Zadanie 2: bezpieczniejszy czat
SKJ (2016)

Wstęp

Celem zadania jest zaimplementowanie komunikatora internetowego, wykorzystującego elementy kryptografii asymetrycznej. System obejmuje:

- **Centralny serwer (PKD)**: będący bazą danych, przechowującą rekordy zawierające następujące dane klientów: (alias, klucz publiczny, numer IP, port).
- **Klienci (usr)**: biorą udział w rozmowach. Każdy z nich ma możliwość zarejestrowania się w **PKD**, uaktualnienia swoich danych oraz pobrania numeru IP i klucza publicznego dowolnego z pozostałych klientów.

Komunikacja **usr-PKD** musi spełniać postawowe wymagania bezpieczeństwa, określone w dalszej części dokumentu. Komunikacja **usr-usr** jest szyfrowana.

O kryptografii asymetrycznej

W zadaniu zakłada się użycie kryptografii asymetrycznej. W tym schemacie każdy z uczestników komunikacji dysponuje parą kluczy: prywatnym (**PrivK**) oraz publicznym (**PubK**). Do zaszyfrowania (funkcja **Enc**) danej wiadomości w potrzebny jest tylko klucz publiczny. Do odszyfrowania (funkcja **Dec**) potrzebny jest klucz prywatny. Żaden z kluczy prywatnych nie ma prawa brać udziału w wymianie komunikatów w sieci. Klucz publiczny każdego z uczestników komunikacji jest ogólnie dostępny.

Poniżej podajemy przykładowy schemat wymiany zaszyfrowanej wiadomości pomiędzy dwoma hostami: A i B. Klucze pierwszego to $\mathbf{PrivK}_A$, $\mathbf{PubK}_A$, zaś drugiego to $\mathbf{PrivK}_B$, $\mathbf{PubK}_B$. Przebieg komunikacji:

1. Host A pobiera (z zaufanego miejsca w sieci, tzn. **PKD**) klucz publiczny $\mathbf{PubK}_B$ hosta B,
2. A szyfruje wiadomość w obliczając $c = \mathbf{Enc}(w, \mathbf{PubK}_B)$ i wysyła wynik do B,
3. Host B deszyfruje c, obliczając $w = \mathbf{Dec}(c, \mathbf{PrivK}_B)$.

Częściowe bezpieczeństwo tego schematu opiera się na założeniu, że zaszyfrowaną kluczem publicznym wiadomość może odczytać tylko posiadacz pasujacego klucza prywatnego.

Specyfikacja

Centralny serwer

Celem **PKD** jest utrzymywanie centralnej bazy danych o klientach. Kluczem głównym jest **alias** klienta, a powiązane z nim dane to: **klucz publiczny**, **numer IP** i **port**. Serwer jest również uczestnikiem szyfrowanej komunikacji, udostępnia więc klientom swój klucz publiczny. **PKD** powinien umożliwiać następujące funkcjonalności:

- **Dodanie nowego aliasu**: klient ma możliwość wprowadzenia do bazy nowego aliasu wraz z danymi.
- **Modyfikacja danych aliasu**: klient może dokonać *dowolnych* zmian w danych związanych z aliasem do którego jest uprawniony.
- **Pobranie danych aliasu**: dowolny klient może pobrać dane związane z dowolnym aliasem.

Wymagania bezpieczeństwa PKD

1. Dodawanie nowego aliasu musi zostać wykonane w sposób bezpieczny: **PKD** musi upewnić się, że wydający to polecenie klient posiada klucz prywatny powiązany z rejestrowanym kluczem publicznym.
2. Modyfikacja danych aliasu powinna być dokonywana w sposób pozwalający na uniknięcie *ataku powtórzenia*, tzn. sytuacji w której wiadomość przesłana od klienta do **PKD** przechwycona przez atakującego wykorzystana jest do wprowadzenia ponownych zmian. Ilustracja udanego ataku:
   (a) **usr** A wysyła do **PKD** zaszyfrowaną wiadomość c, pozwalającą na zmianę w bazie danych jego klucza publicznego na $\mathbf{PubK}_A^1$. Serwer **PKD** wykonuje polecenie, ale wiadomość c zostaje przechwycona przez **Adwersarza**.
   (b) Przez pewien czas trwa normalna komunikacja.
   (c) **usr** A ponownie zmienia swój klucz publiczny w bazie **PKD**, tym razem na $\mathbf{PubK}_A^2$.
   (d) **Adwersarz wysyła do PKD wiadomość *c***. W ten sposób ustawia klucz publiczny **usr** A ponownie na $\mathbf{PubK}_A^1$, zaś **usr** A traci możliwość korzystania ze swojego konta.

Klient

Program klienta powinien pozwalać na korzystanie z **PKD** sposób nienaruszający jego zasad bezpieczeństwa. Wymagane do zaimplementowania funkcjonalności związane są z przygotowaniem do komunikacji, komunikacją z serwerem i z innymi klientami:

- **Wygenerowanie nowej pary kluczy**: **PrivK**, **PubK**.
- **Wybór serwera PKD**: **usr** łączy się z podanym IP na danym porcie, po czym pobiera i zapisuje klucz publiczny **PKD**.
- **Zmiana danych w PKD**: **usr** dokonuje jej w sposób spełniający wyżej opisane wymagania bezpieczeństwa **PKD**.
- **Pobranie danych kontaktowych**: **usr** pobiera rekord powiązany z dowolnie wybranym aliasem.
- **Nawiązanie połączenia z innym klientem**: **usr** łączy się z innym klientem, korzystając z wcześniej pobranych danych kontaktowych.

