Der OpenPGP-Schlüssel 9F7A3119DF9110EB

Annie Yousar*

2020-08-25

Zusammenfassung

In diesem Beitrag wird ein OpenPGP-Schlüssel[1] analysiert und beschrieben, wie er wahrscheinlich am genkey-Kommando vorbei erzeugt wurde. Natürlich haben wir keinen Zugriff auf den privaten Schlüssel, aber wir zeigen hier, welche Strukturen und welche Kommandos dabei zum Ziel führen. Die Grundlage für unsere Untersuchung bildet die aktuelle Version [1] des Nachfolgers vom RFC 4880 [3] mit seiner bereits erfolgten ECC-Ergänzung, dem RFC 6637 [7]. Unsere GnuPG-Version ist 2.2.4, die von OpenSSL ist 1.1.1f.

Auch wenn der Beitrag etwas lang erscheint, ist er doch schnell zu lesen, weil sich der Vollständigkeit halber die Ein- und Ausgabedaten mehrfach wiederholen.

Inhaltsverzeichnis

* mailto: *a.yousar@informatik.hu-berlin.de*

[1] `https://keys.openpgp.org/vks/v1/by-fingerprint/222E67EA2AA253A967408ABE9F7A3119DF9110EB`

1 PGP Nachrichten und Pakete

Alle OpenPGP-Nachrichten bestehen aus verschiedenen TLV-Strukturen[2], die in PGP traditionell *Pakete* heißen. Sie können aneinander gereiht und verschachtelt sein. Das erste Byte eines Pakets ist immer das *Tag*[3], das den Typ des Inhalts bestimmt. Ihm folgen dann die Längenbytes und der eigentliche Paket-Inhalt. Während der normative RFC 4880 [3] mit den Definitionen und Beschreibungen beginnt, gehen wir hier einen mehr ingenieurmäßigen Weg. Wir geben die Struktur an und lernen dabei ihre Bestandteile kennen.

Wir werden im Folgenden den einfach strukturierten Schlüssel

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

xlIEXgvhABMIKoZIzj0DAQcCAwTQ8aTR8t1myluI1AP2rsxyOkCJHgXi+R/EG+G/
mCOpbUT5Z3ohns8ki38HQoLrioDb+IK5uZenyCJu5KavT6MLzTVFcm5zdCBHIEdp
ZXNzbWFubiA8Z2llc3NtYW5uQGluZm9ybWF0aWsuaHUtYmVybGluLmRlPsLADgQT
EwgAdhYhBCIuZ+oqolOpZOCKvp96MRnfkRDrBQJeC+EBAhsDBQkUt8J/BQsJCAcC
BRUKCQgLBBYCAwE4GmhOdHA6Ly93d3cuaW5mb3JtYXRpay5odS1iZXJsaW4uZGUv
fmdpZXNzbWFuL2dwZy1jcC50eHQCHgECF4AACgkQn3oxGd+REOu9EAEA8At3K5XP
qHenbyU5jDXrjs+6L9/Mgd6bwjDCyyC4nBoBAPc6pgo5mh2pdIMv8RhUrbfAwztO
axBBFCEQzSgugd43zsDFBF4TS3ABC8DGGkyFsrb7/4DsvGDWBYqzXRnlwG6kHuX8
//ErnstG+Giessmann//db5W1inL87dyI4QMtOix7yOpBmcHx+KytxQFL7Oas7z9
MRhpZKTZnDtOjHtv9a2Gv/Z+tzTO/aY4poTuo2VHQqjXLd8o+ExDgS3oRkAnOVpw
fH4rIxGGlkpNmfPOO3SMe2/1rYa/9n63NPT9pjimhO6jZUdCqNct3yj4TEOBLehG
QCc5WkNyJM7XdOpacI9jUgTeyj9xJaRjM7/23GKWvYPD6spBmO5voBxHGIQJbr/7
QMxIWoQkfT6gRpEFI6IOMt59MHksbA+ZDVJG8xRtAT//oeZ9GkJk+wl2Xf+5vrne
7Se4+SKNEt/arXNHZlD5L8M4zM7d+Nihd5HOUEhquqsUKa2O0uPWJvaRLrfue4sf
S8wxZhF22QOSOvfOEnshn4H2I5qh1pV57htk2poPBvvsuBmIMdfZFZ3dWrTE8TmL
BUB2hci+CCzO8FGbVzu5ABdAAAHCeAQYEwgAIBYhBCIuZ+oqolOpZOCKvp96MRnf
kRDrBQJeEOtxAhsMAAoJEJ96MRnfkRDrXKMA/jnTE7vNWuiTsYy+TvCRdDfrjN5B
xamgfZk1jqTkvBRwAQDArSh5+vzZ7kOCqLkBcx8pMPoc2M9lBSvLZWyReLqUOg==
=++++
-----END PGP PUBLIC KEY BLOCK-----
```

analysieren und seine Struktur bis ins letzte Byte aufklären. Wir gehen dabei auch auf die Buchstabenfolge /EG+G/ in der ersten Zeile ein. Der aufmerksamen Leserin ist wahrscheinlich auch der Name am Zeilenanfang in der Mitte des Schlüssels nicht entgangen. Wir kennen diese Folge ja bereits aus dem Schlüssel des vergangenen Jahrzehnts.

Die Pakete dieses Schlüssels kann man sich mit dem Kommando[4]

```
sed '/^[-=]/d' | base64 -d | gpg --list-packets | sed '/^\t/d'
```

schon vorab ansehen[5]:

[2] TLV bedeutet *Tag-Length-Value*.

[3] *tag* (engl.) Anhängsel oder Etikett

[4] Die verwendeten Kommandos werden wir nicht erklären. Sie sind getestet, aber nicht immer die kürzeste Lösung. Andere Vorschläge mit den üblichen Kommandos der bash sind selbstverständlich gern gesehen. Die Eingabe ist meist `stdin` und ergibt sich igentlich immer aus dem Zusammenhang. Die Ausgaben werden zur Unterscheidung von den Kommandos selbst etwas eingerückt dargestellt.

[5] Normalerweise kann man die Daten auch gleich aus diesem Dokument kopieren. Nur bei den Schlüsseldaten geht das nicht so direkt, weil die notwendige Leerzeile vor den Schlüsseldaten im PDF-Dokument nicht vorhanden ist. Deshalb ist hier ein Umweg über die Binärdaten erforderlich.

```
# off=0 ctb=c6 tag=6 hlen=2 plen=82 new-ctb
:public key packet:
# off=84 ctb=cd tag=13 hlen=2 plen=53 new-ctb
:user ID packet: "Ernst G Giessmann <giessmann@informatik.hu-berlin.de>"
# off=139 ctb=c2 tag=2 hlen=3 plen=206 new-ctb
:signature packet: algo 19, keyid 9F7A3119DF9110EB
# off=348 ctb=ce tag=14 hlen=3 plen=389 new-ctb
:public sub key packet:
# off=740 ctb=c2 tag=2 hlen=2 plen=120 new-ctb
:signature packet: algo 19, keyid 9F7A3119DF9110EB
```

Ohne den letzten sed-Filter ist die Ausgabe natürlich ausführlicher und mit der GPG-Option `-v` werden auch noch einige Datenfelder explizit angegeben.

Wir werden nun diese fünf Pakete untersuchen und dazu dann auch die Struktur des zugehörigen privaten Schlüssel beschreiben, ohne ihn tatsächlich zu kennen.

2 Public Key Packet

Wir betrachten dazu die Binärdaten des Schlüssels und suchen das erste Paket[6].

```
c652045e0be10013082a8648ce3d030107020304d0f1a4d1f2dd66ca5b88d403f6aecc723a4089
1e05e2f91fc41be1bf9823a96d44f9677a219ecf248b7f074282eb8a80dbf882b9b997a7c8226e
e4a6af4fa30bcd3545726e737420472047696573736d616e6e203c67696573736d616e6e40696e
666f726d6174696b2e68752d6265726c696e2e64653ec2c00e041313080076162104222e67ea2a
a253a967408abe9f7a3119df9110eb05025e0be101021b03050914b7c27f050b0908070205150a
09080b0416020301381a687474703a2f2f7777772e696e666f726d6174696b2e68752d6265726c
696e2e64652f7e67696573736d616e2f6770672d63702e747874021e01021780000a09109f7a31
19df9110ebbd100100f00b772b95cfa877a76f25398c35eb8ecfba2fdfcc81de9bc230c2cb20b8
9c1a0100f73aa60a399a1da974832ff11854adb7c0c33b4e6b1041142110cd282e81de37cec0c5
045e134b70010bc0c61a4c85b2b6fbff80ecbc60d6058ab35d19e5c06ea41ee5fcfff12b9ecb46
f8689eb2c99a9e7fff75be56d629cbf3b77223840cb4e8b1ef2d29066707c7e2b2b714052fb39a
b3bcfd31186964a4d99c3b748c7b6ff5ad86bff67eb734f4fda638a684eea3654742a8d72ddf28
f84c43812de8464027395a707c7e2b231186964a4d99f3f43b748c7b6ff5ad86bff67eb734f4fd
a638a684eea3654742a8d72ddf28f84c43812de8464027395a437224ced7774a5a708f635204de
ca3f7125a46333bff6dc6296bd83c3eaca419b4e6fa01c471884096ebffb40cc485a84247d3ea0
46910523a20e32de7d30792c6c0f990d5246f3146d013fffa1e67d1a4264fb09765dffb9beb9de
ed27b8f9228d12dfdaad73476650f92fc338ccceddf8d8a17791ce50486abaab1429ad8ed2e3d6
26f6912eb7ee7b8b1f4bcc31661176d90d123af7ce127b219f81f6239aa1d69579ee1b64da9a0f
06fbecb8198831d7d9159ddd5ab4c4f1398b05407685c8be082ccef0519b573bb90017400001c2
7804181308002016210422 2e67ea2aa253a967408abe9f7a3119df9110eb05025e134b71021b0c
000a09109f7a3119df9110eb5ca300fe39d313bbcd5ae893b18cbe4ef0917437eb8cde41c5a9a0
7d99358ea4e4bc14700100c0ad2879fafcd9ee4d02a8b901731f2930fa1cd8cf65052bcb656c91
78ba943a
```

Das erste Paket beginnt mit `c6 52` und endet, wie wir gleich sehen werden, nach weiteren 82 Bytes mit `a3 0b`, nach vier Bytes in der dritten Zeile. Wir strukturieren diese Daten nun etwas und

[6] Das entsprechende Kommando ist `sed '/^[-=]/d' | base64 -d > key.bin; xxd -p -c39 key.bin`. Wir merken uns für später auch gleich die Binärdaten als Datei *key.bin*.

verwenden dabei eine solche Darstellung mit kurzen Kommentaren, aus der man leicht mit der Funktion

```
function rehex {
  sed '/^[-=]/d;s/ //g;s/:.*//' | xxd -p -r
}
```

und dem Kommando `rehex > first.packet` wieder die gleichen Binärdaten wie mit dem kommentarlosen `tail -c+1 key.bin | head -c84 > first.packet` erzeugen kann.

```
---------------------------
c6 52    : public key packet
---------------------------
            : packet tag 1100 0110
            : type "2#000110" aka type 6 (Public-Key packet [4.3])
            : packet length 0x52 : one octet : len=82 < 192
 04         : version number
 5e0be100   : key created 2020-01-01 00:00:00Z (1577833200)
 13         : public key algorithm ECDSA (ID 19 [9.1])
  08 2a8648ce3d030107 : curve OID : nistp256
  0203      : public EC point als MPI (multi precision integer)
  04        : public point format byte
   d0f1a4d1f2dd66ca5b88d403f6aecc723a40891e05e2f91fc41be1bf9823a96d
   44f9677a219ecf248b7f074282eb8a80dbf882b9b997a7c8226ee4a6af4fa30b
---------------------------
```

Wenden wir uns zunächst *Tag* und *Länge* des Pakets zu. Für OpenPGP-Pakete gibt es das neue[7] und das alte Längenformat. Aus Kompatibilitätsgründen werden Schlüsseldaten in der Regel im alten Längenformat exportiert. Der vorliegende öffentliche Schlüssel hat das neue Format. Wenn man ihn also importiert und danach wieder exportiert, wird sich seine ASCII-Darstellung ab der vierten Zeile völlig ändern, obwohl nur wenige Tag- und Längenangaben anders sind. Wir kommen darauf später noch einmal zurück.

Die ersten beiden (linken) Bits des Tag-Bytes `0xc6=2#11000110` sind beim neuen Format immer gesetzt, die restlichen bilden die Typ-Bits des Pakets. Hier haben wir also ein Paket vom Typ 6 (Public-Key Packet [1, 4.3]).

Wenn es kleiner als 192 ist, gibt das folgende Längenbyte auch gleich die Länge des folgenden Paket-Inhalts an. Ist es größer als 191, folgen noch weitere Längenbytes. Wir beschäftigen uns mit diesem Fall später.

Es sind also `0x52=82` Bytes in diesem Paket und das Kommando `xxd -p -s0 -l84 key.bin` liefert dann auch die einfache Hex-Darstellung des Pakets.

Zur Kontrolle eines mit Kommentaren formatierten Pakets zählen wir sicherheitshalber immer noch einmal nach:

```
dc -e "$( rehex | wc -c ) 16op"
  54
```

[7] Neu ist relativ. Schon im ersten RFC 2440 zu OpenPGP von 1998 war das so definiert.

Das erste Inhalts-Byte eines Typ-6-Pakets ist immer das Versionsbyte. Alle Datenpakete im Schlüssel sind Version 4, daher können wir uns darauf beschränken. Nach dem Versionsbyte folgen dann der Zeitpunkt der Schlüsselerzeugung (vier Byte), der Schlüsseltyp (ein Byte) und schließlich die dem Schlüsseltyp entsprechenden Schlüsseldaten. In der Version 5 wird in diese Darstellung nur noch eine Längeninformation (vier Byte) vor den eigentlichen Schlüsseldaten eingefügt.

