Exercise 6

Frage 1

Structure of PGP packets

Bruce Schneier organized Alice and Bob's PGP private keys for you and signed a message. For this and the following assignment, you need the files AliceMsg.txt.asc and bob-dsa-private-key.asc, which you can find under the assignment in the course outline.

Read the signature of the message AliceMsg.txt and enter the values *r and s* modulo 0xb0b in hex! Please enter the solutions without leading 0x and without separators (colons, spaces etc.).

Instead of pgpdump you can also use the tool Sequoia PGP, which is usable on the command line with "sq packet dump -x --mpis" and on the web at https://dump.sequoia-pgp.org/.

> **Antwort**
>
> r = 1206ff74fcfe76f159750c0e1ccc34a5d0c23f02
>
> s = a300c6e67d139b66d689e4cdc85835a6067770a4

```
niklas@NB-MSI:~$ pgpdump -i aliceMsg.txt
Old: Signature Packet(tag 2)(82 bytes)
        Ver 4 - new
        Sig type - Signature of a canonical text document(0x01).
        Pub alg - DSA Digital Signature Algorithm(pub 17)
        Hash alg - SHA1(hash 2)
        Hashed Sub: signature creation time(sub 2)(4 bytes)
                Time - Tue Jan 17 17:30:02 CET 2017
        Hashed Sub: signer's User ID(sub 28)(10 bytes)
                User ID - bob@rub.de
        Sub: issuer key ID(sub 16)(8 bytes)
                Key ID - 0xCFB2C05FD70EB4F8
        Hash left 2 bytes - 0c 0b
        DSA r(157 bits) - 12 06 ff 74 fc fe 76 f1 59 75 0c 0e 1c cc 34 a5 d0 c2 3f 02
        DSA s(160 bits) - a3 00 c6 e6 7d 13 9b 66 d6 89 e4 cd c8 58 35 a6 06 77 70 a4
                -> hash(DSA q bits)
```

```
niklas@NB-MSI:~$ python3
Python 3.9.2 (default, Feb 28 2021, 17:03:44)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> ' 12 06 ff 74 fc fe 76 f1 59 75 0c 0e 1c cc 34 a5 d0 c2 3f 02'.replace(" ", "")
'1206ff74fcfe76f159750c0e1ccc34a5d0c23f02'
>>> 'a3 00 c6 e6 7d 13 9b 66 d6 89 e4 cd c8 58 35 a6 06 77 70 a4'.replace(" ", "")
'a300c6e67d139b66d689e4cdc85835a6067770a4'
```

Frage 2

Change the private key *x of* Bob to the value **0xb0b0dead** so that a valid OpenPGP packet is created (which can be read by pgpdump, for example)! The modified field should have minimum length. Also adjust the length specification and the checksum of the secret key and specify the value of the checksum in hex (without leading 0x).

Notes:

· RFC 4880

· You can use a hex editor to edit the file.

· You can use the command line to convert the Base64 encoding from OpenPGP to binary (and vice versa), e.g. with GnuPG "gpg --enarmor" or "gpg --dearmor" or with Sequoia PGP "sq enarmor" or "sq dearmor".

