3

Encryption and Web Server Configuration

3.1. Examples of different web servers

The nature of web software programs, which send pages to the user's browser (also known as web servers), has changed dramatically in recent years.

The 2000s were dominated by competition between Apache software [APA 16] and Microsoft-based servers, Internet Information Server (IIS) [MIC 16].

Although Apache servers are now the most commonly used type of server, a newcomer, NGINX [NGI 16], has recently begun to gain in popularity, renowned for its performance. IIS has largely disappeared and is seldom used, except for very specific platforms.

Each of these servers is configured using different principles. Whereas Apache governs the behavior either from a global configuration file or from *.htaccess* files placed directly within the file system, NgInx works differently, with one single file for each application (website name), usually saved in the */etc/nginx/sites-enabled* folder. This makes code review easier: you do not need to browse the entire tree to determine the system configuration, but it also has disadvantages, especially in the context of hosting platforms. Specifying the desired settings is not always possible, since access to the file might be prohibited.

Each of these web servers can adopt the same configuration, even though the actual commands for doing so can differ. There would be little interest in giving one presentation for each type of server; we will only give examples for Apache, which is the most common, used by almost 50% of all active websites [NET 16].

3.2. Introduction to concepts in encryption

Encryption is the process of modifying a message so that it becomes incomprehensible to anybody who does not know the key or decryption method. We distinguish two main types of encryption: symmetric encryption, and its variant, hashing, and asymmetric encryption. The implementation of the latter is based on digital certificates.

3.2.1. *Symmetric encryption*

Symmetric encryption (or private-key encryption) uses the same key both to encrypt and decrypt the message.

A cipher is applied to the information being encrypted. One of the parameters of this cypher is a key known only by the sender and receiver (Figure 3.1).

Given the key and the cipher, the encrypted message can be decrypted by applying the inverse cipher associated with the key.

There are two techniques for doing this. Block ciphers divide the message into several parts of equal size (between 64 and 256 bits, depending on the algorithm). This is the most common type of encryption in computer systems. Stream ciphers encrypt the message bit by bit: this technique is mainly used for radio transmission systems (GSM – cellphone networks, Bluetooth – wireless networks, for example).

The size of the key itself is typically between 56 and 256 bits.

The strength of a symmetric cipher depends on several factors. The longer the key, the more secure the encryption. It is widely believed that a key of 256 bits (2^{256} is approximately 10^{77}, which is estimated to be close to the number

of electrons in the universe) can never be broken by brute force, i.e. by testing each combination in turn. However, the length of the key is not the only factor that determines the strength of the cipher. Messages are encrypted block by block, and the larger each block, the more robust the cipher. The same computation function is also applied multiple times (number of iterations). The greater the number of iterations, the more robust the cipher; ANSSI recommends performing 65,000 iterations. The relevancy of the algorithm itself must also be considered, and the key must be generated completely at random.

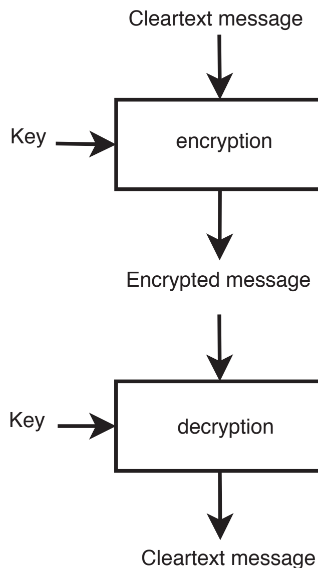

Figure 3.1. *Principle of symmetric encryption*

To improve security, especially for codes or passwords, limiting the number of permitted attempts is also a good idea. For example, smart cards are blocked after three unsuccessful attempts, which means that unlock codes with only 4–6 digits are sufficient.

Today, the most widely used algorithm is *AES256*: the blocks are 128 bits in size, and the key is 256 bits. It is currently believed to be secure.

Symmetric encryption is inexpensive in terms of computation time, due to the simplicity of its algorithms (matrix permutations and boolean XOR-type

functions are applied to the data). However, they have a disadvantage: the sender and the receiver must first exchange the secret key. Over an Internet connection, confirming the identities of the sender and the receiver is problematic, and the key must be transmitted in such a way that nobody else can see it. We will see below that asymmetric encryption provides a solution to this problem.

3.2.2. *Computing hashes and salting passwords*

In computing, a hash is a fixed-length sequence of characters calculated from a file or an arbitrary sequence of characters. The hash is unique: it is impossible to obtain the same hash from different data. But if the algorithm is not sufficiently secure or the number of possible combinations is too small, there can be collisions, i.e. two different strings can lead to the same hash[1]. Finally, it should not be possible to reconstruct the original information from the hash.

There are two main situations in which hashes are useful. The first is when we wish to verify that a copy of a file is identical to the original, for example when downloading an ISO image. The website hosting the download indicates the hash value, and specifies the method used to calculate it. Once the file has been downloaded, it is easy to recalculate the hash and check that both values are identical. If there is a difference, the downloaded file is not identical to the original, either due to an error during transmission or interference from a hacker, which typically takes the form of a *man-in-the-middle* attack [WIK 15a]. In this type of attack, the attackers position themselves between the client's computer and the web server, and rewrite the transmitted information in real time.

