# Hacking Videogames For Fun

By
FoxMaccloud

2022

# Contents

# 1 Introduction

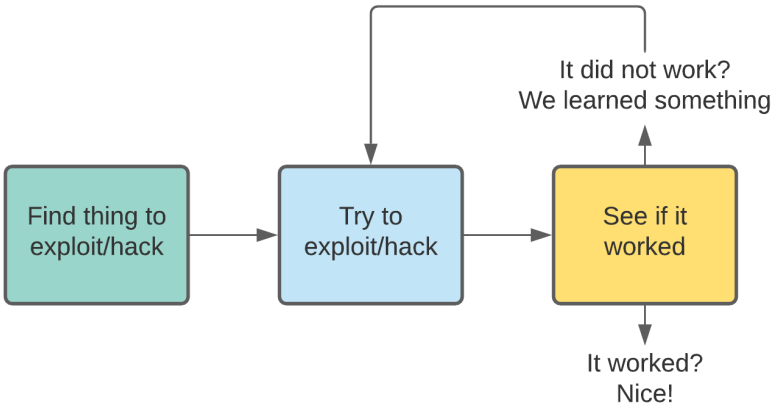
## 1.1 Motivation

If you have ever played any video games such as Counter-Strike, World of Warcraft, Call of Duty and similar games you would eventually stumble upon someone who cheats in video games. In this paper I'm going to try to demystify what goes on behind game hacks, how computers work, some of their techniques and how these hackers affect the game and others playing the game.

There are different reasons why people cheat in video games. Some people cheat because it's fun, others because of financial incentives and others to "boost" or save time. Some games like counter-strike has a ranking system and boosting is the act of skipping to higher ranks by guaranteeing that you are not loosing with cheats. Some games like World of Warcraft it can take an enormous long time to reach the highest character level where the "endgame"starts for many players. Since it takes for some 100 hours to reach this level, they use bots to play for them. In 2018 there was a player named Nikhil "Forsaken" Kumawat that was cough cheating in a professional league. He was using a cheat that made it easier for him to aim at his opponents. The incentive to cheat in professional games increases exponentially since these leagues can have a price pool up to 2 million dollars. In the end, the goal of all these hacks is to give the player a competitive advantage. In most cases the ones that creates the cheats aren't the ones that uses them, but rents them out in exchange for a fee or just outright sells them to "vendors" that rents them out.

## 1.2 Hacking Process

The process of making hacks for games looks something like this;




It looks like a simplified version of the scientific method, which is is. If you e.g., want to change something in your save file, you change it, then see if it worked. If it did not work, you need to try again. If it worked, then you achieved the goal of what you wanted to hack. This is a pattern that repeats for all steps in game hacking.

There are many skills one need to learn how to hack videogames. Looking from a top level, these skills can for the most part be grouped into a few categories.

## 1.3 Memory editing

Memory editing is the bread and butter for hacking video games. It's used for all types of hacks. Since the game runs on your local machine means that all information about the game is in memory. For many offline games, one can search for data values with tools such as cheat engine to find e.g., health value. One can then rewrite this value to be whatever level of health you want. Or you can change your XYZ position to teleport.

## 1.4 Assembly editing

In memory you will also find the code of the game. Assembly editing is a more advanced version of memory editing where you don't edit the values in the memory, but you edit the assembly of the program. For this you need to both read and write assembly. Assembly editing is a lot more powerful then just regular memory editing. If you have 100 health in a game with memory editing you could write 100 health to the health address every millisecond. This is also known as "freezing" a value. Issue with this approach is that if you took 101 damage in 1 tick, you would die because it's larger then your total health pool. With assembly editing you could make it so that damage never happened in the first place.

## 1.5 Packet editing

Packet editing is when you edit packets between you the client and the server. The idea is that you lie to the server about what you have done. You can attempt to buy an item in a game for 100 game currency. You can then intercept this package and tell the server that you want to buy it for 50 game currency instead. This in most cases wouldn't work due to server side checks, but common vulnerabilities are integer overflows.

## 1.6 Botting

Botting is when the computer plays the game for you. People often use bots to automate tasks that are "tedious" such as "farming" resources. These resources could be used by the player or often be sold for RMT (Real Money Trading) or game currency. The problem with bots is that they often destroy the integrity and economy of the game. This is often where the money is made and the lawsuits happen. Botting also often used all of the previous skills mentioned. Some people sometimes makes a living by running hundreds of bots then selling the items gathered by the bots for real money.

# 2 The computer

You might have have heard that a program is just a bunch of 1's and 0's. This is like saying the human body is just a bunch of atoms. This statement is true, but it's not very useful and it's not the best way to look at programs. We use terms like bytes, ints, strings, floats, etc. This is more useful then talking in 1's and 0's.

## 2.1 The operative system

The job of the operative system is to manage the hardware and manage the programs running on the computer. When we run a program, the operative system load a copy of the game from disk into ram. The operative system has power over everything in the computer. Because of this we can abuse this power and tell the OS to do things like edit memory. For windows we can use the windows api for this. Example would be WriteProcessMemory() is a windows api function that you can use to write a select value into an address. Without going to deep into the windows api, here is an example on how WriteProcessMemory can be used to "freeze" the integer value 5000 into an address.

```
1  #include <iostream>
2  #include <Windows.h>
3
4  int money_to_add = 5000;
5  int address = 0xDEADBEEF;
6
7  int main()
8  {
9    HWND hwnd = FindWindowA(NULL, "Game_Window");
10   if (hwnd == NULL)
11   {
12     std::cout << "Cannot find window!" << std::endl;
13     exit(-1);
14   }
15   else
16   {
17     DWORD procID;
18     GetWindowThreadProcessId(hwnd, &procID);
19     HANDLE handle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, procID);
20     if (procID == NULL)
21     {
22       std::cout << "Cannot obtain process ID!" << std::endl;
23       exit(-1);
24     }
25     else
```

```
26      {
27        while (true)
28        {
29          WriteProcessMemory(
30            handle,
31            (LPVOID)address,
32            &money_to_add,
33            sizeof(money_to_add),
34            0
35          );
36          Sleep(1);
37        }
38      }
39    }
40    return 0;
41  }
```

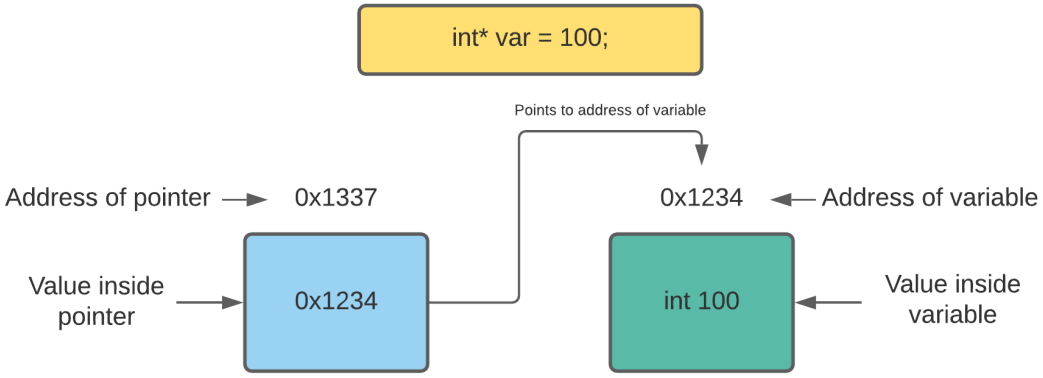
A big problem with this approach is that the address for the money in this game is that the address 0xDEADBEEF is going to change every time you start the game, meaning you also going to modify your code each time. There are multiple ways to solve this, but one of them is to use "offsets" or "signatures".

## 2.2 Offsets

Offsets are basically what they sound like. They are the amount by which something is off something. In software we often see and use offsets from the base address of the program, base address of a module or the base address of a class or struct.

## 2.3 Pointers and multi-level pointers

A pointer in C++ is simply a variable that contains the address another variable in memory. We use pointers to save memory space and get faster execution.




Pointers are very useful and make it so we get less execution time, we can easily work with the original value when passing values to functions, instead of a copy. If we want to move around big sets of data into e.g., a bigger array from a small array. We can just move the reference to the value instead, we can return more then one value from a function, search and sort large datasets easily and with less cost. It's also "closer to how a regular computer works". Since a pointer is just contains an address, we can also access the items next to variable pretty easy.

```
1 struct Entity
2 {
3     const char* name;
4     float hp;
5     int gold;
6     vec3 position;
7 };
8
9 Entity* entity; // notice: This is a pointer!
```

If we want to access the entities gold in this example, we can simply access it like so, &entity + 0x5 = gold. This is because the size of a char* is 1 byte and a float is 4 bytes.

Multilevel pointers are essentially the same. The difference being literally the multi level. This is important because often games keeps important values at different places in different structures. These structures are then often linked together with pointers at different offsets from the base of a structure. Using this we then only need to find the offset to the base entity and from there we can follow down until we find the desired address.

```
1 struct Inventory
2 {
3     unsigned int gold;
4     std::vector<item>items;
5 };
```

```
6
7  struct Entity
8  {
9      const char* name;
10     float hp;
11     Inventory* inventory;
12 };
13
14 Entity* entity;
15
16 void func()
17 {
18     entity->hp = 100;
19     entity->inventory->gold = 2000;
20     entity->name = "FoxMaccloud";
21 }
```

Using the "arrow operator" the compiler automatically knows that when we e.g., want to access our gold. We from the base address of Entity add hex value 0x5 to the address, then dereference. We are then at our inventory structures address. We can then again add hex value 0x0 because gold is the first element in the structure and dereference again. We can then access our gold. Note that our gold is unsigned. This means that this value cannot be negative, and will then become the max possible integer value once we go below. Some games have this vulnerability that can be exploited for a lot of in game money.

## 2.4 Signatures

Signatures are literally just an array of bytes. We use these array of bytes (AOBs) to find a sequence of bytes in memory which matches the one defined in your signature. When we scan for these signatures we look through the code and not the data associated with the process. A signature can look like

```
Assembly instructions;
0:  48 89 d8                mov    rax,rbx
3:  48 31 c0                xor    rax,rax
6:  48 f7 f0                div    rax

Signature: 48 89 D8 48 31 C0 48 F7 F0
```

Sometimes you want to make a signature of something which contains an address such as a jump or similar, we often replace these with wild cards. In most signature schemes, a wildcard looks like a "?".

```
Assembly instructions;
0:  48 89 d8                mov    rax,rbx
3:  e9 00 00 00 00          jmp    0xABCDEF

Signature: 48 89 D8 E9 ? ? ? ?
```

Using these signatures we make a function which first scans byte by byte. Once it finds the first byte of our signature it will then check each subsequent byte against our signature. If the compared by is a wildcard it's always true. If the signature doesn't match, it will start over again from the next byte and the cycle continues. Once the search stops by either finding the pattern or run outside of good regions of memory. It will return the address of the first byte in the signature or it will return null or throw an error depending on implementation.

## 2.5 Assembly

When looking at programs we want to hack, we often look at the programs in the assembly format. This is because the programs are compiled to "machine code". Also known as 1's and 0's. Even when we in most cases cannot revert this back to the original source code, we can atleast disassemble it into assembly, which is the lowest level programming language. Since assembly mirrors what the computer does this means that all programs can be represented in assembly. Compared to other higher level programming languages, assembly gives you more control over the processor, but it also requires more code to perform relative basic tasks. Assembly is cpu architecture specific. This means that e.g., devices that uses the ARM processors uses a different instruction set compared to regular x86_64 processors most people have in their desktops. Since the assembly language works so close with the CPU, it's important to know a bit about how the processors work.

## 2.6 Registers

Registers are the "working memory" for the assembly language. These registers hold values during computation, but data is not ment to be stored in these registers for long time. Data can be loaded into registers from memory, then the results can then be saved into more "long term storage" such as ram. Depending on instruction sets these registers may be different. I'll be mostly focusing exclusively on the x86 and the x64 instruction sets for this paper. Backwards compatibility is maintained between these two by allowing access to subsets of registers, can treat half of the x64 register as it was a x32 one. The x64 architecture have several general purpose registers and several special purpose register sets.

These registers are

- RAX, RBX, RCX, RDX and R8-R15

  These registers are known as GPRs or General Purpose Registers. It acts as a temporary storage location which holds an intermediate value in mathematical and logical calculations. The result is then stored in a register overwriting the previous value. These registers can hold a total of 8 bytes. When using 32 bit registers we are then accessing the lower 4 bytes of the

register. Using 32 bit registers in 64 bit code is common as not all values needs to use up the whole registers. When writing a 32 bit value into a 64 bit register, the top 4 bytes will then be overwritten with null.

- RIP

  IP in RIP stands for Instruction Pointer. RIP points to the next instruction to be executed, and supports a 64 bit flat memory model. When the program runs, you will see the RIP registry change even when there are no instructions that puts anything into this registry. [1][p.2]

- RSP

  SP in RSP stands for Stack Pointer. The job of the stack-pointer is to keep track of the last item that is pushed onto the stack. The stack is used to keep track of return addresses for subroutines and more [1][p.3]

- RBP

  RBP is the base pointer. The job of the base pointer is to save your position in the stack. When a function is called, we need to keep track of where we are because the stack pointer keeps changing all the time depending on items that gets pushed and poped off the stack. RBP is very often used to set up a "stack-frame" I'll get into more detail what this is in the next chapter.

- RSI

  The RSI is mostly used for source indexing for string operations. SI means Source Index.

- RDI

  RDI is pretty much the same as RSI, but instead it is the Destination Index.

- XMM0-XMM15

  XMM registers are different from the other registries as they are floating point registers and works with float and double values. Unlike the normal registers, the floating-point registries are bigger. The XMM registers are 16 bytes in size while the others are 8 bytes. The data type double is still 8 bytes and float still 4 bytes, but this means that XMM registers allow parallel operations on four single or two double precision values per instruction. Floating point registries also have their own instructions that only works with floating points. Another thing with XMM registries is that you cannot put values directly into them. To put a value into the XMM registry, you need to copy that value into the registry from another memory address or another XMM registry.

- RFLAGS

  The RFLAGS registers stores flags which is used by operation instructions such as cmp etc. These flags are;

- CF - Carry

  Opertion generated a carry or borrow.

- PF - Parity

  Last byte has even number of 1, else 0.

- AF - Adjust

  Denotes binary coded decimal in-byte carry

- ZF - Zero

  Result was 0

- SF - Sign

  Most significant bit of result is 1

- OF - Overflow

  Overflow on signed operation

- DF - Direction

  Direction string instructions operate

- ID - Identification

  Changeability denotes presence of CPUID instruction

In game-hacking we rarely have to worry about these flags. [1][p.3]

## 2.7 The Stack

The stack is a region of the memory that behaves as a Last In First Out (LIFO) data structure. We can push items onto the stack to save them and pop them off the stack to read them. The stack is implemented as a contiguous array in memory. Pushing and popping items off the stack is accomplished by moving an index known as a stack pointer. This pointer will always point to the top of the stack as mentioned earlier. It's important to not forget that we still have random access to this memory and not limited to only pushing and popping off the stack.

when we call a function, RBP with the RSP registry sets up something called a "stack-frame". It accomplishes this by first pushing the old RBP onto the stack, then overwriting the base pointer with the stack pointer, then subtracting the stack-pointer by 0x10 or 16 bytes. By doing this we have reserved 16 bytes for local variables below the base-pointer, while arguments that gets sent with the function gets put into higher memory from the base-pointer. Also the return address gets pushed onto the stack so the function knowns where to return once the function completes.

## 2.8 Shared instructions

In programming we often have a lot of shared instructions. These are very straight forward. In programming we never want to write twice when they accomplish the same thing. Even if it's the player, another human or a might dragon, they all probably have some health pool, and they all probably have a way to take damage.

```
1 void damage(Actor& actor, int damageAmount)
2 {
3     actor.health -= damageAmount;
4 }
```

In this code we can see a function that takes two arguments. The first is the actor and the second the amount of damage taken. When this function is called it subtracts the amount of damage taken from the health. It is important when one want to make cheats that one can determine what actor the actor argument is. If you want to make some sort of invincibility hack, you can replace the function with nops or do nothing, but the problem in doing so is that it will do it for all actors and not just yourself. No actors in the game will now be able to take damage. That's bad if you want to gain an unfair advantage, but great if you want to pacify your game.

