

# MICT1

---

## EXERCISE WEEK 6

Joeri van Grimbergen (1244825) | Nursize Bilen (1260235) | Rick van Gorp (1328417)  
ZUYD HOGESCHOOL | HEERLEN | GROEP 4

## Inhoud

1. Voorbereiding van analyse .....	2
Doel van de opdracht .....	2
Tools .....	2
2. Analyse van file6.....	3
PK Validatie.....	3
3. Analyse van x2 .....	7
RAR validatie.....	7
4. Analyse van x .....	9
5. Python scripts .....	11
Headerfinder.py .....	11
ByteSwap.py .....	13
PNGAnomalyFinder.py .....	14
6. Bronnen .....	16

## 1. Voorbereiding van analyse

Er is onderzoek uitgevoerd naar het in week 6 geleverde bestand "file6". Om dit bestand te analyseren zijn XVI32 en HeaderFinder.py gebruikt. XVI32 is een hex-editor die in staat is om binary data in te lezen en eventueel waar nodig aan te passen. HeaderFinder.py is een Python-script dat is gemaakt door Groep 4 in week 4. Dit script is in staat om de hexadecimale waarden van bekende file signatures te zoeken in bestanden die in het script worden opgegeven.

Het script van Headerfinder.py heeft een update gehad om ook meer bestanden tegelijk te kunnen verwerken. De output wordt geleverd in een output file (\*.txt) per input file, omdat de output in de Python shell anders niet langer overzichtelijk zou zijn.

### Doel van de opdracht

Om de analyse uit te kunnen voeren moet het doel van de opdracht bekend zijn. Het doel van de opdracht is om een bericht te herstellen die zich in het bestand "file6" bevindt. Daarnaast zouden er mogelijk bestanden kunnen staan in de *file*- of *diskslack* mocht er sprake zijn van een filesystem.

### Tools

De tools die zijn gebruikt voor dit onderzoek zijn onderstaand weergegeven:

- XVI32: <http://www.handshake.de/user/chmaas/delphi/download/xvi32.zip>;
- HeaderFinder.py: Zie Github Pullrequest van week 6;
- PNGAnomalyFinder.py: Zie Github Pullrequest van week 6;
- ByteSwap.py: Zie Github Pullrequest van week 6.

## 2. Analyse van file6

Het bestand "file6" is geanalyseerd met het script *HeaderFinder.py* om te zoeken naar bekende file signatures die in het bestand zijn opgenomen. Dit heeft de volgende resultaten opgeleverd:

Value	Location
b'o<'	0x1d13d
b'o<'	0x218c4
b'o<'	0x2675e
b'o<'	0x2caf7
b'o<'	0x3803a
b'o<'	0x94042
b'o<'	0xb194d
b'o<'	0xc2690
b'o<'	0xc413b
b'o<'	0xcba22
b'PK\x05\x06'	0xdbf48

De eerste 10 waardes die worden aangegeven zijn meestal false positives, omdat ze bestaan uit twee bytes: 6F 3C. Deze signature kan gebruikt worden bij een SMS text. Er zijn meerdere bestanden die ook data bevatten, waarbij stukken van de data gelijk zijn aan 6F 3C. Naar aanleiding van dit argument wordt eerst gekeken naar de gevonden PK-header op offset 0xdbf48.