Hashes are also used to encode passwords in such a way that they cannot be decoded.

A special procedure can be used to store passwords. When the password is created, its hash is calculated, and the hash is stored in the database. To check the password, the program calculates the hash of the string entered by the user, and then compares it to the value stored in the database. If the two hashes are identical, the password is accepted as correct (Figure 3.2).

1 Today, the *md5* algorithm is no longer used alone, primarily because of this weakness.

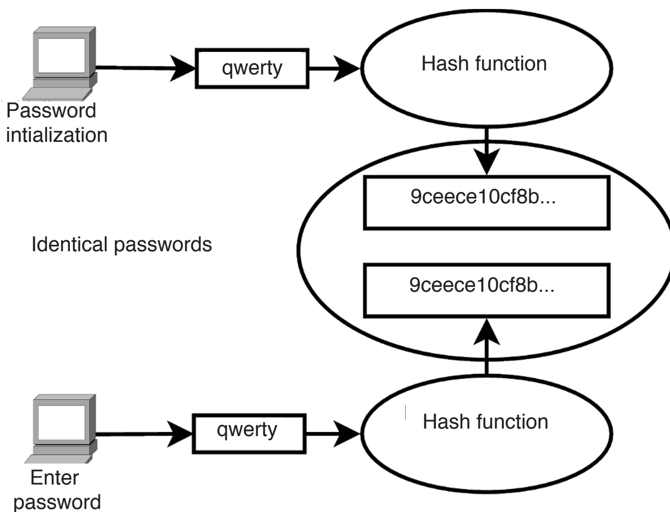

Figure 3.2. *Password verification using hashes*

Today, we extend this procedure with a technique known as salting. The hash is calculated from the data given to the hash algorithm. But if these data are predictable (password too easy to guess, for example), it can be relatively easy to recover the original data from the hash.

This can happen in practice for passwords. Too many people choose passwords that are easy to guess: the strings *password* or *12345678*, first names or the date of birth of a child or spouse, etc., are unfortunately very common choices. If an attacker knows the hashing algorithm and has access to the database following an intrusion or data theft, they will be able to run a search that will easily find some of these passwords.

To protect against the risk of this type of attack, one solution is to mix in a piece of variable information when the hash is calculated. This variable information is different for each user. This is called salting. Usually, the account or login id, which is necessarily unique, is appended to the password: even if two users have the same password, this procedure results in different hashes. Below is an example with the password *password* and the two distinct user accounts *john* and *mark* (the code was generated using a Linux command):

```
echo johnpassword|sha256sum
88071bcc...
```

We joined the username (*john*) and the password (*password*) together before computing the hash. We now do the same with a different username, *mark*:

```
echo markpassword|sha256sum
bd6e27fb08...
```

The two hashes are different.

Therefore, even though the passwords themselves might be the same, the values stored in the database are never identical. Even if the attacker knows the salting algorithm, they will be forced to recalculate all possible values for each account, which makes their task much more complex.

3.2.3. *Asymmetric encryption*

Symmetric encryption is secure enough to protect communications, but it suffers from a fundamental flaw: the encryption key must be shared between both parties. Thus, we need a way to exchange the key without it being intercepted, while verifying the identity of the person with whom we are communicating.

Asymmetric encryption provides a solution to this problem.

Asymmetric protocols generate two keys instead of one, based on two randomly chosen prime numbers. The remarkable property of this procedure is that a message encrypted with one key can only be decrypted using the other:

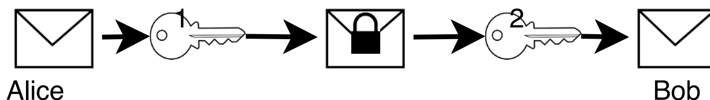

Figure 3.3. *Principle of asymmetric encryption*

The message encrypted with key 1 can only be decrypted with key 2. The reverse is also true: the message encrypted with key 2 can only be decrypted with key 1.

In practice, the first key is kept secret by its owner: it is referred to as the *private key*. The second key, the *public key*, is transmitted to all recipients that request it.

This mechanism provides an easy way of accomplishing two different tasks: encrypting messages and verifying the identity of a communication partner.

If Bob wants to send an encrypted message to Alice, he retrieves her public key, and uses it to encrypt his message. Alice can then decrypt the message using her private key: she is the only one able to do so, as she is the only one who knows the private key.

Now, if Alice sends a message to Bob, and Bob wants to be certain that it was definitely Alice who sent it, the procedure is a little more complicated (Figure 3.4).

The following sequence of operations is performed:

– Alice calculates the hash of her message using a hash function as discussed above;

– she encrypts the hash using her private key;

– she sends a message with the encrypted hash to Bob;

– Bob receives this message, and calculates its hash;

– he decrypts the encrypted hash sent by Alice using her public key;

– finally, he compares both hashes: if they are identical, then it must have been Alice who sent the message.

Of course, this protocol is carried out automatically, and these calculations are performed by software programs, such as email clients like Thunderbird [MOZ 15].

Asymmetric encryption is relatively robust because it is currently not possible to quickly factor the product of two prime numbers if they are

chosen to be sufficiently large (there are other algorithms for managing asymmetric keys based on elliptic curves rather than prime numbers; these algorithms do not require the keys to be so large).

