1 Introduction

The file format is an archive with a folder structure and various files. The file archive can be protected with a password. Verification of the encryption factors takes some time. Thus, a four letter password can already be considered safe because it takes a long time to try out all the possibilities. The protected file archive can be created with the "FileProtected" program. The file extension is called ".pdata" (Protected Data). For multiple files, the extensions are specified with ".001" to ".254".

2 Value types

Type	Description	Area
INT8	8Bit with sign	-128 to 127
INT16	16Bit with sign	-32.768 to 32.767
INT32	32Bit with sign	-2.147.483.648 to 2.147.483.647
INT64	64Bit with sign	-9.223.372.036.854.775.808 to 9.223.372.036.854.775.807
BYTE	8Bit unsigned	0 to 255
UINT16	16Bit unsigned	0 to 65.535
UINT32	32Bit unsigned	0 to 4.294.967.295
UINT64	64Bit unsigned	0 to 18.446.744.073.709.551.615
CHAR	8Bit character	0 to 255
WCHAR	16Bit character	0 to 65.535
FLOAT	32Bit floating point	± 1.5e-45 to ± 3.4e38
DOUBLE	64Bit floating point	± 5.0e-324 to ± 1.7e308
MEMORY	Memory in bytes	
...[]	Array	see section 2.1
?	Different values	see section 2.2
*	Coded value	see section 2.3
...	Next table	see section 2.4
-> {	Start of the loop	see section 2.5
} <-	End of the loop	see section 2.5

Table 2: Value types

2.1 Array

The set consists of a specific value type. The number of the quantity is detailed in the information and is usually the previous format value.

Example: `INT16[] = { INT16, INT16, INT16, INT16, ... }.`

2.2 Different values

There may be different types of values in the format. The dependency of the type is described in the information.

2.3 Coded value

The value must be decrypted with a specific XOR operation. The information describes how the operation is to be performed.

2.4 Next table

The file format is displayed further in the section and the table specified.

2.5 The loop

In a loop, the format is repeatedly run. The number of passages is specified in detail in the information and is usually a previous value.

3 Description

3.1 File format

Type	Name	Description	Info
UINT32	IDNumber	The file can have the ID number (0x54414450).	4.1
UINT32	LoopMax	The number of loop passes for decryption.	4.2
BYTE	PassVersion	The version number and a confirmation of the password.	4.3
BYTE	FileCount	The number of archive files.	4.4
BYTE*	FactorCount	The number of factors for decryption.	4.5
BYTE[]*	FactorMemory	The factors for the decryption algorithm.	4.6
BYTE[]*	ReferenceMemory	A array for verifying the factors.	4.7
UINT32*	CodeValueCount	The number of decryption values.	4.8
UINT32*	CodePageCount	The number of extra decryption values.	4.9
...		3.2 Archive format, table 3.2	

Table 3.1: File format

3.2 Archive format

Type	Name	Description	Info
-> {	EndOfFile		5.1
UINT16*	HeaderSize	The format size without file size in bytes.	5.2
BYTE*	HeaderFlag	Specifies the formattype and the merge values.	5.3
...		3.3 Archive folder format or 3.4 Archive file format	5.4
} <-	EndOfFile		5.1

Table 3.2: Archive format

3.3 Archive folder format

Type	Name	Description	Info
?*	FolderIndex	The index for the parent folder.	6.1
?*	FolderNameSize	The size of the name store in bytes.	6.2
?[]*	FolderName	The name of the folder.	6.3

Table 3.3: Archive folder format

3.4 Archive file format

Type	Name	Description	Info
?*	FolderIndex	The index for the file folder.	7.1
?*	FileSize	The size of the file in bytes.	7.2
?*	FileNameSize	The size of the name store in bytes.	7.3
?[]*	FileName	The name and extension of the file.	7.4
MEMORY*	FileMemory	The file memory in the specified size.	7.5

Table 3.4: Archive file format

4 Information about the file format

4.1 Identification Number

The identification number identifies the file format. The number can also be represented with 4 letters (PDAT: Protected Data). The archive can also be created without a file identification. Then the value must be generated from a random number generator and can accept any value of the type (UINT32). This has effect from the values in Sections 4.2 and 4.3.

