

# Reverse engineering 'Black Desert Online' (2. Speed -> local entity)

Jan 24, 2019

## Intro

In this post we will discuss how to dynamically find data values such as movement speed, and backtrace to our local entity for further analysis. If you followed our advice from the previous post, you will have messed around ingame and figured out the basics of the game mechanics. From doing that, we figured out how to affect our current movement speed through a buff. We personally started out as the character *Striker*, that has a spell called *Flash Step*, that according to documentation, gives us a 30% movement speed buff for 10 seconds. This means that we have complete control over the variable, which is crucial if we have decided to find it dynamically by searching for changing values in memory. This kind of analysis will be done by *CrySearch*. This is one the many ways to find points of interest in video games, and would be classified as *dynamic analysis*.

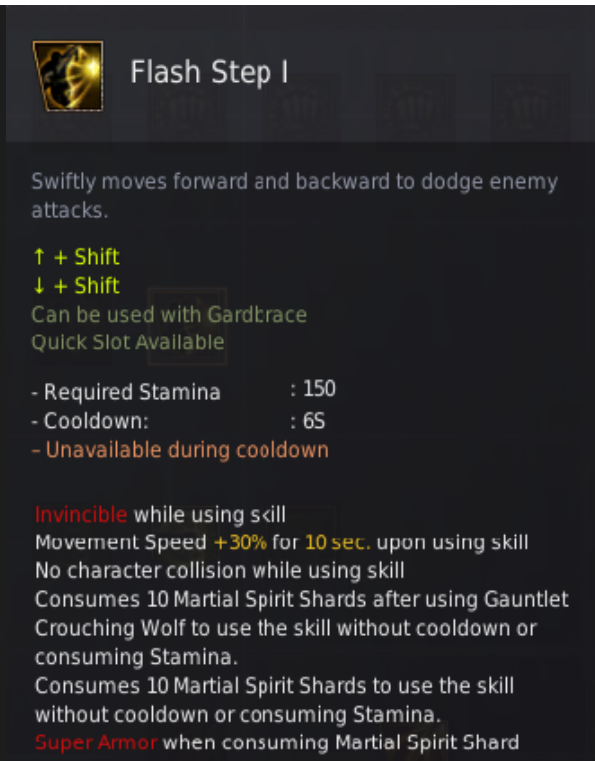


Fig.1 - Flash Step 1

## Finding the ~~needle~~ data in the ~~haystack~~ memory

To set a precedent for the workspace we will be using for this post, open *CrySearch*, *x64dbg* and *ReClass*, if you do not have these tools installed, refer to [part 1](#) of this reverse engineering series. To begin our search of the movement speed variable(s), attach *CrySeach* to the game process *BlackDesert64.exe*

