Versleuteling van een reeks bytes (2)

bijvoorbeeld een tekst of een heel bestand
programmering in FreeBASIC (en Python)

Hans Lunsing

Inleiding

Vorige maal beschreef ik een algoritme waarmee een tekst of een heel bestand kan worden versleuteld, en natuurlijk ook weer ontsleuteld. Nu laat ik zien hoe dit algoritme in FreeBASIC kan worden geprogrammeerd. FreeBASIC is een moderne, veelzijdige BASIC-programmeertaal. Zijn compiler, fbc (van FreeBASIC Compiler), wordt uitgebracht onder een open source licentie, de GPL versie 2, en is dus gratis te gebruiken. Hij is beschikbaar voor zowel Linux als Windows, en kan worden gedownload van:
https://sourceforge.net/projects/fbc/files/
De nieuwste versie is 1.09.0. Uitgebreide documentatie is beschikbaar, en het pakket biedt veel voorbeelden.

Ook al wordt in dit artikel de programmering in FreeBASIC uiteengezet, ik heb me niet beperkt tot FreeBASIC. Het algoritme en bijbehorende testprogramma's heb ik ook geprogrammeerd in de veel gebruikte programmeertaal Python (versie 3).
De programmering loopt langs dezelfde lijnen als in FreeBASIC.

Alle broncode, zowel in FreeBASIC als in Python, en de gecompileerde FreeBASIC programma's, zowel voor Linux als voor Windows, staan voor u ter download klaar op de website van CompUsers. Aan het slot van dit artikel vindt u de precieze downloadlocaties.

Misschien is het een aardig idee om eerst zelf te proberen het algoritme te programmeren, en pas daarna te kijken hoe ik het heb gedaan. Dat is vaak de beste methode om het programmeren onder de knie te krijgen.

De Caesar-versleuteling

We beginnen onze programmering met de Caesar-versleuteling. De Caesar-versleuteling van een byte kunnen we in één regel programmeren[1]:

```
Content[i] = (Content[i] + Taak * Shift - FirstByte + _
             NumBytes) mod NumBytes + FirstByte
```

De hierin gebruikte variabelen zijn in de eerste plaats Content, een string, dat is een reeks bytes, waarvan Content[i] de (i + 1)ste byte is (de telling begint namelijk bij 0). Content[i] is de byte die versleuteld dan wel ontsleuteld wordt. Alle andere in deze regel voorkomende variabelen zijn integers, dat zijn gehele getallen. Ze hebben de volgende betekenis:

- Taak: versleuteling (waarde 1) dan wel ontsleuteling (waarde -1).
- Shift: de sleutel, dat is de bijtelling bij de bytecode. Hij wordt bijgeteld bij versleuteling (Taak is dan 1) en afgetrokken bij ontsleuteling (Taak is dan -1).
- FirstByte: de bytecode van de eerste voor versleuteling in aanmerking komende byte. Als bijvoorbeeld alle mogelijke bytes in aanmerking komen is zijn waarde 0, als we ons beperken tot de afdrukbare bytes is zijn waarde 32 (staat voor de spatie).
- LastByte: de bytecode van de laatste voor versleuteling in aanmerking komende byte. Als bijvoorbeeld alle mogelijke bytes in aanmerking komen is zijn waarde 255, als we ons beperken tot de afdrukbare bytes is zijn waarde 126 (staat voor de tilde).
- NumBytes: het aantal voor versleuteling in aanmerking komende bytes. Zijn waarde is LastByte – FirstByte + 1.

We kunnen deze variabelen als volgt declareren (naam en type aan FreeBASIC bekendmaken):

```
dim as string Content
dim as integer i, Shift, Taak, FirstByte, LastByte, NumBytes
```

Om gemakkelijker te kunnen onthouden wat de twee mogelijke waarden van Taak inhouden, definiëren we twee constanten:

```
const ENCRYPT = 1, DECRYPT = -1
```

Aan Taak kunnen we nu de waarde ENCRYPT dan wel DECRYPT toekennen, waarmee onmiddellijk duidelijk wordt wat de bedoeling is.

