UDC 004.8 **doi: 10.26906/SUNZ.2023.2.098**

G. Golovko, D. Iievliev

National University «Yuri Kondratyuk Poltava Polytechnic», Poltava, Ukraine

ENHANCED AUTHORIZATION FOR SECURE MANAGEMENT OF SENSITIVE DATA IN HYBRID APPLICATIONS

Abstract: Sensitive data is often managed by cloud-based applications, which can be vulnerable to attackers who seek unauthorized access to this data. Traditional approaches to authorization may not be sufficient to protect sensitive data from such attacks. In this article, we propose an enhanced authorization approach that uses a combination of symmetric and asymmetric cryptography to secure sensitive data. Specifically, we propose generating a unique encryption key per file and a set of public and private keys per user, which are used to encrypt and decrypt the data. We demonstrate the feasibility of our approach with examples in Node.js, showing how to generate public and private keys, encrypt and decrypt files, and store encrypted data on a drive. Our approach provides an effective solution to the problem of managing sensitive data in hybrid applications, while preserving user and developer convenience.

Keywords: Cryptography, Public-key cryptography, Authorization, Node.js.

Introduction

Cloud-based applications are commonly used to manage sensitive data, such as customer information and messages. However, such data can be vulnerable to attacks by unauthorized users seeking to gain access to this data. Traditional approaches to authorization, such as role-based authorization and granular permissions systems, may not be sufficient to protect sensitive data from such attacks. This is because even if an application is designed with the strongest authentication and authorization mechanisms, it can still be vulnerable to security issues in the libraries used to build it. For example, in Node.js, one of the most commonly exploited vulnerabilities is Prototype Pollution [1], which can allow attackers to gain administrative privileges and access sensitive data. We need to embrace the fact that attackers could potentially get access to storage with sensitive data.

Proposal

We propose an enhanced authorization approach that uses a combination of symmetric and asymmetric cryptography [2] to secure sensitive data. Specifically, we propose:

1. Do not store unencrypted data. We accept that the attackers could access the data, but they will not be able to read it without the encryption key.
2. Use a unique encryption key per file. We accept that attackers might get much faster hardware in the future and find an encryption key. Using a unique encryption key per file dramatically increases attackers' effort to decrypt data and make it much harder to use the Harvest now, decrypt later [3] attack.
3. Use a unique set of public and private encryption keys per user. Even if one user's key is compromised, the attacker will not be able to decrypt files from other users.
4. Always encrypt a file encryption key with a user public encryption key. This ensures that only authorized users can decrypt it and read file data.

Implementation

We demonstrate the feasibility of our approach with examples in Node.js, a popular platform for building cloud-based applications. Node.js has a built-in standard crypto [4] package which provides routines for symmetric and asymmetric cryptography. Each user should have their own public and private keys. In order to generate a key pair we should use the generateKeyPair() [5] (Listing 1).

```
const { generateKeyPair } = require('node:crypto');
generateKeyPair('rsa', {
 modulusLength: 4096,
 publicKeyEncoding: {
 type: 'spki',
 format: 'pem',
 },
 privateKeyEncoding: {
 type: 'pkcs8',
 format: 'pem',
 cipher: 'aes-256-cbc',
 passphrase: 'top secret',
 },
}, (err, publicKey, privateKey) => {
 // Handle errors and use the generated key pair.
});
```

Listing 1. RSA key pair generation

The following 4 scenarios provide implementation guidelines that could help to understand the main idea.

Scenario 1: create a file with a sensitive information

1. Prior to storing of any secret, the owner needs to generate a unique symmetric key.
2. Owner stores encrypted file on a drive.
3. Owner stores encrypted symmetric key on a drive (Fig. 1).

The randomBytes() [6] routine allows to generate cryptographically strong pseudorandom data which is then used as a symmetric encryption key (Listing 2).

The createCipheriv() [7] is being used to create an encryption object which is then used to encrypt a file stream (Listing 3).

```
const { randomBytes } = require('node:crypto');
const symmetricEncryptionKey = randomBytes(32);
. . .
```

