Rapport d'analyse de sécurité

Mickael Bonjour, Jonathan Zaehringer

January 15, 2020

Contents

Introduction

Dans ce rapport, il sera détaillé la phase de sécurisation d'une application de messagerie web. L'application à sécuriser est une application ayant été codée dans un temps restreint ne permettant pas un développement réfléchi en termes de sécurité. Les points ayant été sécurisés ont été faits dans un court temps, sans prendre en compte l'ensemble de l'application.

Toutefois, le développement a été fait par deux personnes ayant de l'expérience dans le développement ce qui risque d'avoir été un avantage pour réduire le nombre de failles de sécurité en premier lieu.

Vous trouverez une analyse de menaces faite sur ce projet pour voir les points critiques, les scénarios possibles et leurs conséquences. Ainsi que les différentes attaques que nous avons faites.

Analyse de Menaces

Description du système

L'application est une messagerie permettant d'échanger des messages entre les utilisateurs `authentifiés`. Elle se compose d'un frontend de type SPA (Single Page Application) développé en VueJS. Ce frontend communique avec une API backend proposant les différents services permettant le bon fonctionnement de l'application. Développé en expressjs, il communique avec une base de données MariaDB stockant l'ensemble des données persistantes.

Fonctionnalités

Les utilisateurs non authentifiés peuvent uniquement s'authentifier. Les utilisateurs authentifiés peuvent échanger des messages avec les autres utilisateurs. Il peut consulter ces messages et ceux qu'il a envoyés.

Les administrateurs de l'application ont la capacité de créer des nouveaux comptes, de les activer/désactiver et de les supprimer.

Les comptes peuvent potentiellement être payants à l'avenir.

Hypothèses et exigences de sécurité

La partie backend est de confiance avec la base de données ainsi que ces administrateurs.

Les exigences de sécurités sont les suivantes : - L'application ne doit être accessible qu'aux utilisateurs authentifiés. - Les messages doivent être uniquement visibles par l'expéditeur et le destinataire. - Un utilisateur n'ayant pas le rôle d'administrateur ne doit pas pouvoir accéder au panel d'administration. - Un compte désactivé ne doit pas pouvoir s'authentifier ni pouvoir interagir avec l'application.

Actifs à haute valeur

Base de données des messages : - Confidentialité, sphère privée - Intégrité (une modification pourrait avoir de grave conséquence) - Incident nuirait gravement à la réputation de l'application