Open process list	Attach

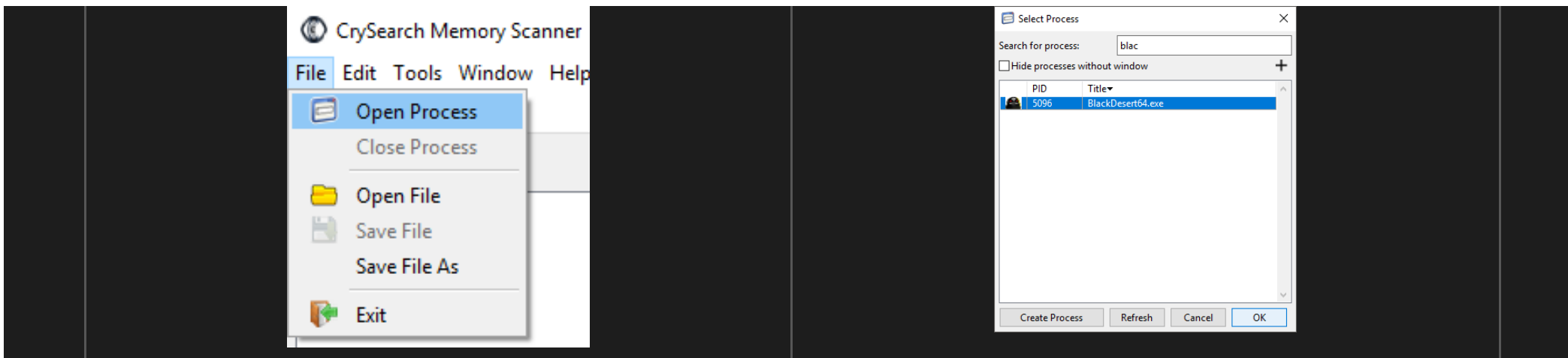


Fig.2/3 - CrySearch open process / CrySearch attach process

*CrySearch* is capable of what we call a *memory scan*, which maps out the entire memory space of the process (presumably paged to disk), to easily search for changes in memory. To do a memory scan, we will need to assume a few things, such as: will the data we're looking for be aligned? What kind of type would the information be stored as? If we were looking for a player name, we should not be searching for a floating-point value, and etc. Since we are looking for the movement speed of our local entity, it is most certainly stored as either an integer or a floating-point value (whether it is double-precision or single-precision). The velocity of our movement can originally be many different values, or stored as a coefficient to a separate speed variable, which is why we will begin our search by mapping out the memory space and not looking for any specific value in memory. To do so, hit the *New Scan* magnifying glass and select: *Size->Integer (4 Bytes)* and *Type->Unknown Initial Value*. This is as slow as it sounds, and will quickly use a lot of the very scanty resources left by the game running at full speed in the background, so sit back and let *CrySearch* do its job.

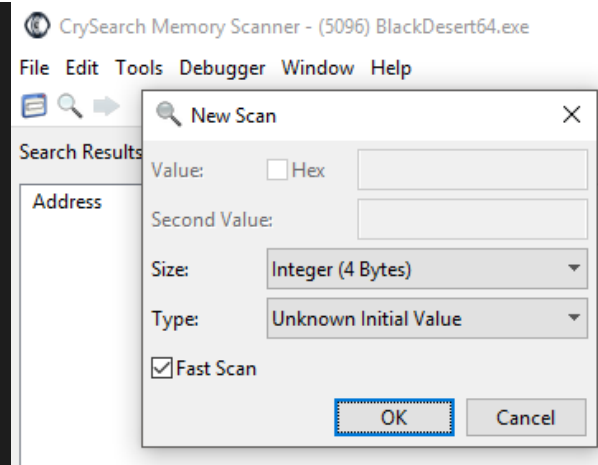
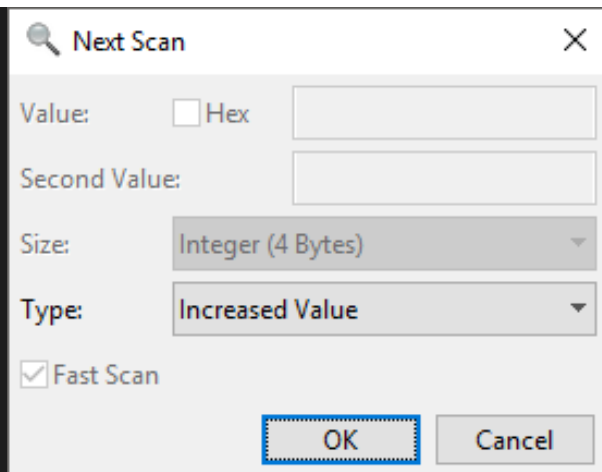


Fig.4 - CrySearch unknown value scan

As we finish, we will be greeted by hundreds of thousands of values, which for us is absolutely useless, as we cannot parse all of that data by ourselves. Now, remember the movement speed buff from earlier? This allows us to 'add' to our desired movement speed value, which comes in **very** handy, as we can isolate the respective data from the enormous amount of junk we will get when we search for an unknown value in memory. To begin our isolation of the desired value, we will need to increase our movement by using the buff, after doing so, we will have a 10 second window to search for an increased value. To do so, hit the *Next Scan* arrow to the right of the original *New Scan* magnifying glass, and select *Type->Increased Value*. If the scan does not finish in a timely manner, use the buff again, as the cooldown is lower than the respective duration of the movement speed buff. When the scan has finished, wait till the buff has expired and start another scan, this time *Type->Decreased Value*, as the movement speed must have decreased back to the original value when the buff expires.

