Reversing the Golden Axe
A Journey to the Past - Fixing a bug from 1990
Written By: Orr, September 2006

Introduction

Let me start by telling a somewhat personal story. (You can skip to the next part if you find it boring). When I was in the 4th grade (circa 1994), my father bought me my first computer. One of the first games I ever installed was the above-mentioned, **Golden Axe** and it soon became my favorite game. The common file-browser was the “Norton Commander”, and one of the options it had was viewing a file’s contents. As a curios kid I once opened the main exe on a simple file view and what I saw was the gibberish ASCII codes. Upon asking him, my father told me it was the language “only a computer understands”. I told myself that perhaps one day I will be able to comprehend that language, and change everything that I want in that game. Twelve years later I found myself downloading Golden Axe from an Abandonware site, only to find a few days later that there is a little...

Problem

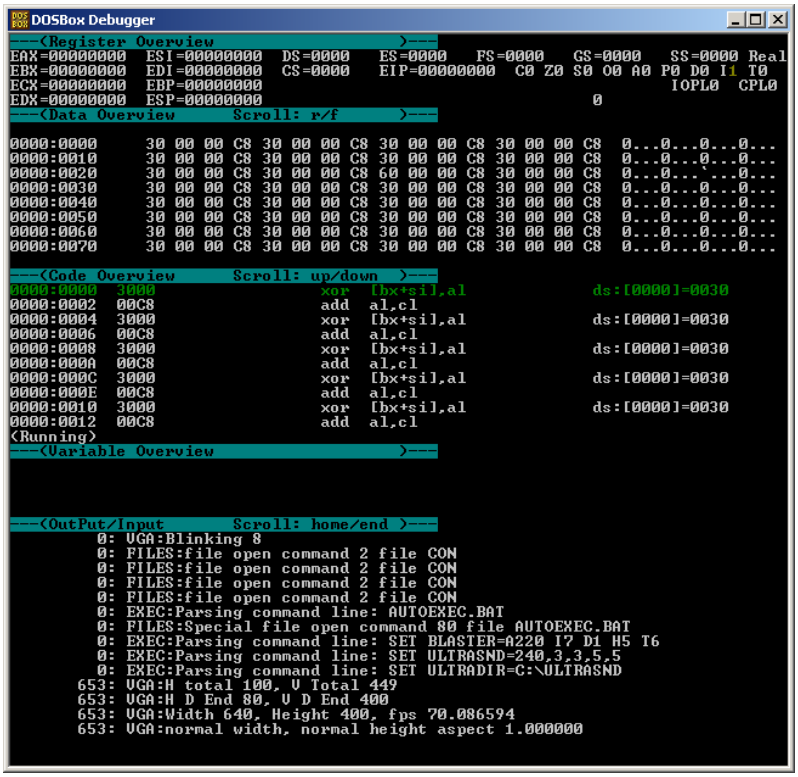
Golden Axe (cracked by Fabulous Furlough of The Humble Guys) has 3 main options, Arcade, Beginner and **The Duel**. You will find no problem playing and finishing the first two, but if you attempt to play the duel mode against the computer, you will find that after you've finished the horrible 13th level, you get a message saying: "**Enter Disk 2 Press Enter**". What is this? Was the cracker lazy and didn't disable all the checks? Should I finish the work? Am I the Chosen One?

Archeology

The first thing I did was to open the file named **GOLD.EXE** in a Hex Editor, and to my surprise I found no strings in the file. The fact that the file was only 7KB aroused my suspicion. This was merely a loader to a bigger file - **AXE.DAT** which appears to be an executable, but it doesn't contain any strings in it as well. A short disassembly revealed to me that the file was exe-encrypted. I thought it was the type of encryption that old viruses used to embody (the LODSB/XOR/STOSB type of protection), but the more I delved into it, it seemed a little bit more complex. The smoking gun was that this loader was actually in the end of the file, so I looked back at the MZ header, and there I saw the signature that evaded my eyes earlier: **LZ91**. LZ of course made me think of the famous Lempel-Ziv compression engine, so after searching a bit, I found that there was a small exe-packer named **LZEXE** that was used back in those days. If only I had searched some more I would have found out that there is a generic unpacker called **UNLZEXE** that does the job without even requiring me to manually unpack it.

Compatibility

If you want to play old DOS games on 2000/XP machines, you'd have to use some sort of an emulator. The one I used in order to play the game was an amazing DOS emulator called **DOSBox**. I tried to execute the TR debugger and later an old version of SoftICE on DOSBox, but they all gave me some issues, mainly regarding TSR's and Virtual Memory. To my pleasant surprise I found that there is a version of DOSBox that comes with an integrated debugger (and a very useful one, I might add), so now I can finally begin to work properly. So, assuming we now have an unpacked file and a working debugger – we can finally stop the babbling and get to work.