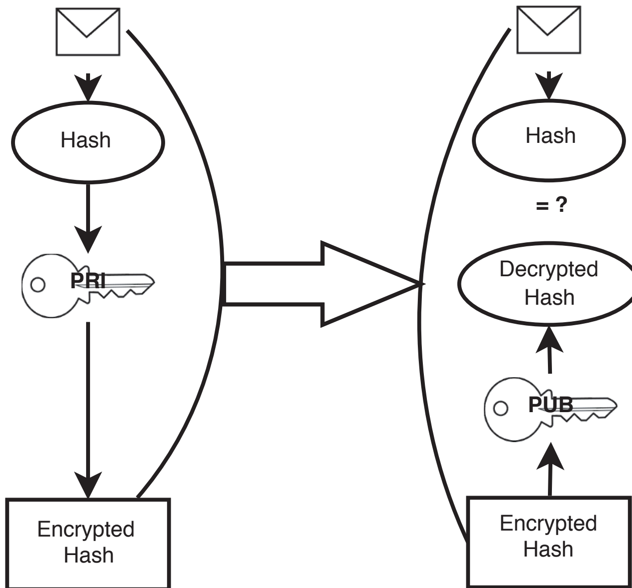

Figure 3.4. *Principle of asymmetric encryption-based signatures*

As it currently stands, the prime numbers used to generate the keys must have sizes of at least 2048 bits to guarantee that they are robust. Even today, ANSSI recommends using keys with 3096 bits, especially if they are intended to remain in usage until after 2030.

3.2.4. *What is the ideal length for encryption keys?*

The figures listed here are taken from a document published by ANSII [ANS 14].

Here are some examples that give an idea of the orders of magnitude involved.

2^n	10^n	Order of magnitude
2^{32}	4,294,967,296	Number of people on Earth
2^{46}	$7.036874418 \times 10^{13}$	Sun–Earth distance, in millimeters
2^{55}	$3.602879702 \times 10^{16}$	Number of operations performed in 1 year at a rate of one billion per second (1 Ghz)
2^{90}	$1.237940039 \times 10^{27}$	Number of operations performed in 15 billion years at a rate of one billion per second
2^{256}	$1.157920892 \times 10^{77}$	Estimated number of electrons in the universe

Table 3.1. *Some examples of orders of magnitude*

For secret-key encryption (symmetric keys), the minimum block size must be at least 64 bits (128 bits after 2020). The length of the encryption key must be at least 128 bits. It is believed that a key of length 256 bits will never be able to be broken by brute force.

The decryption rules for asymmetric encryption are completely different: factoring procedures are used, which are easier to compute. The minimum size of the prime moduli, i.e. the moduli used to generate the keys, should not be less than 2048 bits (3072 bits for certificates intended to remain in usage after 2030).

3.2.5. *Digital certificates and the chain of certification*

One of the problems with using asymmetric encryption lies in the fact that it is difficult to be sure that the public key, provided by Alice, is indeed hers and was not replaced by an attacker.

One solution is to trust a certification authority that guarantees the validity of the public key.

The certification authority signs Alice's public key using the following procedure:

– when Alice generates her two keys, she sends her public key to the certification authority;

– the certification authority verifies Alice's identity, for example by checking her identification documents, and then signs Alice's public key by encrypting its hash with the certification authority's own private key. The public key and the encrypted hash are stored in a *digital certificate*.

To verify that the digital certificate (and hence the public key) indeed belongs to Alice, her communication partner can calculate the hash of Alice's public key and then decrypt the encrypted hash using the public key of the certification authority. If these two hashes are identical, the user can trust that this key belongs to Alice, so long as they trust the certification authority. This is exactly the same procedure as for signing messages, as discussed just above (see section 3.2.3).

The public keys of the certification authorities are built directly into computer systems, which means they can be trusted without necessarily knowing all of them. Software publishers such as the Mozilla foundation for the Firefox browser are responsible for including integrated certificates issued by certification authorities. These public keys are technically implemented by placing them in a self-signed certificate, i.e. a certificate signed directly by the certification authority that created the key.

In most cases, the certification authorities do not use their private key to generate the certificates of their clients, but instead create an intermediate key that is then used to produce the required certificates. This leads to a chain of certification (Figure 3.5).

In this example, the server certificate is generated from a level 3 certificate. To allow the browser to validate the certificate submitted by the server, the server produces three certificates on request, which makes it possible to unravel the chain up to a known certification authority (the root certificate).

Certificates allow persons or devices to be reliably identified. They can therefore be used to authenticate users in applications, sign messages or documents and encrypt messages or communications to prevent them from being read by unauthorized parties. This is the procedure used for web applications, in the form of the https protocol (see section 3.4).

3.3. Generating and managing encryption certificates

3.3.1. *The OpenSSL library*

OpenSSL is a library of tools for managing TLS encryption. Distributed under the Apache license, it is available for almost every operating system, and is the most frequently used library in practical contexts.

