unknownproject

Quick analysis of the game data file

September 21, 2016

Fuck knows what prompted me to write this article.

It's no secret that many games use custom binary formats to store many files in their primary or compressed form. Naturally, during loading of any game, everything necessary from these binaries is read and unloaded into RAM.

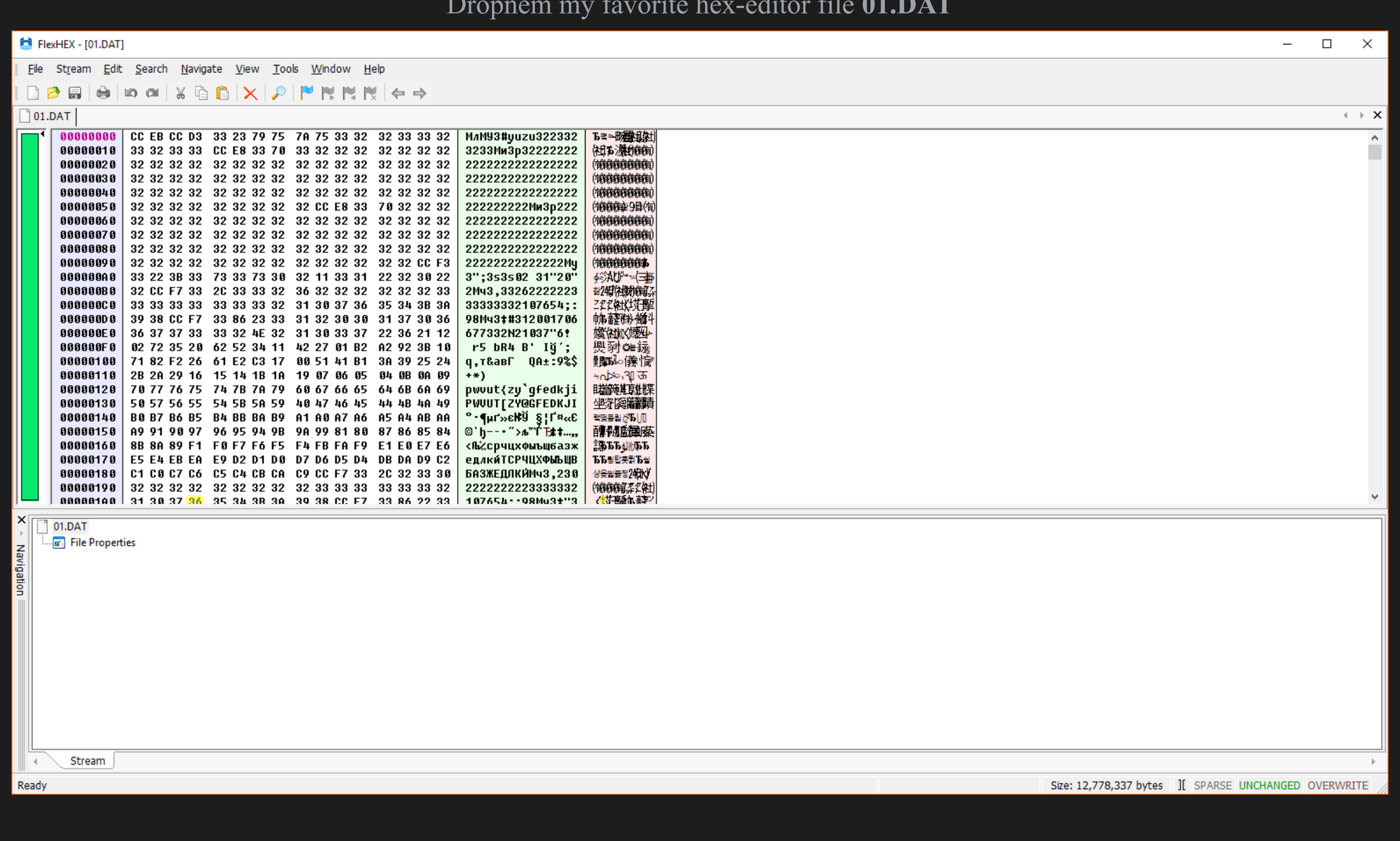
But what if the file of an unknown format is so much that neither TOC nor formatted signatures can be found in it either at the beginning or at the end?

It's very simple.

Any encrypted or compressed file undergoes preliminary decryption / decompression in RAM. This happens due to calls of a certain number of functions in the executable file or game script itself, but we will not reverse the executable file. It will be enough to analyze the binary format in a particular case.

Take the demo version of X - Beyond The Frontier, released back in 1999.

