INSIDE THE ULTIMA ONLINE GOLD DEMO - UODEMO.DAT

GOAL

It's our goal to get a deep understanding of how the Ultima Online Gold Demo works. This demo is a representation of the rule set from the Ultima Online Second Age Era.

There is proof that some people have already reversed this demo partially or as a whole, however so far no tools or knowledge has been published. This project is to overcome those shortcomings.

URL's with some proof for this:
http://www.runuo.com/forums/general-discussion/94767-help-m-files.html
http://azaroth.org/2008/12/31/your-topic/ (posting by Faust)

If we understand the demo there is a big chance we can alter the demo and even create our own demo. By default mounting horses is not possible in the demo, but what if we can alter the demo and unlock horses; can we then see how horses behaved during T2A?

This demo is 10 years old and I do not understand no one published his/her work. Maybe that DMCA thing is in the way?

UTILITIES USED

IDA Pro, a very professional utility, definitely worth buying, Standard version is affordable.
HxD, a very neat hex editor and above all, it's free

ABOUT ME

I'm just a guy who loves the Ultima universe and knows a bit assembler. Why not combine the two? ☺ I've been into computer starting from age twelve, and Ultima VII was the first game I bought myself. Oh yeah, I've been programming since my 13 and I started with assembler at age 15.

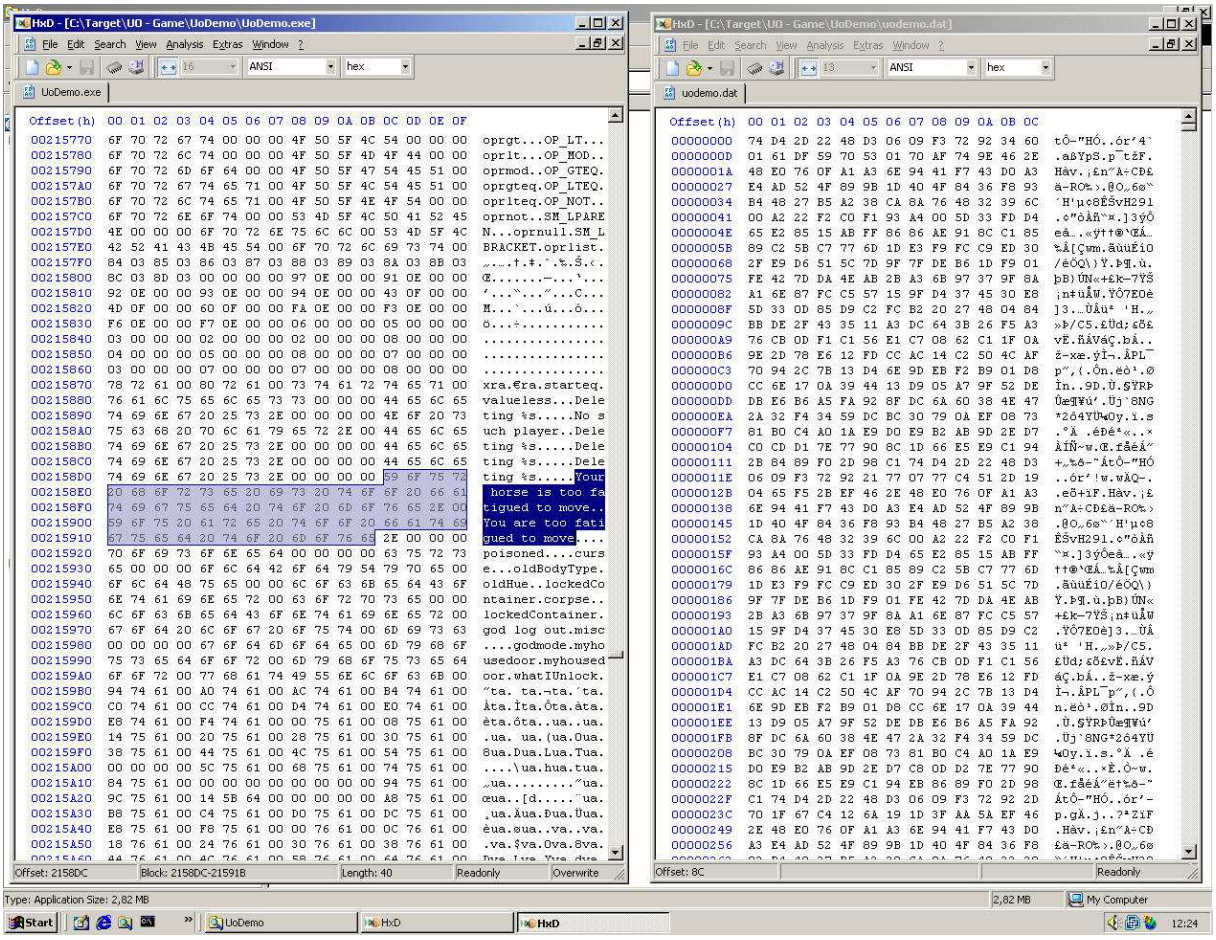
INITIAL LOOK

For the demo to work you must first install the Ultima Online Gold Client and then install the demo. The demo requires the client to be installed and it doesn't like newer clients, so my suggestion is: install the UO client & demo inside a virtual machine. How to do this is beyond the scope of this document and is left as an exercise for the reader.

Download the T2A era client from here: http://www.uosecondage.com/downloads.aspx, but do not install the accompanied patch!

After installing the demo you will see 3 files, 2 files of them being important:

Opening UODEMO.EXE is promising, it contains many readable strings thus OSI did not use encryption to protect their demo. However, UODEMO.DAT, contains 100% binary and therefore we can be sure it's compressed and/or encrypted somehow.



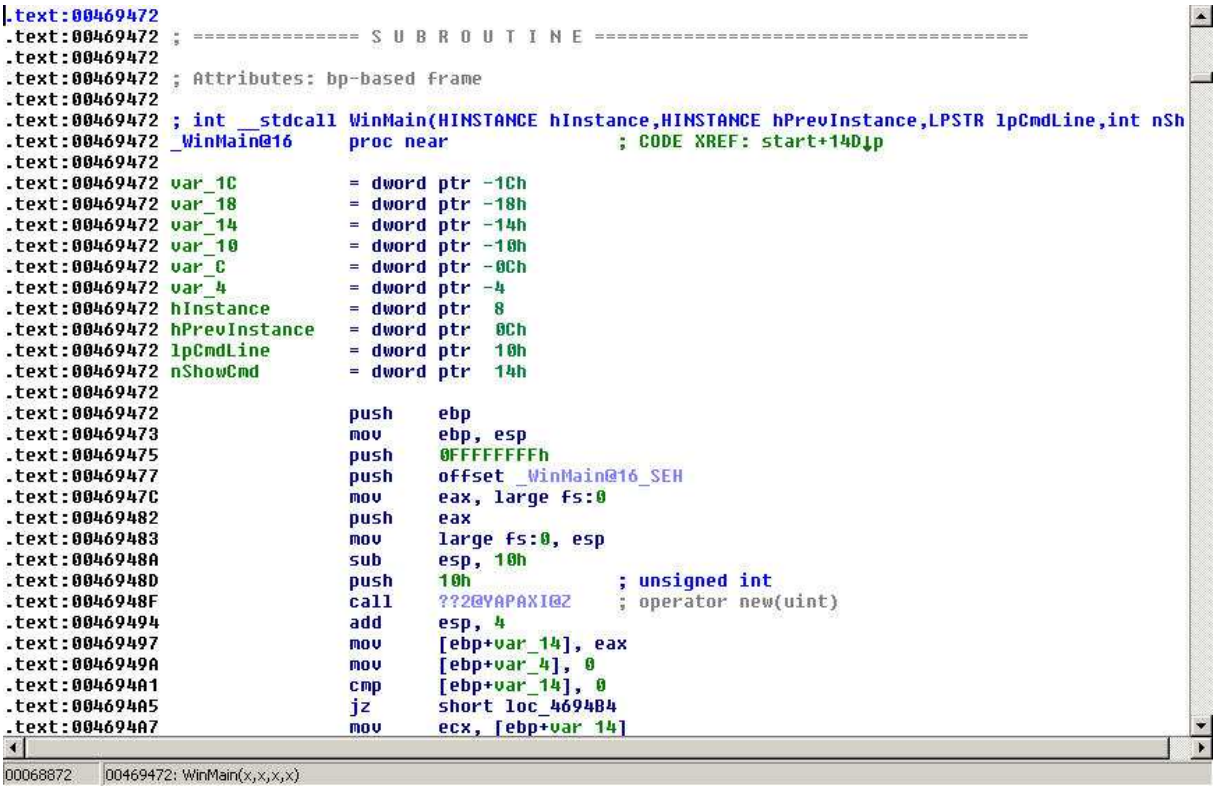
(left side = UoDemo.exe, right side = uodemo.dat)

BEFORE WE CONTINUE

Let's not forget that the demo requires the client to be installed, that's because the graphics and sound are being loaded from the client itself.