Wat gebeurt er in deze programmaregel? Wel, dat is wat ik de vorige maal aan de hand van een voorbeeld heb uitgelegd. We verminderen de uitkomst van de bijtelling dan wel aftrekking (Taak) van de sleutel (Shift) met de kleinste byte van de reeks (FirstByte) en tellen er het aantal in aanmerking komende bytes (NumBytes) bij op om te voorkomen dat het resultaat bij ontsleutelen negatief wordt. Dat mag omdat we toch met NumBytes als modulus werken. Dan berekenen we het resultaat modulo NumBytes zodat we binnen de in aanmerking komende bytereeks blijven, en tellen ten slotte de kleinste byte van de reeks (FirstByte) er weer bij op.

Alleen de NumBytes bytes in de reeks [FirstByte … LastByte] worden versleuteld dan wel ontsleuteld. Alle bytes daarbuiten blijven zoals ze zijn. Dat betekent dat we voor versleuteling dan wel ontsleuteling eerst moeten testen of de byte in kwestie wel in aanmerking komt. Dat doen we zo[2]:

```
if Content[i] >= FirstByte and Content[i] <= LastByte then
  Content[i] = (Content[i] + Taak * Shift - FirstByte + _
       NumBytes) mod NumBytes + FirstByte
end if
```

Tot dusver hebben we ons nog maar met één byte, Content[i], bezig gehouden, maar alle bytes van de string Content moeten zo worden behandeld. Dat doen we door ze in een lus (for i = … next i) allemaal bij langs te lopen:

```
for i = 0 to len(Content)-1
  if Content[i] >= FirstByte and Content[i] <= LastByte then
    Content[i] = (Content[i] + Taak * Shift - FirstByte + _
        NumBytes) mod NumBytes + FirstByte
  end if
next i
```

len() is een FreeBASIC functie die de lengte van het string-argument bepaalt. len(Content) geeft dus het aantal bytes in Content weer. Omdat de telling van de bytes met 0 begint is het nummer van de laatste byte len(Content) – 1.

Bijtelling met pseudo-toevalsgetallen

Nu we de Caesar-versleuteling hebben geprogrammeerd gaan we over op bijtelling met een reeks pseudo-toevalsgetallen in plaats van een vast getal (Shift) zoals bij de Caesar-versleuteling.

Die reeks pseudo-toevalsgetallen slaan we op in een eendimensionaal array met de naam LenShift. De lengte van de reeks kunnen we vrij kiezen, mits niet groter dan het aantal mogelijke bytes, dat is 256. Met 0 als nummer van het eerste toevalsgetal is 255 dus het hoogst mogelijke nummer. We zullen het hoogste nummer LastShift noemen. Het is 1 minder dan het aantal toevalsgetallen. We declareren beide variabelen, LenShift en LastShift, als volgt:

```
dim as integer LastShift = 255
dim as integer LenShift(0 to LastShift)
```

LastShift is een variabele, en daarmee is LenShift een zgn. dynamisch array. LastShift moet natuurlijk een waarde hebben voordat LenShift op deze manier kan worden gedimensioneerd. Daarom is aan LastShift in zijn declaratie een beginwaarde toegekend, in dit voorbeeld 255. Mocht aan LastShift later een andere waarde worden toegekend, dan moet LenShift als volgt opnieuw worden gedimensioneerd:

```
redim as integer LenShift(0 to LastShift)
```

dus nu met *redim* in plaats van *dim*.

Versleuteling en ontsleuteling van een byte (i) van de te versleutelen reeks bytes (Content) gaat nu als volgt:

```
Content[i] = (Content[i] + Taak * LenShift(i _
        mod (LastShift + 1)) - FirstByte + NumBytes) _
        mod NumBytes + FirstByte
```

Als nummer van het te gebruiken toevalsgetal uit LenShift nemen we gewoon het nummer van de te versleutelen byte uit Content, maar dan wel modulo het aantal toevalsgetallen, LastShift + 1. Stel dat LastShift 6 is, dan komt dit erop neer dat we vanaf byte nummer 7 weer opnieuw met de reeks toevalsgetallen beginnen, zoals de vorige maal in een voorbeeld is uitgelegd.