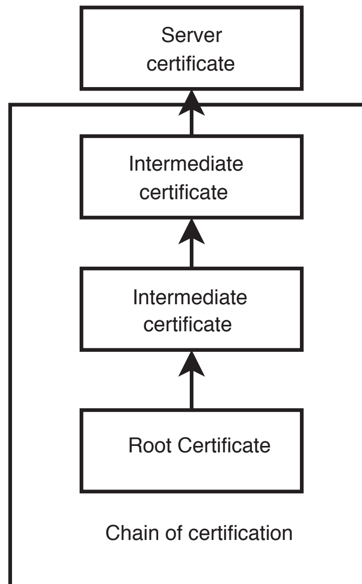

Figure 3.5. *The chain of certification*

3.3.2. *Different types of certificates*

Public keys, private keys and certificates are provided in various formats. The most common are listed below:

– DER: Used to encode X509 certificates written in ASN.1. Standard filename extensions: .der, .cer, .crt, .cert;

– PEM: Privacy Enhanced Mail. This is base64-encoded DER with added ASCII headers. Standard filename extensions: .pem, .cer, .crt, .cert;

– PKCS#12: *Personal Information Exchange Syntax Standard*. This is a standard for storing private keys, public keys and certificates, usually in password-protected form. The data are stored as binary. Filename extension: .p12, .pfx (for Microsoft).

Personal certificates are stored in the .p12 format. They contain both the private key and the public key, and so should always be password-protected.

3.3.3. *Generating certificates*

A certificate is a public key that has been signed by a certification authority. To create a certificate, the best solution is to submit a request to a recognized authority, but we can also self-sign our certificates by creating our own authority. In this case, browsers will not recognize the certificate as secure.

In our discussion, we will begin by signing our own public keys. This is our only option if we do not have access to a certification authority.

3.3.3.1. *Creating the root certificate*

We begin by creating the private key of the authority corresponding to the root certificate.

```
openssl req -new -x509 -keyout cacert.pem -out
   cacert.pem -days 3650
```

This command generates a *private key - public key* pair. When this program is executed, it will request a password: this is the password used to protect the private key, which will need to be given each time that a certificate is signed. Here, the root certificate is valid for 10 years (3650 days).

The *cacert.pem* file contains both the private key and the public key.

We will now generate the certificate belonging to the certification authority, which will be used to validate the other certificates that we will generate subsequently; this certificate therefore contains the public key of the certification authority:

```
openssl x509 -in cacert.pem -out cacert.crt
```

The root certificate is the file *cacert.crt*.

3.3.3.2. *Creating the private and public keys of the server*

We will now generate the key pair (private key and public key), e.g. for a web server:

```
openssl genrsa  -out server.key 2048
```

The key has a length of 2048 bits.

We can protect it to prevent anyone other than *root* or authorized *processes* from accessing it:

```
chmod 600 server.key
```

The *private key - public key* pair does not have an expiration date, unlike certificates, which are issued for limited periods.

3.3.3.3. *Creating the request for a certificate to be validated by the root authority*

Using the key pair that we just generated, we will prepare a request to be submitted to the certification authority. It contains the public key, but also other information, including the name of the server or website for which we want to generate a key:

```
openssl req -new -key server.key -out server.csr
```

After running this command, we will need to fill out a few fields. To leave a field blank, we use a period (.).

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:.
Organization Name (eg, company) [Internet Widgits
   Pty Ltd]: MY COMPANY
Organizational Unit Name (eg, section) []: .
Common Name (eg, YOUR name) []:myserver.company.
   com
Email Address []: .

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

The last two fields are left blank. The *challenge password* is used to generate a password to unlock the private key, and is not required in this example.

Carefully note: it is important for the *Common Name* to match the name of the server or website that will be used. When verifying the key, the software will check that the server or website name is identical to the value stored in this field.

Some certification authorities allow multiple names to be specified here, which allows the same certificate to be used with multiple DNS entries. This is sometimes used for mail servers, enabling them to use the same certificate for both TLS pop access (*pop.myserver.com*) and webmail (*webmail.myserver.com*), etc. It is also possible to create certificates with *wildcard* characters, such as **.mydomain.com*: all subsites of this domain can be validated by the same certificate. Doing this is now officially discouraged because of the risks that it creates: if the certificate is hijacked, all of the websites of a single organization can now be forged.

Multiple certificates can also be created from the same *private key – public key* pair: one request is submitted for each web address hosted by the server.

3.3.3.4. *Signing the certificate locally*

If you do not wish to use the services of a registration authority, you need to generate the certificate yourself. The first time that you do this, you will need to create a file containing the serial numbers of the generated certificates (the *cacert.srl* file) as well as the certificates themselves, using the following command:

```
openssl x509 -req -in server.csr -CA cacert.pem -
   CAkey cacert.pem -out server.crt -
   Cacreateserial
```

If the *cacert.srl* file exists, the command for doing this is:

```
openssl x509 -req -days 3650 -in server.csr -CA
   cacert.pem -out server.crt
```

The *server.crt* file is a certificate that contains not just the public key itself, but also its hash, encrypted with the root certificate.

3.3.3.5. *Requesting a certificate signature from a registration authority*

If you have a certification authority, you can send them the file containing the certificate request (*server.csr*). In return, you will receive a certificate

containing both the public key and its encrypted hash. Usually, the fields *Organization Name* and *Organizational Unit Name* are mandatory: check with your certification authority to know which fields are required.

You will also need to obtain the root certificate (the equivalent of *cacert.crt*, the public key of the certification authority), which will allow you to verify the validity of the certificate.