Komunikacja **usr-usr** powinna być szyfrowana, przy czym sposób szyfrowania zależy od autora rozwiązania. Dozwolone jest wykorzystanie kryptografii asymetrycznej i użycie już istniejących kluczy.

Wymagania i sposób oceny

1. Poprawny i pełny projekt wart jest **6 punktów**. Za zrealizowanie każdej z poniższych funkcjonalności można otrzymać do **2 punktów**:
   - Funkcjonalności offline serwera i klientów (wygenerowanie lub odświeżenie kluczy, modyfikacja bazy danych, GUI, itp.).
   - Bezpieczna komunikacja **usr-PKD**.
   - Komunikacja **usr-usr**.
2. Aplikację piszemy w języku Java zgodnie ze standardem Java 8 (JDK 1.8). Do komunikacji przez sieć można wykorzystać jedynie podstawowe klasy do komunikacji z wykorzystaniem protokołu UDP (DatagramPacket, DatagramSocket). Metody kryptografii zapewnić powinno Java Cryptography Architecture (JCA). Poza tym można używać dowolnych elementów oficjalnej biblioteki Javy, w szczególności do budowy GUI można wykorzystać bibliotekę Swing lub JavaFX. W tym zadaniu GUI ma znaczenie drugorzędne.
3. Projekty powinny zostać zapisane do odpowiednich katalogów w systemie EDUX w nieprzekraczalnym terminie 17.XII.2016 (termin może zostać zmieniony przez prowadzącego grupę).
4. Spakowany plik projektu powinien obejmować:

- Opis działania systemu, w szczególności opisujący jak zastosowano szyfrowanie w komunikacji **usr-PKD** (oraz **usr-usr**, jeśli została zrealizowana dodatkowa funkcjonalność). Opisana powinna być zastosowana metoda metoda uniknięcia ataku powtórzenia.
- Pliki źródłowe (dla JDK 1.8) (włącznie z wszelkimi bibliotekami nie należącymi do standardowej instalacji Javy, których autor użył) - aplikacja musi dać się bez problemu skompilować na komputerach w laboratorium w PJA.
- Plik *Readme.txt* z opisem i uwagami autora (co zostało zrealizowane, czego się nie udało, gdzie ewentualnie są błędy, których nie udało się poprawić).

5. Prowadzący oceniać będą w pierwszym rzędzie poprawność działania programu i zgodność z wymaganiami bezpieczeństwa, ale na ocenę wpływać będzie także zgodność wytworzonego oprogramowania z zasadami inżynierii oprogramowania i jakość implementacji.

6. JEŚLI NIE WYSZCZEGÓLNIONO INACZEJ, WSZYSTKIE NIEJASNOŚCI NALEŻY PRZEDYSKUTOWAĆ Z PROWADZĄCYM ZAJĘCIA POD GROŹBĄ NIEZALICZENIA PROGRAMU W PRZYPADKU ICH NIEWŁAŚCIWEJ INTERPRETACJI.

Materiały

1. Java Cryptography Architecture: `http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html`

2. Przykładowy program ilustrujący szyfrowanie przy pomocy RSA:

```
import java.util.*;
import java.security.*;
import java.security.spec.*;
import javax.crypto.Cipher;

public class CryptoTest {

    public static void main(String[] args) throws Exception {
        //Example: basic RSA encoding/decryption

        // ---- Key generation ----

        //Generate public/private keys
        KeyPairGenerator kpairg = KeyPairGenerator.getInstance("RSA");
        kpairg.initialize(1024);
        KeyPair kpair = kpairg.genKeyPair();
        Key publicKey = kpair.getPublic();
```

```
        Key privateKey = kpair.getPrivate();

        //Key factory, for key-key specification transformations
        KeyFactory kfac = KeyFactory.getInstance("RSA");

        //Generate plain-text key specification
        RSAPublicKeySpec keyspec = (RSAPublicKeySpec)
            kfac.getKeySpec(publicKey, RSAPublicKeySpec.class);
        System.out.println("Public key, RSA modulus: " +
            keyspec.getModulus() + "\nexponent: " +
            keyspec.getPublicExponent() + "\n");

        //Building public key from the plain-text specification
        Key recoveredPublicFromSpec = kfac.generatePublic(keyspec);

        //Encode a version of the public key in a byte-array
        System.out.print("Public key encoded in " +
            kpair.getPublic().getFormat() + " format: ");
        byte[] encodedPublicKey = kpair.getPublic().getEncoded();
        System.out.println(Arrays.toString(encodedPublicKey) + "\n");

        //Building public key from the byte-array
        X509EncodedKeySpec ksp = new X509EncodedKeySpec(encodedPublicKey);
        Key recoveredPublicFromArray = kfac.generatePublic(ksp);

        // ---- Using RSA Cipher to encode simple messages ----

        //Encoding using public key. Warning - ECB is unsafe.
        String message = "Please encode me now!";
        Cipher cipherEncode = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        cipherEncode.init(Cipher.ENCRYPT_MODE, publicKey);
        byte[] encodedMessage = cipherEncode.doFinal(message.getBytes());
        System.out.println("Encoded \"" + message + "\" as: " +
            Arrays.toString(encodedMessage) + "\n");

        //Decoding using private key
        Cipher cipherDecode = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        cipherDecode.init(Cipher.DECRYPT_MODE, privateKey);
        String decodedMessage = new
            String(cipherDecode.doFinal(encodedMessage));
        System.out.println("Decoded: " + decodedMessage);

    }

}
```