Поддержка токенов PKCS#11 с ГОСТ-криптографией в Python.

I. Сборка и установка модуля pyp11

Итак, скачиваем архив и распаковываем его. Заходим в папку PythonPKCS11 и выполняем команду установки:

```
#python3 setup.py install
```

Лично я тестировал на платформах Windows, Linux, OS X. Отметим, что пакет TclPKCS11 успешно работает и на платформе Android.
После установки модуля переходим в папку tests и начинаем тестирование.
Работоспособность модуля pyp11 можно проверить даже без токена. В составе модуля есть функция pyp11.dgst, которая не привязана к токенам и позволяет посчитать хэш по ГОСТ Р 34.10-2012:

```
bash-4.4$ python3
Python 3.7.9 (default, Feb  1 2021, 16:55:33)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyp11
#Считаем хэш по ГОСТ Р 34.11-2012-256 (stribog256)
>>> hash256 = pyp11.dgst("stribog256", "Текст для хэширования")
#Считаем хэш по ГОСТ Р 34.11-2012-512 (stribog512)
>>> hash512 = pyp11.dgst("stribog512", "Текст для хэширования")
>>> print("STRIBOG256=" + hash256)
STRIBOG256=26b8865c37831aa254706e6c3514fb23f386358e9dd858703a24d4825d2c4794
>>> print("STRIBOG512=" + hash512)
STRIBOG512=e92ff2063c586ec6e9c9569dad7dd503de1c88faafc8b1bf43909bfa36db92ccbf382
3f0b8f5d877f10933ed7e670081018dac0929d17729422f05ce1f4c4f25
>>> quit()
bash-4.4$
```

Значение хэш возвращается в шестнадцатеричном виде.
Для перевода хэш-а в бинарный вид можно воспользоваться следующей функцией:

```
>>> hash256_bin = bytes(bytearray.fromhex(hash256))
```

Напомним, как перевести бинарный код в шестнадцатеричный:

```
>>> hash256 = bytes(hash256_bin).hex()
>>> print("STRIBOG256_NEW=" + hash256)
STRIBOG256_NEW=26b8865c37831aa254706e6c3514fb23f386358e9dd858703a24d4825d2c4794
>>>
```

Есть еще одна функция, которая также может работать без токена. Это функция parsecert. На

вход этой функции подается сертификат в DER-формате, упакованный в шестнадцатеричную кодировку:

```
bash-4.4$ python3
Python 3.7.9 (default, Feb  1 2021, 16:55:33)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyp11
>>> #Читаем серификат в DER-кодировке из файла
>>> with open("cert_256.der", "rb") as f:
...     cert_der = f.read()
...
>>> #Упаковываем сертификат der в hex
>>> cert_der_hex = bytes(cert_der).hex()
>>> #Распарсиваем сертификат
>>> pubk = pyp11.parsecert(cert_der_hex)
>>>
```

Результатом выполнения команды pyp11.parsecert является словарь (ассоциированный список):

```
>>>print (pubk.keys())
dict_keys(['pkcs11_id', 'pubkeyinfo', 'pubkey', 'subject', 'issuer',
'serial_number', 'tbsCertificate', 'signature_algo', 'signature'])
>>>
```

В этом словаре находятся as1-структуры элементов сертификата. Все элементы закодированы в шестнадцатеричный формат. Среди элементов находится элемент pubkeyinfo со значением asn1-структуры subjectpublickeyinfo, элемент pubkey со значением публичного ключа, серийный номер сертификата, tbs-сертификат, который будет использоваться для проверки подписи сертификата, алгоритм подписи сертификата и значение самой подписи, а также элементы с информацией о владельце и издателе сертификата, полученные из сертификата и закодированные в шестнадцатеричное представление:

```
>>> subject = pubk['subject']
>>> print ('SUBJECT=' + subject)
SUBJECT=30820205310b3009060355040613025255312a3028060355042a0c21d09fd0b0d0b2d0b5
d0bb20d090d0bdd0b0d182d0bed0bbd18cd0b5d0b2d0b8d1873135303306035504030c2cd09ed09e
d09e20d09ad09ed09cd09fd090d09dd098d0af20d0add09ad09e2dd0a1d0a2d0a0d09ed099203937
311d301b06092a864886f70d010901160e696e666f4072746564632e6f72673118301606052a8503
6401120d3131373737343637333434333931163014060 52a85036403120b31333836323135373737
34311a301806082a85030381030101120c303039373239313130353639313030 2e060355040c0c27
d093d0b5d0bdd0b5d180d0b0d0bbd18cd0bdd18bd0b920d0b4d0b8d180d0b5d0bad182d0bed18031
0a3008060355040b0c0130313530330 60355040a0c2cd09ed09ed09e20d09ad09ed09cd09fd090d0
9dd098d0af20d0add09ad09e2dd0a1d0a2d0a0d09ed099203937315f305d06035504090c56313139
31333620d0b32e20d09cd0bed181d0bad0b2d0b020d0bfd1802dd0b420312dd0b920d0a1d0b5d182
d183d0bdd18cd181d0bad0b8d0b920d0b42e203130d09020d181d182d1802e203120d0bfd0bed0bc
2e203231153013060355040 70c0cd09cd0bed181d0bad0b2d0b0311c301a06035504080c13373720
d0b32e20d09cd0bed181d0bad0b2d0b0311b301906035504040c12d0a5d0b0d180d0b8d182d0bed0
bdd0bed0b2
>>>
```