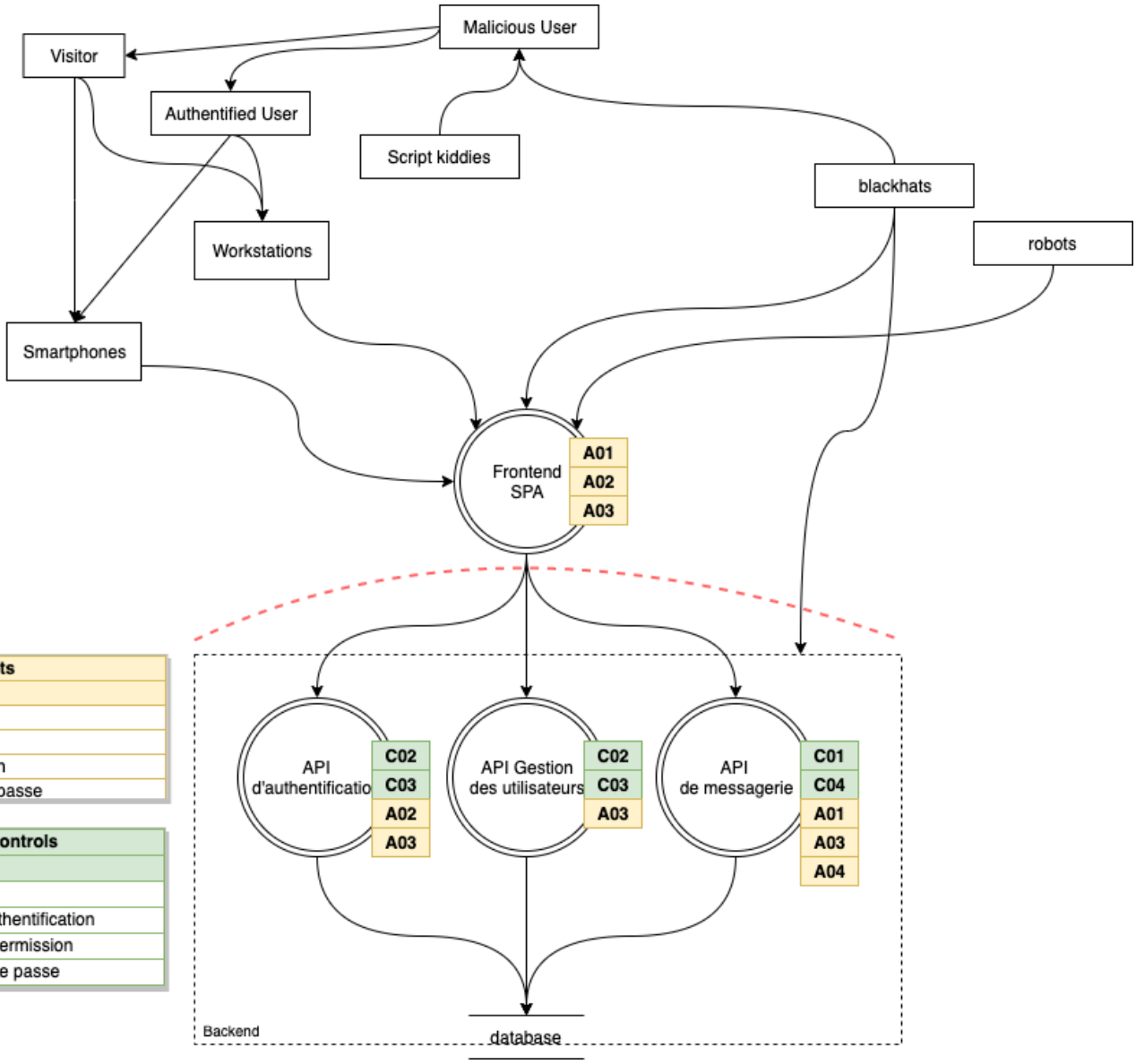
Base de données des utilisateurs : - Confidentialité, sphère privée - Incident nuirait à la réputation de l'application

Source de menaces

Hackers, script kiddies - Motivation : s'amuser - Cible : découverte de bug, spam - Potentialité : Moyenne

Cybercrime (spam, maliciels) - Motivation : financière - Cible : Phising, spam sur les clients authentifiés, lecture/modification des messages, accès au panel admin - Potentialité : Élevée

Concurrent - Motivation : Arrêt du service - Cible : Bug dans le backend - Potentialité : Moyenne



Assets	
ID	**Description**
A01	Credential
A02	Message(s)
A03	Token de Session
A04	Hash du mot de passe

Security Controls	
ID	**Description**
C01	Authentification
C02	Verification de l'authentification
C03	Verification de la permission
C04	Hashage du mot de passe

Figure 1: DFD

DFD

Scénario d'attaque

Lecture de message

- Business impact : Élevé (perte de confidentialité)
- Provenance : Cybercrime
- Motivation : Récupération d'information, destruction de la réputation
- Actif : Base de données des messages
- Sénario d'attaque : Injection SQL, CSRF, Erreur de contrôle
- Contrôles : Contrôle d'origine (CORS), Prepared statement, validation d'entrée

Modification de message

- Business impact : Élevé (perte d'intégrité)
- Provenance : Cybercrime
- Motivation : Modification d'information, destruction de la réputation
- Actif : Base de données des messages
- Sénario d'attaque : Injection SQL, CSRF, Erreur de contrôle
- Contrôles : Contrôle d'origine (CORS), Prepared statement, validation d'entrée

