Laboratorium nr 9

Atak na błędną implementację ECDSA

Czas wykonania: **lab 9**
Liczba punktów: **3 + 2**
PRK: T-L-4

1. Opis laboratorium

Celem laboratorium jest zapoznanie się z algorytmem podpisu cyfrowego wykorzystującym krzywe eliptyczne (ECDSA). W ramach zadania zademonstrujemy jak z pozoru niewielki błąd w implementacji, który łatwo przeoczyć, może umożliwić hakerowi odzyskanie klucza prywatnego. ECDSA wymaga bezpiecznej liczby losowej dla każdego podpisu. Nie można jednej liczby losowej stosować dwa razy, bo inaczej wspomniany atak się powiedzie. Co więcej do przeprowadzenia tego ataku wystarczą jedynie dwie pary wiadomość podpis. Błąd ten pojawił się m. in. w implementacji konsoli PlayStation 3 i umożliwił wydobycie klucza prywatnego używanego do podpisywania gier.

Zadanie wymaga użycia języka Java ze względu na konieczność wykorzystania biblioteki w tym języku i wyjściowego fragmentu kodu.

ECDSA

1 - Generowanie pary kluczy

- parametry publiczne: krzywa eliptyczna, generator G, rząd grupy n
- klucz prywatny: liczba losowa całkowita d_A losowo wybrana z przedziału $[1, n-1]$
- klucz publiczny: $Q_A = d_A \times G$

2 - Podpisywanie

1. Oblicz $e = Hash(m)$
2. Niech z to będzie L_n bitów będących najbardziej z lewej e, L_n to długość w bitach dla rzędu grupy n
3. Wybierz bezpieczną kryptograficzną liczbę losową k z zakresu $[1, n-1]$
4. Oblicz punkt na krzywej $(x_1, y_1) = k \times G$
5. Oblicz $r = x_1 \bmod n$. Jeżeli $r = 0$ wróć do kroku 3
6. Oblicz $s = k^{-1} (z + rd_A) \bmod n$. Jeżeli $s = 0$ wróć do kroku 3
7. Podpisem jest para liczb (r, s)

3 - Weryfikacja

1. Sprawdź czy liczby r i s to liczby całkowite z zakresu $[1, n-1]$. Jeżeli nie to podpis jest nieważny
2. Oblicz $e = Hash(m)$
3. Niech z to będzie L_n bitów będących najbardziej z lewej e
4. Oblicz $u_1 = zs^{-1} \bmod n$ i $u_2 = rs^{-1} \bmod n$
5. Oblicz $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. Jeżeli wynik to 0 to podpis jest nieważny.
6. Podpis jest ważny, gdy $r = x_1 \pmod n$

Atak

1. Mamy dwa podpisy (r, s_1) i (r, s_2) i skróty z wiadomości z_1 i z_2
2. Oblicz $s_1 - s_2 = k^{-1}(z_1 - z_2)$
3. Oblicz $k = (z_1 - z_2) / (s_1 - s_2)$
4. Ponieważ $s = k^{-1}(z + rd_A)$ to klucz prywatny $d_A = (s_1k - z_1) / r$

   wszystkie obliczenia mod n

2. Materiały

- https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm
- https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
- Console Hacking 2010 PS3 Epic Fail https://fahrplan.events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf
- Android Security Vulnerability, https://bitcoin.org/en/alert/2013-08-11-android

3. Zadania do wykonania

Przygotuj narzędzia do programowania:

a) Jeżeli nie masz zainstalowanego środowiska programistycznego do Javy to zainstaluj wraz z JDK (polecane to Intellij Idea: https://www.jetbrains.com/idea/, open-source).
b) Utwórz aplikację Hello World, aby zobaczyć czy wszystko poprawnie działa: https://www.jetbrains.com/help/idea/creating-and-running-your-first-java-application.html
c) Pobierz bibliotekę kryptograficzną ECCelerate, https://jce.iaik.tugraz.at/products/core-crypto-toolkits/, należy pobrać wersje „Evaluation Version", która jest bezpłatna do celów edukacyjnych, wymaga rejestracji.
d) Dodaj biblioteki do projektu (File->Project structure->Modules->Dependecies->**+**)

1) **Zadanie 9.1** (2 pkt na lab9) – przekopiuj i uruchom daną poniżej implementację ECDSA. Odpowiedz na poniższe pytania:
   a) Jaka krzywa eliptyczna jest używana, jaki jest jej rozmiar?
   b) Jak wybierane jest *k*?
   c) Jakie kroki w tej implementacji demonstracyjnej ECDSA zostały pominięte?