3.3.3.6. *Creating a self-signed certificate without using a registration authority*

Some applications need a simple self-signed certificate, i.e. one that is not signed by a registration authority, even a personal one. Once the certificate creation request has been executed, run the following command:

```
openssl x509 -req -days 3650 -in server.csr -
   signkey server.key -out server.crt
```

The file *server.crt* contains the self-signed certificate.

3.3.4. *Where are keys and certificates stored?*

In Linux-based systems using the *OpenSSL* library, the following folders are usually used:

– */etc/ssl/certs*: contains certificates and public keys. This folder is readable by all device users.

– */etc/ssl/private*: contains the private key(s) of the device.

Access to this second folder needs to be strictly managed: if somebody obtains the private key, they can decrypt all encrypted messages, or impersonate the sender. This type of attack is known as *man in the middle* [WIK 15a].

By default, this folder can only be accessed by the *root* account, which prevents web server processes from accessing the private keys, meaning that it will not be able to initiate *https* connections.

To change this behavior, we need to allow the *ssl-cert* group to browse this folder, and give private key read permissions to this group:

```
sudo chmod g+x ssl-cert /etc/ssl/private
sudo chmod g+r ssl-cert /etc/ssl/private/*
```

The *sudo* command grants administrative rights for executing commands.

Now, the account used by the web server, which in this case is *www-data*, is added to the *ssl-cert* group:

```
sudo usermod -a -G ssl-cert www-data
```

After restarting the Apache server, it will be able to access the private key and correctly manage *https* connections.

3.3.5. Commands for viewing keys and certificates

To view the contents of a private key:

```
openssl rsa -in server.key -text

Private-Key: (2048 bit)
modulus:
    00:bb:6c:c9:c5:57:4f:f3:7c:83:56:a9:2d:c1:5d:
(...)
publicExponent: 65537 (0x10001)
privateExponent:
    07:d3:12:d9:5a:3b:cc:3e:76:7d:37:b2:e1:4f:b2:
(...)
prime1:
    00:e4:40:84:cf:08:d9:b5:c8:2e:74:a3:3f:75:72:
(...)
exponent1:
    23:be:72:cd:d5:2d:fa:c8:a1:75:c4:86:d0:86:a1:
(...)
coefficient:
    58:c0:2b:ea:71:eb:a5:60:e0:a0:25:f5:7c:b1:94:
(...)
    75:53:08:da:ea:e1:74:6d
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
```

```
MIIEogIBAAKCAQEAu2zJxVdP83yDVqktwV0saR1QafKL+
   XblgtQT8pHDx2XQCWdW
(...)
IK/QeXnadKjz8jZT78nxL+N1xqK5RSBaAsk/
   hzQsdVMI2urhdG0=
-----END RSA PRIVATE KEY-----
```

To view the associated public key:

```
openssl rsa -in server.key -pubout

writing RSA key
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ
(...)
OwIDAQAB
-----END PUBLIC KEY-----
```

To view a certificate:

```
openssl x509 -in server.crt -text -noout

Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 16235780570068490813 (0
           xe15114685a1f1a3d)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=FR, CN=equinton
        Validity
            Not Before: Jun 22 15:35:19 2016 GMT
            Not After : Jun 20 15:35:19 2026 GMT
        Subject: C=FR, CN=equinton
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:bb:6c:c9:c5:57:4f:f3:7c:83:
(...)
```

```
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha256WithRSAEncryption
         23:29:ee:e1:6b:50:2c:d9:9e:6b:4c:10:2e
            :84:cc
(...)
```

To check the validity of the certificate:

```
openssl verify -CAfile cacert.crt server.crt

server.crt: OK
```

3.4. Implementing the HTTPS protocol

3.4.1. *Understanding the HTTPS protocol*

Today, digital communications require a certain minimum threshold of confidentiality (to protect passwords, bank details, etc.). The principle of the HTTPS protocol, which was originally *HTTP over SSL* and is now *HTTP over TLS* [WIK 15h] (SSL was abandoned as it is no longer considered to be secure), is to encrypt all communications.

TLS is a symmetric-key-based encryption procedure that meets this need perfectly. Its only shortcoming, like all symmetric-key protocols, is that the key must first be exchanged between both parties without being intercepted.

To achieve this, the designers proposed starting the dialogue by using asymmetric encryption. Figure 3.6 gives a slightly simplified diagram of the protocol.

Suppose a browser wishes to connect to a web server by HTTPS (usually the web server redirects the browser to the HTTPS protocol, but the result is the same). It retrieves the certificate provided by the web server and then checks its validity by decrypting the hash using the public key of the certification authority. If the certificate is not recognized, the browser displays a warning message and asks the user to confirm before accessing the requested page.

If the certificate is valid, the browser generates a symmetric key using the TLS protocol. It then encrypts it using the public key of the server and sends it to the server.