Spamming

- Business impact : Moyen (perte de réputation)
- Provenance : Utilisateur authentifié
- Motivation : Financière
- Actif : Base de données utilisateur
- Sénario d'attaque : Accès à de faux comptes ou volés
- Contrôles : Détection de vol de compte

Vol de session

- Business impact : Élevé (perte de confiance)
- Provenance : Script kidding, Cybercrime
- Motivation : Envoi de message illégitime, Elevation de privilège
- Actif : Base de données utilisateur, Base de données de messages
- Sénario d'attaque : XSS
- Contrôles : HTML Sanitize

Mitigation des risques

- Utilisation de Prepare Statement
- Sanitize l'ensemble des données textes
- Vérification de l'origine des requêtes (CORS)
- Vérification des droits et de l'authentification
- Validation de toutes les entrées
- antiCRSF token

Attaques

CSRF

Les attaques CSRF sont problématiques dans ce cas, car aucune protection n'est présente (token, CORS). Cependant l'application utilise une API qui n'accepte que du JSON. Cela complique largement le problème, car il est déjà impossible sur un navigateur moderne d'appeler de manière asynchrone une route qui ne définit pas de CORS.

La seule possibilité est l'attaque par un formulaire classique qui lui redirige le client sur l'API.

Formulaire d'exemple

```
<html>
    <body>
        <form method="POST" action="http://localhost:3000/message" enctype="application/json">
            <input type="number" name='to' value='1'/>
            <input type="text" name='subject' value="CRSF PWNED" />
            <input type="text" name='message' value='HAHAH' />
            <input type="submit" />
        </form>
        <form method="POST" action="http://localhost:3000/message">
            <input type="number" name='to' value='1' />
            <input type="text" name='subject' value="CRSF PWNED" />
            <input type="text" name='message' value='HAHAH' />
            <input type="submit" />
        </form>
        <form method="POST" action="http://localhost:3000/message" enctype="text/plain">
            <input type="text" name='{"to":1,"subject":"CRSF PWNED", "message":"' value='HAHAH"}' />
            <input type="submit" />
        </form>
        <form method="POST" action="http://localhost:3000/message" enctype="application/json">
            <input type="text" name='{"to":1,"subject":"CRSF PWNED", "message":"' value='HAHAH"}' />
            <input type="submit" />
        </form>
    </body>
</html>
```

Mitigation

Dans ce cas, expressjs semble sauvé la situation, car il est très strict sur la forme des données ce qui bloque ces états. Pour le 3e formulaire, un JSON valide (`{"to":1,"subject":"CRSF PWNED", "message":"=HAHAH"}`) est envoyé, mais avec un mime-type `text/plain` mais cela n'est pas accepté par express. Dans le cas d'autres technologies comme `spring`, cette attaque est totalement possible. Dans le 4e formulaire, le résultat n'est clairement pas compris par express.

Par contre, dans le cas du premier formulaire qui n'est pas standard, mais n'a pas d'impact sur le type de contenu (`application/x-www-form-urlencoded`). L'erreur qui survient est une mauvaise construction du parser d'expressjs qui fournit un objet null donc le prototype contient les champs envoyés. Le problème provient de la présence du parser mais le code n'est fait uniquement pour parser du JSON ce qui fait planter lors de l'envoi de ces données.

Cela n'a pas d'impact, car le code suivant fait planter la requête :

```
if (!req.body.hasOwnProperty('to') || !req.body.hasOwnProperty('subject') || !req.body.hasOwnProperty('message')
    res.sendStatus(400);
    return;
}
```

Dans ce cas, la fonction `hasOwnProperty` n'existe pas et crée une erreur `500`. Cependant cela pourrait être exploité au cas de faille dans le parser, il est donc nécessaire de corriger cela en supprimant la ligne `app.use(express.urlencoded({ extended: false }));`.

Pour conclure, le site n'est pas attaquable par CSRF, car express est suffisamment rigoureux pour ne pas laisser passer des JSON forger par du HTML.

Injection SQL

Tentative d'injection SQL sur plusieurs routes de L'API et par l'interface.

