ZADANIE Č.3

I-UPB

Vladimír Hlačina
Fakulta elektrotechniky a informatiky
Slovenská technická univerzita
xhlacina@stuba.sk

Rudolf Bachorík
Fakulta elektrotechniky a informatiky
Slovenská technická univerzita
xbachorik@stuba.sk

Jakub Zigo
Fakulta elektrotechniky a informatiky
Slovenská technická univerzita
xzigo@stuba.sk

Filip Hlavačka
Fakulta elektrotechniky a informatiky
Slovenská technická univerzita
xhlavackaf@stuba.sk

Igor Kuzmin
Fakulta elektrotechniky a informatiky
Slovenská technická univerzita
xkuzmini@stuba.sk

22. Október 2023

ABSTRAKT

Cieľom zadania je implementovať základnú štruktúru webovej aplikácie (podľa konkrétnej témy) s dôrazom na šifrovanie/dešifrovanie súborov. Webovú aplikáciu je potrebné implementovať prostredníctvom platformy Docker.

Informácia ku prácam na projekte

Na tomto projekte sa nepodieľali všetci členovia tímu rovnako. Hlavnú časť implementácie realizoval kolega Rudolf Bachorík. Ďalej sa na implementácii podieľal kolega Jakub Zigo. Spracovanie kompletnej dokumentácie, ako aj prípríprávu databázy zabezpečil Vladimír Hlačina. Ostatný členovia sa na tomto projekte aktívne nepodieľali ani po viacnásobných výzvach.

1 Výber technológií

V tejto časti popíšeme technológie, ktoré sme použili pri implementácii zadania. Pre implementáciu backendu sme sa rozhodli použiť jazyk `Python v.3.11-alpine` v kombinácií s frameworkom `Flask v3.0.0`. `Flask` nám umožňuje jednoduchú implementáciu `REST API`, ktoré sme použili na komunikáciu s frontendom. Pre implementáciu frontendu sme sa rozhodli použiť framework `React v17.0.2`. `React` nám umožňuje implementáciu webovej aplikácie za pomoci jazyka `JavaScript`. Implementácia databázy bola realizovaná pomocou relačnej databázy `MySQL v.2.0.0` [5, 3, 4].

2 Implementácia docker balíčka

Pre docker balíček sme zvolili štruktúru kedy v jednom containery bežia všetky potrebné komponenty. Tento prístup zaručuje jednoduché vytvorenie a spustenie docker balíčka. Všetky potrebné komponenty sú spustiteľné pomocou CLI príkazu `docker-compose up`. Ako príklad uvádzame skrátenú verziu docker-compose.yml súboru, ktorá spúšťa jednotlivé časti implementácie a na ich základe vytvára kontajnery.

```
    services:
      api:
        build: api
        ...
      client:
        build: client
        ...
      mysql:
        image: mysql:latest
        ...
```

3 Implementácia databázy

Databáza bola navrhnutá na základe internej konzultácie v tíme a potrebám špecifikácie zadaného systému. V aktuálnej

id - PK	name	surname	arrival	leave	vacation_days	sick_days
INT, NOT NULL AUTO INCREMENT	VARCHAR(45) NOT NULL	VARCHAR(45) NOT NULL	VARCHAR(45) NOT NULL	VARCHAR(45) NOT NULL	INT NULL	INT NULL

forme boli pre uloženie dátumov použité prevažne typy `VARCHAR`. Toto riešenie neovplyvňuje funkcionalitu aplikácie, avšak v budúcnosti by bolo vhodné tieto typy dátumov pretypovať na typ `DATE` alebo `TIMESTAMP`. Nastavenia databázy sú uvedené v tabuľke 1.

Parameter	Hodnota
Host	mysql
User	admin
Password	admin
Database	mydb

Tab. 1: Nastavenia databázy

4 Implementačné úlohy

4.1 Extrakcia testovacích dát

Na extrakciu dát sme použili databázové API `flask_mysql` pre knižnicu Flask. Implementácia tejto sekcie bola realizovaná pomocou triedy `app.py` za pomoci nasledovnej funkcie.

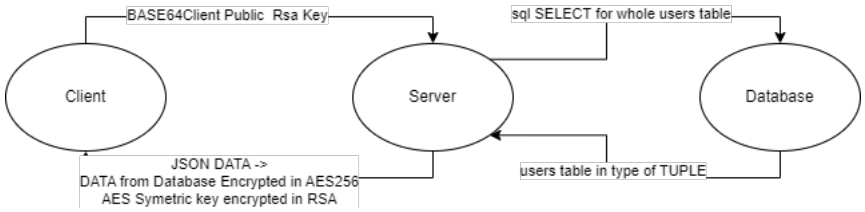
```
def getDataFromDatabase():
    cur = mysql.connection.cursor()
    cur.execute("""SELECT * FROM users""")
    rv = cur.fetchall()
    return rv
```