```
---------------------------
 04          : version number
 5e0be100    : key created 2020-01-01 00:00:00Z (1577833200)
 13          : public key algorithm ECDSA (ID 19 [9.1])
 : 0000004C : add the length of key data in v5
  08 2a8648ce3d030107   : curve OID : nistp256
  0203       : public EC point als MPI (multi precision integer)
  04         : public point format byte
   d0f1a4d1f2dd66ca5b88d403f6aecc723a40891e05e2f91fc41be1bf9823a96d
   44f9677a219ecf248b7f074282eb8a80dbf882b9b997a7c8226ee4a6af4fa30b
---------------------------
```

Wir haben hier also als Zeitpunkt der Schlüsselerzeugung `0x5e0be100` und den Schlüsseltyp `0x13` (19) für EC-DSA.

Für die Konvertierung der hexadezimalen Zeitangabe der Epoch-Zeit (die Anzahl der Sekunden seit dem 1970-01-01 00:00 UTC) in das durch ISO-8601 standardisierte Format und wieder zurück verwenden wir die Kommandos:

```
dc -e 16o$( date "+%s" --date="2020-01-01Z 00:00" )p
  5E0BE100
date -u '+%F %T %Z' --date "@$(( 16#5E0BE100 ))"
  2020-01-01 00:00:00 UTC
```

Zu einem öffentlichen ECDSA-Schlüssel (mit der ID 19 [1, 9.1]) gehören die entsprechende elliptische Kurve und die Koordinaten des öffentlichen Punktes. Die Kurve wird dabei als OID (*Object Identifier*) und der Punkt ohne Kompression angegeben.

Die ASN.1-Darstellung des OIDs einer Kurve findet man im Draft 4880bis [1, 9.2], oder man kann, wenn man den OID-Namen kennt, auch OpenSSL benutzen:

```
openssl asn1parse -genstr OID:prime256v1 -out xx -noout; xxd -p -s1 xx
  082a8648ce3d030107
```

Bei der Umrechnung der Byte-Folge `082a8648ce3d030107` in einen Bezeichner muss man nur noch das ASN.1-Typbyte `06` für einen Object Identifier voranstellen:

```
<<< "06 082a8648ce3d030107" xxd -p -r | openssl asn1parse -inform der
  0:d=0 hl=2 l= 8 prim: OBJECT :prime256v1
```

Der öffentliche Punkt wird in ASN.1 (vgl. [2, 2.3.5]) als Bytefolge kodiert, die aus dem Formatierungsbyte (`0x04`) und der x- und y-Koordinate mit jeweils führenden Nullen besteht. In OpenPGP wird genau diese Folge im Schlüssel-Paket als nur eine ganze Zahl (MPI[8]) dargestellt. Das bedeutet, dass zuerst eine Bitlänge durch zwei Bytes angegeben wird und dass dann die Bytes der Zahl

[8] *multi precision integer*

in big-endian Darstellung (*most significant byte first*) folgen. Bei einer 256-Bit-Kurve ist der öffentliche Schlüssel folglich immer eine fiktive Zahl mit 515 Bits (`0x0203`), weil ja vom „obersten“ Format-Byte `0x04=2#00000100` nur drei Bits in die Zählung eingehen.

Damit ist der öffentliche Schlüssel beschrieben. Wir berechnen jetzt noch mit dem Kommando

```
rehex | base64 -w64
```

die entsprechende „ASCII-Rüstung“ (ASCII armor), also die Kodierung in base64:

```
xlIEXgvhABMIKoZIzj0DAQcCAwTQ8aTR8t1myluI1AP2rsxy0kCJHgXi+R/EG+G/
mC0pbUT5Z3ohns8ki38HQoLrioDb+IK5uZenyCJu5KavT6ML
```

Interessant ist die Zeichenfolge `/EG+G/` am Ende der ersten Zeile. Will man eine bestimmte Folge im öffentlichen Schlüssel haben, dann muss man so lange Schlüsselpaare erzeugen, bis sich „zufällig“ die gesuchte Folge ergibt. Eine direkte Berechnung des zugehörigen privaten Schlüssels geht natürlich nicht.

Für einen Schlüssel, bei dem die x-Koordinate des öffentlichen Punkt zwei führende Null-Bytes hat, würde man beispielsweise

```
t=$( openssl ecparam -genkey -name prime256v1 -noout -outform der \
   | tee x.der | xxd -p -s57 -l2 -u x.der )
```

so lange ausführen, bis `[[ "$t" == "0000" ]]` ist[9]. Oder gibt es eínen schnelleren Trick?

Bei der Darstellung des Pakets im alten Format ändert sich nur das erste Tag-Byte `c5` zu `98`. Beim Tag-Byte ist das erste Bit immer gesetzt, das zweite dagegen nur im neuen Format. Die unteren beiden Bit des Tags im alten Format geben an, wieviele Längenbytes folgen (`00` – ein Längenbyte, `01` – zwei und `10` – vier Längenbytes), bei `11` ist die Länge des Paketinhalts unbestimmt. Nicht nur wegen der damit verbundenen Beschränkung auf 16 Paket-Typen, sondern auch wegen der Möglichkeit weiterer bestimmter Längenangaben sollte man das neue Längenformat verwenden.

Aus den verbleibenden mittleren Bits `0110` im Tag `0x98=2#10011000` ergibt sich nun wie vorher der Typ 6 (Public Key Packet).

Durch dieses Schlüsselpaket wird auch der so genannte Fingerprint und die Key ID bestimmt, deren Berechnung sich im Laufe der Zeit auch immer wieder geändert hat. Bei einem Schlüssel der älteren Version 3 bestand beispielsweise die Schlüssel-ID aus den unteren 64 Bit des RSA-Moduls und der Fingerprint wurde mit dem Hash-Algorithmus MD5 aus Modul und Exponent berechnet. Das ist natürlich nicht kollisionsresistent und kann sogar bewusst erzeugt werden.

Der Fingerprint in der Version 4 wird mit SHA-1 aus den folgenden Daten berechnet. Es beginnt mit dem Byte 99, darauf folgen zwei Längenbytes, die die Länge des gesamten Inhalts des Schlüsselpakets angeben und dann die Daten des Schlüsselpakets selbst. In unserem Fall also der Hash-Wert der folgenden Daten:

```
----------------------------
99 0052    : data to be hashed for fingerprint and Key ID
----------------------------
 04            : version number
 5e0be100      : key created 2020-01-01 00:00:00Z (1577833200)
```

[9] Mit `openssl ec -text -inform der -in x.der -noout` kann man sich das Ergebnis dann ansehen.

```
 13           : public key algorithm ECDSA (ID 19 [9.1])
  08 2a8648ce3d030107 : curve OID : nistp256
  0203         : public EC point als MPI (multi precision integer)
  04           : public point format byte
   d0f1a4d1f2dd66ca5b88d403f6aecc723a40891e05e2f91fc41be1bf9823a96d
   44f9677a219ecf248b7f074282eb8a80dbf882b9b997a7c8226ee4a6af4fa30b
----------------------------
```

Schaut man sich das Tag-Byte `0x99=2#10011001` an, dann sieht man an den ersten beiden Bits `10`, dass es sich bei den Längenbytes um das „alte“ Format handelt und an den letzten beiden Bits `01`, dass für die Längeninformation zwei Bytes reserviert sind. Die verbleibenden Bits `2#0110=0x06` ergeben den Typ *Public Key Packet*. Es handelt sich also hier um ein reguläres OpenPGP-Paket, dessen Hash-Wert als Fingerprint verwendet wird, jedoch nicht um das Paket aus dem Schlüssel.

Mit dem Kommando

```
rehex | openssl sha1 -binary | xxd -p -c32 | sed 'p;s/.\{24\}//'
```

erhält man diesen Hash-Wert

```
  222e67ea2aa253a967408abe9f7a3119df9110eb
  9f7a3119df9110eb
```

Die unteren 64 Bit bilden dabei die Schlüssel-ID.

Der Vollständigkeit halber geben wir hier auch noch die Berechnung der Schlüssel-ID in der neuen Version 5 an. Statt des unsicheren SHA-1 wird hier SHA-256 verwendet. Die Header-Bytes der Eingabedaten für die Hash-Funktion ändern sich auch noch einmal. Auf das Tag `0x9a` folgen jetzt vier Längenbytes:

```
----------------------------
9a 00000056 : data to be hashed
----------------------------
 05           : version number
 5e0be100     : key created 2020-01-01 00:00:00Z (1577833200)
 13           : public key algorithm ECDSA (ID 19 [9.1])
 0000004c     : 4 bytes of the length of all the key data : added in v5
  08 2a8648ce3d030107 : curve OID : nistp256
  0203         : public EC point als MPI (multi precision integer)
  04           : public point format byte
   d0f1a4d1f2dd66ca5b88d403f6aecc723a40891e05e2f91fc41be1bf9823a96d
   44f9677a219ecf248b7f074282eb8a80dbf882b9b997a7c8226ee4a6af4fa30b
----------------------------
```

Auch `0x9a` ist ein korrektes OpenPGP-Tag für den Typ 6 (Public Key Packet), das durch die unteren Bits `10` des Tags vier Längenbytes verlangt. Mit dem Kommando

```
rehex | openssl sha256 -binary | xxd -p -c32 | sed 'p;s/.\{48\}$//'
```

erhält man jetzt als Fingerprint

```
  23b80ad55e7e41d7596d3963b678ccc0dc0b89cb580648a82fb5d10484d67fa6
  23b80ad55e7e41d7
```

wobei in der Version 5 die oberen 64 Bit die Key-ID bilden.

3 User ID Packet

Nachdem wir das erste Paket mit Hilfe von Tag und Länge extrahiert haben, finden wir nun auch leicht das nächste Paket. Es beginnt mit `cd 35` und endet nach `0x35=53` Bytes mit `65 3e` in der Mitte der vierten Zeile (vgl. Seite 3)[10].

```
---------------------------
cd 35    : user ID packet
---------------------------
            : packet tag 1100 1101
            : type "2#001101" aka type 13 (User ID packet [4.3])
            : packet length 0x35 : one octet : len=53 < 192
 45726e737420472047696573736d616e6e203c67696573736d616e6e40696e66
 6f726d6174696b2e68752d6265726c696e2e64653e
            : "Ernst G Giessmann <giessmann@informatik.hu-berlin.de>"
---------------------------
```

Das User ID Packet enthält üblicherweise UTF8-kodierte Daten, die dem Schlüsselinhaber zugeordnet sind. Es gibt für den Inhalt keinerlei Vorgaben, jedoch hat sich hier die Adresse des Schlüsselinhabers in der vom RFC 5322 [5] definierten Form durchgesetzt. Um Darstellungsprobleme zu vermeiden, sollte man auch keine andere Kodierung verwenden.

Das hier beschriebene Paket hat auch wieder das neue Längenformat, das durch die ersten beiden Bits `11` des Tag-Bytes angezeigt wird. Die Typ-Bits `001101` des Tags ergeben den Typ 13 (User ID Packet [1, 4.3]) und, da das erste Längenbyte `0x35=53` kleiner als 192 ist, ist es zugleich die Gesamtlänge des Pakets.

Im alten Längenformat hätte das Paket übrigens das Tag-Byte `0xb4=2#10110100`. Die Interpretation dieser Tag-Bits sollte aber nun jeder Leserin leicht fallen.

4 Signature Packet

Schlüssel und Nutzer-Name sind nun durch OpenPGP-Pakete beschrieben, aber noch nicht aneinander gebunden und noch ungeschützt. In einem Zertifikat würde diese Bindung durch eine Signatur der Zertifizierungsstelle vollzogen. Dabei werden dann auch noch zusätzliche Erweiterungen, wie beispielsweise der Ausstellername, die Schlüsselverwendung oder die Signaturrichtlinien mit erfasst und signiert. Nichts anderes passiert auch bei OpenPGP. Die gleichen Informationen finden wir im Signatur-Paket, das wir nun im Einzelnen behandeln[11].

