SECURE SHELL (SSH)

Architecture & Deployment

WHAT IS SSH?

SSH is a **cryptographic network protocol** for operating network services **securely over an unsecured network**.

WHAT IS IT USED FOR?

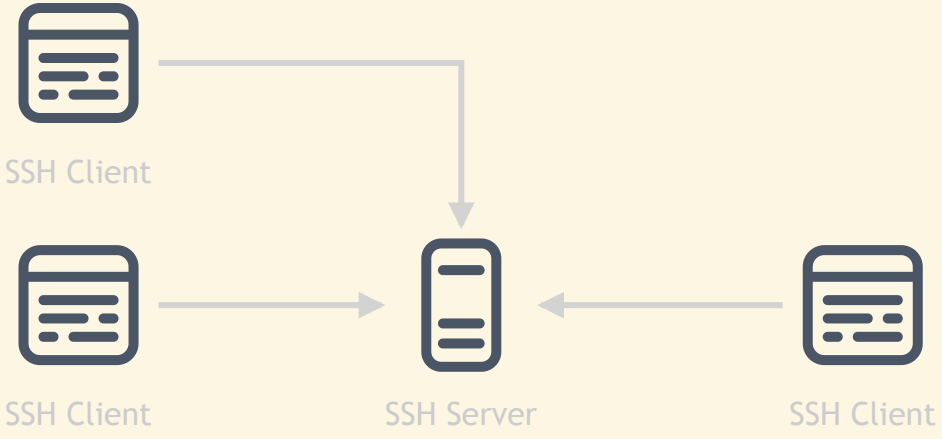
Command line login

Git

FTP

HOW DOES IT WORK?

SSH is a **client-server** protocol.



Speaker notes

Using an SSH client, a user (or application) on machine A can connect to an SSH server running on machine B, either to log in (with a command line shell) or to execute programs.

HOW IS IT SECURE?

1. SSH establishes a **secure channel**.
2. It then requires **authentication**.

Speaker notes

Note that steps 1 and 2 are **separate and unrelated processes**.

STEP 1: THE SECURE CHANNEL

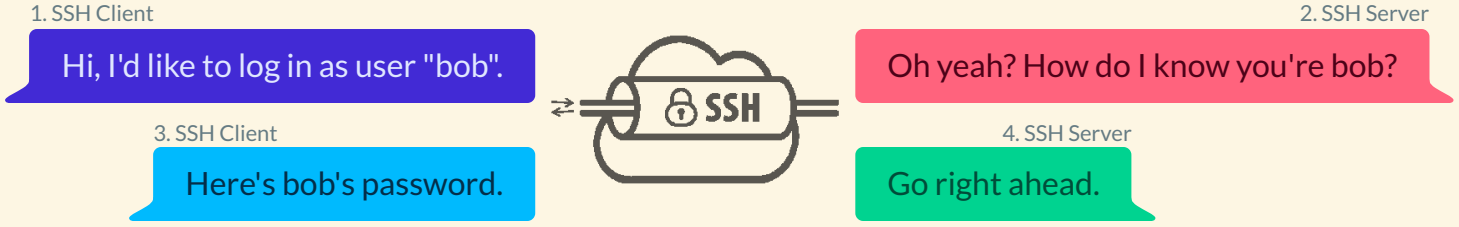
SSH Client  SSH Server

This is done for you and (mostly) automatic.

Speaker notes

SSH establishes a **secure channel** between client and server using various **cryptographic techniques**. This is handled automatically by the SSH client and server.

STEP 2: AUTHENTICATION



Speaker notes

The user or service that wants to connect to the SSH server must **authenticate** to gain access, for example with a password.

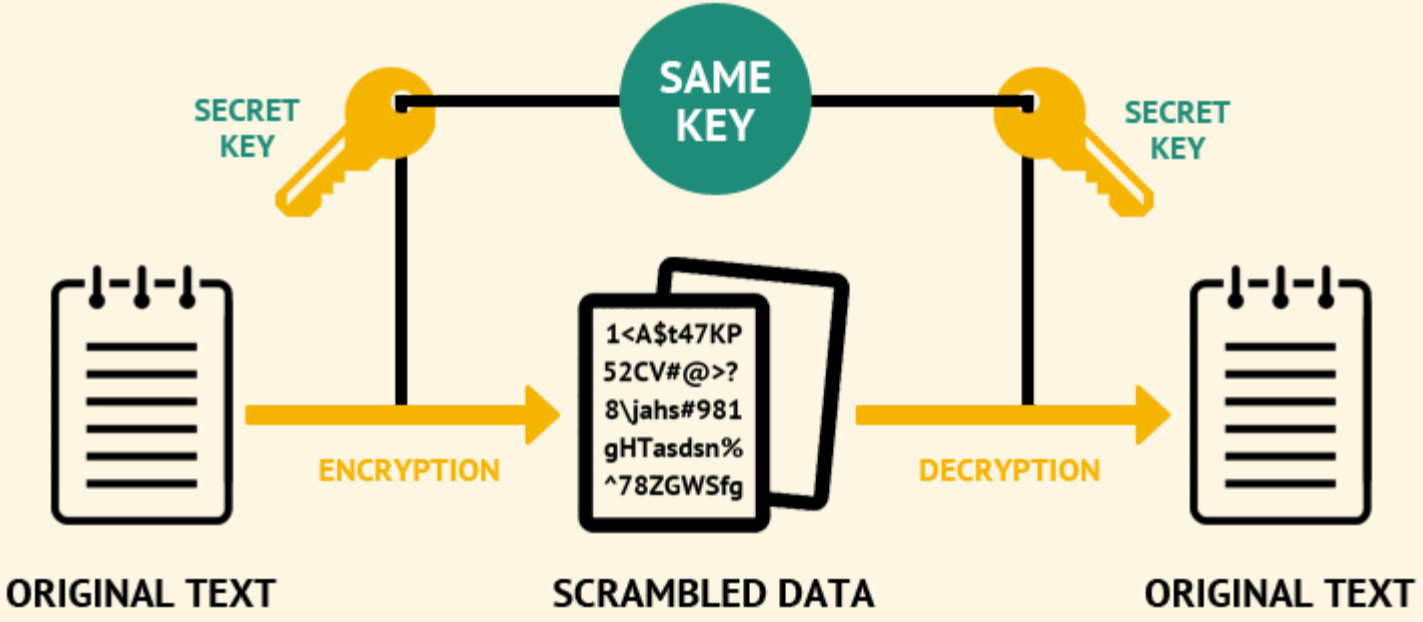
SECURITY THROUGH CRYPTOGRAPHY

- Symmetric encryption
- Asymmetric cryptography
  - Key exchange
  - Digital signatures
- Hash-based Message Authentication Codes (HMAC)

Speaker notes

SSH establishes a **secure channel** between two computers **over an insecure network** (e.g. a local network or the Internet). Establishing and using this secure channel requires a combination of various cryptographic techniques.