4.2 Maximum number of loop passes for decryption

The value specifies the maximum number of passes for the loop in the decryption algorithm. The value can not be less than 100,000 and greater than 2,147,483,647. The number is between 2 to 4 times greater than the actual loop value. If an incorrect password has been entered, the system checks only to this number. If the file archive has no file identification (4.1), the value is undefined and must be generated with a random number generator. The maximum number of passages is thus 2,477,483,647.

4.3 File version and password input

The version number is always 0 for this format description. If the highest bit (0x80) is set, a password is expected. If the file archive has no file identification (4.1), the value is undefined and must be generated with a random number generator. Thus, the user determines whether or not to enter a password.

4.3.1 Password entry

The password (PassLetterArray) must be converted to a value of bytes. The maximum size of the array (PassArray) is 515 bytes. If no password is entered, the value 0 is entered. The current character is added with a byte value (PassSpace) each time the loop is looped. If all characters were read by the password, this value is accepted and the index (PassIndex) is reset to 0 again. If the current character from the password is less than 256, it is simply transferred to the value array (PassArray). If the character is larger than 255, the first byte and then the higher byte are inserted into the array.

```
UInt16[] PassLetterArray; //user input, can all ASCII letter
Byte[] PassArray = new Byte[0];

if(PassLetterArray != null && PassLetterArray.Length > 0) {
  PassArray = new Byte[515]; //size: 2 * 256 + 1 + 2

  Byte PassSpace = 0;
  Int32 PassIndex = 0;

  for(Int32 i = 0; i <= 513; i++) { //use size: 2 * 256 + 1
    PassSpace += (Byte) PassLetterArray[PassIndex];

    if(PassLetterArray[PassIndex] > 255) {
      PassArray[i++] = (Byte) PassLetterArray[PassIndex];
      PassArray[i] = (Byte) (PassLetterArray[PassIndex++] / 0x0100);
    } else {
      PassArray[i] = (Byte) PassLetterArray[PassIndex++];
    }

    if(PassIndex == PassLetterArray.Length) {
      PassArray[++i] = PassSpace;
      PassIndex = 0;
    }
  }
}
```

Program 4.3.1: Password entry

4.3.2 Password determination

The identification number and the byte value (PassVersion) are read from the file. If the identification is present, the first check is whether the version is 1. Then a check is made as to whether no password has been used.

```
UInt32 IDNumber;  //read from file
Byte PassVersion; //read from file
Byte[] PassArray; //see Program 4.3.1

Boolean UseFileIdentifier = IDNumber == 0x54414450 ? true : false;
Boolean UsePassword = PassArray.Length > 0 ? true : false;

if(UseFileIdentifier) {
  if((PassVersion & 0x7F) != 1) { //wrong version }

  if((PassVersion & 0x80) != 0x80) { //no password
    UsePassword = false;
  }
}
```

Program 4.3.2: Password determination

4.4 Number of archive files

An archive can be divided into several files. The value specifies the number of files. If the file archive does not contain an identifier (4.1), the value must be created with a random number generator.

Example: File.pdata, File.001, File.002, ... , File.254

4.5 Number of factors

The factors are a array of byte values used for the decryption algorithm. The number must be generated with a random number generator. The highest bit (0x80) is not used because the number of factors can not be less than 128. If a password is used, the read value must be XOR linked with the first byte of the password array. The result is then the number.

```
Byte FactorCount;    //read from file
Byte[] PassArray;    //see Program 4.3.1
Boolean UsePassword; //see Program 4.3.2

if(UsePassword) FactorCount ^= PassArray[0]; //XOR

FactorCount = (Byte) (FactorCount | 0x80);
```

Program 4.5: Number of factors

4.6 Factor memory

The factors are a array of byte values used for the decryption algorithm. The values were generated with a random number generator. The number of factors (4.5) is expressed in bytes. If a password is used, the read values must be XOR linked with bytes of the password array.

```
Byte[] PassArray;    //see Program 4.3.1
Boolean UsePassword; //see Program 4.3.2
Byte FactorCount;    //see Program 4.5
Byte[] FactorMemory; //read from file

if(UsePassword) {
  for(Int32 i = 0; i < FactorCount; i++)
    FactorMemory[i] ^= PassArray[i + 1]; //XOR
}
```