Listing 2. Symmetric key generation

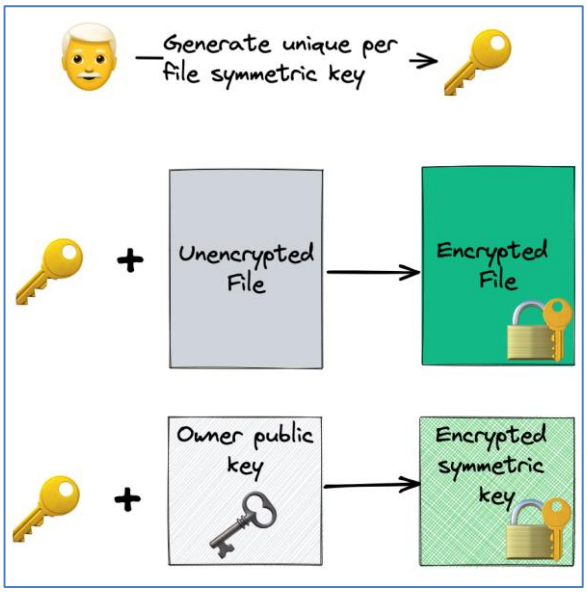

Fig. 1. Encrypted file creation flow

```
const { createCipheriv } = require('node:crypto');
const { createWriteStream } = require('node:fs');
const busboy = require('busboy');
const
INITIALIZATION_VECTOR="23ca408b8ff898bb" //
16 bytes
const bb = busboy({ headers: req.headers });
bb.on("file", async (_, file, { filename }) => {
  const symmetricEncryptionKey = randomBytes(32);
  const cipher = createCipheriv(
   'aes-256-cbc',
   symmetricEncryptionKey,
   INITIALIZATION_VECTOR
  );
  const stream = createWriteStream(filePath);
  file.pipe(cipher).pipe(stream);
  . . .
```

Listing 3. Create encrypted files stream

The publicEncrypt() [8] is being used to encrypt file's symmetric encryption key (Listing 4).

```
const { publicEncrypt } = require('node:crypto');
const { writeFile } = require('node:fs');

const encryptedKey = publicEncrypt(publicKey,
fileSymEncryptionKey)
writeFile(pathToEncryptedKey,
encryptedKey.toString('hex'))
. . .
```

Listing 4. symmetric key encryption

Scenario 2: decrypt file content

1. User needs a private key to decrypt symmetric encryption key.
2. With help of symmetric encryption key user could decrypt encrypted file content (Fig. 2).

The privateDecrypt() [9] is being used to get file's symmetric encryption key (Listing 5). The createDecipheriv() [10] is being used to create a decryption object which then sends decrypted data to an output stream (Listing 6).

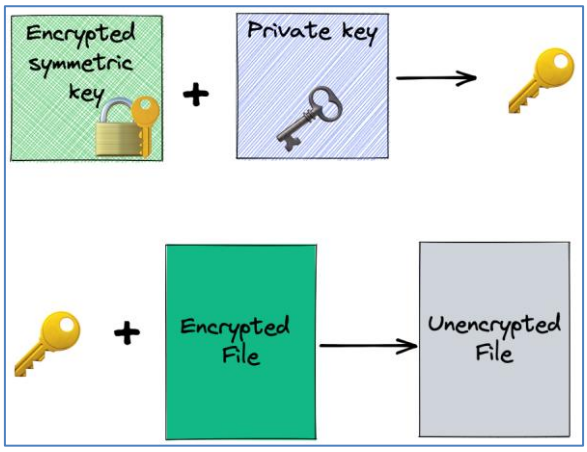

Fig. 2. File decryption flow

```
const { privateDecrypt } = require('node:crypto');
const encryptedKey = Buffer.from(rawFileData, 'hex')
const fileSymEncryptionKey = privateDecrypt({
  privateKey,
  Passphrase,
  },
encryptedKey);
. . .
```

Listing 5. Symmetric key decryption

```
const { createDecipheriv } = require('node:crypto');
const { pipeline, createReadStream } =
require('node:fs');
const
INITIALIZATION_VECTOR="23ca408b8ff898bb" //
16 bytes
const encryptedFileStream =
createReadStream(filePath);
const decipher = createDecipheriv(
  'aes-256-cbc',
  symmetricEncryptionKey,
  INITIALIZATION_VECTOR);
pipeline(encryptedFileStream, decipher,
outoputStream, (error) => {
. . .
```

Listing 6. Decrypt file stream

Scenario 3: share a file with a sensitive information

1. File owner decrypts symmetric encryption key.
2. File owner encrypts symmetric encryption key with user's public key.
3. File owner adds to the storage a new file with encrypted symmetric key.
4. Only user could decrypt symmetric key with its private key and then decrypt file content (Fig. 3).

Important to mention that the encrypted file remains intact. The file storage just got a tiny extra file with encrypted symmetric key

Scenario 4: revoke access to a file with a sensitive information

1. Owner removes encrypted symmetric key of the user.
2. User without a key cannot decrypt file content.
3. Encrypted file remains intact (Fig. 4).

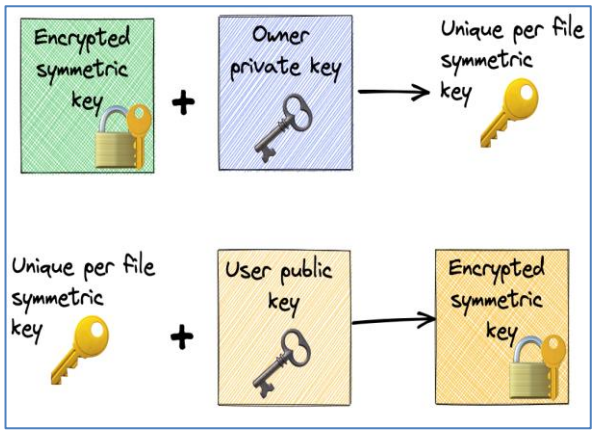

Fig. 3. Encrypted file sharing flow

Conclusion

We have proposed an enhanced authorization approach. Idea behind it is not a new one, it has already been used for decades in EFS [11] in Microsoft Windows. It has proven to be reliable over the years. Since the file encryption is based on the symmetric cryptography, it allows us to encrypt/decrypt files at a high speed. Moreover, modern CPUs for more than a decade have had a hardware acceleration [12] for the AES [13] algorithm. The biggest drawback of the approach is the recovery process.