Discovery

The DOSBox debugger has a very nice feature that shows the names of the files that were loaded by the game. You'll notice that if you load the Duel mode, the game will load the files LEVEL5.MAP and LEVEL5.CHR among other files. What I did was simply to delete those files and load the duel mode again. That dreaded message showed up again before I even started playing. This is certainly not a copy-protection.

The first place to attack is where the check is being made. The game asks for disk 2, and then expects the user to press the Enter key. I set a breakpoint over INT 21h (bpint 21 *) and pressed enter. This is where it got me (I already did the naming):

```
seg000:7808
seg000:7808 OpenFile        proc near                   ; CODE XREF: FileManipulation+45 p
seg000:7808                                             ; sub_1FB0+2B p ...
seg000:7808                 call    sub_7751
seg000:780B                 push    ax                  ;
seg000:780C                 push    dx                  ; Save regs
seg000:780D                 push    ds                  ;
seg000:780E
seg000:780E TryToOpenAgain:                             ; CODE XREF: OpenFile+20 j
seg000:780E                 mov     ax, seg seg002
seg000:7811                 mov     ds, ax              ; ds = address of seg00
seg000:7813                 assume ds:seg002
seg000:7813                 mov     cs:byte_7439, 0
seg000:7819                 mov     ah, 3Dh ; '='       ; Interrupt Service 3Dh
seg000:781B                 mov     al, 0               ; access mode = 0 (read)
seg000:781D                 mov     dx, FileName        ; dx = pointer to file name
seg000:7821                 int     21h                 ; DOS - 2+ - OPEN DISK FILE WITH HANDLE
seg000:7821                                             ; DS:DX -> ASCIZ filename
seg000:7821                                             ; AL = access mode
seg000:7821                                             ; 0 - read
seg000:7823                 jnb     short FileOpened    ; If OK, then proceed
seg000:7825                 call    ErrorMessage        ; Output error
seg000:7828                 jmp     short TryToOpenAgain ; Try again
seg000:782A ; ---------------------------------------------------------------------------
seg000:782A
seg000:782A FileOpened:                                 ; CODE XREF: OpenFile+1B j
seg000:782A                 mov     FileHandle, ax      ; Save the file handle
seg000:782D                 pop     ds                  ;
seg000:782E                 assume ds:nothing           ;
seg000:782E                 pop     dx                  ; Restore regs
seg000:782F                 pop     ax                  ;
seg000:7830                 retn                        ; Return to caller
seg000:7830 OpenFile        endp
```

We see that the game uses interrupt service 3Dh and then calls INT 21h (meaning it would attempt to open a file) while DX holds the pointer to the filename. If the file is opened successfully the handle is stored in a variable (FileHandle), and if not, a function outputting the error message will be called. The function ultimately fails because it cannot find a specific file. But what file?

Anytime you will try to press enter you will receive the following message in the debugger:

FILES:Makename encountered an illegal char ^D hex: 4 !

So, the game asks for a gibberish filename (^D), and upon failure prompts the error message. Since the filename it is looking for is an illegal file name, I was now confident that no file was really missing, and that this is a bug that needs to be fixed. But what and where to look?

Backtrace

First, I wanted to know who the caller of this OpenFile function, but since I couldn't get out of the "Insert Disk 2" loop, I messed with the code-flow a little:

SR EIP 7830

By doing that, I set the EIP (next instruction to run) to the address of the ret instruction (See above). This brought me to this interesting place:

```
seg000:0736 loc_736:                                    ; CODE XREF: sub_721+9 j
seg000:0736                 push    ax
seg000:0737                 mov     bx, [bx+25E3h]
seg000:073B                 mov     dx, [bx]
seg000:073D                 mov     byte ptr unk_10523, 61h ; 'a'
seg000:0742                 call    ProcessFileNames    ; Interesting call
seg000:0745                 pop     ax
seg000:0746                 mov     cs:word_71D, ax
seg000:074A                 push    word_F784
seg000:074E                 pop     cs:word_71F
seg000:0753                 mov     bx, ax              ;
seg000:0755                 shl     bx, 1               ; Get relative offset
seg000:0757                 mov     ax, [bx+25A1h]      ;
seg000:075B                 mov     word_10CC5, ax      ; All of these are
seg000:075E                 mov     ax, [bx+264Bh]      ; operations used to find
seg000:0762                 mov     word_10CC7, ax      ; the filenames in memory
seg000:0765                 mov     ax, [bx+268Dh]      ; as they are stored in some
seg000:0769                 mov     word_10CC9, ax      ; sort of a table
seg000:076C                 mov     ax, word_10CC5      ;
seg000:076F                 mov     FileName, ax        ; Finally open the file
seg000:0772                 call    OpenFile            ;
seg000:0775                 cmp     FileHandle, 0
seg000:077A                 jnz     short loc_789
seg000:077C                 mov     byte ptr unk_10522, 47h ; 'G'
seg000:0781                 mov     byte ptr unk_10523, 32h ; '2'
seg000:0786                 jmp     PrintErrorMsg
```