SYMMETRIC ENCRYPTION



Speaker notes

Symmetric-key algorithms can be used to encrypt communications between two or more parties using a **shared secret**. AES is one such algorithm.

Assuming all parties possess the secret key, they can encrypt data, send it over an insecure network, and decrypt it on the other side. An attacker who intercepts the data **cannot decrypt it without the key** (unless a weakness is found in the algorithm or its implementation).

EXAMPLE: SYMMETRIC ENCRYPTION WITH AES

```
# Create a "plaintext" file
$> cd /path/to/projects
$> mkdir aes-example
$> cd aes-example
$> echo 'too many secrets' > plaintext.txt
```

```
# Encrypt the plaintext
$> cat plaintext.txt | openssl aes-256-cbc > ciphertext.aes
enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
```

Speaker notes

Create a **plaintext** file containing the words "too many secrets".

You may encrypt that file with the OpenSSL library (installed on most computers). Executing the example command pipeline will prompt you for an encryption key.

EXAMPLE: SYMMETRIC DECRYPTION WITH AES

```
# Decrypt the ciphertext
$> cat ciphertext.aes | openssl aes-256-cbc -d
enter aes-256-cbc decryption password:
too many secrets
```

Speaker notes

The resulting **ciphertext** stored in the `ciphertext.aes` file cannot be decrypted without the key. Executing the example command pipeline and entering the same key as before when prompted will decrypt it.

The `-d` option makes the command **d**ecrypt the provided contents instead of encrypting it.

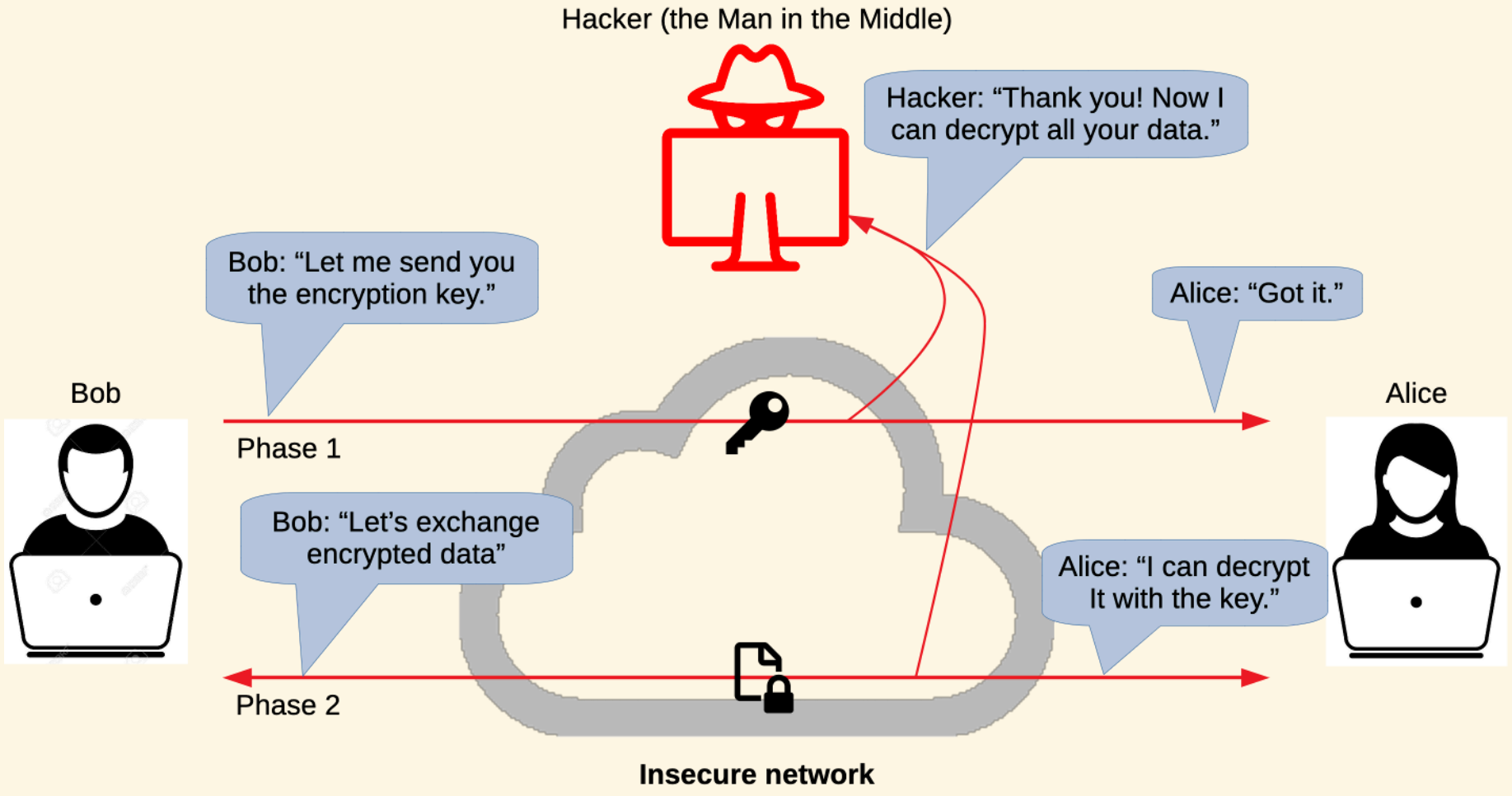
SYMMETRIC ENCRYPTION OVER AN INSECURE NETWORK

- **Both parties must have the key**
- It used to be **physically transferred**

Speaker notes

For example in the form of the codebooks used to operate the German Enigma machine during World War II. But that is **impractical for modern computer networks**.

MAN-IN-THE-MIDDLE ATTACK (MITM)

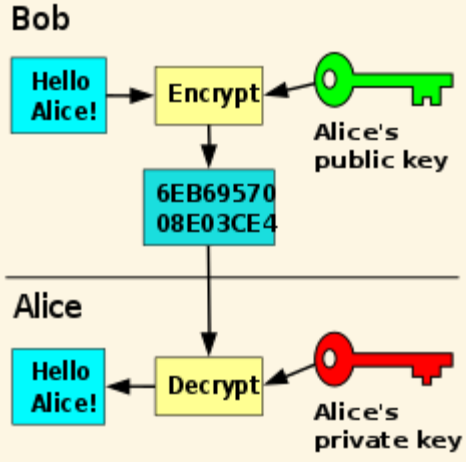


Speaker notes

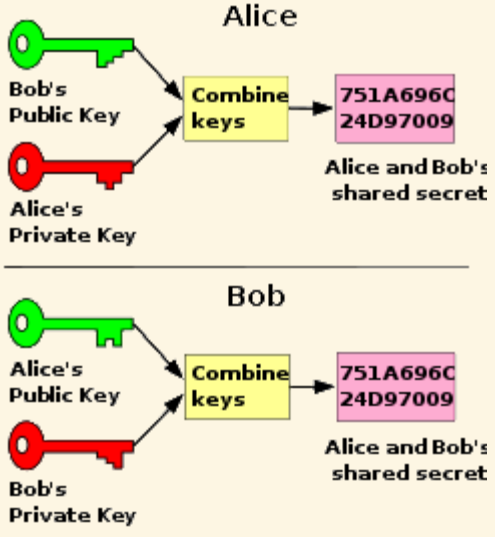
Sending the key over the insecure network risks it being compromised by a Man-in-the-Middle attack.