b	\	+	z	0	s	P	Å	1	í	+	!	í	í	0	*	!	Ö	S	;	\	l	¶	g	„	“	e	•	ú	Å	á	u	0	5	0	é	w	[	ù	ö	4	}	<	œ	C		-	M	Ä	œ	2	0	a	4	-	i
f	0	j	0	"	/	P	H	ä	-	E	+	Ý	#	Ö	~	-	+	Q	?	w	x	¶	y	g	ö	Ö	n	-	u	>	Ö	ÿ	"	³	J		œ	í	f	0	í	:	Ê	Š	ö	ä	·	0	U	„	O	-	0	ä	
í	0	>	M	ž	[	s	"	„	Z	E	0	Ä	/	-	l	;	{	ö	‘	í	E	f	j	•	«	=	‘	Ó	f	é	„	0	ö	a	÷	-	í	ú	;	Ü	W	B	»	1	6	Ä	\	~	b	"	W	Y	Ä		
Ë	x	d	j	b	-	'	v	ÿ	Ä	Ä	=	{	@	0	P	K	0	?	0	0							Ü	»	t	H	"	E	ä	ä	ø	¶	ø	¶											€	'					
				x	2	P	K	0	0					0	0																																								

Aan het eind van het bestand staan 2 PK-headers. Dit kan erop duiden dat *file6* een bestand is met een PK-header (misschien ZIP-bestand). Om dit te valideren wordt aan de hand van PK-specificatie geanalyseerd of het einde van de file correct is.

### PK Validatie

De PK-headers die het betreft zijn vermoedelijk gelijk aan de Central Directory Header en de End of Central Directory Record. Onderstaand worden de resultaten van de controle vanaf offset 0xdbf18 weergegeven. Deze moeten gelezen worden als little endian, maar hebben het big endian formaat genoteerd:

### **Central Directory Structure**

*Central File Header Signature: 4 Bytes -> 504B0102*

*Version: 2 Bytes -> 3F 03*

*Version to Extract: 2 Bytes -> 0A 03*

*Bit Flag: 2 Bytes -> 00 00*

*Compression Method: 2 Bytes -> 00 00 (Stored)*

*Last Mod File Time: 2 Bytes -> DC BB*

*Last Mod file Date: 2 Bytes -> 74 48*

*CRC-32: 4 Bytes -> 22 45 E0 E2*

*Compressed Size: 4 Bytes -> F8 BE 0D 00*

*Uncompressed Size: 4 Bytes -> F8 BE 0D 00*

*File name length: 2 Bytes -> 02 00*

*Extra Field Length: 2 Bytes -> 00 00*

*File comment length: 2 Bytes -> 00 00*

*Disk number start: 2 Bytes -> 00 00*

*Internal file attributes: 2 Bytes -> 00 00*

*External file attributes: 4 Bytes -> 20 80 B4 81*

*Relative Offset: 4 Bytes -> 00 00 00 00*

*File name: [File name Length] (2 Bytes) -> x2 (78 32)*

**END OF Central Directory Header**

### **End of Central Directory Record**

*End of Central dir signature: 4 Bytes -> 50 4B 05 06*

*Number of this disk: 2 Bytes -> 00 00*

*Number of disk with start central dir: 2 Bytes -> 00 00*

*Total number of entries: 2 Bytes -> 01 00*

*Total number of entries2: 2 Bytes -> 01 00*

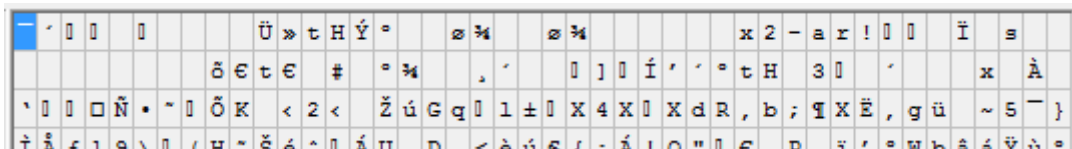
*Size of Central dir: 2 Bytes -> 30 00*

*Offset start central dir: 4 Bytes -> 18 BF 0D 00*

*ZIP comment length: 2 Bytes -> 00 00*

**END OF End of central directory Record**

De lengtes van de secties komen overeen met de structuren van de Central Directory Record uit de PK-file specificatie. Uit bovenstaande gegevens blijkt ook dat de Compression Method gelijk is aan 00 00. Dit houdt in dat het bestand "x2" ongecomprimeerd opgeslagen is in het ZIP-bestand. Het ZIP-bestand hoeft niet hersteld te worden, maar het begin van de data die in het ZIP-bestand staat moet gevonden worden. Om te bevestigen dat de instellingen uit de central directory record gelijk zijn aan de instellingen uit de PK-header, wordt gezocht naar de oorspronkelijke PK-header. Deze PK-header bevindt zich vermoedelijk aan het begin van het bestand.