This is the beginning of a large function that later on goes on to read the file and later closes it, and also involves error-checking. From this chunk we can understand that the function opens the character files and reads from them, but it is of little use to us, since it has the addresses already passed on to it. Again, we have to back-trace a little in order to return to the caller of this function. If you' study the code in IDA you see that it returns in this address:

```
seg000:08E2                 mov     ax, ds:1464h
seg000:08E5                 sub     ax, cs:word_71F
seg000:08EA                 mov     [bx+26CFh], ax
seg000:08EE                 pop     bx
seg000:08EF                 pop     ax
seg000:08F0                 retn
seg000:08F0 FileManipulation endp
```

So, again, we'll set the breakpoint to the ret instruction (you can also scroll to that instruction and press F9):

```
BP [SEG00]:08f0        ;SEG00 is variable
```

Press F5 and return to the game, continue to play a little until you reach the breakpoint. After that simply trace over it and you will be taken to a fantastic little function:

```
seg000:6E4C LoadLevels      proc near                       ; CODE XREF: GameLoop?:loc_67D2 p
seg000:6E4C                 mov     ax, ds:145Ah
seg000:6E4F                 call    FatalErrorM3
seg000:6E52                 mov     bx, ds:2259h            ; bx = Level Number
seg000:6E56                 shl     bx, 1                   ;
seg000:6E58                 cmp     byte ptr ds:0B92h, 3    ; \
seg000:6E5D                 jnz     short Arcade            ; |
seg000:6E5F                 cmp     byte ptr ds:410h, 0FFh  ; | Is it arcade?
seg000:6E64                 jnz     short Arcade            ; /
seg000:6E66                 mov     bx, ds:558h
seg000:6E6A                 shl     bx, 1
seg000:6E6C                 mov     si, [bx+1195h]          ;
seg000:6E70                 mov     dx, [bx+11D1h]          ; A TABLE!
seg000:6E74                 mov     di, [bx+11B3h]          ;
seg000:6E78                 jmp     short Finished?
```

Resolution

Now, as you see, the game loads several items into registers, from a table that looks like that:

```
seg002:1170  00 09 2A 09 2C 0A 0A 0B 10 32 32 06 27 03 03 09
seg002:1180  0B 11 2C 33 33 08 25 08 24 06 08 27 27 07 03 27
seg002:1190  03 07 03 21 03 71 11 71 11 71 11 71 11 73 11 75
seg002:11A0  11 77 11 7B 11 7D 11 7F 11 85 11 87 11 89 11 8D
seg002:11B0  11 91 11 71 11 71 11 71 11 72 11 74 11 76 11 79
seg002:11C0  11 7C 11 7E 11 82 11 86 11 88 11 8B 11 8F 11 93
seg002:11D0  11 00 00 00 00 00 00 01 00 01 00 01 00 02 00 01
seg002:11E0  00 01 00 03 00 01 00 01 00 02 00 20 00 24 00 00
```

Or on a more simplified view:

Level	SI	DI	DX	*SI	*DI
1	1171	1171	00	09	09
2	1171	1171	00	09	09
3	1171	1171	00	09	09
4	1171	1172	01	09	2a
5	1173	1174	01	09	2c
6	1175	1176	01	0A	0a
7	1177	1179	02	0B,10	32,32
8	117B	117C	01	06	27
9	117D	117E	01	03	03
10	117F	1182	03	9,0B,11	2c,33,33
11	1185	1186	01	08	25
12	1187	1188	01	08	24
13	1189	118B	02	06,08	27,27
14	118d	118F	**20**	07,03	27,03
15	1191	1193	**24**	07,03	21,03

What does all of that mean?!
SI is filled with the pointer to the type of enemy you will face.
DI is filled with the pointer to the color of the enemies.
DX is filled with the number of enemies to load (thus functioning as a counter).

Highlighted are the two members of this table, which form an inconsistency. Surprise – the 'bad' value is exactly in the 14th level, the one I was unable to reach!

If you follow the pattern of the table, you will see that *SI and *DI are filled (in a loop) according to the value loaded in DX. In level 14 there will be a loop of 20h times looking for the correct address, and will result in a crash. In order to resolve this situation, all you have to do is change those values to '02', and the 14th and 15th levels will be loaded successfully. All I have to do now is actually pass those levels now ☺

Epilogue
After fixing this bug I was filled with joy, due to the fact that this project completes and closes a circle that had begun many years ago. This was not 'cracking', this was truly Reverse Engineering in my eyes. I am still quite curios, however, on WHY this bug occurred in the first place. Unfortunately, this question still remains a mystery to me.

Thanks for reading this; I hope you didn't get too bored.
Any feedback is always welcomed.

Orr,
September 2006

OrrIscariot@gmail.com