Increased-value scan	Decreased-value scan



Next Scan

Value: ☐ Hex

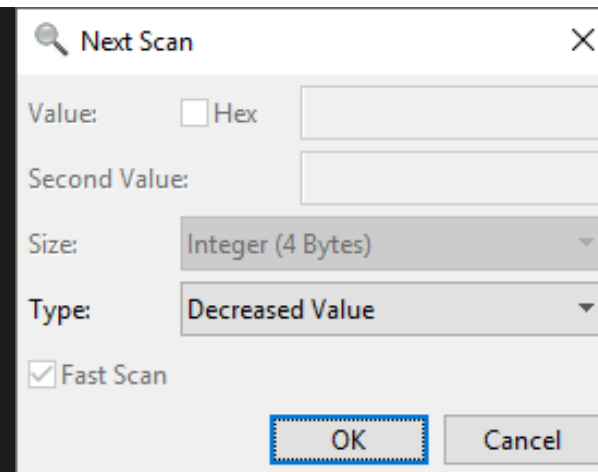
Second Value:

Size: Integer (4 Bytes)

Type: Increased Value

☒ Fast Scan

OK Cancel



Next Scan

Value: ☐ Hex

Second Value:

Size: Integer (4 Bytes)

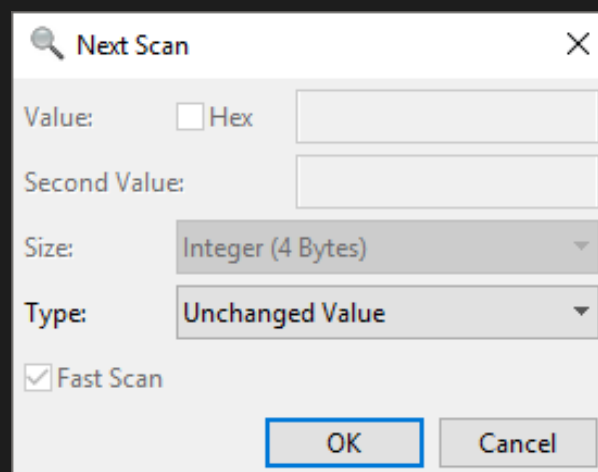
Type: Decreased Value

☒ Fast Scan

OK Cancel

Fig.5/6 - CrySearch increased-value scan / CrySearch decreased-value scan

You can also mix up these types of scans with an unchanged-value scan when you know the movement hasn't changed, to quickly filter out irrelevant data. To do so, start another scan and select *Type->Unchanged Value*



Next Scan

Value: ☐ Hex

Second Value:

Size: Integer (4 Bytes)

Type: Unchanged Value

☒ Fast Scan

OK Cancel

Fig.7 - CrySearch unchanged-value scan

By repeating this process, we can narrow down the movement speed to certain values. When doing these types of scans you will rarely end up with only the desired value, in our case we are left with three values that change accordingly. Two of these values are simply going from 0 to 1 when the buff is active, which might indicate it is a boolean value for whether or not the buff is currently activate. The third value is interesting; it goes from 0 to 300000, which seems like a very odd value for movement speed, but what do we know, we just began analyzing the game? To confirm our suspicion, we can write a quick program to repeatedly modify the value from 0 to 300000 and see if our movement speed increases; it does. **We succesfully found our local entity's movement speed.** The result we found looks like this:

Description	Address	Value	Type
	B3BA9D38	300000	4 Bytes
	B3BA92A0	1	4 Bytes

Fig.8 - CrySearch scan result

This means that our movement speed is stored at `B3BA9D38`, and has the value of `300000` when the respective movement speed buff is active. As we previously mentioned, the scan result is not denoted in a distinct green color, marking it as a static global value, but rather a dynamically constructed structure.

