This article will be one of an ongoing series -- as I have time to write them. I've spent the better part of this past weekend and this week focusing on game hacking. The target was an old classic, Age of Empires II: The Conquerors.

It is an old 2D RTS game where you build up an empire, build armies, make/break alliances, and so on. For a much more in-depth explanation than I care to provide see the Wikipedia article on its predecessor. One of the interesting things that I discovered while messing around with this game is that the statistics (all resource/villager/military/... counts, current age, population, etc) counts are stored in a structure that is indexed into for the appropriate player. This wouldn't be too interesting as a single player game, but Age of Empires is a multiplayer game, with a semi-big community of around 1200 active players. The same base classes/structures are shared between AI bots and active players, meaning there are no modifications to be made between single and multiplayer, in terms of hack development.

Finding the stat structure started off normally, by using Cheat Engine to search for the type of variable the player's resources were stored in, and what was writing to it. Doing the usual things like building/destroying buildings, collecting resources, and making units yielded a wide variety of results. In the end, there were multiple candidates for where values are being written from, but they were narrowed down to a very good function at .text:00555470.

```
 0055546a - fadd dword ptr [eax+edx*4]
 0055546d - lea eax,[eax+edx*4]
>>00555470 - fstp dword ptr [eax]
 00555472 - mov eax,[ecx+04]
 00555475 - test eax,eax
```

This is good candidate because the floating point stack is being utilized, and it was the only one like it that popped up when dropping resources off at the town center. Looking actively into this function yields clues about how it works. The important parts are reproduced below.

```
.text:0055544F                mov     eax, [ecx+0A4h]
.text:00555455                movsx   edx, si
.text:00555458                cmp     edx, eax
.text:0055545A                jge     loc_5554FE
.text:00555460                mov     eax, [ecx+0A8h]
.text:00555466                fld     [esp+4+arg_4]
.text:0055546A                fadd    dword ptr [eax+edx*4]
.text:0055546D                lea     eax, [eax+edx*4]
.text:00555470                fstp    dword ptr [eax]
.text:00555472                mov     eax, [ecx+4]
.text:00555475                test    eax, eax
.text:00555477                jz      loc_5554FE
```

This function is very interesting because it is a __thiscall and does not set up any sort of bp-based stack frame. ECX here is used without being initialized (hint for __thiscall) and the arguments are also referenced directly through the stack pointer. This is where the static code analysis ends though, and a debugger needs to be attached to follow exactly how everything behaves. OllyDbg happens to be one of the best for live code analysis.

```
0055544F  . 8B81 A4000000  MOV EAX,DWORD PTR DS:[ECX+A4]
00555455  . 0FBFD6         MOVSX EDX,SI
00555458  . 3BD0           CMP EDX,EAX
0055545A  .v0F8D 9E000000  JGE aoc.005554FE
00555460  . 8B81 A8000000  MOV EAX,DWORD PTR DS:[ECX+A8]
00555466  . D94424 0C      FLD DWORD PTR SS:[ESP+C]
0055546A  . D80490         FADD DWORD PTR DS:[EAX+EDX*4]
0055546D  . 8D0490         LEA EAX,DWORD PTR DS:[EAX+EDX*4]
00555470  . D918           FSTP DWORD PTR DS:[EAX]
00555472  . 8B41 04        MOV EAX,DWORD PTR DS:[ECX+4]
00555475  . 85C0           TEST EAX,EAX
00555477  .v0F84 81000000  JE aoc.005554FE
0055547D  . 66:85F6        TEST SI,SI
00555480  .v74 12          JE SHORT aoc.00555494
00555482  . 66:83FE 01     CMP SI,1
00555486  .v74 0C          JE SHORT aoc.00555494
```

