**Problem**

The code that performs decryption and padding check in RSA **PKCS#1** v1.5 decryption is data dependant. In particular, code in current (as of 0.8.0-alpha38) master and code in 0.7.5 branch has multiple ways in which it leaks information (for one, **it aborts as soon as the plaintext doesn't start with 0x00, 0x02**) about the decrypted ciphertext (both the bit length of the decrypted message as well as where the first unexpected byte lays).

* <https://en.wikipedia.org/wiki/PKCS>
  + <https://en.wikipedia.org/wiki/PKCS_1>

<https://medium.com/asecuritysite-when-bob-met-alice/whats-so-special-about-pkcs-1-v1-5-and-the-attack-that-just-won-t-go-away-51ccf35d65b7>

**0.7.6\_rsakey.py && 0.8.0-alpha39\_rsakey.py**

def encrypt(self, bytes): **(pas problématique)**

"""Encrypt the passed-in bytes.

This performs PKCS1 encryption of the passed-in data.

:type bytes: bytes-like object

:param bytes: The value which will be encrypted.

:rtype: bytearray

:returns: A PKCS1 encryption of the passed-in data.

"""

paddedBytes = self.\_addPKCS1Padding(bytes, 2)

m = bytesToNumber(paddedBytes) **(Same as \_raw\_public\_key\_op\_bytes function in fix)**

if m >= self.n:

raise ValueError()

c = self.\_rawPublicKeyOp(m)

encBytes = numberToByteArray(c, numBytes(self.n))

return encBytes

def decrypt(self, encBytes): **(le problème se manifeste dans cette fonction)**

"""Decrypt the passed-in bytes.

This requires the key to have a private component. It performs

PKCS1 decryption of the passed-in data.

:type encBytes: bytes-like object

:param encBytes: The value which will be decrypted.

:rtype: bytearray or None

:returns: A PKCS1 decryption of the passed-in data or None if

the data is not properly formatted.

"""

if not self.hasPrivateKey():

raise AssertionError()

if self.key\_type != "rsa":

raise ValueError("Decryption requires RSA key, \"{0}\" present"

.format(self.key\_type))

if len(encBytes) != numBytes(self.n): **(Same as \_raw\_private\_key\_op\_bytes function in fix)**

return None

c = bytesToNumber(encBytes)

if c >= self.n:

return None

m = self.\_rawPrivateKeyOp(c)

decBytes = numberToByteArray(m, numBytes(self.n))

**#Check first two bytes (\*\*\*Leaks information\*\*\*)**

**if decBytes[0] != 0 or decBytes[1] != 2:**

**return None**

**#Scan through for zero separator**

**for x in range(1, len(decBytes)-1):**

**if decBytes[x]== 0:**

**break**

**else:**

**return None**

**return decBytes[x+1:] #Return everything after the separator**

**Fix**

**0.7.6\_constanttime.py && 0.8.0-alpha39\_constanttime.py**

* Fonction utilisée dans l’autre fichier

**Text

Description automatically generated**

**0.7.6\_rsakey.py && 0.8.0-alpha39\_rsakey.py**

* La seule différence (qui n’est pas importante) entre les deux :
  + **0.8.0-alpha39\_rsakey.py contient cette condition de plus :** 
    - if self.key\_type != "rsa": raise ValueError("Decryption requires RSA key, \"{0}\" present".format(self.key\_type))

def encrypt(self, bytes): (pas important pour le CVE)

"""Encrypt the passed-in bytes.

This performs PKCS1 encryption of the passed-in data.

:type bytes: bytes-like object

:param bytes: The value which will be encrypted.

:rtype: bytearray

:returns: A PKCS1 encryption of the passed-in data.

"""

paddedBytes = self.\_addPKCS1Padding(bytes, 2)

**return self.\_raw\_public\_key\_op\_bytes(paddedBytes)**

def \_dec\_prf(self, key, label, out\_len): (utilisée dans la fonction decrypt)

"""PRF for deterministic implicit rejection in the RSA decryption.

:param bytes key: key to use for derivation

:param bytes label: name of the keystream generated

:param int out\_len: length of output, in bits

:rtype: bytes

:returns: a random bytestring

"""

out = bytearray()

if out\_len % 8 != 0:

raise ValueError("only multiples of 8 supported as output size")

iterator = 0

while len(out) < out\_len // 8:

out += secureHMAC(

key,

numberToByteArray(iterator, 2) + label +

numberToByteArray(out\_len, 2),

"sha256")

iterator += 1

return out[:out\_len//8]

def decrypt(self, encBytes):

"""Decrypt the passed-in bytes.

This requires the key to have a private component. It performs

PKCS#1 v1.5 decryption operation of the passed-in data.

Note: as a workaround against Bleichenbacher-like attacks, **it will**

**return a deterministically selected random message in case the padding**

**checks failed.** It returns an error (None) only in case the ciphertext

is of incorrect length or encodes an integer bigger than the modulus

of the key (i.e. it's publically invalid).

:type encBytes: bytes-like object

:param encBytes: The value which will be decrypted.

:rtype: bytearray or None

:returns: A PKCS#1 v1.5 decryption of the passed-in data or None if

the provided data is not properly formatted. Note: encrypting

an empty string is correct, so it may return an empty bytearray

for some ciphertexts.