Done? Not really, we currently know where it resides in memory, but we can also very quickly conclude that this knowledge is not very useful, as the memory page it resides in was dynamically allocated, which means that we cannot assume it's presence at that specific virtual address in the future, or between restarts. If this address was stored as a global value in the *BlackDesert64* binary, our work in this section would be done.

## Tracing back to the local entity

Preferably, we would want to find the local entity, as it is very useful for game hacking. If we're lucky, this movement speed value is stored as a member of the local entity, if not, we're going to have a hard time manually finding the base of the local entity. Some games, like *Overwatch* by *Blizzard* separate entities into a lot of different classes that seemingly have no relation, making reverse engineering very hard. To begin searching for the local entity, we should find out where the game reads our local movement speed, and see if can trace it back to a global variable, which presumably would be our local entity. To trace instructions accessing our movement speed, let's open up x64dbg and set a hardware breakpoint on memory access of the respective movement speed

address. Go to the dump window and hit *Ctrl+G*, then paste in the address of our movement speed value. When x64dbg has navigated the location in memory, right click on the first byte of the movement speed value and hit *Breakpoint->Hardware, Access->Dword*

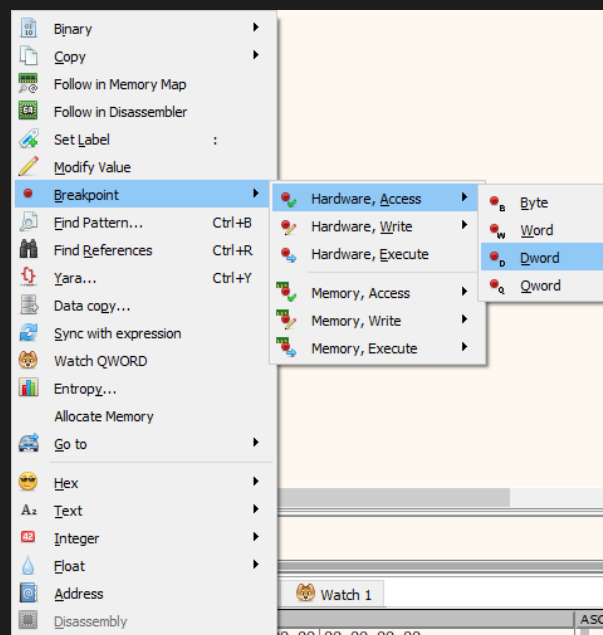


Fig.9 - x64dbg hardware breakpoint on access (dword)

This breakpoint will trigger whenever ANY thread in the current process accesses these specific four bytes (size of 32-bit integer). You can manually log addresses using x64dbg by writing instruction pointers down and continuing, but x64dbg has a great set of tools to automatically log and silently continue the respective hardware breakpoint so it is completely automatic for us reverse engineers. Going to the breakpoint tab, we can edit our current hardware breakpoint by right-clicking on it and hitting *edit*, or hitting *Ctrl+E*. This will allow us to modify the *Log Text* and the *Break Condition*, which respectively is the formatted text output to the log tab and the condition where x64dbg halts execution completely, waiting for user interaction. Our log text should be capable of telling us which instruction accesses our movement speed, or rather, what the current instruction pointer is. The instruction pointer when x64dbg breaks will be the instruction right after the one that accesses our movement speed, so keep that in mind when looking at the disassembly later.

x64dbg formatting allows us to get the current instruction pointer by using `{p:cip}`, and for clarity sake we appended `Movement speed:` so we can be sure that the addresses we are reading in the log are not irrelevant addresses.

The break condition is the evaluated expression that decides whether or not x64dbg halts execution. If this expression is evaluated to anything but zero, x64dbg breaks, so let's set it to zero and stop it from breaking. Make sure your breakpoint settings look like this:

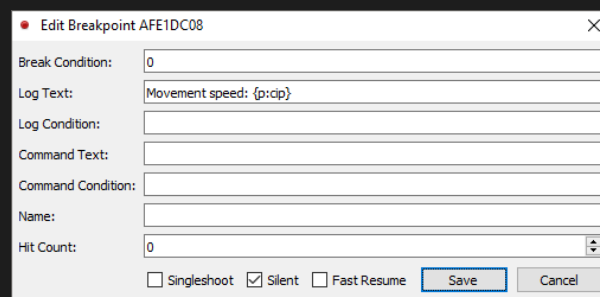


Fig.10 - x64dbg edit breakpoint

When this is done, continue playing the game and let x64dbg silently log all relevant instructions...

```
Movement speed: 00000001408C0D3D
Movement speed: 0000000140952D98
Movement speed: 000000014097E375
Movement speed: 00000001408C43F0
Movement speed: 00000001408C4522
Movement speed: 00000001408C4599
Movement speed: 00000001408C45A2
```

Done. This list is only the unique entries in the log file, as you will have a ton of entries for repetitive calls. Let's look at the first instruction in the log: `00000001408C0D3D`

sub\_1408C0D20:



```

xor     eax, eax
mov     r10, rcx
mov     [rdx], rax
mov     [rdx+8], eax
mov     byte ptr [rdx+0Ch], 1
movsxd  rax, dword ptr [rcx+6514h]
mov     r9d, [rcx+0BF8h]          <--
mov     r8d, [rcx+rax*4+652Ch]
xor     r8d, [rcx+6520h]
sub     r9d, r8d
mov     [rdx], r9d
movsxd  rax, dword ptr [rcx+6518h]
mov     r8d, [r10+0BFCh]
mov     ecx, [rcx+rax*4+692Ch]
xor     ecx, [r10+6524h]
sub     r8d, ecx
mov     [rdx+4], r8d
movsxd  rax, dword ptr [r10+651Ch]
mov     ecx, [r10+rax*4+6D2Ch]
xor     ecx, [r10+6528h]
mov     eax, [r10+0C00h]
sub     eax, ecx
mov     [rdx+8], eax
test    r9d, r9d
jg      short loc_1408C0DA5
test    r8d, r8d
jg      short loc_1408C0DA5
test    eax, eax
jle     short loc_1408C0DA9

```

Okay, looking at the instruction that accesses our movement speed, it has a base pointer at rcx and the member offset `BF8`, subtracting this value from our movement speed address *should* show us the beginning of the local entity, so let's try looking at this

address in ReClass. To do that, attach ReClass (*File->Attach to Process*) and type the address of the type into the default class that ReClass creates. The address is marked with red in the figure below:

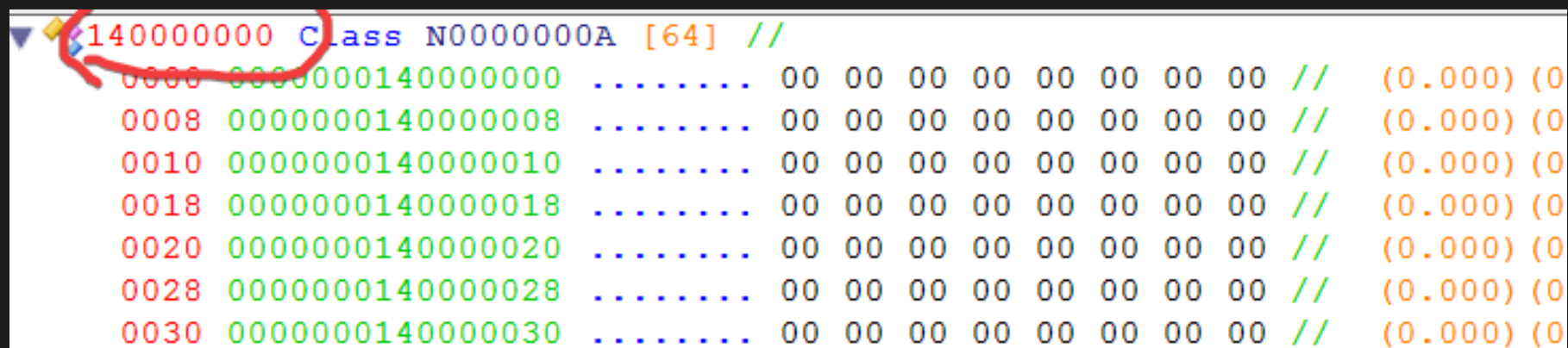


Fig.11 - ReClass.NET local entity

Now, after attaching and inputting the address:

```

00000000B979E020 Class N00003569 [2112] //
0000 00000000B979E020 ...B... A0 FC B4 42 01 00 00 00 // (90.493) (5414124704|0x142B4FCA0) -> BlackDesert64.exe.142B4FCA0 ge::SelfPlayerActorProxy : ge::PlayerActorProxy :
0008 00000000B979E028 ..m...# 00 E9 6D C8 E5 C5 23 C5 // (0.000) (-4241328833352046336|0xC523C5E5C86DE900)
0010 00000000B979E030 .L..... 91 4C A5 C5 00 00 00 00 // (-5289.571) (3315944593|0xC5A54C91) -> <HEAP>C5A54C91
0018 00000000B979E038 ..... 00 FC FF FF 00 00 00 00 // (0.000) (4294966272|0xFFFFFC00)
0020 00000000B979E040 ..... 00 00 00 00 00 00 00 00 // (0.000) (0)
0028 00000000B979E048 4..... 34 00 00 00 00 00 00 00 // (0.000) (52|0x34)
0030 00000000B979E050 (. .... 28 00 00 00 00 00 00 00 // (0.000) (40|0x28)
0038 00000000B979E058 ...UUUU 00 00 00 00 55 55 55 55 // (0.000) (6148914689804861440|0x5555555500000000)
0040 00000000B979E060 ..... 00 00 00 00 00 00 00 00 // (0.000) (0)
0048 00000000B979E068 ..... 00 00 00 00 00 00 00 00 // (0.000) (0)
0050 00000000B979E070 ..... 00 00 00 00 00 00 00 00 // (0.000) (0)
0058 00000000B979E078 ...I.... U 01 04 49 00 01 00 55 55 // (0.000) (6124896592740287489|0x5500010000490401)
0060 00000000B979E080 ...UUUU 01 00 01 FF 55 55 55 55 // (0.000) (6148914694083117057|0x55555555FF010001)
0068 00000000B979E088 .J..... 90 4A 15 B7 00 00 00 00 // (0.000) (3071625872|0xB7154A90) -> <HEAP>B7154A90 L'
0070 00000000B979E090 ...UUUU E1 FF E0 FF 55 55 55 55 // (0.000) (6148914694097797089|0x55555555FFE0FFE1)
0078 00000000B979E098 ..... 0F 00 00 00 00 00 00 00 // (0.000) (15|0xF)
0080 00000000B979E0A0 ..... 0F 00 00 00 00 00 00 00 // (0.000) (15|0xF)
0088 00000000B979E0A8 ..... 00 EF E0 C7 AA FE 7F 83 // (0.000) (-8971171923247436032|0x837FFEAC7E0EF00)
0090 00000000B979E0B0 ..... 00 00 00 00 00 00 00 00 // (0.000) (0)
0098 00000000B979E0B8 .9..... 2E 39 A5 00 88 13 00 00 // (0.000) (21474847308078|0x138800A5392E)
00A0 00000000B979E0C0 .9..... 2E 39 A5 00 00 00 00 00 // (0.000) (10828078|0xA5392E) -> <HEAP>A5392E
00A8 00000000B979E0C8 ..... U 00 00 00 00 E0 EA 7F 55 // (0.000) (6160901063059701760|0x557FEAE000000000)
00B0 00000000B979E0D0 .!*..... C0 21 2A CC 00 00 00 00 // (0.000) (3425313216|0xCC2A21C0) -> <HEAP>CC2A21C0
00B8 00000000B979E0D8 .9..... 2E 39 A5 00 88 13 00 00 // (0.000) (21474847308078|0x138800A5392E)
00C0 00000000B979E0E0 .9..... 2E 39 A5 00 00 00 00 00 // (0.000) (10828078|0xA5392E) -> <HEAP>A5392E
00C8 00000000B979E0E8 ...UUUU 00 00 00 00 55 55 55 55 // (0.000) (6148914689804861440|0x5555555500000000)
00D0 00000000B979E0F0 .*..... F0 24 2A CC 00 00 00 00 // (0.000) (3425314032|0xCC2A24F0) -> <HEAP>CC2A24F0
00D8 00000000B979E0F8 ..... 00 00 00 00 00 00 00 00 // (0.000) (0)
00E0 00000000B979E100 ....Q... 00 00 00 00 51 01 00 00 // (0.000) (1447403978752|0x15100000000)
00E8 00000000B979E108 ....`ppp 00 00 00 00 60 70 70 70 // (0.000) (8102099287258693632|0x7070706000000000)
00F0 00000000B979E110 .\..... A8 08 5C 8A 00 00 00 00 // (0.000) (2321287336|0x8A5C08A8) -> <HEAP>8A5C08A8
00F8 00000000B979E118 ..... 00 00 00 00 00 00 00 00 // (0.000) (0)