Payload d'exemple

```
' OR '1' = '1
```

```
'); SELECT * FROM users; --
```

```
') UNION SELECT * FROM users;
```

Mitigation

L'ensemble des requêtes SQL sont faites avec des prepared statements de la libraire `better-sqlite3` en JavaScript. Il ne semble pas y avoir de vulnérabilités sur les prepared statements dans cette librairie ().

XSS

Nous avons essayé une attaque de type XSS depuis l'interface web ainsi que sur l'API directement.

Payload d'exemple

```
<script>
alert(1);
</script>
```

Mitigation

Aucun résultat n'a été efficace, car Vuejs sanitize l'ensemble des strings qu'il affiche à l'utilisateur. Aucune vulnérabilité permettant une XSS n'existe depuis décembre 2018 version <2.5.17 (https://snyk.io/vuln/npm:vue).

Nous pouvons voir que l'injection du message c'est bien fait dans la base de données :


```
{
  "message":"<script>\nalert(1);\n</script>",
  "id":12,
  "fromId":5,
  "fromName":"jzaehrin",
  "toId":1,
  "toName":"admin",
  "timestamp":1577785913,
  "subject":"XSS",
  "read":1
}
```

La protection provient bien du moment de l'affichage par le biais de vuejs qui protègent l'utilisateur des données récupérées.

Crypto

Analyse de la cryptographie utilisée dans l'application. Lors de l'analyse du code, nous nous sommes concentrés sur la partie d'authentification assez rapidement. Ce que l'on a assez rapidement remarqué c'est l'utilisation de AES-CBC pour chiffrer les données ensuite utilisées lors des contrôles d'authentification.

Nous avons donc décidé d'examiner ce chiffrement et quels en était les possibles vulnérabilités. De là on a pu remarqué que l'IV utilisé n'était jamais changé, c'est donc toujours le même utilisé (on aurait pu le remarquer sans regarder le code d'ailleurs, grâce au fait que l'IV est envoyé dans le token). Ceci induit une première vulnérabilité, en effet en réutilisant ce IV certains chiffrés seront identiques les uns des autres, ce qui permet de savoir à quelle fréquence un utilisateur se connecte si l'on voit les paquets envoyés.

De plus, comme je l'ai mentionné auparavant, l'IV est envoyé dans le token. Celui-ci est donc dans le cookie 'Authorization' et est donc modifiable. Ceci est une très grosse vulnérabilité avec CBC, en effet l'IV est utilisé lors du déchiffrement du premier bloc comme montré ci-après.

Ainsi l'attaque se déroule comme suit : Nous partons d'un utilisateur authentifié quelconque (ici test). Son token Authorization se présente comme suit :

eyJpdiI6IjcyYzMwNTI2N2QwZGNkYTcxZjdjMTk4ZjBkNjZiYmNlIiwiZW5jcnlwdGVkRGF0YSI6IjM1YWZjMzAyNWM5NzE4MGQ3MGMx
NDFkMjZkN2RlMzE4OGI1NzFkZTljYzZkODc3NjU0YTc2YWVhMGFjMDc0MzQwYmNkZDVmYTI3NDM4MjlkYzYxMTBhMGM5MmRkYWRlMCIs
ImtleSI6IkFRam1CdW5GcXFrZlZtdTFiNGVYUXI4RVAyTk5NYkZcbiIsInVzZXJfaWQiOjIsImxldmVsIjowfQ

Qui décodé de la base64 donne :


```
{
"iv":"72c305267d0dcda71f7c198f0d66bbce",
"encryptedData":
  "35afc3025c97180d70c141d26d7de3188b571de9cc6d877654a76aea0ac074340bcdd5fa2743829dc6110a0c92ddade0",
"key":"AQjmBunFqqkfVmu1b4eXQr8EP2NNMbF\n",
"user_id":2,
"level":0
}
```


L'on voit donc l'IV qui nous est donnée en hexadécimal dans le cookie, et qui est donc manipulable. C'est un problème comme on peut le voir dans cette image ci-dessous :

