École Centrale Lyon

Cookies sécurité web

Auteur : Ugo Insalaco (Chest)

12 décembre 2020

Table des matières

1 Introduction

2 Les cookies

2.1 Qu'est ce qu'un cookie ?

Un cookie est une donnée, une information que peut stocker un site web sur le navigateur du client. Ces données sont alors envoyées à chaque requête du client au serveur afin qu'il puisse agir en fonction de ces cookies. Par exemple, on utilise souvent des cookies afin de maintenir une connexion à un site web ouverte, ou garder en mémoire l'historique d'un panier d'achat. Les cookies sont propres à chaque site internet et ne doivent pas pouvoir être récupérés par d'autres site (cela pourrait par exemple impliquer la prise de contrôle du compte d'un utilisateur ou la récupération de données sensibles). Certains cookies sont persistants et restent jusqu'a suppression manuelle par l'utilisateur et d'autre expirent après un certain temps après fermeture du navigateur web. On peut observer les cookies en cours d'utilisation sur chrome en faisant click droit » inspecter » application » cookies

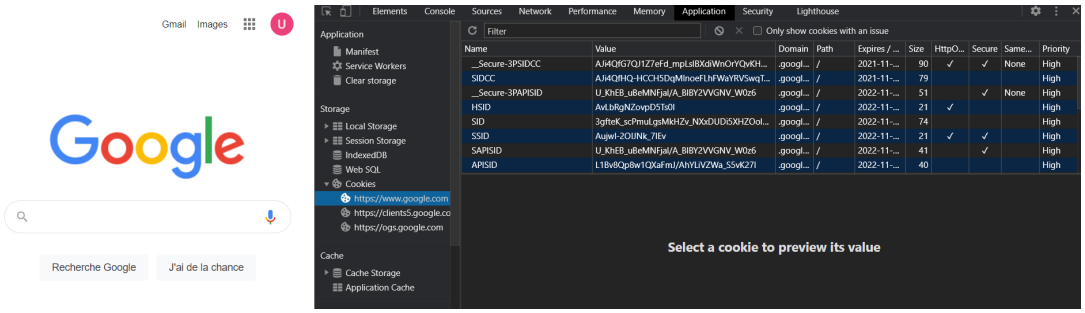


FIGURE 1 – Affichage des cookies dans son navigateur

2.2 Le package cookie-parser

Un parser est un middleware permettant de récupérer des informations provenant de l'utilisateur (du client) au moment de la requête et de les placer dans l'élément de requête côté serveur (par exemple urlencoded-Parser pour les informations dans l'url, body-parser pour les informations dans le corps d'une requête post etc...).
Le cookie-Parser va récupérer les cookies de l'utilisateurs pour chaque requête où il est activé et les rendre disponibles au serveur.

Pour installer le package dans une application nodeJs, on effectue simplement la commande :

npm install –save cookie-parser

Pour utiliser le cookie-parser, on l'importe dans un code nodeJs avec un 'require' et on le définit en tant que middleware sur les requêtes où l'on veut l'utiliser :

Au moment de son utilisation, le cookie-parser peut prendre des arguments, ici "secret" signifie que l'on pourra si l'on souhaite encoder la valeur des cookies.

```
const express = require('express')
const cookieParser = require('cookie-parser')

const app = express()

app.use(cookieParser("secret"))
```

FIGURE 2 – implémentation des cookies dans le code du serveur nodeJs

```
app.get('/getcookies', function(req, res){
    res.send(JSON.stringify({
        cookies: req.cookies,
        signed_cookies: req.signedCookies
        }
    ))
})

app.get('/setcookies', function(req,res){
    res.cookie("Gregory", "Vial")
    res.cookie("Nicolas", "Hourcade", {signed: true})
    res.send("Le cookie est set")
})

app.get('/clearcookies', function(req,res){
    res.clearCookie("Gregory")
    res.clearCookie("Nicolas")
    res.send('les cookies ont été enlevés')
})
```

FIGURE 3 – Gestion du cycle de vie des cookies

Le code suivant montre comment gérer le cycle de vie des cookies :

On peut créer un cookie lors de la réponse de la requête avec la méthode 'res.cookie' et si l'on veut on peut le crypter avec l'option 'signed : true' (il faut que l'argument 'secret' soit passé au moment de l'instanciation du cookie-parser)
Grâce au cookie-parser, les cookies sont placés dans 'req.cookies' et 'req.signedCookies' enfin on peut supprimer des cookies en utilisant la méthode res.clearCookie et en passant en argument le nom du cookie.

