Podstawy kryptografii

Kryptografia asymetryczna

email: contact@cyberskiller.com
cyberskiller.com

Podstawy kryptografii

Kryptografia asymetryczna

Spis Treści

Wiadomości podstawowe 3
Kryptosystem RSA 4
Biblioteka OpenSSL 5
GPG - Gnu Privacy Guard 7

email: contact@cyberskiller.com
cyberskiller.com

Wiadomości podstawowe

W przeciwieństwie do kryptografii symetrycznej, gdzie do szyfrowania i deszyfrowania używa się tego samego klucza, w asymetrycznych systemach kryptograficznych wykorzystuje się parę sprzężonych kluczy: klucz prywatny i publiczny.

Funkcje kluczy w systemach kryptograficznych

Należy pamiętać, że nazwa klucza w pełni definiuje jego sposób wykorzystania. Klucze prywatne są **`prywatne`**, co oznacza, że wymagają największego stopnia ochrony i poufności. Ich ujawnienie (kompromitacja) czyni je całkowicie bezużytecznymi. Dostęp do nich powinien mieć jedynie ich właściciel. Klucze publiczne są **`publiczne`** i powinny być łatwo dostępne, w szczególności dla osób chcących przekazać zaszyfrowaną za ich pomocą wiadomość.

Klucz prywatny:
- deszyfrowanie wiadomości
- składanie podpisów na wiadomościach

Klucz publiczny:
- szyfrowanie wiadomości
- weryfikowanie podpisów złożonych na wiadomościach

Koncepcja klucza prywatnego i publicznego

Użytkownik **`A`** prowadzi korespondencje z użytkownikami **`B`** i **`C`** stosując asymetryczny system kryptograficzny. Oznacza to, że wygenerował parę kluczy: prywatny i publiczny (oznaczone jako **`A_publ`** i **`A_priv`**), podobnie zresztą jak pozostali użytkownicy. Aby taka komunikacja była możliwa, użytkownik **`A`** musi mieć dostęp do kluczy publicznych użytkowników **`B`** i **`C`**. Podobnie, pozostali użytkownicy także współdzielą swoje klucze publiczne. Użytkownik **`A`** chce przekazać zaszyfrowaną informację do użytkownika **`B`**. Jest w posiadaniu jego klucza publicznego (**`B_publ`**) i wykorzystuje go do zaszyfrowania wiadomości. Nikt poza posiadaczem sprzężonego z kluczem **`B_publ`** klucza **`B_priv`** (zakładamy, że jest to właśnie użytkownik **`B`**) nie jest w stanie odszyfrować tej wiadomości. Podobnie, pozostali użytkownicy, chcąc wymieniać zaszyfrowane wiadomości powinni je szyfrować za pomocą odpowiednich kluczy publicznych osób, do których te wiadomości wysyłają. Otrzymane od innych wiadomości deszyfrują za pomocą własnych kluczy prywatnych.

Poza szyfrowaniem wiadomości, gwarantującym przede wszystkim poufność, kryptografia asymetryczna umożliwia też zachowanie integralności wiadomości lub danych dzięki możliwości tworzenia i weryfikacji podpisów. Jeśli użytkownik **`A`** chcę przekazać użytkownikowi **`C`** wiadomość w taki sposób, aby mieć pewność, że to co wysyła jest dokładnie tym co dociera do użytkownika **`C`**, składa za pomocą swojego klucza prywatnego **`A_priv`** podpis na wiadomości. Użytkownik **`C`** po otrzymaniu wiadomości od użytkownika **`A`** weryfikuje podpis za pomocą klucza publicznego użytkownika **`A`**. Proces weryfikacji podpisu polega na sprawdzeniu, czy zawartość podpisu wynika wprost z treści wiadomości. Jeśli, ktoś zmieniłby treść wiadomości, nie mając dostępu do klucza prywatnego użytkownika **`A`** to nie będzie w stanie wygenerować poprawnego podpisu. Podczas weryfikacji okaże się, że wiadomość i podpis "nie pasują do siebie". Na podstawie podpisu nie można odtworzyć treści wiadomości, ale ich niezgodność świadczy o tym, że treść wiadomości nie jest tożsama z tym co wysłał użytkownik **`A`**.