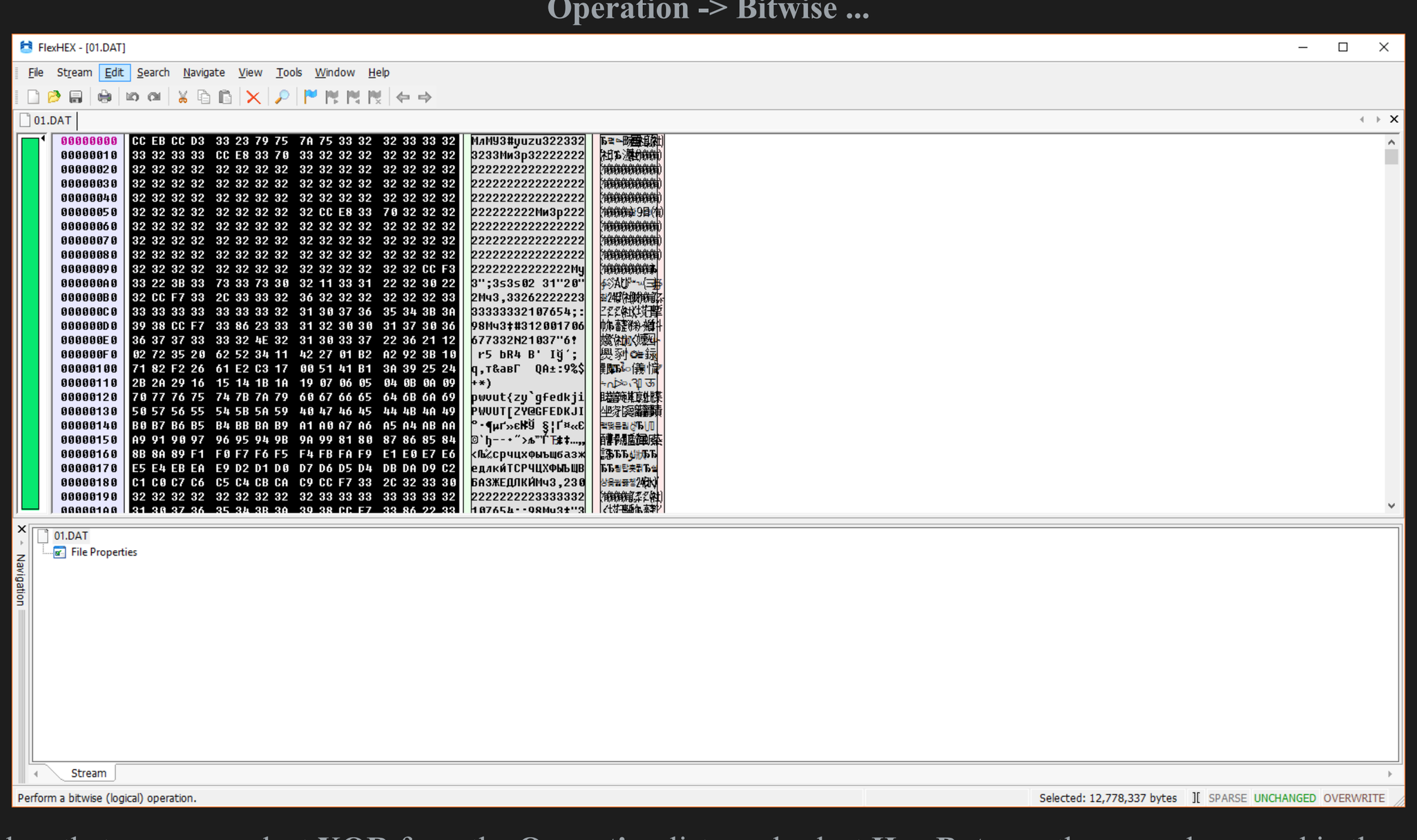
Dropnem my favorite hex-editor file **01.DAT**



