Research Report (Cheat engine for games)

Stefan-Nikola

2024

Contents

Part I

How do you build cheating engine for games?

What are the key functionalities and components of a cheating engine?

A cheating engine is a tool designed to modify games or applications, giving users an unfair advantage or altering game aspects that are typically inaccessible. Here's a simple explanation of how a cheating engine works:

First, it scans all the programs running on the device to find the game you want to modify. Once it identifies the game, it enters the game's memory, where all the in-game information is stored. The cheating engine then searches for the specific in-game value that needs to be changed, such as player health or ammunition count. After locating all instances of that value, it can be changed by writing a new value and saving the changes. This process successfully manipulates the game, altering its behavior as desired.

A cheating engine comprises several key functionalities and components that collectively enable the manipulation of a target game or application. These components are essential for identifying game data, altering the game state, and ultimately achieving the desired cheating outcomes: [1]

1. One of the primary functionalities of a cheating engine is the process scanner. This component is responsible for scanning the running processes on the operating system (OS) to identify the target game or application. By obtaining the Process ID (PID) of the target process, the cheating engine gains access to the process's memory space, laying the foundation for subsequent memory manipulation.

2. Once the target process has been identified and its memory space accessed, the cheating engine can proceed with memory scanning. Memory scanning involves systematically searching through the process's

memory to locate specific data structures, variables, or values relevant to the game's state. Through memory scanning, the cheating engine can identify critical game attributes such as player health, ammunition count, or in-game currency.

3. Following memory scanning, the cheating engine facilitates memory editing as a core functionality. Memory editing allows the cheating engine to modify the values stored in memory addresses associated with various game elements. By altering these values, the cheating engine can manipulate the game's behavior in real-time, enabling cheats such as infinite health, unlimited ammunition, or enhanced character attributes. Memory editing serves as the cornerstone of cheating functionality, empowering users to exert control over the game's mechanics and outcomes.

What programming language should be used for the creation of the cheating engine?

Selecting the appropriate programming language is a critical decision when developing a cheating engine. Among all the existing languages, C++ emerges as a highly suitable choice for the creation of a cheating engine. C++ is known for its power and versatility. It's widely used in many industries but is particularly popular in game development. Interestingly, while it is great for making games, it is also excellent for creating cheat engines to hack those games. The main reason for this is its incredible performance in memory manipulation, which is the core component of a cheating engine. C++ also has extensive documentation, rich libraries, and tools. Here are the reasons why C++ was chosen as the language for this project: [1]

1. First and foremost, C++ offers a wealth of functionalities and libraries that are invaluable for cheat engine development. Its rich standard library provides extensive support for tasks such as memory manipulation, process injection, and low-level system interactions, which are fundamental components of cheating functionality. Additionally, the availability of third-party libraries and frameworks further enhances the capabilities of C++ for cheat engine development, enabling developers to leverage pre-existing solutions for common challenges.

2. Furthermore, C++ excels in memory management, making it particularly well-suited for tasks involving memory manipulation. Given that memory manipulation lies at the core of cheating functionality, C++'s robust memory management capabilities enable developers to imple-

ment sophisticated cheats with precision and efficiency.

3. Another compelling factor in favor of C++ is its widespread adoption and extensive documentation. As one of the most popular programming languages in the software industry, C++ boasts a vast community of developers and resources, including tutorials, forums, and documentation. This abundance of support makes it easier for cheat engine developers to troubleshoot issues, acquire knowledge, and collaborate with peers.

4. Additionally, personal familiarity with C++ can signify expedite the cheat engine development process. For developers who already possess experience with the language, leveraging C++ for cheat engine development allows them to capitalize on their existing knowledge and skills, minimazing the learning curve and accelerating the development.

5. While C could also be considered a viable option, particularly for its simplicity and suitability for low-level programming tasks, C++ offers distinct advantages as it has all the C features, while having additional features and libraries only for C++.

What OS should be used as the base of target? Should it be Windows or Linux?

When considering the OS for the development and usage of a cheating engine, one must weigh various factors to determine the most suitable platform. While both Windows and Linux offer their own advantages and challanges, Windows often emerges as the preferred choice for several reasons. [1]

1. First and foremost, Windows is a more suitable environment for cheat engine development due to its widespread usage in the gaming community. The majority of PC games are developed and optimized for Windows, making it the primary platform for gamers and consequently cheat developers. This ensures a larger user base for cheat engines and a broader range of potential targets.

2. Additionally, Windows offers comprehensive support for the development and execution of cheat functionalities. The operating system provides robust tools and libraries for memory manipulation, process injection and code execution, facilitating the implementation of cheats with relative ease. Moreover, Windows-based cheat engines benefit from plenty of existing documentations, articles, tutorials and community resources explaining the development process.

