**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, dass Dateien, egal welchen Ursprungs, unabhängig von ihrem Dateiformat in binärer Form einließt und eine Analyse erlaubt ohne Gefahr zu laufen schadhafte Teile auszuführen.

**Vorwort**

Dateien in gespeicherter Form dienen primär zur Verarbeitung durch Software. Vor allem das Wiedergeben 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 so genannter Scripts, also Anweisungen die in herkömmlichen Dateien versteckt werden und Schadprogramme auf dem Zielsystem installieren, oder selber 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 öfters nicht in Form von darstellbaren Zeichen enthalten sind oder einfach der Umfang der Datei für eine konventionelle Analyse zu groß ist.

Der Ansatz der meisten konventionellen Virenscanner besteht darin, Dateien nach Signaturen bekannter Schädlinge zu durchsuchen. Was in den meisten Fällen, und vor allem für den Heimanwender, durchaus ausreichend ist, ist für andere Fälle bei Weitem zu wenig. Gerade bei „lohnenswerten Zielen“ wie Banken, staatlichen Einrichtungen oder größeren Firmen 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, das nicht gefährlich sein kann und lenkt seinen Fokus auf die 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 4](#_Toc480588367)

[Textdateien 4](#_Toc480588368)

[Binärdateien 5](#_Toc480588369)

[Executables 8](#_Toc480588370)

[Byte Order 9](#_Toc480588371)

[Container 9](#_Toc480588372)

[Analysieren mit dem „Analyzer“ 9](#_Toc480588373)

[Eine Sache der Interpretation 10](#_Toc480588374)

[reinterpret\_cast und memcpy 12](#_Toc480588375)

[RGB Farben 12](#_Toc480588376)

[Opcodes 12](#_Toc480588377)

[Dateiformate 13](#_Toc480588378)

[Binärdateien 15](#_Toc480588379)

[Bitmap 15](#_Toc480588380)

[Quellenverzeichnis 17](#_Toc480588381)

# 

# Grundlagen

Generell kann man zwischen binär geschrieben Daten und Daten im Textformat unterscheiden. Allerdings ist es in vielen Fällen auch so, dass eine Mischform vorliegt. Dateien die mehrere, eventuell auch unterschiedliche, Dateien vereinen werden in dieser Arbeit als Container bezeichnet.

## Textdateien

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

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 und einige nicht druckbare Zeichen, die meist als Steuerzeichen für Anwendungen verwendet werden, wie zum Beispiel ein Zeilenumbruch, Tabulator, … .

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[[1]](#footnote-1)

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

Der weltweite De-facto-Standard ist die UTF-8[[2]](#footnote-2) Codierung, vielfach auch als Unicode bezeichnet.

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

|  |  |
| --- | --- |
| (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 |

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

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

Bei der Analyse unbekannter Dateiformate kann man z.B. auf dieses Muster prüfen um herauszufinden um welchen Zeichensatz es sich handelt!

## Binärdateien

Anders als die Textformate 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 so genannte „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 sehen, 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 Bytes in Form von Characters, als ein Unicode Character oder als short interpretiert werden.

Hinzu kommt, dass z.B. ein Byte bzw. char, nicht zwangsläufig als Zeichen verwendet wird, sondern auch bitweise verwendet werden kann um so genannte Flags zu speichern und dann ist es natürlich so, dass die obigen Typen, bis auf die Gleitkommawerte, vorzeichenbehaftet sein können.

Der große Vorteil im Speichern von Daten in binärer Form ergibt sich durch den Platzbedarf.

Muss 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, einmal „asInt“ und einmal „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);

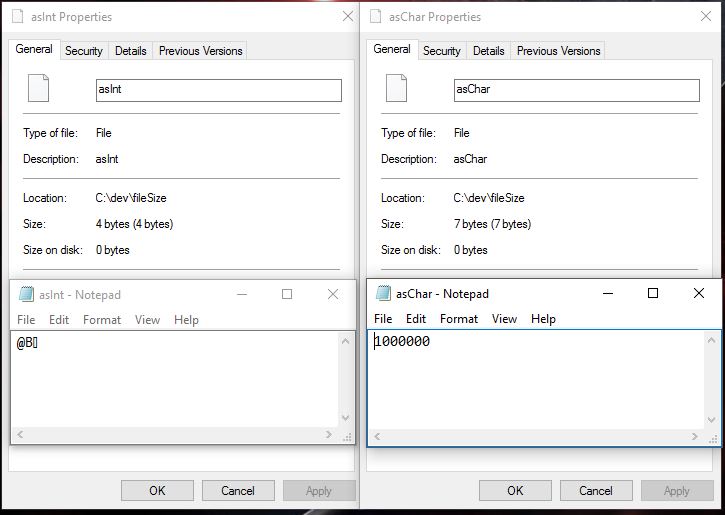
ofChar.flush();

return 0;