Kryptosystemy asymetryczne zapewniają:
- poufność - osiąganą poprzez szyfrowanie
- integralność - osiąganą poprzez podpis

Istotnym zagadnieniem jest problem dystrybucji kluczy publicznych. Ktokolwiek ich używa, musi mieć pewność, że należą one do odpowiedniego, uprawnionego właściciela lub podmiotu i zostały przez niego wygenerowane. Można to zrobić na wiele sposobów, ale tylko dwa z nich zyskały uznanie i są stosowane powszechnie. Jeden z nich zostanie opisany w dalszej części tego dokumentu.

Kryptosystem RSA

Kryptosystem RSA jest jednym z bardziej popularnych kryptosystemów asymetrycznych, którego bezpieczeństwo oparte jest na problemie faktoryzacji (rozkładu liczby na czynniki pierwsze). Twórcami tego kryptosystemu są R. Rivest, A. Shamir i L. Adleman. Użytkownik generuje zawszę parę kluczy: klucz prywatny i klucz publiczny. Podstawą mechanizmu generowania kluczy jest wybór dwóch dostatecznie dużych liczb pierwszych. Klucz publiczny jest wyodrębniany z klucza prywatnego i stanowi w najprostszym ujęciu iloczyn tych liczb. Wielkość generowanych kluczy może zmieniać się w przedziale od 256 do 4096 bitów, przy czym klucze o długości poniżej 768 bitów uważa się za nie gwarantujące wystarczającego bezpieczeństwa.

Proces szyfrowania i deszyfrowania opisany jest wzorami:

$$\textit{szyfrowanie: } c \equiv m^e \bmod n$$
$$\textit{deszyfrowanie: } m \equiv c^d \bmod n$$

gdzie:
- m jest blokiem tekstu jawnego o wartości liczbowej nie większej niż n,
- c jest blokiem kryptogramu,

ponadto:
- $n=pq$ jest iloczynem dwóch dużych liczb pierwszych p i q,
- $\varphi(n)=\varphi(p\cdot q)=(p-1)(q-1)$ oznacza wartość funkcji Eulera dla n,
- wartość e jest wybierana tak aby była względnie pierwsza z $\varphi(n)$,
- wartość d wyliczana jest za pomocą zależności: $d \equiv e^{-1} \bmod \varphi(n)$.
- para n i d definiowana jest jako **`klucz prywatny`**
- para n i e definiowana jest jako **`klucz publiczny`**

Z uwagi na ograniczenia związane z rozmiarem bloku tekstu jawnego, co wynika ze specyfiki samego algorytmu, za pomocą tego kryptosystemu nie szyfruje się danych o rozmiarach przekraczających rozmiary wygenerowanego klucza. Choć można pomyśleć o podziale dużego zbioru danych na fragmenty odpowiedniej wielkości, to podejście takie jest niepraktyczne i nie jest stosowane. Problem szyfrowania znaczonych ilości danych z wykorzystaniem kryptografii asymetrycznej rozwiązuje się w następujący sposób:
- generuje się losowy klucz szyfrujący o rozmiarze wymaganym przez algorytm symetryczny (np. 256 bitów),
- dane szyfruje się wykorzystując kryptografię symetryczną i wygenerowany klucz,
- użyty klucz szyfrujący szyfruje się za pomocą klucza publicznego odbiorcy,
- zaszyfrowane dane oraz zaszyfrowany klucz symetryczny przesyła się odbiorcy,
- odbiorca w pierwszej kolejności deszyfruje klucz symetryczny, a następnie za jego pomocą deszyfruje zaszyfrowane dane.

Biblioteka OpenSSL

Podstawowe informacje o bibliotece OpenSSL przedstawione zostały w materiale dotyczącym kryptografii symetrycznej.

Generowanie kluczy

Wykorzystanie algorytmów kryptografii asymetrycznej z użyciem biblioteki OpenSSL zostanie omówione na przykładzie wspomnianego już wcześniej kryptosystemu RSA. Pierwszym, niezbędnym krokiem jest wygenerowanie pary kluczy: klucza prywatnego i klucza publicznego. Do generowania kluczy prywatnych wykorzystamy komendę genrsa. Niezbędne jest także określenie jakiej długości klucz chcemy wygenerować. Popatrzmy na poniższy przykład:

- polecenie generowania klucza prywatnego (kryptosystem RSA) o długości 2048 bitów wraz z zapisaniem go do pliku **private.pem**:

```
openssl genrsa 2048 > private.pem
```

wynik działania polecenia:

```
Generating RSA private key, 2048 bit long modulus (2 primes)
.....................................+++++
.........................+++++
e is 65537 (0x010001)
```