3. Furthermore Windows APIs and system architecture offer greater and easier control over game processes and memory. This enables cheat developers to bypass anti-cheat mechanisms more effectively and implement sophisticated cheats that are difficult to detect.

4. In contrast, while Linux provides certain advantages such as open-source flexibility and lower resource consumption, it presents far more challanges like the limitation of Linux gaming market, less articles, tutorials and resources in general and the absence of standardized gaming APIs, thus making Windows far more popular of a choice amidst cheat engine developers.

How is a cheating engine detected so it can be build to avoid detection? Example of anti cheats: VAC, VANGUARD, Easy Anti-Cheat.

Anti-cheat systems like VAC (Valve Anti-Cheat), Vanguard, and Easy Anti-Cheat employ various techniques to detect cheating engines and unauthorized modifications to game processes. Some common methods used for cheat detection include: [2]

* Signature Scanning: Anti-cheat systems often employ signature scanning techniques to detect known patterns or signatures of cheat software within game processes. These signatures could be specific sequences of bytes or memory patterns associated with cheats.

* Behavioral Analysis: Anti-cheat systems monitor the behavior of game processes and players in real-time. They analyze various gameplay metrics, such as movement patterns, aiming accuracy, and interaction with game objects, to detect abnormal or suspicious behavior indicative of cheating.

* Memory Inspection: Anti-cheat systems inspect the memory of game processes to detect unauthorized modifications, such as altering game variables (e.g., health, ammunition) or injecting code into the game process to gain unfair advantages.

* Kernel-Level Protection: Some anti-cheat systems operate at the kernel level to protect game processes from unauthorized access and modifications. They utilize kernel-mode drivers to monitor and control system-level activities, preventing cheats from tampering with game processes.

* Heuristic Analysis: Anti-cheat systems use heuristic algorithms to analyze player behavior and game state for deviations from expected norms. They identify suspicious patterns or anomalies that suggest cheating activity, even if no specific cheat signatures are detected.

Countermeasures that can be used in order for the cheats to avoid detection could be:

* Code Obfuscation: Use techniques like code obfuscation to make cheat software harder to analyze and detect. This involves encrypting or obfuscating code logic and data structures to make it more difficult for anti-cheat systems to identify cheat signatures.

* Dynamic Code Injection: Employ dynamic code injection techniques that bypass static signature scanning by injecting cheat functionality into game processes at runtime. This makes it harder for anti-cheat systems to detect cheat signatures during initial scanning.

* Kernel Mode Exploitation: Develop cheats that operate at the kernel level to evade user-mode anti-cheat detection. By bypassing user-mode protections and operating directly in kernel space, cheats can avoid detection by traditional anti-cheat mechanisms.

* Anti-Detection Features: Implement anti-detection features in cheat software to detect and evade anti-cheat detection mechanisms. This includes techniques like heartbeat emulation, which simulates normal player activity to evade behavioral analysis.

* Constant Updates: Regularly update cheat software to adapt to evolving anti-cheat detection methods. Anti-cheat systems frequently update their detection algorithms and techniques, requiring cheat developers to continuously update their software to avoid detection.

Part II

Appendix

Appendix A: Sub-question 1

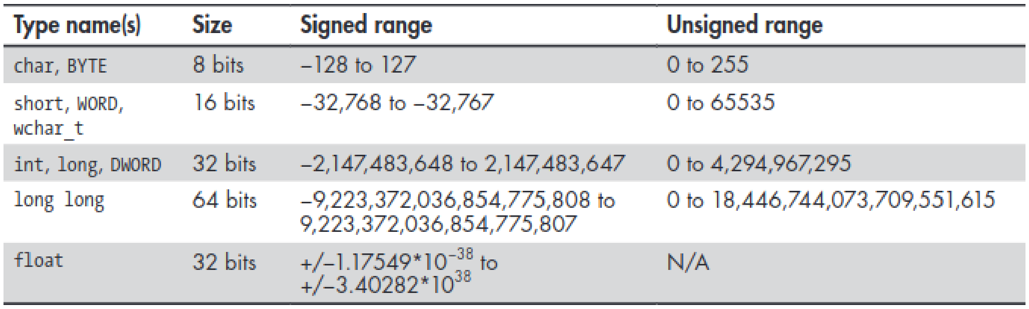
First thing that needs to be understood is how variables and data manifests in memory. That is the main functionality of a cheat engine and understanding this is crucial in the creation of a functional cheating engine.

