Adatbiztonság és Kriptográfia

01. gyakorlat

jegyzőkönyv

Ekart Csaba [ZWPMKP]

2020. február 20.

Előkészületek

A gyakorlathoz előkészületként összeállítottam a megfelelő fejlesztői környezetet, a Python rendszert, telepítettem a leírtaknak megfelelően a *pycryptodome* nevű kriptográfiai modult, illetve frissítettem korábban gyűjtött tudásomat az RSA algoritmusról.

Bevezetés

RSA [1] [5]

Az RSA eljárás egy nyílt kulcsú, azaz asszimetrikus titkosító algoritmus, melyet 1976-ban Ron Rivest, Adi Shamir és Len Adleman fejlesztett ki (és az elnevezést nevük kezdőbetűiből kapta). Az RSA napjain egyik legszélesebb körben alkalmazott nyílt kulcsú algoritmusa. Biztonságát a nagy számokra nézett prímfaktorizációs feladat nehézsége adja. A titkosításhoz két nagy prím számot, és ezek szorzatát használjuk. Minél nagyobb ez a szám, annál biztonságosabb lesz a titkosítás kriptográfiai szempontból. Ennek a számnak biztonságosnak tekintett a mérete 2017-ben 2048 bit. Az algoritmus valójában nem feltörhetetlen, de jelenlegi ismereteink alapján belátható időn belül, elképesztően nagy számítási kapacitás birtokában is lehetetlen.

Kriptográfiai hash függvény [6] [7]

A hash függvények olyan eljárások, melyek segítségével bármilyen hosszúságú adatot adott hosszúságúra képezhetünk le. A kapott adatot hashnek / hasítóértéknek nevezzük. Egy jó hash függvénytől azt várjuk, hogy különböző bemenetekhez különböző, látszólag véletlenszerű egyértelmű eredményt kapjunk. A cél az, hogy a kimenetből lehetetlen legyen a bemenetre következtetni, viszont a bemenetből a kimenet gyorsan előállítható. Elméletben előfordulhat, hogy különböző bemenetek ugyanarra a hashre képződnek, ám megfelelően választott hasítófüggvény esetén ez igen valószínűtlen. Kriptográfiában a hash függvényt főleg adat integritásának biztosítására használunk, mely azt jelenti, hogy az ellenőrző fél le tudja futtatni a hash függvényt az üzenetre, és amennyiben ez egyezik a korábbi üzenet kivonatával, akkor az adat nem módosult. Jelen feladatokban az aláírások létrehozásánál és ellenőrzésénél fogjuk használni.
Az egyik legelterjedtebb kriptográfiai eljárás az SHA azaz Secure Hash Algorithm. (Valójában ez egy eljáráscsalád összefoglaló neve) Első változatát 1993-ban az NSA-nél fejlesztették. Az SHA-256, melyet a gyakorlaton is használtunk rendkívül elterjedt, a bitcoin és egyéb kriptovaluták is ezt használják lenyomatképzésre. A 256 a 256 bites hasht jelenti.

1. feladat

RSA kulcs formátumok

Az első feladatban a megadott kód kipótlása révén az RSA kulcsok tárolási, írási / olvasási módjaival ismertetett meg. A *pycryptodome* dokumentációjában megkerestem az RSA kulcs létrehozásával, tárolásával, illetve annak beolvasásának lehetőségeit leíró részeket [1], illetve részletesebben utána olvastam ezen fájltípusoknak.

Ezek alapján azt mondhatjuk, hogy az RSA kulcsok három általános formátumban tárolhatók: [4]

1. **Binary DER-encoded formátum.** Ez a legkompaktabb tárolási forma, mely a "legfurább" karakteket eredményezi, ha egy editorban beleolvassunk. Általában az első karakter 0.

2. **PEM vagy base64 formátum.** Ez az, amit ezen a gyakorlaton behatóbban is megvizsáltunk. Ez tulajdonképpen ugyanaz, mint az első pontban említett DER file, de ebben az esetben egy további base64-es kódolást is alkalmazunk, illetve hozzáfűzünk egy-egy fejléc és lábléc vonalat, melyben egyértelműen jelöljük, hogy a fájl RSA kulcsot tartalmaz. A gyakorlaton előállított PEM fájlokkal behatóbban foglalkozom a jegyzőkönyvben.