Grâce à cela nous pouvons modifier le premier bloc de données à notre bon vouloir. Analysons donc ce qu'il peut bien avoir dans ce chiffre. Pour cette étape nous avons besoin du code, sinon nous ne pouvons a priori pas deviner le payload du chiffré. (Fichier login.js)

```
let session = {};
    session.user = user.id;
    session.role = user.level;
    // 24 hour validity
    session.validity = Math.round(new Date().getTime() / 1000) + 24 * 60 * 60;

    let cookie = Crypto.encrypt(JSON.stringify(session));
```

Nous pouvons donc savoir que le plaintext sera sous cette forme :

```
{"user":x,"role":x,"validity":x}
```

Et c'est vérifiable en utilisant la clé de déchiffrement (à des fins de démonstration, pas utile dans l'attaque) :

Sachant qu'AES fonctionne en bloc de 16bytes l'on peut déterminer le premier bloc de données comme suit : {"user":x,"role" Et là on voit finalement le problème, en effet l'on peut modifier l'ID de l'utilisateur dans le token.

En remplaçant le 9e byte de l'IV nous pourrons modifier l'id de l'utilisateur. Ainsi 72c305267d0dcda71f7c198f0d66bbce va devenir 72c305267d0dcda71c7c198f0d66bbce(après quelques essais). Et le déchiffrement deviendra (à but de démonstration, la clé n'est pas nécessaire pour l'attaque):

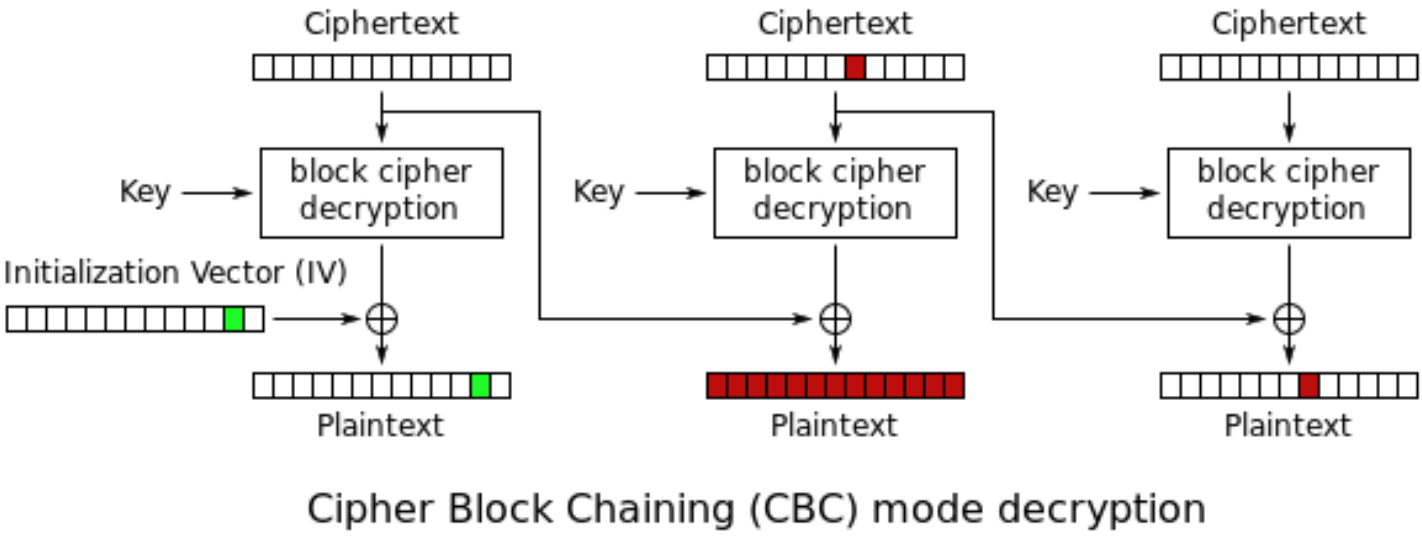


Figure 2: Explanation of CBC bit-flip attack

Figure 3: Example déchiffrement

Figure 4: Example de déchiffrement 2

Et en recodant en base64 le payload modifié, nous pouvons faire des requêtes simulant l'utilisateur choisi (ici 1). Comme l'on peut le voir en faisant cette requête :