Voor de productie van (pseudo-)toevalsgetallen beschikt FreeBASIC over een toevalsgenerator, die daartoe over verschillende algoritmes beschikt. Zo'n algoritme heeft voor zijn werk een startwaarde (in het Engels: *seed*) nodig. Voor een gegeven startwaarde produceert de toevalsgenerator steeds dezelfde reeks getallen. Die startwaarde is dus de sleutel waarmee een reeks bytes kan worden versleuteld en ook weer worden ontsleuteld. Zowel de startwaarde als het te gebruiken algoritme wordt vastgelegd met het commando *randomize*:

```
randomize [startwaarde][, algoritme]
```

'startwaarde' is een getal van het type *double*, dat is een 64-bit drijvende-komma-getal (*floating point*).

Als geen startwaarde wordt opgegeven baseert FreeBASIC de startwaarde op het resultaat van de *Timer* functie. *Timer* geeft het aantal seconden (tot op 6 decimalen) dat sinds een bepaald referentietijdstip is verstreken. Zo'n à priori onbekende en steeds veranderende startwaarde is voor ons echter niet bruikbaar.
'algoritme' is het nummer van een van de beschikbare algoritmes. Als geen nummer wordt opgegeven kiest FreeBASIC de Mersenne-twister (nummer 3) die een hoge graad van willekeurigheid (*randomness*) heeft en daarom voor ons de voorkeur heeft. Dit is ook het algoritme dat in Python wordt gebruikt[3].

Wij initialiseren de toevalsgenerator met een gegeven startwaarde (Sleutel) en een vast algoritme:

```
randomize cdbl(Sleutel), 3
```

Zo laten we niets aan het toeval over (!). Sleutel is de startwaarde. Hij wordt als volgt gedeclareerd:

```
dim as longint Sleutel
```

Een *longint* is een 64-bit geheel getal met teken. We gebruiken alleen de positieve getallen. Het grootste getal dat aan een *longint* kan worden toegekend is 9.223.372.036.854.775.807, ruim 9 triljoen. Dat is voldoende groot om niet gemakkelijk te achterhalen wat de sleutel is. Zoals we zagen wil *randomize* zijn startwaarde krijgen als type *double*. Daarom wordt de *longint* Sleutel met de functie *cdbl* omgezet naar type *double*.
De tweede parameter, 3, is het nummer van het gewenste algoritme, de Mersenne-twister.

Nu de toevalsgenerator is geïnitialiseerd kan de reeks (pseudo-)toevalsgetallen worden gegenereerd. Dat doen we door het achtereenvolgens aanroepen van de functie *rnd*. Deze functie genereert een getal met dubbele precisie (*double*) in het bereik [0, 1), dus met inbegrip van 0 en zonder 1. Om daarvan het gewenste gehele getal (*integer*) in het bereik [0, NumBytes) te maken vermenigvuldigen we het resultaat met Numbytes en converteren we het met de functie *cint* naar een geheel getal. Dat gaat zo:

```
LenShift(i) = cint(NumBytes * rnd)
```

Het resultaat wordt toegekend aan nummer *i* van de reeks met toevalsgetallen LenShift. Om de volledige reeks met toevalsgetallen te vullen voeren we deze opdracht uit in een lus over alle (LastShift + 1) nummers:

```
for i = 0 to LastShift
  LenShift(i) = cint(NumBytes * rnd)
next i
```

Dit alles moet natuurlijk gebeuren voorafgaand aan de versleuteling dan wel ontsleuteling omdat er daarin gebruik van wordt gemaakt.

Nummers pseudo-toevalsgetallen ook door toeval bepaald

Om het voor eventuele hackers nog wat moeilijker te maken, gebruiken we de reeks toevalsgetallen niet achtereenvolgens, maar laten we het voor elke byte te gebruiken toevalsgetal bepalen door een tweede reeks toevalsgetallen. Deze tweede reeks (NumShift) bevat nummers van de te gebruiken toevalsgetallen voor bijtelling in de eerste reeks (LenShift) en is dus even lang als die reeks. De nummers zelf vallen in het bereik [0, LastShift). NumShift wordt als volgt gedeclareerd:

```
dim as integer NumShift(0 to LastShift)
```