```

Fig.12 - ReClass.NET local entity

Bingo. The first field of this class is a pointer to the virtual method table of this class. As we saw in the previous post, *Black Desert Online* ships their release builds with runtime-type information, and ReClass takes care of that. This tells us that the local entity is a:

```
ge::SelfPlayerActorProxy : ge::PlayerActorProxy ...
```

We can very quickly conclude that this is a larger structure for the local entity, and not some inner structure related to movement speed, thanks to the class name provided by the runtime-type information and the presence of a text-pointer at `+68` that contains our character name, that is for obvious reasons blurred. Let's figure out how the game accesses the local player, so we can locate the global. To find out how the game decides the `rcx` value from the previous assembly dump, open up IDA and check cross-references to the previous function `00000001408C0D3D` (hit X while your cursor is inside of the function).

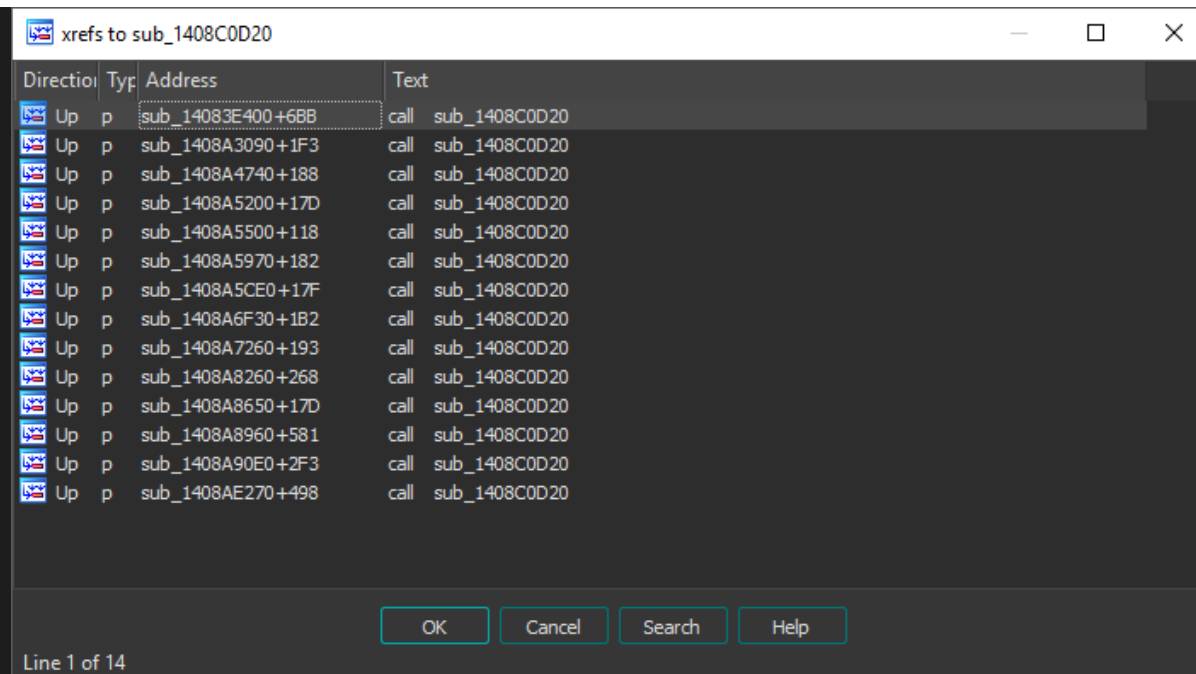


Fig.13 - IDA x-ref

Let us look at the first one:

Reference to sub\_1408C0D20: (sub\_14083E400)

```

mov    rcx, [rbp+238h+var_18]    <-- Local entity stored on stack
call   sub_1408C0D20            <-- Accesses movement speed at [rcx+BF8], where rcx is local entity

```

The local entity is stored on the stack in this function, let's find out what the value is.

Further up in the same function, the stack variable is initialised:

```

mov    rax, cs:qword_14341FE28    <-- Local entity
mov    [rbp+238h+var_18], rax

```

Bingo. `qword_14341FE28` is the local entity.

## [How not to implement] Speedhack preventions

If you were a curious hacker like us, you would try to write the maximum value for a signed integer (assuming this speed coefficient can be negative, otherwise unsigned integer) to the movement speed and observe something resetting the movement speed periodically, making our speedhack stutter. This is bad, this could mean that the server knows we are speedhacking and telling the client to go fuck itself, but as we will show in this paragraph, that's not the case.

To figure out what is overwriting the movement speed value, set a hardware breakpoint on the address, this time for instructions writing to the movement speed value.

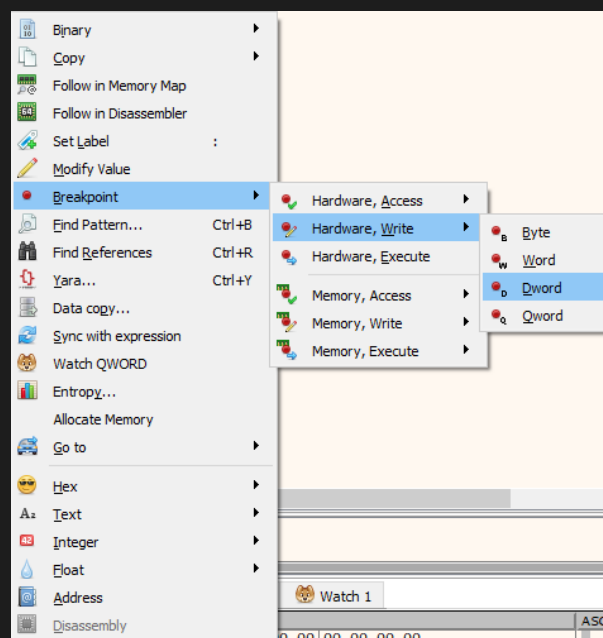


Fig.14 - x64dbg hardware breakpoint on write (dword)

By repeating the aforementioned technique of logging instruction pointers, this will yield one entry:

sub\_1408C4360:



```
movzx    edx, al
movzx    eax, r15b
movzx    ecx, r15b
mov      [rdi+rax*4+6520h], ebx
mov      [rdi+rax*4+6514h], ebp
mov      [rdi+rax*4+0BF8h], esi    <--
mov      [rdi+rax*4+0C04h], r12d
test     r15b, r15b
jz       short loc_1408C4451
sub      ecx, 1
jz       short loc_1408C442C
```

Okay, this function seems to be the overwriting movement speed value with some other calculated value. We don't really care what that value is, we just need to prevent the game from overwriting our speedhack. To do so, we can go to the beginning of the function and patching it with a simple `ret` instruction. This is a very lazy way of preventing anti cheat measures, as we can potentially break the game by nulling whole functions like this, but it's a great way to start and see what else this function does. After patching it, nothing eyecatching happened, but our speed hack is not getting interrupted anymore.

We cannot be sure if the intention of the function is to explicitly prevent speed hacking, but it is writing the correct value periodically and we don't want that.

vmcall

vmcall  
virtualmachinecall@gmail.com

 vmcall  
 vm\_call

vmcall's personal blog.