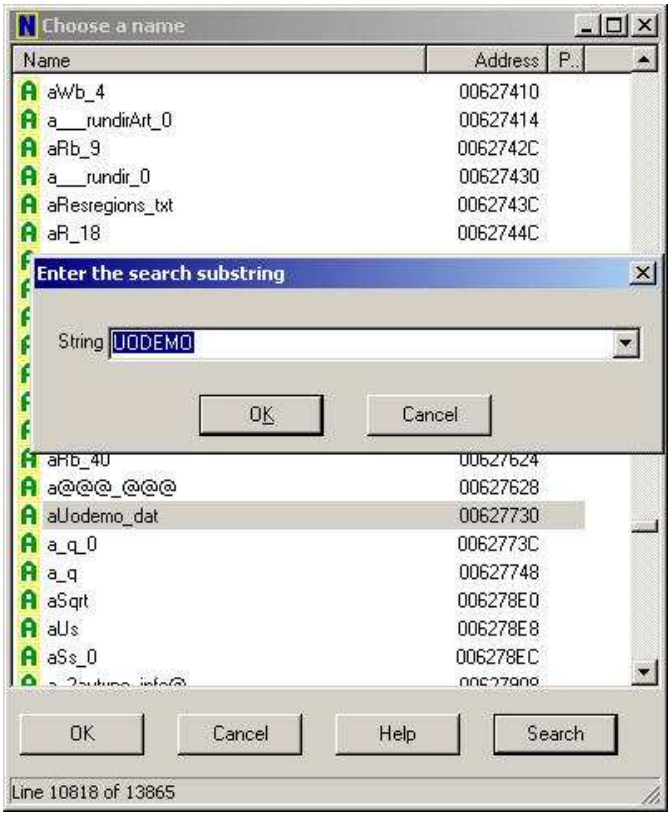
INITIAL DISASSEMBLY

We load IDA and tell it to disassemble the demo executable.



Let's start with looking for this UODEMO.DAT, we want to know where and how it loads.

STEP 1: open the list with labels and search for UODEMO



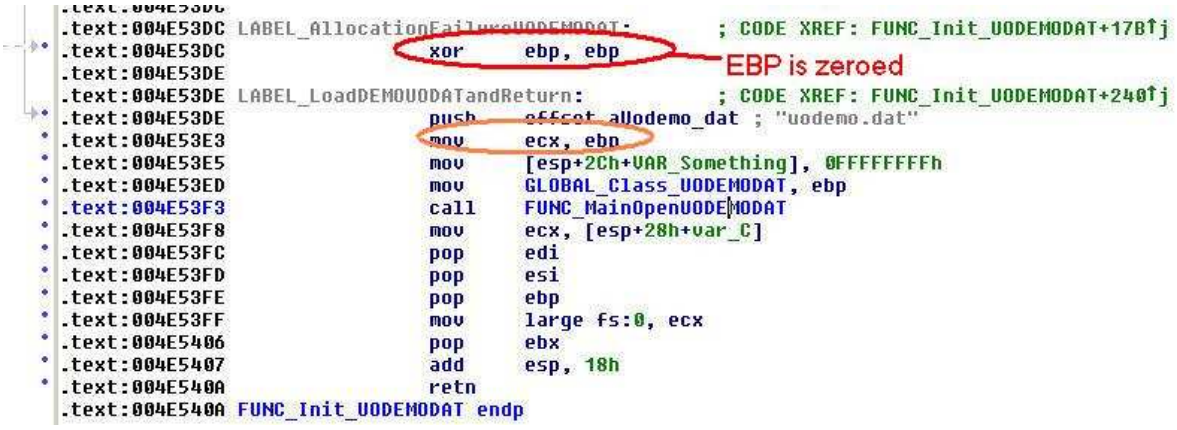
STEP 2: go to the label and follow the XREF

```
.data:00627730 ; char aUodemo_dat[]
.data:00627730 aUodemo_dat     db 'uodemo.dat',0        ; DATA XREF: sub_4E519A:loc_4E53DE↑o
.data:0062773B                 align 4
.data:0062773C a_q_0           db '.q',0                ; DATA XREF: .text:004E5C6B↑o
.data:0062773F                 align 10h
```