## 2.9 External hacks

External hacks use windows api functions such as WriteProcessMemory() (WPM) and ReadProcessMemory() (RPM) to interact with the game process's memory. To do this you need to ask the kernel to give you a handle to the process by using OpenProcess() with the Process Access Rights you require, typically PROCESS_ALL_ACCESS. The handle is a required parameter for RPM/WPM. Kernel mode anticheats can easily block external hacks by using ObjRegisterCallbacks to block handle creation [2]. RPM/WPM is slow because you have the overhead of the API calls into the kernel. You should limit the frequency of these calls and store as much information locally as possible to increase the performance of your external hack. If the game has no method of detecting RPM making an overlay ESP is a good way of making an undetected external ESP because you only need RPM to be undetected.

Pros of external:

- In my opinion none compared to internal unless you just want to super quickly patch some bytes and then close the hack

Cons of external:

- Super easy to detect because of the open process handle
- Harder to use especially for beginners (WPM/RPM, getting the PID, etc.) though easy to master because it has no potential
- Less potential
- Slow

## 2.10 Internal hacks

Internal hacks are created by injecting DLLs into the game process, when you do this you have direct access to the process's memory which means fast performance and simplicity. DLLs gets injected into processes by the operative system through calls such as LoadLibraryExW() or their derivatives. Games often hook these windows functions that checks whenever the game is loading something that it shouldn't. We can bypass this by performing something called "manual mapping" which is essentially the same as LoadLibrary, but is instead implemented by the hacker. On Linux systems, one can use shared objects which is the Linux equivalent of windows dynamic linked libraries (DLLs).

When you are internal you create pointers to objects, typecast them and point them to objects in memory. Then you can access variables of that object easily through the pointer.

Pros of internal:

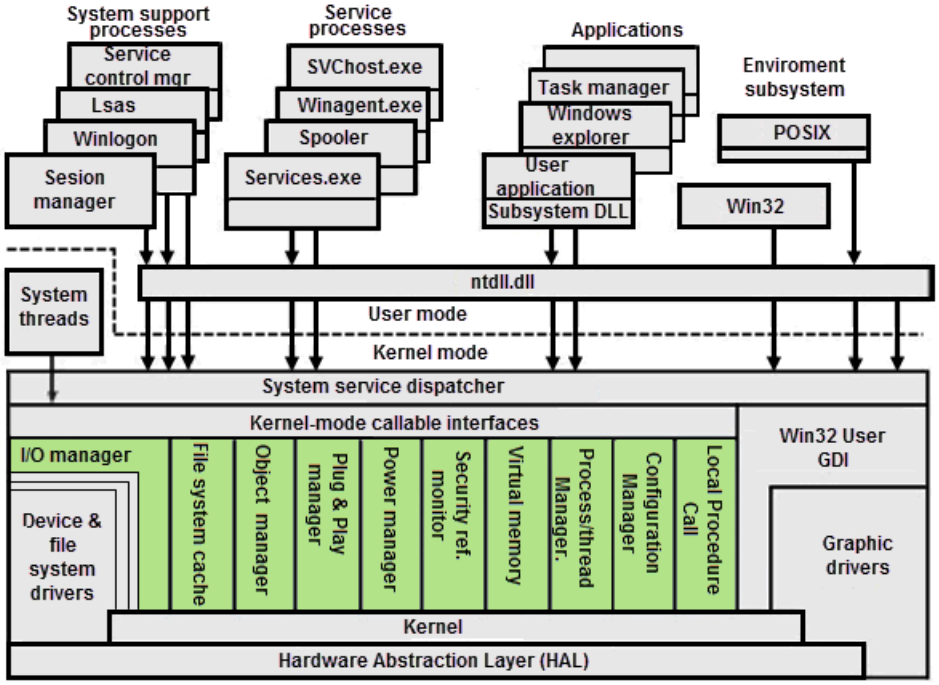
- Sick performance
- Easy to start off with
- Much potential
- Can be super sneaky and almost impossible to detect if done properly

Cons of internal:

- Hard to master
- Easier to detect when you don't know what you're doing

## 2.11 Kernel level

There is also another level of interacting with processes by using the kernel. When you run a process, there are two different spaces these can run within. Those two are user space and kernel space. All normal programs runs in user space while core operating system components runs in the kernel. The kernel has it's own privileged part of memory that is not accessible from user mode and executes with privileged status on the cpu [3].



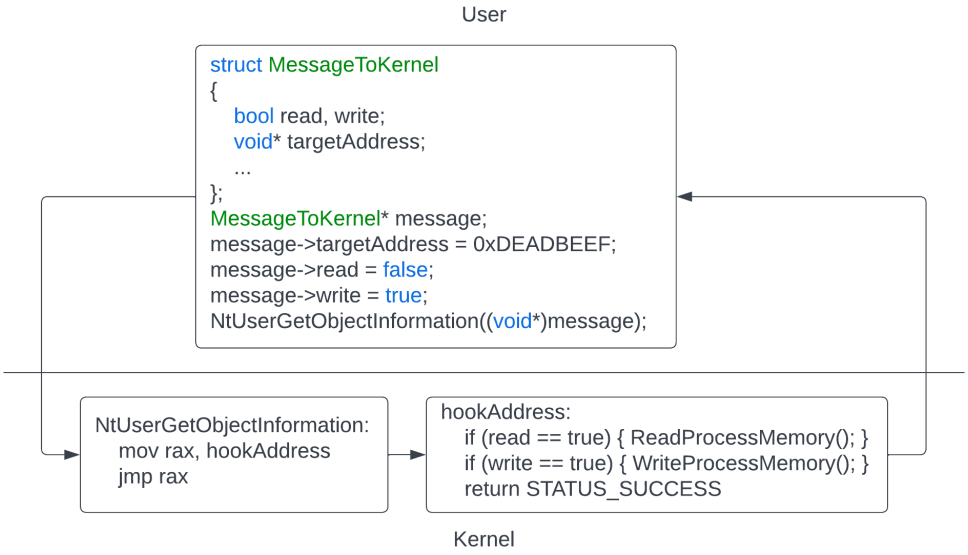

A normal process in the user space, all has it's own private address space that usually doesn't interact with any other processes memory. This is not the case for the kernel. This means if we write bad code the entire operative system will crash. Unlike a normal process the kernel space also shares the entire memory space. The kernel is also not limited to this memory, but has full access to all the memory. Even tho user space cannot directly access the kernel, it can still indirectly do calls to the kernel through system calls (syscalls). Many functions such as Read/WriteProcessMemory() requires the kernel to preform tasks on your behalf. Looking at these functions on msdn, we can see that it uses kernel32.dll. Unlike the name suggest, kernel32.dll itself is fairly high level and is not actually in the kernel. Kernel32.dll does call(s) to ntdll.dll which then interfaces with the kernel. Ntdll in itself is just a dll that contains wrappers for system calls.

Programs written to run in kernel space are called drivers. In windows they have the .sys extension. However in windows, once you have created your driver and you try to load it you will run into a problem. Windows requires your driver to be signed with a security certificate in order for the OS to load it. To obtain one of these certificates, you need to apply for one which has a lot of requirements, then purchase it. For development purposes you can set windows in "TESTSIGNING" mode. In this mode, you could load any kernel driver. Most anticheats requires that you have driver signing on. Since there are a lot of drivers out there in the field, there are guaranteed some vulnerable driver out there. Some drivers takes data from user space and brings it into kernel space is potentially vulnerable [3]. "But you can't just load your driver, you need to manually map it because it is not digitally signed. These vulnerable kernel drivers must have valid security certificates. By utilizing a valid & certified driver, you can manually map your unsigned driver without issue. Microsoft or the

Certificate Authorities can decide to reject these certificates at any time, making them no longer work, but that is extremely rare." [3].

Once you have developed your kernel driver and used vulnerable drivers to map them, you need a way to communicate between your driver and user space. There are several ways to do this, but a common one used is IOCTL (Input/Output Control). IOCTL is a system call for device specific I/O operations which cannot be expressed with regular syscalls. With IOCTL, we can open a handle to a device with CreateFile. Now depending on the driver implementation we can send and receive messages to and from the kernel by using DeviceIoControl, ReadFile and WriteFile.

Another way of communicating with the kernel would be to hook functions in kernel. When we call functions in usermode some of those functions might call functions in the kernel with an argument. By hooking these functions, we can then pass a message with those arguments to the kernel and our kernel hook can handle that message.




So why do we even go into the kernel? The main reason is anti-cheat. Since usermode cannot see into the kernel, we have essentially bypassed all usermode anticheats, sort of. There are still ways to for anticheats to detect kernel drivers. However we can disable these undetected from the kernel. In most cases kernel drivers are unnecessary and/or overkill, but this is only true for games that doesn't come with their own kernel driver. This is because whatever you do in usermode is not able to hide from the kernel.

# 3 Runtime modifications

Most changes we game hackers do happens at runtime when the game is loaded into memory. This also means if we do something wrong, like changing the code wrong. Simply restarting the game will remove all of our changes and give us a fresh start. There are many "runtime modifications" we

can do. Our earlier example with WriteProcessMemory()is an example of this runtime modification, however we can also outright change the functionality of the game by patching the game.

## 3.1 Patching

Patching is the act of overwriting the games code with your own. This can be done for reasons like changing the games functionality or remove one. Since we are trying to edit data in the .text section we first need to change the permission of this area so we can write new data here. We can then overwrite the bytes. After we have overwritten these bytes, we restore back the old read only protection.

```
1 DWORD64 to_patch = 0xDEADBEEF;
2 int len_to_patch = 5
3 void* new_opcodes = "\x90\x90\x90\x90\x90"
4
5 DWORD oldprotect;
6 VirtualProtect(to_patch, len_to_patch, PAGE_EXECUTE_READWRITE, &↘
    oldprotect);
7 memcpy(to_patch, new_opcodes, len_to_patch);
8 VirtualProtect(to_patch, len_to_patch, oldprotect, &oldprotect);
```

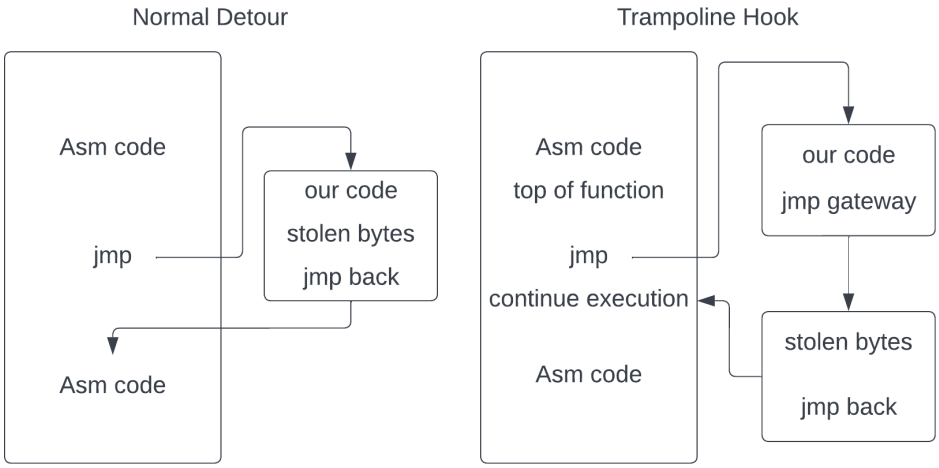
While we use patching for simple changes, we can also change the control flow i.e., redirect the flow of execution of the program by patching in jumps. With these jumps we can make the code skip sections of the gamecode or we can even make it jump to your own code. When we use these patches to jump, it's called hooking.

## 3.2 Hooking

Hooking covers different techniques for altering the flow of an application or other software. It gives us the ability to read and modify functions. When making hooks, we have to make sure we do not corrupt the stack and registers. Example, the games assembly does div rax. If we place a hook before this division instruction and we overwrite the rax with null in our hook, the game would crash as it will try to divide by zero. There are many ways to hook functions and I'll go over some.

- Detour & Trampoline hooks

  Detour hooks and trampoline hooks are the most common ones and in my opinion the easiest. A detour hook is also commonly referred to as a mid-function hook. To do a detour, you simply patch is a jmp with your desired place, execute our code and the bytes we overwrote, then jump back to where we came from with the offset after the jump to avoid an infinite loop. A trampoline hook is essentially the same, but here we also have a "gateway" that handles our overwritten bytes and make sure we don't have recursion. These hooks is very fast and very easy to implement.

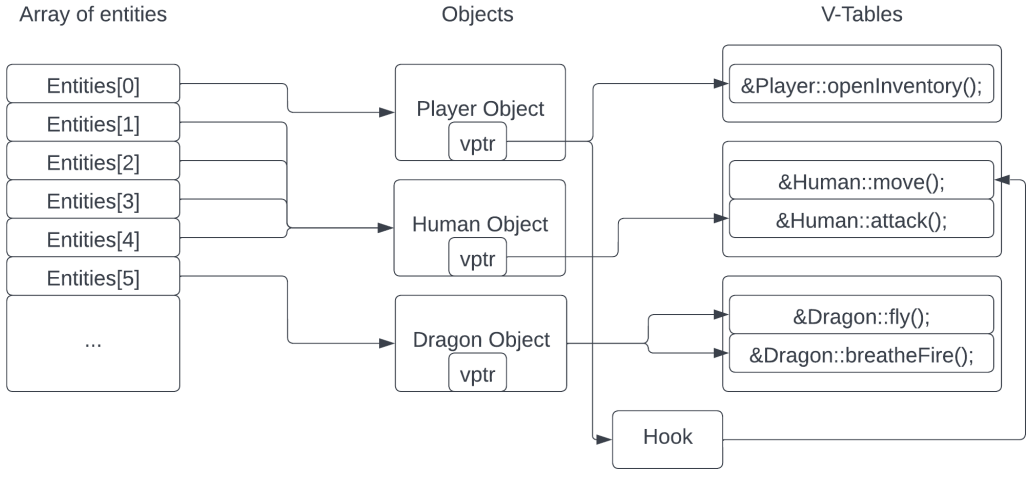



- IAT/EAT hooking

  Import/Export Address Table (IAT/EAT) is based on how the PE files are working on windows. The PE address table contains pointers to APIs and is adjusted by the PE loader when the file is executed. When you want to hook functions using this method, you can simply replace a certain api with your hooked function.

- VMT hooking

  VMT hooking stands for Virtual Method Table hooking. A vtable is just a set of pointers which each points to a different function. Objects call these functions in the vtable. To do a vmt hook we simply overwrite these pointers with our own destination. This way we redirect the execution flow. Any classes that uses a virtual function will have a virtual method table. After we have ran our custom code, we can call the original function and we can continue as normal.




You will often find these vtables within the .data section of the game, this is important to remember as we don't need any VirtualProtect() calls as it will already have read/write per-

missions. However this is completely engine based.

- HWBP hooking (VEH)

  Hardware breakpoints are also another way to redirect the execution flow. To do a hardware breakpoint hook, you have to place an exception handler. These hooks are very versatile because they do not require any code modifications. Hardware breakpoints is implemented as the name suggest, at hardware level. When you use these kinds of breakpoints we use different kinds of registers called "debug registers". These registers are named DR0 to DR7. Hardware breakpoints however are limited to max 4 at the time. When we hit these breakpoints, we can add an VEH exception handler with AddVectoredExceptionHandler() where we have our hook code.

- PageGuard hooking (VEH)

  PageGuard hooking is one of the stealthiest ways to redirect the execution flow. This works a lot like HWBP in that you have to register an exception handler and trigger that exception. Unlike HWBP, the way we trigger this exception is by using VirtualProtect() to change that memory page's protection to include PAGE_GUARD. This will cause the exception STATUS_-GUARD_PAGE_VIOLATION to be thrown whenever any of the memory in the memory page is executed. The problem with pageguard hooking is that it's incredible slow and should only be used for functions that are rarely called.

