Advanced Networks — Laboratory 4

Juliusz Chroboczek

19 March 2025

Exercice 1 (X.509 certificates)**.** Using your favourite web browser, connect to `https://galene.org`.

1. Is the communication encrypted? Click on the padlock[1].
2. Is the server authentified with the client? What is the *Subject* declared in the certificate? The *Subject alternate name*?
3. Who certified the certificate? With what signature algorithm?
4. Use the command `gnutls-cli` to obtain the same information. Which symmetric encryption algorithm has been negotiated?
5. What is the interesting thing about the *Subject alternate name* of the certificate served by `https://www.irif.fr`?

Exercice 2 (An HTTPS server)**.**

1. Create an X.509 certificate with the following command:

   ```
   openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365
               -out cert.pem
   ```

   Which files were created? What do they contain? (Use the command `file`.)
2. Write an HTTP server in Go that serves a single page. (You may for example copy-paste the server from lab 1.) Turn your server into an HTTPS server by replacing the call to `ListenAndServe` with a call to`ListenAndServeTLS`.
3. Test your server using a web browser, using `curl`, using `gnutls-cli`. What is the issue?
4. Why did I write the package `github.com/jech/cert` (two reasons)?

Exercice 3 (Username/password authentication)**.**

1. Modify the handler of your HTTP server so that it checks for an `Authorization` header, and fails the request with a code 401 if there is none. If the header is present, display the value of the header (`log.Println`) and continue. How does the client behave? How is the header encoded? (You may use the command `base64 -di`.)

1. Or the incomprehensible icon to the left of the address bar if you are using a recent version of Chromium.

2. Modify the server to test that the username is "`einstein`" and the password is "`elsa`", and fail the request if that is not the case.
3. What is the name of the authentication mechanism implemented in this exercice? Does it authentify the client or the server? Why did we implement it over HTTPS and not over HTTP (two reasons)?

Exercice 4 (Stateless third-party authentication)**.**

1. Write a program in Go that sends a `POST` request to

   `https://galene.org:8446/get-token`

   whose body contains a JSON object with the following structure:

   ```
   {
       "username": "xxx",
       "password": "yyy"
   }
   ```

   The username is arbitrary (use your dog's name[2]), the password is "`Rosebud`".
2. Modify your program so that it sends a `GET` request to

   `https:/galene.org:8446/top-secret`

   with a `Authorization` header of type `Bearer` with the token as its value. The syntax is described in RFC 6750 Section 2.1.
3. We used two requests, using two distinct authentication mechanisms, in order to access the data. What is the purpose of this procedure? Why is the token called a *bearer* token?
4. The token consists of three parts separated with colons "`.`". Using the command `base64 -di`, examine each of the three parts. Does the client need to know the structure of the token?
5. Suppose that the server were to switch to stateful tokens. What changes would be required in the client?
6. Both interactions are protected by TLS. Is that necessary?
7. What prevents a malicious client from generating a fake token?
8. What happens if you replay the same token? What happens if you replay the token with a 40-second delay?
9. How would you protect against replay? How would you implement token revocation?

2. Or your cat's, but only if they agree.