ASYMMETRIC CRYPTOGRAPHY

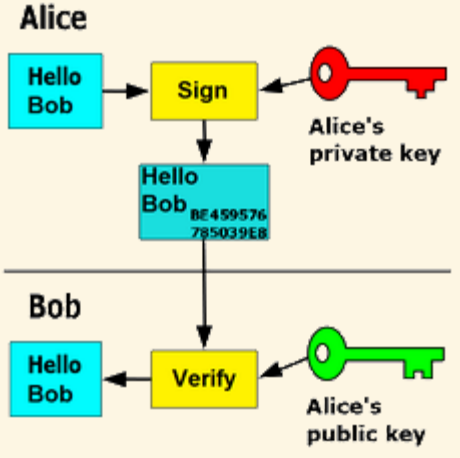
Encryption



Key exchange



Digital Signatures



Speaker notes

Public-key or asymmetric cryptography is any cryptographic system that uses pairs of keys: **public keys** which may be disseminated widely, while **private keys** which are known only to the owner. It has several use cases:

- Encrypting and decrypting data.
- Securely exchanging shared secret keys.
- Verifying identity and protecting against tampering.

THE PROPERTIES OF AN ASYMMETRIC KEY PAIR

- **Quick & easy to generate a key pair**
- **Too slow & hard to find the private key from the public key**
- The private key can solve mathematicalsproblems based on the public key, **proving ownership of that key** *(but not the other way around)*

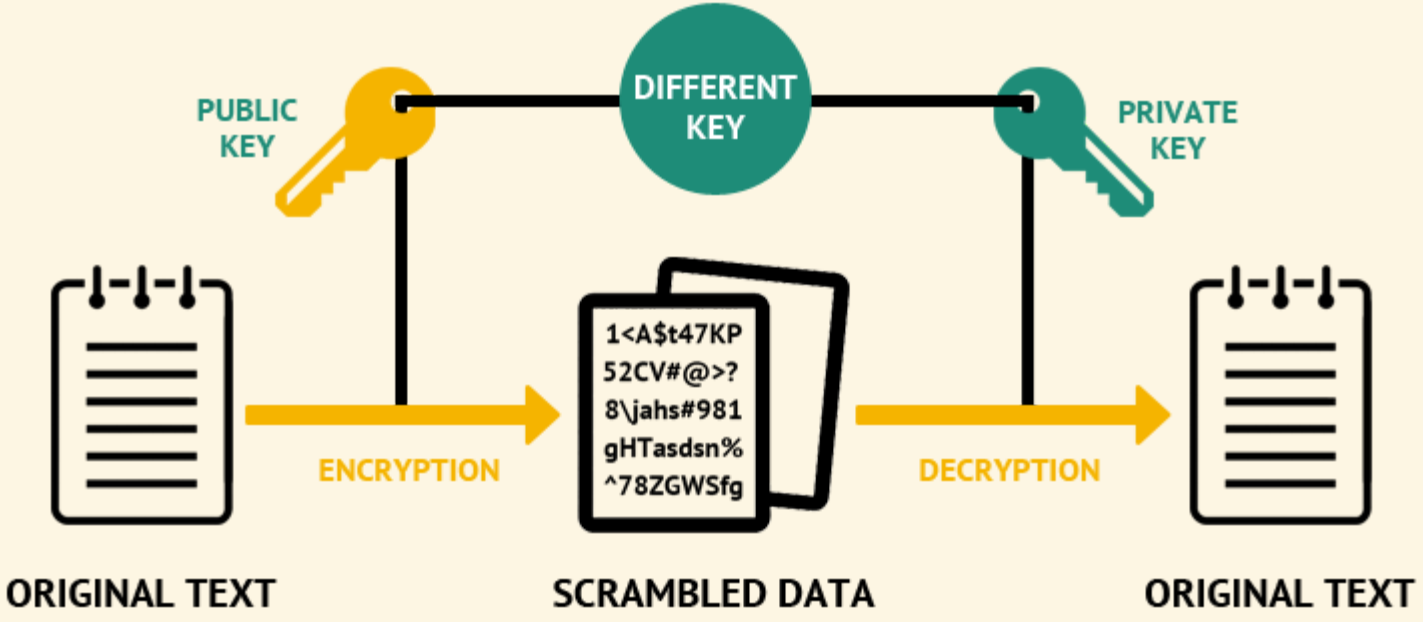
Speaker notes

There is a mathematical relationship between a public and private key, based on problems that currently admit no efficient solution such as integer factorization, discrete logarithm and elliptic curve relationships.

Here's a mathematical example based on integer factorization, a problem that is computationally economical in one direction (multiplication) but very computationally expensive in the other (factorization).

Effective security only requires keeping the private key private; **the public key can be openly distributed without compromising security**.

ASYMMETRIC ENCRYPTION



Speaker notes

One use case of asymmetric cryptography is **asymmetric encryption**, where the **sender encrypts a message with the recipient's public key**. The message can only be **decrypted by the recipient using the matching private key**.

EXAMPLE: GENERATE AN ASYMMETRIC RSA KEY PAIR

```
$> cd /path/to/projects
$> mkdir rsa-example
$> cd rsa-example

# Generate a private key
$> openssl genrsa -out private.pem 2048
Generating RSA private key, 2048 bit long modulus
.............++++++
.................................++++++
e is 65537 (0x10001)

# Generate public key from the private key (quick & easy)
$> openssl rsa -in private.pem \
   -out public.pem -outform PEM -pubout
writing RSA key
```

Speaker notes

Let's try encryption with RSA this time, an asymmetric algorithm. To do that, we need to generate a **key pair, i.e. a private and public key**. The example commands will generate first a private key in a file named `private.pem`, then a corresponding public key in a file named `public.pem`.

By convention, we use the `.pem` extension after the Privacy-Enhanced Mail (PEM) format, a de facto standard format to store cryptographic data.