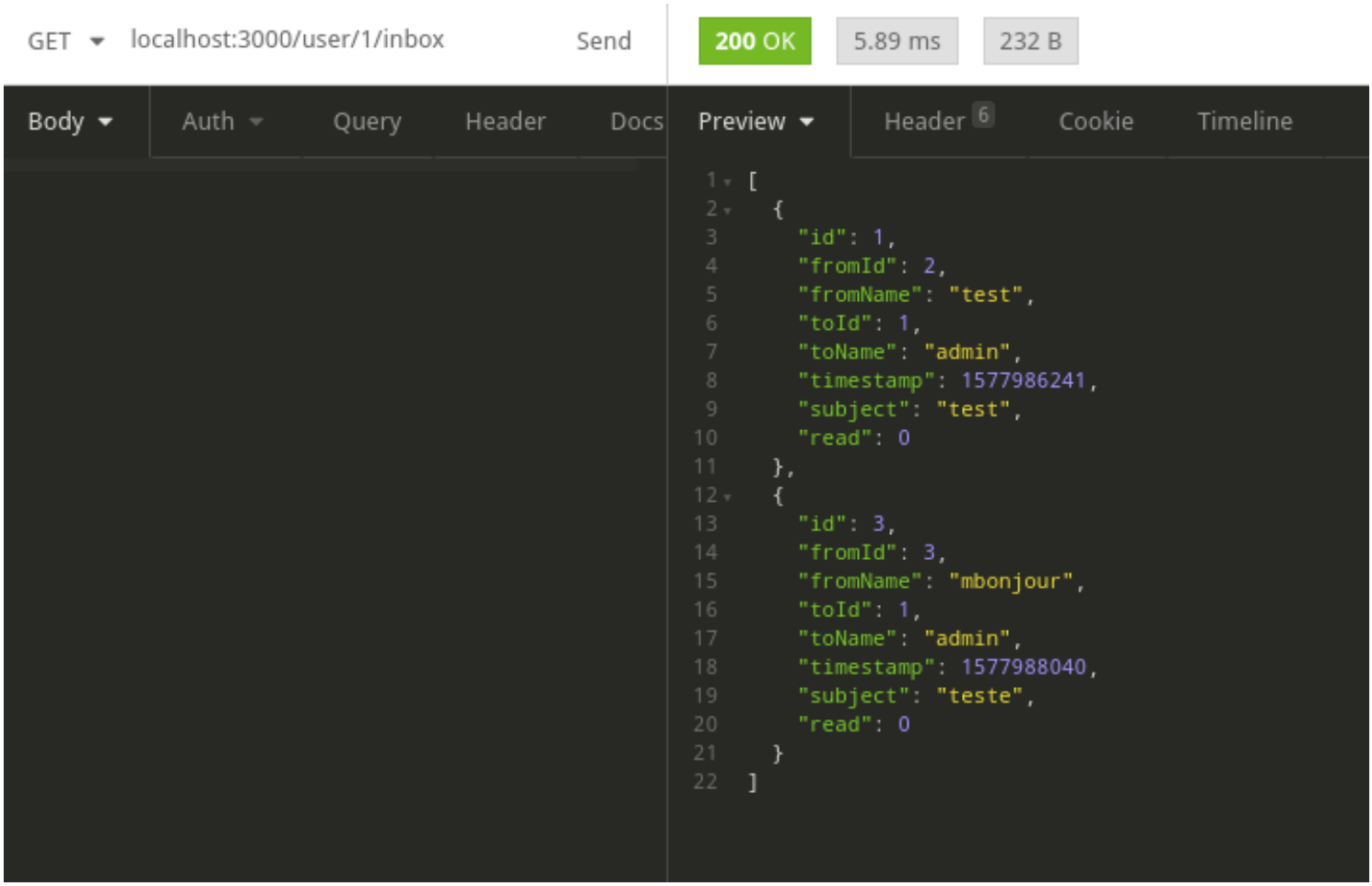


Figure 5: Example attaque

Mitigation

Pour cette attaque c'est une erreur d'implémentation la principale cause du problème. Pour éviter ce genre d'attaque il faudrait passer à un autre mode de chiffrement, ce serait sûrement plus sûr, comme GCM p. ex. qui est un chiffrement authentifié et qui permettrait d'être sûr que l'IV n'est pas modifié. Cependant il faudra modifier le code du fichier crypto.js afin que l'IV soit recréer à chaque chiffrement.

```
const iv = crypto.randomBytes(16);
```

Autrement dit, mettre cette ligne dans la fonction encrypt. Nouvelle version avec GCM :

```
/ TAKEN FROM https://codeforgeek.com/encrypt-and-decrypt-data-in-node-js/

// Nodejs encryption with CTR

const crypto = require('crypto');
const algorithm = 'aes-256-gcm';

const fs = require('fs');

let key;
```

```
fs.readFile('./server.key', 'utf8', function (err, data) {
    if (err) throw err;
    key =data.toString();
});

function encrypt(text) {
 let iv = crypto.randomBytes(16);
 let cipher = crypto.createCipheriv('aes-256-gcm', Buffer.from(key), iv);
 let encrypted = cipher.update(text);
 encrypted = Buffer.concat([encrypted, cipher.final()]);
 return { iv: iv.toString('hex'), encryptedData: encrypted.toString('hex'),
    tag: cipher.getAuthTag().toString('hex') };
}

function decrypt(text) {
 let iv = Buffer.from(text.iv, 'hex');
 let encryptedText = Buffer.from(text.encryptedData, 'hex');
 let decipher = crypto.createDecipheriv('aes-256-gcm', Buffer.from(key), iv);
 decipher.setAuthTag(Buffer.from(text.tag, 'hex'))
 let decrypted = decipher.update(encryptedText);
 decrypted = Buffer.concat([decrypted, decipher.final()]);
 return decrypted.toString();
}

function sha256(text){
 return crypto.createHash('sha256').update(text, 'utf8').digest('hex')