STEP 3: analyze what the code does, use the built-in debugger

```
.text:004E53DC ; ---------------------------------------------------------------------------
.text:004E53DC
.text:004E53DC loc_4E53DC:                              ; CODE XREF: sub_4E519A+17B↑j
.text:004E53DC                 xor     ebp, ebp
.text:004E53DE
.text:004E53DE loc_4E53DE:                              ; CODE XREF: sub_4E519A+240↑j
.text:004E53DE                 push    offset aUodemo_dat ; "uodemo.dat"
.text:004E53E3                 mov     ecx, ebp
.text:004E53E5                 mov     [esp+2Ch+var_4], 0FFFFFFFFh
.text:004E53ED                 mov     dword_701640, ebp
.text:004E53F3                 call    sub_4E2780
.text:004E53F8                 mov     ecx, [esp+28h+var_C]
.text:004E53FC                 pop     edi
.text:004E53FD                 pop     esi
.text:004E53FE                 pop     ebp
.text:004E53FF                 mov     large fs:0, ecx
.text:004E5406                 pop     ebx
.text:004E5407                 add     esp, 18h
.text:004E540A                 retn
.text:004E540A sub_4E519A      endp
```

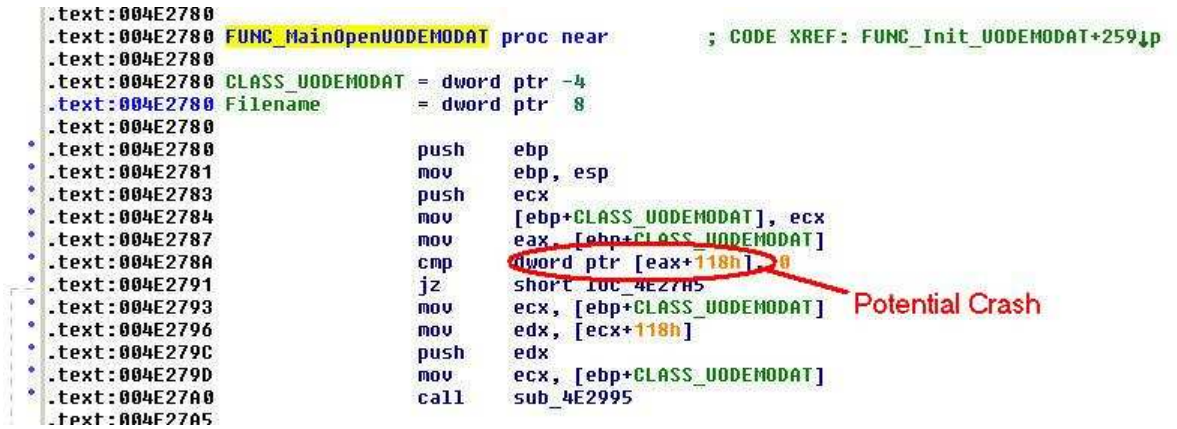
This is a screenshot of the analyzed section:



If you look well there is a possible crash!

If the code reaches LABEL_AllocationFailureUODEMODAT the EBP is zeroed, nothing wrong with that, but then EBP is stored in ECX. This is a standard (old) compiler mechanism; ECX stores the content of the this-pointer for those who know C++. From this you can derive that FUNC_MainOpenUODEMODAT is a class function.