Элемент pkcs11_id берётся не из сертификата, а рассчитывается как значение хэш по SHA-1 от значения публичного ключа. При использовании функции pyp11.parsecert в данном контексте (без подключенного токена) pkcs11_id будет равен -1:

```
>>> pkcs11_id = pubk['pkcs11_id']
>>> print ('PKCS11_ID=' + pkcs11_id)
PKCS11_ID=-1
>>>
```

Кто-то может сказать, а что, разве в Python нет средств разбора сертификатов? А как же, например, asn1crypto? Ответ заключается в том, что в этих средствах не учтены особенности российской криптографии. И вот, чтобы получить максимальную самодостаточность пакета pyp11, в него помимо функций, связанных с генерацией ключевой пары, формирования и проверки подписи, включены дополнительные функции. Например, asn1-структура pubkeyinfo необходима при проверке электронной подписи. И именно поэтому и была включена функция parsecert для частичного разбора сертификата x509.v3 и получения, в частности, asn1-структуры subjectpublickeyinfo (pubkeyinfo).
В папке tests проекта в файлах test0_* находятся соответствующие тесты.
############УБРАТЬ про FSB795 ################################
Отметим также, что для разбора сертификатов с российской криптографией можно воспользоваться пакетом fsb795:

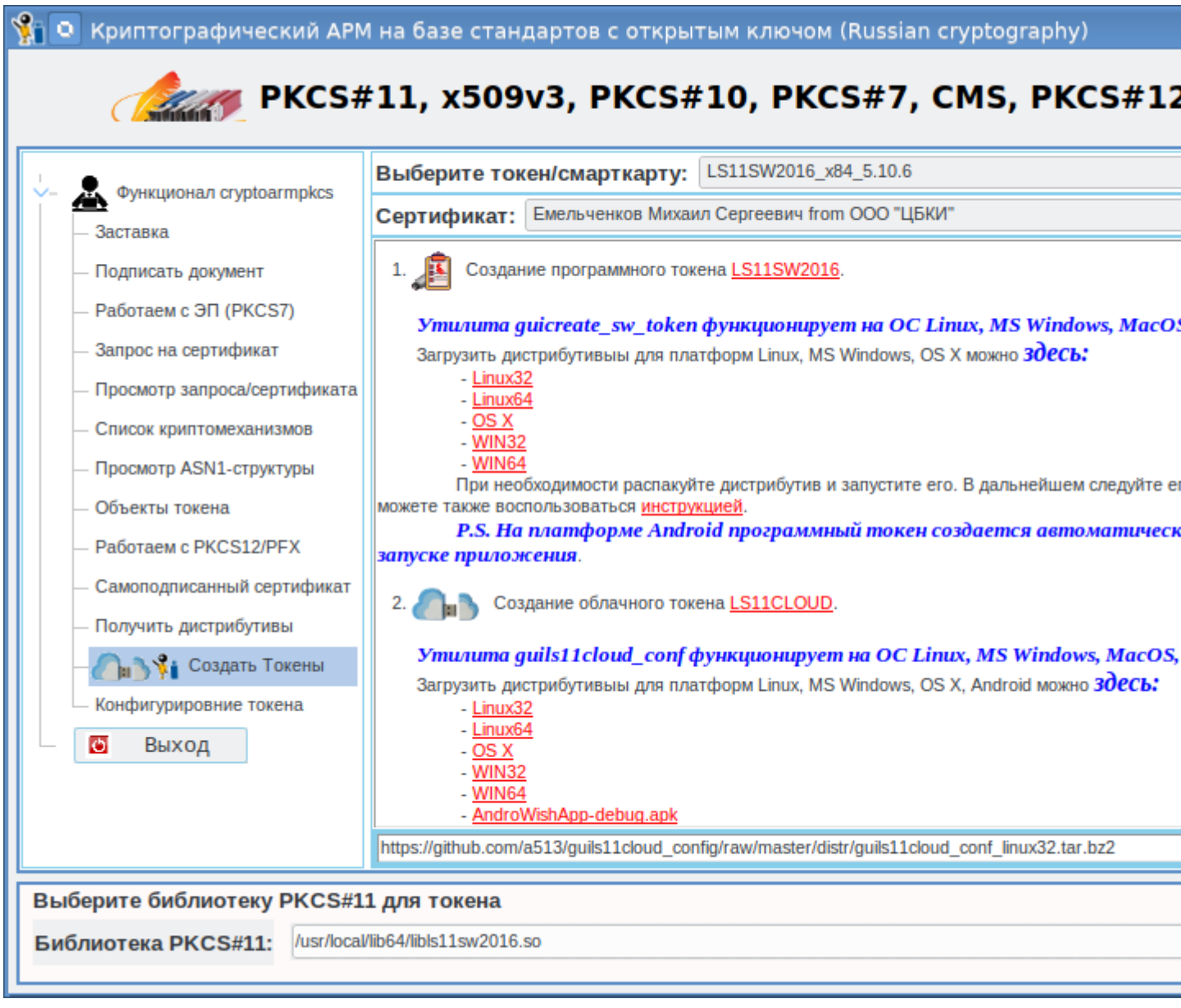
```
>>> import fsb795
>>> #Парсим наш сертификат с помощью fsb795
>>> mycert = fsb795.Certificate(cert_der)
>>> #читаем данные о владельце сертификата и типе владельце
>>> dn, type = mycert.subjectCert()
>>> #DN - это словарь/ассоциированный список
>>> for key in dn.keys():
...     print (key + '=' + dn[key])
...
Country=RU
GN=Имя Отчество
CN=ООО КОМПАНИЯ
E=info@ooo.org
OGRN=xxxxxxxxxxxx
SNILS=xxxxxxxxxxx
INN=xxxxxxxxxxxx
title=Генеральный директор
OU=0
O=ООО КОМПАНИЯ
street=119136 г. Москва
L=Москва
ST=77 г. Москва
SN=Харитонов
>>>
```

Теперь можно переходить к работе с токенами.