```
---------------------------
c2 c00e : signature packet
---------------------------
 : packet information
 04         : version : 4
 13         : signature type : 19
 13         : signature algorithm : 19 : ECDSA
```

[10] `xxd -s84 -l55 -c20 -g10 key.bin`

[11] `xxd -p -s139 -l209 key.bin`

```
 08        : hash algorithm 08 : sha256
 : signed attributes
 0076      : length of hashed sub packets in bytes [5.2.3.1]
  : each subpacket is LTV (length type value) [Table 3]
  16 21           : type 0x21 : Issuer Fingerprint [12.2]
   04 222e67ea2aa253a967408abe9f7a3119df9110eb
  05 02 5e0be101  : type 02 : signature creation time
  02 1b 03        : type 1b : key flags : 01 02
  05 09 14B7C27F  : type 09 : key expiration time
  05 0b 09080702  : type 0b : symm algo preferences : 09 08 07 02
  05 15 0a09080b  : type 15 : hash algo preferences : 0a 09 08 0b
  04 16 020301    : type 16 : zip algo preferences : 02 03 01
  38 1a           : type 1a : policy URI
   687474703a2f2f7777772e696e666f726d6174696b2e68752d6265726c696e2e
   64652f7e67696573736d616e2f6770672d63702e747874
        : http://www.informatik.hu-berlin.de/~giessman/gpg-cp.txt
  02 1e 01        : type 1e : key features : 0x01
  02 17 80        : type 17 : key server preferences : 0x80
 : unsigned attributes
 000a : length of unsigned ("unhashed") sub packets in bytes
  09 10 9f7a3119df9110eb : type 10 : Issuer Key ID ([5.2.3.5])
 : signature data
 bd10 : left two bytes of hash value
 : signature value (r,s) as MPIs
 0100 f00b772b95cfa877a76f25398c35eb8ecfba2fdfcc81de9bc230c2cb20b89c1a
 0100 f73aa60a399a1da974832ff11854adb7c0c33b4e6b1041142110cd282e81de37
----------------------------
```

4.1 Tag und Länge

Wir betrachten zunächst das Tag `0xc2=2#11000010` und das erste Längenbyte `0xc0`. Aus dem Tag mit neuem Längenformat ergibt sich der Paket-Typ 2 (Signature Packet). Das war einfach.

Jetzt das Längenbyte `len1=0xc0=192`. Da es größer als 191, aber kleiner als 224 ist, wird die Länge in zwei Bytes `len1` und `len2` kodiert. Damit kann man jetzt Paketlängen `len` von 192 (entspricht `0xc000`) bis 8383 (`0xdfff`) angeben.

Die Formel für die Berechnung der Paketlänge aus den beiden Längenbytes ergibt sich aus dem folgenden Beispiel[12].

```
# packet length bytes are 0xc00e
n=c00e; dc -e "16i${n^^}d100/C0-100*r100%+C0+p"
  206
```

Man läuft bei `dc` und `bc` leicht in eine Falle, da Hexadezimalzahlen in Großbuchstaben angegeben werden müssen. Die Berechnung der kodierten Bytes aus einer gegebenen Paketlänge ist dann entsprechend[13]:

[12] `n=C00E; bc <<< "ibase=16; ($n/100-C0)*100+$n%100+C0"`

[13] Ganz genau: `bc <<< "obase=16; if ((191<$n)&&($n<8384)) (192+($n-192)/256)*256+($n-192)%256"`

```
# packet length is 206
n=206; dc -e "$n d_192+256/192+256*r_192+256%+16op"
  C00E
```

Wäre das erste Längenbyte gleich `0xff=255`, dann würden auf das Tag-Byte nicht zwei, sondern vier weitere Bytes folgen, mit denen dann die Länge angeben wird. Damit lassen sich Längen bis einschließlich 4 294 967 295 darstellen.

Wenn das erste Längenbyte schließlich aus dem noch verbleibenden Bereich von 224 (`0xe0`) bis 254 (`0xfe`) ist, dann handelt es sich um eine Längenangabe für Teildaten. Diesem Datenblock folgt dann immer ein weiterer Datenblock (mit entsprechender Längenangabe). Das Gesamtpaket aller Teildaten endet mit einem Datenblock, der keine Teildaten-Längenangabe hat.

Für Teildaten sind jedoch nur Pakete zulässig, bei denen die Anzahl der Inhaltsbytes eine Zweierpotenz von $2^0 = 1$ bis $2^{30} = 1\,073\,741\,824$ ist. Dies wird auch noch im Längenbyte kodiert.

Dies kann man leicht den beiden Umrechnungsformeln eines einzelnen Längenbytes[14] `n` in eine dezimale Länge[15] `len` entnehmen, die hier als Beispiel angegeben werden:

```
n=EA; dc -e "16i${n^^}[[error]sr]sad[2rE0%^]xsrd[E0>a]x[FE<a]xlrp"
  1024
len=4096; dc -e "lG$len[16o224+pq]sA[r1+r2/d2%1>G]dsGx1=A[error]p"
  EC
```

Teildatenblöcke können übrigens auch durch das Dateiende oder ein Paket der Länge Null, also ein Tag mit einem Null-Byte (`0x00`), beendet werden.

Für eine Darstellung des Signatur-Pakets im alten Format kommt man mit nur einem Längenbyte `0xce` (206) aus. Das entsprechende Tag-Byte dazu ist `0x88=2#10001000`, die mittleren Bits `0010` definieren den Typ 2. Man kann die Länge aber auch mit zwei Bytes, als `00 ce`, darstellen. Dann müsste man jedoch das Tag-Byte `0x89=2#10001001` verwenden.

4.2 Paket-Informationen

Der Informationsblock des Signatur-Pakets `04 13 13 08` besteht aus der Versionsnummer `0x04`, dem Signaturtyp `0x13=19` (gründliche Prüfung der Bindung von User ID und Public Key), dem Signaturalgorithmus `0x13=19` (EC-DSA) und dem verwendeten Hash-Algorithmus `0x08` (SHA-256).

Für die Bindung von User ID und Public Key kann man die Signaturtypen `0x10`, `0x11`, `0x11` und `0x13` verwenden, die sich in der Stärke der Prüfung dieser Bindung unterscheiden

```
0x10 : generic certification
0x11 : no assertion on certification claim
0x12 : casual certification
0x13 : substantial certification
```

Den Typ 0x13 erhält man automatisch, wenn man sich mit dem Standard-Kommando[16]

[14] `bc <<< "ibase=16; n=EA; if ((DF<n) && (n<FF)) 2^(n%E0)"`

[15] `bc <<< "len=4096; i=0; while (len%2==0) i+=1; len/=2; if (len==1) obase=16;224+i"`

[16] Damit erzeugt man aktuell (Version 2.2.4) einen RSA-Schlüssel mit 3072 Bit. Einen Dialog mit etwas mehr Auswahlmöglichkeiten bekommt man mit dem Kommando `gpg --full-generate-key`.

`gpg --generate-key`

einen Schlüssel erzeugt.

Interessant wäre mal die Untersuchung, ob man in OpenPGP beim Signieren einen ECDSA-Schlüssel auch mit einer stärkeren Hash-Funktion kombinieren kann. Nach dem RFC 4880 [3] wäre beispielsweise die Verwendung von SHA512 (ID `0x0a`) oder nach dem Draft [1] auch schon von SHA3-512 (ID `0x0e`) zulässig. Dann müsste allerdings der Hash-Wert auf die linken (*most significant*) 256 Bit reduziert werden.

4.3 Signierte Daten

Im Signatur Paket folgen nun wie in einer SignerInfo nach CMS [6] die signierten, die unsignierten und dann die Signaturdaten.

Sie sind als Unterpakete im LTV-Format[17] strukturiert. Es beginnt also immer mit dem oder den Längenbytes, die so wie im neuen Längenformat kodiert werden. Wir haben allerdings im Schlüssel keine größeren Unterpakete gefunden, sie kommen immer mit nur einem Längenbyte aus. Die Sortierung der hier vorhandenen zehn Unterpakete ist beliebig, es gibt keine DER (*Distinguished Encoding Rules*) entsprechenden Kodierungsvorschriften.

Nach dem oder den Längenbytes folgt in einem Unterpaket das entsprechende Typ-Byte (vgl. [1, Table 3] oder [3, 5.2.3.1]), mit dem die innere Struktur des Unterpakets festgelegt wird. Zusätzlich könnte man übrigens durch Setzen des zweiten Tag-Bits `0x40` ein Unterpaket als kritisch deklarieren, was ich eigentlich auch hier, etwa bei den `keyflags` erwartet hätte. Dann müsste jede Implementierung dieses Unterpaket und damit die Signatur zurückweisen, wenn sie den Typ des Unterpakets nicht kennt.

Die signierten Daten beginnen mit einer (unkodierten) Längeninformation aus zwei Bytes über alle signierten Unterpakete. Da der Inhalt der meisten Unterpakete beschränkt ist, sollte man damit auch auskommen.

```
---------------------------
 : signed attributes
 0076 : length of hashed sub packets in bytes [5.2.3.1]
  : each subpacket is LTV (length type value) [Table 3]
  16 21            : type 0x21 : Issuer Fingerprint [12.2]
   04 222e67ea2aa253a967408abe9f7a3119df9110eb
  05 02 5e0be101   : type 02 : signature creation time
  02 1b 03         : type 1b : key flags : 01 02
  05 09 14B7C27F   : type 09 : key expiration time
  05 0b 09080702   : type 0b : symm algo preferences : 09 08 07 02
  05 15 0a09080b   : type 15 : hash algo preferences : 0a 09 08 0b
  04 16 020301     : type 16 : zip algo preferences : 02 03 01
  38 1a            : type 1a : policy URI
   687474703a2f2f7777772e696e666f726d6174696b2e68752d6265726c696e2e
   64652f7e67696573736d616e2f6770672d63702e747874
        : http://www.informatik.hu-berlin.de/~giessman/gpg-cp.txt
```

[17] Das ist kein Schreibfehler, das Format ist wirklich *Length-Tag-Value*. Die Längenangabe schließt das Tag mit ein.

```
  02 1e 01        : type 1e : key features : 0x01
  02 17 80        : type 17 : key server preferences : 0x80
---------------------------
```

Wir gehen jetzt auf die einzelnen Unterpakete ein. Anregungen für weitere eigene Pakete findet man in [3, 5.2.3.1].

Issuer Fingerprint

```
---------------------------
  16 21            : type 0x21 : Issuer Fingerprint [12.2]
   04 222e67ea2aa253a967408abe9f7a3119df9110eb
---------------------------
```

Das ist das Äquivalent zum Herausgeber (*Issuer*) eines Zertifikats, oder besser zum *Authority Key Identifier* (AKI).

In der Version 4 des Fingerprints wird SHA-1 verwendet, so dass hier Schlüsselkollisionen nicht grundsätzlich ausgeschlossen sind. Dazu würde sich der Schlüsselinhaber zwei Schlüssel mit dem gleichen Hash-Wert konstruieren und könnte dann beispielsweise bei einer Signaturprüfung den einen durch den anderen ersetzen. So wären plötzlich Signaturen ungültig, wenn der falsche Prüfschlüssel verwendet wird. Bei einem ECDSA-Schlüssel gibt das Problem der Schlüsselkollision jedoch nicht, da man nur einen Eingabeblock für die Kompressionsfunktion hat und man außerdem das Diskrete-Logarithmus-Problem lösen müsste.

Vielleicht sollte man hier noch zusätzlich den Fingerprint in Version 5 angeben (vgl. [3, 12.2]).

```
---------------------------
  16 21            : type 0x21 : Issuer Fingerprint [12.2]
   04 222e67ea2aa253a967408abe9f7a3119df9110eb
  22 21            : type 0x21 : Issuer Fingerprint version 5
   05 23b80ad55e7e41d7596d3963b678ccc0dc0b89cb580648a82fb5d10484d67fa6
---------------------------
```

Ob bei OpenPGP aber zwei Fingerprints überhaupt zulässig sind, habe ich nicht getestet. Bei Zertifikaten ginge das nicht, denn es ist von jedem bestimmten Typ immer nur eine Zertifikatserweiterung erlaubt[18].

Signaturerstellungszeit und Ende der Schlüsselverwendung

Die Signaturerstellungszeit entspricht dem Beginn (`notBefore`) des Gewährleistungszeitraums eines Zertifikats. Zeiten, die vor der Erstellung des Schlüssels (vgl. Seite 5) liegen, werden von GPG zu Recht zurückgewiesen. Das Ende der Schlüsselverwendung (`notAfter`) wird mit Hilfe eines Zeitraums angegeben.

```
---------------------------
  05 02 5e0be101   : type 02 : signature creation time
---------------------------
  05 09 14b7c27f   : type 09 : key expiration time
---------------------------
```

[18] "A certificate MUST NOT include more than one instance of a particular extension" (vgl. [4, 4.2]). Allerdings könnte man den AKI zugleich als `IssuerSerial` und auch als `KeyIdentifier` angegeben.

Hier haben wir mit

```
date -u '+%F %T %Z' --date "@$(( 16#5e0be101 ))"
```

als Erstellung eine Sekunde nach Silvester (2020-01-01 00:00:01 UTC), was nicht ganz glaubwürdig, aber zulässig ist, da der Schlüssel anscheinend genau um Mitternacht erzeugt wurde. Trotzdem sehen diese Zeiten sehr nach der gpg-Option `--faked-system-time` aus.
Der Ablauf der Schlüsselverwendung wird in Sekunden angegeben und ist dann

```
date -u '+%F %T %Z' --date "@$(( 16#5e0be101 + 16#14b7c27f ))"
  2031-01-06 00:00:00 UTC
```

am 6. Januar 2031.

Schlüsselparameter

Beim Typ `0x1b=27` (Key Flags) wird die Schlüsselverwendung (vgl. [3, 5.2.3.22]), so wie bei der Erweiterung `keyUsage` im X.509 [4], durch einzelne Bits beschrieben.