In bovenstaande afbeelding is de vermoedelijke PK file header weergegeven. Een indicatie dat dit de file header is, is de bestandsnaam x2 die hier wederom wordt weergegeven. De PK file header is dus aangepast. Vanaf het begin van het bestand wordt gecontroleerd of het formaat voldoet aan de PK specificatie (lezen als little endian, weergegeven als big endian):

#### Local file header

Local File Header Signature: 4 Bytes -> AF B4 03 04 (Zou moeten zijn 50 4B 03 04)

Version to extract: 2 Bytes -> 0A 03

Bit flag: 2 Bytes -> 00 00

Compression Method: 2 Bytes -> 00 00

Last Mod File Time: 2 Bytes -> DC BB

Last Mod File Date: 2 Bytes -> 74 48

CRC32: 4 Bytes -> DD BA 1F 1D

Compressed Size: 4 Bytes -> F8 BE 0D 00

Uncompressed Size: 4 Bytes -> F8 BE 0D 00

File name length: 2 Bytes -> 02 00

Extra Field Length: 2 Bytes -> 00 00

Filename: [File name Length] = 2 Bytes -> x2

**END OF Local file header**

De lengte en velden komen overeen met de structuur van een PK Local file header, echter zijn er enkele waarden aangepast:

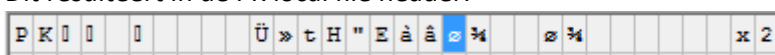
- Local File Header Signature: 50 4B veranderd naar AF B4;
- CRC32: Komt niet overeen met de CRC-waarde uit de Central Directory Structure.

Het ZIP-bestand wordt valide gemaakt door de volgende wijzigingen door te voeren:

#### Wijziging van waarden

Begin (Offset hex)	Eind (Offset hex)	Lengte (Dec)	Oude waarde	Nieuwe waarde
0x0	0x2	2	AF B4	50 4B
0xE	0x12	4	DD BA 1F 1D	22 45 E0 E2


Dit resulteert in de PK local file header:



Door deze aanpassing is het ZIP-bestand valide gemaakt, want:

- De CRC-32 waarden uit de local file header en central directory komen overeen;
- De File signature zoals gedefinieerd door PKWare Inc. klopt weer.

Het ZIP-bestand kan nu uitgepakt worden, het resultaat is een bestand genaamd x2:

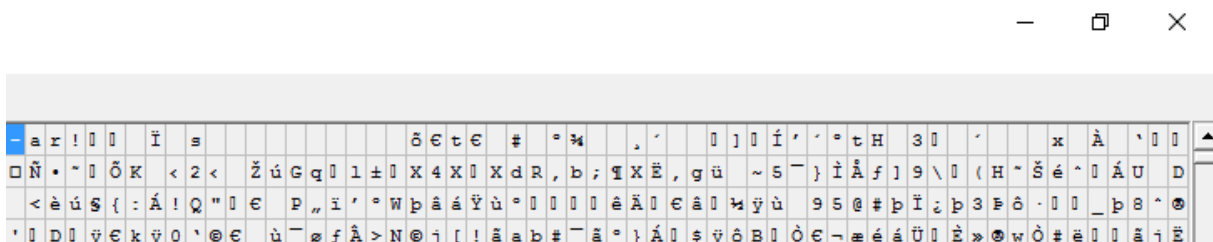
 x2	20-3-2016 23:30	Bestand	880 kB
--	-----------------	---------	--------

