Lucrare practică / Lucrare de laborator nr. 3

Tema: Securitatea software prin criptarea datelor – implementări în C/C++, Java, C# și Python

Introducere

Aplicațiile mobile manipulează frecvent date sensibile (date personale, token-uri de autentificare, informații financiare etc.), de aceea este esențial ca aceste date să fie protejate atât în tranzit (când sunt transmise între dispozitiv și server), cât și în repaus (stocate pe dispozitiv sau pe server).

Criptografie puternică și corectă: Atât pentru datele în tranzit cât și pentru cele în repaus, folosirea criptografiei este eficientă doar dacă se aplică algoritmi și implementări solide. Algoritmi slabi sau implementați necorespunzător pot crea un fals sentiment de securitate. De exemplu, utilizarea unui algoritm de criptare nesigur sau a unei chei prea scurte poate fi la fel de riscantă ca lipsa oricărei criptări. OWASP a introdus categoria *Insufficient Cryptography* pentru cazurile în care dezvoltatorii încearcă să cripteze, dar o fac greșit, de exemplu, folosesc un mod de operare nesigur sau o cheie hardcodată comună pentru toți utilizatorii. Recomandarea este să se folosească algoritmi standard, implementați de biblioteci consacrate ale platformei, în loc să se încerce "reinventarea roții" criptografice.

Lipsa criptării sau implementarea slabă a acesteia a fost menționată de experți drept una dintre principalele amenințări la adresa aplicațiilor mobile. O aplicație sigură va transmite și stoca datele astfel încât, chiar dacă comunicațiile sunt interceptate sau dispozitivul ajunge pe mâna altcuiva, informațiile să rămână confidențiale.

Criptare simetrică: În criptarea simetrică, același secret *cheie* este folosit atât pentru criptare, cât și pentru decriptare. Algoritmi ca AES (Advanced Encryption Standard) și (istoricul) DES (Data Encryption Standard) fac parte din această categorie. AES operează pe blocuri de 128 biți (poate folosi chei de 128, 192 sau 256 de biți) și este **standardul actual în industrie**, considerat extrem de sigur la atacuri brute-force. DES, în schimb, este un algoritm vechi de 56 biți, acum retras de NIST din cauza vulnerabilității la atacuri brute-force (Triple-DES, succesorul său, este și el în curs de retragere).

Avantajele criptării simetrice includ viteză mare și eficiență: algoritmii simetrici sunt rapizi și potriviți pentru volume mari de date. **Dezavantajul** major este necesitatea distribuirii **secrete** a cheii către toți participanții autorizați – dacă cheia este compromisă, securitatea dispare. Simetric este utilizat în practică pentru criptarea datelor în timp real sau a fișierelor mari (ex: criptarea discului, comunicarea în rețea după stabilirea unei chei secrete).

Criptare asimetrică: În criptarea asimetrică (criptografie cu chei publice), se folosește o pereche de chei: o cheie publică (pentru criptare) și o cheie privată (pentru decriptare). Algoritmi populari sunt RSA (Rivest-Shamir-Adleman) și ECC (Elliptic Curve Cryptography). Mesajele criptate cu cheia publică pot fi decriptate **doar** cu cheia privată corespunzătoare, care este ținută secretă de proprietar. Acest model elimină necesitatea distribuirii unei chei secrete unice și permite comunicare securizată cu oricine are cheia publică. RSA își bazează securitatea pe dificultatea factorizării numerelor mari – cheile RSA au de obicei 2048 sau 3072 de biți. ECC se bazează pe probleme matematice ale curbelor eliptice și oferă un nivel echivalent de securitate cu chei mult mai scurte; de exemplu, ECC (256 biți) oferă o securitate comparabilă cu RSA de 3072 biți. Un **avantaj** al criptării asimetrice este scalabilitatea – putem distribui liber cheia publică. **Dezavantajul** principal este performanța: algoritmii asimetrici sunt semnificativ mai lenți și consumatori de resurse față de cei simetrici. De aceea, în practică, asimetricul se folosește pentru cantități mici de date (ex: schimb de chei, certificate digitale, semnături), nu pentru criptarea întregului trafic. Un scenariu comun este criptarea **hibridă**: se generează o cheie simetrică pentru sesiune, folosită cu AES (rapid), iar această cheie de sesiune este la rândul ei criptată cu RSA/ECC (lent, dar practic pentru chei scurte) și transmisă destinatarului. Astfel se combină avantajele ambelor metode.