EXAMPLE: ASYMMETRIC ENCRYPTION WITH RSA

```
# Create a plaintext
$> echo 'too many secrets' > plaintext.txt

# Encrypt the plaintext with the public key
$> openssl pkeyutl -encrypt -in plaintext.txt \
   -inkey public.pem -pubin -out ciphertext.rsa

# See what's there
$> ls
ciphertext.rsa plaintext.txt private.pem public.pem
```

Speaker notes

You can create a plain text and **encrypt it with the public key** using the OpenSSL library.

The example command will read the plaintext file `plaintext.txt` specified with the `-in` (**in**put) option. It will also read the public key in the `public.pem` file with the `-inkey` (**in**put **key**) and `-pubin` (**pub**lic **in**) options.

It will then write the encrypted ciphertext to the `ciphertext.rsa` file with the `-out` (**out**put) option.

In addition to your key pair, you should have two additional files containing the plaintext and ciphertext:

EXAMPLE: ASYMMETRIC DECRYPTION WITH RSA

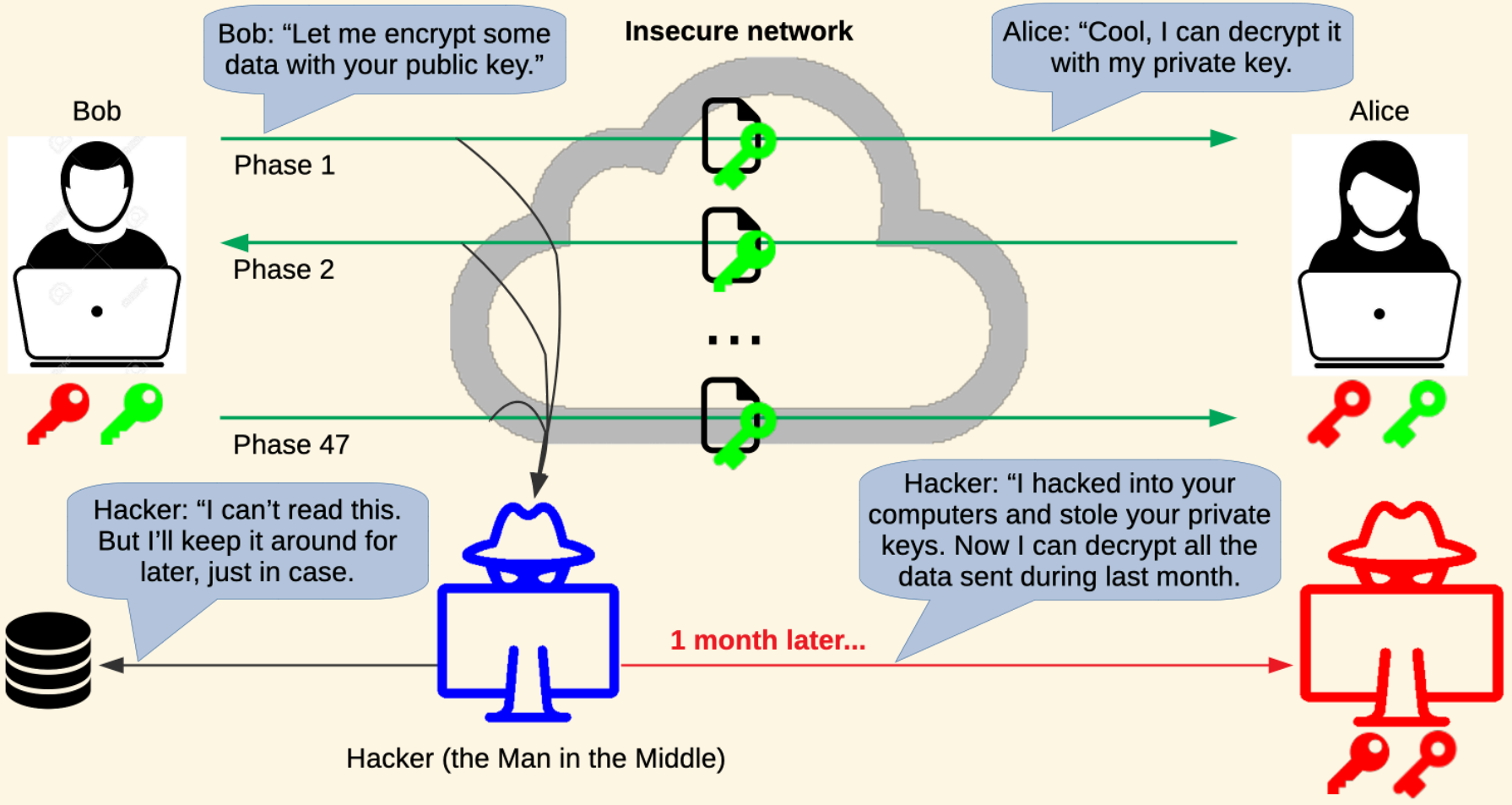
```
# Decrypt the ciphertext with the private key
$> openssl pkeyutl -decrypt \
   -inkey private.pem -in ciphertext.rsa
too many secrets

# It does not work with the public key
$> openssl pkeyutl -decrypt \
  -inkey public.pem -in ciphertext.rsa
unable to load Private Key [...]

# It does not work either with another private key
$> openssl genrsa -out hacker-private.pem 1024
$> openssl pkeyutl -decrypt \
   -inkey hacker-private.pem -in ciphertext.rsa
RSA operation error [...]
```

Speaker notes

The ciphertext can be **decrypted with the corresponding private key**. Note that you **cannot decrypt the ciphertext using the public key**. Of course, a hacker using **another private key cannot decrypt it either**.

Hence, you can encrypt data and send it to another party provided that you have their public key. **No single shared key needs to be exchanged** (the private key remains a secret known only to the recipient).

ASYMMETRIC ENCRYPTION AND FORWARD SECRECY



Speaker notes

Asymmetric encryption protects data sent over an insecure network from attackers, but **only as long as the private keys remain private**. It does not provide **forward secrecy**, meaning that if the private keys are compromised in the future, all data encrypted in the past is also compromised.

