Documentation Complète du Hash Tester

Aimad Hamdaoui

10 Janvier 2025

Table des matières

1 Introduction

Ce document présente la documentation complète d'un *hash tester* qui permet de vérifier si une application est autorisée à démarrer en s'assurant que le hash de l'exécutable se trouve dans une liste blanche (*whitelist*) gérée par un serveur web. Cette solution vise à renforcer la sécurité en autorisant uniquement l'exécution des applications validées.

2 Architecture du Système

Le système est constitué de deux composants principaux :

- **Client** :
  - Télécharge le script `hello_world.js` depuis le serveur.
  - Sauvegarde le script dans un répertoire temporaire.
  - Compile le script en un exécutable à l'aide de l'outil `pkg`.
  - Calcule le hash SHA256 de l'exécutable généré.
  - Envoie ce hash au serveur pour vérification.
  - Exécute l'exécutable si le hash est validé.
- **Serveur** :
  - Charge une whitelist des hashes autorisés à partir du fichier `whitelist.json`.
  - Expose un endpoint `/validate_executable` qui reçoit le hash à valider.
  - Expose un endpoint `/get_hello_world` qui permet de télécharger le script `hello_world.js`.

2.1 Schéma du Flux de Travail

1. **Téléchargement du script** : Le client récupère le fichier `hello_world.js` depuis le serveur.
2. **Compilation** : Le script est sauvegardé temporairement et compilé en un exécutable via `pkg`.
3. **Calcul du Hash** : Un hash SHA256 de l'exécutable est calculé.
4. **Validation** : Le client envoie ce hash au serveur pour validation.
5. **Exécution** : Si le serveur valide le hash, l'exécutable est lancé ; sinon, l'accès est refusé.
6. **Nettoyage** : Les fichiers temporaires utilisés durant le processus sont supprimés.

3 Dépendances et Installation

3.1 Prérequis

- **Node.js** et **npm** doivent être installés sur le système.
- Les packages suivants doivent être installés :
  - `axios` : pour réaliser des requêtes HTTP côté client.
  - `uuid` : pour générer des identifiants uniques.
  - `pkg` : pour compiler le script JavaScript en un exécutable natif.
  - `express` et `body-parser` : pour créer le serveur web.

3.2 Installation des Dépendances

Côté Client : Exécutez la commande suivante pour installer les dépendances nécessaires :

```
npm install axios uuid pkg
```

Côté Serveur : Installez les dépendances avec la commande suivante :

```
npm install express body-parser
```

4 Description du Code

4.1 client.js

Le fichier `client.js` réalise l'ensemble des opérations côté client :

- Téléchargement du fichier `hello_world.js` depuis le serveur.
- Sauvegarde temporaire du script dans un dossier dédié.
- Compilation du script en exécutable via `pkg`, en fonction de la plateforme (Windows, macOS ou Linux).
- Calcul du hash SHA256 de l'exécutable généré.
- Envoi du hash au serveur pour validation.
- Exécution de l'exécutable si le hash est validé par le serveur.
- Nettoyage des fichiers temporaires après exécution.

