**Analyse von Dateien und Dateiformaten**

**Fachbereichsarbeit Informatik**

**Abstract**

Diese Arbeit beschäftigt sich mit der Untersuchung von Inhalten von Dateien in Bezug auf ihr Dateiformat, ohne diese in ihr bestimmungsgemäßes Programm zu laden. So soll zum Beispiel anhand des PDF-Standards der Inhalt von PDF-Dateien untersucht werden, ohne diese in ein PDF-Programm zu laden.

Das PDF Format wurde für diese Arbeit gewählt, weil es sehr viele verschiedenartige Informationen enthalten kann und durch das Einbetten von ausführbaren Inhalten (Java-Script) auch Möglichkeiten zum Missbrauch bietet.

Parallel zur theoretischen Betrachtung dieser Problemstellung entsteht während der Arbeit ein Analysewerkzeug, das Dateien, egal welchen Ursprungs, unabhängig von ihrem Dateiformat in binärer Form einliest und eine Analyse erlaubt, ohne Gefahr zu laufen, schadhafte Teile auszuführen.

**Vorwort**

Dateien in gespeicherter Form dienen primär zur Verarbeitung durch Software. Vor allem die Wiedergabe gespeicherter Informationen ist ein Vorgang, der jeden Tag millionenfach stattfindet.

Durch den Wunsch, auch herkömmliche Textdokumente mit zusätzlichen Interaktionen auszustatten, hat man sehr vielen Programmen die Möglichkeit gegeben, Aktionen dynamisch aufgrund des Dateiinhalts durchzuführen, also das Programm nicht nur zum Erstellen und Wiedergeben der gespeicherten Daten zu verwenden, sondern die Möglichkeit einer Interaktion ohne das Zutun des Benutzers zu schaffen.

Was auf der einen Seite eine sehr effiziente Erweiterung der Funktionalität für nützliche Prozesse ist, kann auf der anderen Seite auch erhebliche Gefahren für den Anwender bzw. dessen System darstellen.

Ein Großteil der breit angelegten Angriffe auf Einzelnutzer und der gezielten Angriffe auf Unternehmen geschieht mithilfe sogenannter Scripts, also Anweisungen die in herkömmlichen Dateien versteckt werden und Schadprogramme auf dem Zielsystem installieren oder selbst schadhaft sein können.

Viele solcher Scripts werden durch das Programm nicht direkt dargestellt und das Herausfiltern der fraglichen Teile durch herkömmliche Texteditoren ist nicht zielführend, da diese häufig nicht in Form von darstellbaren Zeichen enthalten sind oder schlichtweg der Umfang der Datei für eine konventionelle Analyse zu groß ist.

Der Ansatz der meisten konventionellen Virenscanner besteht darin, Dateien nach Signaturen bekannter Schadprogramme zu durchsuchen. Was in den meisten Fällen, und vor allem für den Heimanwender, durchaus ausreichend ist, bietet insbesondere im professionellen Bereich bei Weitem zu wenig. Gerade bei „lohnenswerten Zielen“ wie Banken, staatlichen Einrichtungen oder größeren Unternehmen ist die Wahrscheinlichkeit eines gezielten professionellen Angriffes sehr groß. Die Werkzeuge, die hier von Angreifern benutzt werden, sind meist Unikate, welche speziell für diesen Angriff ausgelegt sind oder dahingehend vorbereitet wurde. Da diese Werkzeuge noch unbekannt sind, werden sie von konventionellen Abwehrmechanismen meist nicht erkannt.

Um diesen Bedrohungen Herr zu werden, braucht es genau den Gegenteiligen Ansatz. Man ignoriert während der Analyse alles, was nicht gefährlich sein kann und lenkt seinen Fokus auf jene Inhalte, die grundsätzlich ein Gefahrenpotential beinhalten (z.B. Skripts), unbekannte Inhalte (z.B. nicht zu klassifizierende Anhänge) oder Anomalien (Inhalte die offensichtlich nicht Bestandteil des jeweiligen Dateiformates sind).

Inhaltsangabe

[Grundlagen 5](#_Toc482565524)

[Textdateien 5](#_Toc482565525)

[Binärdateien 5](#_Toc482565526)

[Byte Order 8](#_Toc482565527)

[Containerdateien 8](#_Toc482565528)

[Analysieren mit dem „Analyzer“ 9](#_Toc482565529)

[Eine Sache der Interpretation 10](#_Toc482565530)

[reinterpret\_cast und memcpy 11](#_Toc482565531)

[RGB Farben 11](#_Toc482565532)

[Opcodes 12](#_Toc482565533)

[Dateiformate 16](#_Toc482565534)

[Binärdateien 17](#_Toc482565535)

[Bitmap 17](#_Toc482565536)

[Executables 19](#_Toc482565537)

[Textdateien 24](#_Toc482565538)

[Scripts 27](#_Toc482565539)

[Container-Dateien 28](#_Toc482565540)

[Open Office XML 28](#_Toc482565541)

[PDF 29](#_Toc482565542)

[Analyseansätze bei Containerdateien 31](#_Toc482565543)

[Conclusio 32](#_Toc482565544)

[Quellenverzeichnis 33](#_Toc482565545)

# Grundlagen

Generell unterscheidet man zwischen binär geschrieben Daten und Daten im Textformat. Jedoch ist es in vielen Fällen auch so, dass eine Mischform vorliegt. Dateien, in welchen weitere, eventuell auch unterschiedliche, Dateien enthalten sind, werden in dieser Arbeit als Containerdateien bezeichnet.

## Textdateien

Wenn man von Textdateien spricht, sind Dateien gemeint, in denen sogenannte druckbare Zeichen benutzt werden. Die Bytes werden in Form definierter Zeichen interpretiert und können so als darstellbarer Text ausgegeben werden.

## Binärdateien

Anders als bei Textformaten verhält es sich mit Binärdateien. Hier kann jedes einzelne Bit durch die Anwendung ausgewertet werden, was eine Analyse sehr schwierig bis nahezu unmöglich macht.

Meist werden jedoch reguläre Größen für die jeweiligen Datentypen verwendet.

In der nachfolgenden Tabelle sind einige Beispiele für sogenannte „native“ Datentypen aufgelistet:

|  |  |
| --- | --- |
| Allgemeine Bezeichnung (Datentyp gemäß C-Standard) | Größe |
| Byte (char) | 1 Byte |
| Int16(short) | 2 Byte |
| Wide Character (wchar\_t) | 2 Byte |
| In32 (int) | 4 Byte (auf 32 Bit-Systemen) |
| Floating Point (float/double) | 4 bzw. 8 Byte |

Man kann erkennen, dass aufgrund der Größen nicht eindeutig festgestellt werden kann, um welchen Datentyp es sich handelt. Es ist Sache der jeweiligen Anwendung festzulegen, ob 2 Byte als zwei einzelne Byte in Form von Characters, als ein Unicode Character oder als short interpretiert werden.

Darüber hinaus wird beispielsweise ein Byte bzw. char nicht zwangsläufig als Zeichen verwendet, sondern kann auch bitweise genutzt werden, um sogenannte Flags zu speichern und dann ist es natürlich so, dass die obigen Typen - ausgenommen die Floating Points (Dezimalzahlen) - „signed“ oder „unsigned“ sein können; hier wird ein Bit als Vorzeichen interpretiert, was zu Lasten des positiven Wertebereichs geht. So hat ein char (numerisch interpretiert) einen Wertebreich von -127 bis 127, ein unsigned char 0 bis 255.

Der große Vorteil im Speichern von Daten in binärer Form ergibt sich durch den verringerten Platzbedarf. Soll z.B. die Zahl 1.000.000 gespeichert werden, ergibt sich bei der binären Speicherung eines Integers ein Platzbedarf von 4 Byte, speichert man die Zahl als Text benötigt man 7 Byte.

Dieses kleine Beispielprogramm in C++ soll das kurz demonstrieren. Es werden zwei Dateien, zum einen „asInt“ und zum anderen „asChar“ gespeichert. In der Datei „asInt“ wird der Wert als 4Byte Integer gespeichert, in „asChar“ als Text im ASCII Format.