Net als LenShift is het een dynamisch array. Om deze tweede reeks in het gewenste bereik [0, LastShift) te vullen gaan we gewoon verder met het genereren van toevalsgetallen met de functie *rnd* zonder een nieuwe startwaarde voor de toevalsgenerator in te voeren. Dus zo:

```
for i = 0 to LastShift
  NumShift(i) = cint(LastShift * rnd)
next i
```

De formule voor versleuteling en ontsleuteling gaat dan over in:

```
Content[i] = (Content[i] + Taak * LenShift(NumShift(i _
          mod (LastShift + 1))) - FirstByte + NumBytes) _
          mod NumBytes + FirstByte
```

Hiermee wordt de volledige versleutelingslus, met test of de byte in kwestie voor versleuteling in aanmerking komt, als volgt:

```
for i = 0 to len(Content) -1
  if Content[i] >= FirstByte and Content[i] <= LastByte then
    Content[i] = (Content[i] + Taak * LenShift(NumShift(i _
          mod (LastShift + 1))) - FirstByte + NumBytes) _
          mod NumBytes + FirstByte
  end if
next i
```

Om de encryptie nog krachtiger te maken zou voor deze tweede reeks toevalsgetallen een aparte sleutel kunnen worden gebruikt. Dan moet de toevalsgenerator weer worden geïnitialiseerd voordat deze tweede verzameling toevalsgetallen wordt gegenereerd, op deze manier:

```
randomize cdbl(Sleutel2), 3
```

waarin Sleutel2 de tweede sleutel is. Voor ontsleuteling zijn dan altijd twee sleutels nodig.

Naar een module

Alle hiervoor aan de orde gekomen formules en bijbehorende declaraties zijn ingebed in een module, genaamd ‘versleuteling.bas’, die na compilatie als library door FreeBASIC-programma’s kan worden gebruikt. Daarvoor is het volgende gedaan:

- De versleuteling en ontsleuteling van een reeks bytes (*string*) is opgenomen in een subroutine. Deze maakt gebruik van een andere subroutine voor het genereren van de benodigde toevalsgetallen, die buiten de module niet zichtbaar (privé) is.
- Een subroutine voor het gemakkelijk versleutelen en ontsleutelen van een bestand is toegevoegd.
- Er is ook een subroutine toegevoegd om de uitgangswaarden (FirstByte, etc.) naar eigen keuze in te kunnen stellen.
- Er is een extra sleutel gedefinieerd voor de tweede reeks toevalsgetallen (*NumShift*). Wil men met 1 sleutel werken dan stelt men de tweede sleutel gewoon gelijk aan de eerste.

De module kan zowel als statische als dynamische[4] library worden gecompileerd. Op dynamische libraries gaan we hier niet verder in. Als statische library krijgt krijgt de module zowel in Linux als in Windows de naam ‘libversleuteling.a’[5] . Een programma kan er gebruik van maken door in de programmacode de opdracht voor het insluiten van een library:

```
#inclib "versleuteling"
```

op te nemen. Let wel dat het voorvoegsel ‘lib’ en de extensie ‘.a’ hier niet worden opgegeven. FreeBASIC voegt ze automatisch toe. Bij zo’n library hoort een zgn. insluitbestand (*include file*), in dit geval genaamd ‘versleuteling.bi’ dat de definities en declaraties bevat die nodig zijn om van de library gebruik te kunnen maken. Met het commando