SYMMETRIC VS. ASYMMETRIC ENCRYPTION

	Pros	Cons
Symmetric encryption	**Fast**, can be implemented in **hardware**	Must send key, no forward secrecy
Asymmetric encryption	**No shared key**	Slow, no forward secrecy

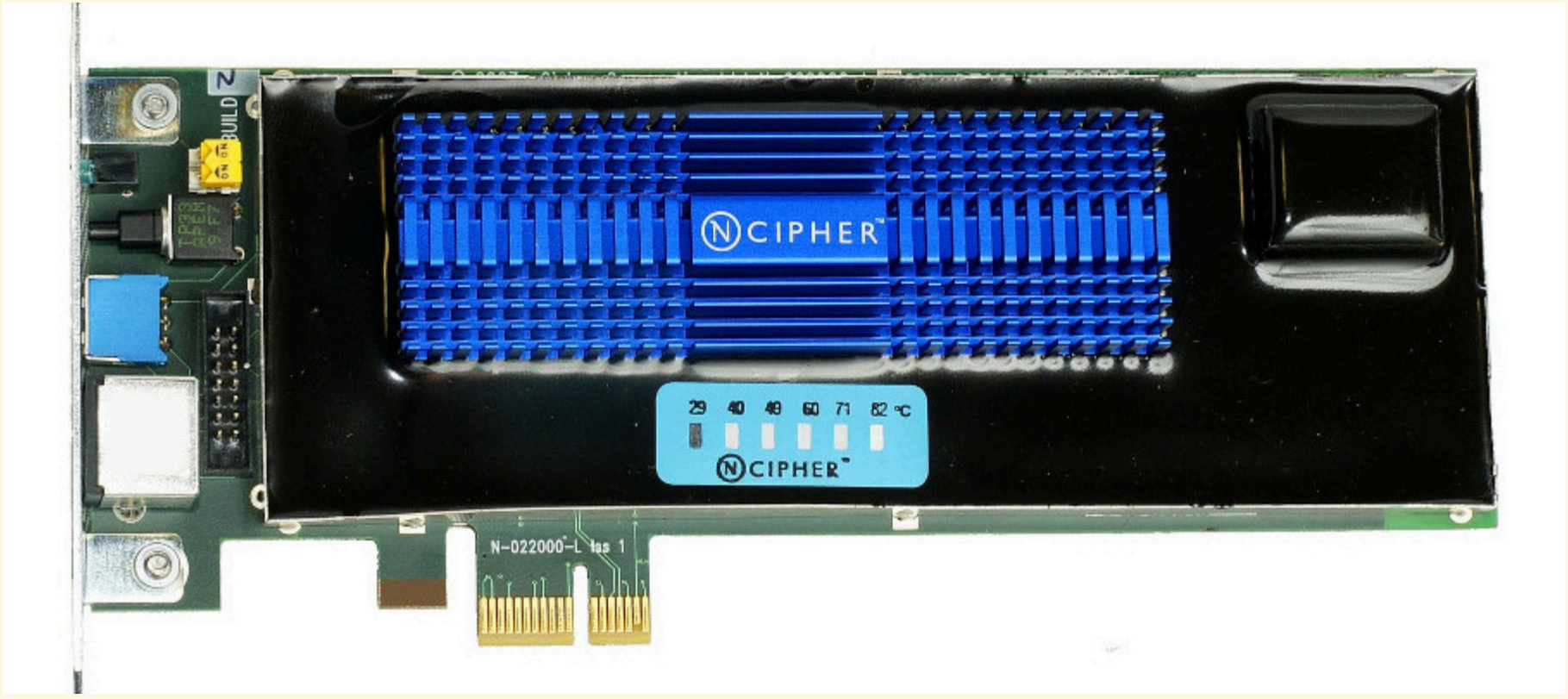
Speaker notes

So far we learned that:

- Symmetric encryption works but provides no solution to the problem of securely transmitting the shared secret key.
- Asymmetric encryption works even better as it does not require a shared secret key, but it does not provide forward secrecy.

Additionally, it's important to note that **symmetric encryption is much faster than asymmetric encryption**.

SYMMETRIC ENCRYPTION IN HARDWARE



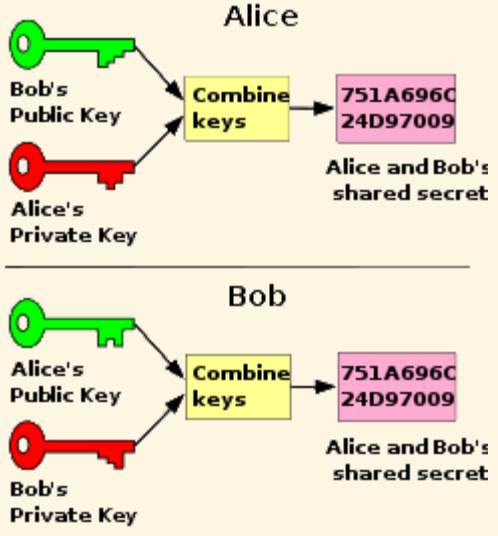
Speaker notes

Symmetric encryption is also less complex and can easily be implemented as hardware (most modern processors support hardware-accelerated AES encryption).

This is a hardware security module, a physical computing device that safeguards and manages secrets, performs encryption and decryption functions for digital signatures, strong authentication and other cryptographic functions

WHAT CAN WE DO?

It would be nice if we could share a **fast symmetric encryption key**... without actually sharing it.

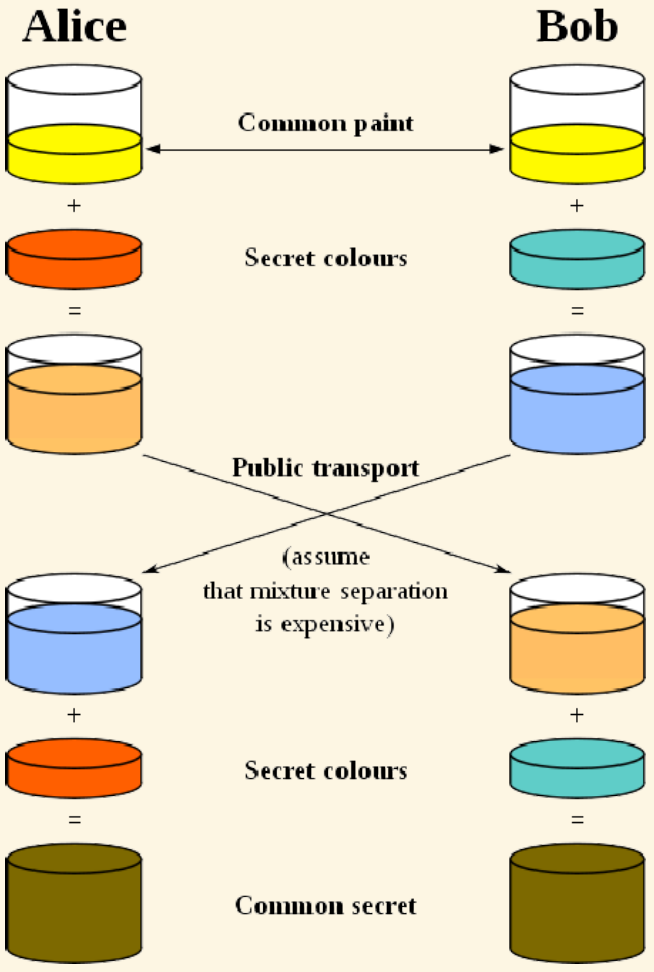


Speaker notes

Ideally, we would want to be able to share a fast symmetric encryption key without transmitting it physically or over the network. This is where asymmetric cryptography comes to the rescue again. Encryption is not all it can do; it can also do **key exchange**.