4.2 Zašifrovanie dát pomocou symetrickej šifry

Na vytvorenie šifry sme využili generovanie 256 náhodných bitov pomocou metódy `os.urandom(32)` na strane servera. Veľkosť kľúča sme zvolili z dôvodu, že AES-256 je aktuálne jedna z najbezpečnejších šifier, ktoré sú v súčasnosti dostupná. V praxy existuje aj AES-512, ktorej rýchlosť je však výrazne pomalšia ako AES-256, čo značne spomaľuje enkryptovanie a dekryprovanie počas komunikácie.

```
def encryptAES(key, iv, data):
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv))
    encryptor = cipher.encryptor()
    ct = encryptor.update(data) + encryptor.finalize()
    return ct
```

Vstupnými parametrami sú symetrický kľúč, inicializačný vektor s veľkosťou 128 bitov a dáta z databázy, ktoré chceme šiftovať. Metóda Cipher zo spomínanej knižnice zabezpečuje vytvorenie logiky pre použitie šifry. Jedným z parametrov tejto metódy je `modes`, kde sme zvolili nastavenie `CFB`. Tento mód má jednoduchšiu a rýchlejšiu dekrypciu oproti `CBC` módu a nevyžaduje autorizáciu ako pri móde `GCM`, ktorú nepotrebujeme vďaka šifrovaniu pomocou `RSA`, čo pri našej implementácií a stupni senzitivity dát je dostačujúce. Následne sa vytvorí enkryptor, ktorým dáta zašifrujeme. Výstupným parametrom sú zašifrované dáta [1]. Pre lepšiu vyzualizáciu implementácie sme uviedli aj UML diagram requestu dát z databázy 1.



Obr. 1: UML diagram requestu dát z databázy a aplikácia šifrovania/dešifrovania

4.3 Zašifrovanie symetrického kľúča asymetrickým kľúčom používateľa

Pomocou knižnice `cryptography` sme vytvorili privátny a verejný kľúč. Privátny kľúč o veľkosti 2048 bitov a exponentom o veľkosti 65537 je uložený v súbore `Private_Server_KEY.pem`. Exponent sme zvolili ako Fermatove číslo, ktoré je všeobecne odporúčané pre rýchlosť operácií s daným kľúčom. V dnešnej dobe je síce veľkosť kľúča pre RSA bežne 4096 bitov napriek tomu sme ju v našom riešení nepoužili, pretože by kalkulácia kľúča trvala dlhšie. S rovnakým problémom by sme sa stretli aj pri enkryptovaní a dekryptovanií dát. Tento kľúč by nemusel byť kompatibilný s niektorými staršími zariadeniami, ktoré nie sú schopné spracovať kľúče väčšie ako 2048 bitov čo by spôsobilo problém vo firmách, ktoré sú našimi poteniálnymi zákazníkmi. Ďalej sme z privátneho kľúča vytvorili verejný kľúč a uložili do súboru `Public_Server_KEY.pem`. Rovnakým spôsobom sme vytvorili privátny a verejný kľúč klienta [1].

```
public_key = private_key.public_key()
pem1 = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)
```

4.4 Implementácia dešifrovania súboru

Prvým krokom k dešifrovaniu súboru je zašifrovanie jeho obsahu. Na zašifrovanie obsahu sme zvolili symetrický algoritmus so zdielaným tajomstvom `Fernet`. Tento algoritmus je založený na symetrickom šifrovaní AES a HMAC autentifikácií. Na zašifrovanie súboru používa aktuálny `Timestamp`, inicializačný vektor o veľkosti 128 bitov a tajomstvo vytvorené z kľúča. V aktuálnej implementácií bolo možné použitie AES-256, rovnako ako pri symetrickom šifrovaní. Avšak sme sa rozhodli o vyššie spomínané šifrovanie, aby sme odlíšili šifrovanie dát z databázy a šifrovanie súborov. Štruktúra šifrovaného súboru je základný textový súbor vo formáte `.txt`. Na nasledovnom príklade je zobrazený kód použitý pre šifrovanie súboru [2].

```
key = Fernet.generate_key()
f = Fernet(key)
with open(file_path, "rb") as file:
    bytes_file = file.read()  # read entire file as bytes
encrypted = f.encrypt(bytes_file)
```

Po úspešnom zašifrovaní súboru sme sa rozhodli dodržať platné štandardy a dešifrovať súbor na strane klienta. Kľúč je doručený klientovi zašifrovaný pomocou verejného kľúča klienta na strane servera. Týmto krokom zabezpečujeme bezpečnosť posielania. Na dešifrovanie súboru na strane klienta sme najskôr dešifrovali symetrický kľúč s ktorého pomocou sme následne dešifrovali obsah súboru. Nasledovná časť našej implementácie zobrazuje dôležité časti spôsobu odšifrovania obsahu súboru na strane klienta [1, 2].

```
1  //Decrypt Fernet encrypted key in RSA from server var decryptedFernetKey =
2  privateKey.decrypt(encryptedFernetKey, 'RSA-OAEP', { md:
3  forge.md.sha256.create(), }); decryptedFernetKey =
4  forge.util.decodeUtf8(decryptedFernetKey) var secret = new
5  fernet.Secret(decryptedFernetKey); var token = new fernet.Token({ secret:
6  secret, token: encryptedFernetData }) let decryptedData = token.decode()
```