II. Управление токенами PKCS#11

Для тестирования функций управления подойдет любой токен PKCS#11, даже токен без поддержки какой-либо криптографии, например RuTokenLite. Но поскольку мы ведём речь о российской криптографии, то целесообразно сразу иметь токен с поддержкой российской криптографии. Здесь мы имеем в виду ГОСТ Р 34.10-2012 и ГОСТ Р 34.11-2012. Это может быть как аппаратный токен, например RuTokenECP-2.0, так и программные или облачные

токены.
Установить программный токен или получить доступ к облачному токену можно, воспользовавшись утилитой cryptoarmpkcs.
Скачать утилиту cryptoarmpkcs можно здесь.
После запуска утилиты необходимо зайти на вкладку «Создать токены»:



На вкладке можно найти инструкции для получения токенов.
Итак, у нас токен и библиотека для работы с ним. После загрузки модуля pyp11 требуется загрузить библиотеку для работы с нашим токеном. В примерах будут использоваться библиотека librtpkcs11ecp-2.0 для работы с аппаратным токеном, библиотека libls11sw2016 для работы с программным токеном и библиотека libls11cloud.so для работы с облачным токеном. Читатели могут использовать любые токены, даже те, на которых нет российской

криптографии, на них тоже можно проверить функции управления.
Итак, загружаем библиотеку командой loadmodule:

```
bash-4.4$ python3
Python 3.7.9 (default, Feb  1 2021, 16:55:33)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> import pyp11
>>> #Выбираем библиотеку pkcs11
>>> lib = "/usr/local/lib64/librtpkcs11ecp_2.0.so"
>>> #Обработка ошибки при загрузке библиотеки PKCS#11
>>> try:
... #Вызываем команду загрузки библиотеки и плохим числом параметров
...     handlelib = pyp11.loadmodule(lib, 2)
... except:
...     print ('Ошибка загрузки библиотеки: ')
...     e = sys.exc_info()[1]
...     e1 = e.args[0]
...     print (e1)
...
Ошибка загрузки библиотеки:
pyp11_load_module args error (count args != 1)
>>> #Загружаем с правильным синтаксисом
>>> idlib = pyp11.loadmodule(lib)
>>> #Печатаем дескриптор библиотеки
>>> print (idlib)
pkcs0
>>>
```

Дескриптор загруженной библиотеки используется при её выгрузке:

```
>>> pyp11.unloadmodule(idlib)
```

Теперь, когда библиотека загружена, можно получить список поддерживаемых её слотов и узнать есть ли в каких слотах токены. Для получения списка слотов с полной информацией о них и содержащихся в них токенах используется команда:

```
>>> slots = pyp11.listslots(idlib)
>>>
```

Команда pyp11.listslots возвращает список, каждый элемент которого содержит информацию о слоте:

```
[<info slot1>, <info slot2>, ... , <info slotN>]
```

.
В свою очередь, каждый элемент этого списка также является списком, состоящим из четырех элементов:

```
[<номер слота>, <метка токена, находящегося в слоте>, <флаги слота и токена>,
<информация о токене>]
```

Если слот не содержит токен, то элементы <метка токена ...> и <информация о слоте>

содержат пустое значение.
Наличие токена в слоте определяется по наличию флага TOKEN_PRESENT в списке <флаги слота и токена>:

```
bash-4.4$ python3
Python 3.7.9 (default, Feb  1 2021, 16:55:33)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> import pyp11
>>> #Выбираем библиотеку
>>> #lib = '/usr/local/lib64/libls11sw2016.so'
>>> lib = '/usr/local/lib64/librtpkcs11ecp_2.0.so'
>>> #Загружаем библиотеку
>>> libid = pyp11.loadmodule(lib)
>>> #Дескриптор библиотеки
>>> print (libid)
pkcs0
>>> #Загружаем список слотов
>>> slots = pyp11.listslots(libid)
>>> tokpr = 0
>>> #Ищем первый подключенный токен
>>> while (tokpr == 0):
... #Перебираем слоты
...     for v in slots:
...         #Список флагов текущего слота
...         flags = v[2]
... #Проверяем наличие в стоке токена
...         if (flags.count('TOKEN_PRESENT') !=0):
...             tokpr = 1
... #Избавляемся от лишних пробелов у метки слота
...             lab = v[1].strip()
...             infotok = v[3]
...             slotid = v[0]
...             break
...     if (tokpr == 0):
...         input ('Нет ни одного подключенного токена.\nВставьте токен и
нажмите ВВОД')
...     slots = pyp11.listslots(libid)
... #Информация о подключенном токене
...
Нет ни одного подключенного токена.
Вставьте токен и нажмите ВВОД
''
>>> #Информация о подключенном токене
>>> print ('LAB="' + lab + '", SLOTID=' + str(slotid))
LAB="Rutoken lite <no label>", SLOTID=0
>>> print ('FLAGS:', flags)
FLAGS: ['TOKEN_PRESENT', 'RNG', 'LOGIN_REQUIRED', 'SO_PIN_TO_BE_CHANGED',
'REMOVABLE_DEVICE', 'HW_SLOT']
>>>
```