Rezumat diferențe și utilizări: Criptarea simetrică folosește o singură cheie secretă comună (ex: AES, DES), oferind viteză mare – potrivită pentru criptarea bulk (fișiere, baze de date, comunicații după stabilirea unei sesiuni). Criptarea asimetrică folosește pereche de chei (publică/privată, ex: RSA, ECC), permițând schimb securizat fără a împărtăși o cheie secretă în prealabil – utilizată pentru distribuirea cheilor simetrice, autentificare (semnături digitale) și stabilirea de conexiuni sigure (ex: TLS/SSL). În practică, aceste metode se complementează: asimetricul asigură schimbul sigur al unei chei simetrice, după care comunicarea efectivă este criptată simetric.

Criptarea datelor – implementări în C/C++, Java, C# și Python

> Vom ilustra cum se poate implementa criptarea/decreptarea simetrică **AES (Advanced Encryption Standard)** și asimetrică cu **RSA (Rivest-Shamir-Adleman)** în diferite limbaje de programare folosite a la dezvoltarea aplicațiilor sau componentelor server pentru mobil. Scopul este de a evidenția existența bibliotecilor de criptografie și simplitatea relativă a utilizării lor corecte, comparativ cu tentația periculoasă de a implementa propriile algoritmi.

1. C și C++ (Utilizând OpenSSL)

1.1 Criptare simetrică **AES** (modul ECB)

În C și C++, cea mai folosită bibliotecă pentru criptografie este **OpenSSL**, care oferă API-uri pentru AES. Exemplul de mai jos prezintă criptarea unui text folosind AES-128 în modul ECB (Electronic Codebook) cu OpenSSL:

```
c
#include <openssl/aes.h>
#include <string.h>
int main() {
    // Cheie de 128 biți (16 octeți)
    unsigned char key[16] = {
        0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,
        0x88,0x99,0xAA,0xBB,0xCC,0xDD,0xEE,0xFF
    };
    // Textul clar (exemplu)
    unsigned char plaintext[] = "MESAJ_SECRET!";
    // Buffer pentru textul criptat (aceeași dimensiune cu
    // plaintext rotunjită la 16 bytes)
    unsigned char ciphertext[16];

    AES_KEY enc_key;
    // Inițializare cheie de criptare AES (128 biți)
    AES_set_encrypt_key(key, 128, &enc_key);
    // Criptarea unui bloc de 16 octeți
    AES_encrypt(plaintext, ciphertext, &enc_key);

    // La acest punct, ciphertext conține textul criptat în
    // format binar.
    // (Într-o utilizare reală, probabil am converti
    //ciphertext la hex sau base64 pentru a fi
    transmis/stocat.)
    return 0;
}
```

Explicații: În exemplu, definim o cheie de 128 de biți și un text clar. Folosim `AES_set_encrypt_key` din OpenSSL pentru a pregăti cheia de criptare, apoi `AES_encrypt` pentru a cripta un bloc de date. Trebuie notat că modul ECB folosit aici este nu recomandat în practică pentru date mai mari, deoarece criptează bloc cu bloc fără amestec, putând lăsa tipare detectabile în textul cifrat. În codul real, ar trebui să utilizăm un mod mai sigur, precum CBC (Cipher Block Chaining) împreună cu un vector de inițializare (IV) aleator:

1.2 Criptare și decriptare AES (mod CBC)

```
c
#include <openssl/aes.h>
#include <openssl/rand.h>
#include <stdio.h>
#include <string.h>
int main() {
    // Cheie și vector de inițializare
    unsigned char key[AES_BLOCK_SIZE];
    unsigned char iv[AES_BLOCK_SIZE];
    RAND_bytes(key, sizeof(key));
    RAND_bytes(iv, sizeof(iv));
    // Datele de criptat
    unsigned char text[] = "Mesaj secret";
    unsigned char enc_out[sizeof(text)];
    unsigned char dec_out[sizeof(text)];

    // Context de criptare
    AES_KEY enc_key, dec_key;
    AES_set_encrypt_key(key, 128, &enc_key);
    AES_set_decrypt_key(key, 128, &dec_key);
    // Criptare
    AES_cbc_encrypt(text, enc_out, sizeof(text), &enc_key,
iv, AES_ENCRYPT);
    // Decriptare
    AES_cbc_encrypt(enc_out, dec_out, sizeof(enc_out),
&dec_key, iv, AES_DECRYPT);
```

```
        // Output
        printf("Text original: %s\n", text);
        printf("Text criptat: %s\n", enc_out);
        printf("Text decriptat: %s\n", dec_out);
        return 0;
    }
```

1.3 Criptare simetrică AES -CBC: Exemplu de folosire a OpenSSL EVP pentru a cripta și decripta un mesaj cu AES-256-CBC (vom folosi API-ul de nivel înalt EVP pentru a simplifica operațiile criptografice):

```
c
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <string.h>

// ... (cod de inițializare OpenSSL, dacă e necesar)

unsigned char key[32];  // cheia de 256 biți (32 bytes) - în
practică, generați random
unsigned char iv[16];   // IV de 128 biți pentru CBC (16 bytes)
- generați random

EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);  //
inițializează contextul de criptare AES-256-
CBC&#8203;:contentReference[oaicite:8]{index=8}

// Criptare:
int len;
int ciphertext_len;
EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext,
plaintext_len);  // criptează
blocuri&#8203;:contentReference[oaicite:9]{index=9}
ciphertext_len = len;
```

```
EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);  //
finalizează, tratează
padding&#8203;:contentReference[oaicite:10]{index=10}
ciphertext_len += len;
EVP_CIPHER_CTX_free(ctx);
```

În exemplul de mai sus, `plaintext` este un buffer de input (de lungime `plaintext_len`), iar `ciphertext` va conține datele criptate. Cheia și IV-ul sunt de lungime potrivită algoritmului (256 biți cheia, 128 biți IV pentru AES). **Notă:** AES fiind un cifru pe blocuri necesită un mod de operare (aici CBC – Cipher Block Chaining) și eventual padding. OpenSSL gestionează automat padding-ul implicit (folosind PKCS#7). Decriptarea se face similar, dar folosind `EVP_DecryptInit_ex` și `EVP_DecryptUpdate/Final_ex` cu aceeași cheie și IV.

1.4 Criptare și decriptare asimetrică RSA

Generarea cheii RSA

```
c
#include <openssl/rsa.h>
#include <openssl/pem.h>
#include <openssl/err.h>

int main() {
    int             ret = 0;
    RSA             *r = NULL;
    BIGNUM          *bne = NULL;
    BIO             *bp_public = NULL, *bp_private = NULL;

    int             bits = 2048;
    unsigned long   e = RSA_F4;

    // Generare chei RSA
    bne = BN_new();
    ret = BN_set_word(bne, e); // e este exponentul public
//uzual (65537)
    r = RSA_new();
// generează o cheie RSA 2048 biți
    ret = RSA_generate_key_ex(r, bits, bne, NULL);
```

```
    // salvare cheie publică
    bp_public = BIO_new_file("public.pem", "w+");
    ret = PEM_write_bio_RSAPublicKey(bp_public, r);
   // salvare cheie privată
    bp_private = BIO_new_file("private.pem", "w+");
    ret = PEM_write_bio_RSAPrivateKey(bp_private, r, NULL,
NULL, 0, NULL, NULL);

    //
    BIO_free_all(bp_public);
    BIO_free_all(bp_private);
    RSA_free(r);
    BN_free(bne);

    return 0;
}
```

Astfel vom avea două fișiere PEM: unul cu cheia publică și unul cu cea privată, pe care le vom folosi în exemplu.