Breakpoints set, everything can be observed as this function gets called. When I created a unit, the following values were in the registers when the first breakpoint hit.

```
Registers (FPU)
EAX 000000C6
ECX 103C79A0 ASCII "Dzc"
EDX 00000050
EBX 3F800000
ESP 0018D37C
EBP 00000050
ESI 103C0050
EDI 00000050

EIP 00555460 aoc.00555460
```

Here ECX is the "this" pointer. It points to the base of the calling class. When the EIP is 0x0055546A, the application had the following state

```
Registers (FPU)
EAX 1063BBD0
ECX 103C79A0 ASCII "Dzc"
EDX 00000050
EBX 3F800000
ESP 0018D37C
EBP 00000050
ESI 103C0050
EDI 00000050

EIP 0055546A aoc.0055546A

C 1  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 1  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000293 (NO,B,NE,BE,S,PO,L,LE)

ST0 valid 1.0000000000000000000
ST1 empty -??? FFFF 1A191818 40180F1C
ST2 empty -NAN FFFF B69A9E9D 0000003D
ST3 empty -??? FFFF 00000000 8282DA00
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 1.3958333333333332590
ST7 empty 100.00000000000000000
```

1.0f was loaded into ST0. However, when I dequeued a unit, -1.0f was loaded into ST0. The second parameter is therefore some sort of flag for whether a unit is being queued or dequeued. This value is added to [EAX+EDX*4], and the result stored in the address that EAX points to. Looking at what is inside [EAX] then yields what is more or less a holy grail.

32-bit float			
99549.00	99999.00	99999.00	99999.00
-24.00000	0.0	0.0	0.0
0.0	0.0	0.0	29.00000
0.0	0.0	0.0	0.0
0.0	0.0	0.0	29.00000
1.000000	0.0	9.194445	102.0000
103.0000	101.0000	0.0	0.0
0.0	0.0	500.0000	0.0
66.00000	0.0	0.0	1.600000
175.0000	5.000000	0.0	178.0000
22.00000	0.0	0.0	0.0
1.000000	1.000000	0.3000000	1.000000
0.0	0.0	0.0	0.0
1.000000	0.0	0.0	0.0
0.0	0.0	104.0000	1.000000
0.0	0.0	20.00000	32000.00
0.0	5.000000	1.000000	1.000000
0.0	0.0	1.000000	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.3000000	1.000000
9.000000	0.0	0.0	2.000000
3.000000	0.0	0.0	1.000000
700.0000	0.0	4.000000	0.0
0.0	0.0	0.0	0.0
3.000000	0.0	7882.000	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	145.0000
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
7140.000	742.0000	0.0	0.0
0.0	0.0	145.0000	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
15.00000	25.00000	25.00000	0.0
0.0	0.0	0.0	0.0
0.0	1.000000	1.000000	30.00000
0.0	0.0	0.0	0.0
0.0	0.0	1.122440e-42	3.825545e-43

While the values may look nonsensical at first, this is actually a layout of a structure that holds a ton of statistics about a player.

The first four members of this structure correlate to the players food, wood, stone, and gold. Then comes the population left before hitting the cap, an unknown value, the current age, and so on (see player_stats.h for my listings). These values can continue to be found by setting memory breakpoints on an individual one or a region, followed by performing an action in the game and observing what breakpoints are triggered. Since the base of this structure is a member of the calling class, it can always be found at (base address of class + 0xA8), as evidenced in the disassembly of the function. Just looking at the base address of the class yields some interesting information.

```
00000362
103B7FF0
1099E400
1099E3E0
1099E2B0
11D00A90 ASCII "Dic"
106098A0
028793A0
10999530
10999030
11D81040 ASCII "qwerty"
00000001
00000201
000000C6
1099EA30
00000800
00000000
00000000
00000000
00000000
10C27688
```