Если взглянуть на флаги (FLAGS:) подключенного токена, то в них отсутствует флаг 'TOKEN_INITIALIZED'. Отсутствие этого флага говорит о том, что токен не инициализирован и требуется его инициализация:

```
#Проверяем, что токен проинициализирован
>>> if (flags.count('TOKEN_INITIALIZED') == 0''):
...         #Инициализируем токен
...         dd = pyp11.inittoken (libid, 0, '87654321',"TESTPY2")
```

```
...
>>>
```

Как видим, для инициализации токена используется следующая команда:

```
pyp11.inittoken (<дискриптор библиоткети>, <номер слота>, <SO-PIN>, <метка токена>)
```

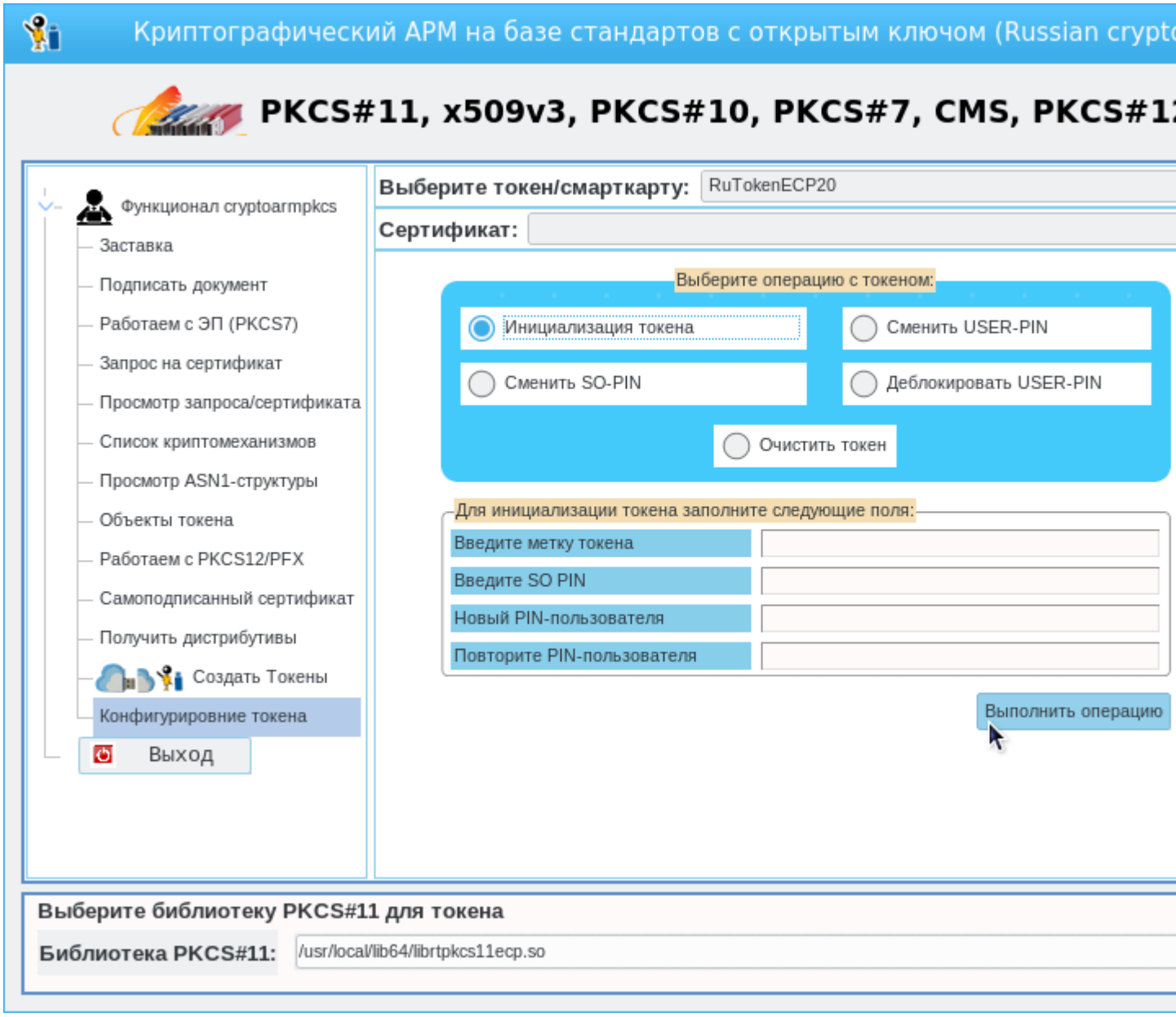
Естественно, токен можно переинициализировать независимо от наличия флага 'TOKEN_INITIALIZED', только надо иметь в виду, что переинициализация токена ведет к уничтожению на нем всех объектов (ключи, сертификаты и т.д).
После инициализации токена должен быть проинициализирован USER-PIN. Эту операцию, как правило, делает производитель или продавец токена:

```
pyp11.inituserpin (<дискриптор библиоткети>, <номер слота>, <SO-PIN>, <USER-PIN>)
```

При этом выставляется флаг 'USER_PIN_TO_BE_CHANGED', который напоминает владельцу токена, что надо бы сменить свой USER-PIN (параметр 'user'):

```
pyp11.setpin (<дискриптор библиоткети>, <номер слота>, <'user'|'so'>,<текущий PIN-код>, <новый PIN-код>)
```

Сегодня «модно» получать в УЦ токены с закрытыми ключами и предустановленными PIN-кодами и ключевой парой. И, как правило, получателей не предупреждают, что целесообразно PIN-коды поменять, и не говорят, как это сделать. Я бы рекомендовал использовать для этого уже упоминавшуюся утилиту cryptoarmpkcs:



В папке tests проекта pyp11 лежат три теста test1_0_inittoken.py, test1_1_inituserpin.py и test1_2_change_userpin, которые наглядно демонстрируют инициализацию токена. Выполнять их надо в порядке перечисления.
Было бы несправедливо не показать инициализацию токена с использованием уже упоминавшегося пакета PyKCS11:

```
$ python3
Python 3.7.9 (default, Feb  1 2021, 16:55:33)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import PyKCS11
>>> #Библиотека PKCS#11
>>> lib = '/usr/local/lib64/librtpkcs11ecp_2.0.so'
>>> pkcs11.load(lib)
>>> #Получаем список слотов с токенами
>>> slots = pkcs11.getSlotList(tokenPresent=True)
>>> #Ищем первый подключенный токен
>>> while (len(slots) == 0):
...     input ('Нет ни одного подключенного токена.\nВставьте токен и нажмите
ВВОД')
```

```
...     #Получаем список слотов с токенами
...     slots = pkcs11.getSlotList(tokenPresent=True)
...
Нет ни одного подключенного токена.
Вставьте токен и нажмите ВВОД
''
>>>
>>> #Берём первый подключенный токен
>>> slot = slots[0]
>>> #Закрываем все сессии на токене
>>> #SO-PIN
>>> so_pin = '87654321'
>>> lab_tok = "myLabel"
>>> #Инициализация токена
>>> pkcs11.initToken(slot, so_pin, lab_tok)
>>> session = pkcs11.openSession(slot, PyKCS11.CKF_SERIAL_SESSION |
PyKCS11.CKF_RW_SESSION)
>>> #Установка первичного USER-PIN
>>> init_pin = '1234'
>>> session.login(so_pin, user_type=PyKCS11.CKU_SO)
>>> session.initPin(init_pin)
>>> session.logout()
>>> #Новый USER-PIN
>>> user_pin = '01234567'
>>> session.login(init_pin)
>>> # change PIN
>>> session.setPin(init_pin, user_pin)
>>> session.logout()
>>> quit()
$
```

III. Ключевая пара, электронная подпись и её проверка

Итак, наш токен готов в работе: мы его проинициализировали и установили метку, но самое главное, мы поменяли PIN-коды (USER, SO).
Первым делом необходимо убедиться, что наш токен поддерживает необходимые нам криптографические механизмы. Поскольку речь идет о ГОСТ Р 34.10-2012 и ГОСТ Р 34.11-2012, то токен должен поддерживать механизмы CKM_GOST* в соответствии рекомендациями ТК-26.
Для получения списка механизмов используется команда pyp11.lictmechs:

```
<список механизмов> = pyp11.listmech(<идентификатор библиотеки>, <номер слота>)
```

Как ни странно, но токены могут не иметь поддержки криптомеханизмов, например, RuToken Lite. Они нас интересовать не будут. Мы будем использовать только токены с поддержкой ГОСТ Р 34.10-2012 и ГОСТ Р 34.11-2012:

```
$ python3
Python 3.7.9 (default, Feb  1 2021, 16:55:33)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyp11
>>> #Выбираем библиотеку
>>> #Программный токен
>>> lib = '/usr/local/lib64/libls11sw2016.so'
```

```
>>> #Для Windows
>>> #lib='C:\Temp\ls11sw2016.dll'
>>> #Облачный токен
>>> #lib = '/usr/local/lib64/libls11cloud.so'
>>> #Аппаратный токен
>>> #lib = '/usr/local/lib64/librtpkcs11ecp_2.0.so'
>>> #Закгружаем выбранную библиотеку
>>> aa = pyp11.loadmodule(lib)
>>> listmech = pyp11.listmechs(aa, 0)
>>> print ('\tКриптографические механизмы токена')
        Криптографические механизмы токена
>>> for mech in listmech:
...     print(mech)
...
```

Криптографические механизмы

```
>>> quit()
$
```

Теперь, когда мы убедились, что токен поддерживает российскую криптографию, можно приступить к созданию ключевой пары на токене и использовать ее закрытый ключ для подписания различных документов.
Напомним, что закрытый и открытый ключи это не только их значения (для открытого ключа ГОСТ Р 34.10-2012-256 это 512 бит, а для открытого ключа ГОСТ Р 34.10-2012-512 это 1024 бита), но и параметры схемы цифровой подписи (п. 5.2 ГОСТ Р 34.10-2012). В дальнейшем параметры схемы цифровой подписи для простоты будем называть параметрами (криптопараметрами) ключевой пары.
Криптопараметры при генерации ключевой пары задаются OID-ами. В настоящее время ТК-26 определил следующие oid-ы для криптопараметров алгоритма подписи ГОСТ Р 34.10-2012 с ключом 256:

- 1.2.643.7.1.2.1.1.1 (id-tc26-gost-3410-12-256-paramSetA);
- 1.2.643.7.1.2.1.1.2 (id-tc26-gost-3410-12-256-paramSetB;
- 1.2.643.7.1.2.1.1.3 (id-tc26-gost-3410-12-256-paramSetC);
- 1.2.643.7.1.2.1.1.4 (id-tc26-gost-3410-12-256-paramSetD).

При этом продолжают действовать так называемые OID-ы параметров от КриптоПро:

- 1.2.643.2.2.35.1 (id-GostR3410-2001-CryptoPro-A-ParamSet);
- 1.2.643.2.2.35.2 (d-GostR3410-2001-CryptoPro-B-ParamSet);
- 1.2.643.2.2.35.3 (id-GostR3410-2001-CryptoPro-C-ParamSet);
- 1.2.643.2.2.36.0 (id-GostR3410-2001-CryptoPro-XchA-Param)Set;
- 1.2.643.2.2.36.1 (id-GostR3410-2001-CryptoPro-XchB-Param)Set.