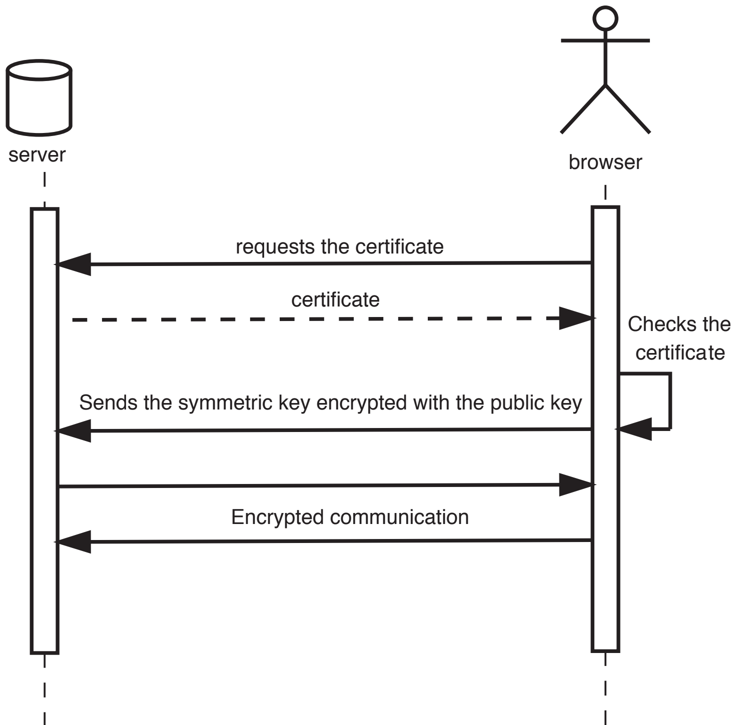

Figure 3.6. *Principle of the HTTPS protocol*

The server retrieves the key and then decrypts it using its private key: information can now be exchanged confidentially between the server and the browser.

In practice, the server sends other information along with the certificate, such as a list of accepted symmetric encryption protocols.

With this procedure, only the server has to justify its identity, and only the server has to present a digital certificate. However, some implementations of the HTTPS protocol require the client to also present a certificate. This is in particular used for direct communications between servers for automatically exchanging messages.

3.4.2. *Implementing the HTTPS protocol*

The HTTPS protocol relies on digital certificates. To protect an application, you must first obtain a certificate. This certificate contains the exact name of the application's web address, for example *my_application.com*. This allows the browser to check that the given certificate matches the name of the requested website.

The certificate must also be provided by a recognized certification authority: otherwise, the browser will indicate that it was not able to verify the certificate. This is because the chain of certification, as presented above (see section 3.2.5), is not complete.

The next step is to make sure that the Apache server redirects all requests sent by the *http* protocol to the *https* protocol. This is done in the virtual host configuration file, which can be found in the */etc/apache2/sites-available* folder. Here is an example configuration:

```
<VirtualHost *:80>
    ServerName myapp.com
    ServerPath /myapp.com
    RewriteEngine On
    RewriteRule ^ https://myapp.com/%{REQUEST_URI}
        [R]
</VirtualHost>
```

Now, all content sent via HTTP, i.e. to port 80, is instead redirected to the HTTPS protocol. To do this, we use the URL rewrite function (*Rewrite*).

We will now see how to configure the access to our application in HTTPS mode:

```
<VirtualHost *:443>
    ServerName myapp.com
    ServerPath /myapp.com
    ServerAdmin admin@myapp.com
    ServerSignature off
```

The *VirtualHost* command now points to port 443, which is reserved for HTTPS. We also see a few general-purpose commands, one of which shows the name of the site administrator (a generic address).

The *ServerSignature off* command hides the version of the Apache server.

Support for the HTTPS protocol is enabled using the commands:

```
  SSLEngine on
  SSLCertificateFile /etc/ssl/certs/myapp.cert
  SSLCertificateKeyFile /etc/ssl/private/myapp.
     key
  SSLCACertificateFile /etc/ssl/certs/cachain.
     pem
```

We can also see the certificate access paths: *SSLCertificateFile* is the certificate (signed public key), *SSLCertificateKeyFile* is the private key and *SSLCACertificateFile* is the file containing the chain of certification, i.e. the public keys of each authority that signed the public key.

Now, we must specify where the application code is located, using the following command:

```
DocumentRoot /var/www/myappApp/myapp
```

And finally, application-specific log files:

```
  CustomLog /var/log/apache2/myapp-access.log
     combined
  ErrorLog /var/log/apache2/myapp-error.log
</VirtualHost>
```

All access information is stored in the file */var/log/apache2/myapp-access.log*, (*combined* specifies the choice of format for the log file), and the errors are logged in the file */var/log/apache2/myapp-error.log*.

3.4.3. *Testing the SSL chain*

Once the certificates have been successfully added and Apache has been configured, it is informative to check that the encryption has been correctly

configured. Several tools are available to do this, either in downloadable form to test the configuration of an intranet, or directly online. With the latter, we simply need to give the address of the web server, then the website launches a more or less comprehensive analysis, first checking the chain of certification and then the protocols being used. SSLLABS [LAB 16] offers possibly one of the most comprehensive tools. This allows us to identify any obsolete protocols that might still be accepted by the server and yields a list of compatible operating systems and browsers.

3.5. Improving the security of the Apache server

In this section, we will briefly discuss how to improve the security of the Apache web server. This is not an exhaustive guide to system administration, but simply gathers together a few useful principles: system administrators are likely to be aware of all of these things, but might nonetheless appreciate a checklist to ensure that nothing has been forgotten.

The configurations listed below are for a Linux server running Ubuntu Server 14.04 LTS.