Code complet de `client.js` :

```
const axios = require('axios');
const fs = require('fs');
const path = require('path');
const crypto = require('crypto');
const { exec } = require('child_process');
const os = require('os');
const { v4: uuidv4 } = require('uuid');

// Installer le package uuid si ce n'est pas d  j   fait
// npm install uuid

const SERVER_URL = 'http://localhost:5000'; // Remplacez par l'adresse de votre
    serveur
const HELLO_WORLD_FILENAME = 'hello_world.js';
const TEMP_DIR = path.join(__dirname, 'temp'); // R pertoire temporaire

// Assurez-vous que le r pertoire temporaire existe
if (!fs.existsSync(TEMP_DIR)) {
    fs.mkdirSync(TEMP_DIR);
}

// Fonction pour calculer le hash SHA256 d'un fichier
function calculateSHA256(filePath) {
    return new Promise((resolve, reject) => {
        const hash = crypto.createHash('sha256');
        const stream = fs.createReadStream(filePath);

        stream.on('error', err => reject(err));

        stream.on('data', chunk => hash.update(chunk));

        stream.on('end', () => resolve(hash.digest('hex')));
    });
}

// Fonction pour valider le hash de l'ex cutable  aupr s du serveur
async function validateHash(hash) {
    try {
        const response = await axios.post(`${SERVER_URL}/validate_executable`, {
            hash });
        return response.status === 200;
```

```
    } catch (err) {
        if (err.response && err.response.status === 403) {
            return false;
        }
        throw err;
    }
}

// Fonction pour t l charger le contenu de hello_world.js depuis le serveur
async function downloadHelloWorld() {
    try {
        const response = await axios.get(`${SERVER_URL}/get_hello_world`, {
            params: {},
            responseType: 'text' // Recevoir le contenu en tant que texte
        });
        return response.data; // Retourne le contenu du fichier
    } catch (err) {
        throw err;
    }
}

// Fonction pour sauvegarder le script dans un fichier temporaire
function saveScript(scriptContent, tempFilePath) {
    return new Promise((resolve, reject) => {
        fs.writeFile(tempFilePath, scriptContent, 'utf8', (err) => {
            if (err) reject(err);
            else resolve();
        });
    });
}

// Fonction pour compiler le script en ex cutable    l'aide de pkg
function compileScript(tempFilePath, outputExecutablePath) {
    return new Promise((resolve, reject) => {
        // D terminez la cible en fonction du syst me d'exploitation
        let target;
        if (os.platform() === 'win32') {
            target = 'node16-win-x64'; // Remplacez par la version de Node.js
                appropri e
        } else if (os.platform() === 'darwin') {
            target = 'node16-macos-x64';
        } else {
            target = 'node16-linux-x64';
        }

        // Commande pkg
        const cmd = `pkg "${tempFilePath}" --target ${target} --output "${
            outputExecutablePath}"`;

        exec(cmd, (error, stdout, stderr) => {
            if (error) {
                console.error(`Erreur lors de la compilation avec pkg: ${error.
                    message}`);
                return reject(error);
            }
            if (stderr) {
                console.error(`stderr: ${stderr}`);
            }
            console.log(`stdout: ${stdout}`);
            resolve();
        });
    });
}
```

```
100
101 // Fonction pour ex cuter l'ex cutable compil
102 function executeExecutable(executablePath) {
103     return new Promise((resolve, reject) => {
104         exec(`"${executablePath}"`, (error, stdout, stderr) => {
105             if (error) {
106                 console.error(`Erreur lors de l'ex cution de l'ex cutable: ${
                        error.message}`);
107                 return reject(error);
108             }
109             if (stderr) {
110                 console.error(`stderr: ${stderr}`);
111             }
112             console.log(`stdout: ${stdout}`);
113             resolve();
114         });
115     });
116 }
117
118 // Fonction pour nettoyer les fichiers temporaires
119 function cleanupTempFiles(files) {
120     files.forEach(file => {
121         fs.unlink(file, (err) => {
122             if (err) console.error(`Erreur lors de la suppression de ${file}: ${
                    err.message}`);
123             else console.log(`Supprim : ${file}`);
124         });
125     });
126 }
127
128 // Fonction principale
129 async function main() {
130     try {
131         // G n rer un nom unique pour le script temporaire
132         const uniqueId = uuidv4();
133         const tempFileName = `${uniqueId}_${HELLO_WORLD_FILENAME}`;
134         const tempFilePath = path.join(TEMP_DIR, tempFileName);
135
136         // G n rer un nom unique pour l'ex cutable compil
137         const outputExecutableName = `hello_world_${uniqueId}${os.platform() ===
                'win32' ? '.exe' : ''}`;
138         const outputExecutablePath = path.join(TEMP_DIR, outputExecutableName);
139
140         console.log(`T l chargement de ${HELLO_WORLD_FILENAME}...`);
141         const scriptContent = await downloadHelloWorld();
142         console.log(`T l chargement r ussi. Sauvegarde du script
                temporairement    ${tempFilePath}...`);
143         await saveScript(scriptContent, tempFilePath);
144
145         console.log(`Compilation de ${HELLO_WORLD_FILENAME} en ex cutable ${
                outputExecutableName}...`);
146         await compileScript(tempFilePath, outputExecutablePath);
147         console.log(`Compilation r ussie. Ex cutable g n r      ${
                outputExecutablePath}`);
148
149         console.log(`Calcul du hash SHA256 de l'ex cutable...`);
150         const hash = await calculateSHA256(outputExecutablePath);
151         console.log(`Hash de l'ex cutable : ${hash}`);
152
153         console.log('Validation du hash aupr s du serveur...');
154         const isValid = await validateHash(hash);
155
156         if (isValid) {
```

```
            console.log('Hash valid . Ex cution de l\'ex cutable...');
            await executeExecutable(outputExecutablePath);
            console.log(`Ex cution r ussie de ${outputExecutableName}`);
        } else {
            console.error('Hash non autoris . Acc s refus .');
        }

        // Nettoyage des fichiers temporaires
        cleanupTempFiles([tempFilePath, outputExecutablePath]);
    } catch (err) {
        console.error('Erreur :', err.message);
    }
}

main();
```

4.2 server.js

Le fichier `server.js` implémente un serveur web avec Express qui a pour rôles :

— Charger une whitelist de hashes depuis le fichier `whitelist.json`.