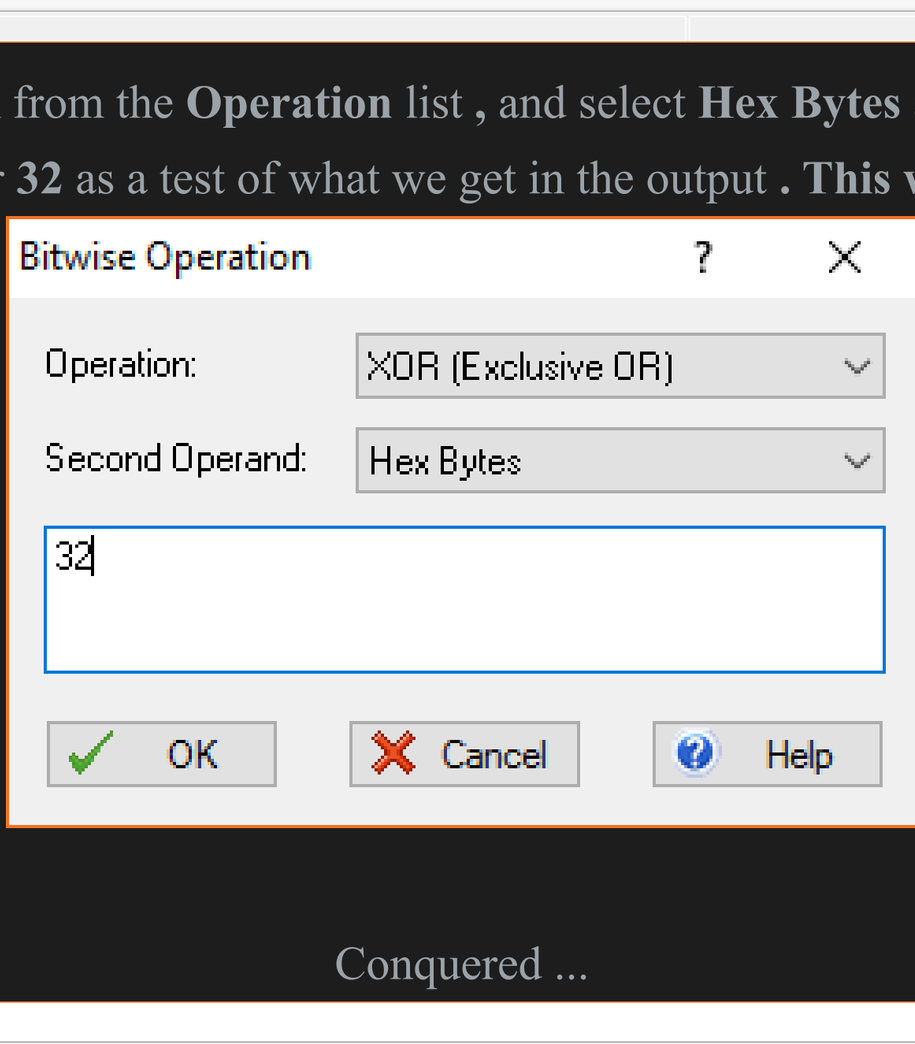
You can immediately notice that there are duplicate bytes in the file, such as **32h** and less common **33h** .

Practice shows that game developers are not particularly bothered with encryption, so we have before us with a probability of 80% a bit XOR operation, which is quite common in various game formats and not only.

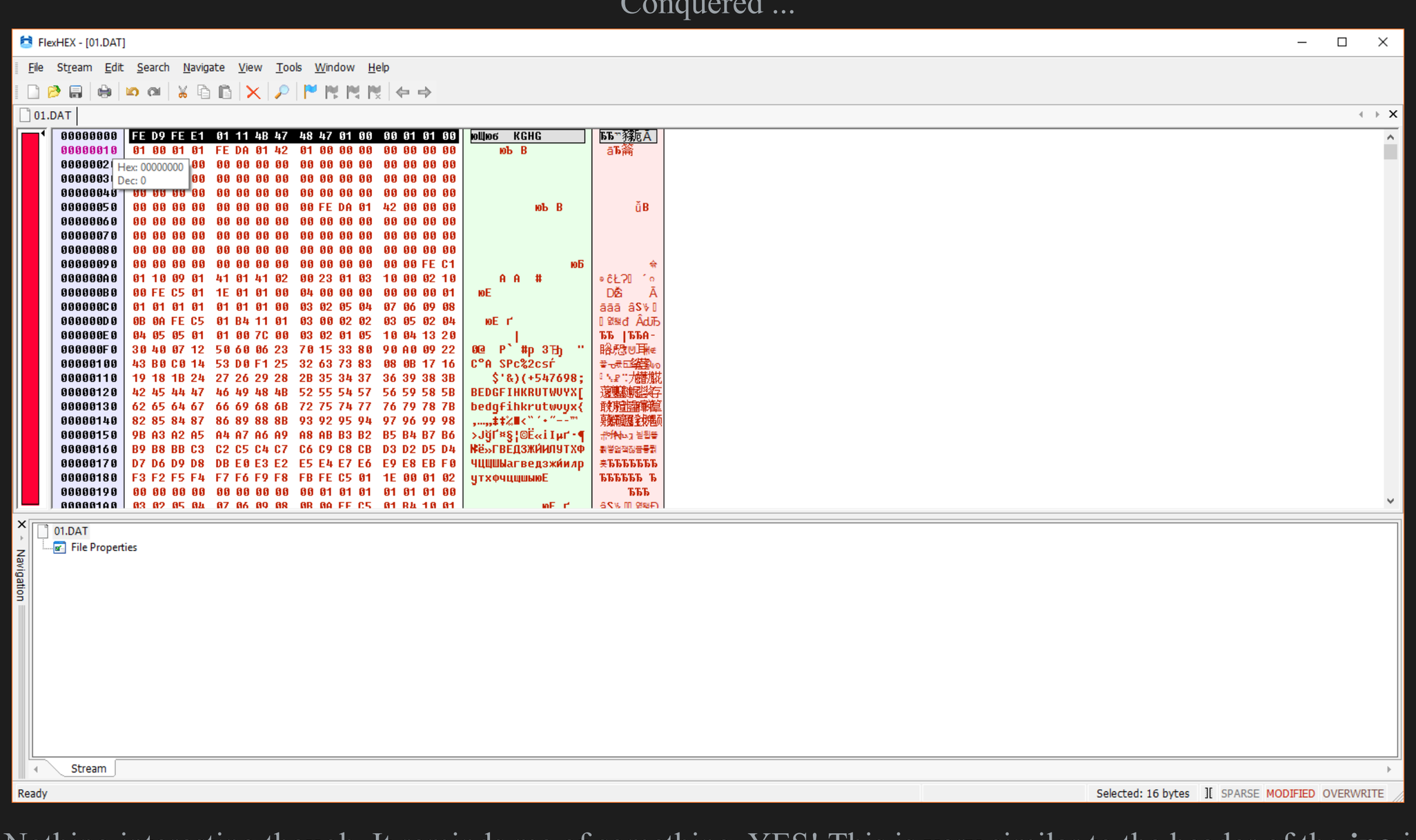
Let's check , select all the bytes with **Ctrl + A** and go to the **Edit** menu and then select **Operation -> Bitwise ...**