Dans l'exemple ci-dessus, on peut faire une requête au navigateur à /setcookie et on observe bien que deux cookies sont créés dont l'un est encodé.

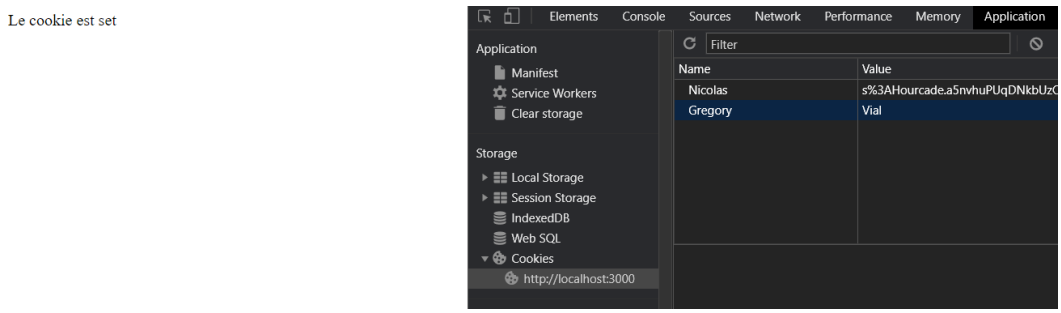


FIGURE 4 – Création de cookie dans le navigateur

Beaucoup d'autres options sur la gestion des cookies sont disponibles sur la Documentation du package cookie-parser

2.3 Gestion des token avec Jsonwebtoken

Le package jsonwebtoken permet de gérer facilement et avec plus d'options l'encodage et le décryptage de valeurs, que l'on peut ensuite définir comme cookie crypté.
On l'installe avec la commande :

npm install –save jsonwebtoken

Et on l'importe comme un package classique dans le code avec un 'require'.

```
app.get('/sign_cookie', function(req,res){
    const token = jwt.sign({prenom:'Pietro', nom:'Salizonni'}, 'cle_secrete')
    res.cookie('identite', token)
    res.send('le cookie est crypté')
})

app.get('/decode_cookie', function(req,res){
    jwt.verify(req.cookies.identite, 'cle_secrete', function(err,decoded){
        if(err){
            res.send('Le cookie n\'a pas été décodé')
        }
        else{
            res.send('le cookie contient : '+JSON.stringify(decoded))//=> {prenom: 'Pietro', nom: 'Salizonni'}
        }
    })
})
```

FIGURE 5 – Utilisation de jsonwebtoken (jwt=require('jsonwebtoken'))

On peut crypter la valeur d'un objet javascript avec une certaine clé secrète. On applique ensuite cette valeur à un cookie. Au moment de faire la requête 'decode_cookie', on utilise la fonction 'verify' qui va chercher à décoder la valeur du cookie. Cette fonction prend en argument une fonction de callback qui sera appelé si l'on a réussi à décoder le token ou non.

De la même façon, une grande documentatin de ce package est disponible ici. Il est par exemple possible de définir un cookie qui expire après fermeture du navigateur.

3 Mysql

4 Securité

4.1 XSS

Le XSS (ou cross site scripting) est le fait d'utiliser une interface d'entrée d'information de l'utilisateur dans un site internet (un champs de texte, tous les formulaires web etc...) afin de faire executer du code côté serveur à son avantage. Côté client un utilisateur pourra toujours modifier les informations sur la page affichée, modifier directement le code Html mais cela n'a pas tant de risque puisque cela reste côté client et donc n'influe pas sur les différents utilisateurs et le serveur. Au contraire si du code peut être executé côté serveur, il sera alors relayé à tous les utilisateurs et peut être la cause de destruction de

différents services web, ou pire, le vol de données, le vol d'argent sur des sites permettant l'utilisation de comptes bancaires etc...

Pour développer ce cas, on peut montrer cette exemple grâce à une application présentant un champs d'entrée de texte envoyé directement au serveur puis renvoyé à l'écran du client.