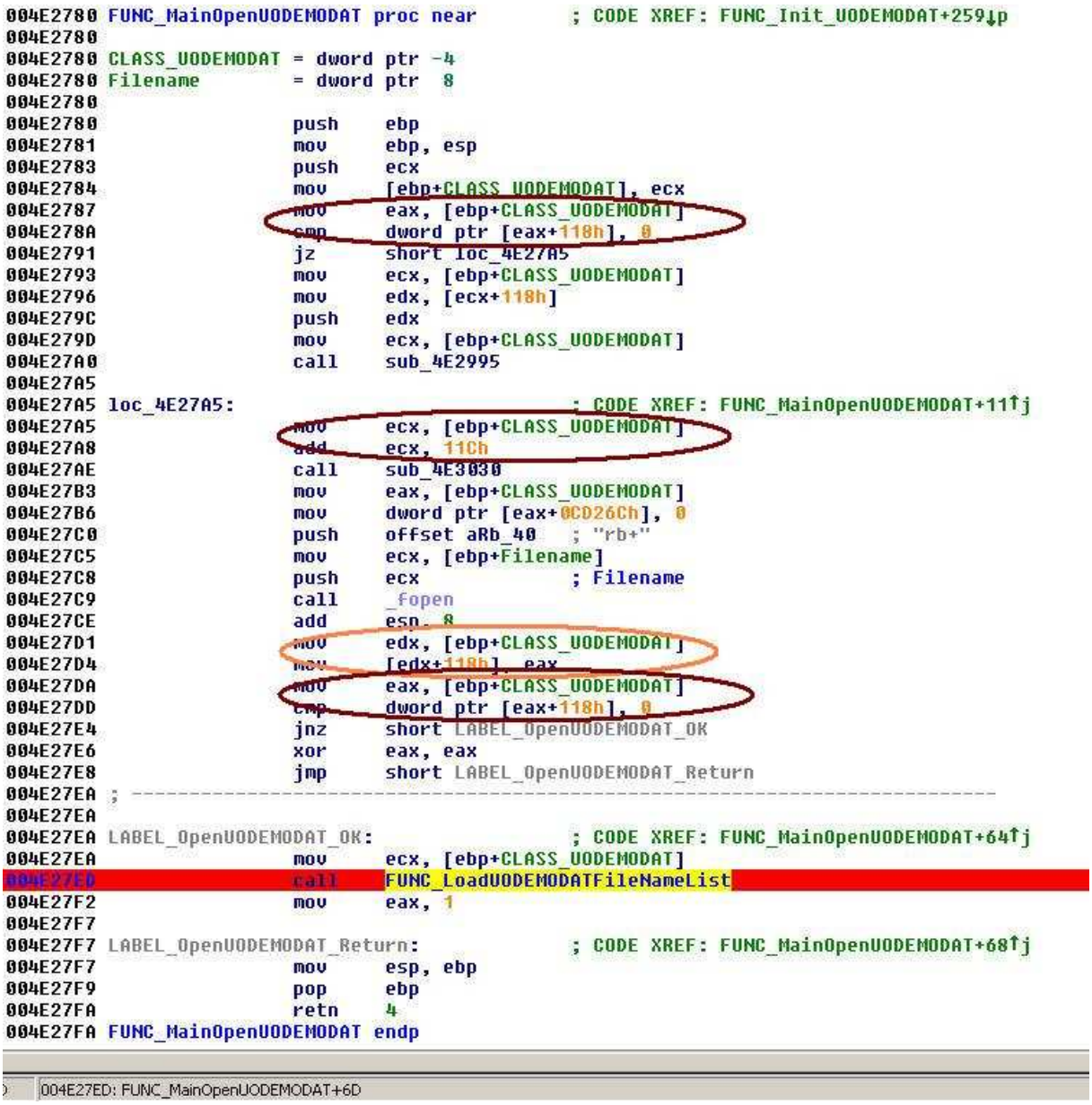
Let's look at FUNC_MainOpenUODEMODAT:



You notice that ECX is stored in [ebp+CLASS_UODEMODAT] and then that value is stored back in EAX. Very silly code? Yes it is, this tells us that the compiler used is really old and you should be amazed how compiler technology has improved. A modern C++-compiler with optimizations enabled will never ever create such redundant code.

About the possible crash: "cmp dword ptr [eax+118h]" -> if EBP was zeroed, ECX will be zero, [ebp+CLASS_UODEMODAT] will be zero and [eax+118h] will refer to [0+118h] and that is an illegal address under win32 bit. Access Violation guaranteed!