Program 4.6: Factor memory

4.7 Verification memory

The memory is a array of byte values which are used as reference values for verifying the file format. The number of values (4.5) is identical to the factors. If a password is used, the read values must be XOR linked with bytes of the password array.

```
Byte[] PassArray;       //see Program 4.3.1
Boolean UsePassword;    //see Program 4.3.2
Byte FactorCount;       //see Program 4.5
Byte[] ReferenceMemory; //read from file

if(UsePassword) {
  for(Int32 i = 0; i < FactorCount; i++)
    ReferenceMemory[i] ^= PassArray[i + FactorCount + 1]; //XOR
}
```

Program 4.7.1: Verification memory

The verification of the file format is determined by the program 4.7.2. The decryption algorithm operates with the factors and the passages of the loops. The number of calculations takes a certain amount of time, ranging from one to two seconds. The value (CodeValue) and the index (CodeIndex) are reused later and are defined globally in the program.

```
UInt32 LoopMax;           //see Program 4.2
Byte FactorCount;         //see Program 4.5
Byte[] FactorMemory;      //see Program 4.6
Byte[] ReferenceMemory;   //see Program 4.7.1

this.CodeValue = 0; //global value: UInt64
this.CodeIndex = 1; //global value: UInt64

Int32 ReferenceIndex = 0;

for(UInt64 i = 1; i <= LoopMax + 1; i++) {
  for(UInt64 j = 1; j <= FactorCount; j++)
    this.CodeValue += (FactorMemory[j - 1] + (this.CodeValue + 1) * i) * j;

  this.CodeIndex++;
  this.CodeValue /= FactorCount;

  //equal reference values
  if((Byte) this.CodeValue == ReferenceMemory[ReferenceIndex]) {
    ReferenceIndex++;

    if(ReferenceIndex == FactorCount) {
      return; //file format: OK
    }
  } else {
    ReferenceIndex = 0;
  }
}

//LoopMax: bad file format
```

Program 4.7.2: Verification

The decryption memory is created with the program 4.7.3. The size (Count) of the array of values (CodeMemory) is different and is specified in sections 4.8 and 4.9. From the result (CodeValue) of the calculations, only the first 8 bits are written to the decryption memory. The resulting values are not periodic and do not follow a pattern. The value (CodeValue) and the index (CodeIndex) are reused and are defined globally in the sections. Both values were used in program 4.7.2 for the first time.

```
Byte FactorCount;    //see Program 4.5
Byte[] FactorMemory; //see Program 4.6

this.CodeValue; //global value: UInt64
this.CodeIndex; //global value: UInt64

UInt32 Count;   //various size

Byte[] CodeMemory = new Byte[(Int32) Count];

UInt32 Index = 0;

for(UInt64 i = this.CodeIndex; i < this.CodeIndex + Count; i++) {
  for(UInt64 j = 1; j <= FactorCount; j++)
     this.CodeValue += (FactorMemory[j - 1] + (this.CodeValue + 1) * i) * j;

  this.CodeValue /= FactorCount;

  CodeMemory[Index++] = (Byte) this.CodeValue;
}

this.CodeIndex += Index;
```

Program 4.7.3: Decryption memory

4.8 Number of decryption values

The number of decryption values can not be less than 500,000 and greater than 500,000,000. The corresponding memory from section 4.10.1 may only be created after determining the number of extra decryption values (4.9).

```
Byte[] CodeMemory; //create with Program 4.7.3, Count = 4
Byte[] ValueArray; //read from file

UInt32[] FactorArray = new UInt32[] { 1, 0x00000100, 0x00010000, 0x01000000 };

UInt32 CodeValueCount = 0; //result for memory count

for(Int32 i = 0; i < 4; i++) {
  ValueArray[i] ^= CodeMemory[i];
  CodeValueCount += FactorArray[i] * ValueArray[i];
}
```