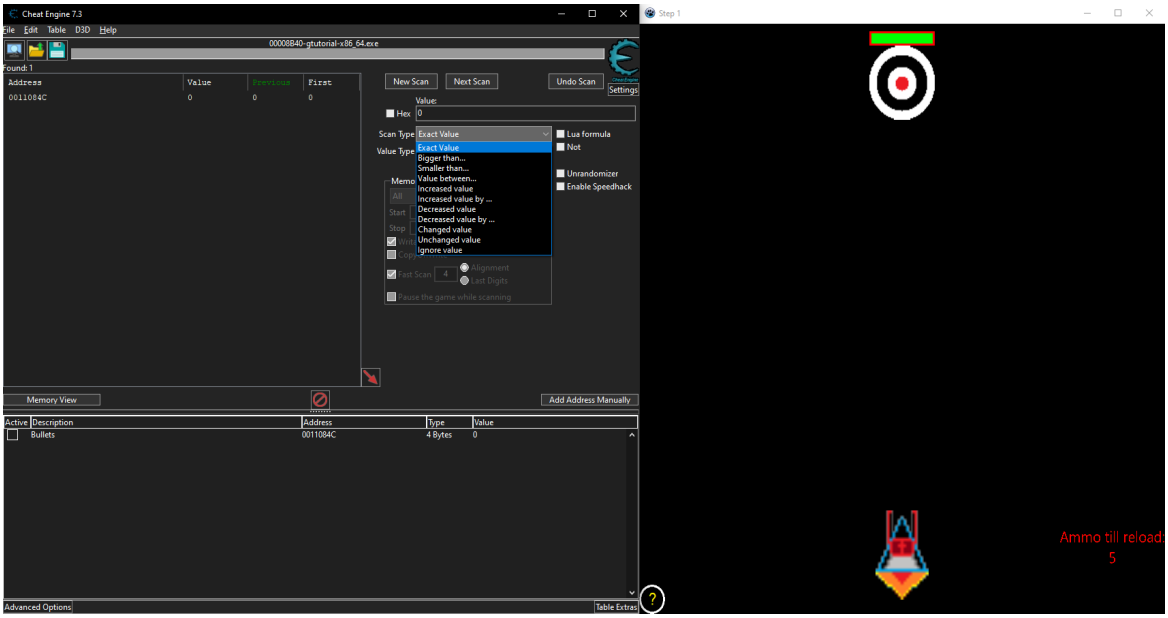
- Forced Exception hooking (VEH)

  Forced exception hooking is just the act of manipulating the game into throwing any kind of exception. It can be both modifying the data and/or modifying the code, deepening on how stealthy you want this hook to be. An example of forced exception is you can take the device pointer and set it to null. This should trigger a nullptr dereference and your VEH exception handler can take over. Lets say you see an instruction like; mov rax, [0xDEADBEEF] div rax. If you write null to 0xDEADBEEF before this instruction, you will trigger a divide by zero exception and your exception handler can take over. Forced exception is very broad and requires the hacker to be creative.

# 4 Reverse engineering

Reverse engineering is the most fundamental skill when hacking video games. This is where most time will be spent. To be able to make fancy edits to the game, we first need to know what we need to edit, if we can edit it, how we edit it and what happens if we edit it. We have many tools that can help us such as IDA Pro, Binary Ninja, x64dbg, REClass.net, but maybe the most fundamental and infamous is Cheat engine. You might have played around with this tool as a kid when cheating in flash games or similar and might therefore not take this tool serious. But cheat engine is an extremely good tool which has many features. The most important tool in Cheat Engine is the memory scanner. Since memory scanning is arguably the most important skill in game-hacking and because Cheat Engine is doing such a good job at it we use this tool a lot and is therefore very

important to learn. This tool scans the addresses within scope of a selected process for values. Forcing these values to change by doing actions in the game we can then rescan results from the earlier scan to get fewer results.



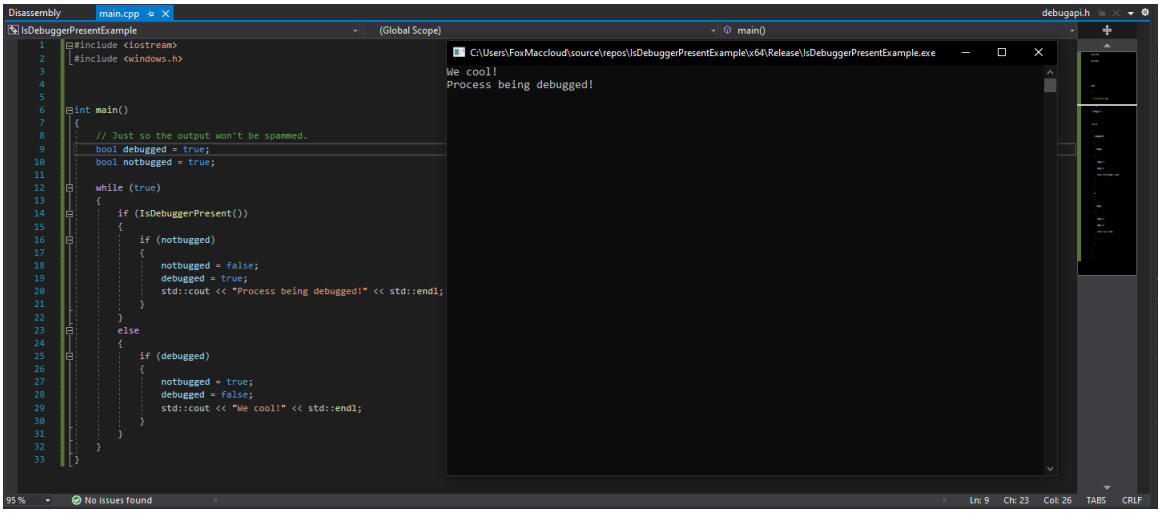

In this small challenge we want to destroy that target. The target regenerate to max health every time we have to reload. Scanning the memory for the value 5 will do us no good. Some games have "weird" logic that is not clear right away. Here when we have full ammunition, the value is 0 and increases as we shoot our bullets.

## 4.1 Anti-debugging

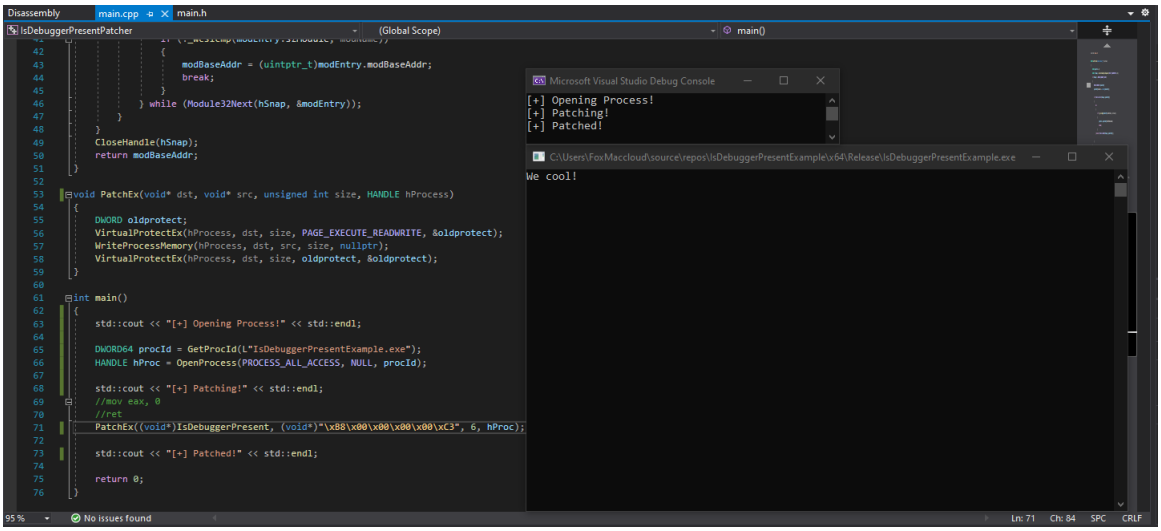
When reverse engineering games anti-debugging and/or anti-cheat is very common, especially online games make use of this. This is to hinder the hacker to reverse engineer and modify the game. Anti-debugging can come in many forms. These can come in ways of hooking internal windows functions such as NtSetInformationThread, NtQueryInfomrationProcess, obfuscation, encryption and decryption, blocking handles to the process and many more. The most common check which is used by pretty much any anti-debugger is IsDebuggerPresent(). IsDebuggerPresent checks the PEB structure for the being debugged flag. The PEB structure comes from the windows kernel which holds the data about the current process. Some of these values can be structs that holds even more values. Every process on windows has its own PEB so it and contains values such as the process name, process id, DLLs it requires and more. Looking at IsDebuggerPresent, we can see that the return value of when the process is not being debugged is zero. A simple way to defeat this would then to just patch the function to always return zero.

Creating a small example using IsDebuggerPresent we can see how it works. In this example instead of printing out to the console that the process is being debugged we can just call exit and stop the

process from continuing execution.




I'm attaching Microsoft Visual Studio 2019 as a debugger to the process and we see that the process immediately tells us that the process is being debugged. Implementing the fix I mentioned earlier I'm replacing the bytes in IsDebuggerPresent with mov eax, 0 which overwrites the entire RAX register with nulls then just ret to return. We will then have successfully tricked the function call to always return the wrong value. Attaching Visual Studio again and we see that the process doesn't report being debugged.




Most of these anti-debug techniques can be avoided by using special debuggers. E.g., Cheat Engine has the option to swap between different debuggers such as the windows debugger, VEH debugger and kernel debugger. The windows debugger is the standard debugger that will trigger on most of these checks.

VEH stands for Vectored Exception Handling. It works a little different from the windows debugger in that it injects some code into the process, then does debug events through exceptions. VEH then intercepts these exceptions and debug related information to Cheat Engine through shared memory between Cheat Engine and the target process. In the case of a breakpoint, it will then wait until Cheat Engine tells the process if the exception was handled or not and is allowed to continue execution.

Kernel debugger is a bit different as it works by having a driver that runs at kernel level. Using cheat engine we are then communicating and controlling this debugger on the kernel level. Since no user-mode applications are allowed to directly communicate with the kernel, makes them very hard to deal with if you want to protect your application.

In some cases, anti-debugging makes it almost impossible to debug and reverse engineer the processes. If you are not able to manually disable these countermeasures one can resolve to static analysis. Static analysis is when you look at the game code without actually running the process. These are often used by crackers that wants to crack software on disk instead of in ram. In many cases this beats most anti-debugging as they are often implemented to stop you at runtime. There are still "anti-debugging" that could still take place when doing static analysis in the form of obfuscation. Developers could pack their software in "packers". These packers obfuscates the code when the game is stored on disk and will only deobfuscate on execution when the process is loaded into ram. One way to go around this is to load the process into memory, then dump the process as a "memory dump", then analyse the code in the dump statically.

Developers can get very creative with how they use anti-debugging. A good example for stopping a basic debugger is to spawn a child process that debugs the main client and another debugger from the client to the child process, thus creating a sort of circular protection for both of the processes [4]. Many games nowadays also checks for instances of certain programs such as Cheat Engine and refuses to execute if it detects these programs. In many cases, compiling Cheat Engine from source and replacing all string instances of "Cheat Engine" to something else is often a solution for these checks.

## 4.2 Anti-cheat

An anti-debugger is there to stop you from debug and reverse engineer the game where as an anti-cheat is to stop you from modifying the game.

The anticheat can be built into the game or it can be some additional software that runs in the background while the game in running. These programs can be developed by the game developers themselves such as Vanguard for Valorant or third parties such as Easy Anti Cheat (EAC) or Battleye. Typically one cannot run these games without these anticheat programs running. Anticheats has features like;

- Anti-debug

  Anti-debug is a part in anti-cheats, but anti-cheats goes much deeper then classic anti-debugging.

- File & Memory integrity checks

  File and memory integrity checks are there to make sure none of the data has been changed. This is done by making checksums or hashes of important data. By doing this, we make sure that the files are legit and not a custom client. This can be bypassed with in two different ways. One is to reverse engineer the checks and patch them and the other is to only modify the game in memory after the game has launched. However this second method only applies if the game doesn't have memory integrity checks.

- String detection

  String detection works in that it checks processes for instances of specific strings. These can be instances of e.g., "cheat engine", "x64dbg", "IDA", etc. The anti-cheat can then refuse to launch the game or close these programs when these strings are detected. This can be bypassed by replacing instances of these strings with something else.

- Obfuscation

  Often anticheats uses obfuscation. This is both for anticheat and for antidebug. Obfusation comes in infinite amount of forms, but a good example of obfuscation is to keep critical sections of memory such as playerstats, player-locations, etc. encrypted and only decrypt them as they are needed. This will make it very hard to find and use once found. This can also be extended to files on disk. You can keep the game files encrypted on disk and decrypt the actual game binary in memory. A way to bypass this is to dump the memory of the game to a file and reverse engineer the dump. A way to take this a step even further is to mix it with virtualization. You keep the binary encrypted on disk and when you run the game it doesn't give you the game code, but custom bytecode which is executed within a virtual machine which then translates it into machine code. Much like how Java works expect with custom bytecode. Some obfuscators that targets JIT based platforms e.g., jvm or .net might try to hook inot the backing virtual machine to alter the JIT compilation process. A common obfuscation technique is control flow based obfuscation. This can look like doing proxy calls e.g., call by hashes to hide external calls, string encryption, constant mutations or false branches. I.e., if/else branches where one of the conditions are never true [5]. This is also known as opaque predicates. Obfuscation comes in many form and often creative forms, so you also have to get creative to defeat them all.

- Signature based detection

  Most anticheats has some sort of signature based detection. This is to make sure that known cheats no longer works after getting caught. Signature based detection works in that it when it finds something that is a cheat it will make a signature of that and add that signature to a database. Now when it later finds that exact same signature it knows the player is cheating. This can sometimes be bypassed by writing polymorphic code so the signature always changes, change the code after getting caught or to never get caught in the first place.

- Hook detection

  There are multiple ways game can check if the game is hooked. Anti-cheats often check the first bytes of the functions. Sometimes they also check specific instructions that could be "interesting". They can also see if hooks are in place by checking what's on disk versus what's in memory. If they don't match, the code has been modified in some way during runtime. A way to bypass this is to use hooks that doesn't modify the code at all such as the earlier VEH types of hooks. However there are ways that these can be detected aswell such as looking for exception handlers or checking if debug registers are used.

- Virtualization

  Virtualization creates simulated computing environments as opposed to physical environments. Within these virtualized environments the rules of the games is a little different. How this virtualized environment is used varies a lot. As mentioned earlier you could run custom code within these environments or sections. It's also common to see the use of "packers" which uses this technology. A packer simply keeps the real binary encrypted on disk, but decrypts it in memory once you run the game. Different fragments of the code could also run in different virtual machines. If someone now wants to crack this software, they are going to have to reverse multiple architectures of virtual machines. Having parts of the game run within virutal environments or even worse, the entire game within a virtual machine comes with a performance cost. Games protected with Denuvo is infamous for this and has seen impacts from negligible to huge performance impacts.

  Another thing anticheats does is to detect virtualization. Anticheats want and the game wants to run on real hardware. This is to ensure system integrity and that the anticheat can see everything. When you run your game in a VM then debug any processes through a named pipe. A debugger will connect through this pipe. Not only can you debug this virtual machine, but you could also use the virtual machines hypervisor to do tasks for you such as read and write to memory. The hypervisor has direct memory access (DMA) and could do pretty much anything as the hypervisor manages the resources for these virtual machines.

- Kernel

  Many anticheats these days comes with kernel level drivers, especially where there are large financial incentive to stop players from cheating. Kernel anticheats has all the features of all of the above in userspace, but comes with even more functionalities for detecting cheats. In many cases if you want to defeat a kernel level anticheat, you also need to jump into the kernel. When you develop your cheats, usually you use your kernel driver to disable or bypass this anticheat driver and keep your hacks in userspace, however the entire cheat could also be in the kernel. Kernel anticheat will therefore check the integrity of themselves, meaning you cannot outright patch them. It will also look for system threads, device objects and so much more. More then I'm able to cover in this small chapter. Example the earlier mentioned IOCTL method of communicating between user and kernel will in most cases be detected by kernel anticheat. This is because for manual mapped drivers the device object will not have a

backed image where as a singed driver would. Some of these kernel level anticheats even loads on boot and will run all the time. This is to ensure integrity from boot-time.