### 3. Analyse van x2

Er is wederom gebruik gemaakt van HeaderFinder.py om in het bestand x2 bekende file signatures te vinden. De output die werd gegenereerd door HeaderFinder.py is de volgende:

Value	Location
b'o<'	0x1d11d
b'o<'	0x218a4
b'o<'	0x2673e
b'o<'	0x2cad7
b'o<'	0x3801a
b'o<'	0x94022
b'o<'	0xb192d
b'o<'	0xc2670
b'o<'	0xc411b
b'o<'	0xcba02

Hier valt op dat de waardes die worden weergegeven een soortgelijke offset bevatten als bij de analyse van *file6*. Dit is te verklaren aan de hand van de compression method die gedefinieerd stond in de PK local file header. Deze was gelijk aan *00 00*, wat inhoudt dat het bestand ongecomprimeerd is opgeslagen in het PK bestand. Het resultaat is dus gelijk aan de data die in het PK-bestand is te zien zonder de PK kenmerkende headers.



De header van het uitgedakte bestand x2 is in bovenstaande afbeelding weergegeven. Hier valt op dat de file signature die gebruikt wordt *-ar!* lijkt op de *Rar!* file signature. De hypothese die gesteld wordt voor de analyse naar bestand x2:

- Bestand x2 is een bestand, gebaseerd op het RAR-formaat.

#### RAR validatie

Om dit te verifiëren is gebruik gemaakt van de RAR file format specificatie. De header wordt gecontroleerd en gevalideerd, alsof er sprake zou zijn van de *Rar!* file signature. De resultaten zijn op de volgende pagina weergegeven.



52 61 72 21 1A 07 00 -> 7 Bytes Magic Nr -> Zou het moeten zijn, in bestand x2 is dit: AD 61 72 21 1A 07 00

*Block 1 (Archive Header):*

HEAD\_CRC: 2 Bytes -> CF 90

HEAD\_TYPE: 1 Byte -> 73

HEAD\_FLAGS: 2 Bytes -> 00 00

HEAD\_SIZE: 2 Bytes -> 0D 00 -> 13 bytes

(ADD\_SIZE: 4 Bytes -> 00 00 00 00)

6 Bytes: 00

*Block 2 (File Header):*

HEAD\_CRC: 2 Bytes -> F5 80

HEAD TYPE: 1 Byte -> 74

**HEAD FLAGS:** 2 Bytes -> 80 90

HEAD\_SIZE: 2 Bytes -> 23 00

*PACK\_SIZE: 4 Bytes -> BA BE 0D 00 (Compressed)*

*UNP\_SIZE: 4 Bytes -> B8 B4 0D 00 (Uncompressed)*

HOST\_OS: 1 Byte -> 03 -> Unix

FILE CRC: 4 Bytes -> 5D 11 CD 92

FTIME: 4 Bytes -> B4 BA 74 48

*UNP\_VER: 1 Byte -> 1D -> 2.9 (Major + Minor Version)*

**METHOD:** 1 Byte -> 33 -> Normal Compression

*NAME SIZE: 2 Bytes -> 01 00 -> 1*

ATTR: 4 Bytes -> B4 81 00 00

HIGH\_PACK\_SIZE -> N/A

HIGH UNP SIZE -> N/A

*FILE\_NAME* -> 1 Byte -> x

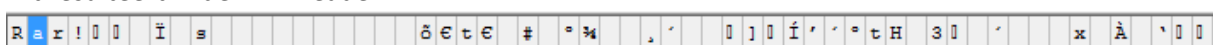
## END OF Header RAR File Structure

Uit bovenstaande resultaten blijkt dat het om een RAR-bestand gaat. Dit bestand is gecomprimeerd omdat de PACK\_SIZE kleiner is dan de UNP\_SIZE. Daarnaast is METHOD 33 gebruikt, wat staat voor Normal Compression. Om het RAR-bestand valide te maken worden de volgende wijzigingen aan het bestand doorgevoerd:

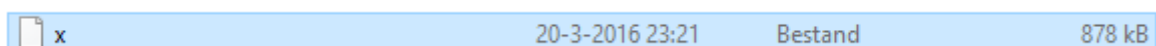
### Wijziging van waardes

Begin (Offset hex)	Eind (Offset hex)	Lengte (Dec)	Oude waarde	Nieuwe waarde
0x0	0x1	1	AD	52

Dit resulteert in de RAR Header:



Het RAR-bestand is nu uit te pakken met het volgende resultaat:



#### 4. Analyse van x

Het bestand x is direct geopend in hex editor XVI-32. De structuur van het bestand komt erg overeen met een PNG-bestand, maar is niet correct.

[illegible]

In bovenstaande afbeelding is het begin van bestand x zichtbaar. De chunk data types komen erg overeen met de chunks uit een PNG-bestand, want:

- $HIRD = IHDR$ ;
- $pYHs = pHYs$ ;
- $ItEM = tIME$ .

De karakters van deze chunk data types zijn onderling omgeruild per groep van 2 bytes:

- Origineel: IH en DR;
- Nieuw: HI en RD.
- Origineel: pH en Ys;
- Nieuw: p YH s.
- Origineel: tl en ME;
- Nieuw: It en EM.

De verandering van PH en Ys naar p YH s is te verklaren doordat het hele bestand binnen groepen van 2 bytes de karakters onderling heeft omgeruild. Dit houdt in dat pHYs op een oneven byte zou hebben gestaan, waardoor het is gewisseld met een andere byte.

Om dit te bewijzen is een Python script gemaakt, genaamd *ByteSwap.py*. Het algoritme dat in dit script gebruikt wordt, wordt uitgelegd in hoofdstuk 5: Python scripts. De uitvoer van dit script is een volledig valide PNG-bestand. Dit bevestigt de hypothese dat bytes binnen een groep van 2 bytes met elkaar worden gewisseld.

Met het script *PNGAnomalyFinder.py* is binnen het PNG-bestand gezocht naar anomalies van chunk data types of lengtes van bestaande chunks. Dit was niet het geval. De afbeelding bevatte echter wel het uiteindelijke bericht: (Zie volgende pagina).



In de visuele weergave van het PNG-bestand staat de tekst: "It's true. All of it". Dit betreft het verborgen bericht.

Er is geen filesystem aangetroffen of soortgelijke structuur, dus zijn er geen bestanden hersteld uit de file/drive-slack.

## 5. Python scripts

In dit hoofdstuk worden de voor deze opdracht ontworpen Python scripts uitgelegd.

### Headerfinder.py

```
import re
import binascii
import os
import sys
```

In bovenstaande afbeeldingen zijn de modules weergegeven die worden gebruikt door het Pythonscript *Headerfinder.py*:

- Module re: Deze wordt gebruikt om op basis van reguliere expressies overeenkomsten te vinden in het geladen bestand;
- Module binascii: Deze wordt gebruikt om de hexadecimale data uit het bestand *allheaders* te converteren naar binaire data, leesbaar voor Python;
- Module os: Deze wordt gebruikt om de bestandsgrootte van een specifiek bestand op te halen;
- Module sys: Deze wordt gebruikt om argumenten die vanuit het systeem worden aangevoerd in het Python script te verwerken.

```
with open("ALLheaders.txt", "r") as f:
    arrHeaders = f.read().splitlines()

for i in range(0, len(arrHeaders)):
    arrHeaders[i] = binascii.unhexlify(arrHeaders[i])
```

In bovenstaande afbeelding is het script weergegeven waar alle hexadecimale waarden van file signatures uit het bestand *allheaders.txt* worden opgehaald. Deze hexadecimale waarden worden geconverteerd naar binary data.

```
arrFiles = []

if len(sys.argv) > 1:
    for i in range(1, len(sys.argv)):
        arrFiles.append(str(sys.argv[i]))
else:
    arrFiles.append(str(input("\nPlease input a valid filename:\n")))
```