3.5.1. *Ensuring that the server hosting Apache has the latest security updates*

When a server is first installed, it is usually up-to-date with the latest security patches. However, since new vulnerabilities are continuously being discovered, servers need to be updated regularly, generally at least once a day. For Ubuntu or Debian-type distributions, we can use the *unattended-upgrades* package, which was specifically designed to automate this process.

This package uses the */etc/apt/apt.conf.d /50unattended-upgrades* file to specify all of the parameters used to configure the update software. This can be used to specify the following, among other things:

– which repositories should be used to perform updates. Typically, for a server, we should only install security updates;

– whether there are any packages that should not be updated: this is often the case for the *libc* library, which is essential for the system to operate properly; any changes to this component can cause significant disruption;

– whether the device should restart after an update that requires it (usually after installing a new kernel). If this is enabled, it is possible to specify the time at which the server should restart. Automatically restarting protects against vulnerabilities in the kernels themselves, which can result in denial of service or privilege escalation attacks. However, restarting is a complex operation, especially in virtual environments where servers share resources, and it is advisable to ensure beforehand that the operation can be completed safely.

There are also other settings, for example for sending emails, which we will not discuss here.

3.5.2. *Prohibiting low-security protocols*

In the 1990s, the American government banned the export of strong security protocols outside of its territory. This led people to use less secure protocols. At the time, this was not a problem, but, with the computing power available today, they have become easy to circumvent.

To remain compatible with existing websites, the Apache web server continues to allow all types of protocol. Unfortunately, even if secure certificates are used, Apache will continue to use weakly encrypted connections, unless these connections are disallowed.

To do this, we must edit the configuration file */etc/apache2/sites-available/default-ssl* and add the following lines to the section beginning with *<VirtualHost _default_:443>*:

```
SSLProtocol All -SSLv2 -SSLv3
SSLHonorCipherOrder On
SSLCipherSuite ECDHE-RSA-AES128-SHA256:AES128-GCM-
   SHA256:HIGH:EECDH+AESGCM:EDH+AESGCM:AES256+
   EECDH:AES256+EDH:!MD5:!aNULL:!EDH:!RC4
SSLCompression off
```

The first line disables the SSL protocol, which is considered to be obsolete, and instead imposes TLS. The next two lines force the use of the SHA256 signing algorithm (the default is SHA-1, which uses 128-bit keys), as well as other new protocols that are considered to be reliable. Finally, the

last command disables SSL compression, which can create security issues (this directive is usually set to *off* by default).

Remy van Elst discusses these questions further in a fairly comprehensive article on comprehensively securing SSL connections with Apache2 [ELS 15].

Today, there are websites that generate configurations adapted to the usage context (web server, supported browsers, etc.). For example, the one provided by the Mozilla foundation [MOZ 16b] is particularly comprehensive. Referring to these tools makes it easier to stay ahead of advancements in encryption technologies.

3.5.3. *Preventing request flooding*

One of the simplest ways of "taking down" a server, i.e. preventing it from responding to queries, is to send a burst of tens of thousands of requests simultaneously. This is called a denial-of-service attack.

The Apache *evasive* module counters these types of attack. It blocks identical requests sent within a give unit of time, and limits the number of requests from the same user, again per unit time.

This module is easy to install:

```
sudo apt-get install libapache2-mod-evasive
```

Its parameters can be configured in the file */etc/apache2/mods-available/evasive.conf*.

Here are example values (the default values are shown in brackets):

– *DOSHashTableSize* (3097): size of the table used to record calls;

– *DOSPageCount* (2): number of requests called in a given unit of time by the same user. The default value is usually sufficient unless the same component is called multiple times from within a single page;

– *DOSSiteCount* (50): number of calls to the website within a given unit of time, summed over all pages;

– *DOSPageInterval* (1): time interval, in seconds, used to count the number of requests;

– *DOSSiteInterval* (1): time interval used to count all incoming requests;

– *DOSBlockingPeriod* (10): period, in seconds, which blocked users will not be able to complete requests.

To make the module work, each of the parameters must be activated by uncommenting them before restarting the Apache server.

With the default parameter values, if a user sends two identical requests to the server within 1 s, they will be blocked for 10 s. They will also be blocked if they send more than 50 requests per second.

Whenever a request is blocked, a message is saved in the error log file */var/log/apache2/error.log*:

```
[Thu Jun 23 09:21:08.539066 2016] [evasive20:error
   ] [pid 19145] [client ::1:54188] client denied
   by server configuration: /var/www/html/eabxcol/
   display/images/logo.png, referer: https://
   localhost/eabxcol/index.php?module=management
```

The *referer* specifies the page that was originally called, i.e. the one that was blocked.

The advantage of this module is that the server will use very few resources if the threshold is exceeded: the request will not be completed, and the processing time will be very short.

It is usually installed in the *reverse-proxy* server, which acts as a gateway for all applications or websites associated with the organization. Since it is still executed on the server itself, it is only effective up to a certain point: if the attack is truly massive, the server resources will eventually be overwhelmed anyway. Other, more complex techniques can be implemented, but require modifications to the actual network equipment, or even at the level of the Internet Service Provider (ISP).

Like any protective measure, one should first consider whether the risk justifies implementing other measures in addition to this module.