> **Antwort**:
>
> Checksum =

```
niklas@NB-MSI:~$ pgpdump -i bob_dsa_private_key.asc
Old: Secret Key Packet(tag 5)(443 bytes)
        Ver 4 - new
        Public key creation time - Tue Jan 17 16:40:43 CET 2017
        Pub alg - DSA Digital Signature Algorithm(pub 17)
        DSA p(1024 bits) - b8 05 37 5d 6e 62 92 84 27 54 83 ef f6 1a 39 e9 13 47 37 a0 7f d2 68 5b f5 16 6d e3 66 3f 0c f3 f5 3b e8 60 2e e
        DSA q(160 bits) - ab c0 b5 a1 ef 74 21 ca 07 71 d1 bd 84 b1 91 ad 97 07 fd ff
        DSA g(1024 bits) - a7 f7 62 32 d0 78 58 54 bc 01 4f c7 a9 5a 7d c3 f2 77 16 bc 20 f1 ba 3c 6e 34 01 7a 94 dc bf b4 dd e5 53 db 6c f
        DSA y(1019 bits) - 05 29 d2 91 7e 05 45 c1 d4 ee 93 c2 5e e8 c5 d9 5c e4 b3 7d 3b 99 b6 ea 1c 01 eb 02 fe b1 58 58 ea 34 95 ac ba 2
        DSA x(160 bits) - 8d 5d 6a 83 d8 40 ed 59 16 23 76 8b b8 9e 60 60 25 29 65 1a
        Checksum - 08 f2
Old: User ID Packet(tag 13)(18 bytes)
        User ID - Bobby <bob@rub.de>
Old: Signature Packet(tag 2)(97 bytes)
        Ver 4 - new
        Sig type - Positive certification of a User ID and Public Key packet(0x13).
        Pub alg - DSA Digital Signature Algorithm(pub 17)
        Hash alg - SHA1(hash 2)
        Hashed Sub: signature creation time(sub 2)(4 bytes)
                Time - Tue Jan 17 16:40:43 CET 2017
        Hashed Sub: key flags(sub 27)(1 bytes)
                Flag - This key may be used to certify other keys
                Flag - This key may be used to sign data
        Hashed Sub: preferred symmetric algorithms(sub 11)(4 bytes)
                Sym alg - AES with 256-bit key(sym 9)
                Sym alg - AES with 192-bit key(sym 8)
                Sym alg - AES with 128-bit key(sym 7)
                Sym alg - Triple-DES(sym 2)
        Hashed Sub: preferred hash algorithms(sub 21)(5 bytes)
                Hash alg - SHA256(hash 8)
                Hash alg - SHA384(hash 9)
                Hash alg - SHA512(hash 10)
                Hash alg - SHA224(hash 11)
                Hash alg - SHA1(hash 2)
        Hashed Sub: preferred compression algorithms(sub 22)(3 bytes)
                Comp alg - ZLIB <RFC1950>(comp 2)
                Comp alg - BZip2(comp 3)
                Comp alg - ZIP <RFC1951>(comp 1)
        Hashed Sub: features(sub 30)(1 bytes)
                Flag - Modification detection (packets 18 and 19)
        Hashed Sub: key server preferences(sub 23)(1 bytes)
                Flag - No-modify
        Sub: issuer key ID(sub 16)(8 bytes)
                Key ID - 0xCFB2C05FD70EB4F8
        Hash left 2 bytes - 4f ea
        DSA r(160 bits) - 94 8b 30 f8 34 ed 57 ee 2b 3a 0f af cb 94 1f 47 c0 d0 b1 bb
        DSA s(159 bits) - 4c 64 e8 2e d1 87 f1 a0 d6 f4 f0 73 16 ba 4b 30 49 fd 1e 1b
                -> hash(DSA q bits)
Old: Secret Subkey Packet(tag 7)(305 bytes)
        Ver 4 - new
        Public key creation time - Tue Jan 17 16:40:43 CET 2017
        Pub alg - ElGamal Encrypt-Only(pub 16)
        ElGamal p(1024 bits) - bb b2 ca 6e ae 34 fa d8 df 32 78 0c 01 03 e3 6d 47 a4 fd 26 0a d8 63 e0 85 29 bf 17 85 3b c2 e2 94 44 eb 8e

        ElGamal g(3 bits) - 05
        ElGamal y(1022 bits) - 20 59 34 1d 16 91 c2 c0 b4 b6 74 c3 00 ce b2 89 8e 84 da 7c 8b 12 00 57 1f 48 08 0c 54 ca d4 6a de 03 bf 71

        ElGamal x(246 bits) - 2e 4e 30 5d 6c a0 6f 06 12 73 90 df eb e0 69 2b 3d 23 e5 f1 9f 31 c8 9e 63 a3 3b e5 40 e1 38
        Checksum - 0f be
```

```
Old: Signature Packet(tag 2)(73 bytes)
        Ver 4 - new
        Sig type - Subkey Binding Signature(0x18).
        Pub alg - DSA Digital Signature Algorithm(pub 17)
        Hash alg - SHA1(hash 2)
        Hashed Sub: signature creation time(sub 2)(4 bytes)
                Time - Tue Jan 17 16:40:43 CET 2017
        Hashed Sub: key flags(sub 27)(1 bytes)
                Flag - This key may be used to encrypt communications
                Flag - This key may be used to encrypt storage
        Sub: issuer key ID(sub 16)(8 bytes)
                Key ID - 0xCFB2C05FD70EB4F8
        Hash left 2 bytes - dc 26
        DSA r(157 bits) - 1b c6 3b 2f 96 b7 c8 e2 41 6b 58 96 18 67 43 6b 8c f0 87 b7
        DSA s(159 bits) - 7a 57 bb 94 e4 bf 28 e9 18 3e 3c 5a 1a e4 dc 38 9a 5b d2 c6
                -> hash(DSA q bits)
```

