Reverse engineering Heroes 3 – Stacks in battle

By Jakub Grzana
Last update: 03.04.2022

Prerequisites:

- IDA Pro (interactive disassembler by Hex-Rays) which is NOT free program!
- IDA Database for Heroes 3 SoD 3.59 by GrayFace, ERA_HD

Note: IDA database for 3.59 and ERA_HD are NOT interchangeable. You must use 3.59 for 3.59. Although version for ERA is still useful, to somehow figure out what's going on, due to better documentation.

Quick reminder for controls in IDA: **Alt+T** to search, **Tab** to switch to/from pseudocode.

Introduction

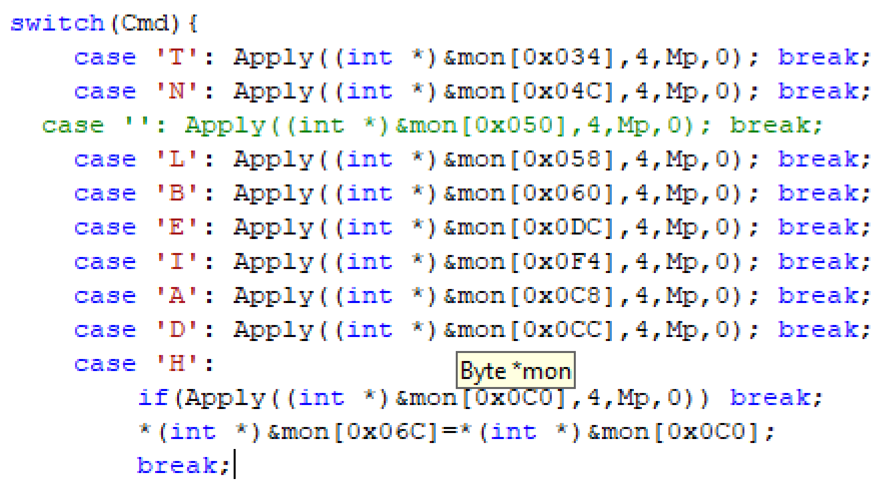
This document is less of tutorial, more of me figuring out how BM receiver works. I wanted to make receiver to get luck and morale of unit in combat, cause for now there's no easy way to do so and it seemed very weird – surely the value you see upon rightclicking during combat is stored somewhere, right?

TL:DR tutorial how to use IDA to better understand original code of WoG, especially in case of operations performed directly on memory.

Beginning

I wanted to get value of morale/luck of stack during combat. Searching WoG source is easier than IDA database and there's receiver that deals with many other combat stack attributes – BM receiver – so it's great starting point. I've searched for `'BM'` which leads to `ERM_Addition` in erm.cpp. Here we learn that BM's code is written within `ERM_BRound` function. You can find it in monsters.cpp.

First glance and things aren't promising though



```
switch(Cmd){
    case 'T': Apply((int *)&mon[0x034],4,Mp,0); break;
    case 'N': Apply((int *)&mon[0x04C],4,Mp,0); break;
  case '': Apply((int *)&mon[0x050],4,Mp,0); break;
    case 'L': Apply((int *)&mon[0x058],4,Mp,0); break;
    case 'B': Apply((int *)&mon[0x060],4,Mp,0); break;
    case 'E': Apply((int *)&mon[0x0DC],4,Mp,0); break;
    case 'I': Apply((int *)&mon[0x0F4],4,Mp,0); break;
    case 'A': Apply((int *)&mon[0x0C8],4,Mp,0); break;
    case 'D': Apply((int *)&mon[0x0CC],4,Mp,0); break;
    case 'H':
        if(Apply((int *)&mon[0x0C0],4,Mp,0)) break;
        *(int *)&mon[0x06C]=*(int *)&mon[0x0C0];
        break;
```

Stacks' data is accessed by raw pointer. There's no neat structure created this time, we must access memory manually at right offset and with proper data size. To learn structure of this data, we must move on to IDA. So open up database created by GrayFace and let's go.