#include <fstream>

int main(int argc, char argv[])

{

int asInt = 1000000;

char asChar[7]{'1', '0', '0', '0', '0', '0', '0'};

std::ofstream ofInt("c:/dev/fileSize/asInt");

//der int wird in ein char-Array kopiert

char intBytes[4]{0};

memcpy(&intBytes, &asInt, 4);

ofInt.write(intBytes, 4);

ofInt.flush();

std::ofstream ofChar("c:/dev/fileSize/asChar");

ofChar.write(asChar, 7);

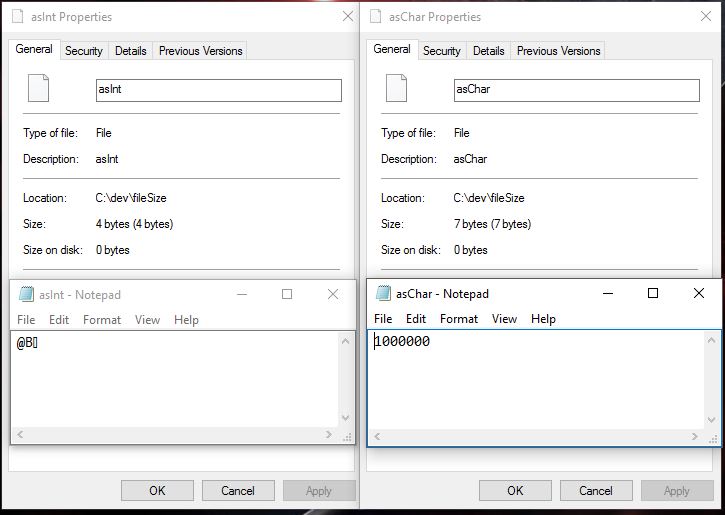
ofChar.flush();

return 0;

}

Das Lesen der Datei würde nun so ablaufen, dass die jeweiligen Bytes in ihre gewünschten Variablen kopiert werden. Dazu muss man jedoch wissen, dass in der Datei „asInt“ die ersten vier Bytes als Integer interpretiert werden müssen.

Im Windows Explorer kann man nun die Dateigrößen sehen und darunter die Interpretation als ASCII Text nach dem Öffnen mit Notepad:



Wie erwartet sind u.a. die entsprechenden Dateigrößen ersichtlich. Natürlich kann der Texteditor den Integer nicht interpretieren und wir sehen nur kryptische Zeichen, während die ASCII Version lesbar ist.

Die Tatsache, dass man bei binären Dateien nur schwer bis gar nicht erahnen kann, welche Rolle die einzelnen Bytes spielen, stellt bei der Analyse eine große Herausforderung dar.

Eine Möglichkeit, dieses Problem zu lösen, besteht darin, sich an veröffentlichte Standards zu halten, die definieren, wie die Datei aufgebaut ist. Stehen solche Standards nicht zur Verfügung, bieten die jedoch äußerst zeitraubenden Techniken des Reverse Engineerings[[1]](#footnote-1) eine Alternative.

Ein sehr bekanntes Beispiel für binäre Dateien ist das JPEG Format:

*Der Header einer JPEG Datei als C++ Struktur[[2]](#footnote-2)*

Beim Einlesen legt man eine Instanz dieser Struktur im Speicher an, liest die ersten 21 Bytes aus der Datei und kopiert diese in die Objekt Instanz.

## Byte Order

Eine weitere Problematik, die auftreten könnte und daher kurz angemerkt werden sollte, stellt die Byte Order dar.

Diese legt fest, in welcher Reihenfolge die Byte integraler Werte im System gespeichert sind.

Big-Endian setzt das höchstwertige Byte an die kleinste Adresse, Little-Endian hingegen das niederwertigste Byte.

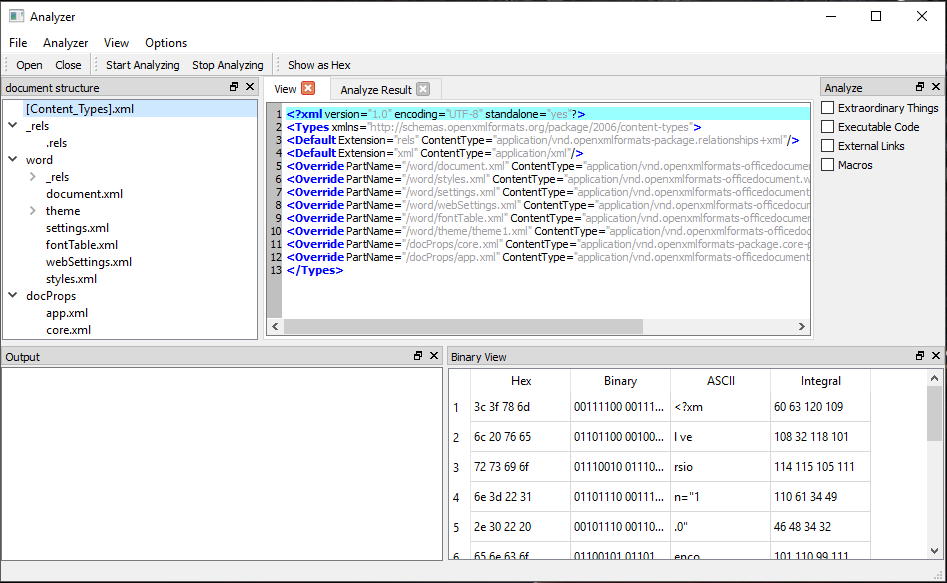
Die meisten PCs verwenden als Basis die 8086er Assembly Struktur, welche Little-Endian verwendet, somit wird kann die oben erwähnte Problemstellung, soweit es diese Arbeit betrifft, vernachlässigt werden. Beachten sollte man diese jedoch insbesondere im industriellen Umfeld, da vor allem für maschinennahe Steuerungen gerne Big-Endian genutzt wird und eine Übersetzung nicht immer gleich erfolgt (z.B. in Pufferspeichern bei Echtzeitsystemen).

## Containerdateien

Als „Containerdatei“ werden in dieser Arbeit alle Formate bezeichnet, welche verschiedene eigenständige Formate aufnehmen können. So lassen sich in einer Containerdatei z.B. Bilddaten in beliebigen Formaten oder formatierte Textdaten innerhalb einer Datei speichern.

# Analysieren mit dem „Analyzer“

Parallel zu dieser Arbeit erstelle ich ein Programm, das es erlaubt, Dateien in verschiedenen Formaten zu betrachten. So kann der Inhalt unabhängig von der eigentlichen Zielanwendung untersucht und ein eingehender Blick auf Aufbau und enthaltene Informationen geworfen werden.



*Der „Analyzer“ beim Öffnen eines MS Word Dokuments*

Ziel des Analyzers ist, sämtliche Inhalte einer Datei einzulesen und unbekannte Teile, sowie Teile, die Unregelmäßigkeiten enthalten und/oder als Bedrohung erkannt wurden, zu lokalisieren. Darüber hinaus soll sich der Anwender einen Überblick über die Gesamtstruktur der Daten verschaffen und einzelne oder Sequenzen von Byte in verschiedenen Interpretationen betrachten können.