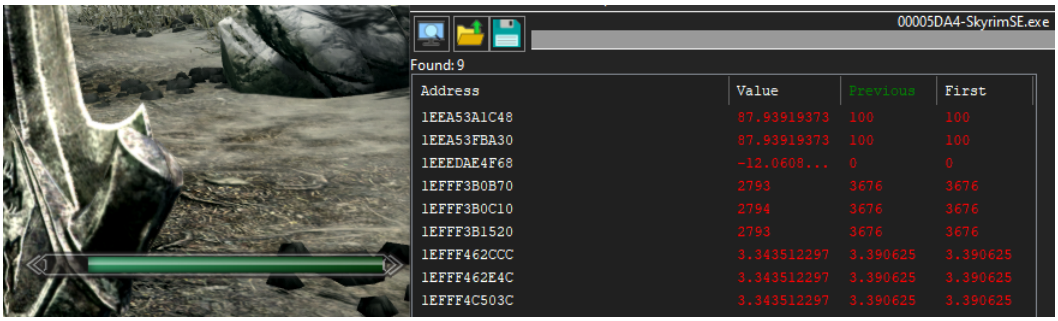
# 5 The Elder Scrolls V: Skyrim

Skyrim is a game that came out in 2011 and was updated from x86 to x64 in 2016. It's an open world action role playing game that was developed by Bethesda [6]. I chose this game to mess with when I was learning because it's a single player game with no anti-debugging or anti-cheat and it has a lot of mod support and documentation. This means that I won't mess with other players or chance that my access to the game will be revoked.

## 5.1 Infinite Health, Stamina and Magic

To get infinite health, stamina and magic we first need to find the game instructions which updates these values. My plan to find these was to change them, use cheat engines scanning tool to see what values changed and look at these addresses. I can then see what instructions accessed these addresses and then find the instructions which updates the values in those given addresses.

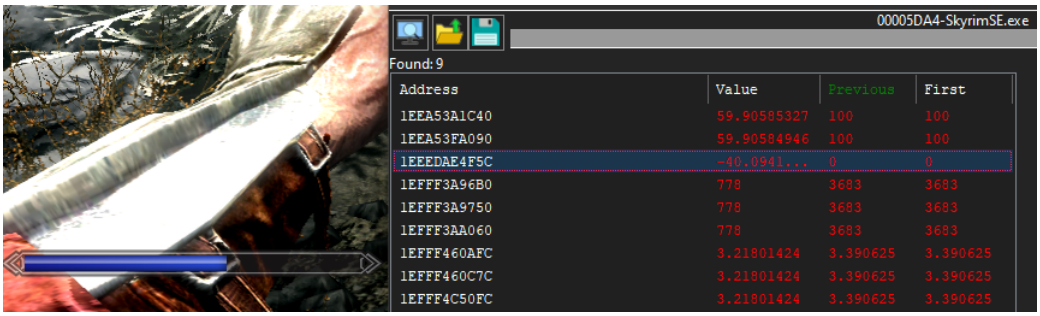
The easiest and fastest value to change is the games stamina. Stamina is used for essentially any action in the game except for magic. It's used when running and it's used when hitting things.



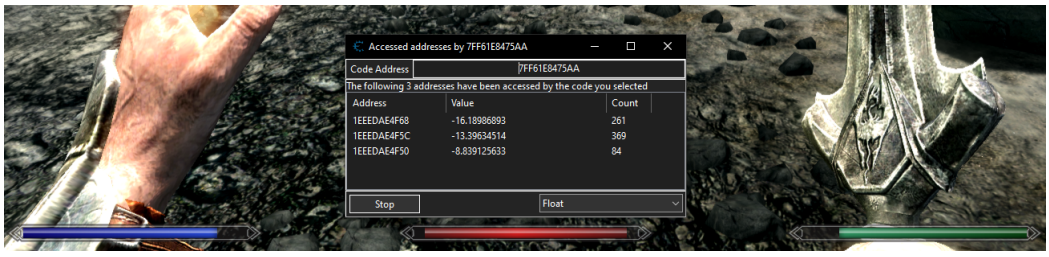

After running around for a while I came to these 9 addresses. These 9 addresses are all related to our stamina in some sense. It could be our actual stamina or it could be something related like the stamina bar which shows us how much stamina we have. Changing these values won't change our actual stamina, but only the visuals. My essential thought was that the two first addresses would be the ones that were my true stamina, but apparently it was not. One was for visuals (bar shown on screen) and the other is unknown. The true stamina value was the one which is -12 in picture. When you are at 100% stamina, the value shown is 0 and when you are at 0% you are at -100. This is a weird design choice. When your stat pool exceeds 100, it's the same, however the game will subtract less stamina from your pool. Looking at what instructions that wrote to this address, I came to this instruction;

```
77 03          ja      7FF61E847594
0F57 F6        xorps   xmm6,xmm6
48 8D 8B B0000000  lea   rcx,[rbx+00000B0]
8B D6          mov     edx,esi
48 8B 01       mov     rax,[rcx]
FF 50 08       call    qword ptr [rax+08]        Sets XMM6 "amount" value.
F3 0F10 0C AF  movss   xmm1,[rdi+rbp*4]
8B D6          mov     edx,esi
F3 0F11 34 AF  movss   [rdi+rbp*4],xmm6          Updates Stamina
48 8B CB       mov     rcx,rbx
F3 0F5C F1     subss   xmm6,xmm1
```

movss [rdi+rbp*0x4], xmm6. We know that this is the right instruction because when we replace it with nop instructions, our stamina no longer goes down. Awesome! Now lets do the same for magic and health.



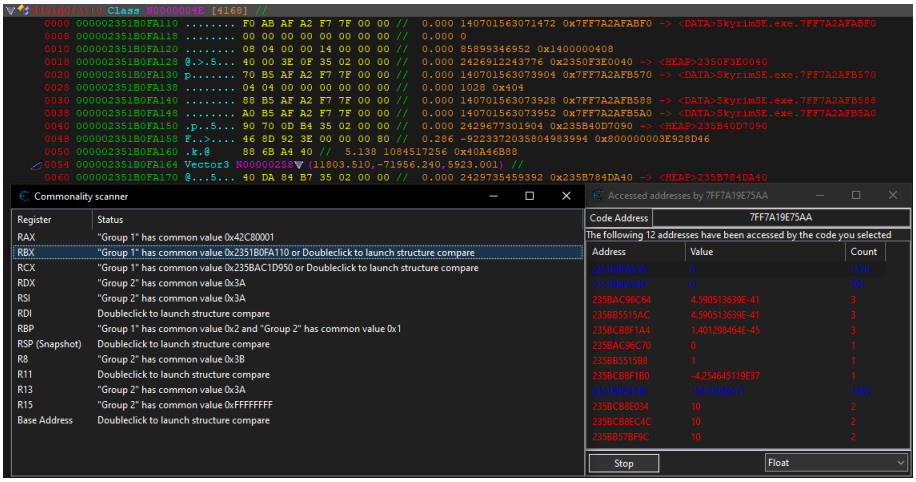

For magic it was exactly the same, but looking at which instruction which wrote to this address, it is the exact same instruction that was used for stamina. This means that the instruction is shared. I'm now thinking that health would be the same story, which is was.



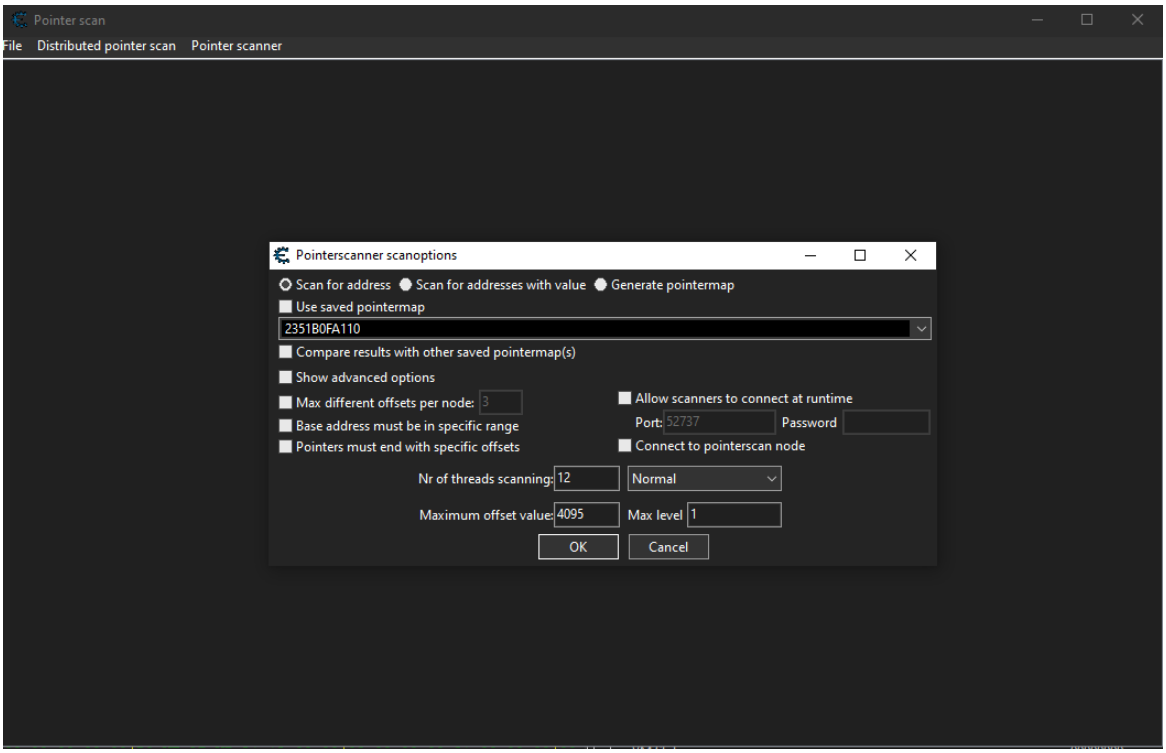

We see that all these addresses are accessed by the same instruction. However this instruction accesses a lot more then just these three. It also updates the stats for all the entities in the game. This is a problem as we cannot just replace these instructions with nops as this would give all the entities in the game invulnerability.

To get infinite health magic and stamina, we have two options. One option is to "freeze" our stats. This means we can find the addresses of our stats and always write (in this case) 0 to the addresses. The only problem with this approach is that if we take more damage then we have health in one go, we would die. The second approach and the approach I went with is to check which entity that is calling the function to update their stats and if that entity is the player, overwrite the change with 0 instead of a negative value. This approach doesn't suffer from the problem that freezing has as this makes it impossible for you to take damage in the first place.

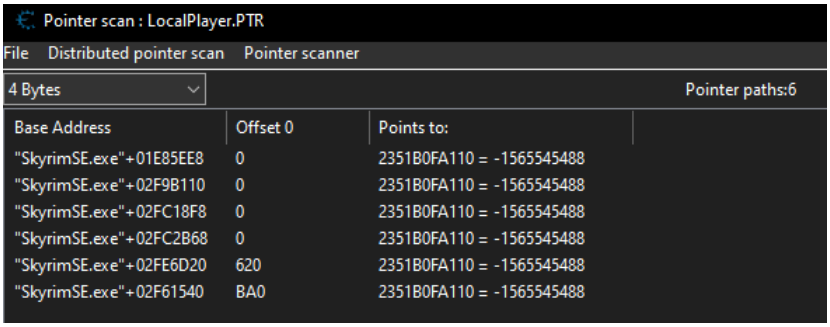
To accomplish this I'm going to install a hook. This hook will overwrite the original instruction with a jump to a different function. This other function will then check if I'm the player or not. To figure out how to compare the different entities, we can use Cheat Engines commonality tool. In this tool you can put addresses into different groups and see what makes those groups different. I put my health, stamina and magic into group one, then everything else into group 2.




Looking at the differences we have, we can see that RBX has the same values in this function. These values also happen to be the address of the base of the player class. If we can then find the player, we can compare if rbx == &player. A way to always find the player is to use an offset from the base of the class to the player. To find this offset I used Cheat Engines pointer scan tool.




The pointer scanner searches the games native pointer paths to find the specific address you are looking for. When it finds it, it saves it. Using this I was able to find multiple offsets to the same address.




Using one of these offsets I could then always find my player class when starting the game. Now that I know where I need to place my hook and how I can differentiate the entities, it's time to write a hook.

```
1 inline DWORD64 returnAddressGod = 0x0;
2
3 inline _declspec(naked) void god_hook_h()
4 {
5     __asm {
```

```
6           pop rax
7       }
8
9       static Entity* playerCheck = 0x0;
10      static float statChange = 0;
11
12      __asm {
13          mov[playerCheck], rbx;
14          movss[statChange], xmm6;
15      }
16
17      if (localPlayer == playerCheck)
18      {
19          statChange = 0;
20      }
21
22      __asm {
23          movss xmm6, [statChange];
24          movss[rdi + rbp * 0x4], xmm6;
25          mov rcx, rbx;
26          subss xmm6, xmm1;
27          movaps xmm9, xmm0;
28          jmp[returnAddressGod];
29      }
30  }
```

_declspec(naked) means that the generated code is without prolog and epilog code. The first pop rax is because my hooking code pushes rax, overwrites rax with a jump, then calls jmp rax. This pop will then just restore our original rax. Next I'm saving values we want to interact with such as rbx which is the entity in question and xmm6 which is the amount the stat in question is chaining by. If the entity is the player, we overwrite xmm6's value with 0. Then the code continues as normal and we jump back to where we came from.

## 5.2 ESP

ESP is short for Extra Sensory Perception. It gives the player an advantage by feeding the player more information that is given by the game. ESPs can provide informations such as names, health, locations and more. In e.g., a hide and seek game where the players hide. It is easy to see why an ESP could be powerful when you can always see wherever everyone else is. To see the player through walls are also often referred to as a "wallhack".

Not all ESPs are created equally. For some games you could use the graphics api to draw your ESP. You could also create your own transparent window and lay that on top of the game and draw on

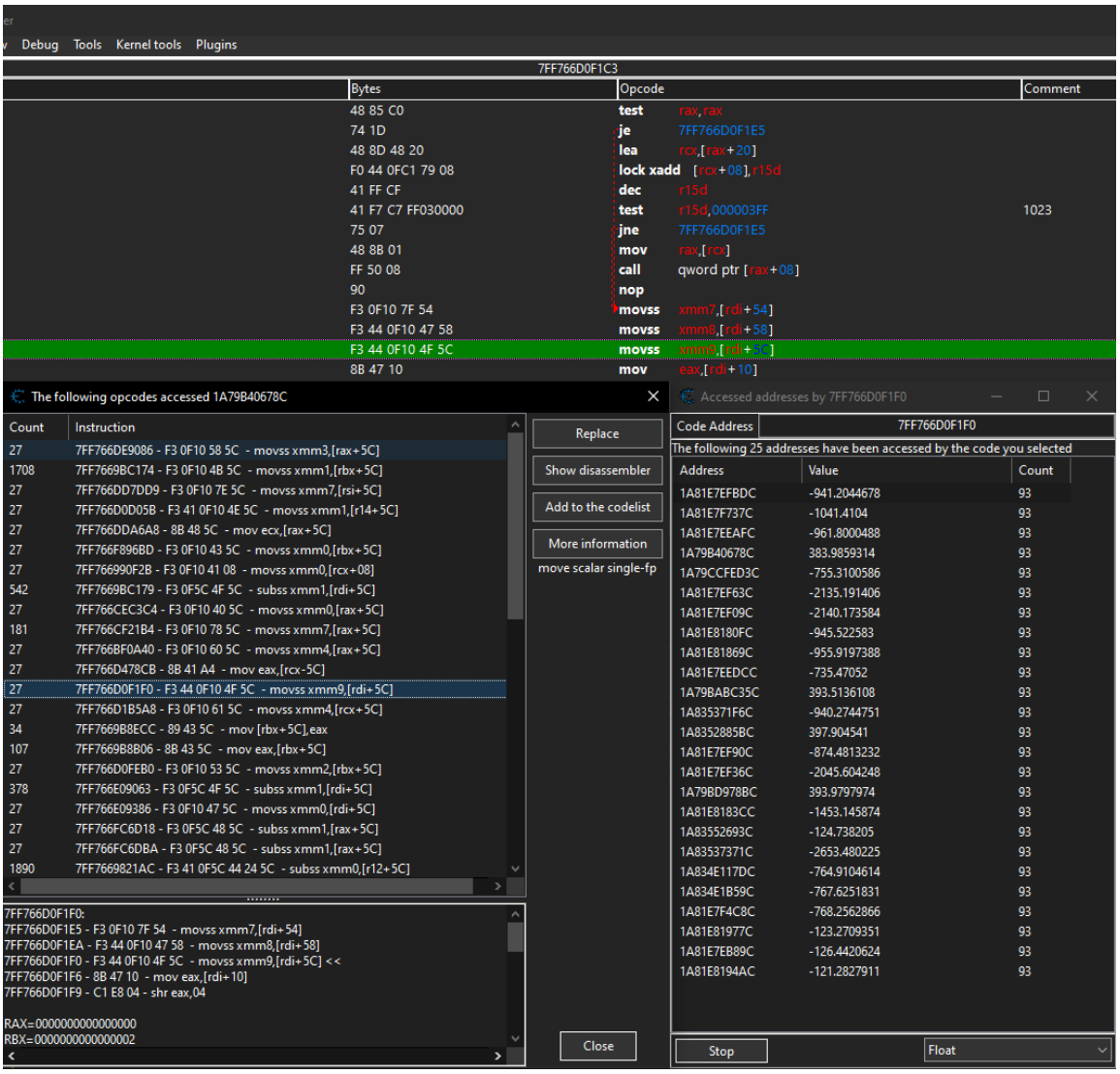
that. One could also call game functions such as forcing the nameplates of characters to be seen through walls.