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.security.spec.ECPoint;

import iaik.security.ec.common.ECParameterSpec;
import iaik.security.ec.common.ECStandardizedParameterFactory;
import iaik.security.ec.common.SecurityStrength;
```

```
System.out.println("---- Demo implementacji ECDSA: Tomasz Hyla 2022");

//---Parametry systemowe
//-krzywa eliptyczna ustandaryzowana
ECParameterSpec ec_params =
ECStandardizedParameterFactory.getParametersByName("secp521r1");
```

```
iaik.security.ec.common.EllipticCurve ec = ec_params.getCurve(); //tej używamy dalej
iaik.security.ec.math.curve.EllipticCurve ec2 = ec.getIAIKCurve(); //klasa EC w innej
bibliotece...
System.out.println(ec2.toString());
BigInteger n = ec2.getOrder();
int size = ec2.getField().getFieldSize();
System.out.println("Rozmiar w bitach: " + size + " liczba elementów (n)= " + n);
//-generator liczb losowych
final SecureRandom random =
SecurityStrength.getSecureRandom(SecurityStrength.getSecurityStrength(size));

//---Generowanie kluczy
//prywatny
BigInteger dA = new BigInteger(size - 1, random);
//publiczny
ECPoint QA = ec.multiplyGenerator(dA); // Q_A=d_A*G

//---Podpisywanie
//1-2.
BigInteger z = BigInteger.ZERO;
try {
    String m = "Kryptologia 2022";
    MessageDigest sha = MessageDigest.getInstance("SHA-512");
    byte[] messageDigest = sha.digest(m.getBytes());
    z = new BigInteger(1, messageDigest);
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
//3.
BigInteger r = BigInteger.ZERO;
BigInteger s = BigInteger.ZERO;
do
{
    BigInteger k = new BigInteger(size - 1, random);
    //4.
    var kG = ec.multiplyGenerator(k);
    //5.
    r = kG.getAffineX().mod(n);
    System.out.println(r.toString());
    s = k.modInverse(n).multiply(z.add(r.multiply(dA))).mod(n);
}
while (r.equals(BigInteger.ZERO) || s.equals(BigInteger.ZERO));
// podpis to para (r,s)

//Weryfikacja
//2.
BigInteger z2 = BigInteger.ZERO;
try {
    String m = "Kryptologia 2022";
    MessageDigest sha = MessageDigest.getInstance("SHA-512");
    byte[] messageDigest = sha.digest(m.getBytes());
    z2 = new BigInteger(1, messageDigest);
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
//4.
BigInteger s1 = s.modInverse(n);
BigInteger u1 = z2.multiply(s1).mod(n);
BigInteger u2 = r.multiply(s1).mod(n);

ECPoint tmp1 = ec.multiplyGenerator(u1);
ECPoint tmp2 = ec.multiplyPoint(QA, u2);
ECPoint C = ec.addPoint(tmp1, tmp2);

if (C.getAffineX().equals(BigInteger.ZERO) == false ||
        C.getAffineX().mod(n).equals(r))
```

```
{
    System.out.println("Podpis poprawny");
}
else { System.out.println("Podpis niepoprawny");
}
```

2) **Zadanie 9.2** (3 pkt) – na podstawie opisu zawartego na początku tej instrukcji zademonstruj atak polegający na odzyskania klucza prywatnego w przypadku, gdy zamiast losowej wartości *k* dla każdego podpisu wybierana jest ta sama wartość.