

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 napjaink egyik legszélesebb körben alkalmazott nyílt kulcsú algoritmus. 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 lenyomat-képzésre. 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 ismertett meg. A *pycryptodome* dokumentációjában megkerestem az RSA kulcs létrehozásával, tárolásával, illetve annak beolvasásának lehetőségeit leíró részeket [1], illetve részletesebben utána olvastam ezen fájl típusoknak.

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" karaktereket eredményezi, ha egy editorban beleolvassunk. Általában az első karakter 0.
2. **PEM vagy base64 formátum.** Ez az, amit ezen a gyakorlaton behatóbban is megvizsgáltunk. 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 írasa 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 kipótoltt 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:

```
csabapriv.pem - Jegyzetfömb
Fájl Szerkesztés Formátum Nézet Súgó
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKQAQEAFR1+0ULBE/qz0cmSgvxQ8+j9DxJ6p23gj+8dcgMaxAHVSsP
xSTyhtBK22BwY1eNk3AmR6WA+oYu1ALrQW0Hs1XWkRqImvgbN17A6S77kY5A8Ba
sxwOvFymhQe60GE+T9MJ/k11IkZ65qI/2G1EUMrVpbKh8hgWfXzZD49oiC5NwbVA
WnyXooXwKrD0J1F8Y1WQ87I1UueAWgNshbi4m1K2cVoGNAdFpuoM7RSjBK+rhog+
05N4ERzfb0Bntu1wRcF5MSA5kFnEnNh2jgEINUsoiN3QHC0Yw5VJZ89Hm8Quc4e+i
ryJ3pPUBD7LbsIs020CbZXP3f6FT7Q2RdenRQIDAQAABAAyIq3uBv02HwG8
/f/tbWRLyGk1sEj4Ea+1IaThN60R1MohTKeLZCL70RY3obJ5+hoI+8JFaPKRQbnk
MJKBmn+RKGF0AJe2M4gJbFVWDH84XcNB0e/iBrW0Em81Mq8gvnq+vtY8Zc1xc2r
73bZ5bkeY8R13PpFB8s7CV01oHemX32cmD11IZufp0K5x+n5efzIK+wQeoA3xUdp
q02FBV09U4C2yBmXi0MQNgc65v5c5HyfWdN5xcDqKfxVTDonVLbMIeojOPPAU1F
+xtL0mVBsCwt2meXbMe1ARXa72pI6TSeL2sCQh8vCuSrDyFrEwDrL2uRBizTDfv
jEPYxUECgYEAE83wjEW1BRB+9HCj3OC9auC4A99pHDru4QCqjXtV55GkPacWcPnX
BAyMBXNss10MXEDf/5Mz3vegByv31I11Y83kUk7VfBk/DZFzD6NQ1kegOqmF3RG
BBRBUdbuHwVAbK+j4UYI30Ptdjmfq9cCbAWQMsnbPZx1D5IGfXmECgYEAEa+19+
HbqEar306B6v6wQ/TWAr9nPcm0r1D5aQ/kDgAgo0EFJAozb4VvGbwFVLEyK59c
uuAFd1sD8LqNShirVthHDo/KSAqjv1oIUurJ36BoGPeQvMXJF4qSvJd6E14WLX
yHBz8oX0HwGD/H411oRqUmC4sG5+59a3P9nED2UCgYEArfVH/L2NosVmUEmGvE
zggYKvpfe5Ccp3EARJWnmUuIYjp5S/H2HkLFy3jP9bEbN1XUH00V4D2Hv1Ydg
UN103v1KS1kpXUTU2JxJcF96W88q50AFjp1ryn3EPTCAbCaE2IM3+9Fg4fk8NB
e2pETJRHARKpAzh01yVPAKECgYAN1xban/Yhdj4PDRkNCABKQXQEpG/gL6owFLA
7n3qLqoM5f75TCMhkmms8xs3BuIawI/gptq1AimKXkKsCICELyts9xKd5zYN2QY
quuZCaMw+zvP1tInNrFR1s1ixsmgI/xi2pA+1XQeVyaEp6UzoyX5nqFmvr5zQgX
R9UGoQKBgcV13fjV7dPk/pr9jgcyhpQ4nh78IWE0m0xS8K8GtUqVn1i4T20I1161J
qhfyqsbgt1hMyV7Voc17F10UeSoTav99p1LDbRHEY5mqe9JyC1m7tWF+hcuUDCO
DIU2IYJm8+TVah/7xdZ2SRfAbgP424Er4N44U1qAo1UojuIy5m7
-----END RSA PRIVATE KEY-----
1. sor, 1. oszlop 100% Windows (CRLF) UTF-8
```