Program 4.8: Determination of the number of decryption values

4.9 Number of extra decryption values

The number of extra decryption values can not be less than 500,000 and greater than 1,000,000 If the number is equal to 0, the extra memory (4.10.2) is not used. The memory may only be created after the simple decryption memory (4.10.1). The highest byte indicates the file version (4.3). The version is always specified because the archive may not have a file identification (4.1).

```
Byte[] CodeMemory; //create with Program 4.7.3, Count = 4
Byte[] ValueArray; //read from file

UInt32[] FactorArray = new UInt32[] { 1, 0x00000100, 0x00010000 };

UInt32 CodePageCount = 0; //result for memory count

Byte FileVersion = (Byte) (ValueArray[3] ^ CodeMemory[3]); //result for version

for(Int32 i = 0; i < 3; i++) {
  ValueArray[i] ^= CodeMemory[i];
  CodePageCount += FactorArray[i] * ValueArray[i];
}
```

Program 4.9: Determination of the number of extra decryption values

4.10 Decryption values

4.10.1 Simple decryption memory

The simple decryption memory will be used for all other endcodes. It is created with the number from section 4.8 and the program 4.7.3.

```
this.CodeValueMemory; //create with Program 4.7.3, Count = CodeValueCount (4.8)
```

If there is no extra decryption memory, the program 4.10.1 is used for the end coding of all further data. The file position (FilePosition) must be determined before reading the data. The original data is copied into the array (ByteArray) and then decrypted. The size (count) of the data array is different and is specified in archive format (section 3.2).

```
UInt64 FilePosition;      //file position before reading

UInt32 Count;             //size to read from file
Byte[] ByteArray;         //read from file

UInt32 CodeValueCount;    //Program 4.8

UInt32 ValuePosition = (UInt32) (FilePosition - FilePosition / CodeValueCount *
                                 CodeValueCount);

UInt32 ValueCount = CodeValueCount - ValuePosition;

UInt32 Index = 0;

while(true) {
  if(ValueCount > Count - Index) ValueCount = Count - Index;

  for(UInt32 i = 0; i < ValueCount; i++)
     ByteArray[Index++] ^= this.CodeValueMemory[ValuePosition++];

  if(Index == Count) break;

  ValuePosition = 0;
  ValueCount = CodeValueCount;
}
```

Program 4.10.1: Using the decryption memory

4.10.2 Extra decryption memory

The simple and extra decryption memory will be used for all further end coding. The memory is created with the number from section 4.9 and the program 4.7.3.

```
this.CodePageMemory; //create with Program 4.7.3, Count = CodePageCount (4.9)
```

If the extra decryption memory is present, the program 4.10.2 is used for the end coding of all further data. The file position (FilePosition) must be determined before reading the data. The original data is copied into the array (ByteArray) and then decrypted. The size (count) of the data array is different and is specified in archive format (section 3.2).

```
UInt64 FilePosition;      //file position before reading

UInt32 Count;             //size to read from file
Byte[] ByteArray;         //read from file

UInt32 CodeValueCount;    //Program 4.8
```

```
UInt32 PagePosition = (UInt32) (FilePosition / CodeValueCount);
UInt32 ValuePosition = (UInt32) (FilePosition - PagePosition * CodeValueCount);

UInt32 ValueCount = CodeValueCount - ValuePosition;

UInt32 Index = 0;

while(true) {
  if(ValueCount > Count - Index) ValueCount = Count - Index;

  Byte Codepage = this.CodePageMemory[PagePosition];

  for(UInt32 i = 0; i < ValueCount; i++)
     ByteArray[Index++] ^= (Byte) (FilePosition++ *
                                   this.CodeValueMemory[ValuePosition++] +
                                   Codepage);

  if(Index == Count) break;

  PagePosition++;

  ValuePosition = 0;
  ValueCount = CodeValueCount;
}
```

Program 4.10.2: Use of the two decryption memories

4.10.3 Example for reading the encrypted data

In the other format descriptions, the data must be read out decrypted. Because the decryption is defined as a array of bytes, the "BinaryReader" class can be used for this purpose. The Array (ByteArray) is generated in program sections 4.10.1 and 4.10.2. The number of bytes is determined by the size of the data to be read.

```
BinaryReader HeaderReader; //System.IO.BinaryReader
MemoryStream HeaderStream; //System.IO.MemoryStream

if(this.CodePageMemory == null) {
  UInt32 Count;      //size to read from file

  Byte[] ByteArray; //create with Program 4.10.1

  HeaderStream = new MemoryStream(ByteArray);
} else {
  UInt32 Count;      //size to read from file

  Byte[] ByteArray; //create with Program 4.10.2

  HeaderStream = new MemoryStream(ByteArray);
}

HeaderReader = new BinaryReader(HeaderStream);
```

Program 4.10.3: Read the encrypted data

5 Information about the archive format

The archive format must be decrypted with the program 4.10.3. It can contain folder or file information.