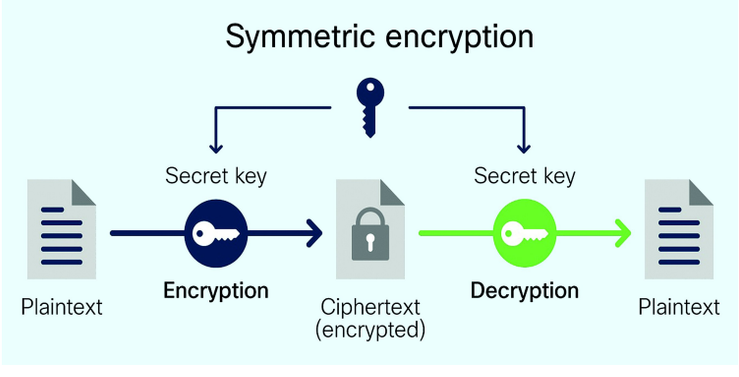
```
#include "versleuteling.bi"
```

worden deze in het programma opgenomen. Het gaat om de volgende definities en declaraties:

```
declare sub initcrypt(Firstbyte as integer, _
        Lastbyte as integer, Lastshift as integer)
declare sub docrypt (Taak as integer, Content as string, _
        Sleutel1 as longint, Sleutel2 as longint)
declare function doCryptFile(Taak as integer, _
        FilenameSrc as string, FilenameTgt as string, _
        Sleutel1 as longint, Sleutel2 as longint) as integer

const ENCRYPT = 1
const DECRYPT = -1
```

Zoals we al zagen zijn de constanten ENCRYPT en DECRYPT handig om op een duidelijke manier aan te geven wat er moet gebeuren.



De twee subroutines en de functie hebben de volgende doelen:

- *initcrypt*: gewenste instelling van de uitgangswaarden;
- *docrypt*: versleuteling dan wel ontsleuteling van een reeks bytes (de *string Content*) met de sleutels *Sleutel1* en *Sleutel2*. Beide sleutels kunnen gelijk zijn.
- *doCryptFile*: versleuteling dan wel ontsleuteling van een heel bestand met dito sleutels. *FilenameSrc* is het te behandelen bestand, en *FilenameTgt* is het resulterende bestand.

Let wel dat het ‘include’ commando, waarmee de definities en declaraties van het insluitbestand in het programma worden opgenomen, vooraf moet gaan aan het eerste gebruik dat van die definities of declaraties wordt gemaakt. Anders kent FreeBASIC ze nog niet.

Tot slot

Voor zowel Linux als Windows heb ik pakketten samengesteld met

- de bronbestanden van de module, het bij de module behorende insluitbestand, een testprogramma voor het ver-/ontsleutelen van een tekst, en een testprogramma voor het ver-/ontsleutelen van een bestand.
- de gecompileerde module als statische library, en de gecompileerde uitvoerbare programma's.

Mocht u belangstelling hebben dan kunt u, als abonnee van de SoftwareBus, deze pakketten van de website van CompUsers downloaden (wel eerst even inloggen!):

Ze werken op dezelfde manier als de FreeBASIC-programma's.

Voor vragen en opmerkingen kunt u contact met me opnemen via mijn contactpagina op de website (na inlog):
https://www.compusers.nl/user/15/contact

Ik wens u veel genoegen met programmeren in FreeBASIC en Python!

- *Broncode:*
  https://www.compusers.nl/system/files/programmeren/versleuteling_freebasic_broncode.7z

  en (met dezelfde inhoud):
  https://www.compusers.nl/system/files/programmeren/versleuteling_freebasic_broncode.tar.gz

- *Library en testprogramma's voor Windows:*
  https://www.compusers.nl/system/files/programmeren/versleuteling_freebasic_windows.7z

- *Library en testprogramma's voor Linux:*
  https://www.compusers.nl/system/files/programmeren/versleuteling_freebasic_linux.tar.gz

Bovendien heb ik een module met het versleutelingsalgoritme en bijbehorende testprogramma's samengesteld in de programmeertaal Python (versie 3). Ook deze zijn voor belangstellenden te downloaden:

- *Voor Windows:*
  https://www.compusers.nl/system/files/programmeren/versleuteling_python.7z

- *Voor Linux (met vrijwel dezlfde inhoud):*
  https://www.compusers.nl/system/files/programmeren/versleuteling_python.tar.gz

Links:

1 Met het teken _ (onderstreping) wordt aangegeven dat de programmaregel op de volgende regel wordt vervolgd.
2 Het teken >= betekent: groter dan of gelijk aan, en <= betekent: kleiner dan of gelijk aan.
3 Met bij dezelfde startwaarde wel andere resultaten dan in FreeBASIC.
4 .dll (dynamic link library) in Windows, en .so (shared object) in Linux.
5 De 'a' staat voor 'archive'