To make an ESP or wallhack, you are gonna need to reverse engineer the game and find some things. You need the position of the entities, a "viewmatrix" and a "world to screen" function. We also need something to draw on. I was initially just going to hook the graphics api, DirectX11. However because of how the engine worked, this is inconvenient. So instead I'm going to use a transparent overlay which lays ontop of the game. To know where to draw, one need to know where the entities are in 3d space. We also need the viewmatrix. A viewmatrix is a world transformation matrix that determines the position and orientation of an object in 3D space. It transforms a 3D world world-space to a 2D view-space. Much like a camera when it takes pictures of our 3D world [7]. A world2screen function is a function that uses this viewmatrix to figure out where on our screen it should draw. Essentially translating 3D space to 2D space. When we have all these, we grab the position of the entity and give it to our draw function. The draw function will then use the world2screen function which again uses the viewmatrix to draw the entities names.

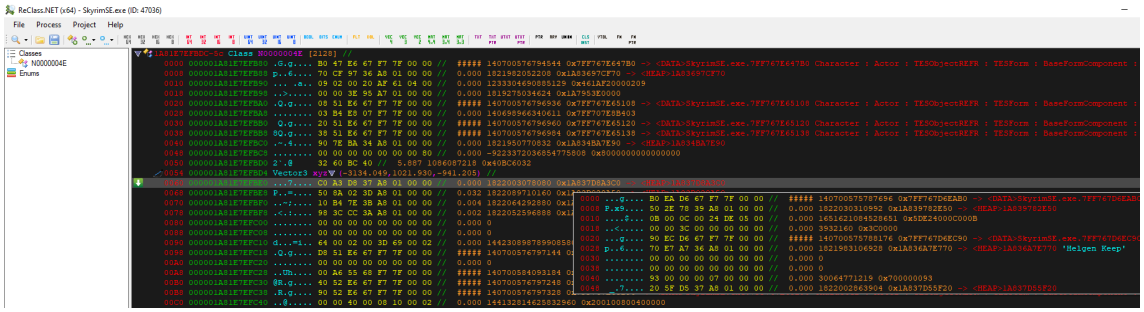
Finding the entities and viewmatrix can in some cases be difficult. As mentioned earlier entities in most cases are just classes or structs. The game needs to keep track of these entities. Usually these are kept in an array of pointers where each of these pointers point to an entity. If we find the start of this array, when can then loop through it and find all the entities. This is sometimes difficult to find. Some of you might also remember shared instructions. We can use these shared instructions to find the entities. Since all entities must have a position and this instruction is often shared. We just have to find this function, write a hook that keeps track of all the entities passing by and saving them to an array.

To find the instructions that updates our position, we first need to find our position. We can find our position in Skyrim by looking for a float value. We are guessing for a float value for our position, but it could also be a double. Both of these would make sense, but it should be a decimal value. In the case the value was a double, but you are searching for a float, you'd in many cases find half the bits of the double as a float. This can still guide you to finding the right value. Now that we have searched for float we can see that we have a large amount of results. Way to many to go through. If we move up an hill, the float value of our Z position should change. Going up a hill would increase or Z position value and going down would decrease it. Doing this a few times we have gotten the amount of results down by a lot and we will eventuality get a conceivable amount of addresses to work through. We have all these results because there are many things which is at our position outside of our position. E.g., positions of our bones, position of our gear, etc. which moves with us.

To find the right instruction, the one that updates our position, I went through all the addresses in our possible results. I looked at what instructions accessed my address and see if the result made any sense to me. In the end I came to an instruction that looked like; movss xmm9, [rdi+5c], which not only made sense, but also made sense in the disassembled code as I'd assume that X, Y, Z positions would be right next to each other. If we see what accesses these instructions we get a list of presumably entities.




Now we have our entitylist. We can take a sample of the entities and start reverse engineering the entity class structure. There is a tool called REClass.net which is very helpful with this. Since CheatEngine looks at the rdi + 5c, we can assume that rdi is the entity base and we therefore need to subtract 5c from the address we have in CheatEngine. We can assume that the offset 0x54, 0x58, 0x5c are the XYZ, position, we can then assign these offsets as a vec3 struct.



We can also find pointers that points to strings and other fun stuff. We can then follow these pointers, name them and let REClass automatically generate structs that we can then use in our code.

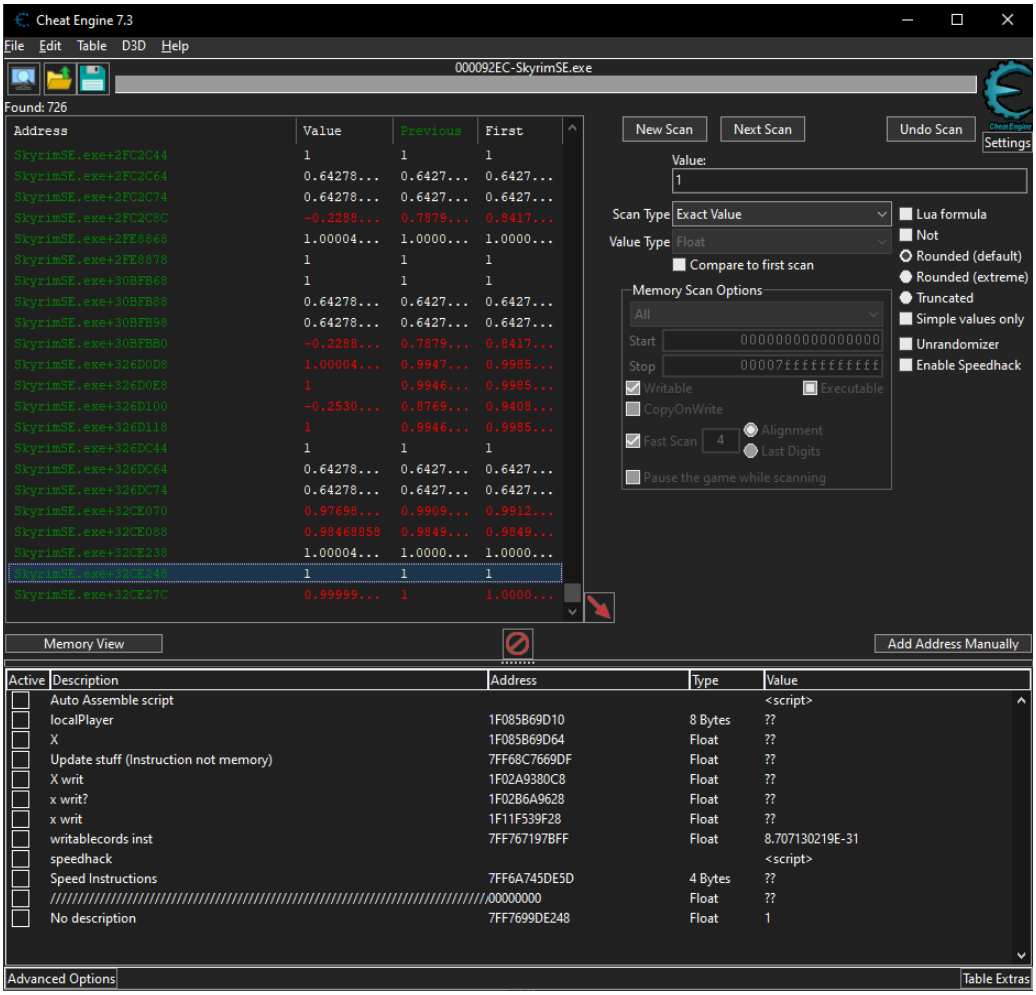
```
class LocationName
{
public:
    char pad_0000[40];
    char* locationName;
};

class Name
{
public:
    char pad_0000[40];
    char* name;
};

class Entity
{
public:
    char pad_0000[84]; //0x0000
    vec3 xyz; //0x0054
    LocationName* locationPtr; //0x0060
    char pad_0068[400]; //0x0068
    Name* namePtr; //0x01F0
    char pad_01F8[1624]; //0x01F8
}; //Size: 0x0850
```

A viewmatrix is a 4x4 matrix filled with float values, these float values represents a “right” vector, an “up” vector, a “forward” vector and a “translation” vector. Essentially this matrix shows how the camera sees. The looking up/down right/left goes from a max 1 to a minimum -1 we can use these values to find the viewmatrix in memory. If we look straight up one of these values has to be 1 and if we look straight down, the same address has to be -1. Going back and forth like this, we

can get a comprehensible amount of addresses we can look through. If we look at these addresses in memory view, then move around a bit. We can find "candidates" for our viewmatrix.




I ended up finding this one which was different from the rest.

```
Protect:Read/Write  AllocationBase=7FF766710000 Base=7FF7699DE000
address       B8          BC          89ABCDEF
7FF7699DE1B8 0.00        0.00        ........
7FF7699DE1C0 0.00        0.00        ........
7FF7699DE1C8 0.00        0.00        ........
7FF7699DE1D0 0.00        0.00        ........
7FF7699DE1D8 0.00        0.00        ........
7FF7699DE1E0 0.00        0.00        ........
7FF7699DE1E8 0.00        0.00        ........
7FF7699DE1F0 0.00        0.00        ........
7FF7699DE1F8 0.00        0.00        ........
7FF7699DE200 0.00        0.00        ........
7FF7699DE208 0.00        0.00        ........
7FF7699DE210 -1.15       -0.30           `b
7FF7699DE218 0.00        0.00        ........
7FF7699DE220 -0.54       2.05        .;.   ..@
7FF7699DE228 0.00        0.00        X    ....
7FF7699DE230 0.00        0.00        d.. d. 3
7FF7699DE238 1.00        -15.00      d. ? .p
7FF7699DE240 0.00        0.00        ... .. 3
7FF7699DE248 1.00        0.00        .. ?....
7FF7699DE250 -0.81       -0.12         O F
7FF7699DE258 0.00        0.00        ... 5 .
```

To truly know if this is the correct viewmatrix, I have to test it out and see if my drawings gets translated from 3d to 2d space. If it's not the correct one, it will be very obvious.

Alright, now we have everything we need to make our ESP. I'll probably use entities later, so I want to create an array to store my entities in. I also decided that I wanted to hook the update position function with a midfunction hook. I made a hook which looks like;

```
1  inline std::vector<Entity*> g_entities;
2  inline Entity* ent;
3
4  inline __declspec(naked) void entity_hook_h()
5  {
6      ...
7      bool alreadyThere = false;
8      if (ent != nullptr)
9      {
10         if (*std::find(g_entities.begin(), g_entities.end(), ent)
               == ent)
11         {
12             alreadyThere = true;
13         }
14         if (!alreadyThere)
15         {
16             for (int i = 0; i < g_entities.size(); i++)
```

```
17              {
18                  if (g_entities.at(i) == nullptr)
19                  {
20                      g_entities.at(i) = ent; // insert
21                      break;
22                  }
23              }
24          }
25      }
26      ...
27  }
```

This hook will then look for empty space in our array and put the new entity into an empty spot in the array. Now every time we want to access our entities, we can just loop through this array.

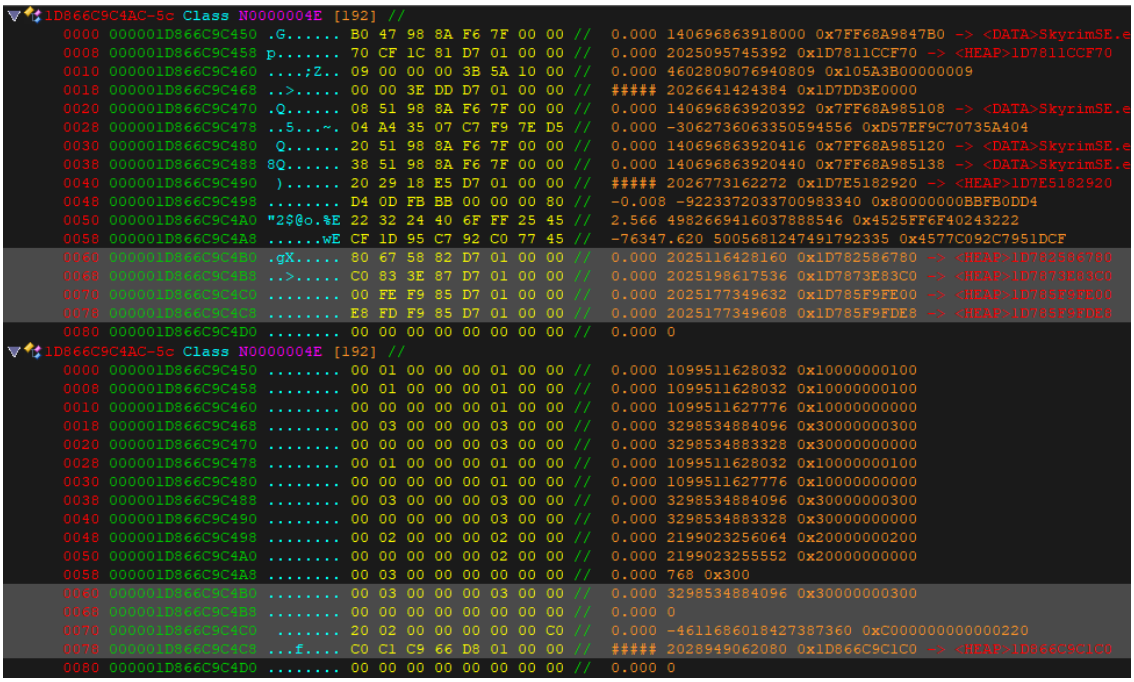
I implemented my ESP function like so;

```
1  void Cheats::esp_text(int windowWidth, int windowHeight)
2  {
3      memcpy(&m_viewmatrix, (PBYTE*)(addrSkyrim+offsets::
           view_matrix_offset), sizeof(m_viewmatrix));
4      for (int i = 0; i < g_entities.size(); i++)
5      {
6          vec2 vScreen;
7          if (HelperFunctions::worldToScreen(g_entities.at(i)->xyz,
               vScreen, m_viewmatrix, windowWidth, windowHeight))
8          {
9              draw_string(g_entities.at(i)->namePtr->name, vScreen.x,
                   vScreen.y);
10         }
11     }
12 }
```

And sucess, I now have a working ESP!


What's nice about this is that because of how the map is implemented in the game, the text even shows up in the map. This means we also technically have a radar.


Running around in the game I noticed that sometimes my game crashes randomly. I noticed that this problem happened because when you get out far enough or enter an instanced area such as a dungeon, the entities de-spawn. When they despawn, we are then trying to access entities that no longer existed. I tried to look at one of the entities in REClass while I moved in a direction. Once that entity got out of range I saw that certain pointers disappeared.