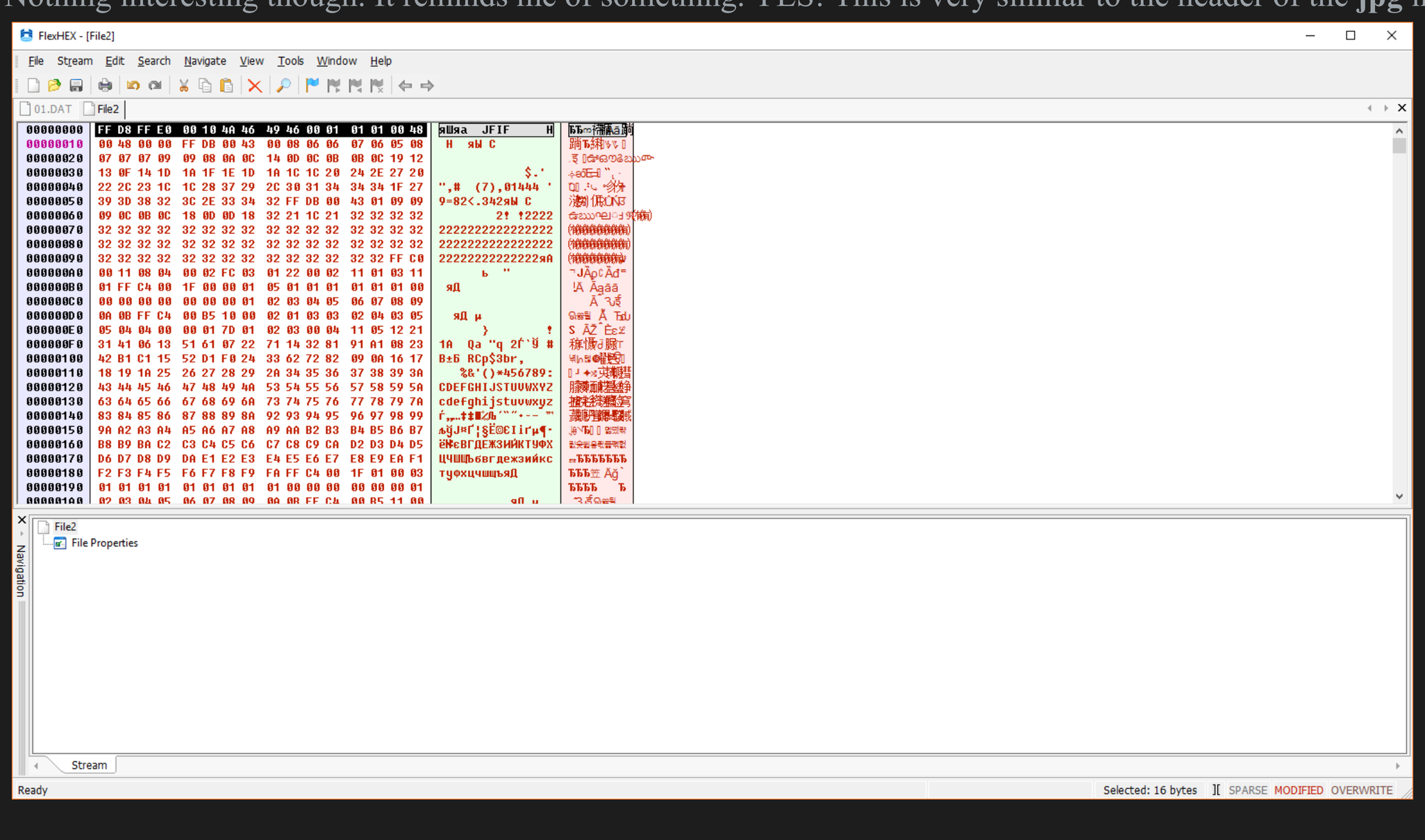
In the window that appears, select **XOR** from the **Operation** list , and select **Hex Bytes** as the second operand in the second list . In the bottom field, we enter **32** as a test of what we get in the output . **This will be our second operand.**



Conquered ...