(a) Privát kulcs PEM fájlja

```
csabapub.pem - Jegyzetfömb
Fájl Szerkesztés Formátum Nézet Súgó
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAFR1+0ULBE/qz0cmSgvx
Q8+j9DxJ6p23gj+8dcgMaxAHVSsPSTyhtBK22BwY1eNk3AmR6WA+oYu1ALrQW0
Hs1XWkRqImvgbN17A6S77kY5A8BasxwOvFymhQe60GE+T9MJ/k11IkZ65qI/2G1E
UMrVpbKh8hgWfXzZD49oiC5NwbVAWnyXooXwKrD0J1F8Y1WQ87I1UueAWgNshbi4
m1K2cVoGNAdFpuoM7RSjBK+rhog+05N4ERzfb0Bntu1wRcF5MSA5kFnEnNh2jgE
NUsoiN3QHC0Yw5VJZ89Hm8Quc4e+i ryJ3pPUBD7LbsIs020CbZXP3f6FT7Q2RdenR
RQIDAQA8
-----END PUBLIC KEY-----
1. sor, 1. oszlop 100% Windows (CRLF) UTF-8
```

(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 jelen esetben 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 kibővíttem egy *passphrase* nevű paraméterrel, melybe egy tetszőleges jelszót adok bemenetnek. Ez jelen esetben "biztonságos" módon *password* lesz: `rsakey.export_key(format="PEM", passphrase="password")` Az így generált fájlok azonban egy kicsit másként néznek ki:

```
csabapriv_titkos.pem - Jegyzetfömb
Fájl Szerkesztés Formátum Nézet Súgó
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4, ENCRYPTED
DEK-Info: DES-EDE3-CBC, A7F1E70EB8672900

SVZ0/aEuYg7kgaZ0xUX56z45N0nFRNZ5JGr33BFZFR6Qp1T127434oetoz+F19W
EPwdNxtHtU+0PG6bXtuABwzB5G7FPWLd3+vZAhxk3AZJDwpod2zcVx03wbfmoJI
JuISorHC1x27shdSyrokj5Pr20y8Mqm31uVKGDH//8CnPDUTkwaKrKaI9eVEXBq
Qo7eSQNBwi28HRd1qakR0tJmBCqdIupFZSWCzTxxS76bD01T1R8eM79Lsty/t38u
1Hhb/QTiuiqiuR+goJXnxr97T+Qq5jPLcd5B5foCrIwxIKvAKwSipxb15PvEsIYL
MY0tKE6/ba1JksTZ7YgrJfJIKrvxAQFmaQ1IVB8Tsc93+L4ckTvEcH41MgPn5ABWk
gnshz1mhNRAv7EVXVbE6PYnEV1QJn2oRoG/r6Xnj0yn1IzoSfgYssfn5Y34Mx6Kk
aY6o/bWPTDGJP1fBnxbfQ8ZKAk17nWL5mTs8oheikVDADQ9ZP8so14MTz1Pr4CeI
BwmWV/VFBE5ZMRj90IpE533Xbn0MH/YrgyaSG4iATHHE8RZfaeBW061Hk1/108um
k74kyf+tjeRmyFEiR8B8wz4KxpnDZ54n1058fw7gvSw3pA090pW1NFQKA3cko85h
2u8KCPZCLqN5FeqswGIYmPYrYHk/7mYjc10pYcSzm0hJr0Bw4NBUG685BL5iZ11
I6hDZZ186jHesAks7ZdYYARL1+E/FHPK5z1Bc4+ZG1Tm/vLSbIzzRkQ5abbaB1XW
GmKu2PY91t+o20PRGwC7rLqkKnsTpBAM3hyylQ4N05bTVQVSYg24G0sXgLN1Kqhm