In bovenstaande afbeelding wordt een nieuw element van het script *HeaderFinder.py* weergegeven. Dit deel van het script maakt het mogelijk om op basis van argumenten bestanden aan te voeren aan het script of het pad door te geven van een specifiek bestand. Voor deze opdracht is steeds één bestand geanalyseerd, dus is gebruik gemaakt van de handmatige invoerfunctie. De invoer dient bestandsnaam + extensie en eventueel een pad ernaartoe te zijn.

```

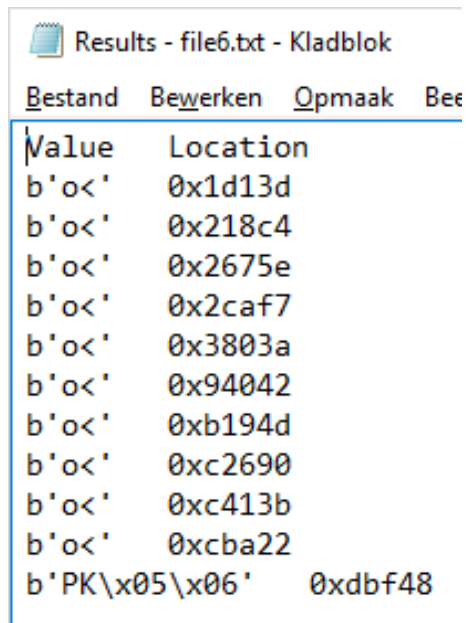
for i in range(0, len(arrFiles)):
    with open(arrFiles[i], "rb") as file:
        intLen = 0
        arrOutPutData = []
        while intLen < os.path.getsize(arrFiles[i]):
            content = file.read(16384) #Block = 16384 bytes
            intLen = intLen + 16384
            for k in range(0, len(arrHeaders)):
                for m in re.finditer(re.compile(re.escape(arrHeaders[k])), content):
                    arrOutPutData.append([hex(m.start()+intLen-16384), arrHeaders[k]])
                    print("\n" + hex(m.start()+intLen-16384))
                    print(arrHeaders[k])
            file.close()
        strOutputFile = "Results - " + arrFiles[i] + ".txt"
        with open(strOutputFile, "w") as fOutPut:
            fOutPut.write("Value\tLocation\n")
            for j in range(0, len(arrOutPutData)):
                fOutPut.write(str(arrOutPutData[j][1]) + "\t" + str(arrOutPutData[j][0]) + "\n")

```

In bovenstaande afbeelding wordt het belangrijkste onderdeel van het script *HeaderFinder.py* weergegeven. De bestanden die zijn opgenomen in array *arrFiles* worden één voor één geopend in binaire modus. Vervolgens wordt per blocksize van 16384 bytes aan de hand van de regex van de binary data file signature gecontroleerd of er een bekende file signature in het bestand staat. Deze blocksize is ingevoerd om te voorkomen dat grote bestanden te veel geheugen gebruiken, waardoor Python crasht.

Als hier sprake van is wordt de betreffende header met hexadecimale offset opgeslagen in array *arrOutPutData*. Vervolgens wordt per bestand in *arrFiles* een TXT-bestand geschreven met de headers en offsets die zijn gevonden.

Voorbeeld van uitvoer:



Value	Location
b'o<'	0x1d13d
b'o<'	0x218c4
b'o<'	0x2675e
b'o<'	0x2caf7
b'o<'	0x3803a
b'o<'	0x94042
b'o<'	0xb194d
b'o<'	0xc2690
b'o<'	0xc413b
b'o<'	0xcba22
b'PK\x05\x06'	0xdbf48

### ByteSwap.py

```
import binascii
import os

file = open("x", "rb")
t = os.path.getsize("x")

file.seek(4)

byteFile = b""
pngSig = binascii.unhexlify("89504e47")
byteFile = byteFile + pngSig

while file.tell() < t:
    try:
        byteFirst = file.read(1)
        byteSecond = file.read(1)
        byteFile = byteFile + byteSecond + byteFirst
    except:
        print("EOF")

f = open("t.png", "wb")
f.write(byteFile)
f.close()
```

In bovenstaande afbeelding is het script *ByteSwap.py* weergegeven. Dit script heeft als uitvoer een werkend PNG-bestand, maar is alleen toepasbaar in deze opdracht of bestanden die volgens dezelfde constructie als dit bestand bewerkt zijn.