FIGURE 6 – Exemple de page potentiellement vulnérable au XSS

Ici le texte est entré dans la zone de texte puis envoyé au serveur dans une liste qui est renvoyé au client (tel une liste de commentaires sur un forum) et affiché sur la page web.

```
const express = require('express')
const path = require('path')
const bodyParser = require('body-parser')

const app = express()
const jsonParser = bodyParser.json()
const password = "pwd"
let values = [];

app.set('views', path.join(__dirname, '/'))
app.set('view engine', 'ejs')

app.use('/', express.static(path.join(__dirname, './')))

app.get('/', function(req,res){
    res.cookie("Catherine", "Musy")
    res.render('xss.ejs')
})

app.get('/password', function(req,res){
    res.send(password)
})

app.post('/setvalue', jsonParser, function(req,res){
    values.push(req.body.text)
    console.log(values)
    res.json({text:"la requête a fonctionné"})
})

app.get('/getvalue', function(req, res){
    res.send(values)
})
app.listen(3001)
```

FIGURE 7 – Exemple de serveur vulnérable au XSS

Lors de l'envoi du commentaire avec le bouton submit, la requête est faite en POST sur la route '/setvalue' qui va ajouter le commentaire à la liste sur le serveur (On récupère les informations de la requête POST grâce au package Body-parser). Lors du chargement

de la page, c'est la requête '/getvalue' qui est faite et qui va renvoyer la liste présente sur le serveur.

Maintenant si on envoie par exemple le code "<script>alert('vous avez été hack')</script>" celui-ci va être exécute lorsqu'il sera affiché sur le client car le navigateur ne vérifie pas s'il s'agit de code qui s'exécute au moment de l'envoyer au client ou bien du simple texte. De plus, on peut vouloir volontairement envoyer du code js au client afin de créer un rendu dynamique au moment du chargement de la page. Comme ce code est maintenant stocké sur le serveur en tant que simple commentaire, il sera exécuté à chaque fois qu'un utilisateur ira chercher les commentaires et se connectera au site web ce qui n'est pas du tout un comportement désiré.

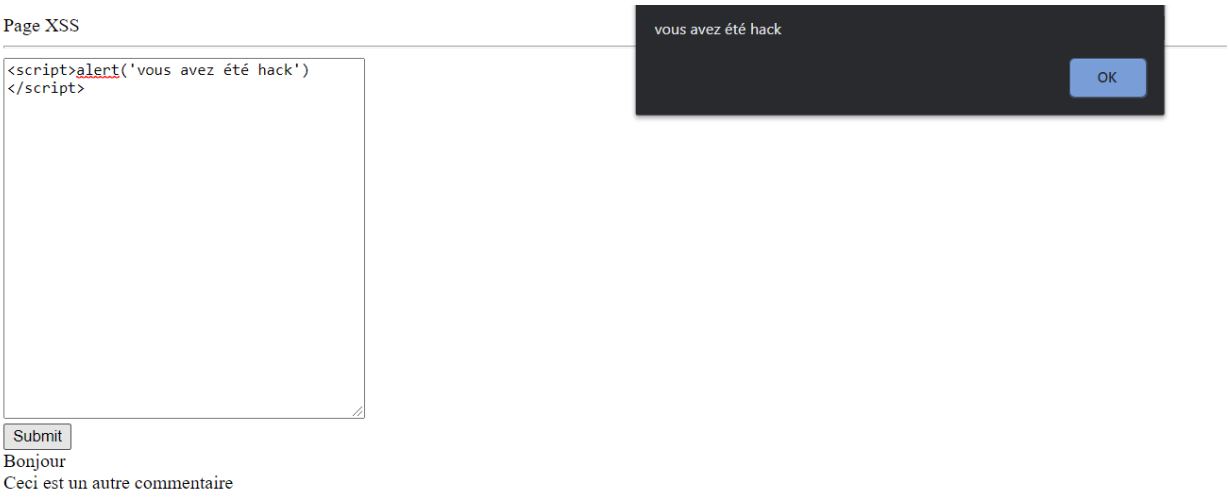


FIGURE 8 – hack d'un page par une faille XSS

On peut imaginer d'autres dangers à cette faille comme par exemple l'envoi de requêtes à un site pirate contenant par exemple nos cookies et données du site où l'on est connecté actuellement, ou encore le vol d'informations personnelles de chaque utilisateur (les exemples sont disponibles dans les fichiers du code exemple).