Plik **private.pem** zawiera klucz prywatny zapisany w formacie PEM (Privacy Enhanced Mail) z wykorzystaniem kodowania **Base64**. Oto jego fragment:

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAn1HMolkYW7gJOZYBXTiw8xDNX4zaMxTrYW7w6/i3pHbpPGPC
o+bclJvPoVZohG+QnyNJV25ILW6oJ9rqv4qqxesNQTTepKSguKwMAqPOlnSYwbFB
...
zHWG74GQHoWhR0TDAmZiTnvuw3GBoKW1BNXlQIBH/yDS9eKjuqTbnw==
-----END RSA PRIVATE KEY-----
```

Base64 jest rodzajem kodowania transportowego, pozwalającego na przesyłanie danych binarnych za pomocą tekstowego formatu zapisu informacji.

Kolejnym krokiem jest wyodrębnienie klucza publicznego. Wymaga to wykorzystania kolejnej komendy z repertuaru biblioteki **OpenSSL**:

- generowanie klucza publicznego na podstawie klucza prywatnego:

```
openssl rsa -in private.pem -pubout -out public.pem
```

wynik działania polecenia:

```
writing RSA key
```

W pliku **public.pem** zapisany został klucz publiczny (także z wykorzystaniem kodowania **Base64**).

email: contact@cyberskiller.com
cyberskiller.com

Szyfrowanie i deszyfrowanie

Dysponując parą kluczy możemy przystąpić do szyfrowania danych. Wymaga to wykorzystania trzeciej już komendy biblioteki **OpenSSL - rsautl**. Dane do zaszyfrowania znajdują się w pliku **message.txt**.

```
echo „The quick brown fox jumps over the lazy dog” > message.txt
```

- szyfrowanie:

```
openssl rsa -encryption -inkey public.pem -pubin -in
message.txt > message.enc
```

  - parametr -encryption oznacza, że generowany będzie szyfrogram na podstawie tekstu jawnego, wskazanego poprzez parametr -in,
  - parametr -inkey pozwala na wskazanie pliku z kluczem (w tym przypadku jest to klucz publiczny),
  - parametr -pubin informuje bibliotekę, że ma do czynienia z kluczem publicznym.

Szyfrogram został zapisany w pliku **message.enc**

- deszyfrowanie:

```
openssl rsa -decryption -inkey private.pem -in message.enc >
message_2.txt
```

  należy zwrócić uwagę na fakt, iż do odszyfrowania wykorzystaliśmy klucz prywatny sprzężony z kluczem publicznym użytym do szyfrowania.

Możliwe jest szyfrowanie i deszyfrowanie danych z wykorzystaniem jedynie klucza prywatnego (zawiera on w sobie klucz publiczny), nie ma to jednak znaczenia praktycznego z uwagi na powody dla których opracowany systemy kryptografii asymetrycznej.

Duże bloki danych

Próba zaszyfrowania pliku data.bin o rozmiarze przekraczającym rozmiar klucza powoduje wyświetlenie komunikatu o błędzie:

```
openssl rsautl -encrypt -inkey private.pem -in data.bin -out data.enc

RSA operation error
139797173126528:error:0406D06E:rsa routines:RSA_padding_add_PKCS1_
type_2:
data too large for key size:../crypto/rsa/rsa_pk1.c:124:
```

W takim przypadku należy wykorzystać procedurę opisaną w paragrafie **Kryptosystem RSA**. Do generowania losowego klucza symetrycznego można wykorzystać bibliotekę **OpenSSL** wraz z komendą **rand**, pozwalającą generować ciągi pseudolosowe:

```
openssl rand 1024 | shasum -a 256
>7554fe049add0038de080d9b945ef82c8b3377392365f51acb22aeddc0d7c40a -
```

W powyższym przykładzie, z 1 kB ciągu pseudolosowych bajtów generowany jest skrót **sha-256** o długości 256 bitów, wykorzystywany później jako klucz szyfrujący. Klucz można także wygenerować bezpośrednio z pseudolosowego ciągu bajtów wykorzystując polecenie **xxd**:

```
openssl rand 32 | xxd -p -c 0
> 6460845db50cc46490e9dacaf65f1ec9bae659d6c8dd5dddd31c595e36c4ea96
```


email: contact@cyberskiller.com
cyberskiller.com

Składanie podpisu i jego weryfikacja

Z uwagi na ograniczenia dotyczące wielkości plików, które mogą być przetwarzane przez algorytmy RSA nie można wygenerować podpisu na dowolnym pliku. Z uwagi na fakt, iż celem podpisu jest zagwarantowanie przede wszystkim integralności, podpis można złożyć na pliku zawierającym skrót wygenerowany z pomocą funkcji skrótu. Rozważmy następujący przykład - chcemy podpisać plik **`message.txt`**.

- generujemy skrót i zapisujemy go do pliku **`message.sha`**
  ```
  shasum -a 512 message.txt > message.sha
  ```

- na pliku message.sha generujemy podpis wykorzystując klucz prywatny (**`private.pem`**)
  ```
  openssl rsautl -sign -inkey private.pem -in message.sha > message.sig
  ```
  podpis zostaje zapisany w binarnym pliku **`message.sig`**

- weryfikacja podpisu
  ```
  openssl rsautl -verify -inkey public.pem -pubin -in message.sig
  ```
  polecenia zwraca w zasadzie zawartość pliku message.sha co pozwala na weryfikację integralności - jeśli skrót z pliku i wynik działania polecenia weryfikacji są zgodne, to integralność wiadomości została zachowana.

GPG - Gnu Privacy Guard

GNU Privacy Guard (GnuPG lub GPG) to darmowy program zastępujący pakiet oprogramowania kryptograficznego PGP firmy Symantec. Oprogramowanie to jest zgodne z dokumentem RFC 4880, specyfikacją OpenPGP opracowaną przez IETF w ramach ścieżki standardów. Współczesne wersje PGP są kompatybilne z GnuPG i innymi systemami zgodnymi z OpenPGP.

GnuPG jest hybrydowym systemem szyfrującym, z racji tego, że wykorzystuje kombinację kryptografii klucza symetrycznego w celu zapewnienia szybkości działania oraz kryptografii klucza publicznego. Pozwala ponadto na zarządzanie lokalnym repozytorium kluczy oraz bezpieczną dystrybucję kluczy publicznych. w oparciu o model zaufania “Web of Trust”.

GnuPG szyfruje wiadomości przy użyciu par kluczy asymetrycznych generowanych indywidualnie przez użytkowników GnuPG. Uzyskane w ten sposób klucze publiczne mogą być wymieniane z innymi użytkownikami na różne sposoby, najczęściej poprzez tak zwane serwery kluczy. Należy zawsze wymieniać je z ostrożnością, aby zapobiec fałszowaniu tożsamości i mieć pewność, że dany klucz publiczny należy faktycznie do jego właściciela. Służy temu mechanizm podpisywania kluczy publicznych przez innych użytkowników, potwierdzających w ten sposób tożsamość ich właścicieli.

Procedura generowania kluczy

1. Generowanie pary kluczy dla kryptosystemu RSA:

```
gpg --full-generate-key
```

- wybór kryptosystemu - w omawianym przykładzie wybrano opcję **(1)**:

```
gpg (GnuPG) 2.2.27; Copyright (C) 2021 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
      (1) RSA and RSA (default)
      (2) DSA and Elgamal
      (3) DSA (sign only)
      (4) RSA (sign only)
      (14) Existing key from card
Your selection?
```

- wybór rozmiaru klucza - wybrano domyślną opcję:

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (3072)
```

- wybór okresu ważności klucza - wybrano domyślną opcję:

```
Please specify how long the key should be valid.
         0 = key does not expire
       <n>  = key expires in n days
       <n>w = key expires in n weeks
       <n>m = key expires in n months
       <n>y = key expires in n years
Key is valid for? (0)
```

- potwierdzenie wcześniejszych wyborów:

```
Key does not expire at all
Is this correct? (y/N)
```

- podanie informacji, kto jest właścicielem klucza, klucz zostaje związany z adresem email właściciela:

```
GnuPG needs to construct a user ID to identify your key.

Real name:
potwierdzenie informacji o właścicielu klucza
Real name: John Brown
Email address: john.brown@mail.com
Comment:
You selected this USER-ID:
    „John Brown <john.brown@mail.com>”

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit?
```

- proces generowania kluczy dla potwierdzonej wcześniej tożsamości:

```
You selected this USER-ID:
    „John Brown <john.brown@mail.com>”

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: /home/kali/.gnupg/trustdb.gpg: trustdb created
gpg: key 828E1DA191F57F18 marked as ultimately trusted
gpg: directory ,/home/kali/.gnupg/openpgp-revocs.d’ created
gpg: revocation certificate stored as ,/home/kali/.gnupg/openpgp-revocs.d/
  578338F03A30F224CBB7E368828E1DA191F57F18.rev’
public and secret key created and signed.

pub   rsa3072 2022-03-13 [SC]
      578338F03A30F224CBB7E368828E1DA191F57F18
uid                      John Brown <john.brown@mail.com>
sub   rsa3072 2022-03-13 [E]
```

Podczas zapisywania kluczy na dysku użytkownik proszony jest o podanie hala. Halo to służy do zaszyfrowania kluczy, po to aby zwiększyć bezpieczeństwo i nie zapisywać ich n dysku w jawnej postaci.