}

Das Lesen der Datei würde nun so funktionieren, dass man die jeweiligen Bytes in ihre gewünschten variablen kopiert. Dazu muss man allerdings wissen, dass z.B. 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 sehen wir die entsprechenden Dateigrößen. 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 uns bei der Analyse vor eine große Herausforderung.

Eine Möglichkeit ist, sich an veröffentlichte Standards zu halten, die definieren wie die Datei aufgebaut ist. Stehen diese Standards nicht zur Verfügung, kann man sich mit den sehr zeitraubenden Techniken des Revers Engineerings[[3]](#footnote-3) helfen.

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

struct JpegHeader

{

char SOI[2]; //Start of Image Marker

char APP0[2]; //Use Marker

char Length[2]; //File Length

char Identifier[5]; //Id String

char Version[2]; //Format Revision

char Units; //Units for the X and Y densities

char Xdensity[2]; //Horizontal pixel density

char Ydensity[2]; //Vertical pixel density

char XThumbnail; //Thumbnail horizontal pixel count

char YThumbnail; //Thumbnail vertical pixel count

};

Der Header einer JPEG Datei als C++ Struktur[[4]](#footnote-4)

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

### Executables

Eine Form der Binärdaten welche genauere Betrachtung verdient sind ausführbare Dateien. Sie liegen in einer Form vor, die vom System in den Speicher geladen und abgearbeitet werden kann, also alle nativen Programme die auf dem Rechner laufen können.

In Windows Systemen erkennt man diese Dateien an den Endungen .exe, .dll, .lib oder .sys.

Sie sind so aufgebaut, dass sie eine Startkennung haben, die abhängig vom Betriebssystem ist und in weiterer Folge natürlich das eigentliche Programm, das von der jeweiligen Hardware Architektur verwendet wird.

MS Windows benutzt für ausführbare Dateien das Common Object Fileformat (COFF).

Offset Size Field Description

------ ---- ------------------- -----------------------------------------------

0 2 Machine Number identifying type of target machine.

2 2 Number of Sections Number of sections; indicates size of the

Section Table, which immediately follows

the headers.

4 4 Time/Date Stamp Time and date the file was created.

8 4 Pointer to Symbol Offset, within the COFF file, of the symbol

Table table.

12 4 Number of Symbols Number of entries in the symbol table.

This data can be used in locating the

string table, which immediately follows

the symbol table.

16 2 Optional Header Size of the optional header, which is

Size included for executable files but not

object files. An object file should have a

value of 0 here.

18 2 Characteristics Flags indicating attributes of the file.

COFF-Header[[5]](#footnote-5)

Die Programmlogik selber ist in Maschinensprache, in so genanntem „Opcode“ (operation code) geschrieben.

Bei der Analyse geht es darum, die Dateien auf die Evidenz von Opcodes zu prüfen. Ein gängiges Verfahren zum Einschleusen von Schadsoftware besteht darin, Teile dieser Programme in offensichtlich ungefährlichen Dateien zu übermitteln. Die Zielanwendung ignoriert diese Teile, weil sie sie nicht interpretieren kann und so bleiben sie meist unentdeckt.

### Byte Order

Eine Sache mit der man noch konfrontiert werden könnte und die hier deshalb kurz erwähnt werden sollte ist die Byte Order.

Diese legt fest, in welcher Reihenfolge die Bytes integraler Werte im System gespeichert ist.

Big-Endian setzt das höchstwertige Byte an die Anfangsadresse, also der kleinsten Speicheradresse, Little-Endian hingegen setzt das niederwertigste Byte an die Anfangsadresse.