De façon générale, il ne faut jamais faire confiances aux informations envoyées par l'utilisateur (ou le frontend) au serveur. Il faut au maximum échapper le code qui pourrait être envoyé au serveur afin d'éviter tout comportement pirate. Le package XSS de npm permet de faire cela et de sécuriser le serveur aux potentielles attaques XSS
Pour utiliser le package, il suffit comme tous les autres de l'installer avec npm :

npm install –save xss

puis de l'importer et d'utiliser la méthode importée A CHAQUE information envoyée par l'utilisateur :

On a donc pu échapper le code envoyé par l'attaquant et éviter une faille XSS

```
app.post('/setvalue', jsonParser, function(req,res){
    values.push(xss(req.body.text))
    console.log(values)
    res.json({text:"la requête a fonctionné"})
})
```

FIGURE 9 – Utilisation du package XSS sur le serveur

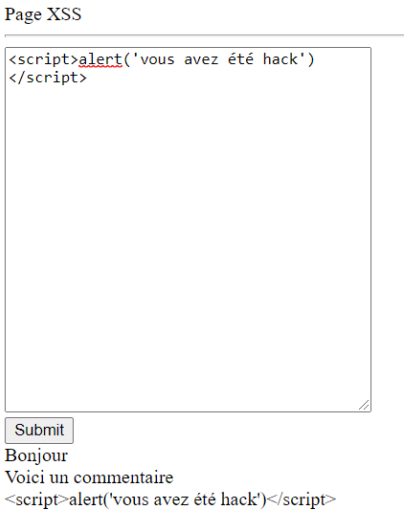


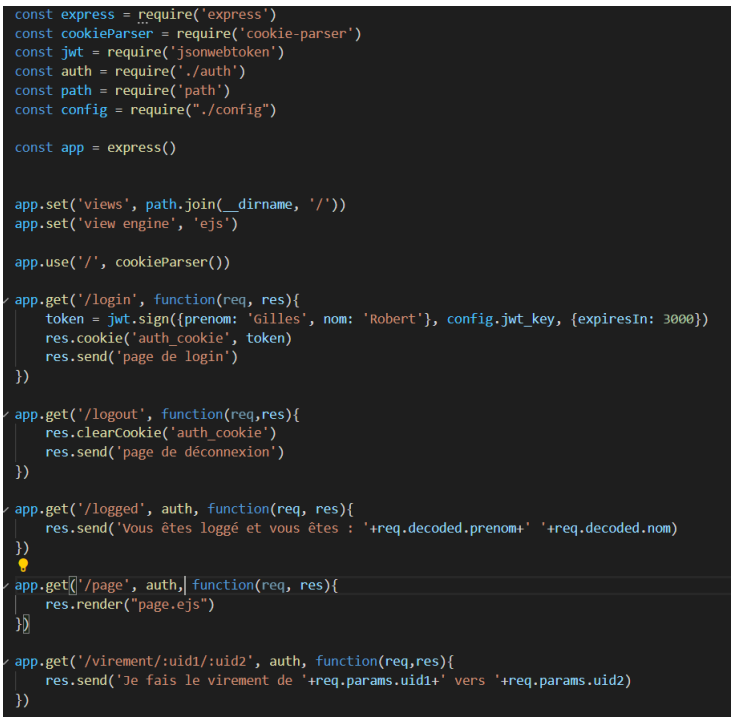
FIGURE 10 – Sauvetage de notre faille xss

4.2 CSRF

Le CSRF (Cross Site Request Forgery) est l'utilisation des droits d'un autre utilisateur pour attaquer un serveur. Sur une application web, une personne possédant les droits suffisants peut faire appel à n'importe quelle requête et effectuer des actions qui peuvent être dangereuse si elles sont disponibles à un utilisateur lambda (Seul le propriétaire d'un compte en banque doit pouvoir effectuer un virement depuis son compte par exemple). L'attaquant CSRF va se baser là dessus pour forcer l'utilisateur à effectuer une telle action à son insu par exemple en lui faisant faire une requête vers le service de virement par un lien qui lui serait envoyé.

Création d'un système d'authentification

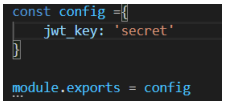
Pour mettre cette vulnérabilité en évidence, on commence par créer un système d'authentification. Pour cela on met en place un serveur où l'utilisateur peut se login (route /login), ce qui va créer un cookie contenant des informations d'authentification (nom, prénom, cryptées avec jsonwebtoken)



```
const express = require('express')
const cookieParser = require('cookie-parser')
const jwt = require('jsonwebtoken')
const auth = require('./auth')
const path = require('path')
const config = require("./config")

const app = express()


app.set('views', path.join(__dirname, '/'))
app.set('view engine', 'ejs')

app.use('/', cookieParser())

app.get('/login', function(req, res){
    token = jwt.sign({prenom: 'Gilles', nom: 'Robert'}, config.jwt_key, {expiresIn: 3000})
    res.cookie('auth_cookie', token)
    res.send('page de login')
})

app.get('/logout', function(req,res){
    res.clearCookie('auth_cookie')
    res.send('page de déconnexion')
})

app.get('/logged', auth, function(req, res){
    res.send('Vous êtes loggé et vous êtes : '+req.decoded.prenom+' '+req.decoded.nom)
})

app.get('/page', auth, function(req, res){
    res.render("page.ejs")
})

app.get('/virement/:uid1/:uid2', auth, function(req,res){
    res.send('Je fais le virement de '+req.params.uid1+' vers '+req.params.uid2)
})
```