```
---------------------------
02 1b 03 : type 1b : key flags : 01 02
---------------------------
02 1e 01 : type 1e : key features : 01
---------------------------
```

Das achte Bit (`0x01`) steht für Schlüsselsignatur (`certSign`), das siebte Bit (`0x02`) für Datensignatur (`nonRepudiation`). Soll der Schlüssel auch zur Authentisierung benutzbar sein, was in X.509 dem Bit für `digitalSignature` entsprechen würde, müsste das dritte Bit (`0x20`) gesetzt werden. Interessant ist auch das erste Bit (`0x80`), mit ihm kann signalisieren, dass der private Schlüssel im Besitz von mehreren Personen sein kann und folglich eher als Gruppenschlüssel anzusehen ist.

Weitere Bytes könnten folgen, insbesondere wird im aktuellen Draft [1] vorgeschlagen, dass das neunte Bit, also `0x80` im zweiten Byte, einen Zeitstempeldienst anzeigt.

Da sich OpenPGP weiterentwickelt, ist es sinnvoll, wenn eine Schlüsselinhaberin ihrem Gegenüber signalisiert, welche Optionen sie unterstützt. Wir kennen das bereits aus dem RFC 6664 [8] als *Public Key Capabilities*. Im RFC [3, 5.2.3.25] ist bisher nur das Bit (`0x01`) definiert, im Draft [1] sind aktuell schon weitere aufgeführt:

```
---------------------------
  0x01     : packet 18 and 19 (Modification Detection)
  0x02     : packet 20 (AEAD Encrypted Data) and
             version 5 encrypted session key packets (packet 3)
  0x04     : version 5 public key packet and corresponding
             fingerprint format
---------------------------
```

Hier bedeutet das gesetzte Bit `0x01`, dass von anderen Nutzern die neuen Paket-Typen 18 und 19 des RFC 4888 [3] mit eingebautem Integritätsschutz benutzt werden können.

Unterstützte Algorithmen

Dies ist der Hinweis einer Schlüsselinhaberin auf die von ihr unterstützten Algorithmen (vgl. [3, 5.3.7ff] und die Reihenfolge, in der sie verwendet werden sollten.

```
----------------------------
  05 0b 09080702    : type 0b : symm algo preferences : 09 08 07 02
  05 15 0a09080b    : type 15 : hash algo preferences : 0a 09 08 0b
  04 16 020301      : type 16 : zip algo preferences : 02 03 01
----------------------------
```

Wir sehen hier als symmetrische Algorithmen (vgl. [3, 9.3]) AES-256 (0x09), AES-192 (0x08), AES-128 (0x07) und 3DES (0x02) mit 168-Bit-Schlüssel.

Bei den Hash-Funktionen (vgl. [3, 9.5]) sind es die SHA2-Algorithmen: SHA512 (0x0a), SHA384 (0x09), SHA256 (0x08) und SHA224 (0x0b). Die SHA3-Funktionen, wie beispielsweise SHA3-256 (0x0c) und SHA3-512 (0x0e), sind im Draft 4880bis [1] auch schon vorgesehen, sind aber im RFC [3] noch nicht definiert. SHA-1 (0x02) findet man nicht in der Liste, er wird aber nach dem RFC 4880 (vgl. [3, 13.3.2]) automatisch ergänzt und kann durch diese Auflistung daher nicht ausgeschlossen werden. Das wird aber wahrscheinlich zukünftig geändert (vgl. [1, 9.5]). Denn dann wird nicht mehr die Unterstützung von SHA-1, sondern die von SHA256 verlangt und automatisch ergänzt. Damit wird dann durch die hier angegebene Liste der unterstützten Algorithmen SHA-1 auch ausgeschlossen.

Bei den Kompressionsfunktionen (vgl. [3, 9.4]) haben wir ZLIB (0x02), die Kompression nach RFC 1950 [9], Bzip2 (0x03) und ZIP (0x01), die von PGP ursprünglich genutzte Kompression mit `deflate` nach RFC 1951 [10]. Hier wird zusätzlich natürlich auch Uncompressed (0x00) unterstützt.

Signaturrichtlinie

Dieses Unterpaket (Typ 26 vgl. [3, 5.2.3.21]) ist das Äquivalent zum Link `CPSuri` des X.509 [4] auf die Zertifizierungsrichtlinie. Sie ist natürlich auch signiert[19].

```
----------------------------
  38 1a             : type 1a : policy URI
   687474703a2f2f7777772e696e666f726d6174696b2e68752d6265726c696e2e
   64652f7e67696573736d616e2f6770672d63702e747874
        : http://www.informatik.hu-berlin.de/~giessman/gpg-cp.txt
----------------------------
```

Eigenschaften des Schlüsselservers

Bei dem Typ 23 (Key Server Preferences [3, 5.2.3.5]) ist bislang nur ein Bit, das erste (0x80) definiert.

```
----------------------------
  02 17 80         : type 17 : key server preferences : 0x80
----------------------------
```

[19] Die CRC24-Prüfsumme in der Signatur ist übrigens `EG+G` (vgl. Abschnitt 8).

Ist es gesetzt, darf der Schlüsselserver keine weiteren Schlüsselbestätigungen aufnehmen und verteilen, es sei denn, dass die Schlüsselinhaberin sie selbst in ein Signaturpaket aufgenommen hat.

4.4 Unsignierte Daten

Die unsignierten Daten, im RFC [3] aber nur als „unhashed data“ bezeichnet, dienen ausschließlich der Information. Warum sich die Key ID noch hier befindet, wo sie doch durch den Fingerprint (vgl. Abschnitt 4.3) bereits festgelegt ist, entzieht sich meiner Kenntnis.

```
--------------------------
 : unsigned attributes
 000a : length of unsigned ("unhashed") sub packets in bytes
  09 10 9f7a3119df9110eb : type 10 : Issuer Key ID ([5.2.3.5])
--------------------------
```

4.5 Signaturdaten

Die Signaturdaten enthalten zuerst die oberen zwei Byte des Hash-Wertes der zu signierenden Daten. Damit wäre eine einfache Plausibilitätsprüfung schon vor einer eigentlichen Signaturprüfung möglich. Diese beiden Bytes waren bei der Schlüsselanalyse sehr hilfreich, zeigten sie doch sofort, ob die zu signierenden Daten schon richtig zusammengestellt waren. Allerdings werden diese Bytes von GnuPG seit langem schon nicht mehr geprüft. Wir nutzen das im Abschnitt 8 aus.

```
--------------------------
 : signature data
 bd10 : left two bytes of hash value
 : signature value (r,s) as MPIs
 0100 f00b772b95cfa877a76f25398c35eb8ecfba2fdfcc81de9bc230c2cb20b89c1a
 0100 f73aa60a399a1da974832ff11854adb7c0c33b4e6b1041142110cd282e81de37
--------------------------
```

Für die Signaturerstellung und dann auch jede Signaturprüfung werden die folgenden Daten gehasht. Man beachte jedoch dabei, dass hier, obwohl es auf den ersten Blick so aussieht, nicht die entsprechenden Pakete des Schlüssels verwendet werden. Statt dessen werden die Tags und Längenangaben geändert.

```
-----BEGIN dtbs.bin to be signed-----
99 0052     : tag 99 and two bytes length [5.2.4]
            : Note: this is not the Public-Key Packet header
--------------------------
 04           : version number
 5e0be100     : key created 2020-01-01 00:00:00Z (1577833200)
 13           : public key algorithm ECDSA (ID 19 [9.1])
  08 2a8648ce3d030107 : curve OID : nistp256
  0203        : public EC point als MPI (multi precision integer)
  04          : public point format byte
   d0f1a4d1f2dd66ca5b88d403f6aecc723a40891e05e2f91fc41be1bf9823a96d
```

```
   44f9677a219ecf248b7f074282eb8a80dbf882b9b997a7c8226ee4a6af4fa30b
-----CONTINUED-----
b4 00000035 : tag b4 and four bytes length
            : Note: this is not the User ID packet header
--------------------------
 45726e737420472047696573736d616e6e203c67696573736d616e6e40696e66
 6f726d6174696b2e68752d6265726c696e2e64653e
            : "Ernst G Giessmann <giessmann@informatik.hu-berlin.de>"
-----CONTINUED-----
=c2 c00e : signature packet : do not include this header
--------------------------
 : packet information
 04     : version : 4
 13     : signature type : 19
 13     : signature algorithm : 19 : ECDSA
 08     : hash algorithm 08 : sha256
 : signed attributes
 0076 : length of hashed sub packets in bytes [5.2.3.1]
  : each subpacket is LTV (length type value) [Table 3]
  16 21            : type 0x21 : Issuer Fingerprint [12.2]
   04 222e67ea2aa253a967408abe9f7a3119df9110eb
  05 02 5e0be101   : type 02 : signature creation time
  02 1b 03         : type 1b : key flags : 01 02
  05 09 14B7C27F   : type 09 : key expiration time
  05 0b 09080702   : type 0b : symm algo preferences : 09 08 07 02
  05 15 0a09080b   : type 15 : hash algo preferences : 0a 09 08 0b
  04 16 020301     : type 16 : zip algo preferences : 02 03 01
  38 1a            : type 1a : policy URI
   687474703a2f2f7777772e696e666f726d6174696b2e68752d6265726c696e2e
   64652f7e67696573736d616e2f6770672d63702e747874
        : http://www.informatik.hu-berlin.de/~giessman/gpg-cp.txt
  02 1e 01        : type 1e : key features : 0x01
  02 17 80        : type 17 : key server preferences : 0x80
=000a : unhashed sub packets : do not include these sub-packets
= 09 10 9f7a3119df9110eb : keyID
-----CONTINUED-----
04ff 0000007C : version 4 signature with a final trailer:
              :  04 ff followed by four bytes big endian length of
              :  the sum of length of the hashed sub packets (0x76)
              :   + 4 : length of sig packet header (04131308)
              :   + 2 : length of len octets 0076 itself
              : stopping right before the 0x04ff octets [5.2.4]
-----END dtbs.bin to be signed-----
```

Das Schlüsselpaket und das Paket mit der User ID bekommen für die Signaturerstellung standardisierte Tag- und Längenbytes. Sie sind deshalb unabhängig vom verwendeten Paket-Format. Die Inhalte werden unverändert übernommen. Das geänderte Schlüsselpaket bleibt dabei sogar als

OpenPGP-Paket noch korrekt, das Format des geänderten „Pakets“ mit der User ID ist dagegen nicht mehr OpenPGP-konform.

Vom Signaturpaket wird der Header nicht verwendet, was auch sinnvoll ist, da ja die Längenangaben auch die Signaturdaten einschließen, die aber zum Zeitpunkt der Signaturerstellung noch gar nicht vorhanden sind.

Die Länge der mitsignierten Daten aus dem Signaturpaket wird jedoch durch zusätzliche Bytes am Ende der signierten Daten geschützt. Dieser Anhang besteht in Version 4 aus dem Versions-Byte `0x04`, dem Byte `0xff` und einer Längenangabe (vier Bytes) der signierten Daten aus dem Signaturpaket, beginnend mit der Versionsangabe `04` im Header und endend vor den beiden ersten Trailer-Bytes `04 ff`.

Mit dem Kommando

```
rehex > dtbs.bin; openssl sha256 -binary dtbs.bin | xxd -p -c32
  bd10735c8a4650451fc2dfac0c7ecd2d58a71e59bd642887b40fdf1f24a1aa40
```

erzeugen wir uns die zu signierenden Daten `dtbs.bin` und lassen uns den Hash-Wert dafür anzeigen. Wenn bis jetzt alles richtig ist, sind beiden oberen Bytes des Hash-Werts `bd 10`, die wir bereits im Signatur-Paket gesehen haben.

4.6 Signaturprüfung

Für eine unabhängige Signaturprüfung benötigen wir den öffentlichen Schlüssel aus dem Schlüsselpaket. Dafür nutzen wir eine Konfigurationsdatei *ec_pub.conf* von OpenSSL.

```
#-----BEGIN ec_pub.conf-----
asn1=SEQUENCE:ECC_PubKeyInfo
# pubkeyinfo contains an algorithm identifier and the public key wrapped
# in a BIT STRING [SECG1, C.3]
[ECC_PubKeyInfo]
algorithm=SEQUENCE:ecc_algorithm
pubkey=FORMAT:HEX,BITSTRING:04\
d0f1a4d1f2dd66ca5b88d403f6aecc723a40891e05e2f91fc41be1bf9823a96d\
44f9677a219ecf248b7f074282eb8a80dbf882b9b997a7c8226ee4a6af4fa30b
# algorithm identifier is one OID for ECC and one for a curve
[ecc_algorithm]
algorithm=OID:id-ecPublicKey
parameter=OID:prime256v1
#-----END ec_pub.conf-----
```

In diese Datei tragen wir die Bytefolge des öffentlichen Schlüssel einschließlich des Formatierung-Bytes `0x04` ein. Die beiden Backslashs \ im Schlüssel sind hier notwendig. Mit dem OpenSSL-Kommando[20]

```
openssl asn1parse -genconf ec_pub.conf -out ec.pub
```

erzeugen wir uns den öffentlichen OpenSSL-Schlüssel, dessen Binärdaten man auch base64-kodiert (PEM-Format) darstellen kann:

[20] Das Openssl-Kommando `asn1parse` ist etwas geschwätzig. Deshalb ignorieren wir zukünftig die Ausgaben des Kommandos mit `&>/dev/null`.

```
openssl ec -pubin -inform der -in ec.pub 2>/dev/null
  -----BEGIN PUBLIC KEY-----
  MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE0PGk0fLdZspbiNQD9q7McjpAiR4F
  4vkfxBvhv5gjqW1E+Wd6IZ7PJIt/B0KC64qA2/iCubmXp8gibuSmr0+jCw==
  -----END PUBLIC KEY-----
```

Um die Signaturdaten aus dem Signatur-Paket zu prüfen, müssen wir sie noch in das Format von X.509 übertragen. Auch dazu können wir mit *sig.conf* eine Konfigurationsdatei verwenden.