Criptarea cu cheie publică: pentru a cripta un mesaj cu RSA (mod PKCS#1 v1.5 sau OAEP), se folosește `RSA_public_encrypt`. De exemplu:

```
c
unsigned char *msg = (unsigned char*)"Mesaj secret";
int msg_len = strlen((char*)msg);
unsigned char *encrypted = malloc(RSA_size(rsa));
int out_len = RSA_public_encrypt(msg_len, msg, encrypted,
rsa, RSA_PKCS1_OAEP_PADDING);
```

Parametrul de padding `RSA_PKCS1_OAEP_PADDING` specifică folosirea schemei OAEP (mai sigură). Rezultatul criptat are lungimea `RSA_size(rsa)` (în bytes, eg. 256 bytes pentru o cheie de 2048 biți).

Decriptarea cu cheie privată: se folosește `RSA_private_decrypt` cu aceeași schemă de padding:

```
c
unsigned char *decrypted = malloc(msg_len);
```

```
int dec_len = RSA_private_decrypt(out_len, encrypted,
decrypted, rsa, RSA_PKCS1_OAEP_PADDING);
```

După decriptare, `decrypted` conține textul original (și `dec_len` ar trebui să fie egal cu `msg_len`).

Notă de securitate: RSA este folosit în general pentru a cripta *chei* sau mesaje scurte, nu fișiere întregi, din motive de performanță. Pentru fișiere mari se adoptă schema hibridă: se generează o cheie random pentru AES, se criptează fișierul cu AES, apoi cheia AES se criptează cu RSA

2. Java

2.1 Exemplu de criptare simetrică AES

Java are incluse în JDK librării de criptografie (Java Cryptography Architecture – **JCA**) care simplifică mult operațiile de criptare. Mai jos, criptăm un șir de caractere folosind AES-128-CBC:

```
java
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import java.util.Arrays;
import java.nio.charset.StandardCharsets;

public class CryptoExample {
    public static void main(String[] args) throws Exception
{
        // Cheie simetrică de 16 bytes pentru AES-128
        byte[] keyBytes =
"0123456789ABCDEF".getBytes(StandardCharsets.UTF_8);
        SecretKeySpec key = new SecretKeySpec(keyBytes,
"AES");
        // IV (vector de inițializare) de 16 bytes - ales
//aleator în practică
        byte[] ivBytes =
"AAAABBBBCCCCDDDD".getBytes(StandardCharsets.UTF_8);
        IvParameterSpec iv = new IvParameterSpec(ivBytes);
```

```
        // Inițializare cifru AES în modul CBC cu padding
//PKCS5
        Cipher cipher =
Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, key, iv);

        String plaintext = "Mesaj super secret";
        byte[] ciphertext = cipher.doFinal(
plaintext.getBytes(StandardCharsets.UTF_8) );

        System.out.println("Text clar: " + plaintext);
        System.out.println("Text criptat (hex): " +
bytesToHex(ciphertext));
    }

    // Utilitar pentru a transforma un array de octeți
//într-un șir hexazecimal
    private static String bytesToHex(byte[] bytes) {
        StringBuilder sb = new StringBuilder();
        for(byte b: bytes){
            sb.append(String.format("%02X", b));
        }
        return sb.toString();
    }
}
```

Explicații: Codul creează o cheie secretă de 128 biți dintr-un șir ASCII de 16 caractere și un IV fictiv (în practică IV-ul trebuie generat aleator și de obicei transmis împreună cu textul criptat). Se obține un obiect `Cipher` pentru algoritmul "AES/CBC/PKCS5Padding" – modul CBC cu padding standard PKCS#5 – apoi se inițializează în modul de criptare cu cheia și IV-ul. Metoda `doFinal` aplică automat padding și realizează criptarea datelor, rezultând un tablou de octeți cu textul criptat. La final, imprimăm textul criptat în hexazecimal. Într-o aplicație reală, acest text criptat ar putea fi trimis către server sau stocat local, urmând ca pentru decriptare să se folosească aceeași cheie și IV (sau IV-ul transmis). Avantajul JCA este că dezvoltatorul nu trebuie să se ocupe manual de fiecare pas (padding, operare pe blocuri); totuși, trebuie să aibă

grijă la gestionarea cheilor și IV-urilor (să fie securizate și sincronizate între emițător și receptor).