Finding info in IDA

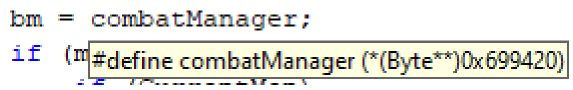
But what should we search for? It’s best to check out variables and constants used in function. Here’s part important for us.

```
int mn=GetVarVal(&sp->Par[0]); // So mn is index of stack
if((mn<-1)||(mn>41)){
    MError("\"!!BM:\"-monster index is incorrect (-1, 0...41).");
    RETURN(0)
}

bm = combatManager;
if (mn == -1)
    if (CurrentMon)
        mon = CurrentMon;
    else
        mn = M2B_GetMonNum(bm);

if (mn >= 0)
    mon=&bm[0x54CC+0x548*mn];
```

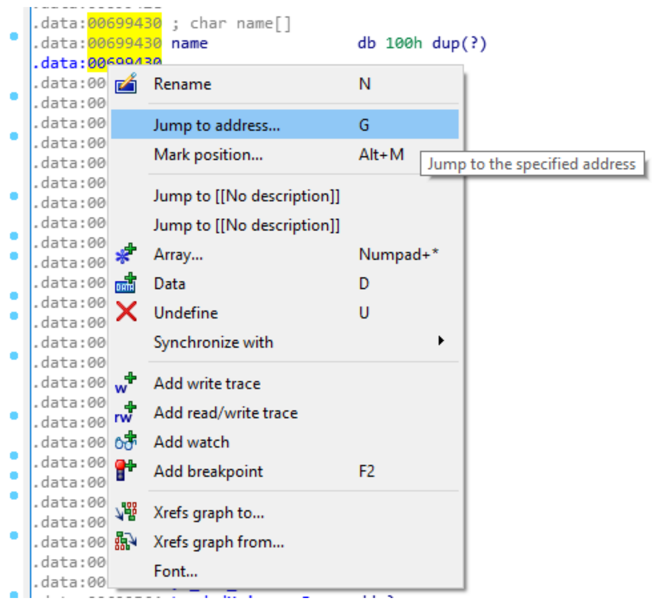
Inside variable `mon` we’ve pointer to particular stack data in game’s memory. As you can see, this data is derived from thing called `combatManager`, taken with offset. What is `combatManager`?



```
bm = combatManager;
if (m#define combatManager (*(Byte**)0x699420)
```

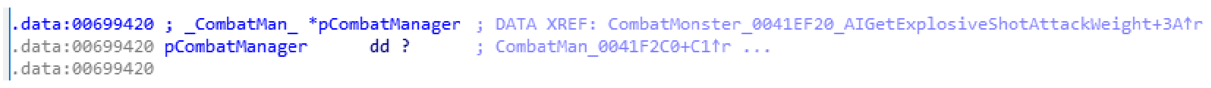
Pretty much what we’re searching for. Both address and structure name can be used as starting point to dig for information in IDA. So let’s start.

Rightclick on leftmost column and click “jump to address”. Type in address we’ve found: 0x699420



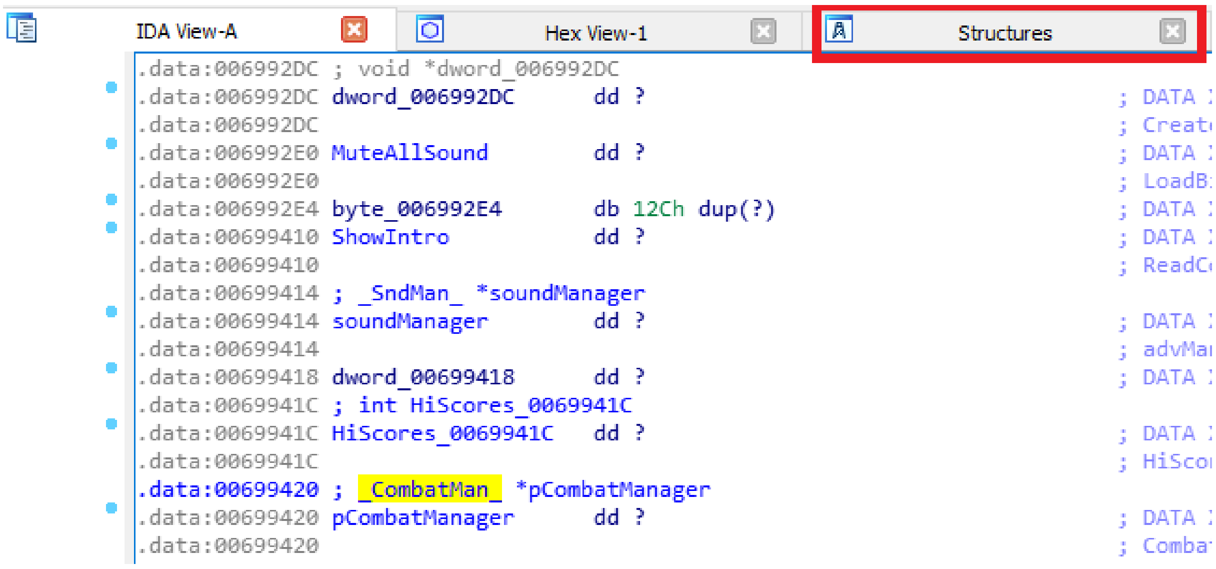
```
.data:00699430 ; char name[]
.data:00699430 name              db 100h dup(?)
.data:00699430
```