```
#-----BEGIN sig.conf-----
asn1=SEQUENCE:ECDSA_Signature
# signature is a pair (r,s) of integers
[ECDSA_Signature]
r=INT:0xf00b772b95cfa877a76f25398c35eb8ecfba2fdfcc81de9bc230c2cb20b89c1a
s=INT:0xf73aa60a399a1da974832ff11854adb7c0c33b4e6b1041142110cd282e81de37
#-----END sig.conf-----
```

Beide Signaturwerte werden im Signatur-Paket als ganze Zahlen (MPI) angegeben, also jeweils mit zwei vorangehenden Längenbytes, die die jeweilige Bit-Länge der Zahlen angeben. Da hier beide Signaturwerte volle Bitlänge haben, finden wir dort entsprechend `0x0100` (256). In der Konfigurationsdatei benötigen wir diese Angaben aber nicht.

Nachdem wir die Signaturdatei *sig* aus der Konfigurationsdatei *sig.conf* erzeugt haben, können wir nun auch die Signatur prüfen

```
openssl asn1parse -genconf sig.conf -out sig &>/dev/null
openssl dgst -sha256 -binary -verify ec.pub -keyform der -signature sig dtbs.bin
  Verified OK
```

Damit ist auch das Signaturpaket vollständig beschrieben.

5 Secret Key Packet

Den ersten Teil des Schlüssels, also das Schlüsselpaket[21] `c6 52` ... (Abschnitt 2), das Paket[22] mit der User ID `cd 35` ... (Abschnitt 3) und das Signaturpaket[23] `c2 c00e` ... (Abschnitt 4) kann man schon zusammen in den Schlüsselring von GPG importieren.

```
head -c348 key.bin | gpg --import
  gpg: key 9F7A3119DF9110EB: public key imported
  gpg: Total number processed: 1
  gpg: imported: 1
```

Da wir das aber noch nicht wollen, löschen wir ihn wieder.

```
gpg --delete-key 9F7A3119DF9110EB
```

[21] `xxd -p -s  0 -l 84 key.bin`
[22] `xxd -p -s 84 -l 55 key.bin`
[23] `xxd -p -s139 -l209 key.bin`

Bevor wir nun den zweiten Teil des Schlüssels untersuchen, wollen wir uns überlegen, wie man sich ohne GnuPG einen geheimen Schlüssel im OpenPGP-Format erzeugen kann.

Dazu benötigt man die Schlüsseldaten, die bei EC-DSA nur aus einer ganzen Zahl $1 < d < q$ bestehen, wobei q die Ordnung der elliptischen Kurve ist. Da diese Zahl zufällig ausgewählt werden muss, kann man davon ausgehen, dass sie bei einer 256-Bit-Kurve in der Größenordnung von 256 Bit liegt.

Wir nehmen mal als informatives Beispiel[24]

```
d=0xefffffffffffffbefbefa189d75e9feb1e72b7adfa47b2fbefbefbfffffffff
```

Den Schlüssel im X.509-Format erzeugen wir auch diesmal wieder mit dem OpenSSL-Kommando asn1parse aus einer Konfigurationsdatei

```
#-----BEGIN ec_sec.conf-----
asn1=SEQUENCE:EC_SecretKeyInfo
# secretkeyinfo contains a version (always INT:1), the private factor and
# optional curve parameters (tagged [0]) and the public key (tagged [1])
[EC_SecretKeyInfo]
version=INT:1
secretkey=FORMAT:HEX,OCT:\
efffffffffffffbefbefa189d75e9feb1e72b7adfa47b2fbefbefbfffffffff
parameter=EXPLICIT:0,OID:prime256v1
publickey=EXPLICIT:1,FORMAT:HEX,BITSTRING:04\
b5162400febab5d94a732cd6fee7b94ec3cf0aacffd45646348f7f5fbbf6ae9e\
03168ccd36a8ba6a839d4d0ae91736b5313c650b084ec638a11b5e0f682ef982
#-----END ec_sec.conf-----
```

Den zugehörigen öffentlichen Schlüssel haben wir mit einem kleinen Magma-Programm im Online-Calculator [11] berechnet. Die angegebenen Parameter erzeugen die Kurve prime256v1.

```
// EC-Paramater curve prime256v1
p:=StringToInteger(\
"ffffffff00000001000000000000000000000000ffffffffffffffffffffffff",16);
b:=StringToInteger(\
"5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b",16);
x:=StringToInteger(\
"6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296",16);
y:=StringToInteger(\
"4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5",16);
// secret key
d:=StringToInteger(\
"efffffffffffffbefbefa189d75e9feb1e72b7adfa47b2fbefbefbfffffffff",16);
// compute public key
E:=EllipticCurve([GF(p)!(-3),b]); G:=E![x,y]; Q:=d*G;
Qx:=IntegerToString(Integers()!Q[1],16); "0"^(64-#Qx)*Qx;
Qy:=IntegerToString(Integers()!Q[2],16); "0"^(64-#Qy)*Qy;
```

Mit den Kommandos

[24] Die Begründung für gerade diese nicht ganz zufällige Wahl sieht man später.

```
openssl asn1parse -genconf ec_sec.conf -out ec.sec &> /dev/null
openssl ec -check -text -in ec.sec -inform der -noout
  read EC key
  Private-Key: (256 bit)
  priv:
      ef:ff:ff:ff:ff:ff:ff:be:fb:ef:a1:89:d7:5e:9f:
      eb:1e:72:b7:ad:fa:47:b2:fb:ef:be:fb:ff:ff:ff:
      ff:ff
  pub:
      04:b5:16:24:00:fe:ba:b5:d9:4a:73:2c:d6:fe:e7:
      b9:4e:c3:cf:0a:ac:ff:d4:56:46:34:8f:7f:5f:bb:
      f6:ae:9e:03:16:8c:cd:36:a8:ba:6a:83:9d:4d:0a:
      e9:17:36:b5:31:3c:65:0b:08:4e:c6:38:a1:1b:5e:
      0f:68:2e:f9:82
  ASN1 OID: prime256v1
  NIST CURVE: P-256
  EC Key valid.
```

haben wir dann einen Signaturschlüssel zur Verfügung, mit dem man beliebige Daten, also auch die des Signaturpakets *dtbs.bin*, signieren kann:

```
openssl dgst -binary -sign ec.sec -keyform der -sha256 dtbs.bin > sig
openssl asn1parse -inform der -in sig
    0:d=0  hl=2 l=  69 cons: SEQUENCE
    2:d=1  hl=2 l=  33 prim: INTEGER
             :B5162400FEBAB5D94A732CD6FEE7B94EC3CF0AACFFD45646348F7F5FBBF6AE9E
   37:d=1  hl=2 l= 32 prim: INTEGER
             :03168CCD36A8BA6A839D4D0AE91736B5313C650B084EC638A11B5E0F682EF982
```

Daraus erstellen wir jetzt den geheimen OpenPGP-Schlüssel. Er besteht wie der öffentliche aus drei Paketen, dem Schlüsselpaket, dem Paket mit der User ID und dem Signatur-Paket. Die letzteren beiden unterscheiden sich nicht von denen des öffentlichen Schlüssels. Nur das erste Paket mit dem geheimen Schlüssel hat eine etwas erweiterte Struktur (vgl. [1, 5.5.3]):

```
---------------------------
c5 77   : secret key packet [5.5.3]
---------------------------
           : packet tag 1100 0101
           : bit 1 and 2 set (new length format)
           : bit 3-8 : tag 5 (Secret Key Packet [4.3])
           : packet length 0x77=119 : one octet : len < 192
 04        : version
 5e0be100  : key creation date
 13        : ECDSA
  08 2a8648ce3d030107   : OID : nistp256
  0203                  : MPI : public point
   04
   b5162400febab5d94a732cd6fee7b94ec3cf0aacffd45646348f7f5fbbf6ae9e
   03168ccd36a8ba6a839d4d0ae91736b5313c650b084ec638a11b5e0f682ef982
```

```
 00 : string-to-key indicator "00"=not encrypted
    : secret key as MPI [5.6.4]   :  mind the upper case for dc
    : d=$( tr -d ' :%\n' <<< "
 0100 effffffffffffbefbefa189d75e9feb1e72b7adfa47b2fbefbefbfffffffffff
    : " ); dc -e "16ddoi2^sm${d^^}[lm~lr+srd0<G]dsGxlrlmd*%[0n]sndI3^>ndlm>ndI>np"
 196a  : check sum s of secret key data : sum of all bytes mod 2^16 ([5.5.3])
---------------------------
```

Das Tag im neuen Paket-Format ist `0xc5=2#11000101`, woraus sich der Typ 5 (Secret Key Packet [1, 4.3] ergibt. Bei der Länge `0x77=119` kommt man auch wieder mit einem Längenbyte aus.

Danach folgen zuerst die `0x52=82` Bytes des Public Key Packets, danach ein Byte, das den Verschlüsselungstyp angibt und gegebenenfalls noch weitere Parameter des dabei verwendeten kryptographischen Verfahrens. Das kann man sich alles ersparen, wenn man die Schlüsseldaten nicht verschlüsselt (`0x00`).

Bei EC-DSA wird der geheime Schlüssel als ganze Zahl nach zwei Längenbytes für die Bitlänge angegeben. Die letzten beiden Bytes (`0x196a`) sind schließlich ein einfacher Prüfwert[25], die Summe modulo 2^{16} über alle Klartext-Bytes des geheimen Schlüssels.

Kombiniert man die drei Pakete, das Schlüsselpaket `c5 77 ...` mit dem Paket der User ID `cd 35 ...` und dem Signaturpaket `c2 c00e ...` in eine Datei *xx.bin*,

```
rehex > xx.bin
tail -c +85 key.bin | head -c 55 >> xx.bin
tail -c+140 key.bin | head -c209 >> xx.bin
gpg --list-packets xx.bin
```

so erhält man einen geheimen OpenPGP-Schlüssel. Der Import dieses Schlüssels schlägt natürlich fehl, weil das Signaturpaket mit der Issuer Key ID `9F7A3119DF9110EB` nicht zu unserem „informativen“ Schlüssel mit der Key ID `26E06177FD22A189` passt.

Aber auch mit dem Schlüsselpaket

```
---------------------------
c5 77   : secret key packet [5.5.3]
---------------------------
           : packet tag 1100 0101
           : bit 1 and 2 set (new length format)
           : bit 3-8 : tag 5 (Secret Key Packet [4.3])
           : packet length 0x77=119 : one octet : len < 192
 04        : version
 5e0be100  : key creation date
 13        : ECDSA
  08 2a8648ce3d030107   : OID : nistp256
  0203                  : MPI : public point
   04
   d0f1a4d1f2dd66ca5b88d403f6aecc723a40891e05e2f91fc41be1bf9823a96d
   44f9677a219ecf248b7f074282eb8a80dbf882b9b997a7c8226ee4a6af4fa30b
 00      : string-to-key indicator "00"=not encrypted
```

[25] `bc <<< "obase=16; ibase=16; d=${d^^}; s=0; while (d>0) {s+=d%100; d/=100}; s%=10000; s"`

```
         : secret key as MPI [5.6.4]
 0100 EFFFFFFFFFFFFBEFBEFA189D75E9FEB1E72B7ADFA47B2FBEFBEFBFFFFFFFFFF
 196A    : check sum s of secret key data : sum mod 65536 ([5.5.3])
---------------------------
```

bei dem wir den korrekten öffentlichen Schlüssel eingetragen haben, lässt sich GPG nicht überlisten und verweigert sich mit einer etwas krausen Fehlermeldung

```
gpg --import xx.bin
  gpg: key 9F7A3119DF9110EB: public key "Ernst G Giessmann <giessmann@informa
  tik.hu-berlin.de>" imported
  gpg: key 9F7A3119DF9110EB/9F7A3119DF9110EB: error sending to agent: Inappro
  priate ioctl for device
  gpg: Total number processed: 1
  gpg: imported: 1
  gpg: secret keys read: 1
```

Mal abgesehen von den eigentlichen Schlüsseldaten würde der (unverschlüsselte!) geheime Schlüssel etwa so aussehen:

```
base64 -w64 xx.bin
  xXcEXgvhABMIKoZIzj0DAQcCAwTQ8aTR8t1myluI1AP2rsxyOkCJHgXi+R/EG+G/
  mCOpbUT5Z3ohns8ki38HQoLrioDb+IK5uZenyCJu5KavT6MLAAEA7///////++++
  +hidden+secret+key+++++///////8Zas01RXJuc3QgRyBHaWVzc21hbm4gPGdp
  ZXNzbWFubkBpbmZvcm1hdGlrLmh1LWJlcmxpbi5kZT7CwA4EExMIAHYWIQQiLmfq
  KqJTqWdAir6fejEZ35EQ6wUCXgvhAQIbAwUJFLfCfwULCQgHAgUVCgkICwQWAgMB
  OBpodHRwOi8vd3d3LmluZm9ybWF0aWsuaHUtYmVybGluLmRlL35naWVzc21hbi9n
  cGctY3AudHh0Ah4BAheAAAoJEJ96MRnfkRDrvRABAPALdyuVz6h3p28lOYw1647P
  ui/fzIHem8IwwssguJwaAQD3OqYKOZodqXSDL/EYVK23wMM7TmsQQRQhEMOoLoHe
  Nw==
```

Nur sollte man eine private Schlüsseldatei nie[26] in dieser Form speichern, weil die Schlüsseldaten darin ungeschützt sind.

In durch eine Passphrase gesicherter Form stellt sich ein Schlüsselpaket dagegen so dar:

```
---------------------------
c5 a5
---------------------------
 04        : version number
 5e0be100  : key created 2020-01-01 00:00:00Z
 13        : public key algorithm ECDSA
  08 2a8648ce3d030107  : curve OID : nistp256
  0203     : public EC point as MPI (multi precision integer)
  04       : public point format byte
   D0F1A4D1F2DD66CA5B88D403F6AECC723A40891E05E2F91FC41BE1BF9823A96D
   44F9677A219ECF248B7F074282EB8A80DBF882B9B997A7C8226EE4A6AF4FA30B
------------- encrypted secret key data follows
 fe   : string-to-key identifier given
```

[26] nie!