Кто-то может сказать, а что это за каша такая? Но если смотреть по сути, то окажется, что параметры КриптоПро с OID-ами 1.2.643.2.2.35.1, 1.2.643.2.2.35.2, 1.2.643.2.2.35.3 соответствуют параметрам ТК-26 с OID-ами 1.2.643.7.1.2.1.1.1, 1.2.643.7.1.2.1.1.2, 1.2.643.7.1.2.1.1.3 соответственно. Далее ещё интереснее. Параметр КриптоПро id-GostR3410-2001-CryptoPro-XchA-Param соответствует параметру id-GostR3410-2001-

CryptoPro-A-ParamSet, а параметр id-GostR3410-2001-CryptoPro-XchB-Param — параметру id-GostR3410-2001-CryptoPro-C-ParamSet того же КриптоПро. Если не запутались, то идём дальше.
С криптопараметрам для алгоритма подписи ГОСТ Р 34.10-2012 с ключом 512 проще:

- 1.2.643.7.1.2.1.2.1 (id-tc26-gost-3410-2012-512-paramSetA);
- 1.2.643.7.1.2.1.2.2 (id-tc26-gost-3410-2012-512-paramSetB);
- 1.2.643.7.1.2.1.2.3 (id-tc26-gost-3410-2012-512-paramSetC);

Для генерации ключевой пары используется следующая команда:

```
<идентификатор словаря> = pyp11.keypair(<идентификатор библиотеки>, <номер слота с токеном>, <тип ключевой пары>, <OID криптопараметра>, <метка/CKA_LABEL>)
```

Единственное, с чем мы не сталкивались, это <тип ключевой пары>:
<тип ключевой пары> := 'g12_256' | 'g12_512'
Таким образом, если мы хотим получить пару по алгоритму подписи ГОСТ Р 34.10-2012 с ключом 512, то задаем тип 'g12_512', например:

```
genkey = pyp11.keypair(libid, slotid, 'g12_512', '1.2.643.7.1.2.1.2.2', 'KeyGost512')
```

Для алгоритма подписи ГОСТ Р 34.10-2012 с ключом 256 генерация может выглядеть так:

```
genkey256 = pyp11.keypair(libid, slotid, 'g12_256', '1.2.643.7.1.2.1.1.3', 'KeyGost256')
```

Перед генерацией ключевой пары необходимо обязательно залогиниться на токене:

```
pyp11.login(<идентификатор библиотеки>, <номер слота>, 'USER-PIN')
```

После выполнения требуемой операции целесообразно выполнить logout:

```
pyp11.logout(<идентификатор библиотеки>, <номер слота>)
```

При успешной генерации ключевой пары возвращается ассоциированный список (словарь), например:

```
>>> pyp11.login(libid, slotid, '01234567')
1
>>> genkey256 = pyp11.keypair(libid, slotid, 'g12_256', '1.2.643.7.1.2.1.1.3', 'KeyGost256')
>>> print (genkey256.keys())
dict_keys(['pkcs11_handle', 'pkcs11_slotid', 'hobj_pubkey', 'hobj_privkey', 'pkcs11_id', 'pkcs11_label', 'pubkey', 'pubkey_algo', 'pubkeyinfo', 'type'])
>>> pyp11.logout(libid, slotid)
1
>>>
```

Среди возвращаемых значений находятся указатели на открытый ('hobj_pubkey') и закрытый ключи ('hobj_privkey'). Последний мы будем использовать при подписании. Среди возвращаемых значений находится и CKA_ID открытого и закрытого ключей ('pkcs11_id'). Элемент pkcs11_id также может использоваться при подписании для поиска закрытого ключа. Напомним, CKA_ID это значение хэша SHA-1 от значения открытого ключа, которое находится в элементе 'pubkey'. При генерации ключевой пары CKA_ID автоматически выставляется для закрытого и открытого ключей. Именно по ним, как правило, ищут соответствие между ключами. Можно распечатать все возвращаемые значения:

```
>>> for key in genkey256.keys():
...     print (key + '= ' + str(genkey256.get(key)))
...
pkcs11_handle= pkcs0
pkcs11_slotid= 0
hobj_pubkey= hobj0100000000000000
hobj_privkey= hobj0200000000000000
pkcs11_id= dd22fe35aeb7eb2ebcad7199b117eb3a7b5f5813
pkcs11_label= KeyGost256
pubkey=
4c2ed60bc5771b2a6616af58c8dd202b9463dde9bd1de028335e718634761e360a25b2f337c2e67c
28402cd49fff4f708130a80dc479301b21ceb9324c47464b
pubkey_algo= 1 2 643 7 1 1 1 1
pubkeyinfo=
302106082a85030701010101301506092a85030701020101030608 2a85030701010202034300044 0
4c2ed60bc5771b2a6616af58c8dd202b9463dde9bd1de028335e718634761e360a25b2f337c2e67c
28402cd49fff4f708130a80dc479301b21ceb9324c47464b
type= pkcs11
>>>
```

Для формирования электронной подписи и её проверки сохраним следующие значения:

```
>>> hprivkey = genkey256.get("hobj_privkey")
>>> pkcs11_id = genkey256.get("pkcs11_id")
>>> pubkeyinfo = genkey256.get("pubkeyinfo")
>>>
```