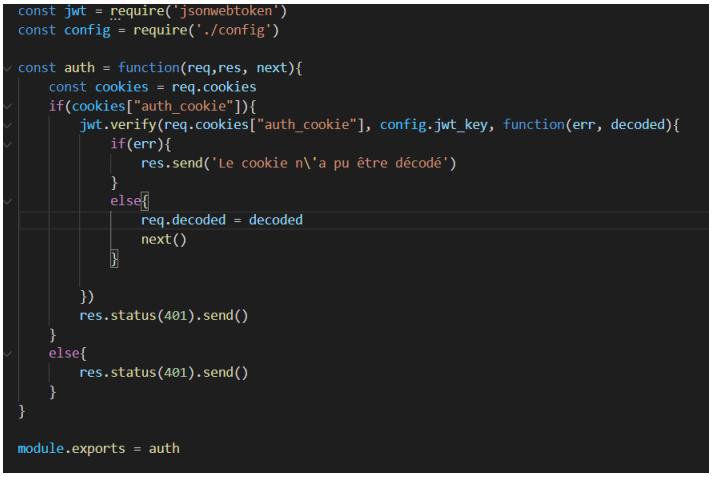
FIGURE 11 – Exemple de serveur exposé au CSRF

Ensuite il faut pouvoir récupérer les informations d'authentifications de l'utilisateur à chaque requête pour savoir quels privilèges lui attribuer. c'est pour cela qu'on utilise le middleware 'auth' qui sert à récupérer les informations dans le cookie (en décryptant à l'aide de la clé secrète) pour adapter la réponse du serveur :



```
const config ={
    jwt_key: 'secret'
}

module.exports = config
```

FIGURE 12 – fichier config.js



```
const jwt = require('jsonwebtoken')
const config = require('./config')

const auth = function(req,res, next){
    const cookies = req.cookies
    if(cookies["auth_cookie"]){
        jwt.verify(req.cookies["auth_cookie"], config.jwt_key, function(err, decoded){
            if(err){
                res.send('Le cookie n\'a pu être décodé')
            }
            else{
                req.decoded = decoded
                next()
            }

        })
        res.status(401).send()
    }
    else{
        res.status(401).send()
    }
}

module.exports = auth
```

FIGURE 13 – middleware 'auth'

Les requêtes passant maintenant par le middleware 'auth' sont accessibles seulement si l'utilisateur s'est auparavant authentifié par le requête login et qu'il possède ainsi un cookie d'authentification.

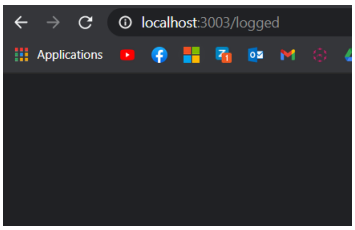


FIGURE 14 – utilisateur non loggé

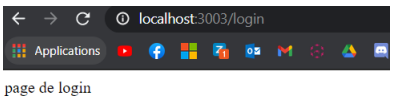


FIGURE 15 – page de login

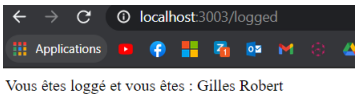


FIGURE 16 – utilisateur loggé

Imaginons maintenant que sur une page d'un forum se trouve maintenant un lien du site où l'on est connecté où la requête correspondante est un virement bancaire entre le compte de celui qui est connecté et celui qui ne l'est pas. Evidemment celui qui n'est pas connecté ne doit pas pouvoir faire ce virement, mais celui qui l'est, lui, peut donc en cliquant dessus, effectuer une action à son insu en faisant une requête qu'il ne prévoyait pas.
Une page dangereuse peut donc avoir un tel lien :

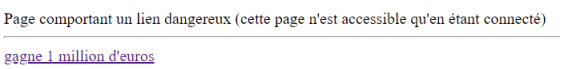


FIGURE 17 – page avec un lien dangereux

et en cliquant dessus, la requête non désirée est envoyée.