In dit script wordt een volledig nieuwe bestand geconstrueerd die de statische header 89 50 4E 47 bevat. Een PNG-bestand heeft altijd deze header. Vervolgens wordt de data die erna komt toegevoegd aan variabele *byteFile*. Eerst worden beide bytes gelezen uit het bestand en worden bij het toevoegen aan variabele *byteFile* omgedraaid, zodat ze omgedraaid in het nieuwe bestand terecht komen. Dit wordt uitgevoerd voor het gehele bestand.

Het resultaat hiervan wordt geplaatst in het in binaire modus geopende bestand *t.png*.

## PNGAnomalyFinder.py

Dit script wordt gebruikt om anomalies in PNG-bestanden te identificeren. Dit script test of Chunk Data Types volgens het PNG-format zijn gedefinieerd en of er anomalies zijn in de lengte van chunks van PNG-bestanden.

```
| from struct import *
```

Het script gebruikt de bibliotheek struct om binaire data van bestanden te converteren naar, bijvoorbeeld, *unsigned integers* en *unsigned chars*. *Signed* is ook mogelijk. De bibliotheek is gebaseerd op *struct* dat in programmeertaal C wordt toegepast.

```
#PNG Header definition
intDefinition = 8 #Length of PNG identifier = 8 bytes

#PNG Chunk definition
intLength = 4      #Length of PNG chunk length identifier = 4 bytes
intChunkType = 4   #Length of PNG chunk type = 4 bytes
intChunkData = 0   #Depends on intLength
intCRC = 4         #Length of CRC = 4 bytes
```

De waarden die zijn toegewezen aan de variabelen zijn gebaseerd op het aantal bytes per chunk sectie zoals gedefinieerd in de PNG specificatie. De lengte van chunk data is gebaseerd op de in het bestand gedefinieerde chunk data length. Hoe deze wordt opgehaald wordt verderop weergegeven.

```
#Function to read file by pre-defined chunks
def ReadChunk(f):
    offset = hex(f.tell()) #Get current location in file
    intLchunk = f.read(intLength)
    intLchunk = unpack('>I', intLchunk)[0] #Read as Big Endian -> Unsigned Integer
    strType = f.read(intChunkType)
    strType = str(strType)[2:len(str(strType))-1]
    strChunkData = f.read(intLchunk)
    intChecksum = f.read(intCRC)
    intChecksum = unpack('>I', intChecksum)[0] #Read as Big Endian -> Unsigned Integer
    return [strType, intLchunk, offset]
```

De functie *ReadChunk* laadt data van een chunk door binaire data te lezen van input bestand *f*. De lengte van deze binaire strings zijn reeds eerder gedefinieerd. De functie bepaalt de huidige offset en leest de data ter grootte van precies één chunk van het PNG-bestand. Zoals eerder genoemd wordt de lengte van de chunk data bepaald door de waarde in variabele *intLchunk*. Voor sommige waarden wordt *struct.unpack* gebruikt. Deze waarden worden gelezen als Big Endian en Unsigned Integers. Dit resulteert in een python variabele met als type integer. Aan het einde van de functie *ReadChunk* worden de waarden *Chunk Type*, *Chunk Length* en *Offset* return gestuurd.

```
#Count most common values to determine general size used per chunk data type
def FindMostCommon(chunkType):
    counter = {}
    for i in range(0,len(arrValues)):
        if arrValues[i][0] == chunkType:
            counter[arrValues[i][1]] = counter.get(arrValues[i][1], 0) + 1
    counts = [(j,i) for i,j in counter.items()]
    if counts:
        intMostCommon = max(counts)[1]
        return(intMostCommon)
```