Электронная подпись (ЭП) документа представляет собой подписанный хэш от этого документа.
Поэтому сначала считается соответствующий хэш:
<переменная для хранения хэш > = pyp11.digest(<идентификатор библиотеки>, <слот токена>, 'stribog256' | 'stribog512', <документ>)
Значение хэш всегда возвращается в шестнадцатеричном виде.
Итак, если мы хотим получить подпись по алгоритму ГОСТ Р 34.10-2012 с ключом 256 бит, то нам сначала надо посчитать хэш по алгоритму хэширования ГОСТ Р 34.11-2012 с длиной 256 бит, а затем подписать полученный хэш с использованием механизма 'CKM_GOSTR3410':

```
<переменная для  ЭП> = pyp11.sign(<идентификатор библиотеки>, <слот токена>,
'CKM_GOSTR3410' | 'CKM_GOSTR3410_512', <хэш документа>, <hobj_privkey|
pkcs11_id>)
```

Хэш документа должен быть в шестнадцатеричном виде. Электронная подпись также

возвращается в шестнадцатеричном виде.

Ниже приведем пример кода формирования :

```
bash-4.4$ python3
Python 3.7.9 (default, Feb  1 2021, 16:55:33)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyp11
>>> lib = '/usr/local/lib64/libls11sw2016.so'>>> libid =
pyp11.loadmodule(lib)>>> slotid = 0
>>> #Незабывайте лигиниться на токене!!!
>>> pyp11.login(libid, slotid, '01234567')
1
>>> genkey256 = pyp11.keypair(libid, slotid, 'g12_256', '1.2.643.7.1.2.1.1.3',
'KeyGost256')
>>> hprivkey = genkey256.get("hobj_privkey")
>>> pkcs11_id = genkey256.get("pkcs11_id")
>>> pubkeyinfo = genkey256.get("pubkeyinfo")
>>> hashdoc_hex = pyp11.digest(libid, slotid, 'stribog256', 'Подписываемый
документ')
>>> #Для ЭП используется hobj_privkey
>>> sign1 = pyp11.sign(libid, slotid, 'CKM_GOSTR3410', hashdoc_hex, hprivkey)
>>> #Для ЭП используется pkcs11_id (CKA_ID)
>>> sign2 = pyp11.sign(libid, slotid, 'CKM_GOSTR3410', hashdoc_hex, pkcs11_id)
>>> print ('SIGN1=' + sign1 )
SIGN1=5b3f881153f50d9a8da6bb37bb83f54fe997d074672c29c2c0aeb22739a14f1e776b8427e2
62b098c75abe3a4faffe383d3e2cc406afa09efb3e783919b4ca11
>>> print ('SIGN2=' + sign2)
SIGN2=441a29206a3622a9c76282b71b4fcdbf4c15034d0f0be7b1f711c6d5eef8162a2a2876a5d3
75cb56e23fc76173cacf88b620fd793cf756589a76cbee6b1fd27a
>>> pyp11.logout(libid, slotid)
1
>>>
```

Для проверки подписи используется asn1-структура открытого ключа subjectPublicKeyInfo, которую мы сохранили после генерации ключевой пары в переменной pubkeyinfo:

```
 pubkeyinfo = genkey256.get("pubkeyinfo")
```

Для проверки подписи используется следующая команда:

```
pyp11.sign(<идентификатор библиотеки>, <слот токена>, <хэш документа в hex>,
<подпись документа в hex>, <asn1-subjectPublicKeyInfo>)
```

Команда возвращает 1 (единицу), если подпись прошла проверку, и 0 (ноль), если проверка не прошла.

Продолжим наш пример проверкой двух полученных подписей:

```
>>> verify1 = pyp11.verify(libid, slotid, hashdoc_hex, sign1, pubkeyinfo)
>>> print (verify1)
1
>>> verify2 = pyp11.verify(libid, slotid, hashdoc_hex, sign2, pubkeyinfo)
>>> print (verify2)
1
>>>
```

Как видим обе полученные подписи корректны.

IV. Проверка электронной подписи сертификата

Используя полученные знания, напишем пример проверки электронной подписи сертификата:

```
#!/usr/bin/python3
#-*- coding: utf-8 -*-
import pyp11
print('Проверка подписи сертификата')
#Библиотека для токена
lib = '/usr/local/lib64/libls11sw2016.so'
aa = pyp11.loadmodule(lib)
print (aa)
#Файл с корневым сертификатом в DER-кодировке
fileCA = "CA_12_512.der"
#Файл с сертификатом пользователя в DER-кодировке
fileUser = "habrCA_12_512.der"
#Читаем корневой сертификат в DER-кодировке из файла
with open(fileCA, "rb") as f:
    certCA = f.read()
#Упаковываем der в hex
certCA_hex = bytes(certCA).hex()
#Читаем сертификат пользователя в DER-кодировке из файла
with open(fileUser, "rb") as f:
    certHabr = f.read()
#Упаковываем der в hex
certHabr_hex = bytes(certHabr).hex()
print ('Разбираем корневой сертификат')
parseCA = pyp11.parsecert (aa, 0, certCA_hex)
print ('Разбираем сертификат пользователя')
parseHabre = pyp11.parsecert (certHabr_hex)
print (parseHabre.keys())
#Проверяем, что издатель сертификата совпадает с владельцем корневого
сертификата
if (parseCA.get('subject') != parseHabre.get('issuer')):
    print ('Сертификат выдан на другом УЦ')
    quit()
print ('Сертификат выдан на данном УЦ')
#Переводим tbsCertificate пользователь в binary
tbs_hex = parseHabre.get('tbsCertificate')
tbsHabrDer = bytes(bytearray.fromhex(tbs_hex))
#tbsHabrDer = '1111'
#Получаем хэш для tbs-сертификата
hashTbs_hex = pyp11.digest(aa, 0, "stribog512", tbsHabrDer)
#hashTbs_hex = pyp11.digest(aa, 0, "stribog256", tbsHabrDer)
verify = pyp11.verify(aa, 0,  hashTbs_hex, parseHabre.get('signature'),
parseCA.get('pubkeyinfo'))
#verify = pyp11.verify(aa, 0,  hashTbs_hex, parseHabre.get('signature'),
parseHabre.get('pubkeyinfo'))
print (verify)
if (verify != 1):
    print ('Подпись сертификата не прошла проверку')
    quit()
print ('Подпись сертификата прошла проверку')
quit()
```