3. **XML formátum.** Az utolsó formátumban XML-ben tároljuk az RSA kulcsokat a W3C standardnak megfelelően. A használandó XML tageket a W3C szabja meg.

A feladat megoldása

Az első feladatban az előre megadott kód alapján létrehoztam az *ITKoin_01_ekart_csaba.py* állományt, melyben a hiányzó kódrészletek megírása volt a feladatom.

Az első megvalósítandó függvény szerepe a privát és publikus kulcsok legenerálása, illetve fájlba írása volt.

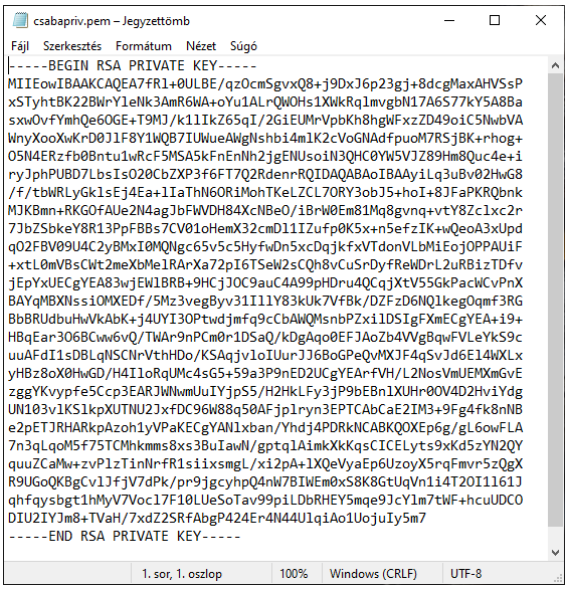
```
@staticmethod
def generate_rsa_key(filename):
    rsakey = RSA.generate(2048)  # generálj 2048 bites RSA kulcsot
    rsapublickey = rsakey.publickey()
    print(rsakey)
    pprint(rsakey)
    print(vars(rsakey))
    pprint(vars(rsakey))
    pprint(vars(rsakey))
    pprint(rsapublickey)
    pprint(vars(rsapublickey))
    PEMrsakey = rsakey.export_key(format="PEM")# PEM formátumra alakítsd az RSA kulcsot
    pprint(PEMrsakey)
    PEMrsapublickey = rsapublickey.export_key(format="PEM") # PEM formátumra alakítsd a kulcs publikus részét
    pprint(PEMrsapublickey)
    privatekeyfilename = filename + 'priv.pem'
    f = open(privatekeyfilename, 'wb')
    f.write(PEMrsakey)
    f.close()
    publickeyfilename = filename + 'pub.pem'
    f = open(publickeyfilename, 'wb')
    f.write(PEMrsapublickey)
    f.close()
    return
```

1. ábra. A *generate_rsa_key* függvény implementációja

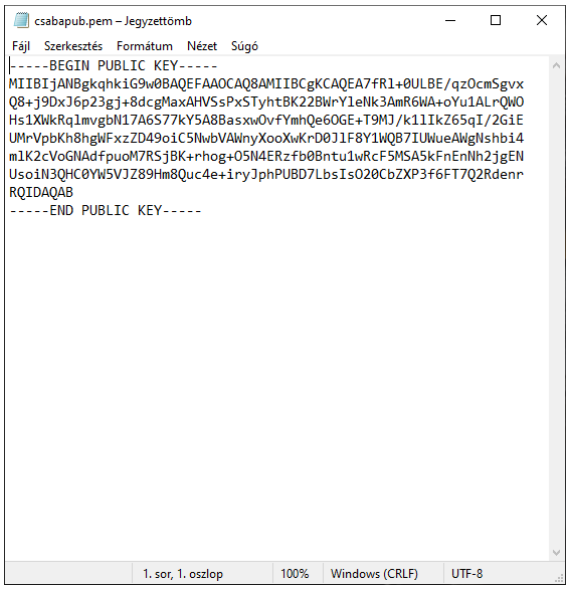
Az kódban kipótolt sorok magyarázata:

- Az olvasottak alapján a `RSA.generate(2048)` parancs segítségével egy 2048 bites RSA kulcsot generáltam, melyből egyből ki is nyertem a publikus kulcsot a `rsakey.publickey()` kódsor használatával.
- Az elkészült privát és publikus kulcsot PEM formátumban kiexportáltam az `rsakey.export_key(format='PEM')` parancs segítségével.
- A létrejött publikus, illetve privát kulcsok megtalálhatóak lesznek a függvényparaméteréből képzett fájlneveknek megfelelően.