résolution par token csrf

Cette attaque se base donc sur le fait que l'attaquant puisse "créer" de toute pièce les requêtes qui lui sont avantageuses et que ces requêtes sont accessibles sans vérification totale que c'est l'utilisateur qui l'a bien initiée.
Pour résoudre ce problème on introduit le concept de token csrf. C'est une valeur / un jeton qui va être échangé à chaque requête entre le client et le serveur de la même manière qu'un cookie, à la différence que le token n'est pas envoyé par défaut au serveur, il doit être définit manuellement au moment de la requête et ainsi la requête ne peut pas être forgée de toute pièce puisque l'attaquant ne connaît pas ce token (les cookies n'auraient donc pas eu de valeur en terme de sécurité).
Pour implémeter cela, on créé une page de login qui au moment de clicker sur un bouton de login va faire la requête login côté serveur et 'set' côté client (dans le localStorage) le token csrf qui sera renvoyé

```
const login = function(e){
    $.ajax({
        method:'GET',
        url: '/login',
        success: function(res){
            localStorage.setItem("csrf", res)
            $("#log_response").html("l'utilisateur est bien loggé")
        },
        error: function(err){
            $("#log_response").html("l'utilisateur n'a pas pu être loggé")
        }
    })

}

const logout = function(e){
    $.ajax({
        method: 'GET',
        url: '/logout',
        success: function(res){
            localStorage.removeItem("csrf")
            $("#log_response").html("vous avez bien été logout")
        },
        error: function(err){
            $("#log_response").html("vous n'avez pas pu être logout")
        }
    })
}
$("#login").on("click", login)
$("#logout").on("click", logout)
```

FIGURE 18 – code javascript de la page de login

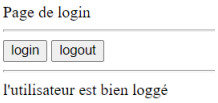


FIGURE 19 – nouvelle page de login

Côté serveur on modifie la requête de login pour prendre en compte l'ajout de token csrf. On place aussi le token dans le cookie pour plus tard vérifier que les données de session et le token correspondent bien :

```
app.get('/login', function(req, res){
    token = jwt.sign({prenom: prenom, nom: nom, id: id}, config.jwt_key, {expiresIn: 3000})
    res.cookie('auth_cookie', token)
    res.send(id)
})
```

FIGURE 20 – nouvelle requête de login

Et on modifie enfin le middleware d'authentification pour vérifier le token csrf lorsque les requêtes sont assez sensibles :

```
const jwt = require('jsonwebtoken')
const config = require('./config')

const auth = function(req,res, next, csrf = false){
    const cookies = req.cookies
    if(cookies["auth_cookie"]){
        jwt.verify(req.cookies["auth_cookie"], config.jwt_key, function(err, decoded){
            if(err){
                res.send('Le cookie n\'a pu être décodé')
            }
            else{
                const csrf_token = req.get("X-csrf-token")
                if(!csrf || decoded.id ===csrf_token){
                    req.decoded = decoded
                    next()
                }
                else{
                    res.status(401).send("erreur csrf")
                }
            }

        })
        res.status(401).send()
    }
    else{
        res.status(401).send()
    }
}

module.exports = auth
```

FIGURE 21 – nouvelle requête de login

maintenant si l'on revient sur notre page comportant un lien dangereux, le fait de cliquer dessus n'implique pas la requête de virement de se faire. En revanche lorsque cette requête est définie directement dans le code Js de la page, on peut la faire. On a donc un plus grand contrôle dessus.

erreur csrf

FIGURE 22 – click sur le lien dangereux

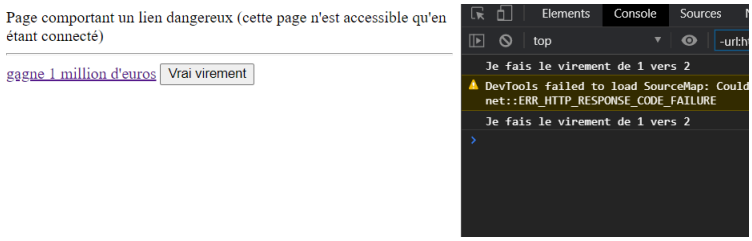


FIGURE 23 – click sur le bouton

Autres liens :

— https ://www.imperva.com/learn/application-security/csrf-cross-site-request-forgery/
— https ://stackoverflow.com/questions/3220660/local-storage-vs-cookies
— https ://beta.hackndo.com/attaque-xss/
— https ://fr.wikipedia.org/wiki/Cross-site_request_forgery