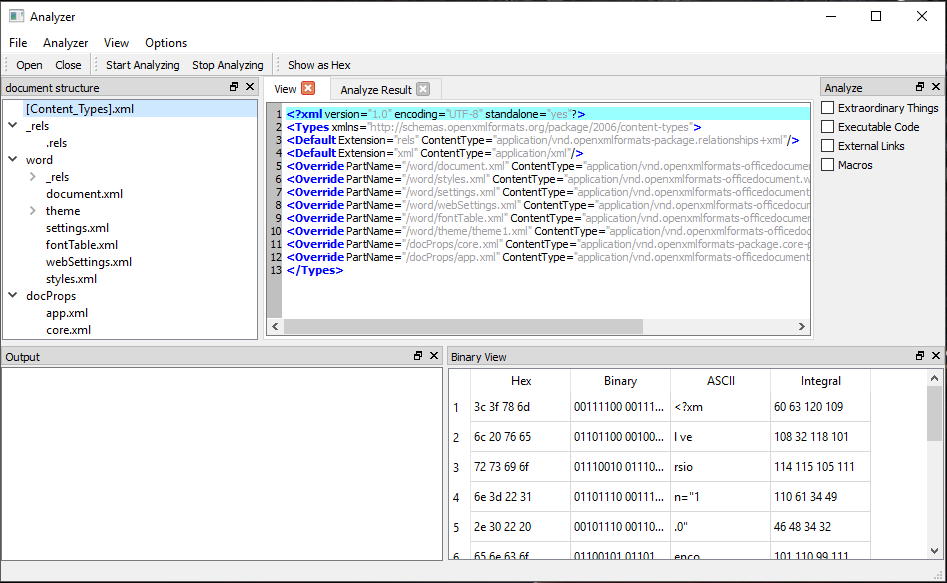
Die meisten PCs verwenden als Basis die 8086er Assembly Struktur, welche Little-Endian benutzt, somit wird das, soweit es diese Arbeit betrifft, nicht zu Problem werden. Achtgeben sollte man jedoch insbesondere im industriellen Umfeld, da vor allem für maschinennahe Steuerungen gerne Big-Endian verwendet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

## Container

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. In diesem Fall handelt es sich um das **Open Office XML** Format[[6]](#footnote-6), das von der Firma Microsoft© entwickelt wurde. Es beinhaltet im Prinzip 2 Arten von Dateien, die als ZIP-Archiv[[7]](#footnote-7) zusammengefügt sind. So genannte „Parts“, die den Inhalt des Dokuments enthalten, und „Items“, die beschreiben wie der Inhalt dargestellt wird.

Ein anderes Format das weiterer Folge auch noch gezeigt wird, ist PDF (Portable Document Format).

# Analysieren mit dem „Analyzer“

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

Der „Analyzer“ beim Öffnen eines MS Word Dokuments

Ziel des Analyzers ist es, alle Inhalte einer Datei einzulesen und unbekannte Teile, Teile die Unregelmäßigkeiten enthalten und Teile die 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 Bytes in verschienen Interpretationen betrachten können.

Technische Eckdaten:

Sprache: C++

Bibliotheken: Google Test Framework (gtest)

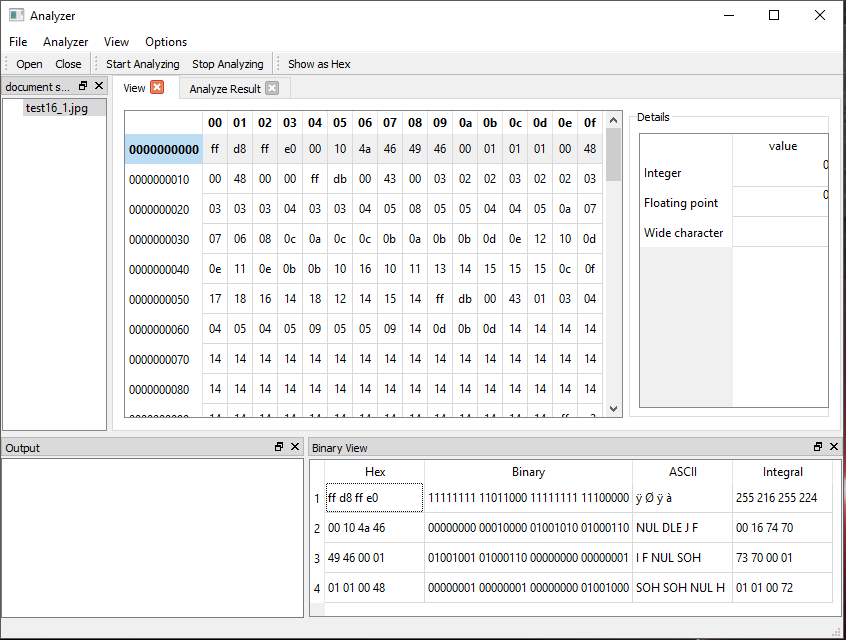
Qt (ausschließlich für die graphische Oberfläche)