— Valider le hash envoyé par le client via l'endpoint `/validate_executable`.

— Fournir le fichier `hello_world.js` via l'endpoint `/get_hello_world`.

Code complet de `server.js` :

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

const app = express();
const PORT = 5000;

// Middleware
app.use(bodyParser.json());

// Chemin vers les fichiers
const WHITELIST_FILE = path.join(__dirname, 'whitelist.json');
const HELLO_WORLD_FILE = path.join(__dirname, 'hello_world.js');

// Charger la whitelist
function loadWhitelist() {
    try {
        const data = fs.readFileSync(WHITELIST_FILE, 'utf8');
        return JSON.parse(data).map(hash => hash.toLowerCase());
    } catch (err) {
        console.error("Erreur lors du chargement de la whitelist :", err);
        return [];
    }
}

// Endpoint pour valider le hash de l'ex cutable
app.post('/validate_executable', (req, res) => {
    const { hash } = req.body;
    if (!hash) {
        return res.status(400).json({ status: 'error', message: 'Hash manquant.'
            });
    }

    const whitelist = loadWhitelist();

    if (whitelist.includes(hash.toLowerCase())) {
```

```
        return res.status(200).json({ status: 'success', message: 'Hash valid.' });
    } else {
        return res.status(403).json({ status: 'error', message: 'Hash non autoris .' });
    }
});

// Endpoint pour t l charger hello_world.js
app.get('/get_hello_world', (req, res) => {
    const hash = req.query.hash;
    if (!hash) {
        return res.status(400).json({ status: 'error', message: 'Hash manquant.' });
    }

    // Ici, nous ne validons pas le hash du client, mais vous pouvez ajouter une logique suppl mentaire si n cessaire.

    return res.sendFile(HELLO_WORLD_FILE, (err) => {
        if (err) {
            console.error("Erreur lors de l'envoi du fichier :", err);
            res.status(500).json({ status: 'error', message: 'Erreur serveur.' });
        }
    });
});

// D marrer le serveur
app.listen(PORT, () => {
    console.log(`Serveur en  coute  sur http://localhost:${PORT}`);
});
```

4.3 hello_world.js

Ce fichier est un script simple qui affiche le message « Hello, World ! ». Il est téléchargé par le client et compilé en un exécutable.

Code de `hello_world.js` :

```
// hello_world.js
function main() {
    console.log("Hello, World!");
}

module.exports = { main };
```

5 Fonctionnement Global

5.1 Étapes du Processus Côté Client

1. **Téléchargement du Script** : Le client télécharge le fichier `hello_world.js` depuis le serveur.
2. **Sauvegarde Temporaire** : Le script est sauvegardé dans un répertoire temporaire.
3. **Compilation** : Le script est compilé en un exécutable via `pkg`, en ciblant la plateforme (Windows, macOS ou Linux).
4. **Calcul du Hash** : Le client calcule le hash SHA256 de l'exécutable généré.
5. **Validation** : Le hash est envoyé au serveur via l'endpoint `/validate_executable` pour validation.

6. **Exécution** : Si le hash est présent dans la whitelist, l'exécutable est lancé. Sinon, l'accès est refusé.
7. **Nettoyage** : Les fichiers temporaires utilisés sont supprimés.

5.2 Validation et Sécurité

— La sécurité de ce mécanisme repose sur la gestion d'une whitelist de hashes autorisés.
— Il est recommandé d'utiliser une connexion sécurisée (HTTPS) entre le client et le serveur pour éviter toute interception.
— Des contrôles supplémentaires (authentification, vérification d'intégrité, etc.) peuvent être ajoutés pour renforcer la sécurité.

6 Conclusion

Ce hash tester permet de contrôler l'exécution d'applications en validant que le hash de l'exécutable figure dans une liste blanche maintenue sur un serveur web. Ce mécanisme garantit que seules les applications autorisées peuvent être lancées, offrant ainsi un niveau de sécurité additionnel. Des améliorations et des extensions (telles que l'authentification ou l'utilisation de protocoles sécurisés) peuvent être envisagées pour adapter ce système à des environnements nécessitant une sécurité renforcée.