exp(g,X)` really comes from A
    - and `exp(g,Y)` really comes from B

Modeling Agents and Fixed Key-Infrastructures

- Normally variables (uppercase) like A,B,C,...
  - ★ can be played by any concrete (lowercase) agent like a,b,c,...,i
- Special agent: i – the intruder
- Honest agent: constant like s for a trusted server
  - ★ Cannot be instantiated (especially the intruder), fixed in all protocol runs
- Given key infrastructures: use functions e.g.
  - ★ sk(A,B) the shared key of A and B
  - ★ pw(A,B) the password of A at server B
  - ★ pk(A) the public key of A
    - inv(K) is the private key that belongs to public key K.
    - Note inv and exp are a built-in function (do not declare as a function).
  - ★ Give every role the necessary initial knowledge

AnB: Things to Note

- Identifiers that start with uppercase: variables (E.g., `A,B,KAB`)
- Identifiers that start with lowercase: constants and functions (E.g., `s,pre,sk`)
- One should declare a type for all identifiers; OFMC can search for *type-flaw* attacks when using the option `-untyped` (in which case all types are ignored).
- The (initial) knowledge of agents MUST NOT contain variables of any type other than `Agent`.
  - ★ For long-term keys, passwords, etc. use functions like $sk(A, B)$.
- Each variable that does not occur in the initial knowledge is freshly created during the protocol by the first agent who uses it.
  - ★ In the NSSK example, `A` creates `NA`, `s` creates `KAB`, `B` creates `NB`.

Cryptographic Building Blocks (I)

- Symmetric Cryptography:
  - ★ two (or more) agents share a secret key K
  - ★ $\{\!|M|\!\}_K$ denotes symmetric encryption of message M with key K
  - ★ one can decrypt $\{\!|M|\!\}_K$ only when knowing K

Cryptographic Building Blocks (I)

- Symmetric Cryptography:
  - ★ two (or more) agents share a secret key K
  - ★ $\{|M|\}_K$ denotes symmetric encryption of message M with key K
  - ★ one can decrypt $\{|M|\}_K$ only when knowing K
- Asymmetric Encryption (Public-key Encryption):
  - ★ every agent A has a key-pair $(K, \mathrm{inv}(K))$ consisting of
    - ▶ the public key K that everybody knows
    - ▶ the private key $\mathrm{inv}(K)$ that only A knows
  - ★ $\{M\}_K$ denotes asymmetric encryption of M with public key K.
  - ★ one can decrypt $\{M\}_K$ only when knowing $\mathrm{inv}(K)$

Cryptographic Building Blocks (I)

- Symmetric Cryptography:
  - ★ two (or more) agents share a secret key K
  - ★ $\{\!|M|\!\}_K$ denotes symmetric encryption of message M with key K
  - ★ one can decrypt $\{\!|M|\!\}_K$ only when knowing K
- Asymmetric Encryption (Public-key Encryption):
  - ★ every agent A has a key-pair $(K, \mathsf{inv}(K))$ consisting of
    - ▶ the public key K that everybody knows
    - ▶ the private key $\mathsf{inv}(K)$ that only A knows
  - ★ $\{M\}_K$ denotes asymmetric encryption of M with public key K.
  - ★ one can decrypt $\{M\}_K$ only when knowing $\mathsf{inv}(K)$
- Digital Signatures
  - ★ Signing is "encryption" with a private key
  - ★ Signature checking is "decryption" with a public key
  - ★ Thus, if $(K, \mathsf{inv}(K))$ is the key pair of A then
    - ▶ $\{M\}_{\mathsf{inv}(K)}$ can only be produced by A
    - ▶ but everybody can read M and check that it comes from A.

Cryptographic Building Blocks (II)

- Nonces (From Number once): random numbers that is used only once for a challenge-response

Cryptographic Building Blocks (II)

- Nonces (From Number once): random numbers that is used only once for a challenge-response
- Hashes
  - $h(M)$ denotes the cryptographic hash of message M.
  - Hard to invert: given $h(M)$, find M.
  - Hard to find collisions: find M and M' with $h(M) = h(M')$.
  - Variant: Message Authentication Code (MAC, keyed hash) $h(K, M)$ additionally has a symmetric key K.

Cryptographic Building Blocks (II)

- Nonces (From Number once): random numbers that is used only once for a challenge-response
- Hashes
  - $h(M)$ denotes the cryptographic hash of message M.
  - Hard to invert: given $h(M)$, find M.
  - Hard to find collisions: find M and M' with $h(M) = h(M')$.
  - Variant: Message Authentication Code (MAC, keyed hash) $h(K, M)$ additionally has a symmetric key K.
- Timestamps

Cryptographic Building Blocks (II)

- Nonces (From Number once): random numbers that is used only once for a challenge-response
- Hashes
  - $h(M)$ denotes the cryptographic hash of message M.
  - Hard to invert: given $h(M)$, find M.
  - Hard to find collisions: find M and M' with $h(M) = h(M')$.
  - Variant: Message Authentication Code (MAC, keyed hash) $h(K, M)$ additionally has a symmetric key K.
- Timestamps
- Concatenation, i.e., a sequence of messages
  - written as M_1, M_2, M_3 for simplicity
  - reality: quite complex encodings and source of mistakes

    see also: Mödersheim & Katsoris. *A sound abstraction of the parsing problem*, CSF 2014.