Na riadku 2 môžeme vidiet, že prvým krokom je dešifrovanie symetrického kľúča pomocou privátneho kľúča klienta. Šifrované dáta boli obohatené o `OAEP` padding, ktorý pridáva náhodnosť za pomoci hešovacích a xor operácií čo je potrebné zohľadniť pri dešifrovaní. Následne kľúč dekódujeme do formátu `UTF-8` a vytvoríme nový objekt typu `Secret`, ktorý je potrebný pre dešifrovanie obsahu súboru. Ďalej vytvoríme nový objekt typu `Token`, ktorý obsahuje enkódovaný obsah súboru a dekódovaný symetrický kľúč. Na záver dekódujeme obsah súboru a získame pôvodný obsah súboru [1, 2].

4.5 Implementácia overenia integrity súboru

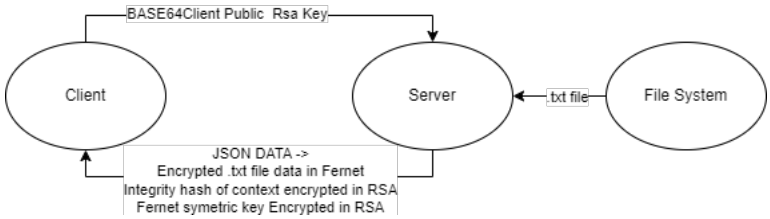
Overenie integrity súboru sme realizovali na základe overenia hashu súboru. Na vytvorenie hashu sme použili hash typu `sha256`, ktorý je často používaný na overenie integrity súborov. Vytvorenie hashu spočíva v samotnom vygenerovaní na strane servera pomocou funkcie `forge.md.sha256.create()` pred odoslaním dát. Na strane klienta sa hash vytvorí pomocou rovnakej funkcie a následne sa porovná s hashom odoslaným zo servera. Na nasledovnej ukážke je zobrazený kód použitý pre vytvorenie hashu súboru na strane servera [1, 2].

```
1  # Read entire file as bytes
2  with open(file_path,"rb") as f: bytes = f.read()
3      readable_hash = hashlib.sha256(bytes).hexdigest().encode()
```

Na riadku 2 môžeme vidieť, že najskôr sa súbor načíta ako byty a následne sa vytvorí hash pomocou funkcie `hashlib.sha256()`. Na riadku 3 sa hash prevedie do formátu `hex` a následne sa uloží do premennej `readable_hash`. Takto vytvorený hash je následne odoslaný klientovi enkódovaný vo pomocou `base64` enkódingu. Následne sme uviedli časť kódu, ktorý zodpovedá za overenie integrity súboru na strane klienta [1, 2].

```
1  let decryptedData = token.decode()
2  var md = forge.md.sha256.create()
3  md.update(decryptedData)
4  let clientSideIntegrity = md.digest().toHex()
5  setIntegrityHashClient(clientSideIntegrity)
6  if (clientSideIntegrity == integrityHash) {
7      setIsIntegrityKept("true") }
8  else { setIsIntegrityKept("false") }
```

Vyššie uvedená časť kódu je zodpovedná za vytvorenie hashu súboru na strane klienta. Na riadku 2 môžeme vidieť, že najskôr sa vytvorí hash pomocou funkcie `forge.md.sha256.create()`. Následne sa hash updatuje pomocou funkcie `md.update()` a nakoniec sa hash prevedie do formátu `hex` pomocou funkcie `md.digest().toHex()`. V poslednej časti kódu sa porovná hash vytvorený na strane klienta a hash odoslaný zo servera. Ak sa tieto dva hashe zhodujú, tak sa nastaví premenná `isIntegrityKept` na hodnotu `true`, inak sa nastaví na hodnotu `false` a vyhodnotí jeho integrita. Pre lepšiu vizualizáciu riešenia sme vytvorili UML diagram 2.



Obr. 2: UML diagram requestu dát zo súboru a aplikácia šifrovania/dešifrovania

Literatúra

[1] cryptography: High level recipes and low level interfaces to common cryptographic algorithms. https://cryptography.io/en/latest/#.

[2] Fernet: Symmetric encryption algorithm. https://github.com/fernet.

[3] Flask: Backend framework for python. https://flask.palletsprojects.com/en/3.0.x/#.

[4] Mysql: The world's most popular open source database. https://www.mysql.com/.

[5] React: The library for web and native user interfaces. https://react.dev/reference/react.