aW3zA3Y94sg5jrYrQDmUshs1v1WQrQxR55UA8oEDLxZh1pN7s4XtaW1vGj2SV5
G5mgt9/QC4TmPRZHPSPYgEb5RcEjJy9zD1s2NtJy5Lox8x8JNf6Zum1d05GFCRp
Eb3vNHmhwl1bQmAnfZK14ZCY/ULNprUjUmQQQ6vR0JrGgWV17qYXoG2Wn0pG+devWY
j6mGy/egT2VGebd4VyQLr58gEBn646PNJ50LTf+AEfIyb2Fxa7UbxVZ82Ik1jY
qWb22n97MvC+djYoWruH903jTe8kcRZYH0PqjLZ6W9jkZbJWUw0wterZ77Fa1
f5OFL54zHXCKfCoB0pH1zu1wTeNF+EQHx3LHLzwBf7BuN96yVJ9IFG5oDCU2H
zhH3xQQ2T0qg/h7nurXUHQ1VnnvHfFFY8q6x80kATY22kH/hUBJF9k8g6S5c4E4
QQFPKka1ePu6YC1B6gn1WBDsdZE4fcpzBW8TjCjMtZBWSRtxSYwWlz5YhNqjn9r1
26z0KME1jB8+pw4b5RB68MCEY7XqIPGXs16jWqhQmx14TeRIE6hWryAmYjvhVh
Yfg9JPEjdm/SrL04FZ6zVb13AUXbj10xqjYnk6bfsST0g+1Eu7qr4/zPIUQfP/a
mM40Bx6Nep6Y8Q/iKV+pkXZNYSCSEKYEUwAG1puwNXIKK+VmwAARBNUzSszZ/
TSAFTF4W1i0JuK1S6nHygfyf8YFu/miXqHo1/b1NxEpJYPr0FQ==
-----END RSA PRIVATE KEY-----
1. sor, 1. oszlop 100% Windows (CRLF) UTF-8
```

(a) Privát kulcs PEM fájlja titkosítva

```
csabapub_titkos.pem - Jegyzetfömb
Fájl Szerkesztés Formátum Nézet Súgó
-----BEGIN PUBLIC KEY-----
Proc-Type: 4, ENCRYPTED
DEK-Info: DES-EDE3-CBC, 556EAB7D884D98C6

0BM6D38J3Wj5FeHhUK2+PzJxQNoHMYeZ8xGbZvTKsk5FHU75xw5ubUktwQrzo
jCP1X5zeffmfg312yZrmuYw/adZuAN1ePsnFFm12okHgdve4kkir+vzJpmgildITo
wsGcGKwKd6gYw+03o3FwF0dImdrAyAD/G1DH5M0ZewU9eWaj7eFpq0vTzH3sYj
LvC2D020SGesDWSGTPPgYsgTvh34FuzdMgWPHCNrsz1R61cmmaa5CmG23n8z0xs
RAmmdKkxyMabzFx+FkuxKRxvz01gE2orsSIUmdHhBYCY5YddFmR59rhJaiF8D8
2wPOBE602VjEb9T9nfvp0CCw5r40LSZ7F735Xwe11Qjn+50P7uwW81gw57scg
jYeFvcq7bgc=
-----END PUBLIC KEY-----
12. sor, 25. oszlop 100% Windows (CRLF) UTF-8
```

(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 módszer

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 módszerben 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* módszer 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 Standards-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 létrehozuk 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 állítjuk.

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%A1fia_i_hash_f%C3%BCggv%C3%A9ny
- [8] https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html