Technische Eckdaten:

Sprache: C++

Bibliotheken: Google Test Framework (gtest)

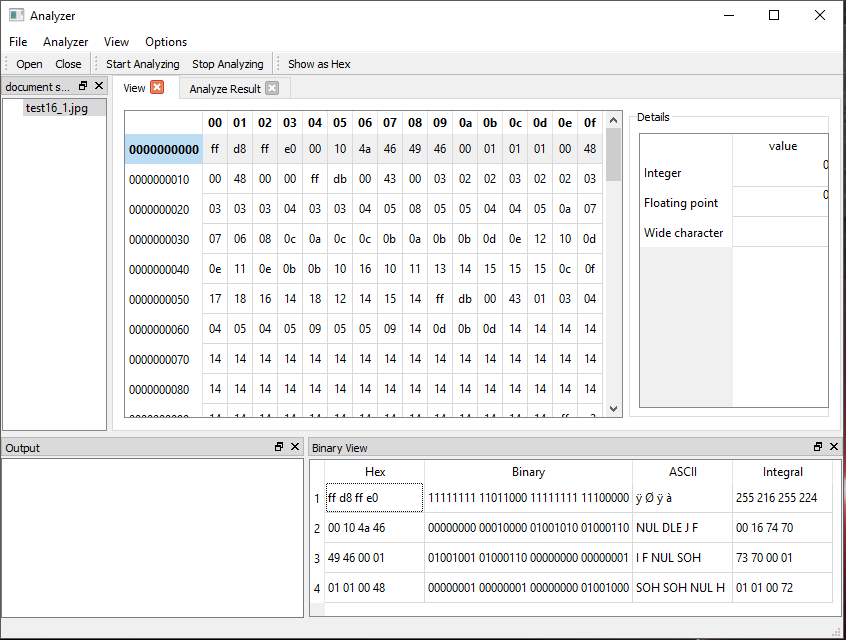
Qt (ausschließlich für die graphische Oberfläche)

IDE: VisualStudio 2015 Pro

Das Projekt wird unter <https://github.com/MarkusKastner/Analyzer> verwaltet.

# Eine Sache der Interpretation

Wie eingangs schon erwähnt, werden binäre Daten vom jeweiligen Programm interpretiert. Da die Datei aber unabhängig des Zielprogrammes gelesen werden soll und das Format oft nicht klar bzw. bei der Analyse auf schadhafte Teile auch weitgehend irrelevant ist, wird eine Möglichkeit benötigt, die Daten auf andere Art lesbar zu machen. Diese Aufgabe kommt dem „Hex-Editor“ zu.



*Öffnen einer Datei im .jpg Format in hexadezimaler Darstellung.*

Alle Daten, die in der Datei „test16\_1.jpg“ enthalten sind, wurden in das Programm geladen und werden in einem Raster als Hexadezimalwerte angezeigt. Da man sich darunter in der Regel aber wenig vorstellen kann, werden die selektierten Werte in verschiedenen Interpretationen angezeigt.

Im Fenster unten rechts befindet sich eine Standardansicht. Alle selektierten Byte werden in Hex, binär, als ASCII Code und als numerischer Wert angezeigt.

Im Fenster „Details“ rechts des Rasters werden, so es eine Interpretation gibt, die möglichen Interpretationen der selektierten Byte angezeigt.

Eine Interpretation bzw. Reinterpretation von Daten nennt sich Typumwandlung oder englisch Typecasting.

Operationen zu Typumwandlungen von primitiven Datentypen wie int, double oder wchar\_t sind in C++ und natürlich allen anderen Hochsprachen standardmäßig enthalten und kommen im Analyzer auch zur Anwendung. Umwandlungen zu komplexeren Typen müssen allerdings implementiert werden.

Wenn von einer „Umwandlung“ gesprochen wird, bedeutet dies, dass lediglich die Interpretation verändert wird, die „Bits und Bytes“ bleiben gleich.

## reinterpret\_cast und memcpy

Zwei Operationen kommen bei Typumwandlungen in C++ sehr häufig vor, zum einen die durch den C++ Standard definierte Funktion reinterpret\_cast und zum anderen die aus der Sprache C stammende memcpy Funktion.

Ein Array von 2 chars, also insgesamt 16 Bit, wird mit reinterpret\_cast in einen integralen 16 Bit Typen „umgewandelt“. Man beachte, dass hier kein 16 Bit Typ im Speicher angelegt wird, sondern nur ein Zeiger auf das erste Byte des Arrays besteht:

char buffer[2];

buffer[0] = 0;

buffer[1] = 0;

int16\_t \* value = reinterpret\_cast<int16\_t\*>(&buffer);

Ein wenig anders verhält es sich bei der Funktion memcpy. Hier wird zunächst ein Speicher angelegt, in den das Array dann physikalisch hineinkopiert wird.

char buffer[2];

buffer[0] = 0;

buffer[1] = 0;

int16\_t value;

memcpy(&value, buffer, 2);

## RGB Farben

Das RGB Farbmodel ist ein additives Farbmodel. Die jeweiligen Farbtöne von rot, grün und blau werden hier übereinandergelegt, woraus sich schließlich der Farbton ergibt.

Für rot, grün und blau steht jeweils ein Byte zur Verfügung, somit hat ein RGB Datensatz 3 Byte.

In numerischen Werten, die des Öfteren in Editoren für User Interfaces zu finden sind, würde das so aussehen:

[255][0][0] … ergibt ein sattes rot

[0][0][0] … ist schwarz

usw.

Die Erweiterung um ein viertes Byte ermöglicht die Realisierung von Transparenz.

## Opcodes

Opcodes sind die grundlegenden Befehle und somit die Basis von Software. Jeder Einzelbefehl eines Programmes wird durch einen Opcode definiert.

Hier der prinzipielle Aufbau:

|  |  |  |  |
| --- | --- | --- | --- |
| 1. Byte | 2. Byte | 3.&4. Byte | 5.&6. Byte |
| Opcode | Adressmodus | Adresse | Daten |

1.Byte:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| x | x | x | x | x | x | d | w |

x… Befehlscode

d… destination: 0 = das Register ist die Quelle, 1 = Register ist das Ziel

w... word: 0 = 8-Bit-Operation, 1 = 16-Bit-Operation

2.Byte:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| mod | | reg | | | r/m | | |

reg… Registeroperand

mod-r/m… Speicheroperant

Für den Registeroperanden ergeben sich folgende Werte:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| reg | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| w=0 | AL | CL | DL | BL | AH | CH | DH | BH |
| w=1 | AX | CX | DX | BX | SP | BP | SI | DI |

Als Beispiel sei hier der Move-Befehl angeführt, welcher Werte zwischen Registern Transferieren kann.

8 Bit Move-Befehl mit allgemeiner Adressierung, bei dem „reg“ das Quellregister ist:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

Das entspricht dem Hex-Wert 0x88.

Die restlichen Byte des Befehls sind zur Adressierung und somit dynamisch, also nutzen sie zu einer Vergleichsauswertung eigentlich nichts.

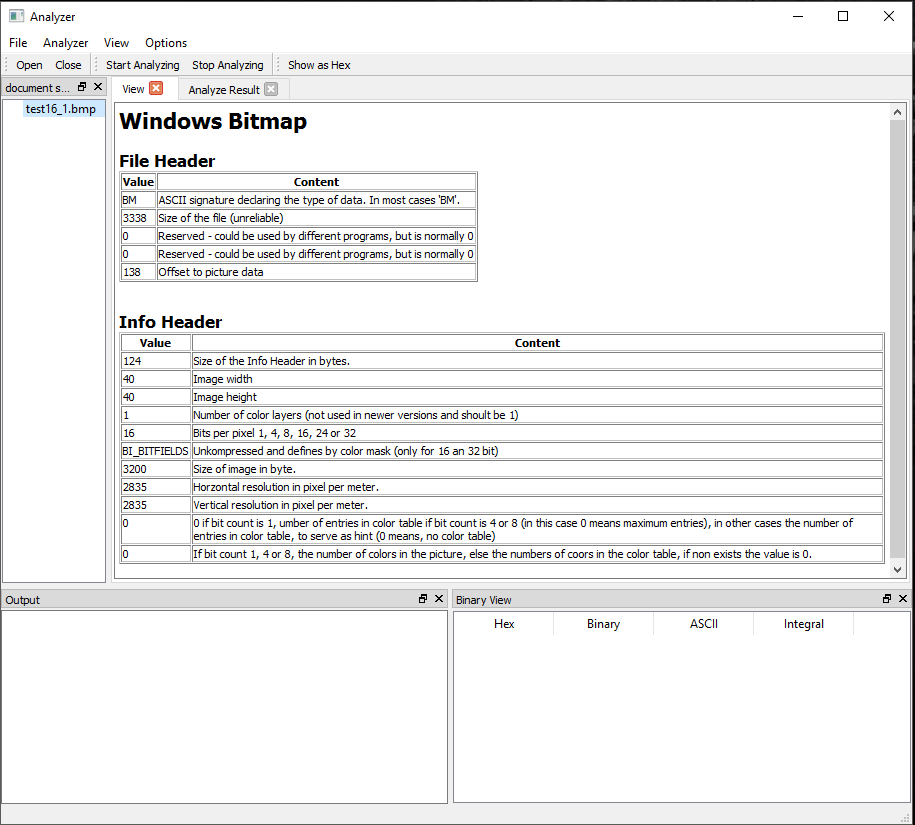
Als gängige Basis moderner Rechner dient der Befehlssatz des 8086-Prozessors:

|  | | |
| --- | --- | --- |
| **Instruction** | **Meaning** | **Opcode** |
| AAA | ASCII adjust AL after addition | 0x37 |
| AAD | ASCII adjust AX before division | 0xD5 |
| AAM | ASCII adjust AX after multiplication | 0xD4 |
| AAS | ASCII adjust AL after subtraction | 0x3f |
| ADC | Add with carry | 0x10…0x15, 0x80/2…0x83/2 |
| ADD | Add | 0x00…0x05, 0x80/0…0x83/0 |
| AND | Logical AND | 0x20…0x25, 0x80/4…0x83/4 |
| CALL | Call procedure | 0x9A, 0xE8, 0xFF/2, 0xFF/3 |
| CBW | Convert byte to word | 0x98 |
| CLC | Clear carry flag | 0xF8 |
| CLD | Clear direction flag | 0xFC |
| CLI | Clear interrupt flag | 0xFA |
| CMC | Complement carry flag | 0xF5 |
| CMP | Compare operands | 0x38…0x3D, 0x80/7…0x83/7 |
| CMPSB | Compare bytes in memory | 0xA6 |
| CMPSW | Compare words | 0xA7 |
| CWD | Convert word to doubleword | 0x99 |
| DAA | Decimal adjust AL after addition | 0x27 |
| DAS | Decimal adjust AL after subtraction | 0x2F |
| DEC | Decrement by 1 | 0x48, 0xFE/1, 0xFF/1 |
| DIV | Unsigned divide | 0xF6/6, 0xF7/6 |
| ESC | Used with floating-point unit |  |
| HLT | Enter halt state | 0xF4 |
| IDIV | Signed divide | 0xF6/7, 0xF7/7 |
| IMUL | Signed multiply | 0x69, 0x6B, 0xF6/5, 0xF7/5, 0x0FAF |
| IN | Input from port | 0xE4, 0xE5, 0xEC, 0xED |
| INC | Increment by 1 | 0x40, 0xFE/0, 0xFF/0 |
| INT | Call to interrupt | 0xCD |
| INTO | Call to interrupt if overflow | 0xCE |
| IRET | Return from interrupt | 0xCF |
| Jcc | Jump if condition | 0x70…0x7F, 0xE3, 0x0F83, 0x0F87 |
| JCXZ | Jump if CX is zero | 0xE3 |
| JMP | Jump | 0xE9…0xEB, 0xFF/4, 0xFF/5 |
| LAHF | Load FLAGS into AH register | 0x9F |
| LDS | Load pointer using DS | 0xC5 |
| LEA | Load Effective Address | 0x8D |
| LES | Load ES with pointer | 0xC4 |
| LOCK | Assert BUS LOCK# signal | 0xF0 |
| LODSB | Load string byte | 0xAC |
| LODSW | Load string word | 0xAD |
| LOOP/LOOPx | Loop control | 0xE0..0xE2 |
| MOV | Move |  |
| MOVSB | Move byte from string to string | 0xA4 |
| MOVSW | Move word from string to string | 0xA5 |
| MUL | Unsigned multiply |  |
| NEG | Two's complement negation |  |
| NOP | No operation | 0x90 |
| NOT | Negate the operand, logical NOT |  |
| OR | Logical OR |  |
| OUT | Output to port |  |
| POP | Pop data from stack |  |
| POPF | Pop FLAGS register from stack | 0x9D |
| PUSH | Push data onto stack |  |
| PUSHF | Push FLAGS onto stack | 0x9C |
| RCL | Rotate left (with carry) |  |
| RCR | Rotate right (with carry) |  |
| REPxx | Repeat MOVS/STOS/CMPS/LODS/SCAS |  |
| RET | Return from procedure |  |
| RETN | Return from near procedure |  |
| RETF | Return from far procedure |  |
| ROL | Rotate left |  |
| ROR | Rotate right |  |
| SAHF | Store AH into FLAGS | 0x9E |
| SAL | Shift Arithmetically left (signed shift left) |  |
| SAR | Shift Arithmetically right (signed shift right) |  |
| SBB | Subtraction with borrow |  |
| SCASB | Compare byte string | 0xAE |
| SCASW | Compare word string | 0xAF |
| SHL | Shift left (unsigned shift left) |  |
| SHR | Shift right (unsigned shift right) |  |
| STC | Set carry flag | 0xF9 |
| STD | Set direction flag | 0xFD |
| STI | Set interrupt flag | 0xFB |
| STOSB | Store byte in string | 0xAA |
| STOSW | Store word in string | 0xAB |
| SUB | Subtraction |  |
| TEST | Logical compare (AND) |  |
| WAIT | Wait until not busy | 0x9B |
| XCHG | Exchange data |  |
| XLAT | Table look-up translation | 0xD7 |
| XOR | Exclusive OR |  |

# Dateiformate

Im letzten Kapitel ging es darum, wie Daten interpretiert werden können. Ob ein Byte als ASCII-Zeichen, Befehlscode oder sonst wie interpretiert wird, bestimmt der Zweck der Verwendung. Um diesen klar zu machen und die Daten in eine geordnete Struktur zu bringen, benötigt man klar definierte Formate.

Der Analyzer wurde so gestaltet, dass er definierte Formate gleich speziell behandelt und enthaltene Informationen entsprechend ausgibt.



*Öffnen einer .bmp Datei*

Um ein Format zu erkennen, wird nicht wie üblich die Dateiendung verwendet, sondern es werden Teile des Inhaltes überprüft und mit der Spezifikation der Datei verglichen. Bei einer Übereinstimmung wird der Inhalt der Datei in den passenden Interpreter geladen und ausgegeben.

Von der Verwendung der Dateiendung wird insbesondere deshalb Abstand gehalten, da diese ohne weiteres verändert werden kann und somit irreführend ist. So könnte man zum Beispiel eine .exe Datei in eine .txt Datei umbenennen und versenden. Falls am Zielrechner kein Virenscanner aktiv ist, würde die Datei lediglich dadurch auffallen, dass ihr Inhalt im Texteditor nur aus kryptischen Zeichen besteht. Der eigentliche Inhalt der Datei ist aber von der Änderung nicht betroffen und sie bleibt eine voll funktionsfähige Anwendung, die, sobald sie in den Speicher geladen ist, auch ausgeführt wird.

Im Prinzip sind Dateiendungen nur eine Orientierung für das jeweilige Betriebssystem um festzulegen, mit welchem Programm die Datei geöffnet werden soll - auf den Inhalt der Datei haben sie keinerlei Auswirkung.

## Binärdateien

### Bitmap

Um eine Binärdatei zu lesen, müssen die Offsets und die Datentypen bekannt sein. Danach kann man die eingelesenen Daten gemäß ihrer Funktion nutzen.

In der Regel haben diese Dateien am Anfang eine statische Struktur, die die eigentlichen Nutzdaten beschreibt. Diese Struktur wird als Header bezeichnet.