Check current length:

```
>>> '8d 5d 6a 83 d8 40 ed 59 16 23 76 8b b8 9e 60 60  25 29 65 1a'.replace(" ", "")
'8d5d6a83d840ed591623768bb89e60602529651a'
>>> '8d 5d 6a 83 d8 40 ed 59 16 23 76 8b b8 9e 60 60  25 29 65 1a'.replace(" ", "")
KeyboardInterrupt
>>> len('8d5d6a83d840ed591623768bb89e60602529651a')
40
>>> hex(40)
'0x28'
>>> 40*4
160
```

```
000000125                  05 29 d2  91 7e 05 45 c1 d4 ee 95   dsa_public_y
00000130  c2 5e e8 c5 d9 5c e4 b3  7d 3b 99 b6 ea 1c 01 eb
00000140  02 fe b1 58 58 ea 34 95  ac ba 27 9e 4b 1a 01 19
00000150  d1 d0 4f 0b da 84 09 ba  cc 6e d7 00 02 8a 59 1e
00000160  40 f8 17 ca c1 30 ce c8  c0 c2 53 dc 2b 52 9e 9b
00000170  b2 91 59 44 a7 2f 99 55  16 a7 db 80 e5 99 18 fa
00000180  54 7a 06 1f 73 a9 66 d0  3d 42 cc 06 4d db aa e5
00000190  45 0f 2a eb 70 65 16 1f  1a 3c 8c 70 bb dc 90 cd
000001a0  a8 c7 c1 28 ed
000001a5                  00                               s2k_usage
000001a6                     00 a0                         dsa_secret_len
000001a8                        8d 5d 6a 83 d8 40 ed 59    dsa_secret
000001b0  16 23 76 8b b8 9e 60 60  25 29 65 1a
000001bc                                    08 f2          checksum

User ID Packet, old CTB, 2 header bytes + 18 bytes
    Value: Bobby <bob@rub.de>

    00000000  b4                                            CTB
    00000001     12                                          length
    00000002        42 6f 62 62 79 20  3c 62 6f 62 40 72 75 62   value
    00000010  2e 64 65 3e

Signature Packet, old CTB, 2 header bytes + 97 bytes
    Version: 4
```

Get value for new x: (value **0xb0b0dead)**

```
# Size of new x:
>>> len('b0b0dead')
8
>>> hex(8*4)
'0x20'

# Checksum
"""
From https://www.rfc-editor.org/rfc/rfc4880#section-5.5.3
```

```
    The two-octet checksum that follows the algorithm-specific portion is
    the algebraic sum, mod 65536, of the plaintext of all the algorithm-
    specific octets (including MPI prefix and data).  With V3 keys, the
    checksum is stored in the clear.  With V4 keys, the checksum is
    encrypted like the algorithm-specific data.  This value is used to
    check that the passphrase was correct.  However, this checksum is
    deprecated; an implementation SHOULD NOT use it, but should rather
    use the SHA-1 hash denoted with a usage octet of 254.  The reason for
    this is that there are some attacks that involve undetectably
    modifying the secret key.
"""

old_checksum = 0x08f2

sum_of_algo =


new_checksum = (sum_of_algo % 65536)
```

Frage 3

RSA Fault Attack and Signature

A smart card manufacturer uses the RSA signature method on its smart cards. The signature is calculated using the Chinese Remainder Theorem (CRT).

Why are values such as *pInv* or *p*, *q* also stored on the chip card, although these are not absolutely necessary for the calculation?

Wählen Sie die richtige(n) Antwort(en) aus:

- [x] ~~Because these values can be used for other, faster algorithms to calculate the same result more efficiently.~~
- [x] ~~These values make it possible to develop algorithms that can check themselves. (unsicher??)~~
- [ ] These values do not match the others and have been introduced only to make the system more complex and thus make attacks more difficult.
- [ ] These values are included for backward compatibility, as they were necessary for old algorithms.

Frage 4

Given an RSA public key (n,e) with modulus n and public exponent *e* , where *e*=7 and *n*=*pq*=585209. Assume that you were able to obtain an erroneous signature *s'*=357672 for message m = "NetSeC" (the quotes are not part of the message) by manipulating the map. Determine *p* and *q* using *m* and *s'*.

Specify for this:

*m*mod*n* = `31778`

```
# perform modulo operation
m = number % 585209 # = 31778
print(m)
```

*s'^emod**n*** = `61769`

```
# calculate s'^e mod n
s = 357672
e = 7
print(pow(s, e) % n) = 61769
```

p=ggt(m−s'e,n) = `769`

```
# ---------------------------------------------------------
# calculate s'^e mod n

s = 357672
e = 7

inv_s = pow(s, e) % n
print(inv_s) # = 61769


# ---------------------------------------------------------
# calculate ggt(m−s'^e,n)

def getGgt(x,y):
  if(y == 0):
    return abs(x)
  else:
    return getGgt(y, x % y)


p = getGgt(m - inv_s, n)  # = 769
print(p)
```

q=n/p = `761`

```
# ---------------------------------------------------------
# calculate q = n/p

q = n/p   # = 761
print(q)
print()
```

💡 ***Note 1**: Interpret the binary representation of ASCII encoded words as the binary representation of an integer (left most significant bit). We encode ASCII characters in 7 bits. Example: XY -> 10110001011001 -> 0b10110001011001 = 11353*

***Note 2**: It is possible that the message is larger than the modulus, which is normally not allowed. However, this does not affect the solution of the task here.*

Frage 5

This task related to the previous task!

> **Now that you know *p* and *q*, calculate:**
>
> - the secret exponent d_p= d mod p-1 and d_q= d mod q-1, used for signature generation with the CRT
>   - *d_p* = Antwort
>   - *d_q* = Antwort
> - the values *a* and *b* in the formula 1=*ap*+*bq*.
>   - *a* =Antwort
>   - *b* = Antwort
> - a valid signature σ for *m*.
>   - σ = Antwort