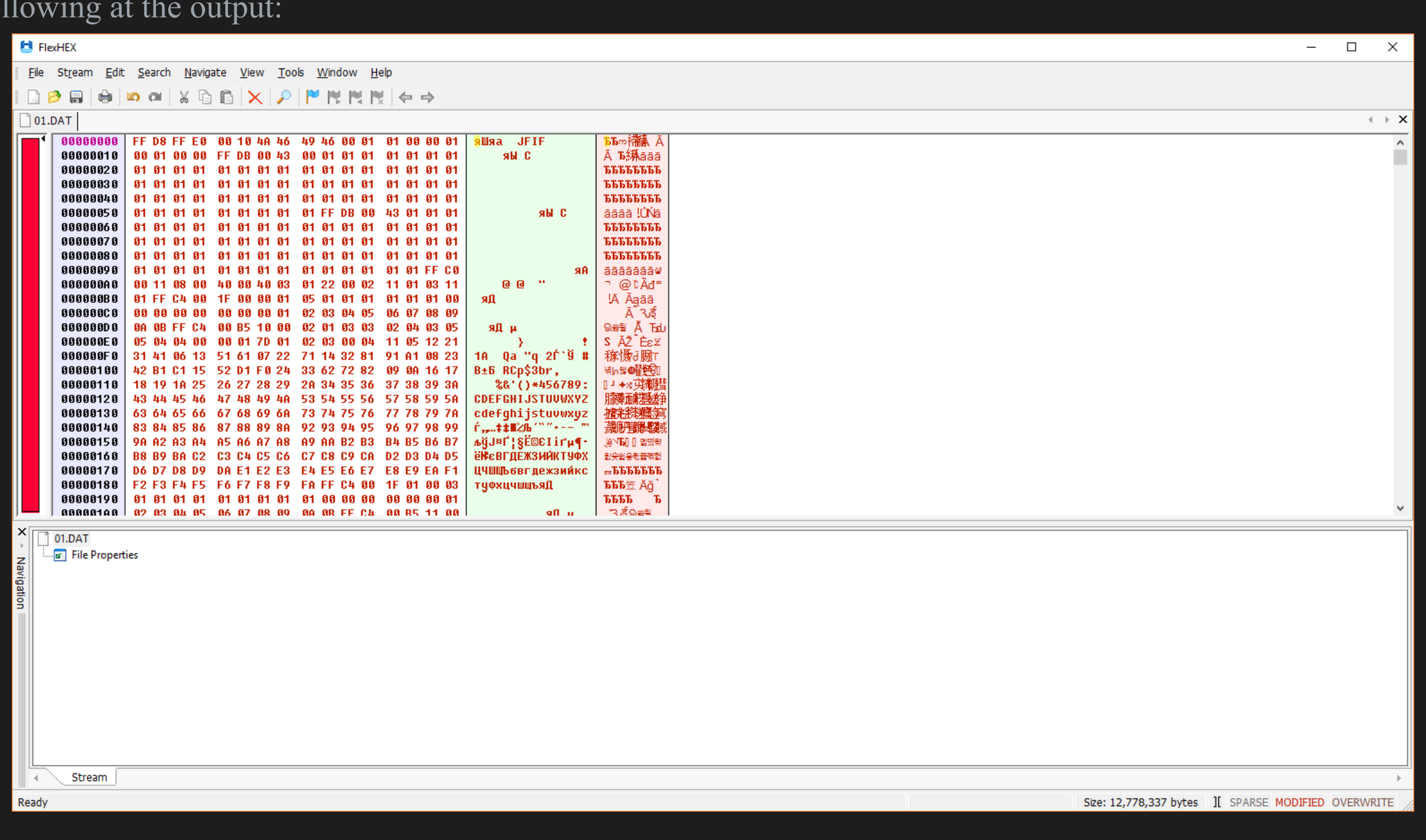
Hmm. Nothing interesting though. It reminds me of something. YES! This is very similar to the header of the **jpg** images.