Since I know these pointers has to be valid for the entity to exists. I can just write a function which checks if it's valid.

```
bool validate_entities(int i)
{
    if (g_entities.at(i) == nullptr
        || g_entities.at(i)->locationPtr == nullptr
        || g_entities.at(i)->validationPtr1 == nullptr
        || g_entities.at(i)->validationPtr2 == nullptr
        || g_entities.at(i)->validationPtr3 == nullptr
        )
        return false;
    else
        return true;
}
```

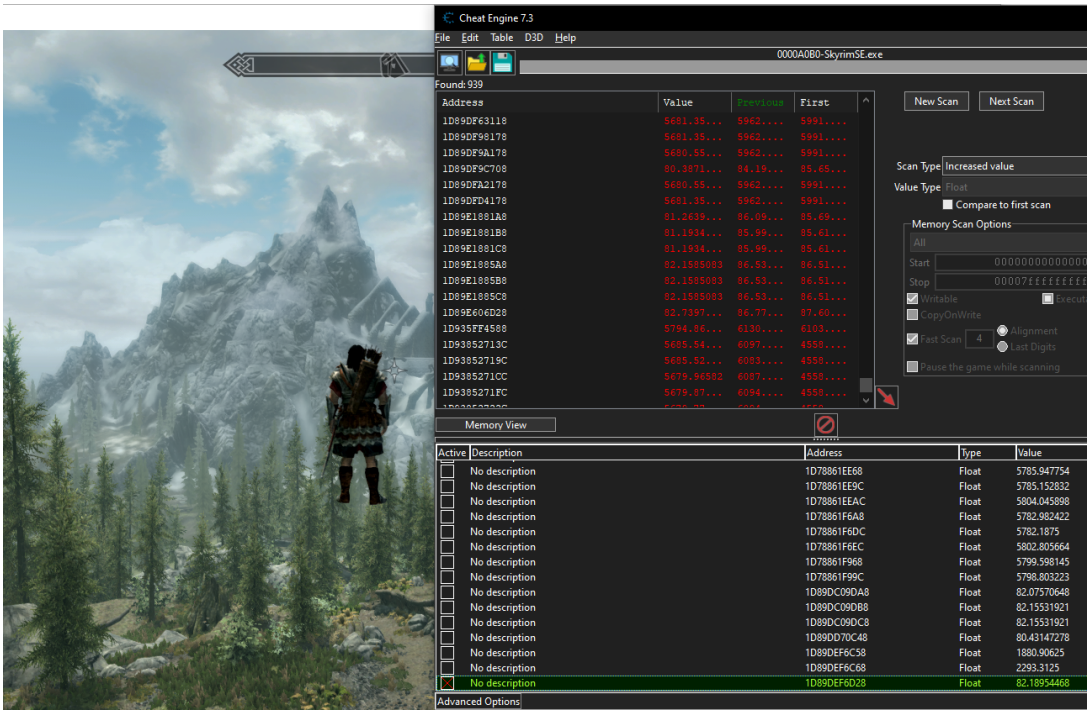
Now I have a functional ESP that doesn't crash my game.

## 5.3 Teleport

Trying to overwrite these previous coordinates I used for my ESP doesn't affect the real position of the entities or the player. This means that the "real" player coordinates are somewhere else. Back to scanning memory like I did for my ESP hack I tried freezing values until I saw changes in the game. Sometimes when freezing values I got funny behaviours like arms, head and other body

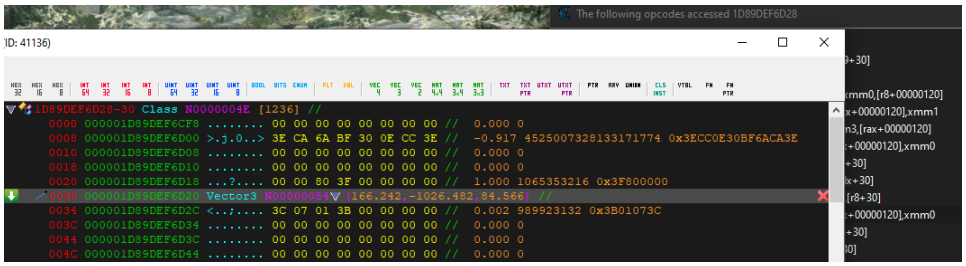
parts getting stuck. I believe these are the real coordinates for some of the bones in the players body.

I did this for all results until I was able to walk on the air.

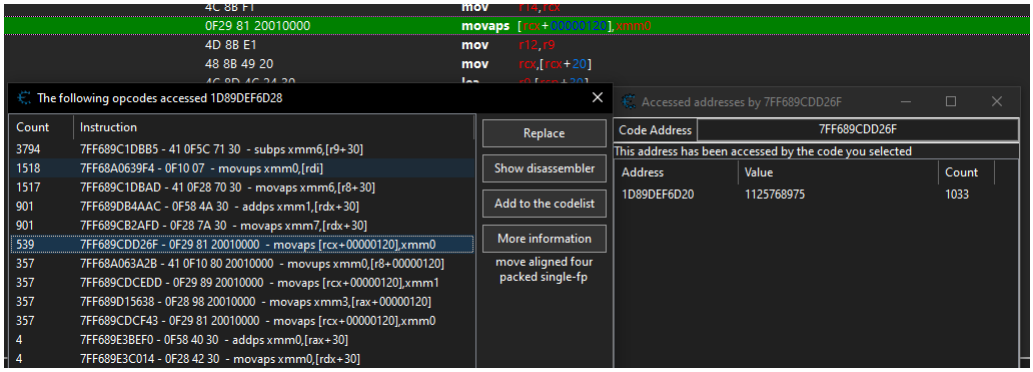



I'm assuming that the engine has implemented these real coordinates with a simple data structure

like a vec3. This means that each direction should be fairly close to each other. Maybe even right beside each other. Looking in REClass I can see that I have three floats right beside each other in memory and modifying these values moves me accordantly in the game.




Now I need to find a way to always find this value. I checked what wrote to this address, then looked to the instructions that accessed. I decided on inserting a hook where this instruction were.




To find it every time, I made a signature of this instruction.

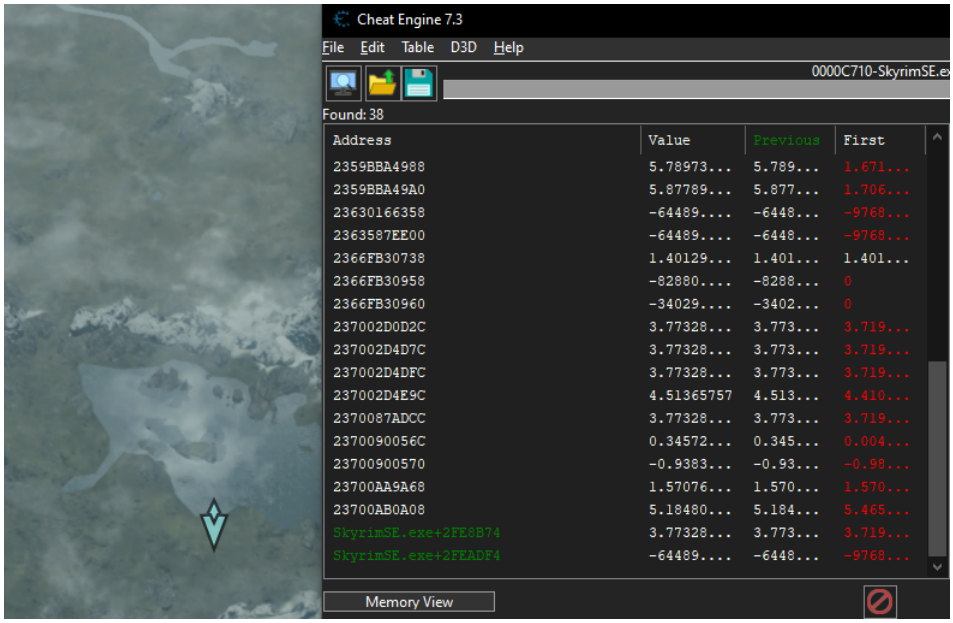
Since I now have my writeable coordinates, I now need to incorporate this into my cheat. Since I know that just editing this value will teleport me I can just read and write to address.

```
1 void Cheats::teleport(vec3 coordinates)
2 {
3     if (playerCoords != nullptr)
4     {
5         playerCoords->xyz = coordinates;
6     }
7 }
```

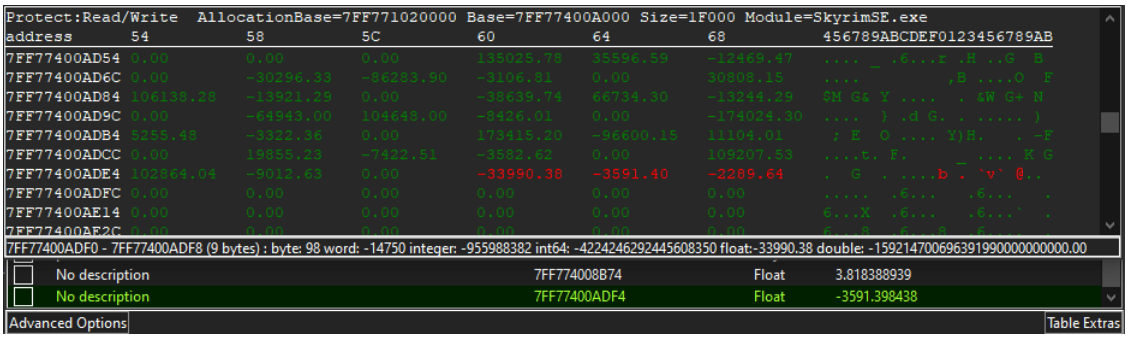
Now having an input where you put in cordinates of where you want to go is not very convenient. You need to know where you are and what coordinates you want to go to.

In Skyrim, there is a marker you can place down on the map. This marker will show up in 3d space. I could find the location of this marker, then teleport to this marker. To find this marker I'm again using cheat engine to search for it. My assumption is that you could draw the map within a graph where the origin is either in the bottom left or the middle of the map. This means if I search for the

marker when I have it marked at the bottom, then move it to the top. The markers position value should increase.




After searching like this I ended up with about 40 results, however only two static ones. Checking these two static addresses I found that in memory view that one of them are "alone" in that it doesn't have any close values that changes with it unlike the 2nd which changes.




However none of these values matches with the player position coordinates. Maybe they fit the "world coordinates" I used for my ESP. And they do. I could now take the waymarkers position and divide it by my own coordinates and I'll have a constant, about 70, I can multiply with to switch between world coordinates and my coordinates.

## 5.4 Fly hack

If we instantly teleport a long distance we might run into a problem where the game knows that we are doing an impossible move which can lead to different consequences. Especially for online games.

However instead of teleporting the entire distance in one go, we could split them into small chucks, so basically we can fly there.

To make a fly hack I could simply do it like;

```
void Cheats::fly()
{
    vec3 newPos = get_player_coords();
    if (GetAsyncKeyState(VK_SPACE))
    {
        newPos.z += 1.0f;
    }
    if (GetAsyncKeyState(VK_CONTROL))
    {
        newPos.z -= 1.0f;
    }
    if (GetAsyncKeyState(VK_W))
    {
        newPos.x += 1.0f;
    }
    if (GetAsyncKeyState(VK_S))
    {
        newPos.x -= 1.0f;
    }
    if (GetAsyncKeyState(VK_D))
    {
        newPos.y += 1.0f;
    }
    if (GetAsyncKeyState(VK_A))
    {
        newPos.y -= 1.0f;
    }
    teleport(newPos);
}
```

Here we are moving small distances in each direction. This works, however is not very convenient. There are two main problems with this method. One is that gravity will always try to drag us down back to the ground, so one cannot stand in place in the sky. The other is that the direction where you look does not equal to the direction you will fly in. This is because we are just updating our position in the world along the axises.

To solve the first problem I'm going to patch out the instruction which updates our position with nops. When we do this we can no longer walk in any direction at all, which includes down.

The second problem requires a little linear algebra knowledge. We can use the viewmatrix to calculate our X, Y and Z values for us. Since the viewmatrix contains both the "forward" and the "right", we can use these to do the calculations for us. For flying forwards and backwards where we are viewing, we can use the "forward" vec3 in the viewmatrix and for the left and right, we can use the "right" vec3. Our code will then look something like;

```
1  void Cheats::fly(bool active)
2  {
3      memcpy(&m_viewmatrix, (PBYTE*)(Offsets::skyrimBase + Offsets::↘
           viewmatrix), sizeof(m_viewmatrix));
4      vec3 newPos = get_player_coords();
5
6      float speed = 2.0f;
7
8      if (GetAsyncKeyState(VK_W))
9      {
10         newPos.x += m_viewmatrix[8] * speed;
11         newPos.y += m_viewmatrix[9] * speed;
12         newPos.z += m_viewmatrix[10] * speed;
13     }
14     if (GetAsyncKeyState(VK_S))
15     {
16         newPos.x -= m_viewmatrix[8] * speed;
17         newPos.y -= m_viewmatrix[9] * speed;
18         newPos.z -= m_viewmatrix[10] * speed;
19     }
20     if (GetAsyncKeyState(VK_D))
21     {
22         newPos.x += m_viewmatrix[0] * speed;
23         newPos.y += m_viewmatrix[1] * speed;
24         newPos.z += m_viewmatrix[2] * speed;
25     }
26     if (GetAsyncKeyState(VK_A))
27     {
28         newPos.x -= m_viewmatrix[0] * speed;
29         newPos.y -= m_viewmatrix[1] * speed;
30         newPos.z -= m_viewmatrix[2] * speed;
31     }
32
33     if (GetAsyncKeyState(VK_SPACE))
34     {
35         newPos.z += speed;
```

```
36      }
37      if (GetAsyncKeyState(VK_CONTROL))
38      {
39          newPos.z -= speed;
40      }
41      teleport(newPos);
42  }
```

We can also add a speed variable for how big increments we should go in our directions. If the game had some checks to see if our move was legal or not, we could tweak our speed multiplier to see how fast we could fly.

## 5.5 Speed hack

If flying doesn't work, we can try to walk insanely fast. There are mainly two different ways of doing this. One is to modify the "speed" of the game, i.e., how fast one "game tick" is. If we make this faster, everything in the game will run faster or slower. By everything it means literally anything. A minute in the game becomes seconds. This is often an undesired way of doing it as this often doesn't give you alone an advantage.

The other way is to modify your players velocity. Many games has their own velocity address, which then gets added to the players position. We could read what's in this address, then write back the double amount for a double running speed. We could also hook this and double the values in their registers to ensure a smooth speedhack.

I intend to go with the second option. To find this address we need to make some educated guesses on how they work. My assumption is that if I run in one direction the acceleration address gets filled with some value and once you run in constant speed, this speed value stays at a positive value. I'm also thinking that there might be multiple speed addresses, but for each direction. So if you are running towards positive X and positive Y, you would have two speed addresses which both makes up XY direction. Because of this it might be useful to look at Z direction when in doubt. I'll attempt to just look for speed just running around because if I'm right I think this would be easier.

Loading up cheat engine I first look for an unknown float value. While standing still I then look for unchanged values. I then move around a little then scan again for unchanged values, making sure I stand still. I now try to walk a little and while walking I look for increased value. Now while walking around at this walking speed, I look for unchanged values. I then start to run and repeat. When I have reached max walking speed, I then go back to walking speed and look for decreased value, then repeat until I'm standing still. I'm now left with a little over 800 addresses.

