UPB  
Zadanie 2 - Kryptografia

Riešitelia: Emília Čurillová, Filip Harvančík, Lenka Puškášová

**Úloha1 (Filip 25%, Lenka 75%)**

Pre tento projekt bude použití knižnica cryptography z nasledujúcich dôvodov

* Je aktívne vyvíjaná a údržovaná a reaguje na bezpečnostné zraniteľnosti.
* Je široko používaná v Python ekosystéme a odporúčaná komunitou.
* Dokumentácia otvorene uvádza zoznam známych zraniteľností a používa externé databázy (napr. OSV) na ich sledovanie.

Inštalácia: **pip install cryptography**

Zdroje:

<https://cryptography.io/en/latest/security>

**Úloha2 (Lenka)**

Pre Generovanie asymetrického páru kľúčov bude použitý Rivest-Shamir-Adleman (RSA) algoritmus z dôvodu jeho rozšíreného použitia, robustnosti a bezpečnosti

Použitá knižnica: werkzeug.security, ktorá je súčasťou široko používanej Flask knižnice

**Postup vytvorenia páru kľúčov:**

1. Vytvorenie private key
   1. bude použitá funkcia **rsa.generate\_private\_key(**public\_exponent, key\_size)
      1. **public\_exponent** = [65537](https://www.daemonology.net/blog/2009-06-11-cryptographic-right-answers.html) – nie je to nevyhnutne ale tato hodnota je odporúčaná v dokumentácii z legacy dôvodov
      2. **key\_size** = 2048 – odporúčaná minimálna hodnota pre veľkosť kľúča
2. Odvodenie public key z private key
   1. **private\_key.public\_key()**
3. Serializácia kľúčov – kľúče sú uložené ako RSAPrivateKey a RSAPublicKey objekty. Treba ich serializovat, aby mali bajtový format a mohli byť uložené
   1. **private\_key.private\_bytes**( encoding, formate, encryption\_algorithm)
      1. encoding – PEM (base64)
      2. formate – PKCS8 (štandardne používaný format pre privátne kľúče)
      3. encryption\_algorithm - serialization. NoEncryption()
   2. public\_key.public\_bytes(encoding, formate)
      1. encoding – PEM
      2. formate – SubjectPublicKeyInfo – najčastejšie používaný pre verejné kľúče
4. Vytvorenie užívateľa – do db sa bude pridávať nový užívateľ (alebo upravovať už existujúci) s uživateľským menom user, heslom password a s verejným heslom public\_key (využije sa metóda decode na prevod z bajtov na string typ)
5. V api volaní sa vykoná generovanie privátneho a public kľúča. Nastane serializácia a vytvorí sa nový užívateľ spolu s public key v db tabuľke. Následne metoda vracia odpoved v súbore s využitím Content-Disposition v hlavičke odpovedi

Zdroje:  
RSA algoritmus:   
https://www.ibm.com/think/topics/asymmetric-encryption  
Privátny kľúč: <https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#cryptography.hazmat.primitives.asymmetric.rsa.generate_private_key>  
Serializácia kľúčov:  
<https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/#serialization-of-private-keys>  
Serialization encoding  
<https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/#serialization-encodings>  
Serialization formats  
<https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/#serialization-formats>  
Serialization Encryption Types  
<https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/#serialization-encryption-types>  
sqlalchemy ORM  
<https://docs.sqlalchemy.org/en/20/tutorial/orm_data_manipulation.html>  
Content-disposition header  
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Disposition>

**Úloha 3 (Emília)**

1. Prijatie požiadavky a načítanie obsahu súboru
2. Načítanie vejerného kľúča z DB pre daného používateľa
3. Deseralizácia verejného kľúča
   * Verejný kľúč načítaný z databázy je v textovom PEM formáte (string). Pre použitie v kryptografických operáciách je potrebné ho deserializovať do objektu verejného kľúča pomocou funkcie serialization.load\_pem\_public\_key(). Pred deserializáciou je potrebné string skonvertovať na bytes pomocou metódy .encode()
4. Vygenerovanie náhodného 256-bitového symetrického kľúča pre AES
   * Server vygeneruje kryptograficky bezpečný náhodný 32-bytový (256-bitový) kľúč pomocou funkcie os.urandom(32). Tento kľúč bude použitý na šifrovanie obsahu súboru. Funkcia os.urandom() využíva generátor náhodných čísel operačného systému, ktorý je vhodný pre kryptografické účely.
5. Generovanie 12-bytového IV pre GCM
   * Pre AES v GCM móde sa generuje 12-bytový (96-bitový) inicializačný vektor pomocou os.urandom(12). IV musí byť unikátny pre každé šifrovanie s rovnakým kľúčom. Dĺžka 12 bytov je optimálna pre GCM mód - zabezpečuje najlepší výkon a bezpečnosť.
6. Zašifrovanie obsahu súboru pomocou AES-256 v GCM móde
   * **Výber symetrickej šifry a parametrov:**