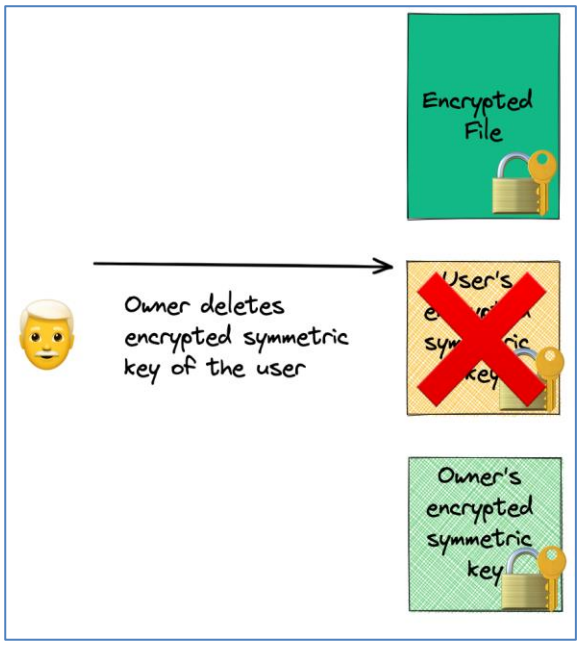

Fig. 4. Revoking file access flow

As a developer of an application, you need to take care of the recovery key option and the organization should define security policy for creation and storing of the recovery keys.

REFERENCES

1. *Prototype Pollution*. Snyk. https://learn.snyk.io/lessons/prototype-pollution/javascript/
2. Г. В. Головко. *Конспект лекцій з дисципліни "Захист інформації в комп'ютерних системах і Кібербезпека"* (2021). Національний університет «Полтавська політехніка імені Юрія Кондратюка». https://dist.nupp.edu.ua/mod/resource/view.php?id=122282
3. *Harvest now, decrypt later*. Wikipedia. https://en.wikipedia.org/wiki/Harvest_now,_decrypt_later
4. *Crypto package*. Node.js. https://nodejs.org/api/crypto.html#crypto
5. *crypto.generateKeyPair()* Node.js. https://nodejs.org/api/crypto.html#cryptogeneratekeypairtype-options-callback
6. *crypto.randomBytes()*. Node.js. https://nodejs.org/api/crypto.html#cryptorandombytessize-callback
7. *crypto.createCipheriv()*. Node.js. *Crypto package*. Node.js. https://nodejs.org/api/crypto.html#cryptocreatecipherivalgorithm-key-iv-options
8. *crypto.publicEncrypt()*. Node.js. https://nodejs.org/api/crypto.html#cryptopublicencryptkey-buffer
9. *crypto.privateDecrypt()*. Node.js. https://nodejs.org/api/crypto.html#cryptoprivatedecryptprivatekey-buffer
10. *crypto.createDecipheriv()*. Node.js. https://nodejs.org/api/crypto.html#cryptocreatedecipherivalgorithm-key-iv-options
11. *Encrypting File System*. Wikipedia. https://en.wikipedia.org/wiki/Encrypting_File_System
12. *AES instruction set*. Wikipedia. https://en.wikipedia.org/wiki/AES_instruction_set
13. *Advanced Encryption Standard*. Wikipedia. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

Надійшла (received) 22.02.2023
Прийнято до друку (accepted for publication) 16.05.2023

Розширена авторизація для безпечного керування конфіденційними даними в гібридних програмах

Г. Головко, Д. Ієвлєв

Анотація. Конфіденційними даними часто керують хмарні додатки, які можуть бути вразливими для зловмисників, які прагнуть несанкціонованого доступу до цих даних. Традиційних підходів до авторизації може бути недостатньо для захисту конфіденційних даних від таких атак. У цій статті ми пропонуємо розширений підхід авторизації, який використовує комбінацію симетричної та асиметричної криптографії для захисту конфіденційних даних. Зокрема, ми пропонуємо створити унікальний ключ шифрування для кожного файлу та набір відкритих і закритих ключів для кожного користувача, які використовуються для шифрування та дешифрування даних. Ми демонструємо здійсненність нашого підходу на прикладах у Node.js, показуючи, як генерувати відкритий і закритий ключі, шифрувати та розшифровувати файли та зберігати зашифровані дані на диску. Наш підхід забезпечує ефективне вирішення проблеми управління конфіденційними даними в гібридних програмах, зберігаючи при цьому зручність для користувачів і розробників.

Ключові слова: криптографія, криптографія з відкритим ключем, авторизація, Node.js.