}

function randomString(){
 return crypto.randomBytes(10).toString('hex');
}

exports.encrypt = encrypt;
exports.decrypt = decrypt;
exports.sha256 = sha256;
exports.randomString = randomString;
```

Conclusion

Pour conclure ce rapport, nous avons pu voir les points critiques de l'application. L'intégrité et la confidentialité sont des points essentiels à cette application.

Une seule faille s'est avérée possible à exploiter. Une erreur dans la conception du token de session chiffrée a été découverte. Cette faille ne permet pas de devenir, `admin` mais permet de lire les messages de certains utilisateurs et même d'utiliser leur compte. Cela reste très grave pour l'application.

Hormis cette faille, l'application se voit être relativement bien sécurisée. L'utilisation de bonne pratique comme les `prepared statements` permet d'éviter toutes injections SQL. De plus, il est important de noter qu'un grand nombre de failles ne sont pas présentes par le faite que les technologies sont récentes et protègent automatiquement de ces attaques si on ne cherche pas à le désactiver. L'aspect strict de `expressjs` concernant les requêtes HTTP protège contre les CSRF en évitant de considérer du JSON avec un `mime type` incorrect. Ainsi que `vuejs` qui s'occupe de `sanitize` l'ensemble des données affichées à l'utilisateur pour éviter tout problème comme des XSS.

On pourra conclure par le faîte que l'utilisation de technologies moderne ayant été penser dans le cadre actuel d'attaque permet d'éviter un grand nombre d'attaques "simple". Toutefois, cela n'est pas magique, il est nécessaire de faire attention, car cela n'est pas forcément le cas. Il faut donc nécessairement tester pour s'assurer que les protections existent et qu'elles sont correctement utilisées.