A létrejött publikus és privát kulcs az alábbiak szerint alakult:



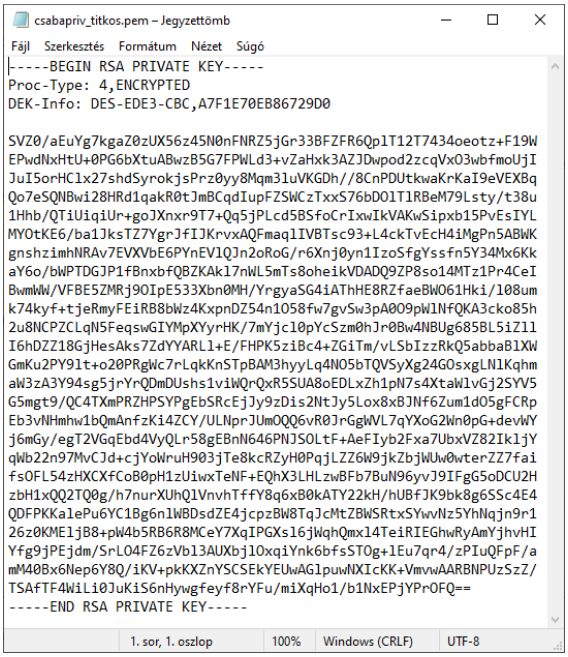
(a) Privát kulcs PEM fájlja



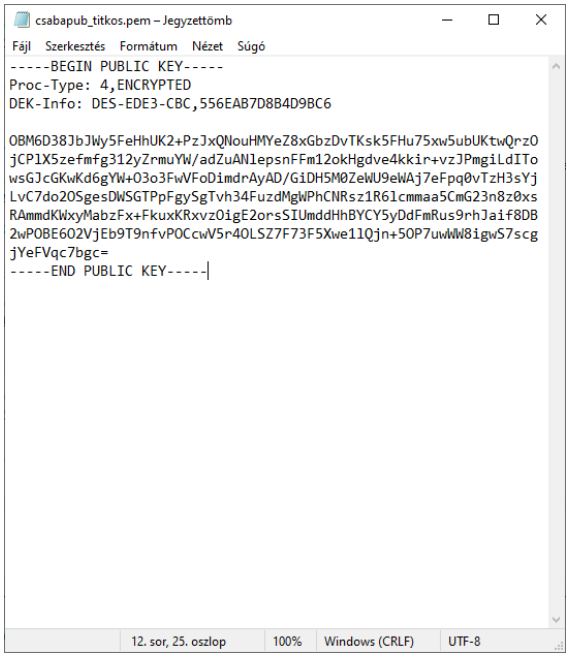
(b) Publikus kulcs PEM fájlja

Azt láthatjuk, hogy ahogy a bevezetésben a formátumoknál már leírtam egyértelműen meg van jelölve, hogy mi a privát, illetve a publikus kulcs (ez azért van, hogy könnyen kitalálható legyen a fájlról, hogy milyen célra használjuk), majd a base64 kódolt adat, ami jelenesetben a kulcs.

A kulcs generálásakor dönthetünk úgy, hogy egy jelkulcs megadásával titkosítjuk az exportálandó RSA kulcsokat. Ennek módja a *pycryptodome*ban, hogy az `rsakey.export_key(format="PEM")` sort kibővítem egy *passphrase* nevű paraméterrel, melybe egy tetszőleges jelszót adok bemenetnek. Ez jelenesetben "biztonságos" módon *password* lesz: `rsakey.export_key(format="PEM", passphrase="password")` Az így generált fájlok azonban egy kicsit másként néznek ki:



(a) Privát kulcs PEM fájlja titkosítva



(b) Publikus kulcs PEM fájlja titkosítva

Azt láthatjuk, hogy a PEM fájlok fejlécében megjelenik két extra mező: [3]

- A *Proc-Type* a verziót, és a vedelmét jelzi a tárolt adatnak, míg a
- *DEK-Info* két vesszővel elválasztott látszólag értelmetlen karaktersorozatot tartalmaz. Ezek a titkosítási algoritmus neve, illetve a hexadecimális inicializáló vektor.