Das BMP-Format hat gleich zwei davon:

Dem Fileheader kann entnommen werden, wie die Datei aufgebaut ist. Wichtig ist hier vor allem der Offset, an welchem die eigentlichen Nutzdaten beginnen.

|  |  |  |  |
| --- | --- | --- | --- |
| **BITMAPFILEHEADER (Größe: 14 Byte)** | | | |
| **Offset (Byte)** | **Datentyp** | **Größe**  **(Byte)** | **Inhalt** | |
| 0 | uint16\_t | 2 | bfType:  ASCII-Zeichenkette "BM" (Hex: 0x424D, Dezimal: 16973). | |
| 2 | uint32\_t | 4 | bfSize:  Größe der BMP-Datei in Byte. (unzuverlässig) | |
| 6 | uint32\_t | 4 | bfReserved:  Reserviert, von der Software abhängig, standardmäßig 0 | |
| 10 | uint32\_t | 4 | bfOffBits:  Offset der Bilddaten in Byte vom Beginn der Datei. | |

Der Infoheader macht nähere Angaben über die Beschaffenheit der Nutzdaten:

|  |  |  |  |
| --- | --- | --- | --- |
| **BITMAPINFOHEADER (Größe: 40 Byte)** | | | |
| **Offset (Byte)** | **Datentyp** | **Größe**  **(Byte)** | **Inhalt** | |
| 14 | uint32\_t | 4 | biSize:  Größe der BITMAPINFOHEADER-Struktur in Byte | |
| 18 | int32\_t | 4 | biWidth:  Breite der Bitmap in Pixel. | |
| 22 | int32\_t | 4 | biHeight:  Der Betrag gibt die Höhe der Bitmap in Pixel an.  - Ist der Wert positiv, so ist die Bitmap eine sogenannte "bottom-up"-Bitmap (die Bilddaten beginnen mit der untersten und enden mit der obersten Bildzeile).  - Ist der Wert negativ, so ist die Bitmap eine "top-down"-Bitmap (die Bilddaten beginnen mit der obersten und enden mit der untersten Bildzeile). | |
| 26 | uint16\_t | 2 | biPlanes:  1 (Stand in einigen älteren Formaten wie PCX für die Anzahl der Farbebenen, wird aber für BMP nicht verwendet) | |
| 28 | uint16\_t | 2 | biBitCount:  Farbtiefe der Bitmap in bpp an; muss einer der folgenden Werte sein: 1, 4, 8, 16, 24 oder 32. Bei 1, 4 und 8 bpp sind die Farben indiziert. | |
| 30 | uint32\_t | 4 | biCompression:  Einer der folgenden Werte:   * 0 (BI\_RGB): unkomprimiert. * 1 (BI\_RLE8): lauflängenkodiert für 8 bpp. Nur erlaubt wenn biBitCount=8 und biHeight positiv. * 2 (BI\_RLE4): lauflängenkodiert für 4 bpp. Nur erlaubt wenn biBitCount=4 und biHeight positiv. * 3 (BI\_BITFIELDS): unkomprimiert und benutzerdefiniert (mittels Farbmasken) kodiert. Nur erlaubt wenn biBitCount=16 oder 32. | |
| 34 | uint32\_t | 4 | biSizeImage:   * *Wenn biCompression=BI\_RGB:* Entweder 0 oder die Größe der Bilddaten in Byte. * *Ansonsten:* Größe der Bilddaten in Byte. | |
| 38 | int32\_t | 4 | biXPelsPerMeter:  Horizontale Auflösung des Zielausgabegerätes in Pixel pro Meter; meistens auf 0 gesetzt. | |
|  |  |  |  | |
| 42 | int32\_t | 4 | biYPelsPerMeter:  Vertikale Auflösung des Zielausgabegerätes in Pixel pro Meter; meistens auf 0 gesetzt. | |
| 46 | uint32\_t | 4 | biClrUsed:   * *Wenn biBitCount=1:* 0. * *Wenn biBitCount=4 oder 8:* die Anzahl der Einträge der Farbtabelle; 0 bedeutet die maximale Anzahl (2, 16 oder 256). * *Ansonsten:* Die Anzahl der Einträge der Farbtabelle (0=keine Farbtabelle). | |

Danach kommen, je nach oben definierter Variante, Farbmasken, Farbtabellen für indizierte Farben und schließlich die Bilddaten für das Raster.

Will man diese Datei nun analysieren, können folgende Überlegungen gemacht werden:

* **Passt die errechnete Größe gemäß den Headerdaten zur tatsächlichen Größe?**

Bei nicht komprimierten Variationen ohne indizierte Farben kann das sehr einfach berechnet werden. Auch kann geprüft werden, ob der Offset für die Bilddaten mit dem benötigten Speicher für Header, Farbmasken und Farbtabellen übereinstimmt.

Bei codierten und indizierten Varianten müssen die Daten durchlaufen werden um festzustellen, welche Größe die Nutzdaten wirklich haben.

* **Handelt es sich wirklich um RGB-Pixel?**

Je nach Farbtiefe können die einzelnen Byte verschiedene Wertebereiche annehmen.

Bei 16 bpp werden beispielsweise nur die beiden niederwertigen Bytes der Farbmaske berücksichtigt, das andere Byte sollte leer sein.

Bei indizierten Farben kann geprüft werden ob der Index stimmt. Manche Programme ignorieren fehlerhafte Indizes ohne Meldungen auszugeben, was es sehr einfach macht, unerkannte Inhalte zu platzieren.

Bei hohen Auflösungen, bei denen der gesamte Wertebereich eines Bytes ausgenutzt wird, ist es jedoch nicht möglich zu sagen, ob dieses Byte einen Farbcode oder etwas anderes darstellen soll.

### Executables

Das PE-Format basiert auf dem Common Object File Format und wurde bei der Einführung der 32 Bit-Basis entwickelt.

Wenn man Quellcode kompiliert, generiert der Compiler Object-Dateien. In ihnen sind die Daten und Funktionen in Maschinensprache enthalten. Um aus dem Sammelsurium letztendlich ein lauffähiges Programm zu erstellen, werden diese Daten und Funktionen „gelinkt“, man könnte auch sagen, der Linker kopiert die einzelnen Funktionen in eine Datei zusammen.

Das ist natürlich nur eine sehr vereinfachte Beschreibung, aber für das Verständnis der folgenden Seiten ausreichend.

|  |
| --- |
| DOS Header |
| PE Signature |
| COFF Header |
| Optional Header |
| Section Table |
| Mappable Sections |

*Struktur einer PE Datei[[3]](#footnote-4)*

Als erstes kommt der DOS Header:

struct DOSHeader

{

char Signature[2];

short Lastsize;

short NumBlocks;

short NumReloc;

short hdrsize;

short MinAlloc;

unsigned short MaxAlloc;

short ss;

short sp;

short Checksum;

short ip;

short cs;

short relocpos;

short noverlay;

short Reserved1[4];

short OEMID;

short OEMInfo;

unsigned short Reserved2[10];

long PEOffset;

};

Hier sind für die Analyse eigentlich nur die ersten zwei Byte und der PEOffset wichtig.

Die ersten beiden Byte sind immer MZ, die Initialen für Mark Zbikowski, dem Schöpfer des ersten Linkers für DOS.

Der PEOffset gibt an, wo in der Datei die PE-Signatur (4Byte -> PE/0/0) zu finden ist.

Nach der PE-Signatur kommt der Eingangs schon erwähnte COFF-Header.

Im COFF-Header kann man zum Beispiel den Parameter „Machine“ auf bekannte Maschinentypen prüfen. Der Parameter „Characteristics“ ist ebenso wichtig und verrät, ob es sich um eine ausführbare Datei oder eine Programmbibliothek handelt.

struct COFFHeader

{

short Machine;

short NumbSections;

long TimeDateStamp;

long PointerToSymbolTable;

long NumOfSymbols;

short SizeOfOptionalHeader;

short Characteristics;

};

Weiter ist der Parameter „SizeOfOptionalHeader“ wichtig, da er die Größe der nächsten wichtigen Struktur angibt.