```
 07    : encryption algorithm : AES-128
  03   : iterated and salted S2K [3.7.1.3]
  02   : hash algo : SHA-1
  be02e2693cf457f4  : 8-octet salt value
  e8   : encoded count : 25165824
       : c=e8; dc -e "16i${c^^}d10%10+r2r10/6+^*p"
 c411fa2bfe5f7d951af3e3da04a8a913 : IV
       : encoded MPI including bit length prefix
 4716 30b6e1c36d2d45fe7a772875e77eb1e72b7adfa47b2fb1502d8fb07917b52d9d
       : encoded hash value (SHA-1)
 34177eb216b5fa16ac87ed3ad222d19dc0ee5b01
-----------------------
```

Statt der einfachen Prüfsumme modulo 2^{16} wird hier der SHA1-Hashwert der geheimen Schlüsseldaten mit verschlüsselt. Zusammen mit der User ID und dem Signaturpaket ergibt sich dann folgender geschützter geheimer OpenPGP-Schlüssel

```
  xaUEXgvhABMIKoZIzj0DAQcCAwTQ8aTR8t1myluI1AP2rsxy0kCJHgXi+R/EG+G/
  mC0pbUT5Z3ohns8ki38HQoLrioDb+IK5uZenyCJu5KavT6ML/gcDAr4C4mk89Ff0
  6MQR+iv+X32VGvPj2gSoqRNHFjC24cNtLUX+encoded+secret+key+xUC2PsHkX
  tS2dNBd+sha1+hash+060iLRncDuWwHNNUVybnN0IEcgR2llc3NtYW5uIDxnaWVz
  c21hbm5AaW5mb3JtYXRpay5odS1iZXJsaW4uZGU+wsAOBBMTCAB2FiEEIi5n6iqi
  U6lnQIq+n3oxGd+REOsFAl4L4QECGwMFCRS3wn8FCwkIBwIFFQoJCAsEFgIDATga
  aHR0cDovL3d3dy5pbmZvcm1hdGlrLmh1LWJlcmxpbi5kZS9+Z2llc3NtYW4vZ3Bn
  LWNwLnR4dAIeAQIXgAAKCRCfejEZ35EQ670QAQDwC3crlc+od6dvJTmMNeuOz7ov
  38yB3pvCMMLLILicGgEA9zqmCjmaHal0gy/xGFStt8DDO05rEEEUIRDNKC6B3jc=
```

Da der private Schlüssel nach der Entschlüsselung nicht zum hier gewählten verschlüsselten Hash-Wert passt, kann man damit wegen der korrekten Signatur im Signaturpaket zwar wieder den korrekten öffentlichen Schlüssel, aber natürlich nicht den privaten Schlüssel importieren.

Die Paket-Struktur dieser Schlüsseldatei ist jedoch formal in Ordnung[27] und wird hier noch einmal zum Vergleich angegeben:

```
base64 -d | gpg --list-packets
  # off=0 ctb=c5 tag=5 hlen=2 plen=165 new-ctb
  :secret key packet:
        version 4, algo 19, created 1577836800, expires 0
        pkey[0]: [72 bits] nistp256 (1.2.840.10045.3.1.7)
        pkey[1]: [515 bits]
        iter+salt S2K, algo: 7, SHA1 protection, hash: 2, salt: BE02E2693CF457F4
        protect count: 25165824 (232)
        protect IV:  c4 11 fa 2b fe 5f 7d 95 1a f3 e3 da 04 a8 a9 13
        skey[2]: [v4 protected]
        keyid: 9F7A3119DF9110EB
  # off=167 ctb=cd tag=13 hlen=2 plen=53 new-ctb
  :user ID packet: "Ernst G Giessmann <giessmann@informatik.hu-berlin.de>"
  # off=222 ctb=c2 tag=2 hlen=3 plen=206 new-ctb
```

[27] Salt-Wert und Initialisierungsvektor kann man selbstverständlich auch nicht voraussagen. Sie werden bei jeden Export neu erzeugt.

```
  :signature packet: algo 19, keyid 9F7A3119DF9110EB
        version 4, created 1577836801, md5len 0, sigclass 0x13
        digest algo 8, begin of digest bd 10
        hashed subpkt 33 len 21
               (issuer fpr v4 222E67EA2AA253A967408ABE9F7A3119DF9110EB)
        hashed subpkt 2 len 4 (sig created 2020-01-01)
        hashed subpkt 27 len 1 (key flags: 03)
        hashed subpkt 9 len 4 (key expires after 11y7d23h59m)
        hashed subpkt 11 len 4 (pref-sym-algos: 9 8 7 2)
        hashed subpkt 21 len 4 (pref-hash-algos: 10 9 8 11)
        hashed subpkt 22 len 3 (pref-zip-algos: 2 3 1)
        hashed subpkt 26 len 55
               (policy: http://www.informatik.hu-berlin.de/~giessman/gpg-cp.txt)
        hashed subpkt 30 len 1 (features: 01)
        hashed subpkt 23 len 1 (keyserver preferences: 80)
        subpkt 16 len 8 (issuer key ID 9F7A3119DF9110EB)
        data: [256 bits]
        data: [256 bits]
```

6 Subkey Packet

Tag- und Längenbytes des nächsten Pakets[28] sind `0xce` und `0xc0c5`. Es handelt sich also wegen der Bitdarstellung `1100 1110` um ein Paket im neuen Format mit dem Typ 14 (Public-Subkey Packet [1, 4.3]) für einen öffentlichen Unterschlüssel. Die Länge `0x0185` (389) ist durch zwei Bytes kodiert

```
n=c0c5; dc -e "16i${n^^}d100/C0-100*r100%+C0+p[0x]n10op"
  389
  0x185
```

und mit Berücksichtigung der drei Header-Bytes erhalten wir durch `xxd -p -s348 -l392 key.bin` das nächste Paket.

```
----------------------------
ce c0c5      : subkey packet
----------------------------
 04             : version
                : packet tag 1100 1110
 5e134b70       : created 2020-01-06 15:00:00 UTC
                : date '+%F %T' --date "@$(( 16#5e134b70 ))"
 01             : public key algorithm RSA (ID 01)
 0bc0           : modulus as MPI
  c61a4c85b2b6fbff80ecbc60d6058ab35d19e5c06ea41ee5fcfff12b9ecb46f8689eb2c99a
  9e7fff75be56d629cbf3b77223840cb4e8b1ef2d29066707c7e2b2b714052fb39ab3bcfd31
  186964a4d99c3b748c7b6ff5ad86bff67eb734f4fda638a684eea3654742a8d72ddf28f84c
  43812de8464027395a707c7e2b231186964a4d99f3f43b748c7b6ff5ad86bff67eb734f4fd
```

[28] `xxd -p -s348 -l3 key.bin`

```
  a638a684eea3654742a8d72ddf28f84c43812de8464027395a437224ced7774a5a708f6352
  04deca3f7125a46333bff6dc6296bd83c3eaca419b4e6fa01c471884096ebffb40cc485a84
  247d3ea046910523a20e32de7d30792c6c0f990d5246f3146d013fffa1e67d1a4264fb0976
  5dffb9beb9deed27b8f9228d12dfdaad73476650f92fc338ccceddf8d8a17791ce50486aba
  ab1429ad8ed2e3d626f6912eb7ee7b8b1f4bcc31661176d90d123af7ce127b219f81f6239a
  a1d69579ee1b64da9a0f06fbecb8198831d7d9159ddd5ab4c4f1398b05407685c8be082cce
  f0519b573bb9
 0017           : exponent as MPI
  400001
---------------------------
```

Es handelt sich um einen RSA-Schlüssel und mit dem Kommando[29]

```
rehex | gpg --list-packets
  # off=0 ctb=ce tag=14 hlen=3 plen=389 new-ctb
  :public sub key packet:
        version 4, algo 1, created 1578322800, expires 0
        pkey[0]: [3008 bits]
        pkey[1]: [23 bits]
        keyid: 703E8DB88FBBF0A3
```

erfahren wir die Schlüssellänge des Moduls (3008 Bit) und des Exponenten (23 Bit), sowie die Key ID, obwohl man die letztere natürlich auch wie im Abschnitt 2 mit dem Kommando

```
rehex | openssl sha1 -binary | xxd -p -c32 | sed 's/.\{24\}//'
```

aus dem folgenden Datenblock selbst berechnen könnte (`0x703e8db88fbbf0a3`)[30].

```
---------------------------
99 0185  : data to be hashed for fingerprint and Key ID
---------------------------
 04         : version
 5e134b70   : created 2020-01-06 15:00:00 UTC
 01         : public key algorithm RSA (ID 01)
 0bc0
  c61a4c85b2b6fbff80ecbc60d6058ab35d19e5c06ea41ee5fcfff12b9ecb46f8689eb2c99a
  9e7fff75be56d629cbf3b77223840cb4e8b1ef2d29066707c7e2b2b714052fb39ab3bcfd31
  186964a4d99c3b748c7b6ff5ad86bff67eb734f4fda638a684eea3654742a8d72ddf28f84c
  43812de8464027395a707c7e2b231186964a4d99f3f43b748c7b6ff5ad86bff67eb734f4fd
  a638a684eea3654742a8d72ddf28f84c43812de8464027395a437224ced7774a5a708f6352
  04deca3f7125a46333bff6dc6296bd83c3eaca419b4e6fa01c471884096ebffb40cc485a84
  247d3ea046910523a20e32de7d30792c6c0f990d5246f3146d013fffa1e67d1a4264fb0976
  5dffb9beb9deed27b8f9228d12dfdaad73476650f92fc338ccceddf8d8a17791ce50486aba
  ab1429ad8ed2e3d626f6912eb7ee7b8b1f4bcc31661176d90d123af7ce127b219f81f6239a
  a1d69579ee1b64da9a0f06fbecb8198831d7d9159ddd5ab4c4f1398b05407685c8be082cce
  f0519b573bb9
 0017
  400001
---------------------------
```

[29] oder `tail -c+349 key.bin | head -c392 | gpg --list-packets`

[30] Mit dem Austausch des Tag-Bytes allein ist es nicht getan, auch die Längenbytes müssen geändert werden.

Interessant ist hier vielleicht auch noch der Exponent, der keine Primzahl ist[31]:

```
dc -e "16i400001[lfp/dlf%0=Fdvsr]sF[dsf]sJdvsr2sf[dlf%0=Flfdd2%+1+sflr<Jd1<M]dsMx"
  5
  397
  2113
dc -e "5 397 2113 **16op"
  400001
```

Aber auch dieser unscheinbare Modul birgt das charakteristische Geheimnis, das wir schon vom PGP-Schlüssel von 2001 kennen und das dem Schlüssel eine gewisse Authentizität verleiht[32]:

```
rehex | base64
  C8DGGkyFsrb7/4DsvGDWBYqzXRnlwG6kHuX8//ErnstG+Giessmann//db5W1inL87dyI4QMtOix
  7yOpBmcHx+KytxQFL7Oas7z9MRhpZKTZnDtOjHtv9a2Gv/Z+tzTO/aY4poTuo2VHQqjXLd8o+ExD
  gS3oRkAnOVpwfH4rIxGGlkpNmfPOO3SMe2/1rYa/9n63NPT9pjimhO6jZUdCqNct3yj4TEOBLehG
  QCc5WkNyJM7XdOpacI9jUgTeyj9xJaRjM7/23GKWvYPD6spBmO5voBxHGIQJbr/7QMxIWoQkfT6g
  RpEFI6IOMt59MHksbA+ZDVJG8xRtAT//oeZ9GkJk+wl2Xf+5vrne7Se4+SKNEt/arXNHZlD5L8M4
  zM7d+Nihd5HOUEhquqsUKa2OOuPWJvaRLrfue4sfS8wxZhF22QOSOvfOEnshn4H2I5qh1pV57htk
  2poPBvvsuBmIMdfZFZ3dWrTE8TmLBUB2hci+CCzO8FGbVzu5
```

In der base64-Kodierung taucht der Name als Zeichenkette auf, was darauf hindeutet, dass zumindest bei einem Primfaktor 114 Bit nicht ganz zufällig gewählt wurden. Es damit zwar ein 3008-Bit-Schlüssel, aber nur mit der Entropie eines 2894-Bit-Schlüssels. Wenn jedoch die verbleibenden Bits zufällig gewählt wurden, erfüllt er damit für die nächsten Jahre hohe Sicherheitsanforderungen und ist deutlich besser als der 1984-Bit Schlüssel von 2001.

7 Subkey Signature Packet

Das letzte Paket ist wieder ein Signaturpaket[33] mit dem Tag `0xc2` (Typ 2 Signature Packet) und der Länge `0x78` (120). Damit wird der Unterschlüssel an den Hauptschlüssel gebunden. Das Paket mit der User ID bleibt hierbei unberücksichtigt.

Mit dem Kommando `xxd -p -s740 -l 122 key.bin` erhalten wir die folgende Darstellung

```
----------------------------
c2 78    : sub key signature packet
----------------------------
 04         : version 4
 18         : signature type 18 : Subkey Binding Signature [5.2.1]
 13         : signature algorithm 19 : ECDSA
 08         : hash algorithm 08 : sha256
 0020       : length of signed ("hashed") sub packets in bytes
  16 21     : type 0x21 : Issuer Fingerprint
   04 222e67ea2aa253a967408abe9f7a3119df9110eb
  05 02     : type 0x02 : signature creation time : 2020-01-06 15:00:01 UTC
```

[31] Es gibt für die Faktorisierung schnellere Implementierungen in `dc`, aber die passen nicht auf eine Zeile.

[32] Die Eingabe ist hier der oben angegebene Modul einschließlich seiner Längenbytes (`0bc0 c61a4c...573bb9`).

[33] `xxd -p -s740 -l2 key.bin`