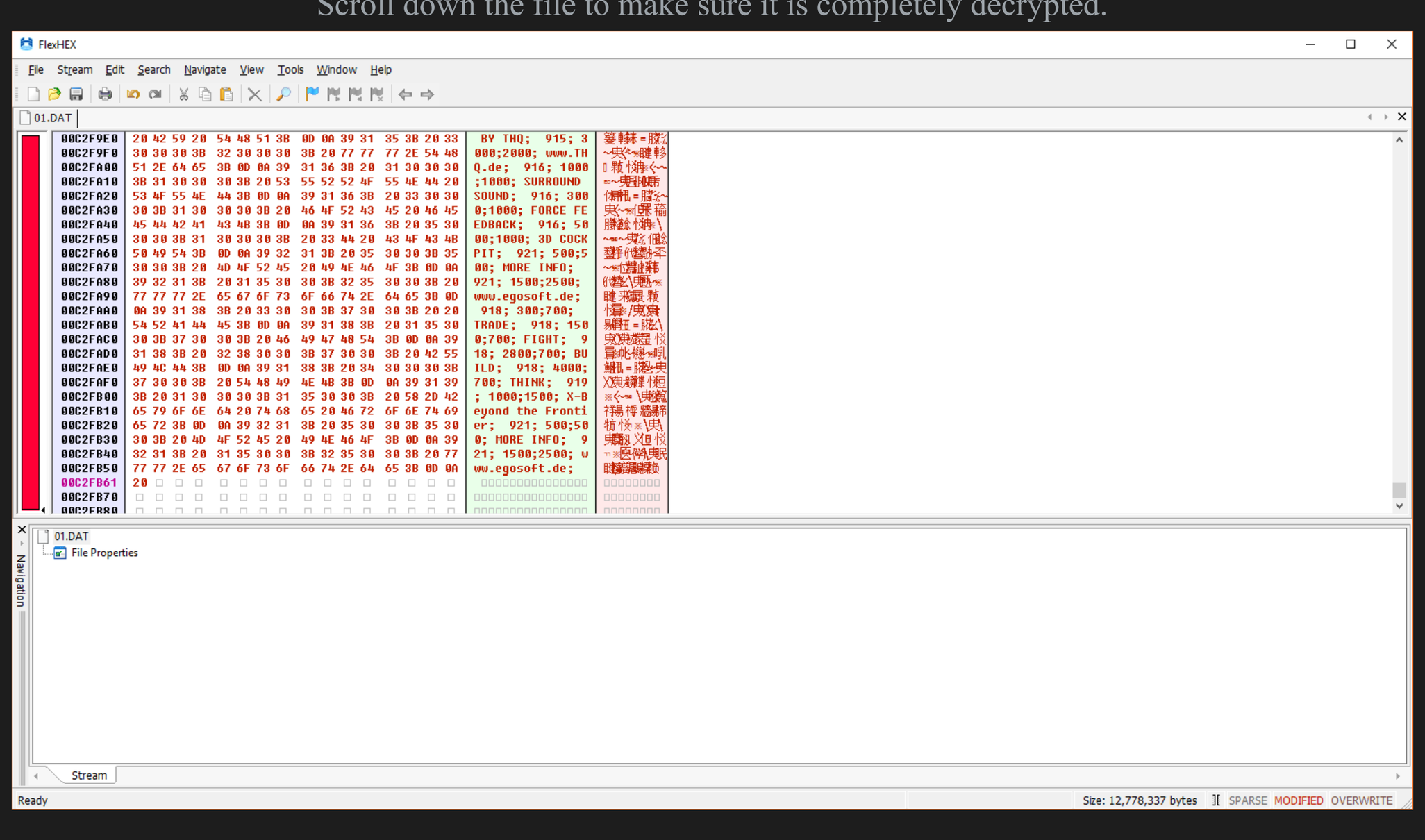
So this is really **XOR** and this time we will have **33 as the** second operand .

Press **Ctrl + Z** to undo the previous operation and re- **XOR** with the new operand.

We get the following at the output:



Scroll down the file to make sure it is completely decrypted.



Everything is ok, but there is problem number two, how do you determine the file boundaries?

Next to the above file is **01.CAT** , which, judging by its name, is a specific header and **TOC** , and due to its small size it is quite suitable for storing data about the file table of names and offsets.

And it is also encrypted, but using a different algorithm. I will not go into the details of the search, I can only add that the decryption code is executed after calling standard functions for working with files, namely in this place:

===

00472A13> / 8A62 01 MOV AH, BYTE PTR DS: [EDX + 1]
00472A16. | 8AF8 MOV BH, AL
00472A18. | FEC7 INC BH
00472A1A. | 8A1A MOV BL, BYTE PTR DS: [EDX]
00472A1C. | 32C3 XOR AL, BL
00472A1E. | 8ADF MOV BL, BH
00472A20. | 8802 MOV BYTE PTR DS: [EDX], AL
00472A22. | 32DC XOR BL, AH
00472A24. | 8AC7 MOV AL, BH
00472A26. | FEC0 INC AL
00472A28. | 885A 01 MOV BYTE PTR DS: [EDX + 1], BL
00472A2B. | 8B5D F0 MOV EBX, DWORD PTR SS: [EBP-10]
00472A2E. | 83C2 02 ADD EDX, 2
00472A31. | 3BD3 CMP EDX, EBX
00472A33. | 8B5D F8 MOV EBX, DWORD PTR SS: [EBP-8]
00472A36. ^ \ 7C DB JL SHORT x.00472A13

===

Decryption loop in x.exe.