Back, to the task at and, let us view the whole function:

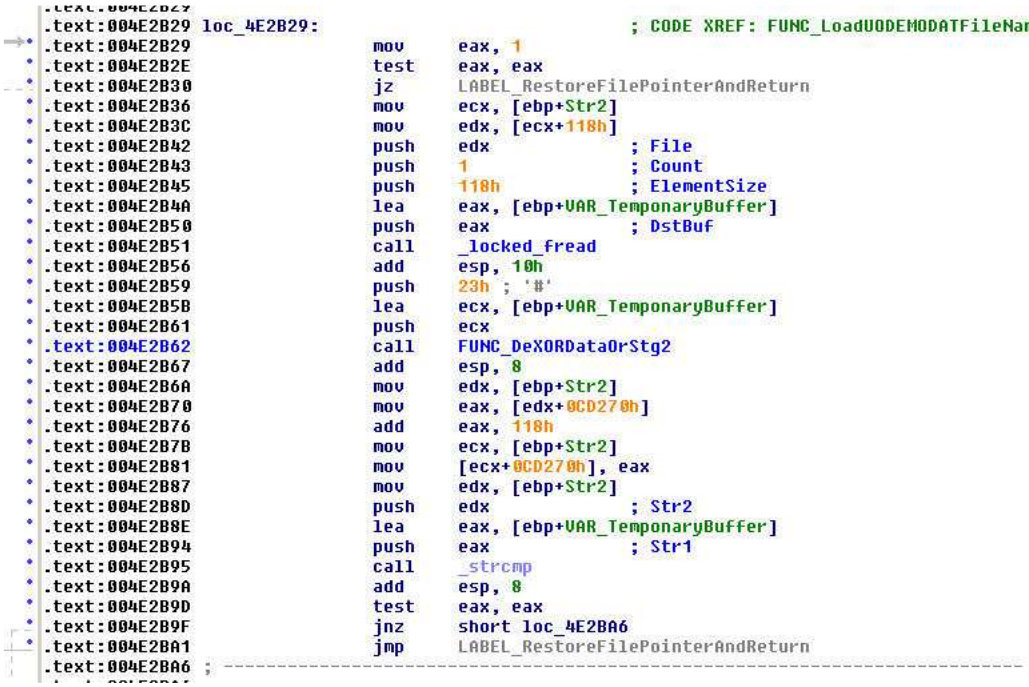


There are some things going on which I didn't analyze 100% yet.

You notice that at 004E27D1 (in orange) the returned file handle is stored in [CLASS_UODEMODAT+ 118h]. That means the code at 004E2791 will only be executed when this function is called and the file handle is non-zero. The file handle will only be non-zero when the function has been called before. However, the XREFs teach us that this is not the case in this demo!?

LABEL_OpenUODEMODAT_OK is called when the file was opened correctly. Please know that UODEMO.DAT is opened by fopen and fopen does not support file sharing. Look up your C documentation please. ☺

The following screenshot is inside this FUNC_LoadUODEMODATFileNameList (see the red line on the previous one)



```
.text:004E2B29 loc_4E2B29:                             ; CODE XREF: FUNC_LoadUODEMODATFileNameList+1F6↓j
.text:004E2B29                 mov     eax, 1
.text:004E2B2E                 test    eax, eax
.text:004E2B30                 jz      LABEL_RestoreFilePointerAndReturn
.text:004E2B36                 mov     ecx, [ebp+Str2]
.text:004E2B3C                 mov     edx, [ecx+118h]
.text:004E2B42                 push    edx             ; File
.text:004E2B43                 push    1               ; Count
.text:004E2B45                 push    118h            ; ElementSize
.text:004E2B4A                 lea     eax, [ebp+VAR_TemponaryBuffer]
.text:004E2B50                 push    eax             ; DstBuf
.text:004E2B51                 call    _locked_fread
.text:004E2B56                 add     esp, 10h
.text:004E2B59                 push    23h ; '#'
.text:004E2B5B                 lea     ecx, [ebp+VAR_TemponaryBuffer]
.text:004E2B61                 push    ecx
.text:004E2B62                 call    FUNC_DeXORDataOrStg2
.text:004E2B67                 add     esp, 8
.text:004E2B6A                 mov     edx, [ebp+Str2]
.text:004E2B70                 mov     eax, [edx+0CD270h]
.text:004E2B76                 add     eax, 118h
.text:004E2B7B                 mov     ecx, [ebp+Str2]
.text:004E2B81                 mov     [ecx+0CD270h], eax
.text:004E2B87                 mov     edx, [ebp+Str2]
.text:004E2B8D                 push    edx             ; Str2
.text:004E2B8E                 lea     eax, [ebp+VAR_TemponaryBuffer]
.text:004E2B94                 push    eax             ; Str1
.text:004E2B95                 call    _strcmp
.text:004E2B9A                 add     esp, 8
.text:004E2B9D                 test    eax, eax
.text:004E2B9F                 jnz     short loc_4E2BA6
.text:004E2BA1                 jmp     LABEL_RestoreFilePointerAndReturn
.text:004E2BA6 ; ---------------------------------------------------------------------------
```