The Diffie-Hellman Key Exchange, invented in 1976 by Whitfield Diffie and Martin Hellman, was one of the first public key exchange protocols allowing users to **securely exchange secret keys** even if an attacker is monitoring the communication channel.

DIFFIE-HELLMAN KEY EXCHANGE



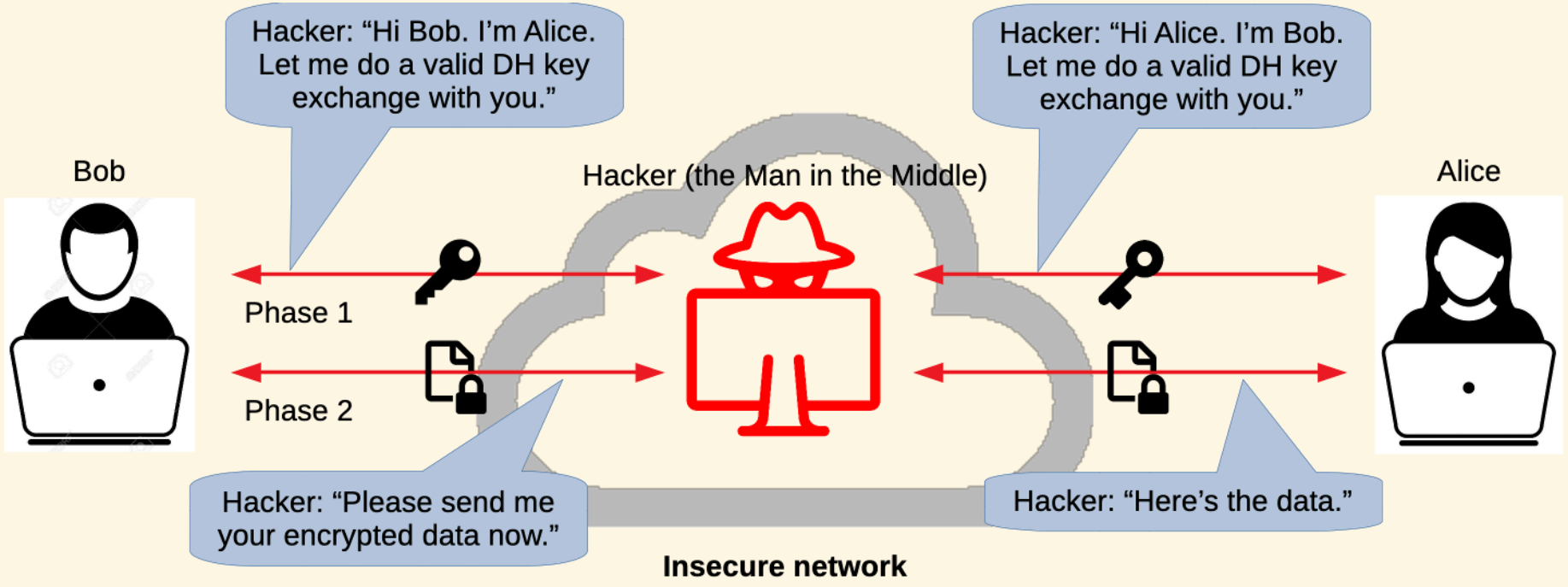
Speaker notes

This conceptual diagram illustrates the general idea behind the protocol:

- Alice and Bob choose a **random, public starting color** (yellow) together.
- Then they each separately choose a **secret color known only to themselves** (orange and blue-green).
- Then they **mix their own secret color with the mutually shared color** (resulting in orange-tan and light-blue) and **publicly exchange** the two mixed colors.
- Finally, Alice and Bob **mix the color he or she received** from each other **with his or her own private color** (yellow-brown).

The result is a final color mixture that is **identical to the partner's final color mixture**, and which was never shared publicly. When using large numbers rather than colors, it would be computationally difficult for a third party to determine the secret numbers.

MAN-IN-THE-MIDDLE ATTACK ON DIFFIE-HELLMAN

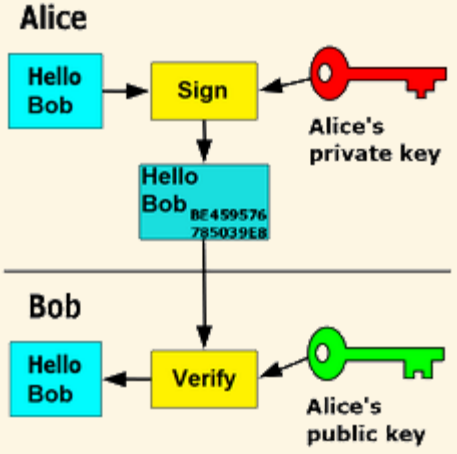


Speaker notes

The Diffie-Hellman key exchange solves the problem of transmitting the shared secret key over the network by computing it using asymmetric cryptography. It is therefore never transmitted.

However, **a Man-in-the-Middle attack is still possible** if the attacker can position himself between the two parties to **intercept and relay all communications**.

ASYMMETRIC DIGITAL SIGNATURE



Speaker notes

One of the other main uses of asymmetric cryptography is performing **digital signatures**. A signature proves that the message came from a particular sender.

- Assuming **Alice wants to send a message to Bob**, she can **use her private key to create a digital signature based on the message**, and send both the message and the signature to Bob.
- Anyone with **Alice's public key can prove that Alice sent that message** (only the corresponding private key could have generated a valid signature for that message).
- **The message cannot be tampered with without detection**, as the digital signature will no longer be valid (since it based on both the private key and the message).

Note that a digital signature **does not provide confidentiality**. Although the message is protected from tampering, it is **not encrypted**.

EXAMPLE: DIGITAL SIGNATURE WITH RSA

```
# Create a message file
$> echo "Hello Bob, I like you" > message.txt

# Create a digital signature for
# that message with the private key
$> openssl dgst -sha256 -sign private.pem \
   -out signature.rsa message.txt

# See the signature (base64-encoded)
$> openssl base64 -in signature.rsa
```

Speaker notes

In the same directory as the previous example (asymmetric encryption with RSA), create a `message.txt` file with the message that we want to digitally sign.

The example OpenSSL command will use the private key file `private.pem` (from the previous example) and generate a digital signature based on the message file `message.txt`. The signature will be stored in the file `signature.rsa`.

If you open the file, you can see that it's simply binary data. You can see it base64-encoded with the second example command.