In addition to having pointers to some ridiculously large structures (possibly covered in the future), the player's name is also listed at (base address of class + 0xA8). Sidenote: The address pointing to it is different since I started another game instance between taking the two screenshots. This provides identifying information for each class that calls .text:00555470. Then the theory is that if I hook .text:00555470, I can store all of the pointers to every player in the game. This is true because this function is called by every player on the games start, and also throughout the game. Using Microsoft's Detours library makes this extremely easy. The hook function looks as follows

```
__declspec(naked) void resources_changed_hook(short int res_type, float
usage_type, int unused) {
        __asm {
                pushad
                mov eax, temp_pointer
                mov dword ptr[eax], ecx         //temp_pointer->base_pointer
points to calling class
    }
    temp_pointer->player_name = (char*)(*(temp_pointer->base_pointer + (0x98
/ sizeof(DWORD_PTR))));
    temp_pointer->player_stat = (player_stats*)(*(temp_pointer->base_pointer
+ (0xA8 / sizeof(DWORD_PTR))));
    if(insert(&base_pointers, temp_pointer) == true)
        temp_pointer = (item_set*)HeapAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY, sizeof(item_set));
    __asm {
                popad
                jmp resources_changed
        }
}
```

The stack is preserved, the address of the calling class is stored, the offsets calculated and set, and the calling function inserted into a set (no repetitions allowed).

The actual structure is a shell of the calling class and stores only the class pointer, the statistics structure, and the players name

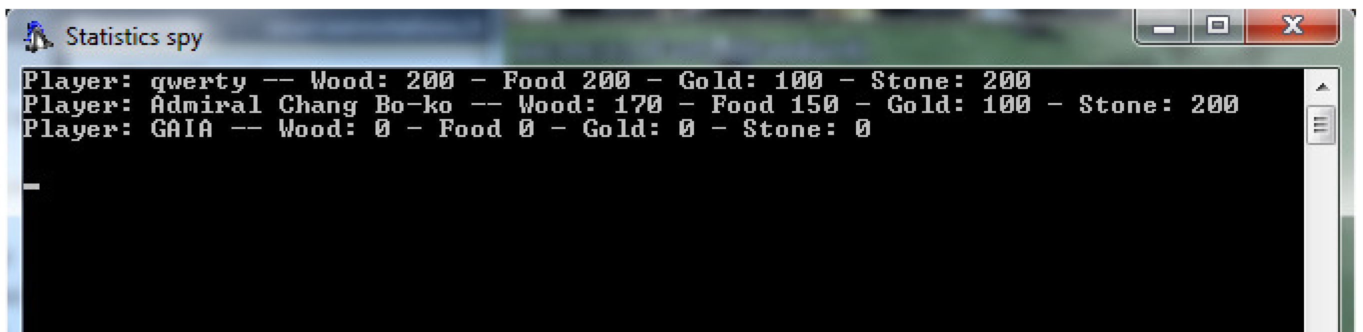
```
typedef struct ITEM_SET {
        int* base_pointer;
        char* player_name;
```

```
        player_stats* player_stat;
        ITEM_SET *next;
} item_set, *pitem_set;
```

This can/will all be expanded and redone as more of the calling class is reverse engineered. The stored items can then be simply traversed like a list and the data for each player printed out.

```
void print(item_set** head) {
        if(*head == NULL)
                return;
        item_set* node_ptr = *head;
        while(node_ptr != NULL) {
        printf("Player: %s -- Wood: %1.0f - Food %1.0f - Gold: %1.0f - Stone:
%1.0f\n", node_ptr->player_name,
            node_ptr->player_stat->wood, node_ptr->player_stat->food,
node_ptr->player_stat->gold, node_ptr->player_stat->stone);
                node_ptr = node_ptr->next;
        }
        printf("\n");
}
```