Important things to learn from this screenshot:

Data is read into a buffer of 0x118 bytes at VAR_TemponaryBuffer.

Data is XORed (the parameter 0x23 equals 0x118 / 4 / 2).

The function stops when VAR_Temponary is equal to Str2.

Str2 is not shown on the screenshot, but its content is “@@@.@@@”.

The next obvious step is to go analyze the FUNC_DeXORDataOrStg2:

```
.text:004E4F02
.text:004E4F02 loc_4E4F02:                             ; CODE XREF: FUNC_DeXORDataOrStg2+A3↓j
.text:004E4F02                 cmp     edi, ebp
.text:004E4F04                 jge     short loc_4E4F6A
.text:004E4F06                 mov     eax, [esp+20h+var_10]
.text:004E4F0A                 add     eax, 1010101h
.text:004E4F0F                 cmp     eax, 1010101h
.text:004E4F14                 mov     [esp+20h+var_10], eax
.text:004E4F18                 jnb     short loc_4E4F1F
.text:004E4F1A                 inc     eax
.text:004E4F1B                 mov     [esp+20h+var_10], eax
.text:004E4F1F
.text:004E4F1F loc_4E4F1F:                             ; CODE XREF: FUNC_DeXORDataOrStg2+53↑j
.text:004E4F1F                 mov     eax, [esp+20h+var_C]
.text:004E4F23                 add     eax, 1010104h
.text:004E4F28                 cmp     eax, 1010104h
.text:004E4F2D                 mov     [esp+20h+var_C], eax
.text:004E4F31                 jnb     short loc_4E4F38
.text:004E4F33                 inc     eax
.text:004E4F34                 mov     [esp+20h+var_C], eax
.text:004E4F38
```

And this is were at all became interesting, I'm not going to copy/paste the whole algorithm here, but the values 0x1010101 and 0x1010104 I found so weird and interesting that I used google to look them up. I reached a Russian forum that contained the source code of the GOST cipher. That source code fitted into this main loop directly!

```
/* The constants for addition */
#define C1 0x01010104
#define C2 0x01010101

void
gostofb(word32 const *in, word32 *out, int len,
        word32 const iv[2], word32 const key[8])
{
        word32 temp[2];         /* Counter */
        word32 gamma[2];        /* Output XOR value */

        /* Compute starting value for counter */
        gostcrypt(iv, temp, key);

        while (len--) {
                temp[0] += C2;
                if (temp[0] < C2)       /* Wrap modulo 2^32? */
                        temp[0]++;      /* Make it modulo 2^32-1 */
                temp[1] += C1;
                if (temp[1] < C1)       /* Wrap modulo 2^32? */
                        temp[1]++;      /* Make it modulo 2^32-1 */

                gostcrypt(temp, gamma, key);

                *out++ = *in++ ^ gamma[0];
                *out++ = *in++ ^ gamma[1];
        }
}
```

So, UODemo uses the GOST cipher, http://en.wikipedia.org/wiki/GOST_28147-89. For implementations of the cipher try this google search: http://www.google.com/search?q=gost+ gostofb&meta=.

I'm sure that 10 years ago it would have been more difficult to discover this. But, what saddens me is that other people have already decrypted the demo and they never published this interesting fact. Come on people! Share you knowledge. Don't take it with you when you die. Aargh! Calm Blue Ocean. Calm Blue Ocean.

With the information we have now, we can actually write a utility that will decrypt this UODEMO.DAT file.

NOTE: the UoDemo only supports up to 3000 files (look at the disassembly yourself to verify this)

An idea I have: remove the encryption code from the demo and have it operate on an unencrypted version of UODEMO.DAT named UODEMO.BIN (for example).

That's it for now, I got bored of writing. Back to code please.