- listę zapisanych kluczy (prywatnych i publicznych) można sprawdzić za pomoca poniższego polecenia:

```
gpg --list-keys
wynik działania komendy:
gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
/home/kali/.gnupg/pubring.kbx
------------------------------
pub   rsa3072 2022-03-13 [SC]
      578338F03A30F224CBB7E368828E1DA191F57F18
uid           [ultimate] John Brown <john.brown@mail.com>
sub   rsa3072 2022-03-13 [E]
```

Wygenerowany klucz posiada unikalny “fingerprint” (unikalną sygnaturę):
578338F03A30F224CBB7E368828E1DA191F57F18
a jej osiem ostatnich cyfr **91F57F18** to tak zwany ID klucza.

2. Import klucza publicznego z serwera na podstawie ID klucza

- polecenie importu klucza na podstawie jego ID

gpg --keyserver keyserver.ubuntu.com -recv E074AAB9

w tym przykładzie wykorzystano serwer kluczy **keyserwer.ubuntu.com**.

Klucz został pomyślnie zaimportowany:

```
gpg: key 4359ED62E074AAB9: public key „Example.Key@example.org>” imported
gpg: Total number processed: 1
gpg:               imported: 1
```

3. Import klucza publicznego z pliku

System GnuPG pozwala również na zaimportowanie klucza publicznego zapisanego w pliku, w formacie (PEM). W tym przypadku klucz publiczny zapisany jest w pliku **public_key.gpg** z wykorzystaniem kodowania **Base64**.

gpg --import public_key.gpg

Poniżej przedstawiono fragment pliku z kluczem publicznym.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Comment: Hostname:
Version: Hockeypuck ~unreleased

xsBNBFLCn/gBCAC8WGhF5Trwalt0PPMFfiwaaYZZMRClrbEh7vO/ip0D9PImdMIG
aR8x26akXDsUgckSDuJiIAWZ0y9uekmMm807wZt74oHlLoqH6NxGMaqKz1pBRtlR
5VdqP5nUukcDfA10/1p6e6bAYE6kep/m6NWubMumTpvJ0OnFLMiMF+uwNpNVBFWR
...
8vTWufNomt5o1D5E+gGZL2NQNE6rMuEYEK2Lgu/pyAFcwVSDa2mLK/cSM7IRDMud
+6JeNq3Bv5jZmqpN8lzm3BqKoN0=
=0Amd
-----END PGP PUBLIC KEY BLOCK-----
```

Eksport klucza publicznego

Aby wyeksportować klucz publiczny (w tym przypadku użytkownika **alice.brown@mail.com**) do pliku **public.key** należy użyć polecenia:

gpg --export alice.brown@mail.com > public.key

Klucz zostanie wyeksportowany w postaci binarnej. Użycie parametru **--armour** pozwala zapisać klucz z wykorzystaniem kodowania Base64. W takiej postaci może on być dołączany np. do korespondencji.

Ze względu na model dystrybucji klucza w systemie GnuPG możliwe jest wyeksportowanie klucza publicznego na serwer kluczy. Należy jednak pamiętać, że klucze wyeksportowane na serwer nie mogą zostań z niego usunięte. Należy zatem czynić to z rozwagą aby nie “zaśmiecać” serwerów kluczy.

Szyfrowanie i deszyfrowanie

W przypadku systemu GnuPG nie ma znaczenia jaki plik będzie szyfrowany, nie istnieje także ograniczenie na rozmiar tego pliku. Istotne jest natomiast określenie odbiorcy (ang. recipient), ponieważ w domyśle to jego klucz publiczny zostanie użyty do szyfrowania. Załóżmy, że chcemy zaszyfrować plik **message.txt** i wysłać go do użytkownika **Example.Key@example.org**:

```
gpg -encrypt -r Example.Key@example.org message.txt
```

Po wykonaniu powyższego polecenia utworzony zostaje plik binarny **message.txt.gpg**, który oprócz zaszyfrowanego pliku wejściowego zawiera dodatkowe informacje o użytym kluczu. Można to sprawdzić wykonując polecenie:

```
file message.txt.gpg
```

Wynika wykonania polecenia:

```
message.txt.gpg: PGP RSA encrypted session key - keyid: 495532A2 22657D58
RSA
(Encrypt or Sign) 3072b .
```

System GnuPG odbiorcy nie będzie miał problemu z wyborem odpowiedniego klucza przy próbie odszyfrowania. Deszyfrowanie po stronie odbiorcy wygląda następująco:

```
gpg --decrypt message.txt.gpg
```

Użytkownik zostanie poproszony o podanie hasła, z pomocą których został zaszyfrowany klucz prywatny. Odszyfrowana zawartość zostanie wyświetlona na ekranie. Aby zapisać ją do pliku należy wskazać go za pomocą parametru **ioutput nazwa_pliku** lub przekierować standardowe wyjście:

```
gpg --decrypt message.txt.gpg > message.txt
```

Jeśli istnieje taka potrzeba, można zaszyfrowane dane zapisać z wykorzystaniem kodowania **Base64** po to aby móc transportować je w plikach tekstowych.
Służy do tego parametr **--armour**:

```
gpg --encrypt --armour -r Example.Key@example.org message.txt
```

Po wykonaniu powyższego polecenia zostanie utworzony plik **message.txt.asc**. Podczas odszyfrowywania także należy użyć parametru **--armour**.

Podpisywanie plików i weryfikowanie podpisu

Jak już wspomniano wcześniej, pliki podpisujemy swoim kluczem prywatnym natomiast do weryfikacji wykorzystywany jest sprzężony z nim klucz publiczny. Oznacz to, że aby zweryfikować poprawność złożonego przez nadawcę podpisu musimy mieć dostęp do jego klucza publicznego.

W systemie GnuPG mamy do dyspozycji trzy sposoby na złożenie podpisu:

1. Podpis i skompresowany, podpisywany plik umieszczane są w jednym pliku:
   `gpg --sign message.txt`

   W tym przypadku tworzony jest jeden binarny plik **`message.txt.gpg`**. Podczas składania podpisu zostaniemy poproszeni o podanie hasła dostępu do klucza prywatnego ustawionego jako klucz domyślny. Weryfikacja za pomocą polecenia
   `gpg --verify message.txt.gpg`

   powoduje wyświetlenie komunikatu podobnego do poniższego:
   ```
   gpg: Signature made Thu 17 Mar 2022 06:02:15 AM EDT
   gpg:                using RSA key FE6241844316F1C4F32656C4B91412B8E57B0AA5
   gpg:Good signature from „Alice Brown<alice.brown@mail.com>”[ultimate]
   ```

   Dostęp do podpisywanego pliku można uzyskać używając zamiast parametru **`--verify`** parametru **`--decrypt`**. Zostanie wtedy wyświetlona zawartość podpisywanego pliku oraz informacje o podpisie.

2. Podpis dołączony do nieskompresowanego pliku:
   `gpg --clearsign message.txt`

   Tworzony jest plik **`message.txt.asc`** zawierający nieskompresowany plik **`message.txt`**wraz z dołączonym podpisem zakodowanym do formatu **`Base64`**. Metoda ta nadaje się do podpisywania plików tekstowych. Weryfikacja przebiega podobnie jak poprzednio.

3. Podpis odłączony od podpisywanego pliku:
   `gpg --detach-sign message.txt`

   Tworzony jest plik message.txt.sig zawierający binarny podpis związany z plikiem **`message.txt`**, który pozostaje niezmieniony w żaden sposób. Podczas weryfikacji konieczny jest dostęp do pliku, którego dotyczy podpis. Informacja o tym pojawia się w komunikacie przy próbie weryfikacji:
   `gpg --verify message.txt.sig`

   Wynik procesu weryfikacji podpisu:
   ```
   gpg: assuming signed data in ‚message.txt’
   gpg: Signature made Thu 17 Mar 2022 02:00:05 PM EDT
   gpg:                using RSA key FE6241844316F1C4F32656C4B91412B8E57B0AA5
   gpg:Good signature from „Alice Brown<alice.brown@mail.com>”[ultimate]
   ```

   Jeśli podczas generowania podpisu dodamy parametr **`--armour`** wygenerowany podpis zostanie zapisany w pliku **`message.txt.asc`** z wykorzystaniem kodowania **`Base64`**.