2.2 Java (Utilizând biblioteca BouncyCastle)

Java are o infrastructură criptografică bogată (JCE – Java Cryptography Extension). Vom folosi **BouncyCastle** ca furnizor, deoarece oferă suport extins (inclusiv ECC). Asigurați-vă că ați adăugat BouncyCastle în claspath și înregistrat ca Security Provider, de exemplu:

```
java
Security.addProvider(new BouncyCastleProvider());
```

După aceasta, putem utiliza API-urile standard Java (`Cipher`, `KeyPairGenerator`, `MessageDigest`, `Signature` etc.), specificând algoritmii doriți (BouncyCastle oferă implementare pentru aceștia).

Criptare și decriptare simetrică AES

```
java
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

public class AESExample {
    public static void main(String[] args) throws Exception
{
        // Generare cheie AES de 128 biți
        KeyGenerator keyGenerator =
KeyGenerator.getInstance("AES");
        keyGenerator.init(128);
        SecretKey secretKey = keyGenerator.generateKey();

        // Inițializare Cipher
      Cipher cipher = Cipher.getInstance("AES"); // mod ECB
// Cipher.getInstance("AES/CBC/PKCS5Padding"); // mod CBC
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
// byte[] iv = cipher.getIV();reținem IV-ul generat (pentru CBC)
        byte[] text = "Mesaj secret".getBytes();
```

```
        byte[] textEncrypted = cipher.doFinal(text);

        // Decriptare
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
// cipher.init(Cipher.DECRYPT_MODE, secretKey,
//new IvParameterSpec(iv))
  byte[] textDecrypted = cipher.doFinal(textEncrypted);

        System.out.println("Text original: " + new
String(text));
        System.out.println("Text criptat: " + new
String(textEncrypted));
        System.out.println("Text decriptat: " + new
String(textDecrypted));
    }
}
```

La decriptare vom folosi `cipher.init(Cipher.DECRYPT_MODE, secretKey, new IvParameterSpec(iv))` și apoi `cipher.doFinal(encrypted)` pentru a obține textul original.

Criptare și decriptare asimetrică RSA

```
java
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import javax.crypto.Cipher;
public class RSAExample {
    public static void main(String[] args) throws Exception
{
        // Generare chei
        KeyPairGenerator keyPairGenerator =
KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048);
```

```
        KeyPair keyPair =
keyPairGenerator.generateKeyPair();
        PublicKey publicKey = keyPair.getPublic();
        PrivateKey privateKey = keyPair.getPrivate();

        // Criptare
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
        byte[] input = "Mesaj secret".getBytes();
        byte[] encrypted = cipher.doFinal(input);
        // Decriptare
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        byte[] decrypted = cipher.doFinal(encrypted);
        System.out.println("Text original: " + new
String(input));
        System.out.println("Text criptat: " + new
String(encrypted));
        System.out.println("Text decriptat: " + new
String(decrypted));
    }
}
```

3. C# (C Sharp)

3.1 Exemplu de criptare AES

Platforma .NET oferă clasa **Aes** în spațiul de nume `System.Security.Cryptography` pentru a realiza criptarea AES. Un exemplu simplu care folosește AES-CBC cu un IV generat automat:

```
csharp
using System;
using System.Security.Cryptography;
using System.Text;
class CryptoExample {
    static void Main() {
        string plaintext = "Secretul lui Polichinelle";
```

```
        // Inițializare obiect AES cu cheia și IV generate
//implicit
        using (Aes aesAlg = Aes.Create()) {
            aesAlg.KeySize = 128; // putem specifica 128,
//192 sau 256
            aesAlg.GenerateKey();
            aesAlg.GenerateIV();
             ICryptoTransform encryptor =
aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
            byte[] plaintextBytes =
Encoding.UTF8.GetBytes(plaintext);
byte[] cipherBytes =

encryptor.TransformFinalBlock(plaintextBytes, 0,
plaintextBytes.Length);
      Console.WriteLine("Text clar: {0}", plaintext);
      Console.WriteLine("Text criptat (Base64): {0}",
Convert.ToBase64String(cipherBytes));
        }
    }
}
```