Found: 886

| Address | Value | Previous | First |
|---|---|---|---|
| 1C8200018C8 | 0 | 0 | 0 |
| 1C821E543E4 | -15.08274174 | -15.08274174 | -15.08274174 |
| 1C821E543E8 | -1.298328638 | -1.298328638 | -1.298328638 |
| 1C824B552F4 | 5.732670307 | 5.732670307 | 5.732670307 |
| 1C825E14884 | -6.615082741 | -6.615082741 | -6.615082741 |
| 1C825E170A8 | 0 | 0 | 0 |
| 1C825FA1540 | 5.887461662 | 5.887461662 | 5.887461662 |
| 1C826188E50 | -4.287798405 | -4.287798405 | -4.287798405 |
| 1C82618953C | 7.040314674 | 7.040314674 | 7.040314674 |
| 1C826343C30 | 8.096235275 | 8.096235275 | 8.096235275 |
| 1C826382C00 | 2.18217802 | 2.18217802 | 2.18217802 |
| 1C8263838B8 | -12.58135223 | -12.58135223 | -12.58135223 |
| 1C8263838BC | -16.62672043 | -16.62672043 | -16.62672043 |
| 1C8263D71CC | 27.94971466 | 27.94971466 | 27.94971466 |
| 1C8264DFD28 | 0 | 0 | 0 |
| 1C8264DFD30 | 0 | 0 | 0 |
| 1C82676AAAC | 0 | 0 | 0 |
| 1C82691E19C | 2.129440784 | 2.129440784 | 2.129440784 |

Since my assumption is that it's adding it's speed to the coordinates, the value should be null once I'm standing still. So I could probably remove all the values that are not null. I'm not left with about 50 values. I tried to set them to a speed and freeze it there, however this didn't seem to give much results. I noticed that my character sometimes glitches a little when walking, but not the enhanced speed I was looking for. I now need to go through all of them and see what accesses these addresses. After going through them all I found this;

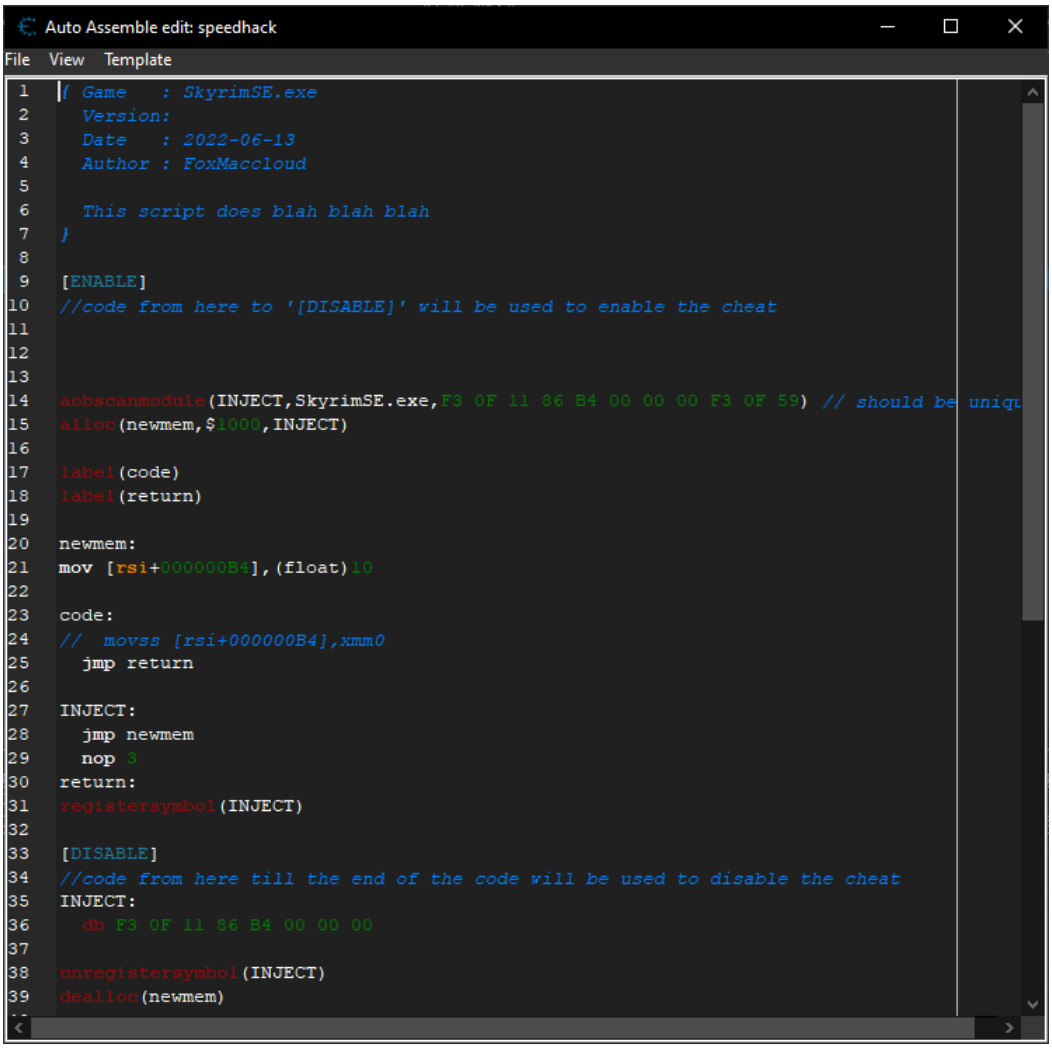
```
movss   xmm1,[7FF73FFC8AA0]
mulss   xmm0,xmm1
movss   [rsi+000000B0],xmm0
movaps  xmm0,xmm2
mulss   xmm0,xmm11
mulss   xmm0,xmm1
movss   [rsi+000000B4],xmm0
mulss   xmm2,xmm6
mulss   xmm2,xmm1
movss   [rsi+000000B8],xmm2
mov     [rsi+000000BC],edi
mov     rcx,r13
call    7FF73F46E900
```

The following opcodes accessed 222465AAAF4

| Count | Instruction |
|---|---|
| 151 | 7FF73F7914AC - F3 0F11 86 B4000000 - movss [rsi+000000B4],xmm0 |
| 151 | 7FF73F7F42A5 - 0F10 A7 B0000000 - movups xmm4,[rdi+000000B0] |

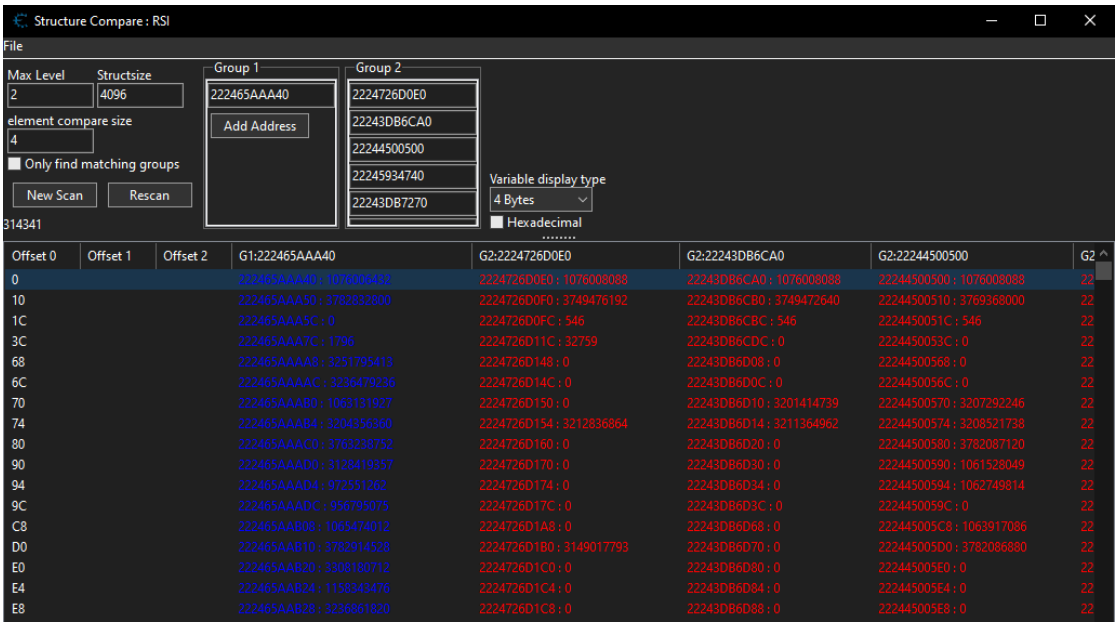
I do also see the above instruction movss [rsi+000000B0],xmm0 which accesses right by. Looking at this rsi address and going to the offsets B0 and B4 and moving in the game, I see that these addresses correlates to my movments in the game. I can then name them "Forwards" and "Right" as they are positive for forwards and right and negative for backwards and left.


To test if this is the correct address, I can use Cheat Engines auto assemble tool and modify the game code quickly right before it's used.

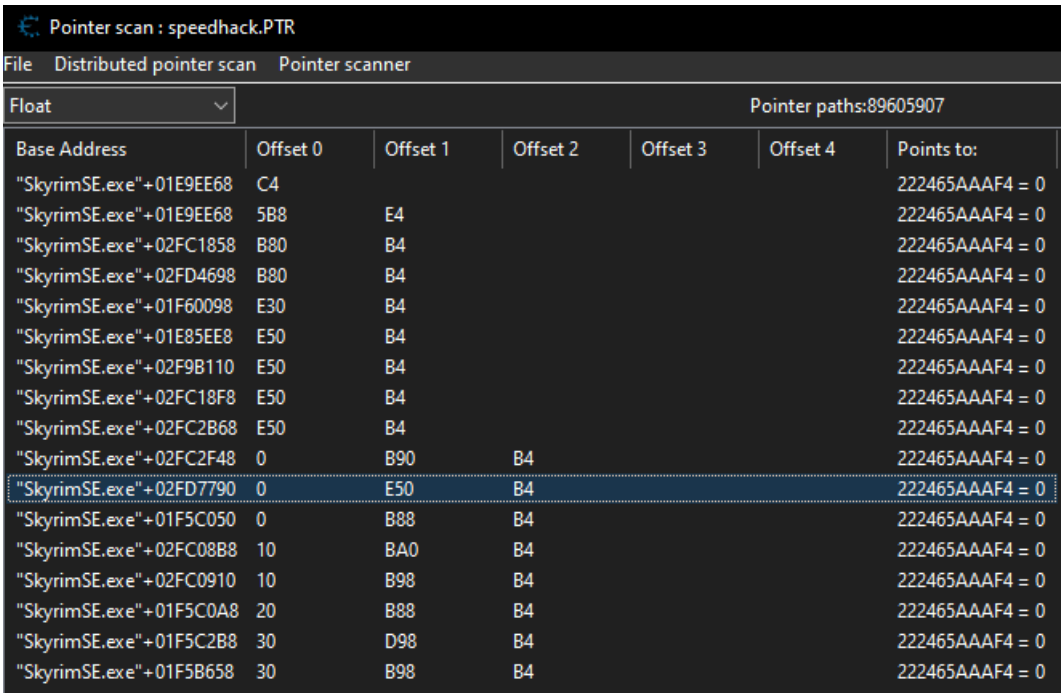



My script will now never write the correct speed into our speed address, but it will always write 10 into the register. Enabling my script I see my character suddenly moving forwards quickly without stopping. Running around at this speed, I can also see NPCs going at full speed. This means that the instruction is shared and all entities uses this to instruction to move. It means we need to do a check to see if it's the player or not, but atleast we know it is the correct address. Why freezing the value didn't work I'm guessing comes down to a race condition. To make sure we get always get our right speed, I'll insert a hook here and if it's the player that's "passing by" I'll multiply the values in these addresses with a constant. If it's not the player, the hook will execute the original bytes and continue as nothing had happened.

Since it's a shared instruction I need to find a way to differentiate out the player from the rest. First I looked at the commonality tool in Cheat Engine to see what differences there were between entities and players.




I found a few differences I could potentially use, however not any I was sure would work. Before I put in the effort to try implementing I used the ptr scan tool to see if I could get some perspective of what where this structure containing acceleration was.




When looking at the these I noticed that one of the paths came from base of the game plus

0x02FD7790. Following the 0x0 offset from here I get into my earlier declared "local player" and if I now go to offset 0xE50 I get taken to the struct which also has my speed. I can now check via the entity list if the rsi register is the same. There are probably a lot of useful stuff one can find at base + 0x02FD7790 however this is beyond the scope of this paper.

In my hook I first grab the address of the current entity from the rsi register, then I compare that one to the local player to see if it's the same address. If it is I'm checking if I'm actually moving in any direction and if I am, I'm multiplying that value with my speed modifier constant. My implementation looks like;

```
1  inline _declspec(naked) void speed_h()
2  {
3      __asm {
4          pop rax
5          mov[currentEntity], rsi;
6
7          movss[rsi + 0x000000B4], xmm0;
8          mulss xmm2, xmm6;
9          mulss xmm2, xmm1;
10     }
11     if (localPlayer->accelerationBase == currentEntity)
12     {
13         float forwards = localPlayer->accelerationBase->Forwards;
14         float right = localPlayer->accelerationBase->Right;
15         if (forwards > 1.0f
16             || forwards < -1.0f
17             || right > 1.0f
18             || right < -1.0f
19             )
20         {
21             localPlayer->accelerationBase->Forwards = forwards * ↘
                   speedHackMultiplier;
22             localPlayer->accelerationBase->Right = right * ↘
                   speedHackMultiplier;
23         }
24     }
25         __asm {
26         jmp[returnAddressSpeed];
27     }
28 }
```

Now I can run at insane speeds through out the ground. However one must be careful when running fast as terrain could fast become a ramp throwing you far up into the air. It's then easy to die to

fall damage.

### 5.6 Conclusion

The goal of this chapter was to give some insight into how cheats for video games are developed. How they are developed, what challenges a game hacker could stumble upon, and some of my thought process. The cheats I developed for Skyrim are only some of the more "desirable" modifications, however there are a lot more, such as automatic aim for shooters, spawning in items and currency and much more. Skyrim in itself is an offline game with no anti-debug/cheat. Hackers who develops cheats usually makes them for online games. Many of these cheats I have made for Skyrim will not work for online games, atleast not in the way they are currently implemented.

## 6 Offline vs Online

Differences between making hacks for offline games compared to online are both very similar and very different. Many cheats such as teleportation, freezing values, etc. will no longer work, however Cheats like ESP often works. This is because usually important values such as what items you have, how much health you have, where you are and more is handled by the server and are validated by the server. The "state" of the game is stored on the server and the server is the one and only truth. An ESP or aimbot could still work just fine because these are hacks that uses information which is given to the client by the server. For other players to appear on your screen, your client needs to know where they are. For games such as first person shooters, this usually is sufficient, but if you want to make cheats for games such as e.g., World of Warcraft, Final Fantasy XIV, runescape, etc. you need to trick the server. There are many ways to trick the server. Simplest way is to play the game in such a way you break it. This clever use of game mechanics is also refereed to as an exploit. Exploits comes in many forms and is found by simply playing the game. Exploits can be used to go to places where you aren't supposed to go, get bosses stuck so you can easily kill them, or duplicate game currency. Since these are accomplished with just playing the game, often exploitation goes unpunished since the exploiters didn't exactly do anything other then playing the developers game. However if you really want to trick the server, you gotta talk to it.

### 6.1 Packets

The game is stored on the server and the server is the one and only truth. The server will sync your game client, which means that, after you authenticate to the game, you will receive packets containing your game data such as gold, location, inventory, etc. When you play the game, you generate some sort of "events", which results in packets being sent to the server. An example of such events, if you move in the game, you are sending a packet to the server telling it that you are moving in one direction and the server should update your XYZ location. Another example is that you are requesting the purchase of a new item. In both of these cases, the server will check if you could make this move and will either accept or deny that repositioning. If you try to move from one side of the world to the other, the server will obviously know that this is impossible and deny your

move. You could possibly make a flyhack or speedhack out of this depending on how good their validation is. You might not go from one side of the world to the other, but you could probably split it into segments that the server could accept. For your purchase request it will check if you can afford this purchase and if you can, it will put that item into your inventory and subtract the currency from your total. If you cannot afford it, you are denied the request.

Because of this server side validation, hacking multiplayer games is hard as you cannot just set your inventory, location etc. A good way to do this then is to just automate the boring stuff. You could automate leveling, resource gathering, etc. which is done by tricking the client to send packets on your behalf. Since packets are the interface to tell the server what you are doing you could intercept these packets, make a man in the middle proxy. With this proxy, you can inspect, modify, create, resend packets to the server and see what happens. There are infinite possibilities if you make a proxy. You could run a headless client. Headless client is a game that you could run in the terminal. Since you are running it in the terminal you don't need to allocate a lot of resources to the game and you could run hundreds of games in parallel. All these accounts can then bot to farm resources, etc.

Since you can now create your own packets with a proxy, you could also attempt to just ask the server to give you infinite money. The server would probably say now, but you could ask indirectly. The "holy grail" of these kinda hacks are integer overflows. Often times currencies in games are stored as an unsigned datatype. This means that the value can never be negative. So if could send a package which subtracts an amount from yourself that would lead to you going negative, you would overflow and you'd suddenly end up with the maximum possible value. Hackers can then sell this currency create out of thin air on the black market, trading fake money for real money. Creating an absurd amount of currency out of thin air has the effect that it's destroying these games economy and integrity which makes the game worse for everyone else.

Because of how powerful these packet tools can be these packets are in most cases encrypted. Their implementation usually comes like;

```
1 game::assemble_packet();
2 Winsock Layer
3 Network Layer
4 game::disassemble_packet();
```

Usually we don't care about the winsock layer as by the time the packet reaches here it's already encrypted. However we could hook the assemble_packet function and read the data before it's encrypted. To find this packet creation function, we can look at windows internal send function, ws2_32.dll.send(). Here we can set a breakpoint and see the return address. We can also try to see where the encrypted data comes from and go upstream from there.

# 7 Final Fantasy XIV

Final Fantasy XIV is a an MMO (massive multiplayer online) game created by Square Enix. It was release in 2010. The game re-released in 2013 to replace the 2010 versions commercial failure. With this re-release the game got a new engine, improved server infrastructure and revamped gameplay. After this re-release, the game has gotten a huge following [8]. This game comes with no client-side anticheat and minimal antidebugging, but it has a lot of server-side checks.