Vybrali sme si AES-256 (Advanced Encryption Standard s 256-bitovým kľúčom), pretože je toštandardizovaná šifra schválená NIST (National Institute of Standards and Technology), má veľmi vysokú bezpečnosť, je odolná voči všetkým známym útokom a 256-bitový kľúč poskytuje maximálnu bezpečnosť spomedzi AES variant (128, 192, 256 bitov)

* + GCM mód (Galois/Counter Mode):

AEAD (Authenticated Encryption with Associated Data) - poskytuje šifrovanie aj autentifikáciu v jednom kroku, automaticky zabezpečuje integritu dát - deteguje akúkoľvek modifikáciu šifrovaných dát, nepotrebuje padding - funguje s ľubovoľnou dĺžkou vstupných dát, navyše generuje authentication tag (16 bytov), ktorý slúži na verifikáciu integrity pri dešifrovaní, je to odporúčaný mód pre moderné aplikácie

* + Proces šifrovania:

Vytvorí sa šifrovací objekt pomocou Cipher() s algoritmom AES-256 a GCM módom. Metóda .encryptor() vráti encryptor objekt, ktorý postupne spracuje obsah súboru pomocou metód update() a finalize(). Výsledkom je zašifrovaný obsah (ciphertext).

1. (Získanie authentication tag)
   * Po dokončení šifrovania GCM mód automaticky vygeneruje 16-bytový authentication tag, ktorý je dostupný cez encryptor.tag. Tento tag slúži na verifikáciu integrity a autenticity dát pri dešifrovaní. Ak niekto zmení šifrované dáta, tag nebude validný a dešifrovanie zlyhá.
2. Šifrovanie symetrického kľúča pomocou RSA-OAEP
   * Asymetrické šifrovanie: Symetrický AES kľúč (32 bytov) sa zašifruje verejným RSA kľúčom používateľa pomocou padding schémy OAEP (Optimal Asymmetric Encryption Padding).
   * RSA-2048:Veľkosť kľúča: 2048 bitov (256 bytov), poskytuje dostatočnú bezpečnosť pre súčasné aj budúce použitie, odporúčaná minimálna veľkosť kľúča podľa bezpečnostných štandardov (NIST, BSI)
   * OAEP padding (s SHA-256):Zabezpečuje, že rovnaký plaintext sa zakaždým zašifruje inak
3. Formát zašifrovaného súboru

* 4 byty: dĺžka zašifrovaného kľúča
* 256 bytov: zašifrovaný symetrický kľúč (RSA-2048)
* 12 bytov: IV pre AES
* 16 bytov - GCM authentication tag
* N bytov: zašifrovaný obsah súboru

Zdroje: [Symmetric encryption — Cryptography 47.0.0.dev1 documentation](https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/#cryptography.hazmat.primitives.ciphers.Cipher.encryptor)

[Symmetric encryption — Cryptography 47.0.0.dev1 documentation](https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/#cryptography.hazmat.primitives.ciphers.algorithms.AES)

[Symmetric encryption — Cryptography 47.0.0.dev1 documentation](https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/#cryptography.hazmat.primitives.ciphers.algorithms.AES256)

[Symmetric encryption — Cryptography 47.0.0.dev1 documentation](https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/#module-cryptography.hazmat.primitives.ciphers.modes)

Anthropic (2025), Claude 4.5 Sonnet, generovanie textu

**Úloha 4 (Emília)**

1. Prijatie požiadavky a načítanie vstupov z multipart požiadavky
2. Načítanie obsahu zašifrovaného súboru pomoocu metódy .read(), ktorá vráti kompletné binárne dáta. Server vykoná základnú validáciu - súbor musí mať minimálne 4 byty (dĺžka zašifrovaného kľúča)
3. Deserializácia privátneho kľúča z PEM formátu. Pre použitie v kryptografických operáciách je potrebné ho deserializovať do objektu privátneho kľúča pomocou funkcie serialization.load\_pem\_private\_key(). Parameter password=None indikuje, že kľúč nie je chránený heslom (zodpovedá tomu, že pri generovaní kľúča v /api/gen sa použilo NoEncryption()
4. Parsovanie formátu zašifrovaného súboru:
   * Čítanie dĺžky zašifrovaného kľúča (4 byty):

Použije sa struct.unpack('<I', ...) na konverziu bytov na celé číslo, pre RSA-2048 je táto hodnota typicky 256 bytov. Premenná offset sleduje aktuálnu pozíciu čítania v súbore.

* + Čítanie zašifrovaného symetrického kľúča:

Zo súboru sa prečíta encrypted\_key\_length bytov obsahujúcich zašifrovaný AES kľúč a offset sa posunie o encrypted\_key\_length bytov

* + Čítanie IV (12 bytov):

Initialization Vector použitý pri šifrovaní AES-GCM musí byť identický s tým, ktorý bol použitý pri šifrovaní. 12 bytov je štandardná a optimálna veľkosť pre GCM mód.

* + Čítanie authentication tag (16 bytov):

Používa sa na overenie integrity a autenticity dešifrovaných dát. Bez správneho tagu nie je možné úspešne dešifrovať súbor.

* + Čítanie zašifrovaného obsahu:

Všetky zostávajúce byty v súbore predstavujú zašifrovaný obsah (ciphertext) a jeho dĺžka zodpovedá originálnemu súboru.