Explicații: Codul folosește `Aes.Create()` pentru a obține o instanță a algoritmului AES cu parametri implicit siguri (mod CBC, padding PKCS7). Cheia și IV-ul sunt generate aleator (`GenerateKey()` și `GenerateIV()`), putând alternativ să fie setate manual (`aesAlg.Key = ..., aesAlg.IV = ...`). Apoi se creează un obiect `encryptor` pe baza cheii și IV-ului actual, și se apelează `TransformFinalBlock` pentru a cripta datele (efectuând și padding-ul necesar). Rezultatul, `cipherBytes`, este afișat ca text în baza64. Într-o aplicație reală, am transmite probabil și IV-ul alături de textul criptat, deoarece este necesar la decriptare (fără IV nu putem decoda mesajul criptat). .NET oferă și metode convenabile precum `write()` pe un `CryptoStream` dacă dorim să criptăm fluxuri de date (de exemplu, fișiere mai mari). Ca bună practică, folosim blocul `using` pentru a ne asigura că obiectul Aes este eliminat din memorie imediat după utilizare, ștergând astfel și datele sensibile (cheia) din memorie.

3.2 Criptare și decriptare simetrică AES

```
csharp
using System;
using System.Security.Cryptography;
using System.Text;

public class AESExample {
    public static void Main() {
        // Generare cheie și IV
        Aes aesAlg = Aes.Create();
        aesAlg.GenerateKey();
        aesAlg.GenerateIV();

        // Criptare
        ICryptoTransform encryptor =
aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
        byte[] original = Encoding.UTF8.GetBytes("Mesaj
secret");
        byte[] encrypted =
encryptor.TransformFinalBlock(original, 0, original.Length);

        // Decriptare
        ICryptoTransform decryptor =
aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
        byte[] decrypted =
decryptor.TransformFinalBlock(encrypted, 0, encrypted.Length);

        Console.WriteLine("Text original: " +
Encoding.UTF8.GetString(original));
        Console.WriteLine("Text criptat: " +
BitConverter.ToString(encrypted));
        Console.WriteLine("Text decriptat: " +
Encoding.UTF8.GetString(decrypted));
    }
}
```

3.3 Criptare și decriptare asimetrică RSA

```
csharp
using System;
using System.Security.Cryptography;
using System.Text;

public class RSAExample {
    public static void Main() {
        using (RSA rsa = RSA.Create()) {
            rsa.KeySize = 2048;

            // Criptare
            byte[] original = Encoding.UTF8.GetBytes("Mesaj
secret");
            byte[] encrypted = rsa.Encrypt(original,
RSAEncryptionPadding.OaepSHA256);

            // Decriptare
            byte[] decrypted = rsa.Decrypt(encrypted,
RSAEncryptionPadding.OaepSHA256);

            Console.WriteLine("Text original: " +
Encoding.UTF8.GetString(original));
            Console.WriteLine("Text criptat: " +
BitConverter.ToString(encrypted));
            Console.WriteLine("Text decriptat: " +
Encoding.UTF8.GetString(decrypted));
        }
    }
}
```

4. Python

4.1 Exemplu de criptare/decriptare AES

În Python, putem folosi biblioteca cryptography (care la rândul ei folosește OpenSSL în spate) sau PyCryptodome. Vom demonstra cu biblioteca cryptography utilizarea unui Fernet (simetric, bazat pe AES128 in CBC cu HMAC pentru integritate):

```
python
from cryptography.fernet import Fernet
# Generare cheie simetrică random
key = Fernet.generate_key()
f = Fernet(key)
plaintext = b"Date ultra-secrete"
ciphertext = f.encrypt(plaintext)
print("Text clar:", plaintext)
print("Text criptat:", ciphertext)
print("Text decriptat:", f.decrypt(ciphertext))
```