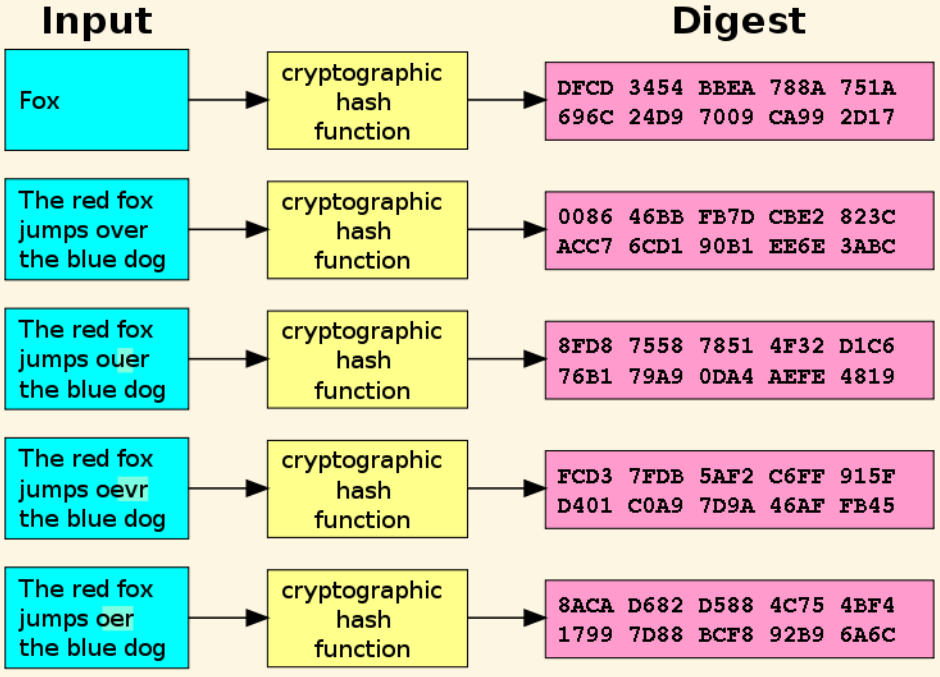
EXAMPLE: VERIFYING A DIGITAL SIGNATURE WITH RSA

```
$> openssl dgst -sha256 -verify public.pem \
   -signature signature.rsa message.txt
Verified OK

# Modify the message...

$> openssl dgst -sha256 -verify public.pem \
   -signature signature.rsa message.txt
Verification Failure
```

Speaker notes

The example command uses the public key to check that the signature is valid for the message. If you modify the message file and run the command again, it will detect that the digital signature no longer matches the message:

CRYPTOGRAPHIC HASH FUNCTIONS & MACS



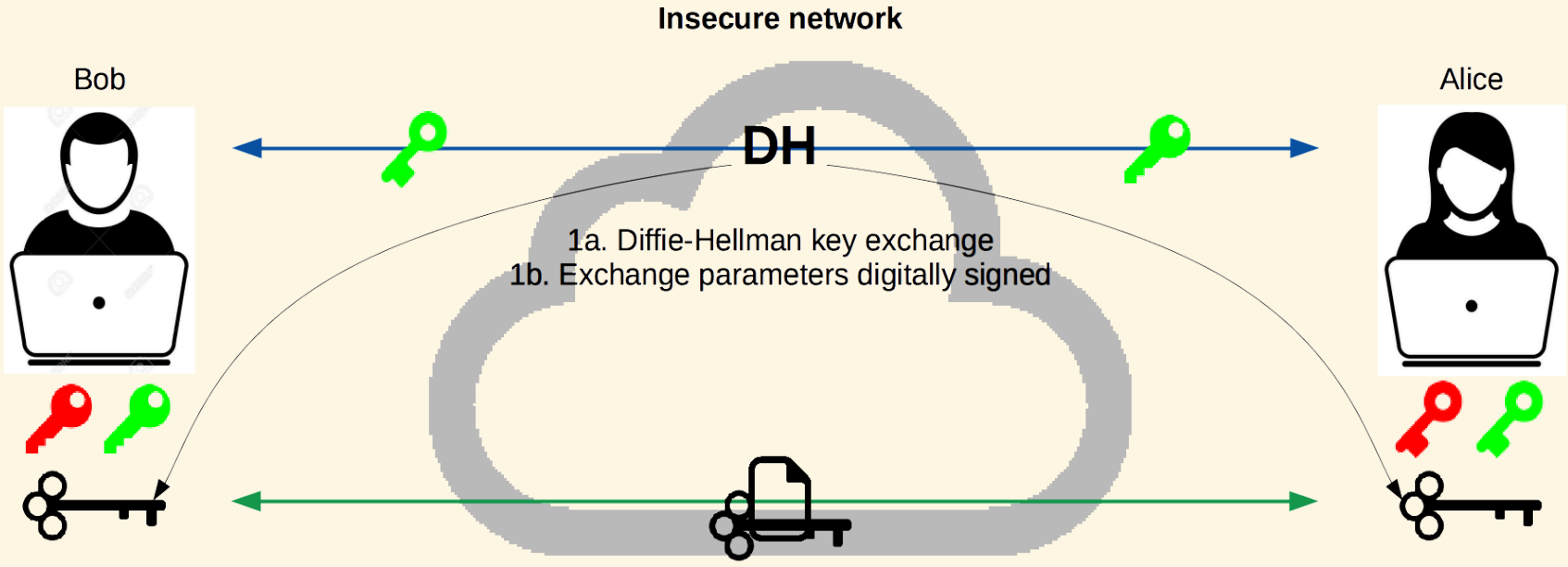
Speaker notes

A cryptographic hash function is a hash function that has the following properties:

- The same message always results in the same hash (deterministic).
- Computing the hash value of any message is quick.
- It is infeasible to generate a message from its hash value except by trying all possible messages (one-way).
- A small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value.
- It is infeasible to find two different messages with the same hash value (collisions).

SSH uses Message Authentication Codes (MAC), which are based on cryptographic hash functions, to protect both the data integrity and authenticity of all messages sent through the secure channel.