1. Dešifrovanie symetrického kľúča pomocou privátneho RSA kľúča:
   * Zašifrovaný symetrický kľúč sa dešifruje pomocou privátneho RSA kľúča používateľa. Používa sa rovnaká padding schéma OAEP (Optimal Asymmetric Encryption Padding) ako pri šifrovaní. Výsledkom je pôvodný 32-bytový (256-bitový) AES kľúč.
2. Dešifrovanie obsahu súboru pomocou AES-256-GCM:
   * Vytvorenie decryptor objektu:

Vytvorí sa dešifrovací objekt pomocou Cipher() s algoritmom AES-256 a GCM módom. Do GCM módu sa okrem IV musí explicitne poskytnúť aj authentication tag pomocou modes.GCM(iv, tag). Metóda .decryptor() vráti decryptor objekt pripravený na dešifrovanie.

* + Proces dešifrovania: Metódy update() a finalize() postupne spracujú zašifrovaný obsah a vrátia pôvodný plaintext.
  + Automatická kontrola integrity: GCM mód pri volaní finalize() automaticky overí authentication tag.

1. Vrátenie dešifrovaného súboru

**Úloha 5 (Filip)**

1. Požiadavka je typu HTTP POST a obsahuje multipart/form-data, v ktorej klient odosiela:

pôvodný súbor, ktorý chce podpísať (file),

privátny kľúč (key).

Na serveri tieto dáta získavame pomocou request.files.get('file') a request.files.get('key').  
Je dôležité skontrolovať, či tieto parametre skutočne prišli – ak chýbajú, vrátime chybovú odpoveď s kódom 400

1. Aby sme mohli vytvoriť digitálny podpis, potrebujeme pracovať s obsahom súboru v binárnej forme a mať k dispozícii privátny kľúč v objektovej podobe.

Súbor načítame pomocou .read()

Privátny kľúč, ktorý je odoslaný vo formáte PEM, musíme načítať pomocou funkcie serialization.load\_pem\_private\_key()

1. Keď máme obsah súboru a privátny kľúč, môžeme vytvoriť digitálny podpis.  
   Na to slúži metóda .sign(), ktorá:

vezme binárne dáta súboru,

použije špecifikovaný padding (PKCS1v15()),

a zvolenú hashovaciu funkciu (SHA256).

Výsledkom je podpis v binárnej podobe, ktorý je viazaný na obsah súboru a privátny kľúč. Ak by sa súbor zmenil alebo by sme použili iný kľúč, podpis by nebol overiteľný.

1. Ak podpisovanie prebehne úspešne, server vráti binárny podpis ako HTTP odpoveď so správne nastavenými hlavičkami:

Content-Type: application/octet-stream – označuje, že ide o binárny súbor,

Content-Disposition: attachment; filename=signature.bin – spôsobí, že klient si podpis stiahne pod týmto názvom.

**Úloha 6 (Filip)**

1. Overenie podpisu začína prijatím POST požiadavky na endpoint /api/verify/<user>, kde <user> je identifikátor používateľa, pre ktorého podpis overujeme. Požiadavka obsahuje:

pôvodný súbor (file), ktorý bol podpísaný,

samotný digitálny podpis (signature), ktorý má byť overený.

Dáta získavame cez request.files.get(). Ak niektorý parameter chýba, vrátime chybu 400

1. Na overenie podpisu potrebujeme:

obsah pôvodného súboru (ten musí byť presne rovnaký ako pri podpise),

samotný podpis v binárnej podobe.

Pomocou .read() načítame tieto dáta do premenných

1. Na overenie podpisu potrebujeme verejný kľúč, ktorý zodpovedá privátnemu kľúču, ktorým bol súbor podpísaný. Tento kľúč získame z databázy podľa identifikátora používateľa <user>. Verejný kľúč je uložený vo formáte PEM, preto ho musíme previesť na objekt pomocou serialization.load\_pem\_public\_key().
2. Kľúčový krok je overenie podpisu pomocou metódy .verify(). Táto metóda porovná podpis, obsah súboru a verejný kľúč. Ak všetko súhlasí, overenie prebehne bez chyby. Ak podpis nesedí (napr. bol pozmenený súbor, iný podpis alebo kľúč), vyhodí výnimku InvalidSignature
3. Ak .verify() neskončí výnimkou, znamená to, že podpis je platný. Server následne vráti JSON odpoveď s výsledkom overenia

**Úloha 7 (Filip)**

Táto úloha bola veľmi podobná úlohám 3, 4, preto nepôjdem do hĺbky o jej fungovaní.

Mechanizmus ochrany integrity je založený na:

* Autentizačnom tagu AES-GCM, ktorý zaručuje, že obsah nebol po šifrovaní zmenený.
* Použití encrypted\_key ako AAD, čo navyše zabezpečuje, že obsah súboru je pevne zviazaný s konkrétnym symetrickým kľúčom.
* Akákoľvek manipulácia s ktoroukoľvek časťou šifrovaného súboru znamená okamžité zlyhanie dešifrovania, keďže autentizačné tagy sa nebudú zhodovať.