## 7.1 Creating my packet tool

Creating my packet tool took some time. Before I started look for the encrypted buffer I just made a framework which hooks the ws32_2.send() and ws32_2.recv() functions. We can see on MSDN that these functions takes 4 arguments. The send socket, the send buffer, the length of the buffer and flags. Knowing the x64 calling convention, I know that these goes into rcx, rdx, r8, r9 and the rest goes on the stack.

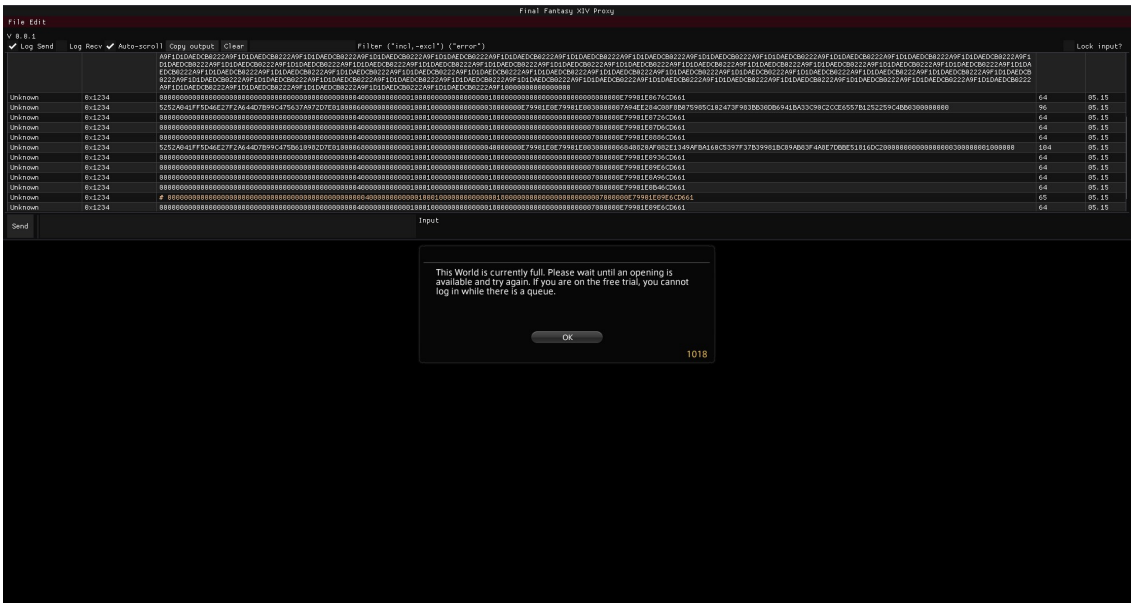
```
1  void __declspec(naked) send_hook_hk()
2  {
3      __asm {
4          pop rax;
5      }
6
7      __asm {
8          mov this_socket_send, rcx
9          mov this_buffer_send, rdx
10         mov this_len_send, r8
11         mov this_flags, r9
12         mov[rsp + 0x8], rbx
13         mov[rsp + 0x10], rbp
14         mov[rsp + 0x18], rsi
15     }
16     __asm {
17         jmp[return_address_send]
18     }
19 }
```

After implementing my hooks, I made it write the buffer to an allocated console. When I now move in the game I can see that I send packages to the server.


I also noticed that when I sent messages in the chat, that I could read my message. This could hint to the traffic not being encrypted and that I had the correct buffers.

I made a graphical user interface to show the packet traffic live in game. This gui took the form of a console at the top of my screen I can show and hide at will. I also gave it some settings such as being able to check if I want to see only sent packages or received packages. I also translated the data from ascii to hex as it's easier to dissect bytes instead of strings.

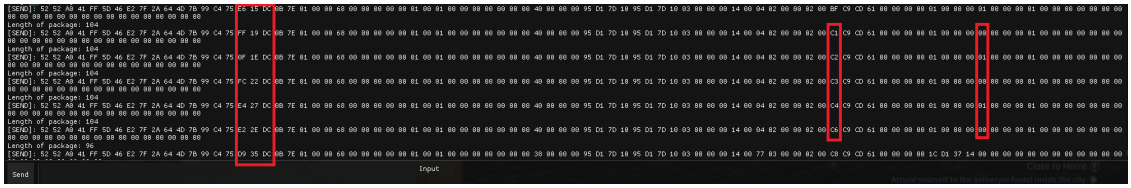

Since I now have a kinda working proxy I tried to resend a package with it. When I did, I instantly disconnected from the server.



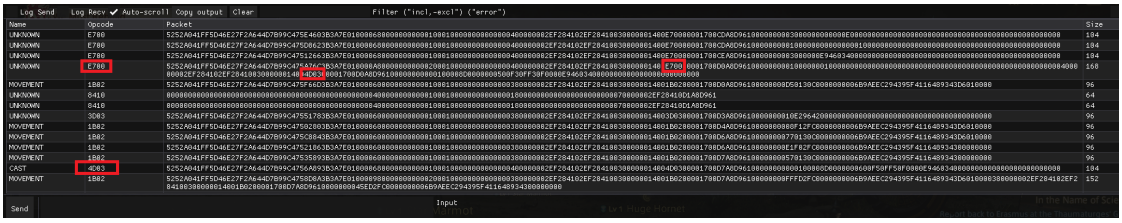

My assumption of what happened is that the server is getting a package that is to old and it therefore think the client is in desync. The server will then kick you off to force you to resync with the server.

## 7.2 Dissecting the protocol

So to be able to modify the data, I need to understand how it's built up. To do that I need multiple packets as samples and figure out how they are different. I started changing my stance in the game over and over again and looking at my packet tool and looking at each packet I could see that some packet that are very similar still had some differences.



There are some unknown bytes, something that looks like a counter and what appears to be a boolean. I then started comparing other packets and their differences. At some offset I believe to have found a "packet ID". A packet ID is a packet which describes what kinda packet it is. E.g., the ID "0xDEAD" could be that you are moving and "0xBEEF" could be that you are casting a spell. I added this offset to my packet tool and gave it it's own column.



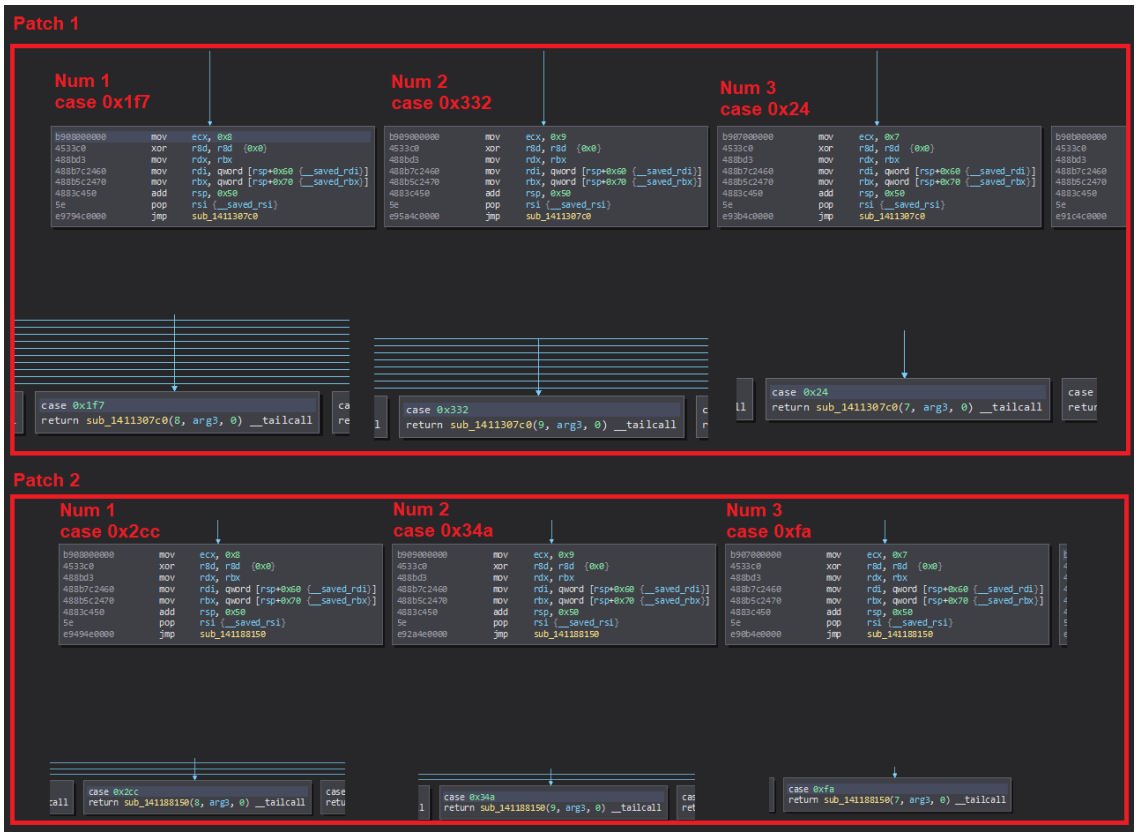
Looking back at the packages I found that more of those unknown bytes changed, they were longer then 3 bytes. With the help of a friend we came to the conclusion that this was a timestamp. The timestamp was milliseconds since epoch that the packet was sent. If I wanted to have a working proxy I'm going to have to add both the counter and the epoch timestamp to the new packets. There is also a problem that multiple packets could be send in package, but I'm ignoring that for now.

```
1  uint64_t get_time_ms()
2  {
3      std::chrono::milliseconds epoch = std::chrono::duration_cast<↘
           std::chrono::milliseconds>
4          (std::chrono::system_clock::now().time_since_epoch());
5      return epoch.count();
6  }
7  ...
8      if (timestamp)
9      {
10         uint64_t current_time = get_time_ms();
11         memcpy(&send_buffer[16], &current_time, sizeof(current_time↘
```

```
            ));
    }
    if (counter)
    {
        counter += 0x1;
        memcpy(&send_buffer[64], &counter, sizeof(counter));
    }
...
```

After Adding these features, I'm sometimes able to send custom packages to the server without disconnecting. I can buy and sell items to shopkeepers using my proxy. However it's not perfect as I sometimes disconnect. I'm guessing that I might have gotten the counter or timestamp a little wrong, but just good enough to work.

I now started mapping out known packet IDs. Like 0x034D was spell casting etc. however in the meanwhile, the game updated. After the update all the packet IDs changed. My friend again told me that this is common and for some games they might even be rotated daily. The amount of work done to demystify these IDs compared to how often they are changed far exceeds what it's worth. This means I have to automate it. When researching this topic, I found that there was a private server project called "Sapphire". They seem to have walked this path before me and stumbled upon this problem as well. Their way of solving this problem was very creative. To me it looked like they solved it by seeing that the order of handlers are almost the same between builds unless new handlers are added or removed.



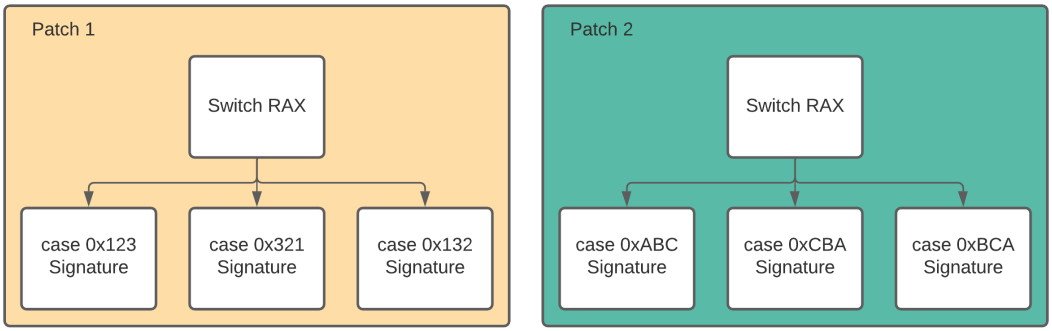

By going after order and then seeing what jumps takes one to each function you can get the new IDs automatically [9]. His implementation of this used the IDA Pro python API to calculate the jump.

I wanted to do this pragmatically with having to rely on expensive software. Making my own implementation proved very difficult. I was thinking about making signatures of all the jumps then calculate which case argument would take me to each signature. To calculate out the case arguments, I could take the default argument and use that as an index for how many case arguments there is. I could then find the jumptable and loop through each entry in the jump list to find the address of where I end up after the jump.

```
PacketFunction:
48895c2418          mov     qword [rsp+0x18 {__saved_rbx}], rbx
56                  push    rsi {__saved_rsi}
4883ec50            sub     rsp, 0x50
8bf2                mov     esi, edx
498bd8              mov     rbx, r8
410fb75002          movzx   edx, word [r8+0x2]
8bce                mov     ecx, esi
e8d5805bff          call    sub_14073b390
0fb75302            movzx   edx, word [rbx+0x2]
8d429a              lea     eax, [rdx-0x66]
3d80030000          cmp     eax, 0x380
0f8761010000        ja      0x14118342e
```

Here we can see that it's 0x380. The next jump will determine if it should jump to default or if it has a case.


The base address of the binary gets loaded into r8, then in rax we will have our case argument. It's multiplied by 4 because that's the size of the entries in the jumptable. The 0x112f840 is the offset from the base to our jumptable. If I now make signatures of all the packets, I now have their addresses. For finding the case argument for my packet I could then loop through the jumptable and find all the addresses. If e.g., a movment signature gives me the address 0xDEADBEEF and looping through the jumptable gives me 0xDEADBEEF I would have a match and the case argument would be the index of that address in the table.




I was not able to make this work sadly and my signatures looked to be very hit or miss between patches. I have been studying this assembly for months without any progress, but I'm not sure if it's an implementation problem or if my understanding is wrong.

Sadly I was not able to solve the packet ID problem. However even if I don't exactly know what packet I'm sending out, I can still guess. So integer overflows and similar are still possible. The next step is to use the proxy to try to find something which could be exploited.

# 8 Conclusion

I hope this paper has given some insight into what it takes to make cheats for video games and the amount of work that is behind it. There are many reasons why people modify video games. Some people do it for fun, some for profit and some for both. If you decide to have fun with the games, it's important not to harm the player and player experience, disrupt game services, but the general rule is "don't do anything that would inpact a game companies bottom line". "if you do you may expect a knock from a lawyer at your doorstep, or any other legal consequence. Cheating usually goes against game policies because most of them breaks the games integrity completely. For first person shooters, they can make you unbeatable and for MMOs break down the "market value" completely. In skill based games you are going to have black markets that sells unfair advantages such as wallhacks, aimbots, etc. In more social games with in game currencies and items you are going to see these for these currencies and powerful items for sale on third party markets. You will also see bots who play for you while you are not at your computer.

# 9 Bibliography

[1] Intel. *Introduction to x64 Assembly*. https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf. [Online; accessed 10-Febuary-2022]. 2021.

[2] Douggem. *ObRegisterCallbacks and countermeasures*. https://douggemhax.wordpress.com/2015/05/27/obregistercallbacks-and-countermeasures/. [Online; accessed 12-Febuary-2022]. 2015.

[3] Rake. *How to Bypass Kernel Anticheat & Develop Drivers*. https://guidedhacking.com/threads/how-to-bypass-kernel-anticheat-develop-drivers.11325/. [Online; accessed 04-June-2022]. 2020.

[4] Cheat-Engine. *Anti-Cheat Bypassing Guide for Noobs*. https://vk.com/@reverse_engine-how-to-bypass-anticheat-start-here-beginners-guide. [Online; accessed 19-Febuary-2022]. 2020.

[5] Hellscream, Ferib. *Reversing Common Obfuscation Techniques*. https://ferib.dev/blog.php?l=post/Reversing_Common_Obfuscation_Techniques. [Online; accessed 30-May-2022]. 2022.

[6] Wikipedia. *The Elder Scrolls V: Skyrim — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/The_Elder_Scrolls_V:_Skyrim. [Online; accessed 14-January-2022]. 2022.

[7] Jeremiah. *Understanding the View Matrix*. https://www.3dgep.com/understanding-the-view-matrix/. [Online; accessed 23-April-2022]. 2011.

[8] Wikipedia. *Final Fantasy XIV — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Final_Fantasy_XIV. [Online; accessed 15-June-2022]. 2022.

[9] Adam. *Fixing Packet Opcodes*. https://sapphireserver.github.io/dev/2019/12/23/fixing-opcodes.html. [Online; accessed 15-June-2022]. 2019.