V. Работа с объектами токена

Основными объектами, с которыми приходится иметь дело, работая с токенами PKCS#11, являются сертификаты и ключи. И те и другие имеют атрибуты. Нас в первую очередь интересуют атрибуты CKA_LABEL или метка объекта и CKA_ID или идентификатор объекта. Именно атрибут CKA_ID используется для доступа и к сертификатам и ключам.
Уже имея в своем распоряжении рассмотренные выше команды модуля pyp11, можно создать ключевую пару и сформировать подписанный запрос на сертификат. Отправить полученный запрос в удостоверяющий центр и получить там сертификат. Но получив сертификат, возникает вопрос как его поставить на токен и привязать к ключевой паре? Именно эту задачу решает команда pyp11.importcert:

```
<переменная для CKA_ID> = pyp11.importcert(<идентификатор библиотеки>, <слот токена>, <сертификат в DER-формате и hex-кодировке>, <метка CKA_LABEL>)
```

Как работает команда? Первым делом она вычисляет по открытому ключу сертификата идентификатор CKA_ID. Именно этот идентификатор будет возвращен в hex-кодировке после успешного размещения сертификата на токене. После установки сертификата на токен в DER-формате, устанавливаются его атрибуты CKA_ID и CKA_LABEL.
Если вам необходимо связать тройку <сертификат> x <открытый ключ> x <закрытый ключ> не только по CKA_ID, но и по метке CKA_LABEL, то необходимо установить метку у ключевой пары аналогичную метке сертификата. Для этого используется команда rename:

```
pyp11.rename(<идентификатор библиотеки>, <слот токена>, <тип объекта>, <ассоциированный список>)
```

В <типе объекта> указывается, к каким типам объектов будет применяться команда: 'cert' | 'key' | 'all' (сертификаты, ключевая пара, к тому и другому).
Команда rename позволяет менять не только CKA_LABEL, но и CKA_ID. Конкретные объекты могут задаваться идентификаторами объектов CKA_ID (pkcs11_id), например:

```
#Импортируем сертификат и получаем его CKA_ID
labcert = 'LabelNEW'
ckaid = pyp11.importcert(aa, 0, cert_der_hex, labcert)
#Устанавливаем метку сертификата и для ключей
#Готовим словарь
ldict = dict(pkcs11_id=ckaid, pkcs11_label=labcert)
#Меняем метки у ключей
pyp11.rename(aa, 0, 'key', ldict)
```

Аналогичным образом меняется атрибут CKA_ID. В этом случае в словарь вместо метки указывается новый CKA_ID:

```
ldict = dict(pkcs11_id=ckaid, pkcs11_id_new=11111)
```

Аналогичным образом можно удалить объекты:

```
pyp11.delete(<идентификатор библиотеки>, <слот токена>, <тип объекта>, <ассоциированный список>)
```

При уничтожении в словарь попадает только один элемент, который будет указывать на удаляемые объекты. Это либо CKA_ID (ключ pkcs11_id) либо непосредственно handle-объекта (как правило, его можно получить по команде pyp11.listobjects, ключ pkcs11_handle):

```
ldict = dict(pkcs11_id=ckaid)
#Или с handle-объекта:
#ldict = dict(hobj=pkcs11_handle)
#Уничтожить личный сертификат с ключами
pyp11.login(aa. 0, '01234567')
pyp11.delete(aa, 0, 'all', ldict)
pyp11.logout(aa, 0)
```

Упомянем еще об одной очень редко используемой команде. Это команда закрытия сессий на токене:

```
pyp11.closesession(<идентификатор библиотеки>)
```

Эту команду следует вызывать, когда возникнет ошибка «PKCS11_ERROR SESSION_HANDLE_INVALID», а затем повторить команду, на которой возникла ошибка. Эта ошибка может возникнуть при кратковременном извлечении токена из компьютера при работе вашей программы.
И завершим мы рассмотрение командой pyp11.listcertsder:

```
<список сертификатов> = pyp11.listcerts(<идентификатор библиотеки>, <слот токена>)
```

Вот пример кода:

```
#!/usr/bin/python3
#-*- coding: utf-8 -*-
import sys
import time
import pyp11
print('Список сертификатов токена')
aa = pyp11.loadmodule('/usr/local/lib64/libls11sw2016.so')
lcerts = pyp11.listcerts(aa, 0)
if (len(lcerts) == 0):
    print ('На токене нет сертификатов')
    quit()
#Перебираем сертификаты
for cert in lcerts:
    #Информация о сертификате
    for key in cert:
        print (key + ': ' + cert[key])
#Сравним с pyp11.listobjects
lm = pyp11.listobjects(aa, 0, 'cert', 'value')
print('Работа с listobjects:')
for obj in lm:
    for key in obj:
        print (key + ': ' + obj[key])
quit()
```

Команды pyp11.listobjects для сертификатов и команда pyp11.listcerts фактически дублируют друг друга, но так сложилось исторически.