And the results are:



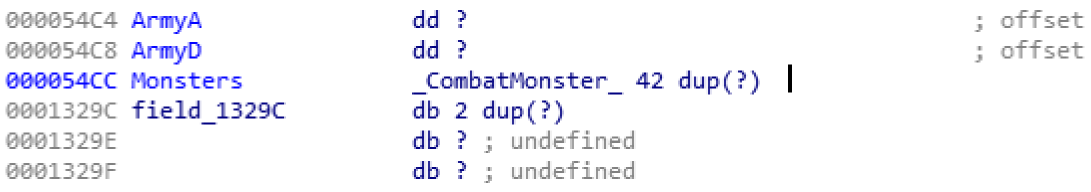
```
.data:00699420 ; _CombatMan_ *pCombatManager  ; DATA XREF: CombatMonster_0041EF20_AIGetExplosiveShotAttackWeight+3A↑r
.data:00699420 pCombatManager      dd ?       ; CombatMan_0041F2C0+C1↑r ...
.data:00699420
```

Two names, `_CombatMan_` and `CombatManager`. Let's search for it. It ain't function so there's no pseudocode. We expect this to be structure, right? Let's check structures



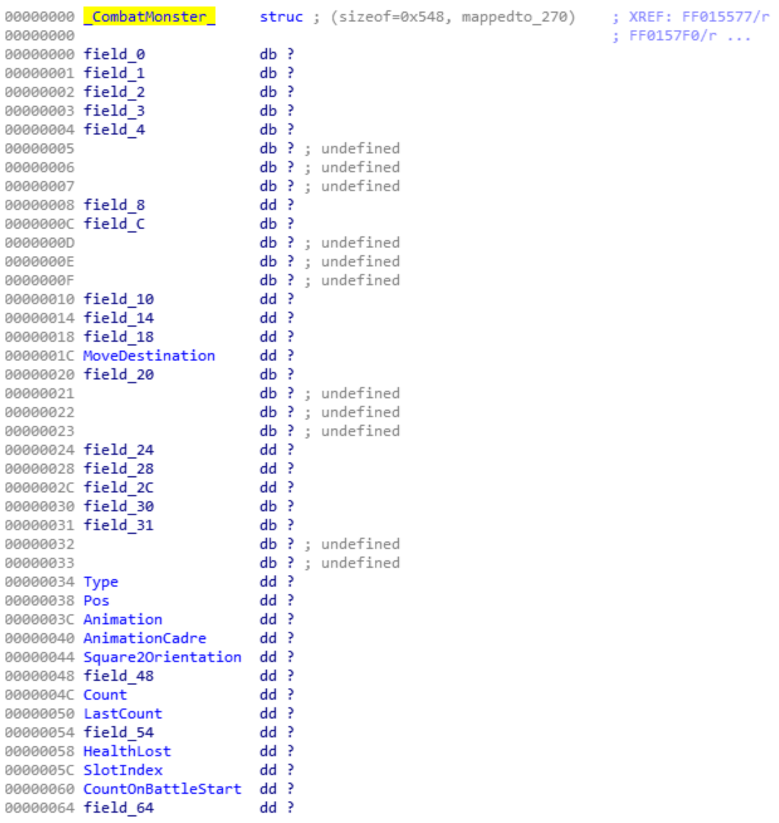
```
.data:006992DC ; void *dword_006992DC
.data:006992DC dword_006992DC      dd ?                                    ; DATA
.data:006992DC                                                             ; Creat
.data:006992E0 MuteAllSound        dd ?                                    ; DATA
.data:006992E0                                                             ; LoadB
.data:006992E4 byte_006992E4       db 12Ch dup(?)                          ; DATA
.data:00699410 ShowIntro           dd ?                                    ; DATA
.data:00699410                                                             ; ReadC
.data:00699414 ; _SndMan_ *soundManager
.data:00699414 soundManager        dd ?                                    ; DATA
.data:00699414                                                             ; advMa
.data:00699418 dword_00699418      dd ?                                    ; DATA
.data:0069941C ; int HiScores_0069941C
.data:0069941C HiScores_0069941C   dd ?                                    ; DATA
.data:0069941C                                                             ; HiSco
.data:00699420 ; _CombatMan_ *pCombatManager
.data:00699420 pCombatManager      dd ?                                    ; DATA
.data:00699420                                                             ; Comba
```

