From: pqc-forum@list.nist.gov on behalf of D. J. Bernstein <djb@cr.yp.to>
Sent: Monday, July 17, 2023 1:09 PM
To: pqc-comments
Cc: pqc-forum
Subject: [pqc-forum] OFFICIAL COMMENT: KAZ-SIGN
Attachments: signature.asc

Running the Sage script below in the
KAZ-SIGN/Reference_Implementation/kaz458 directory rapidly forges a signature on any desired message under essentially any desired public key, and checks that the signature passes verification with the reference crypto_sign_open() software.

The script uses a particular message and the first public key in *.rsp as an example, but I've also tested it with another random public key and with various further messages. The reason I'm saying "essentially"
is that the KAZ-SIGN integer encoding looks like it will fail for 1/256 of all possible inputs; the reference software doesn't seem to handle this, and this Sage script also doesn't handle this.

---D. J. Bernstein

```
#!/usr/bin/env sage

import os
import subprocess
import ctypes
from ctypes import c_int,c_char_p,c_ulonglong,POINTER,byref,create_string_buffer
import hashlib
import random
def hash(seed): h = hashlib.sha256(); h.update(seed); return h.digest()

proof.all(False)

# ----- copied from kaz_api.h

N =
37470874733837919416563211326754079989324849463818175868172713496859968436633910633680216649416805806774541289433279788468718778634973256 5
PHIN =
71467427390759841729059757466289459181369050713019533645557376916391119997637068708795979336636964345506399166398464000000000000000000000000
G =
37260025342153877976366022431631274000323014010699999970111761875964418126632549841893154834592996350134421573562483983305651325914860373 3
ORDERG = 144070022526464542998162540305862391968000
PHIORDERG = 17966317053413597259085197820821504000000
R =
64411872697932469627298024328629501373966883091743836804603337113399879222101827567038858750269057628226431053470498775743682787912336229
```

```
ORDERR =
8815508518609347572770628774410485749927476308817571946515551877469252673303656529920000000000000
00000000

ALPHABYTES = 18
VBYTES = 59
S1BYTES = 19
S2BYTES = 58
SALTBYTES = 4

# ----- public information copied from *.rsp

mlen = 32
msg = 'D81C4D8D734FCBFBEADE3D3F8A039FAA2A2C9957E835AD55B22E75BF57BB556A'
pk =
'2020C105E4CE23ABB476713D7805654CF78802EF11CA4B6903B0407FE23897F33CD4B41A4A15AF68E7BCD6B486920E
5D6E42E5C3E86ECF6FF57F49'
sm =
'20019D011CE55E5F96EACC650084407061DC0520085CB9DACF314194F1F254D8EAF6D815D5D7B9D82FDD0D0AE1C63
F4B9C0FA19DE06D640FFF775FA8DAB052D8576CB53AB7DEF64C26E038B6C2D81C4D8D734FCBFBEADE3D3F8A039FAA
2A2C9957E835AD55B22E75BF57BB556A7C9935A0'

# ----- miscellaneous tests

assert PHIN == euler_phi(N)
assert ORDERG == Mod(G,N).multiplicative_order() assert ORDERR == Mod(R,PHIN).multiplicative_order()

msg = bytes.fromhex(msg)
pk = bytes.fromhex(pk)
sm = bytes.fromhex(sm)

def decode(b):
  b = bytearray(b)
  while b[:1] == b' ': b = b[1:]
  return sum(c<<(8*i) for i,c in enumerate(reversed(b)))

def encode(i,targetbytes):
  result = bytearray()
  while i > 0:
    result = bytearray([i%256])+result
    i >>= 8
  while len(result) < targetbytes:
    result = bytearray([32])+result
  assert len(result) == targetbytes
  return bytes(result)

assert decode(encode(31415,5)) == 31415

def open(sm,pk):
  s1,sm = sm[:S1BYTES],sm[S1BYTES:]
  s2,sm = sm[:S2BYTES],sm[S2BYTES:]
  m,salt = sm[:-SALTBYTES],sm[-SALTBYTES:]
```

```
  assert len(s1) == S1BYTES
  assert len(s2) == S2BYTES
  assert len(salt) == SALTBYTES
  h = hash(m+salt+m+salt)
  pk,s1,s2,h = map(decode,(pk,s1,s2,h))
  assert Mod(G,N)^(Mod(s1,PHIN)^s2) == Mod(pk,N)^(Mod(R,PHIN)^h)
  return m

subprocess.run('gcc -shared -o libkaz.so kaz_api.c sign.c rng.c sha256.c -fPIC -lcrypto -lgmp',shell=True)

libkaz = ctypes.CDLL(f'{os.getcwd()}/libkaz.so')
libkaz_open = libkaz.crypto_sign_open
libkaz_open.argtypes = c_char_p,POINTER(c_ulonglong),c_char_p,c_ulonglong,c_char_p
libkaz_open.restype = c_int

def reference_open(sm,pk):
  smlen = c_ulonglong(len(sm))
  m = create_string_buffer(len(sm))
  mlen = c_ulonglong(0)
  pk = create_string_buffer(pk)
  assert libkaz_open(m,byref(mlen),sm,smlen,pk) == 0
  return m.raw[:mlen.value]

assert open(sm,pk) == msg
assert reference_open(sm,pk) == msg
assert reference_open(sm,pk) == msg

phiphin = euler_phi(PHIN)
realRorder = Mod(R,ORDERG).multiplicative_order()

def forge(m,pk):
  salt = os.urandom(SALTBYTES)
  h = hash(m+salt+m+salt)
  pk = decode(pk)
  h = decode(h)
  r = random.randrange(2**256)
  while not Mod(r,ORDERG).is_unit(): r += 1
  s1 = ZZ(Mod(R,ORDERG)^r)
  loggV = Mod(pk,N).log(Mod(G,N))
  assert Mod(G,N)^loggV == Mod(pk,N)
  alpha = Mod(loggV,ORDERG).log(Mod(R,ORDERG))
  alpha += realRorder*random.randrange(2**256)
  assert Mod(R,ORDERG)^alpha == Mod(loggV,ORDERG)
  s2 = ZZ(Mod(alpha+h,ORDERG)/r)
  s2 += ORDERG*random.randrange(2**256)
  s2 %= phiphin
  assert Mod(G,N)^(Mod(s1,PHIN)^s2) == Mod(pk,N)^(Mod(R,PHIN)^h)
  return encode(s1,S1BYTES)+encode(s2,S2BYTES)+m+salt

newmsg = b'forged message'
while True:
  sm = forge(newmsg,pk)
```

```
 try:
   assert newmsg == open(sm,pk)
   assert newmsg == reference_open(sm,pk)
   break
 except:
   pass

assert newmsg == open(sm,pk)
assert newmsg == reference_open(sm,pk)
print(f'newmsg: {newmsg}')
print(f'sm: {sm.hex()}')
```