Brief essence of the algorithm:

1. We take the first byte;
2. Take the decryption key = **DBh** ;
3. Xor it with the first byte;
4. Increase (we get **DCh**) ;
5. Xorim the second byte with the increased value;
6. Increase again (we get **DDh**);
7. We repeat from the first step, moving on to the next pair of bytes.

On Delphi, you would get something like this loop code:

===

```
for I: = 0 to Fsize + 3 - 1 do // add three bytes to equalize the size for a multiple of two
  begin
    try
      Application.ProcessMessages;
      // For every first byte
      BlockRead (DataFile, ReadBuf1,1);
      WriteBuf1: = ReadBuf1 xor IncXor;
      Seek (datafile, offset);
      BlockWrite (Datafile, Writebuf1,1);
      inc (count);
      Label2.Caption: = IntToStr (count);
      ProgressBar1.Position: = ProgressBar1.Position + 1;
      inc (offset);
      IncXor: = IncXor + 1;
     // For every second byte
      BlockRead (DataFile, ReadBuf2,1);
      WriteBuf2: = IncXor xor ReadBuf2;
      Seek (datafile, offset);
      BlockWrite (Datafile, Writebuf2,1);
      inc (count);
      Label2.Caption: = IntToStr (count);
      ProgressBar1.Position: = ProgressBar1.Position + 1;
      inc (offset);
      IncXor: = IncXor + 1;
    except
    end;
  end;
```

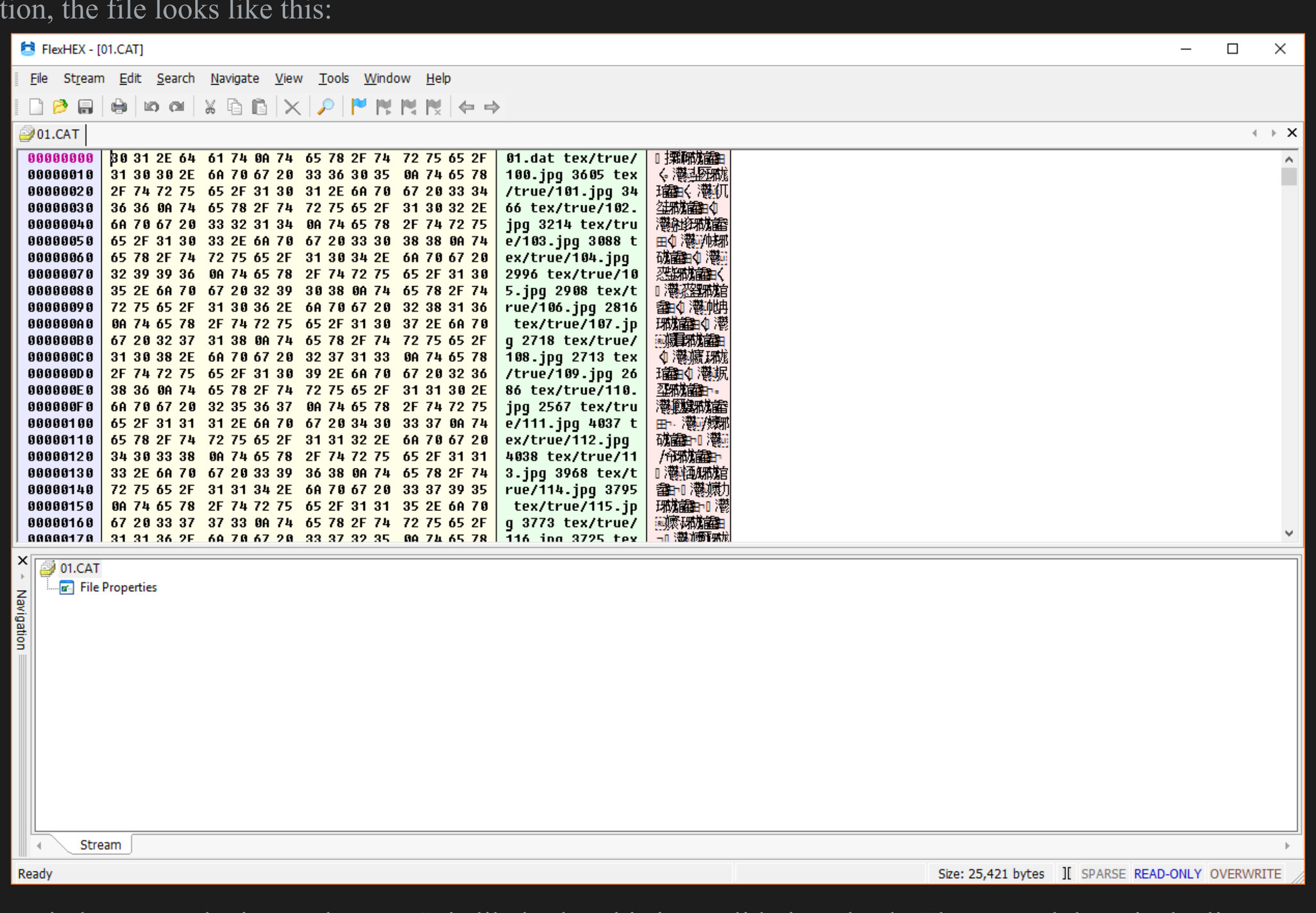
===

Where:

1. DataFile: File of byte;
2. IncXor, readbuf1, writebuf1, writebuf2, readbuf2, i, count, offset, Fsize: integer;
3. IncXor has an initial value of **$ DB** ;

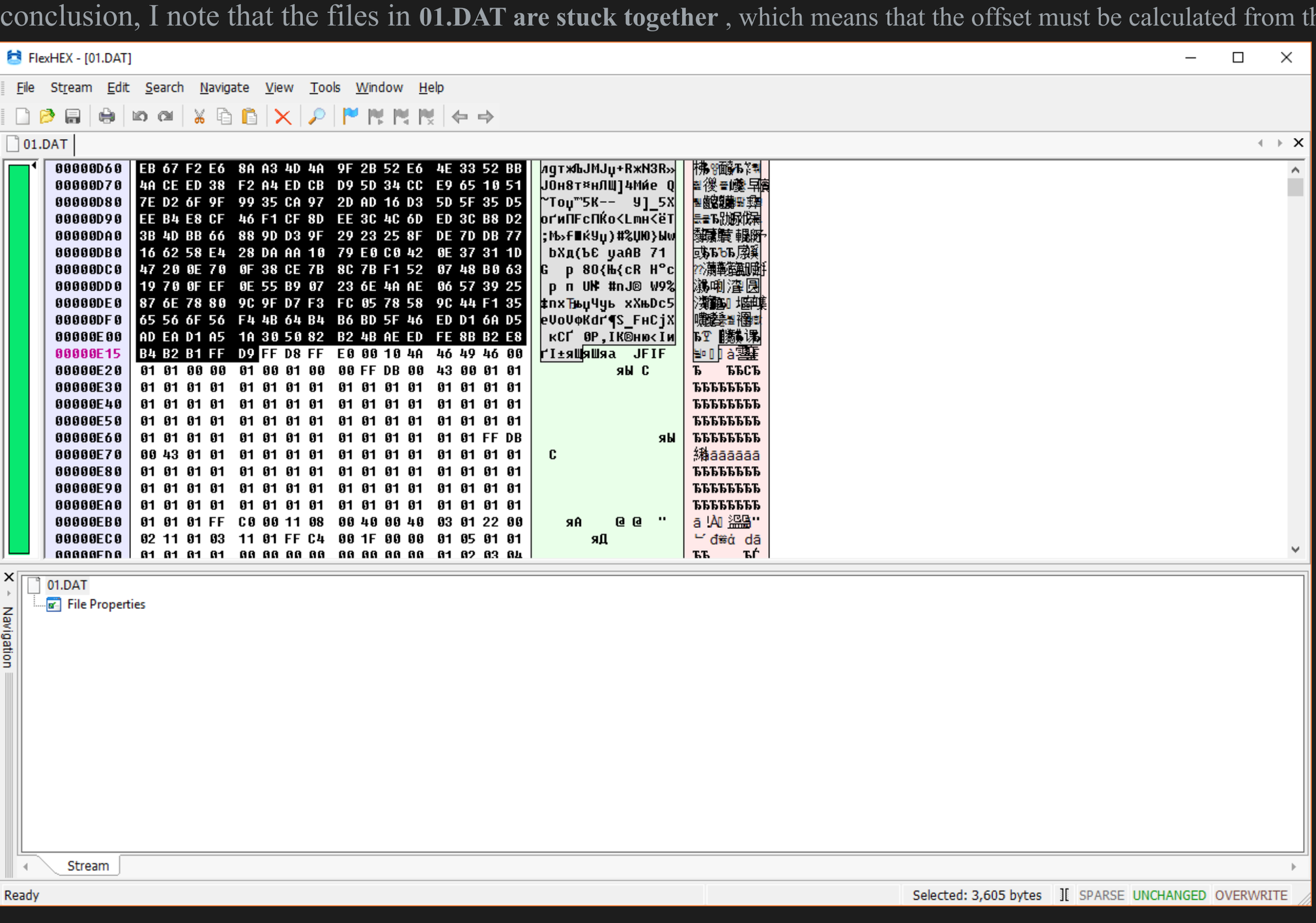
===

After decryption, the file looks like this:



The first six bytes are the internal name. It is likely that this is a validation check. The seventh byte is the line terminator. Further, the names of files with full paths and their sizes are clearly visible, presented as strings.

In conclusion, I note that the files in **01.DAT are stuck together** , which means that the offset must be calculated from the size of the previous one.



This was the end of the murky garbage.