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, braucht man eine Möglichkeit die Daten anders lesbar zu machen. Diese Aufgabe kommt dem „Hex-Editor zu.



Öffnen der Datei einer .jpg Datei 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 aber eher weniger vorstellen kann, werden die selektierten Werte in verschiedenen Interpretationen angezeigt.

Im Fenster unten rechts ist eine Standartansicht. Alle selektierten Bytes 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 Bytes angezeigt.

Eine Interpretation, bzw. Reinterpretation von Daten nennt sich Typumwandlung, oder auf Englisch Typecasting.

Typumwandlungen für primitive Datentypen wie int, double oder wchar\_t hat die Programmiersprache C++, und natürlich alle anderen Hochsprachen, standartmäßig und diese kommen im Analyzer auch zur Anwendung. Umwandlungen zu komplexeren Typen müssen allerdings implementiert werden.

Wenn von einer „Umwandlung“ die Rede ist, heißt das nicht, dass die Daten selber verändert werden, es wird lediglich die Interpretation geändert, die „Bits und Bytes“ bleiben gleich.

## reinterpret\_cast und memcpy

Zwei Operationen kommen bei Typumwandlungen in C++ sehr oft vor. Das ist zum ersten die durch den C++ Standard definierte Funktion reinterpret\_cast und 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 erst ein Speicher angelegt und diesen wird das Array dann physikalisch hineinkopiert.

char buffer[2];

buffer[0] = 0;

buffer[1] = 0;

int16\_t value;

memcpy(&value, buffer, 2);

## RGB Farben

Das RGB Frabmodel ist ein additives Farbmodel. Das heißt, dass die jeweiligen Farbtöne von rot, grün und blau übereinandergelegt werden und sich aus dem Resultat 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 Unser Interfaces zu finden sind würde das so aussehen:

[255][0][0] … ergibt ein sattes rot

[0][0][0] … ist schwarz

…

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

## Opcodes

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

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-BitOperation, 1 = 16-Bit-Operation

2.Byte:

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

reg… Registeroperand

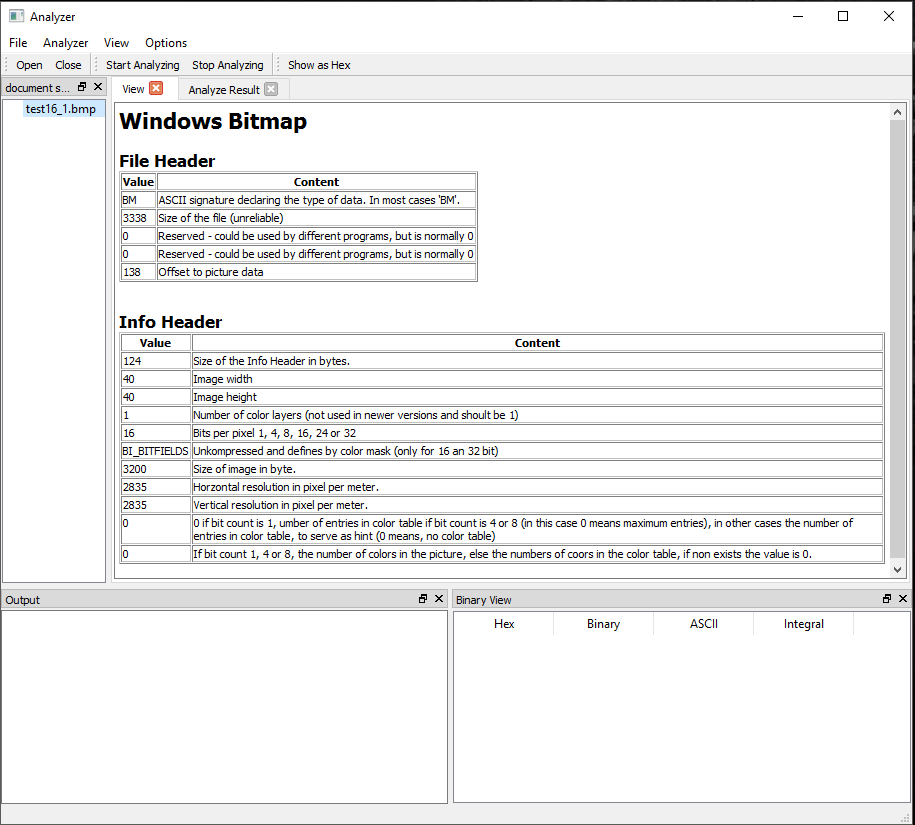
mod-r/m…

[…]

# Dateiformate

Im letzten Kapitel ging es darum wie Daten interpretiert werden können. Ob ein Byte jetzt 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 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 des 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 verschicken. Falls am Zielrechner kein Virenscanner aktiv ist, würde die Datei nur 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 es bleibt eine voll Funktionsfähige Anwendung, die, sobald sie in den Speicher geladen ist, auch ausgeführt wird.

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

Es ist zwar sehr selten, dass diese Funktion bei einzelnen Dateien notwendig ist, aber in Containern, vor allem bei PDF-Dateien, in denen es keine Unterdateien, sondern nur binäre „Streams“ gibt, kommt diese Funktion sehr wohl zum Tragen.

## 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 gleich 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 dem 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 indizierten Farben kann das sehr einfach berechnet werden indem man die Anzahl der Bildpunkte mal 3 Byte rechnet. 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 zum Beispiel 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 machen, 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 nächsten paar Seiten ausreichend.

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

Struktur einer PE Datei[[8]](#footnote-8)

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 zwei 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.

struct COFFHeader

{

short Machine;

short NumbSections;

long TimeDateStamp;

long PointerToSymbolTable;

long NumOfSymbols;

short SizeOfOptionalHeader;

short Characteristics;

};

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

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

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[];

};

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.

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

#### 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. Dabei überprüft man die Daten auf das Vorkommen der Sequenz „MZ“. Findet man diese Signatur, kann man die Bytes am Offset des „PEOffset“-Parameters prüfen. Ergibt dieser Wert einen Sinn und an der Stelle des „PEOffset“ Offsets findet man die Werte für den PE-Header, ist die Wahrscheinlichkeit, dass es sich um eine Ausführbare Datei handelt schon 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 würde darin bestehen häufig vorkommende Muster von Opcodes zu suchen, also logische Sequenzen.

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, doch kann das Ergebnis sehr unterschiedlich aussehen, da die beiden Hochsprachen komplexere Aufgaben sicher anders darstellen als selber implementierte Assembler-Logik.

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

Eingehende und verwertbare Versuche in diese Richtung sind im Rahmen dieser Arbeit leider nicht möglich.

## Textdateien

# 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

1. https://de.wikipedia.org/wiki/American\_Standard\_Code\_for\_Information\_Interchange [↑](#footnote-ref-1)
2. https://tools.ietf.org/html/rfc3629 [↑](#footnote-ref-2)
3. Revers Engineering bezeichnet das Verfahren ein erstelltes Produkt zurück zu entwickeln. [↑](#footnote-ref-3)
4. https://www.w3.org/Graphics/JPEG/jfif3.pdf [↑](#footnote-ref-4)
5. https://support.microsoft.com/de-de/kb/121460 [↑](#footnote-ref-5)
6. https://de.wikipedia.org/wiki/Office\_Open\_XML

   https://msdn.microsoft.com/de-de/office/bb906068.aspx [↑](#footnote-ref-6)
7. https://de.wikipedia.org/wiki/ZIP-Dateiformat [↑](#footnote-ref-7)
8. https://en.wikibooks.org/wiki/X86\_Disassembly/Windows\_Executable\_Files [↑](#footnote-ref-8)