3.5.4. *Implementing a request filter*

Apache proposes a module, *mod_security*, for analyzing the requests received by the server and filtering them, if required, before forwarding them to the PHP engine. This is called the application firewall, which acts as an interface before the actual connection to the web server.

This module can be complex to configure, but examples of basic configurations are available.

First, it needs to be installed on the server:

```
sudo apt-get install libapache2-mod-security2
```

The Apache server automatically restarts, but the module is not yet operational: no directives have been defined. The default root configuration file is located in the */etc/modsecurity* folder, and needs to be renamed to be visible to the module:

```
cd /etc/modisecurity
sudo cp modsecurity.conf-recommended modsecurity.
   conf
```

You will need to modify the contents of this file to suit your own configuration. To allow files of up to 64 MB to be downloaded, you need to modify the following directive:

```
SecRequestBodyLimit 67108864
```

We will now activate the basic directives used by the module. Type the following commands:

```
for f in ‘ls /usr/share/modsecurity-crs/base_rules
   /*‘ ; do sudo ln -s $f ; done
```

and restart the Apache server:

```
sudo service apache2 restart
```

By default, the module is configured to only register abnormal events (directive *SecRuleEngine DetectionOnly* in the file *modsecurity.conf*). You can test that the program properly detects anomalies, for example by entering the string; *or 1 = 1*- - into a submitted field (this command represents an SQL injection attack (see section 4.2.1)). An entry will be created in the log file */etc/apache2/modsec_audit.log*.

Check the log file, and disable any modules that you do not wish to use by removing their links.

Very well-written English documentation is available from DigitalOcean [DIG 15]. You can also visit the official project website [MOD 15].

You should test this module before activating it on a production server. Check the log files that it generates in particular (they can become large very quickly), and only activate the modules that you actually need.

If your application is properly coded and secured, and basic precautions have been taken directly in Apache, this module may not necessarily be useful to you. However, for open-source applications taken from the net, it can be invaluable. For example, room booking software written several years ago might not necessarily include the latest protections against the most common types of attack, such as SQL injection. Rather than rewriting it or investing time in modifying its code to make it secure, it is likely more beneficial to set it up behind an application firewall such as *modsecurity*. If this firewall is properly configured, it will block most common types of attack.

Of course, this approach should only be used for low-priority applications that pose limited risk in the event that an attack should succeed.

3.5.5. *Allowing page header modifications*

Modern browsers are now capable of implementing several different security checks, provided that the server asks for them to be activated. They are disabled by default because, in some cases, they can prevent applications from working.

The checks are, for example, designed to limit the risk that cookies from other webpages are stolen using specially designed scripts written in

JavaScript. To ask the browser to implement these precautions, they need to be included in the HTML header sent by the web server.

To allow the application to modify HTML headers, we first need to load a module to complement Apache, using the following commands:

```
a2enmod headers
service apache2 restart
```

Below, we will see which instructions should be used to improve the protection of the application.

3.5.6. *Authorizing .htaccess files*

.htaccess files allow us to modify the behavior of the Apache server in individual folders. This is mainly used to prevent access to critical folders, e.g. the folder that contains the connection details for the database.

Since these files contain special instructions, we must ask Apache to allow them to be executed.

Usually, when a new application is installed on a server, a new virtual site is registered in the */etc/apache2/sites-available* folder using a configuration file. This file indicates, among other things, the physical location of the application within the server and specifies various parameters, such as encryption certificates for *https* connections. We need to make sure that this file contains the following instructions:

```
<Directory /myfolder>
      AllowOverride all
      Order allow,deny
      allow from all
</Directory>
```

The first instruction, *AllowOverride*, allows all Apache directives in the folder */myfolder* to be overridden, which ensures that all of the *.htaccess* files of the application are correctly taken into account.

3.5.7. *Hiding the version information of Apache and PHP*

Allowing attackers to view the versions of installed software gives them valuable information, especially if everything is not fully up-to-date. If we are using an obsolete version of PHP that is known to be vulnerable to a particular type of attack, revealing the version number tells the attackers exactly how to penetrate the system.

It is easy to check whether a web application is displaying its version numbers by installing the *wappAlyzer* [WAP 15] module in the Firefox browser. This module collects information about all components of the application and the server: the operating system, web engine, language and all third-party software components, such as *JQuery* [JQU 15]. If the version numbers are not hidden, this module will show them.

To prevent the Apache server and its PHP module from revealing their version numbers, we simply need to edit the configuration files. For Apache, we need to edit the file */etc/apache2/conf.d/security*, and check the configuration of the following parameters:

```
ServerTokens Prod
ServerSignature Off
```

PHP can be configured by modifying the following parameter in the */etc/php5/apache2/php.ini* file:

```
expose_php = Off
```

Apache must then be restarted for these changes to take effect.

3.6. In summary

Applications can never be considered to be intrinsically safe: their run-time environment plays a significant role in their overall security.

Nowadays, encryption is essential for protecting applications. Only allowing access in *https* mode is a necessary first step, but is not enough: the web server must also be properly configured to disallow obsolete protocols.

Setting up measures such as the Apache *evasive* mode or an application firewall strengthens the security of the application, provided that these measures are configured correctly. Launching an application into production requires close collaboration between the system administrators and developers. The needs of both parties must be taken into account. This will allow a good balance between performance and protection to be achieved.