"""

if not self.hasPrivateKey():

raise AssertionError()

if self.key\_type != "rsa":

raise ValueError("Decryption requires RSA key, \"{0}\" present"

.format(self.key\_type))

try:

dec\_bytes = self.\_raw\_private\_key\_op\_bytes(encBytes)

except ValueError:

# \_raw\_private\_key\_op\_bytes fails only when encBytes >= self.n,

# or when len(encBytes) != numBytes(self.n) and that's public

# information, so we don't have to handle it

# in sidechannel secure way

return None

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Aucune différence au niveau logique avant ce point \*\*\*\*\*\*\*\*\*\*\*\*\***

###################

# here be dragons #

###################

# While the code is written as-if it was side-channel secure, in

# practice, because of cPython implementation details IT IS NOT

# see: <https://securitypitfalls.wordpress.com/2018/08/03/constant-time-compare-in-python/>

**n = self.n**

# maximum length we can return is reduced by the mandatory prefix:

# (0x00 0x02), 8 bytes of padding, so this is the position of the

# null separator byte, as counted from the last position

**max\_sep\_offset = numBytes(n) - 10**

# the private exponent (d) doesn't change so `\_key\_hash` doesn't

# change, calculate it only once

**if not hasattr(self, '\_key\_hash') or not self.\_key\_hash:**

**self.\_key\_hash = secureHash(numberToByteArray(self.d, numBytes(n)),**

**"sha256")**

**kdk = secureHMAC(self.\_key\_hash, encBytes, "sha256")**

# we need 128 2-byte numbers, encoded as the number of bits

**length\_randoms = self.\_dec\_prf(kdk, b"length", 128 \* 2 \* 8)**

**message\_random = self.\_dec\_prf(kdk, b"message", numBytes(n) \* 8)**

# select the last length that's not too large to return

**synth\_length = 0**

**length\_rand\_iter = iter(length\_randoms)**

**length\_mask = (1 << numBits(max\_sep\_offset)) - 1**

**for high, low in zip(length\_rand\_iter, length\_rand\_iter):**

# interpret the two bytes from the PRF output as 16-bit big-endian

# integer

len\_candidate = (high << 8) + low

len\_candidate &= length\_mask

# equivalent to:

# if len\_candidate < max\_sep\_offset:

# synth\_length = len\_candidate

**mask = ct\_lt\_u32(len\_candidate, max\_sep\_offset)**

**mask = ct\_lsb\_prop\_u16(mask)**

**synth\_length = synth\_length & (0xffff ^ mask) \**

**| len\_candidate & mask**

**synth\_msg\_start = numBytes(n) - synth\_length**

**error\_detected = 0**

# enumerate over all decrypted bytes

**em\_bytes = enumerate(dec\_bytes)**

# first check if first two bytes specify PKCS#1 v1.5 encryption padding

**\_, val = next(em\_bytes)**

**error\_detected |= ct\_isnonzero\_u32(val)**

**\_, val = next(em\_bytes)**

**error\_detected |= ct\_neq\_u32(val, 0x02)**

# then look for for the null separator byte among the padding bytes

# but inspect all decrypted bytes, even if we already find the

# separator earlier

**msg\_start = 0**

**for pos, val in em\_bytes:**

# padding must be at least 8 bytes long, fail if any of the first

# 8 bytes of it are zero

# equivalent to:

# if pos < 10 and not val:

# error\_detected = 0x01

**error\_detected |= ct\_lt\_u32(pos, 10) & (1 ^ ct\_isnonzero\_u32(val))**

# update the msg\_start only once; when it's 0

# (pos+1) because we want to skip the null separator

# equivalent to:

# if pos >= 10 and not msg\_start and not val:

# msg\_start = pos+1

**mask = (1 ^ ct\_lt\_u32(pos, 10)) & (1 ^ ct\_isnonzero\_u32(val)) \**

**& (1 ^ ct\_isnonzero\_u32(msg\_start))**

**mask = ct\_lsb\_prop\_u16(mask)**

**msg\_start = msg\_start & (0xffff ^ mask) | (pos+1) & mask**

# if separator wasn't found, it's an error

# equivalent to:

# if not msg\_start:

# error\_detected = 0x01

**error\_detected |= 1 ^ ct\_isnonzero\_u32(msg\_start)**

# equivalent to:

# if error\_detected:

# ret\_msg\_start = synth\_msg\_start

# else:

# ret\_msg\_start = msg\_start

**mask = ct\_lsb\_prop\_u16(error\_detected)**

**ret\_msg\_start = msg\_start & (0xffff ^ mask) | synth\_msg\_start & mask**

# as at this point the length doesn't leak the information if the

# padding was correct or not, we don't have to worry about the

# length of the returned value (and thus the size of the buffer we

# pass to the caller); but we still need to read both buffers

# to ensure that the memory access patern is preserved (that both

# buffers are accessed, not just the one we return)

# equivalent to:

# if error\_detected:

# return message\_random[ret\_msg\_start:]

# else:

# return dec\_bytes[ret\_msg\_start:]

**mask = ct\_lsb\_prop\_u8(error\_detected)**

**not\_mask = 0xff ^ mask**

**ret = bytearray(**

**x & not\_mask | y & mask for x, y in**

**zip(dec\_bytes[ret\_msg\_start:], message\_random[ret\_msg\_start:]))**

**return ret**

def \_raw\_private\_key\_op\_bytes(self, message): (pas important pour le CVE)

n = self.n

if len(message) != numBytes(n):

raise ValueError("Message has incorrect length for the key size")

m\_int = bytesToNumber(message)

if m\_int >= n:

raise ValueError("Provided message value exceeds modulus")

dec\_int = self.\_rawPrivateKeyOp(m\_int)

return numberToByteArray(dec\_int, numBytes(n))

def \_raw\_public\_key\_op\_bytes(self, ciphertext): (pas important pour le CVE)

n = self.n

**if len(ciphertext) != numBytes(n):**

**raise ValueError("Message has incorrect length for the key size")**

c\_int = bytesToNumber(ciphertext)

if c\_int >= n:

raise ValueError("Provided message value exceeds modulus")

enc\_int = self.\_rawPublicKeyOp(c\_int)

return numberToByteArray(enc\_int, numBytes(n))