Functie *FindMostCommon* wordt gebruikt om de meest voorkomende lengte van chunk data per chunk data type te vinden in de input PNG-file. De meest voorkomende waarde wordt return gestuurd als *intMostCommon*.

```
#Print anomalies in length per chunk type
def PrintAnomalyCount(chunkType):
    intMostCommon = FindMostCommon(chunkType)
    for i in range(0, len(arrValues)):
        if arrValues[i][0] == chunkType:
            if arrValues[i][1] != intMostCommon and arrValues[i][1] != 0:
                print("Found anomaly in lengths of: " + arrValues[i][0])
                print("Explanation: It differs from the most used chunk length " + str(intMostCommon))
                print("Length found:" + str(arrValues[i][1]) + "\n")
```

Functie *PrintAnomalyCount* wordt gebruikt om chunk data types weer te geven die een afwijkende lengte dan de meest voorkomende lengte hebben die in de vorige beschreven functie werd bepaald.

```
newfile = open("file.png", 'rb')
magic_val = newfile.read(8)

arrDefaultPNG = [b"IHDR", b"PLTE", b"IDAT", b"IEND", b"tRNS", b"gAMA", b"cHRM", b"sRGB", b"iCCP", b"tEXt", b"zTXc", b"iTXc", b"bRGD", b"pHYs", b"sBIT", b"sPLT", b"hIST", b"tIME"]
```

De code hierboven weergegeven heeft de volgende functies:

- Open het input PNG-bestand in binaire modus;
- Lees de eerste 8 bytes. Deze representeren de magic value van een bestand (file signature);
- Definitie van het array *arrDefaultPNG* die alle mogelijke waarden van chunk data typen bevat. Deze waarden komen uit de PNG-specificatie.

```
#Read complete file until end
blnEOF = False
arrValues = []

while blnEOF == False:
    try:
        arrValues.append(ReadChunk(file))
    except:
        blnEOF = True
```

De hierboven weergegeven *While-Loop* zorgt ervoor dat alle chunks tot het eind van het bestand gelezen worden. De waardes die return worden gestuurd uit *ReadChunk* worden toegevoegd aan het array *arrValues*.

```
for i in range(0, len(arrDefaultPNG)):
    PrintAnomalyCount(arrDefaultPNG[i])
```

Deze *For-Loop* callt functie *PrintAnomalyCount* om per waarde in *arrDefaultPNG* te controleren.

```
#Check if anomaly in Chunk Type exists
for i in range(0, len(arrValues)):
    if ((arrValues[i][0]) not in arrDefaultPNG):
        print("Found anomaly for Chunk Type used in PNG-file:\n")
        print("Chunk Type:\tChunk Length\tChunk Offset")
        print(str(arrValues[i][0]) + "\t\t" + str(arrValues[i][1]) + "\t\t" + str(arrValues[i][2]))
```

Deze *For-Loop* wordt gebruikt om te bepalen of het bestand chunk data types bevat die niet bestaan in de PNG-specificatie. Als er een afwijking wordt gevonden, wordt de chunk data type met offsets en lengtes weergegeven.



## 6. Bronnen

1. ForensicsWiki. (z.j.). RAR. Geraadpleegd van <http://www.forensicswiki.org/wiki/RAR>
2. PKWare Inc.. (2014, 1 september). APPNOTE.TXT - .ZIP File Format Specification. Geraadpleegd van <https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT>
3. PNG (Portable Network Graphics) Specification, Version 1.2. (1999). Retrieved March 16, 2016, from <http://www.libpng.org/pub/png/spec/1.2/PNG-Contents.html>