Dieser Header ist für ausführbare Dateien unverzichtbar und enthält sehr viele Informationen. Prüfen kann man hier den Parameter „Signature”, der aussagt, ob es sich um ein 32 oder 64 Bit ausführbares oder ein ROM Image handelt.

struct PEOptionalHeader

{

short Signature;

char MajorLinkerVersion;

char MinorLinkerVersion;

long SizeOfCode;

long SizeOfInitializedData;

long SizeOfUninitializedData;

long AddressOfEntryPoint;

long BaseOfCode;

long BaseOfData;

long ImageBase;

long SectionAlignment;

long FileAlignment;

short MajorOSVersion;

short MinorOSVersion;

short MajorImageVersion;

short MinorImageVersion;

short MajorSubsystemVersion;

short MinorSubsystemVersion;

long Win32VersionValue;

long SizeOfImage;

long SizeOfHeaders;

long Checksum;

short Subsystem;

short DLLCharacteristics;

long SizeOfStackReserve;

long SizeOfStackCommit;

long SizeOfHeapReserve;

long SizeOfHeapCommit;

long LoaderFlags;

long NumberOfRvaAndSizes;

data\_directory \* DataDirectory[];

};

Der Parameter „Subsystem“ definiert, wie das Programm vom Betriebssystem verwendet wird, also ob zum Beispiel ein GUI verwendet wird, es sich um einen Systemprozess (z.B. Treiber) handelt etc..

#### Analysemöglichkeit Struktur

Es ist eine gängige Praxis, Schadsoftware in eigentlich harmlosen Dateien wie PDFs zu verstecken. Abgesehen von sehr ausgefeilten Techniken funktioniert das so, dass das Schadprogramm, also der „Payload“ in die Datei an einer unauffälligen Stelle hineinkopiert und durch eine geeignete Funktionalität des Zielprogrammes ausgeführt wird.

Falls die Datei dabei nicht aufgeteilt wird, kann mithilfe der oben beschriebenen Strukturen eine Analyse durchgeführt werden. Hierbei überprüft man die Daten auf das Vorkommen der Sequenz „MZ“. Findet man diese Signatur, kann man die Byte am Offset des „PEOffset“-Parameters prüfen. Ergibt dieser Wert einen Sinn und findet man die Werte für den PE Header an der Stelle des „PEOffset“ Offsets, ist die Wahrscheinlichkeit, dass es sich um eine ausführbare Datei handelt, bereits sehr groß.

Natürlich kann man anhand der anderen gezeigten Parameter in selber Weise weiter analysieren.

#### Analysemöglichkeit Code

Eine weitere Möglichkeit, um ausführbaren Code zu finden, besteht darin, häufig vorkommende Muster von Opcodes – also logische Sequenzen - zu suchen.

Für die Implementierung werden meist native Sprachen wie C, C++, aber auch Assembler verwendet. Eine Programmsyntax wird zwar von C oder C++ im Endeffekt auch auf Assembler heruntergebrochen, jedoch kann das Ergebnis sehr unterschiedlich aussehen, da die beiden Hochsprachen komplexere Aufgaben sicher anders darstellen als eine selbst implementierte Assembler-Logik.

Auch die Umsetzung unterschiedlicher Compiler wird nicht zwingend gleich sein. So müsste man die zu suchende Logik in C und C++ mit verschiedenen Compilern übersetzen und gegen jedes Ergebnis analysieren.

Eingehende und verwertbare Versuche in diese Richtung würden den Rahmen dieser Arbeit sprengen.

## Textdateien

Der Begriff „Textdatei“ soll hier nicht irreführend sein; gemeint sind Formate, welche ihre Struktur über ASCII Zeichen beschreiben, was aber nicht zwangsweise heißen muss, dass diese leicht lesbar sind.

Natürlich ist es nicht ganz so extrem wie Binärformate im Hex-Editor, doch ein kurzer Blick auf das PDF Format verrät schon einiges:

%PDF-1.4

1 0 obj

<< /Title (Hallo Welt) >>

endobj

2 0 obj

<< /Type /Catalog

/Pages 3 0 R

>>

endobj

3 0 obj

<< /Type /Pages

/MediaBox [0 0 595 842]

/Resources

<< /Font << /F1 4 0 R >>

/ProcSet [/PDF /Text]

>>

/Kids [5 0 R]

/Count 1

>>

endobj

…

*Auszug einer sehr simplen PDF Datei[[4]](#footnote-5)*

Die Standards dieser Formate sind nicht weniger aufwändig als jene der binären Formate, wenngleich man sie leichter entziffern kann.

Der wohl am häufigsten vorkommende und einer der ältesten Zeichensätze ist der ASCII (American Standard Code for Information Interchange). Er definiert die 10 arabischen Ziffern, das lateinische Alphabet in Groß- und Kleinschreibung, einige druckbare Zeichen sowie einige nicht druckbare Zeichen, die meist als Steuerzeichen für Anwendungen verwendet werden, beispielsweise ein Zeilenumbruch, Tabulator, etc.

Insgesamt sind in diesem Standard 128 Zeichen definiert, welche über eine 7 Bit Codierung realisiert werden. Das 8. Bit des Bytes bleibt.



*ABC als 7 Bit Code[[5]](#footnote-6)*

Da man allerdings schon bei den Umlauten in der deutschen Sprache auf Probleme stößt, gibt es eine Reihe von Erweiterungen und Lösungen, bei denen sich ein Zeichen aus mehreren Byte zusammensetzen kann. Man nennt diese Formate auch Multibyte.

Der weltweite De-facto-Standard ist die UTF-8[[6]](#footnote-7) Codierung, vielfach auch als Unicode bezeichnet.

Bei dieser bleibt der ASCII Zeichensatz vollständig gültig. Werden mehr Byte zur Darstellung benötigt, wir mit Hilfe der Anfangsbits des Bytes bekannt gegeben wie viele Byte für die Definition des Zeichens verwendet werden.



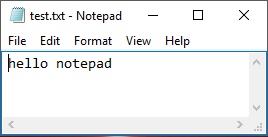
Die Bits in Klammern werden genutzt, um die Anzahl der Byte pro Zeichen festzulegen, die Bits außerhalb dienen zur Definition des eigentlichen Zeichens.

So können maximal 8 Byte zur Definition eines Zeichens verwendet werden.

Bei der Analyse unbekannter Dateiformate kann man beispielsweise gegen dieses Muster prüfen, um herauszufinden, um welchen Zeichensatz es sich handelt.

#### ASCII

Die einfachste Form einer Textdatei unter Windows ist ein mit Notepad.exe verfasstes Dokument.



C:\Users\Markus\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Capture.png

Lässt man sich diese Datei mit dem Analyzer im Hex-Format anzeigen, sieht man, dass sie tatsächlich nicht mehr als diese 13 Zeichen enthält.

#### Unicode

Eine sehr speichereffiziente Lösung um mehr als 255 Zeichen darstellen zu können ist Unicode. Anstatt statisch mehr Byte für jedes Zeichen zu verwenden, erfolgt die Erweiterung dynamisch:

|  |  |
| --- | --- |
| (0)0000000 | Einzelnes Byte wird als ASCII Zeichen interpretiert |
| (110)00000 (10)000000 | Unicode mit zwei Bytes, wobei im ersten Byte die Gesamtzahl der Bytes festlegt wird und weiteren Bytes mit 10 beginnen. |
| (11110)000 (10)000000(10)000000 (10)000000 | Zeichen mit 4 Bytes |

#### XML

Vor allem XML (Extensible Markup Language) erfreut sich in den letzten Jahren zunehmend großer Bedeutung.

<?xml version="1.0" encoding="UTF-8"?>

<text>hello world</text>

*„Hello World“ in XML.*

Die erste Zeile gibt die Version des XML-Standards und das Format der Text-Codierung an und muss in allen XML konformen Dokumenten enthalten sein. Danach hat man an sich sehr viel Gestaltungsfreiheit.

Die in den spitzen Klammern stehenden Ausdrücke nennt man „Tags“. Es gibt immer einen öffnenden und einen schließenden Teil und in der Mitte stehen die Daten, wobei beliebig viele dieser Tags geschachtelt werden können.

<text>

<h1>hello world</h1>

<simple size=12>some simple text.</simple>

</text>

Der Tag <h1> könnte hier zum Beispiel, wie in HTML eine Überschrift bedeuten, der Tag <simple> könnte einfach Text in einer Schriftgröße 12 (als „attribute“ im Tag) sein, usw.