5.1 The loop

The archive format is read out until the file end is reached. If the end is reached before or not completely the file is faulty.

5.2 Size of the archive format

The size of the archive format in bytes, without this 16-bit value. If a file format is used, the file size is not contained in the archive size. The value can not be less than 4 or greater than 535 bytes.

5.3 States in archive format

The states are defined as bits. How the bits are used will be described later.

Name	Werte	Beschreibung
Folder	0x80	Ordner
FolderIndexInt32	0x40	Ordnernummer als 32Bit Wert
NameWChar	0x20	Zeichen im Namen als 16Bit Werte
NameSizeUInt16	0x10	Speichergröße vom Namen als 16Bit Wert
FileSizeUInt64	0x03	Angabe der Dateigröße mit einem 64Bit Wert
FileSizeUInt32	0x02	Angabe der Dateigröße mit einem 32Bit Wert
FileSizeUInt16	0x01	Angabe der Dateigröße mit einem 16Bit Wert
FileSizeByte	0x00	Angabe der Dateigröße mit einem 8Bit Wert

Table 5.3: States in archive format

5.4 Branch to the specific archive format

If the "Folder" state (Table 5.3) is present, the archive order format is used. Otherwise the archive file format.

```
BinaryReader HeaderReader; //create with program 4.10.3, Count = 2
UInt16 HeaderSize = HeaderReader.ReadUInt16();

HeaderReader; //create again, Count = HeaderSize

Byte HeaderFlags = HeaderReader.ReadByte();

if((HeaderFlags & 0x80) == 0x80) //Table 5.3: Folder
  //jump to folder format
else
  //jump to file format
```

Program 5.4: Branch to the specific archive format

6 Information about the archive folder format

6.1 Folder number

The folder number indicates the membership of the folder. If the value is -1, the folder will be used as the root folder, otherwise the index will be contained by the parent folder. All folders are numbered. The first folder has the number 0. If the "FolderIndexInt32" state (Table 5.3) is present, a 32Bit value is used, otherwise an 8Bit value.

```
BinaryReader HeaderReader; //see Program 5.4
Byte HeaderFlags;          //see Program 5.4
Int32 FolderIndex;

if((HeaderFlags & 0x40) == 0x40) //Table 5.3: FolderIndexInt32
  FolderIndex = HeaderReader.ReadInt32(); //Int32
else
  FolderIndex = HeaderReader.ReadSByte(); //Int8
```

Program 6.1: Folder number

6.2 Folder name size

The memory size of the folder name is specified in bytes. If the "NameSizeUInt16" state (Table 5.3) is included, a 16Bit value is used, otherwise an 8Bit value. The value can not be 0 or greater than 520 bytes.

```
BinaryReader HeaderReader; //see Program 5.4
Byte HeaderFlags;          //see Program 5.4

UInt16 NameSize;

if((HeaderFlags & 0x10) == 0x10) //Table 5.3: NameSizeUInt16
  NameSize = HeaderReader.ReadUInt16(); //UInt16
else
  NameSize = HeaderReader.ReadByte(); //UInt8
```

Program 6.2: Folder name size

6.3 Folder name

If the "NameWChar" state (Table 5.3) is present, 16Bit characters are used, otherwise 8Bit characters. The memory size is determined in section 6.2.

```
BinaryReader HeaderReader; //see Program 5.4
Byte HeaderFlags;          //see Program 5.4

UInt16 NameSize;           //see Program 6.2

String Name = String.Empty;

if((HeaderFlags & 0x20) == 0x20) { //Table 5.3: NameWChar
  for(UInt16 i = 0; i < NameSize / 2; i++)
    Name += (Char) HeaderReader.ReadUInt16(); //WChar
} else {
  for(UInt16 i = 0; i < NameSize; i++)
    Name += (Char) HeaderReader.ReadByte();   //Char
}
```

Program 6.3: Folder name

7 Information about the archive file format

7.1 Folder number

The number identifies the archive folder in which the file exists. All folders are numbered. The first folder has the number 0. If the "FolderIndexInt32" state (Table 5.3) is present, a 32Bit value (Int32) is used, otherwise an 8Bit value (Int8). The structure is identical to the program 6.1.