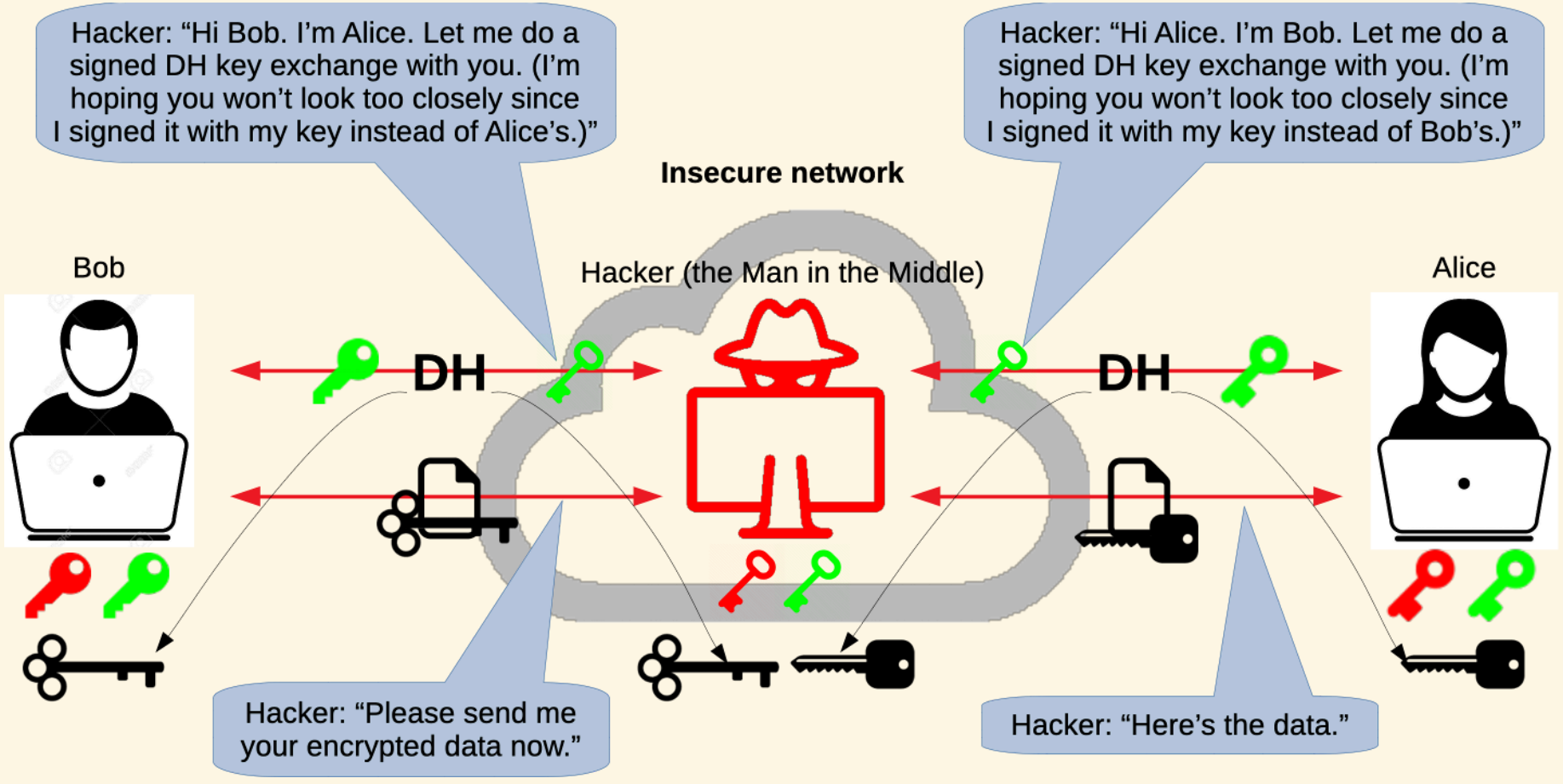
COMBINING IT ALL TOGETHER IN SSH



Speaker notes

SSH uses most of the previous cryptographic techniques we've seen together to achieve as secure a channel as possible.

MAN-IN-THE-MIDDLE ATTACK ON SSH



THREATS COUNTERED

- Eavesdropping
- Connection hijacking
- DNS an IP spoofing
- Man-in-the-Middle attack

*As long as you **check the public key**!*

Speaker notes

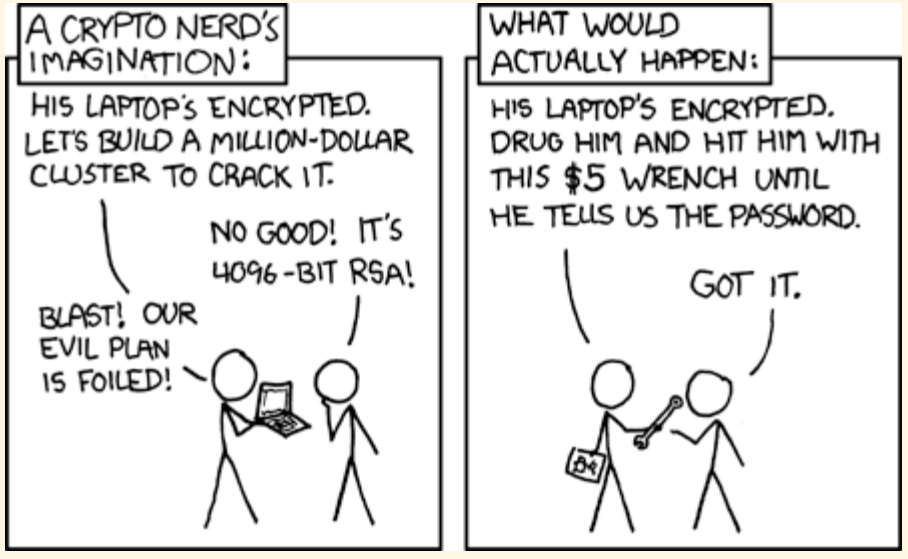
SSH counters the following threats:

- **Eavesdropping:** an attacker can intercept but not decrypt communications going through SSH's secure channel.
- **Connection hijacking:** an active attacker can hijack TCP connections due to a weakness in TCP. SSH's integrity checking detects this and shuts down the connection without using the corrupted data.
- **DNS and IP spoofing:** an attacker may hack your naming service to direct you to the wrong machine.
- **Man-in-the-Middle attack:** an attacker may intercept all traffic between you and the real target machine.

The last two are countered by the asymmetric digital signature performed by the server on the DH key exchange, **as long as the client actually checks the server-supplied public key**. Otherwise, there is no guarantee that the server is genuine.

THREATS NOT COUNTERED

- Password cracking *(common passwords: 123456, password, qwerty1)*
- IP/TCP denial of service
- Traffic analysis
- Carelessness and coffee spills ☕
- Genius mathematicians *(did you see Sneakers?)*



Speaker notes

SSH does not counter the following threats:

- **Password cracking:** if password authentication is enabled, a weak password might be easily brute-forced or obtained through side-channel attacks. Consider using public key authentication instead to mitigate some of these risks.
- **IP/TCP denial of service:** since SSH operates on top of TCP, it is vulnerable to attacks against weaknesses in TCP and IP, such as SYN flood.
- **Traffic analysis:** although the encrypted traffic cannot be read, an attacker can still glean a great deal of information by simply analyzing the amount of data, the source and target addresses, and the timing.
- **Carelessness and coffee spills:** SSH doesn't protect you if you write your password on a post-it note and paste it on your computer screen.
- **Genius mathematicians:** did you see Sneakers?