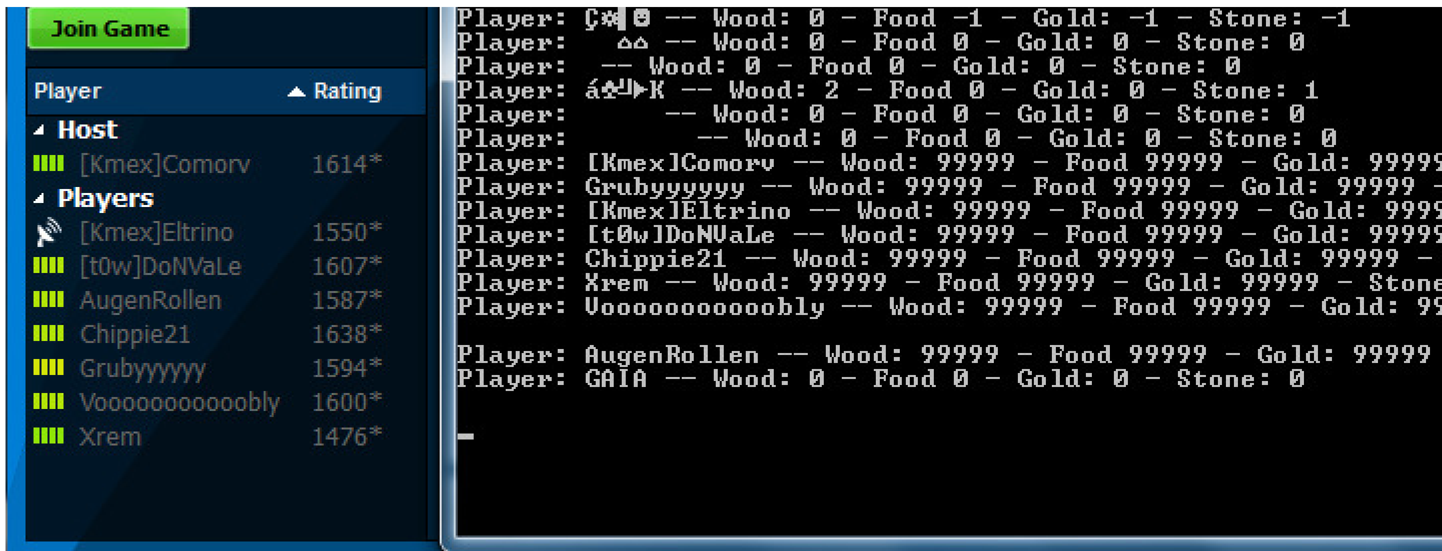
Here is an example of it in action:



Using this, a player is able to modify their own stats in addition to reading the stats of others. Doing something like

```
while(node_ptr != NULL) {
    if(strcmp("qwerty", node_ptr->player_name) == 0) {
        node_ptr->player_stat->food = 10000.0f;
        break;
    }
    node_ptr = node_ptr->next;
}
```

Would work fine on single player. However, on multiplayer this would cause an out of sync error. Since each player keeps a copy of the other players information in their game, any unwarranted change would cause the game to go out of sync. However, simply reading memory will still be fine. The player_stats structure consists of 198 members with roughly half of them documented by me. Additional help is welcome.



A screenshot from a CBA game at Voobly -- technique still holds.

Warning: The version posted here will result in a ban from Voobly. Their anti-cheat checks functions for hooks and will ban any user found to be modifying the functions in memory. This is still extremely easy to get around using hardware breakpoints and a different DLL injection technique with no keyboard hook -- but that might be another part of this series. The risk takers who are uninitiated with those topics may want to try to attach the hooks prior to an in-game lobby launch and detach immediately upon entering the game to beat the anti-cheat scan. This is definitely not recommended however. Also, the fact that it is in a separate window makes this more of a proof of concept than an actual functional hack until further work/posts are done on the game.

Possible future articles as time permits:

Hooking DrawTextA or DirectDraw functions to draw text on the screen

Reversing the protocol and packet structure

Bypassing Voobly's anti-cheat system (large hints given)

The source code to the hooking DLL can be found here.

A downloadable PDF of this post can be found here.