7.2 File size

The file size is in bytes. A file can be 0 in size. The state (Table 5.3) is and linked with the value 0x03. The result is a number from 0 to 3. In Table 5.3 the corresponding numbers are marked with "FileSizeByte", "FileSizeUInt16", "FileSizeUInt32" and "FileSizeUInt64". The file size must be read as either 8Bit, 16Bit, 32Bit or 64Bit value.

```
BinaryReader HeaderReader; //see Program 5.4
Byte HeaderFlags;          //see Program 5.4

UInt64 FileSize;
```

```
switch(HeaderFlags & 0x03) { //FileSizeMask = 0x03
  case 0: { //Table 5.3: FileSizeByte
    FileSize = HeaderReader.ReadByte(); //Byte
    break;
  }
  case 1: { //Table 5.3: FileSizeUInt16
    FileSize = HeaderReader.ReadUInt16(); //UInt16
    break;
  }
  case 2: { //Table 5.3: FileSizeUInt32
    FileSize = HeaderReader.ReadUInt32(); //UInt32
    break;
  }
  case 3: { //Table 5.3: FileSizeUInt64
    FileSize = HeaderReader.ReadUInt64(); //UInt64
    break;
  }
}
```

Program 7.2: File size

7.3 File name size

The file size is specified in bytes. If the "NameSizeUInt16" state (Table 5.3) is included, a 16Bit value is used, otherwise an 8Bit value. The value can not be 0 or greater than 520 bytes. The structure is identical to the program 6.2.

7.4 File name

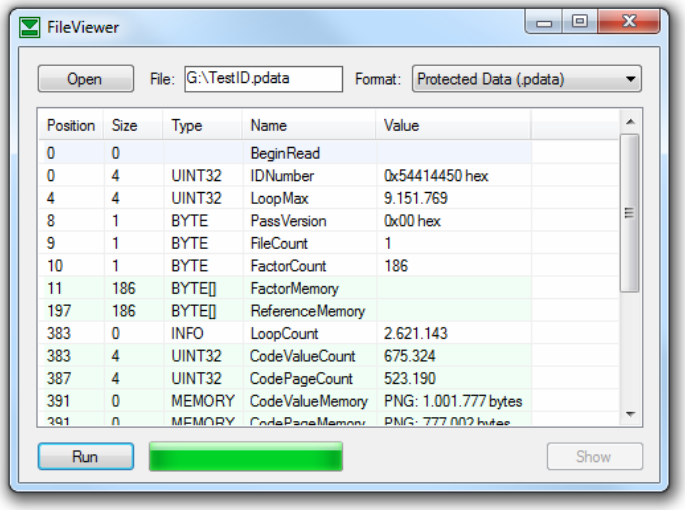
If the "NameWChar" state (Table 5.3) is present, 16Bit characters are used, otherwise 8Bit characters. The memory size is determined in section 7.3. The design is identical to the program 6.3.

7.5 File memory

The size of the file storage is determined in section 7.2. The data can be read out using program 4.10.1 or 4.10.2. It is recommended to decrypt the files in several parts. For example, with a memory size of 66000 bytes.

8 Program for reading the file format

On the PanotiSoft website, there is a test program under technical documents, with which the file format can be read out in a structured manner. In addition, the program code can also be downloaded. The program was written under Visual Studio 2008 with the programming language C#.



Program: FileViewerX64.zip
FileViewerX32.zip

Project file: FileViewerCode.zip

Description: FileViewer.pdf

FileViewerCode:

Format file: FileViewerFormat.cs
Format class: FileViewerProtectedData