```
  5e134b71  : date -u '+%F %T %Z' --date "@$(( 16#5e134b71 ))"
  02 1b 0c  : type 0x1b : key flags :
            : bits set: 0x04=encrypt communications
            :           0x08=encrypt storage
 000a       : length of unsigned sub packets in bytes
  09 10     : type 0x10 : Issuer aka 8-octet Key ID
   9f7a3119df9110eb
 : signature data
 5ca3 : left two bytes of hash value
 : signature value (r,s) as MPIs
 00fe 39d313bbcd5ae893b18cbe4ef0917437eb8cde41c5a9a07d99358ea4e4bc1470
 0100 c0ad2879fafcd9ee4d02a8b901731f2930fa1cd8cf65052bcb656c9178ba943a
---------------------------
```

Hier gibt es keine unbekannten Unterpakete mehr. Bei der Schlüsselverwendung sind hier die Bits für die Kommunikations- und Datenverschlüsselung gesetzt.

So können wir uns gleich der Signaturprüfung widmen. Wie schon im Abschnitt 4 gehen die folgenden Daten in die Signaturberechnung ein:

```
-----BEGIN dtbs.bin to be signed-----
99 0052   : tag 99 and two bytes packet length : do not use the Public Key header
---------------------------
 04          : version
 5e0be100    : key creation date
 13          : ECDSA
 08 2a8648ce3d030107 : NIST curve P-256
 0203        : public key data as MPI
  04
   d0f1a4d1f2dd66ca5b88d403f6aecc723a40891e05e2f91fc41be1bf9823a96d
   44f9677a219ecf248b7f074282eb8a80dbf882b9b997a7c8226ee4a6af4fa30b
-----CONTINUED-----
99 0185   : tag 99 and two byte packet length : do not use the subkey header
---------------------------
 04          : version
 5e134b70    : created 2020-01-06 15:00:00 UTC
 01          : RSA
  0bc0 : modulus
   c61a4c85b2b6fbff80ecbc60d6058ab35d19e5c06ea41ee5fcfff12b9ecb46f8689eb2c99a
   9e7fff75be56d629cbf3b77223840cb4e8b1ef2d29066707c7e2b2b714052fb39ab3bcfd31
   186964a4d99c3b748c7b6ff5ad86bff67eb734f4fda638a684eea3654742a8d72ddf28f84c
   43812de8464027395a707c7e2b231186964a4d99f3f43b748c7b6ff5ad86bff67eb734f4fd
   a638a684eea3654742a8d72ddf28f84c43812de8464027395a437224ced7774a5a708f6352
   04deca3f7125a46333bff6dc6296bd83c3eaca419b4e6fa01c471884096ebffb40cc485a84
   247d3ea046910523a20e32de7d30792c6c0f990d5246f3146d013fffa1e67d1a4264fb0976
   5dffb9beb9deed27b8f9228d12dfdaad73476650f92fc338ccceddf8d8a17791ce50486aba
   ab1429ad8ed2e3d626f6912eb7ee7b8b1f4bcc31661176d90d123af7ce127b219f81f6239a
   a1d69579ee1b64da9a0f06fbecb8198831d7d9159ddd5ab4c4f1398b05407685c8be082cce
   f0519b573bb9
  0017 : exponent
```

```
   400001
-----CONTINUED-----
=c2 78    : sub key signature packet header : do not include this header
--------------------------
 04 18 13 08  : version sigclass algo hash
 0020         : length of signed sub packets in bytes [5.2.3.1]
  16 21       : type 0x21 : Issuer Fingerprint [12.2]
   04222e67ea2aa253a967408abe9f7a3119df9110eb
  05 02       : type 0x02 : signature creation time
   5e134b71
  02 1b 0c    : type 0x1b : key flags
=000a : unsigned sub packets : do not hash these sub-packets
= 09 10 : type 0x10 : Issuer aka 8-octet Key ID ([5.2.3.5])
= 9f7a3119df9110eb
-----CONTINUED-----
04ff 00000026 : version 4 signature with final trailer:
              :  04 ff followed by four bytes big endian length of
              :  the sum of length of the hashed sub packets (0x20)
              :   + 4 : length of sig packet header (04131308)
              :   + 2 : length of len octets 0020
              : stopping right before the 0x04ff octets [5.2.4]
-----END dtbs.bin to be signed-----
```

Davon berechnen wir den Hash-Wert

```
rehex > dtbs.bin; openssl sha256 -binary dtbs.bin | xxd -p -c32
  5ca3cb34e9f03661c4d3400f2ab34ed85cf6e3f89663a18bd8f3e1bb1a06d769
```

von denen die beiden oberen 5c a3 mit den Bytes im Signaturpaket übereinstimmen. Das bedeutet, dass wir auf dem richtigen Weg sind.

Das Weitere geht schnell. Zuerst erstellen wir uns nach bewährtem Vorbild für die Signaturdatei *sig* eine Konfigurationsdatei *sig.conf*.

```
#-----BEGIN sig.conf-----
asn1=SEQUENCE:ECDSA_Signature
# signature is pair (r,s) of integers
[ECDSA_Signature]
r=INT:0x39d313bbcd5ae893b18cbe4ef0917437eb8cde41c5a9a07d99358ea4e4bc1470
s=INT:0xc0ad2879fafcd9ee4d02a8b901731f2930fa1cd8cf65052bcb656c9178ba943a
#-----END sig.conf-----
```

Und dann sind es mit dieser Datei nur noch zwei Kommandos

```
openssl asn1parse -genconf sig.conf -out sig &>/dev/null
openssl dgst -binary -verify ec.pub -keyform der -sha256 -signature sig dtbs.bin
  Verified OK
```

Mit „informativen“ Parametern können wir nun auch noch eine Vorlage für den kompletten (unverschlüsselten) privaten Schlüssel nachliefern.

```
  xXcEXgvhABMIKoZIzj0DAQcCAwTQ8aTR8t1myluI1AP2rsxyOkCJHgXi+R/EG+G/
  mCOpbUT5Z3ohns8ki38HQoLrioDb+IK5uZenyCJu5KavT6MLAAEA////////++++
  +hidden+secret+key+++++///////8PJMO1RXJuc3QgRyBHaWVzc21hbm4gPGdp
  ZXNzbWFubkBpbmZvcm1hdGlrLmh1LWJlcmxpbi5kZT7CwA4EExMIAHYWIQQiLmfq
  KqJTqWdAir6fejEZ35EQ6wUCXgvhAQIbAwUJFLfCfwULCQgHAgUVCgkICwQWAgMB
  OBpodHRwOi8vd3d3LmluZm9ybWF0aWsuaHUtYmVybGluLmRlL35naWVzc21hbi9n
  cGctY3AudHh0Ah4BAheAAAoJEJ96MRnfkRDrvRABAPALdyuVz6h3p28lOYw1647P
  ui/fzIHem8IwwssguJwaAQD3OqYKOZodqXSDL/EYVK23wMM7TmsQQRQhEMOoLoHe
  N8fEfAReEOtwAQvAxhpMhbK2+/+A7Lxg1gWKs1OZ5cBupB7l/P/xK57LRvhonrLJ
  mp5//3W+VtYpy/O3ciOEDLTose8tKQZnB8fisrcUBS+zmrO8/TEYaWSk2Zw7dIx7
  b/Wthr/2frc09P2mOKaE7qNlROKo1y3fKPhMQ4Et6EZAJzlacHx+KyMRhpZKTZnz
  9Dt0jHtv9a2Gv/Z+tzTO/aY4poTuo2VHQqjXLd8o+ExDgS3oRkAnOVpDciTO13dK
  WnCPY1IE3so/cSWkYzO/9txilr2Dw+rKQZtOb6AcRxiECW6/+ODMSFqEJHO+oEaR
  BSOiDjLefTB5LGwPmQ1SRvMUbQE//6HmfRpCZPsJdl3/ub653uOnuPkijRLf2q1z
  R2ZQ+S/DOMzO3fjYoXeRzlBIarqrFCmtjtLj1ib2kS637nuLHOvMMWYRdtkNEjr3
  zhJ7IZ+B9iOaodaVee4bZNqaDwb77LgZiDHX2RWd3Vq0xPE5iwVAdoXIvggszvBR
  m1c7uQAXQAABAAu8D/////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  /////////////////+++++hidden+secret+key+++++/////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  /////wXg////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  //8F4P////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  BeD/////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////6UJ
  wngEGBMIACAWIQQiLmfqKqJTqWdAir6fejEZ35EQ6wUCXhNLcQIbDAAKCRCfejEZ
  35EQ61yjAP450xO7zVrok7GMvk7wkXQ364zeQcWpoH2ZNY6k5LwUcAEAwKOoefr8
  2e5NAqi5AXMfKTD6HNjPZQUry2VskXi6lDo=
```

Neu ist hier nur das vierte (unverschlüsselte) Paket mit dem geheimen Unterschlüssel. Die Parameter des geheimen RSA-Schlüssels sind in dieser Reihenfolge: der geheime Exponent d (`skey[2]`), die Primzahl q (`skey[3]`), die Primzahl p (`skey[4]`), und die q-Inverse qi (`skey[4]`). Sie müssen natürlich nicht die hier angegebenen „informativen“ Bitlängen haben.

```
base64 -d | tail -c+386 | head -c1343 | gpg --list-packets
  # off=0 ctb=c7 tag=7 hlen=3 plen=1340 new-ctb
  :secret sub key packet:
        version 4, algo 1, created 1578322800, expires 0
```

```
pkey[0]: [3008 bits]
pkey[1]: [23 bits]
skey[2]: [3004 bits]
skey[3]: [1504 bits]
skey[4]: [1504 bits]
skey[5]: [1504 bits]
checksum: a509
keyid: 703E8DB88FBBF0A3
```

8 Die Prüfsumme am Ende

Nun haben wir den öffentlichen Schlüssel bis ins Kleinste analysiert und damit auch gezeigt, wie man sich den eigenen Schlüssel mit selbst gewählten Parametern und Eigenschaften erstellen kann. Es bleibt jetzt eigentlich nur noch die bemerkenswerte Prüfsumme (++++). Sie ergibt sich als base64-kodiertes Ergebnis `0xfbefbe` der CRC24-Prüfung über alle Bytes des Schlüssels. Da diese alle vorgegeben sind, ist ein so cooler Wert schon ein merkwürdiger Zufall.

Die Zeile beginnt mit einem =, das nicht mit einem Padding-Byte der base64-Kodierung verwechselt werden kann. Denn wegen der Zeilenlänge von 64 Zeichen sind ja die Padding-Bytes der kodierten Schlüsseldaten niemals am Zeilenanfang.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

xlIEXgvhABMIKoZIzj0DAQcCAwTQ8aTR8t1myluI1AP2rsxyOkCJHgXi+R/EG+G/
mCOpbUT5Z3ohns8ki38HQoLrioDb+IK5uZenyCJu5KavT6MLzTVFcm5zdCBHIEdp
ZXNzbWFubiA8Z2llc3NtYW5uQGluZm9ybWF0aWsuaHUtYmVybGluLmRlPsLADgQT
EwgAdhYhBCIuZ+oqolOpZOCKvp96MRnfkRDrBQJeC+EBAhsDBQkUt8J/BQsJCAcC
BRUKCQgLBBYCAwE4GmhOdHA6Ly93d3cuaW5mb3JtYXRpay5odS1iZXJsaW4uZGUv
fmdpZXNzbWFuL2dwZy1jcC50eHQCHgECF4AACgkQn3oxGd+REOu9EAEA8At3K5XP
qHenbyU5jDXrjs+6L9/Mgd6bwjDCyyC4nBoBAPc6pgo5mh2pdIMv8RhUrbfAwztO
axBBFCEQzSgugd43zsDFBF4TS3ABC8DGGkyFsrb7/4DsvGDWBYqzXRnlwG6kHuX8
//ErnstG+Giessmann//db5W1inL87dyI4QMtOix7yOpBmcHx+KytxQFL7Oas7z9
MRhpZKTZnDtOjHtv9a2Gv/Z+tzTO/aY4poTuo2VHQqjXLd8o+ExDgS3oRkAnOVpw
fH4rIxGGlkpNmfPOO3SMe2/1rYa/9n63NPT9pjimhO6jZUdCqNct3yj4TEOBLehG
QCc5WkNyJM7XdOpacI9jUgTeyj9xJaRjM7/23GKWvYPD6spBmO5voBxHGIQJbr/7
QMxIWoQkfT6gRpEFI6IOMt59MHksbA+ZDVJG8xRtAT//oeZ9GkJk+wl2Xf+5vrne
7Se4+SKNEt/arXNHZlD5L8M4zM7d+Nihd5HOUEhquqsUKa2OOuPWJvaRLrfue4sf
S8wxZhF22QOSOvfOEnshn4H2I5qh1pV57htk2poPBvvsuBmIMdfZFZ3dWrTE8TmL
BUB2hci+CCzO8FGbVzu5ABdAAAHCeAQYEwgAIBYhBCIuZ+oqolOpZOCKvp96MRnf
kRDrBQJeEOtxAhsMAAoJEJ96MRnfkRDrXKMA/jnTE7vNWuiTsYy+TvCRdDfrjN5B
xamgfZk1jqTkvBRwAQDArSh5+vzZ7kOCqLkBcx8pMPoc2M9lBSvLZWyReLqUOg==
=++++
-----END PGP PUBLIC KEY BLOCK-----
```