As expected, `_CombatMan_` is a complex structure. And here's the moment I asked: what is 0x54CC offset that you can see in line `mon=&bm[0x54CC+0x548*mn]`? Ah yes.



```
000054C4 ArmyA              dd ?                          ; offset
000054C8 ArmyD              dd ?                          ; offset
000054CC Monsters           _CombatMonster_ 42 dup(?)
0001329C field_1329C        db 2 dup(?)
0001329E                    db ? ; undefined
0001329F                    db ? ; undefined
```

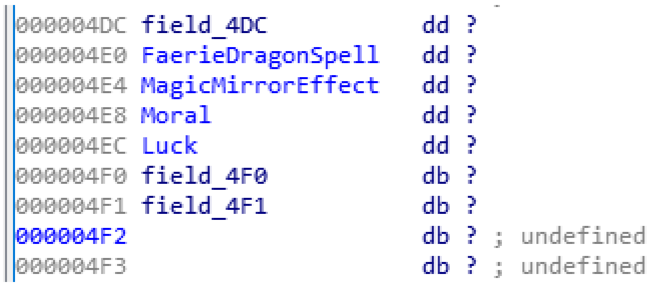
Quick search for `_CombatMonster_` shows the following:



```
00000000 _CombatMonster_    struc ; (sizeof=0x548, mappedto_270)    ; XREF: FF015577/r
00000000                                                            ; FF0157F0/r ...
00000000 field_0            db ?
00000001 field_1            db ?
00000002 field_2            db ?
00000003 field_3            db ?
00000004 field_4            db ?
00000005                    db ? ; undefined
00000006                    db ? ; undefined
00000007                    db ? ; undefined
00000008 field_8            dd ?
0000000C field_C            db ?
0000000D                    db ? ; undefined
0000000E                    db ? ; undefined
0000000F                    db ? ; undefined
00000010 field_10           dd ?
00000014 field_14           dd ?
00000018 field_18           dd ?
0000001C MoveDestination    dd ?
00000020 field_20           db ?
00000021                    db ? ; undefined
00000022                    db ? ; undefined
00000023                    db ? ; undefined
00000024 field_24           dd ?
00000028 field_28           dd ?
0000002C field_2C           dd ?
00000030 field_30           db ?
00000031 field_31           db ?
00000032                    db ? ; undefined
00000033                    db ? ; undefined
00000034 Type               dd ?
00000038 Pos                dd ?
0000003C Animation          dd ?
00000040 AnimationCadre     dd ?
00000044 Square2Orientation dd ?
00000048 field_48           dd ?
0000004C Count              dd ?
00000050 LastCount          dd ?
00000054 field_54           dd ?
00000058 HealthLost         dd ?
0000005C SlotIndex          dd ?
00000060 CountOnBattleStart dd ?
00000064 field_64           dd ?
```

So we're at home. Number on leftside is offset given in HEX, next to it you can see name of particular attributes withing structure. Not all are named, not all are known. All the names were added by creators of database, if something wasn't discovered by them in the past you need to reverse engineer it yourself – which is task much harder to do.

You can compare offsets in wog source with the ones in database. Also, pay attention to datasizes. At addresses 0x4E8 and 0x4EC you can find Moral and Luck – two attributes we've wanted to find, both are 4 bytes long so can be interpreted as (32bit) integers.



```
000004DC field_4DC          dd ?
000004E0 FaerieDragonSpell  dd ?
000004E4 MagicMirrorEffect  dd ?
000004E8 Moral              dd ?
000004EC Luck               dd ?
000004F0 field_4F0          db ?
000004F1 field_4F1          db ?
000004F2                    db ? ; undefined
000004F3                    db ? ; undefined
```

I've made simple script and it checks out. This is the way you can use IDA to help you understand original wog source and create your own. I think it would be much better to always use structures like we've seen in previous documents, but I'm not going to remake all the old source – it would be too much work for no ingame difference (aside from bugs I would inevitably add in the process.