Explicații: Fernet este o schemă de criptare simetrică autenticată furnizată de bibliotecă care include tot ce e necesar (genera automat un IV, folosește AES-128-CBC intern și atașează un HMAC-SHA256 pentru a asigura integritatea și autenticitatea mesajului criptat). În exemplu, generăm o cheie random de 32 bytes, criptăm un mesaj și apoi îl decriptăm pentru a verifica. Rezultatul `ciphertext` este un token în formă URL-safe base64 care include atât IV-ul cât și HMAC-ul. Avantajul folosirii unei biblioteci de nivel înalt precum Fernet este că reduce riscul de a uita vreo etapă (de ex. verificarea integrității sau folosirea unui IV aleator). Dacă dorim un control mai detaliat, putem folosi direct modul AES din `cryptography.hazmat`:

```
python
from cryptography.hazmat.primitives.ciphers import Cipher,
algorithms, modes
from cryptography.hazmat.primitives import padding
from os import urandom
key = urandom(32)  # cheie de 256 biți
iv = urandom(16)   # IV de 128 biți
cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
encryptor = cipher.encryptor()
```

```
padder = padding.PKCS7(128).padder()  # bloc de 128 biți
pentru AES
plaintext = b"Exemplu"
padded_data = padder.update(plaintext) + padder.finalize()
ciphertext = encryptor.update(padded_data) +
encryptor.finalize()
```

Acest fragment arată pașii manual pentru AES-CBC în Python: generăm cheie și IV, configurăm un cifru, facem padding manual (bloc de 128 biți) și criptăm. Este mai supus erorilor, de aceea pentru majoritatea aplicațiilor se preferă Fernet sau alte abstracții.

4.2 Criptare și decriptare simetrică AES (Modul EAX pentru autentificare și confidențialitate)

```
python
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
key = get_random_bytes(16)  # Cheie de 128 biți
cipher = AES.new(key, AES.MODE_EAX)  # Modul EAX pentru
#autentificare și confidențialitate
data = b'Mesaj secret'
ciphertext, tag = cipher.encrypt_and_digest(data)
# Decriptare
cipher_dec = AES.new(key, AES.MODE_EAX, nonce=cipher.nonce)
decrypted = cipher_dec.decrypt_and_verify(ciphertext, tag)
print("Text original:", data.decode())
print("Text criptat:", ciphertext.hex())
print("Text decriptat:", decrypted.decode())
```

În acest exemplu, `encrypt_and_digest` returnează atât textul criptat `ciphertext` cât și un cod de autentificare `tag` calculat cu algoritmul intern (similar HMAC).

4.3 Criptare și decriptare asimetrică RSA

```
python
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
key = RSA.generate(2048)
public_key = key.publickey()
cipher_rsa = PKCS1_OAEP.new(public_key)
```

```
data = b'Mesaj secret'
encrypted = cipher_rsa.encrypt(data)
# Decriptare
cipher_rsa_dec = PKCS1_OAEP.new(key)
decrypted = cipher_rsa_dec.decrypt(encrypted)
print("Text original:", data.decode())
print("Text criptat:", encrypted.hex())
print("Text decriptat:", decrypted.decode())
```

Concluzie la criptare: Indiferent de limbaj, dezvoltatorii au la dispoziție biblioteci solide care implementează algoritmi standard. Este important să le folosească corect (de ex., să nu folosească același IV static mereu, să includă și mecanisme de verificare a autenticității datelor criptate sau moduri de operare autentificate).
Implementarea proprie a algoritmilor sau utilizarea de scheme triviale (ex: “criptarea” datelor prin XOR cu o cheie) trebuie evitate, deoarece adversarii le vor sparge ușor. În schimb, criptarea bine implementată asigură confidențialitatea datelor utilizatorilor și integritatea comunicațiilor, fiind un pilon esențial al securității aplicațiilor.