Importiert man diesen Schlüssel und exportiert man ihn darauf, so verschwindet der Name in der Mitte des Schlüsselblocks und natürlich ändert sich auch die Prüfsumme.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mFIEXgvhABMIKoZIzj0DAQcCAwTQ8aTR8t1myluI1AP2rsxyOkCJHgXi+R/EG+G/
mCOpbUT5Z3ohns8ki38HQoLrioDb+IK5uZenyCJu5KavT6MLtDVFcm5zdCBHIEdp
ZXNzbWFubiA8Z2llc3NtYW5uQGluZm9ybWFOaWsuaHUtYmVybGluLmRlPojOBBMT
CAB2FiEEIi5n6iqiU6lnQIq+n3oxGd+REOsFAl4L4QECGwMFCRS3wn8FCwkIBwIF
FQoJCAsEFgIDATgaaHROcDovL3d3dy5pbmZvcm1hdGlrLmh1LWJlcmxpbi5kZS9+
Z2llc3NtYW4vZ3BnLWNwLnR4dAIeAQIXgAAKCRCfejEZ35EQ670QAQDwC3crlc+o
d6dvJTmMNeuOz7ov38yB3pvCMMLLILicGgEA9zqmCjmaHal0gy/xGFStt8DDO05r
EEEUIRDNKC6B3je5AYUEXhNLcAELwMYaTIWytvv/gOy8YNYFirNdGeXAbqQe5fz/
8Suey0b4aJ6yyZqef/91vlbWKcvzt3IjhAyO6LHvLSkGZwfH4rK3FAUvs5qzvP0x
GGlkpNmcO3SMe2/1rYa/9n63NPT9pjimhO6jZUdCqNct3yj4TEOBLehGQCc5WnB8
fisjEYaWSk2Z8/Q7dIx7b/Wthr/2frc09P2mOKaE7qNlROKo1y3fKPhMQ4Et6EZA
JzlaQ3Ikztd3Slpwj2NSBN7KP3ElpGMzv/bcYpa9g8PqykGbTm+gHEcYhAluv/tA
zEhahCR9PqBGkQUjog4y3nOweSxsD5kNUkbzFGOBP/+h5nOaQmT7CXZd/7m+ud7t
J7j5Io0S39qtc0dmUPkvwzjMzt342KF3kc5QSGq6qxQprY7S49Ym9pEut+57ix9L
zDFmEXbZDRI6984SeyGfgfYjmqHWlXnuG2Tamg8G++y4GYgx19kVnd1atMTxOYsF
QHaFyL4ILM7wUZtXO7kAFOAAAYh4BBgTCAAgFiEEIi5n6iqiU6lnQIq+n3oxGd+R
EOsFAl4TS3ECGwwACgkQn3oxGd+REOtcowD+OdMTu81a6JOxjL5O8JFON+uM3kHF
qaB9mTWOpOS8FHABAMCtKHn6/NnuTQKouQFzHykw+hzYz2UFK8tlbJF4upQ6
=6Qiv
-----END PGP PUBLIC KEY BLOCK-----
```

Der Grund dafür findet sich in der Export-Funktion. Während alle Pakete des ursprünglichen Schlüssels das neue Format haben, sind die Pakete des exportierten Schlüssels noch im alten Format. Der Unterschied von einem Byte entsteht aus den Längen-Bytes im Signatur-Paket des Hauptschlüssels. Im neuen Format sind dafür zwei Bytes `0xc00e` notwendig, während im alten Format ein Längenbyte `0xce` ausreichend war (vgl. Abschnitt 4.1). Ansonsten ändern sich nur die Tags und beim Unterschlüssel-Paket noch die Werte der Längen-Bytes. Wandelt man beide Schlüsseldaten in eine Hexadezimaldarstellung mit je einem Byte pro Zeile um[34], kann man den Unterschied durch einfaches `diff` sehen.

```
diff -u0 NeuesFormat.hex AltesFormat.hex
  --- NeuesFormat.hex    2020-04-25 18:11:18.307841700 +0200
  +++ AltesFormat.hex    2020-04-25 18:11:20.420136800 +0200
  @@ -1 +1 @@
  -c6
  +98
  @@ -85 +85 @@
  -cd
  +b4
  @@ -140,3 +140,2 @@
  -c2
  -c0
  -0e
  +88
  +ce
```

[34] `xxd -p -c1`

```
@@ -349,3 +348,3 @@
-ce
-c0
-c5
+b9
+01
+85
@@ -741 +740 @@
-c2
+88
```

Wenn man jetzt die Tag- und Längen-Bytes `88 ce` des Signatur-Pakets im alten Format gegen die ebenfalls zulässigen Bytes `89 00 ce` austauscht[35], dann enthält dieser OpenPGP-Schlüssel im alten Format auch wieder beide Namen.

Ohne korrekte Prüfsumme kann man diesen Schlüssel aber nicht importieren. Um sie nicht selbst berechnen zu müssen, trägt man einfach erstmal irgendeinen Wert ein. GnuPG beschwert sich dann bei dem Kommando `gpg --list-packets` und gibt den korrekten Wert auch gleich mit an. Den übernimmt man und hat dann einen korrekten Schlüssel im alten Format.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mFIEXgvhABMIKoZIzj0DAQcCAwTQ8aTR8t1myluI1AP2rsxyOkCJHgXi+R/EG+G/
mCOpbUT5Z3ohns8ki38HQoLrioDb+IK5uZenyCJu5KavT6MLtDVFcm5zdCBHIEdp
ZXNzbWFubiA8Z2llc3NtYW5uQGluZm9ybWF0aWsuaHUtYmVybGluLmRlPokAzgQT
EwgAdhYhBCIuZ+oqolOpZOCKvp96MRnfkRDrBQJeC+EBAhsDBQkUt8J/BQsJCAcC
BRUKCQgLBBYCAwE4GmhOdHA6Ly93d3cuaW5mb3JtYXRpay5odS1iZXJsaW4uZGUv
fmdpZXNzbWFuL2dwZy1jcC50eHQCHgECF4AACgkQn3oxGd+REOu9EAEA8At3K5XP
qHenbyU5jDXrjs+6L9/Mgd6bwjDCyyC4nBoBAPc6pgo5mh2pdIMv8RhUrbfAwztO
axBBFCEQzSgugd43uQGFBF4TS3ABC8DGGkyFsrb7/4DsvGDWBYqzXRnlwG6kHuX8
//ErnstG+Giessmann//db5W1inL87dyI4QMtOix7yOpBmcHx+KytxQFL7Oas7z9
MRhpZKTZnDtOjHtv9a2Gv/Z+tzTO/aY4poTuo2VHQqjXLd8o+ExDgS3oRkAnOVpw
fH4rIxGGlkpNmfPOO3SMe2/1rYa/9n63NPT9pjimhO6jZUdCqNct3yj4TEOBLehG
QCc5WkNyJM7XdOpacI9jUgTeyj9xJaRjM7/23GKWvYPD6spBmO5voBxHGIQJbr/7
QMxIWoQkfT6gRpEFI6IOMt59MHksbA+ZDVJG8xRtAT//oeZ9GkJk+wl2Xf+5vrne
7Se4+SKNEt/arXNHZlD5L8M4zM7d+Nihd5HOUEhquqsUKa2OOuPWJvaRLrfue4sf
S8wxZhF22QOSOvfOEnshn4H2I5qh1pV57htk2poPBvvsuBmIMdfZFZ3dWrTE8TmL
BUB2hci+CCzO8FGbVzu5ABdAAAGIeAQYEwgAIBYhBCIuZ+oqolOpZOCKvp96MRnf
kRDrBQJeEOtxAhsMAAoJEJ96MRnfkRDrXKMA/jnTE7vNWuiTsYy+TvCRdDfrjN5B
xamgfZk1jqTkvBRwAQDArSh5+vzZ7kOCqLkBcx8pMPoc2M9lBSvLZWyReLqUOg==
=qJkt
-----END PGP PUBLIC KEY BLOCK-----
```

Man kann auch sogar die beeindruckende Prüfsumme (++++) des Original-Schlüssel beibehalten. Denn da die beiden linken Bytes des Hash-Wertes in den Signaturpaketen von `GnuPG` nicht mehr geprüft werden, kann durch eine passende Änderung dieser Bytes die CRC24-Prüfsumme jeden beliebigen Wert annehmen[36].

[35] `sed '^[-=]/d' | base64 -d | xxd -p | sed 's/88ce/8900ce/' | xxd -p -r | base64 -w64`

[36] Für den Prüfwert `++++` kann man zum Beispiel die Bytes `bd10` gegen `d19a` und `5ca3` gegen `b304` austauschen.

Nicht nur im eigenen Schlüssel, sondern auch in jedem fremden, kann folglich jede die jeweiligen linken Bytes der Hash-Werte so verändern, dass als Prüfsumme eine gewünschte Zeichenfolge entsteht und ohne dass der Schlüssel dabei ungültig wird. Eine zugegebenermaßen etwas merkwürdige und wenig vertrauenerweckende Eigenschaft.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mFIEXgvhABMIKoZIzj0DAQcCAwTQ8aTR8t1myluI1AP2rsxyOkCJHgXi+R/EG+G/
mCOpbUT5Z3ohns8ki38HQoLrioDb+IK5uZenyCJu5KavT6MLtDVFcm5zdCBHIEdp
ZXNzbWFubiA8Z2llc3NtYW5uQGluZm9ybWFOaWsuaHUtYmVybGluLmRlPokAzgQT
EwgAdhYhBCIuZ+oqolOpZOCKvp96MRnfkRDrBQJeC+EBAhsDBQkUt8J/BQsJCAcC
BRUKCQgLBBYCAwE4GmhOdHA6Ly93d3cuaW5mb3JtYXRpay5odS1iZXJsaW4uZGUv
fmdpZXNzbWFuL2dwZy1jcC5OeHQCHgECF4AACgkQn3oxGd+REOvRmgEA8At3K5XP
qHenbyU5jDXrjs+6L9/Mgd6bwjDCyyC4nBoBAPc6pgo5mh2pdIMv8RhUrbfAwztO
axBBFCEQzSgugd43uQGFBF4TS3ABC8DGGkyFsrb7/4DsvGDWBYqzXRnlwG6kHuX8
//ErnstG+Giessmann//db5W1inL87dyI4QMtOix7yOpBmcHx+KytxQFL7Oas7z9
MRhpZKTZnDtOjHtv9a2Gv/Z+tzTO/aY4poTuo2VHQqjXLd8o+ExDgS3oRkAnOVpw
fH4rIxGGlkpNmfPOO3SMe2/1rYa/9n63NPT9pjimhO6jZUdCqNct3yj4TEOBLehG
QCc5WkNyJM7XdOpacI9jUgTeyj9xJaRjM7/23GKWvYPD6spBmO5voBxHGIQJbr/7
QMxIWoQkfT6gRpEFI6IOMt59MHksbA+ZDVJG8xRtAT//oeZ9GkJk+wl2Xf+5vrne
7Se4+SKNEt/arXNHZlD5L8M4zM7d+Nihd5HOUEhquqsUKa2OOuPWJvaRLrfue4sf
S8wxZhF22QOSOvfOEnshn4H2I5qh1pV57htk2poPBvvsuBmIMdfZFZ3dWrTE8TmL
BUB2hci+CCzO8FGbVzu5ABdAAAGIeAQYEwgAIBYhBCIuZ+oqolOpZOCKvp96MRnf
kRDrBQJeEOtxAhsMAAoJEJ96MRnfkRDrswQA/jnTE7vNWuiTsYy+TvCRdDfrjN5B
xamgfZk1jqTkvBRwAQDArSh5+vzZ7kOCqLkBcx8pMPoc2M9lBSvLZWyReLqUOg==
=++++
-----END PGP PUBLIC KEY BLOCK-----
```

Und abschließend bemerkt, die Zeichenkette `ABdAA` habe ich übrigens auch wieder gefunden [12].

Literatur

[1] W. Koch, B. Carlson, R.H. Tse, D.A. Atkins, D.K. Gillmor: OpenPGP Message Format, draft-ietf-openpgp-rfc4880bis-09, IETF Internet-Draft, 2020-03-09[37]

[2] W. Polk, R. Housley, L. Bassham: Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, RFC 3279, IETF Standards Track, 2002-14

[3] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, R. Thayer: OpenPGP Message Format, RFC 4880, IETF Standards Track, 2007-11

[4] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. Polk: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, RFC 5280, IETF Standards Track, 2008-05

[37] Die RFCs findet man im Internet unter https://ietf.org. Bei einer Suche reicht die Kombination von „RFC“ und Nummer auch aus.

[5] P. Resnick, Ed.: Internet Message Format, RFC 5322, IETF Standards Track, 2008-10

[6] R. Housley: Cryptographic Message Syntax (CMS), RFC 5652, IETF Standards Track, 2009-09

[7] A. Jivsov: Elliptic Curve Cryptography (ECC) in OpenPGP, RFC 6637, IETF Standards Track, 2012-07

[8] J. Schaad: S/MIME Capabilities for Public Key Definitions, RFC 6664, IETF Informational RFC, 2012-07

[9] P. Deutsch, J-L. Gailly: ZLIB Compressed Data Format Specification version 3.3, RFC 1950, IETF Informational RFC, 1996-05

[10] P. Deutsch: DEFLATE Compressed Data Format Specification version 1.3, RFC 1951, IETF Informational RFC, 1996-05

[11] MAGMA: Online-Calculator, http://magma.maths.usyd.edu.au/calc

[12] A. Yousar: Wer hat ABdAA in seinem PGP-Schlüssel?, Manuskript, 2006-01-16