XML ist aber keinesfalls auf die Formatierung von Ausgabetexten beschränkt, es können sämtliche Daten so gespeichert werden, wobei es wieder auf die Interpretation der Anwendung ankommt.

Sieht man sich im Analyzer eine Word Datei an, wird man schnell feststellen, dass diese intern aus vielen einzelnen XML Dateien besteht.

Um also zu beurteilen, ob eine XML Datei irgendwie bedenklich ist, muss man sich ansehen, wie diese von der Zielanwendung verarbeitet wird und welche Art von Daten sie enthält.

### Scripts

Von reinem Text, der von einer Anwendung als solcher interpretiert und ausgegeben wird, geht keine große Gefahr aus. Sehr viele der Anwendung nutzen jedoch die Möglichkeit, Behandlungsroutinen gemeinsam mit den Daten zu speichern. Auch bieten Container-Formate die Möglichkeit, Binärdaten einzubinden, was meist als „Stream“ bezeichnet wird. Die Zielanwendung erhält dabei die Anweisung oder weiß bereits, wie mit den Inhalten dieser Streams umzugehen ist.

Was hier gleichermaßen wie in binären Formaten aber verstärkt auftritt, ist die Verwendung von Scripts.

Unter einem Script versteht man ein Set aus Anweisungen, die von der Zielanwendung umgesetzt werden. Es gibt dafür eine relativ große Anzahl von Scriptsprachen, die bekanntesten sind Visual Basic (VB) und seine Derivate (VBA, VBS, …), Java Script (JS) und vor allem auch Shell Scripts, wie die Power Shell unter MS Windows.

Der große Unterschied zu nativen Sprachen wie C++ oder Assembler ist, dass die Scripts bis zur effektiven Ausführung durch die Zielsoftware im Quellcode vorliegen, es erfolgt also kein Kompilieren in Maschinencode.

Für unsere Zwecke ist dies insofern nützlich, als dass wir gleich nach der Syntax dieser Scripts suchen können, ohne uns große Gedanken darüber zu machen, wie das Dateiformat genau aussieht.

Die wohl häufigsten Scripts sind Java Script und Visual Basic:

|  |  |
| --- | --- |
| Java Script | Visual Basic |
| var x = 0 | Dim x  x= 0 |
| function Add(a, b){  return a + b  } | Function Add(ByVal x, ByVal y)  Add = x + y  End Function |
| if (a < b){  …  } | If a < b Then  …  End If |
| for (int i = 0; i > 5; i++){  …  } | For I As Int = 0 To 4  …  Next |
| … | … |

Beim direkten Vergleich der beiden Sprachen fällt auf, dass der Unterschied primär in der Schreibweise liegt, der syntaktische Aufbau aber im Großen und Ganzen gleich ist.

Insbesondere diese Syntaxkombinationen können in großen, ASCII-basierten Dateien leicht aufgestöbert werden. Man braucht zum Beispiel nur nach dem Wort „function“ suchen, das Leerzeichen danach berücksichtigen und wenn das nächste Zeichen nach allen ASCII Buchstaben und den Zahlen von 0 bis 9 eine öffnende Klammer ist, handelt es sich mit großer Wahrscheinlichkeit um eine Funktionsdeklaration.

Da diese Kombinationen im normalen Sprachgebrauch nicht vorkommen (i.d.R. ein Leerzeichen vor der Klammer), ist die Wahrscheinlichkeit, ein Stück Code entdeckt zu haben, schon sehr groß.

## Container-Dateien

### Open Office XML [[7]](#footnote-8)

Dieses Dokument zum Beispiel ist im Programm MS Word© verfasst. Die Textbereiche inklusive ihrer Steuerzeichen sind im Textformat, genauer gesagt im XML Format, gespeichert, es enthält jedoch auch Bilddateien, die in binärer Form vorliegen.

Unterschieden wird im Prinzip zwischen 2 Arten von Dateien, die als ZIP-Archiv[[8]](#footnote-9) zusammengefügt sind. Sogenannte „Parts“, die den Inhalt des Dokuments enthalten, und „Items“, die beschreiben, wie der Inhalt dargestellt wird.

### PDF

Ein anderes Format ist das PDF (Portable Document Format). Hier werden Importdateien als „Streams“ abgelegt, wobei es für das PDF theoretisch irrelevant ist, was in dem Stream enthalten ist.

Hier eine PDF Datei in ihrer einfachsten Ausführung:

%PDF-1.4

1 0 obj

<< /Title (Hallo Welt) >>

endobj

2 0 obj

<< /Type /Catalog /Pages 3 0 R>>

endobj

3 0 obj

<< /Type /Pages/MediaBox [0 0 595 842]/Resources

<< /Font /F1 4 0 R >> ProcSet [/PDF /Text] >>

/Kids [5 0 R]/Count 1>>

endobj

4 0 obj

<< /Type /Font /Subtype /Type1 /BaseFont /Helvetica

/Encoding /WinAnsiEncoding>>

endobj

5 0 obj

<< /Type /Page /Parent 3 0 R /Contents 6 0 R>>

endobj

6 0 obj

<< /Length 41>>

stream

/F1 48 Tf BT 72 746 Td (Hallo Welt) Tj ET

endstream

endobj

xref

0 7

0000000000 65535 f

0000000009 00000 n

0000000050 00000 n

0000000102 00000 n

0000000268 00000 n

0000000374 00000 n

0000000443 00000 n

trailer

<< /Size 7 /Info 1 0 R /Root 2 0 R>>

startxref

534

%%EOF

Ganz oben haben wir eine Versionskennung, hier 1.4.

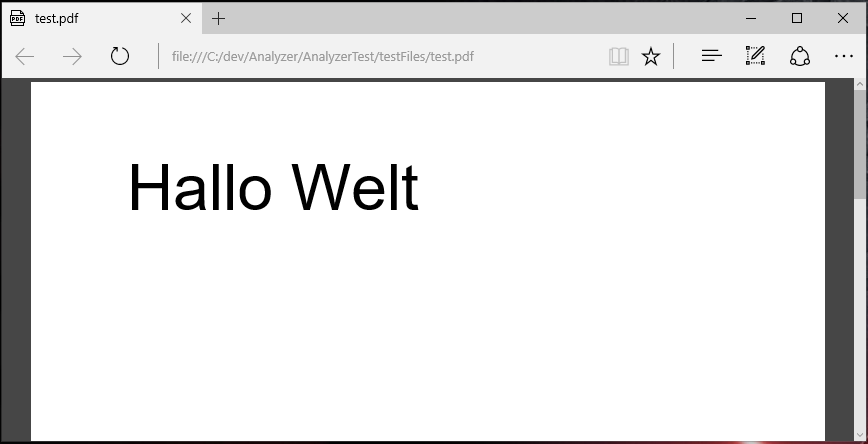
Dann kommt in den meisten Fällen erst die Referenztabelle, welche hier ganz unten ist. In dieser sind alle Offsets der einzelnen Objekte enthalten.

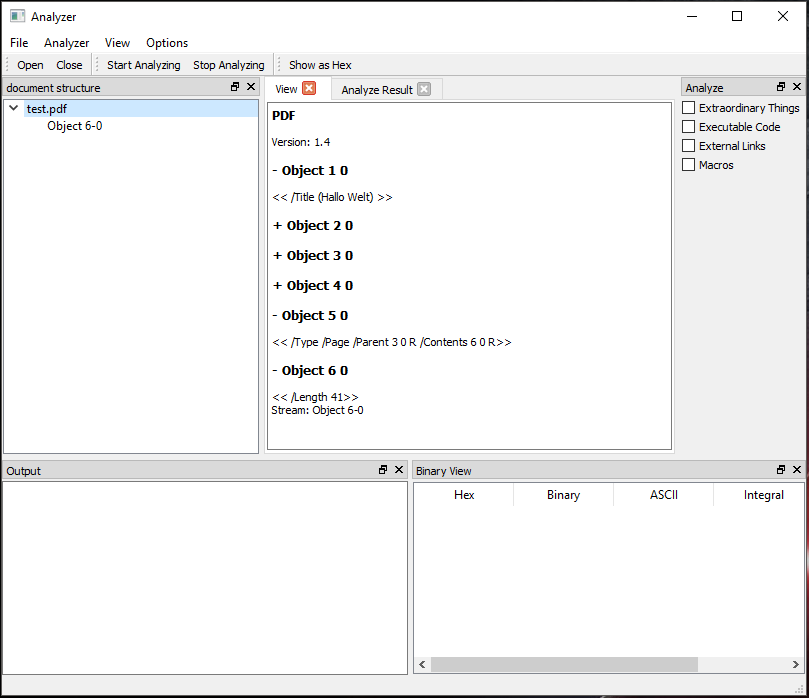
Die Objekte sind der eigentliche Speicherort der Daten. Die erste Zahl der Nummer ist fortlaufend, die nächste eine Revisionskennung.