Beolvasásnál hasonlóan az `rsakey.import_key(format="PEM", passphrase="password")` paranccsal beolvashatjuk a titkosított kulcsokat. Természetesen a publikus kulcsot teljesen értelmetlen titkosítani, azonban privát kulcsnál hasznos lehet. A publikus kulcs titkosítását csak a példa kedvéért végeztem el.

A következő két függvényben a kulcsok beolvasását valósítottam meg:

```
def load_key(self, filename):
    privatekeyfilename = filename + 'priv.pem'
    privatekeyfileobject = open(privatekeyfilename, 'r')
    privatekeyfilecontent = privatekeyfileobject.read()
    pprint(privatekeyfilecontent)
    rsakey = RSA.import_key(privatekeyfilecontent)
    self.rsakey = rsakey
    pprint(vars(self.rsakey))
    rsapublickey = rsakey.publickey()
    self.rsapublickey = rsapublickey
    pprint(vars(self.rsapublickey))
    return

def load_public_key(self, filename):
    publickeyfilename = filename + 'pub.pem'
    publickeyfileobject = open(publickeyfilename, 'r')
    publickeyfilecontent = publickeyfileobject.read()
    pprint(publickeyfilecontent)
    rsakey = RSA.import_key(publickeyfilecontent)
    rsapublickey = rsakey.publickey()
    self.rsapublickey = rsapublickey
    pprint(vars(self.rsapublickey))
    return
```

4. ábra. Az RSA kulcsokat beolvasó metódus

A kulcsok beolvasásához az `RSA.import_key(privatekeyfilecontent)` parancsot, illetve ennek megfelelőjét a publikus kulcshoz az `RSA.import_key(publickeyfilecontent)`-et használtam. Az *rsapublickey* változót az importált kulcsból a generáló metódusban használt módszerhez hasonlóan az `rsakey.publickey()` paranccsal nyertem ki.

2. feladat

A második feladatban olyan függvények megvalósítása volt a feladat, melyek a lenyomat készítés, az RSA aláírást, illetve ezen aláírások ellenőrzését valósítják meg.

Az első függvény egy lenyomatoló (hash) objektumot hoz létre tetszőleges adatból:

```
@staticmethod
def create_hashobject(data):
    stringdump = json.dumps(data)
    binarydump = stringdump.encode()
    hashobject = SHA256.new(data=binarydump)

    hashhexvalue = hashobject.hexdigest()
    print(hashhexvalue)
    return hashobject
```

5. ábra. A *create_hashobject* metódus implementációja

Minden alkalommal mikor szeretnénk egy üzenetet hashelni, egy hash objektumot kell létrehoznunk. Ehhez az `SHA256.new(data=binarydump)` metódust kellett használni.
Mivel ez a metódus csak bináris 8-bites adatokat "eszik meg", így nem tudja kezelni a Python3-ban használt string típust, mivel az Unicode karaktereket tartalmaz. E probléma orvoslására szerepel a feladatban a `stringdump = json.dumps(data)` és `binarydump = stringdump.encode()` sor. Ez megoldható lenne a string elé helyezett b karakterrel is.
Ezt követően a hashobjectet hex típusúvá alakítottuk a `hashobject.hexdigest()` paranccsal.

A következő részfeladatban egy digitális aláírást létrehozó és azt leellenőrző függvény implementációját kellett kiegészíteni. A digitális aláírásokhoz RSA-t használunk. Ennek lényege, hogy a privát kulcsunk segítségével legenerálunk egy aláírást egy hash objectre, melyet az ellenőrző fél a publikus kulcsunkkal könnyen ellenőrizhet. [8]

```
def create_signature(self, data):
    signatureobject = pkcs1_15.new(self.rsakey)
    hashobject = ITKoin.create_hashobject(data)
    signaturevalue = signatureobject.sign(hashobject)
    print(signaturevalue)
    b64signaturevalue = b64encode(signaturevalue)
    print(b64signaturevalue)
    print(b64signaturevalue.decode())
    return b64signaturevalue.decode()
```

6. ábra. A *create_signature* metódus implementációja

- A signature object létrehozása a `pkcs1_15.new(self.rsakey)` parancs segítségével lehetséges. Itt egy privát kulcsot várunk bemenetnek, mellyel létre fogjuk hozni az aláírást. A pkcs1_15-ből a pkcs a Public-Key Cryptography Standrads-t jelenti, az utána következő szám pedig a verziót jelöli.
- Ezt követően az adatot betöltjük egy hash objektumba a korábban megírt `create_hashobject(data)` segítségével.
- Majd létrehozzuk az aláírást a kulcs és a hash objektumot felhasználva: `signatureobject.sign(hashobject)`.
- A kimenetet base64 kódolásra kellett alakítani, ami a `b64encode(signaturevalue)` sor hozzáadásával volt megvalósítható.

Az utolsó metódus feladata az imént létrehozott aláírás ellenőrzése.

```
def verify_signature(self, data, b64signaturevalue, rsapublickey):
    verifyobject = pkcs1_15.new(rsapublickey)
    hashobject = ITKoin.create_hashobject(data)
    signaturevalue = b64decode(b64signaturevalue)
    try:
        signatureerror = verifyobject.verify(hashobject, signaturevalue)
        validsignature = True
    except (ValueError, TypeError):
        validsignature = False
    return validsignature
```

7. ábra. A *verify_signature* metódus implementációja

- A verify object létrehozása hasonlóan történik a signature objecthez, azzal a különbséggel, hogy a bemenet a publikus, nem pedig a privát kulcs: `pkcs1_15.new(rsapublickey)`.
- Az előzőhöz hasonlóan létrehozunk egy hash objectet: `create_hashobject(data)`
- Majd dekódoljuk a base64 dekódolással: `b64decode(b64signaturevalue)`.

- Ezt követően a dokumentációban szereplő példához hasonlóan [8] egy try-catch blokkban végrehajtjuk a verifikációt, mely ha sikeres, akkor a validsignature változó értékét True-ra, ha sikertelen False-ra allítjuk.

3. feladat

A harmadik feladat áttekintéséig nem jutottunk el a gyakorlaton, megoldása nem volt feladat, így nem is szerepel a jegyzőkönyvben.

Ellenőrzés

A megvalósított osztály metódusainak ellenőrzésre létrehozott rövid példakód:

```
csabaKoin = ITKoin()
print("######## Generate Keys #########")
ITKoin.generate_rsa_key("csaba")
print("######## Load Keys #########")
csabaKoin.load_key("csaba")
csabaKoin.load_public_key("csaba")

data = 'hutyutyu'
print("######## Create Signature #########")
signature = csabaKoin.create_signature(data)
#signature.encode('ascii')
signature_input = json.dumps(signature, sort_keys=True).encode()
print("######## Validate Signature #########")
valid = csabaKoin.verify_signature(data, signature_input, RSA.import_key(open("csabapub.pem").read()).publickey())
if (valid):
    print("valid sign")
else:
    print("not valid sign")
```

8. ábra. Az ellenőrzéshez használt rövid programkód

- Ebben létrehozok egy objektumot csabaKoin néven, majd legenerálom hozzá az RSA kulcsokat, és betöltöm ezeket.
- Létrehozok egy üzenetet, melyhez generálunk egy aláírást a privát kulcsunkkal, amit az imént töltöttünk be a csabaKoin objektumba
- A generált aláírás azonban még nem lesz használható, mert a formátuma Python3 string, mely többek között az első feladatban már említett okok miatt nem lesz jó. Ezért a `json.dumps().encode()` parancsot használjuk (a megfelelő bemenetekkel) az adat megfelelő formátumra alakításához.

Futás során jól látható az összes logolt adat, és a végén az aláírás validálása is sikeresen megtörténik, tehát a kód jól működik.

Hivatkozások

[1] `https://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html`

[2] `https://pycryptodome.readthedocs.io/en/latest/src/examples.html`

[3] `https://www.openssl.org/docs/man1.1.0/man3/PEM_read_bio_X509_REQ.html`

[4] `https://www.cryptosys.net/pki/rsakeyformats.html`

[5] `https://hu.wikipedia.org/wiki/RSA-elj%C3%A1r%C3%A1s`

[6] `https://www.thesslstore.com/blog/difference-sha-1-sha-2-sha-256-hash-algorithms/`

[7] https://hu.wikipedia.org/wiki/Kriptogr%C3%A1fiai_hash_f%C3%BCggv%C3%A9ny

[8] https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html