Jedes Objekt kann mit seiner Objektnummer im ganzen Dokument referenziert werden.

So sieht man zum Beispiel im Obj 5 0 wie mit „/Parent 3 0 R“ auf Obj 3 0, und mit „/Contents 6 0 R“ auf Obj 6 0 verwiesen wird, wobei das „R“ hier das Zeichen für die Referenz ist.

Die Ausgabe der obigen Datei mit einem PDF Viewer:





Die Betrachtung über den Analyzer soll erleichtert werden, indem die Objekte in einer „Folded List“ untergebracht und alle Streams im Strukturbaum als eigene Dateien, die für sich alleine behandelt werden können, auswählbar sind.

Das erlaubt einen schnellen Überblick, ohne die Möglichkeit einer eingehenden Betrachtung zu verlieren.

### Analyseansätze bei Containerdateien

Für enthaltene Importdokumente wie Bilddaten oder XML Dateien, kommen natürlich alle vorher schon besprochenen Möglichkeiten in Betracht.

Sieht man sich die beiden Formate an, fällt auf, dass für sehr wenig Ausgabe relativ Aufwand betrieben wird. Das mag sich bei großen Dateien relativieren, aber wenn man sich ein PDF mit 40 MB und einigen hundert Objekten vorstellt, wird schnell klar, dass man rasch den Überblick verlieren kann. Vor allem ist nicht mehr in zumutbarer Zeit möglich, sich sämtliche Streams anzusehen und zu prüfen, wie und wo sie im Dokument verlinkt sind. Somit wäre es ein leichtes, den Payload in einem Objekt oder einem Unterordner einer Word-Datei zu verstecken und diesen einfach nicht zu verlinken. Ob die Datei dann 40 oder 41 MB hat, fällt nicht auf, bzw. kann man auch gar nicht prüfen, wie groß sie genau sein sollte.

Ein Ansatz wäre daher, die Referenzen jedes Objektes / jeder Datei zu prüfen und weiters zu sehen, wie diese Daten verwendet werden. Findet man Objekte, die nicht regulär referenziert werden, ist das zumindest sehr verdächtig.

In Bezug auf PDF Streams kann man natürlich auch prüfen, ob die Stream Offsets mit der tatsächlichen Länge des Streams zusammenpassen. In der Regel werden PDFs so eingelesen, dass man zunächst anhand der Liste der Start-Offsets die Objekte bestimmt und diese dann interpretiert, wobei jeder Stream mit seiner Länge angegeben ist. Liest eine Anwendung nur diese die Daten anhand der Längenangabe aus, ohne zu prüfen, ob die „endstream“ Marke erreicht wurde, bleiben die restlichen Daten möglicherweise unbemerkt.

Ein weiteres Problem ist, dass der PDF Standard Java Script und MS Office VB-Makros verwenden können. Ein Script, das einen Payload zur Ausführung bringt, ist nur ein paar Zeilen lang und es braucht kein großes Knowhow zur Realisierung, da das Internet voll von solchen Informationen ist.

Generell sollte jedes Makro oder Script unter die Lupe genommen werden, ganz besonders; wenn nicht dokumentierte „Magic-Numbers“ verwenden werden, Kopieroperationen stattfinden und externe Referenzen ausgeführt werden.

# Conclusio

Diese Formen der Analyse sind eine Gratwanderung. Einerseits muss den einzelnen Formaten Rechnung getragen werden, andererseits sollten Prüfungen aber so allgemein wie möglich gehalten werden.

Auch hinsichtlich der Auswertung und Präsentation der Daten ist es schwierig, ein Mittelmaß zu finden. Sind die Daten unübersichtlich, ermüdet man während der Durchsicht, dauert eine Auswertung zu lange, würde sich mit der Zeit sicher eine „Prüfungsmüdigkeit“ einstellen.

Hier ist daher ein gutes Mittelmaß aus Effizienz und Genauigkeit wichtig.

Die Annahme, dass textbasierte Formate wesentlich leichter zu prüfen seien als binäre, hat sich in diesem Ausmaß nicht bestätigt. Teilweise war sogar das Gegenteil der Fall.

So unzugänglich die Binärformate beim Parsen sind, so schwierig ist es auch, sie unauffällig zu manipulieren.

Der Gedanke, ein PDF Dokument komplett gegen den PDF Standard gegen Abweichungen zu prüfen, hat sich sehr schnell wieder verflüchtigt. Alleine die Spezifikation zum Standard der Version 1.7 hat einen Umfang von ca. 1100 Seiten. Das wäre, auch für ein kommerzielles Unterfangen, ein Aufwand, der sich nicht lohnen würde, zumal es sich dabei ja nur um einen Standard handelt.

Effektiver waren hier schon die Versuche mit dem Metasploit-Framework, um gezielt nach Auswirkungen der Manipulationen zu suchen.

Die hier behandelten Formate sind zwar sehr häufig, aber in Wirklichkeit nur ein sehr kleiner Teil der in unserer modernen IT-Landschaft kursierenden Daten. Gerade im industriellen Bereich gibt es sehr viele spezielle Formate, die zwar nicht die große Masse betreffen, jedoch durch die hohe wirtschaftliche Potenz ihrer Anwendung lohnenswerte Ziele für Angriffe darstellen. Besonders hier sollte es Bestrebungen zur Früherkennung von Angriffen geben.

Die Anzahl der verschiedenen Formate, die im täglichen Gebrauch meist nur beiläufig Beachtung finden, wird in Zukunft sicher noch zunehmen. Das gilt auch für die Größe der Dateien, welche analog zur Leistungsfähigkeit von Hard- und Software weiterwachsen wird.

# Quellenverzeichnis

**„Das PDF Howto“:** Eine Seite von Christof Bürgi, zu finden unter

http://www.p2501.ch/pdf-howto/

Detaillierte Beschreibung des PE Standards:

<https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files>

Bonusmaterial: Outtakes (by Karin Postner)

So kann der Inhalt unabhängig von der eigentlichen Zielanwendung untersuchen und einen eingehender Blick auf Aufbau und enthaltene.

Unser Interfaces (…)

(…) die Realisierung von Transparanetz.

Man diese Formate auch Multibyte.

1. Reverse Engineering bezeichnet das Verfahren, ein erstelltes Produkt zurückzuentwickeln. [↑](#footnote-ref-1)
2. https://www.w3.org/Graphics/JPEG/jfif3.pdf [↑](#footnote-ref-2)
3. https://en.wikibooks.org/wiki/X86\_Disassembly/Windows\_Executable\_Files [↑](#footnote-ref-4)
4. Erstes „Hello World“ Beispiel aus „Das PDF Howto“ [↑](#footnote-ref-5)
5. https://de.wikipedia.org/wiki/American\_Standard\_Code\_for\_Information\_Interchange [↑](#footnote-ref-6)
6. https://tools.ietf.org/html/rfc3629 [↑](#footnote-ref-7)
7. https://de.wikipedia.org/wiki/Office\_Open\_XML

   https://msdn.microsoft.com/de-de/office/bb906068.aspx [↑](#footnote-ref-8)
8. https://de.wikipedia.org/wiki/ZIP-Dateiformat [↑](#footnote-ref-9)