In computer memory, variables and data are stored in a structured and organized manner, allowing programs to efficiently access and manipulate them. When a game or any application runs, it allocates specific regions of memory to store different types of data, including variables that represent the game's state, such as player health, ammunition, or score.

Memory is essentially a large array of storage locations, each with a unique address. Variables and other data are stored at these addresses. For example, an integer variable might be stored at memory address 0x00400000, while a string could start at address 0x00400004. The exact memory address where a variable is stored is determined at runtime by the operating system and the application's memory management system. [1]

There's different types of data. The main ones needed are:

1. Numeric Data - Numeric data types are fundamental in game hacking because they represent critical values like player health, mana, location, and level. These data types are straightforward in memory, having fixed sizes and predictable alignments. Understanding their representation is essential for manipulating game data effectively.



Type name(s)	Size	Signed range	Unsigned range
char, BYTE	8 bits	–128 to 127	0 to 255
short, WORD, wchar_t	16 bits	–32,768 to –32,767	0 to 65535
int, long, DWORD	32 bits	–2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
long long	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615
float	32 bits	$+/-1.17549*10^{-38}$ to $+/-3.40282*10^{38}$	N/A

Figure 1: Numeric Data Types

Something to keep in mind is that the memory is represented in little-endian ordering, meaning: Numeric values are stored in little-endian format, meaning the least significant byte is at the lowest address. For instance, a DWORD value of 0x0A0B0C0D is stored as 0x0D (the least significant byte) 0x0C 0x0B 0x0A (the most significant) in memory.

Another thing is that Floats are more complex due to their three components: sign, exponent, and mantissa. These components allow floats to represent a wide range of values, including fractions. The actual value of a float is derived by evaluating the expression mantissa × 10 n (where n is the exponent) and adjusting for the sign.

```cpp
unsigned char ubyteValue = 0xFF;
char byteValue = 0xFE;
unsigned short uwordValue = 0x4142;
short wordValue = 0x4344;
unsigned int udwordValue = 0xDEADBEEF;
int dwordValue = 0xDEADBEEF;
unsigned long long ulongLongValue = 0xEFCDAB8967452301;
long long longLongValue = 0xEFCDAB8967452301;
float floatValue = 1337.7331;
```

Listing 1: Example C++ initializing various numeric types

Address	Size	Data	Object
0x00BB3018	1 byte	0xFF	ubyteValue
0x00BB3019	1 byte	0xFE	byteValue
0x00BB301A	2 bytes	0x00 0x00	Padding
0x00BB301C	2 bytes	0x42 0x41	uwordValue
0x00BB301E	2 bytes	0x00 0x00	Padding
0x00BB3020	2 bytes	0x44 0x43	wordValue
0x00BB3022	2 bytes	0x00 0x00	Padding
0x00BB3024	4 bytes	0xEF 0xBE 0xAD 0xDE	udwordValue
0x00BB3028	4 bytes	0xEF 0xBE 0xAD 0xDE	dwordValue
0x00BB302C	4 bytes	0x76 0x37 0xA7 0x44	floatValue
0x00BB3030	8 bytes	0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF	ulongLongValue
0x00BB3038	8 bytes	0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF	longLongValue

Table 1: Memory Representation

2. String Data - Strings are commonly associated with text, but they are more versatile than just holding textual information. At a low level, strings are essentially arrays of numeric objects that are arranged linearly and without alignment in memory. While they are often used to represent text, they can also hold other types of data.

   Array of Numeric Objects: Strings are essentially arrays, with each element representing a numeric object.

   Null Terminator: In C-style strings, the end of the string is marked by a null terminator, which is a character with the value 0x0. This null character indicates the end of the string.

```c
// char will be 1 byte per character
char* thinStringP = "my_thin_terminated_value_pointer";
char thinStringA[40] = "my_thin_terminated_value_array";
// wchar_t will be 2 bytes per character
wchar_t* wideStringP = L"my_wide_terminated_value_pointer";
wchar_t wideStringA[40] = L"my_wide_terminated_value_array";
```

Listing 2: Example String Declarations

In the provided example, thinStringP is a pointer to a null-terminated string, thinStringA is an array of characters with a maximum length of 40, and similar declarations are made for wide strings (wchar-t) as well. Understanding how strings are represented in memory, including their null terminators and encoding types, is essential for efficient manipulation and processing of string data.

3. Data Structures - Unlike individual data types, structures are containers that hold multiple pieces of simple, related data. In the context of game hacking, understanding structures is crucial because many game data are organized into structured formats. By identifying these structures in memory, game hackers can effectively mimic them in their own code, reducing the number of memory addresses they need to manipulate.

   Structures represent a collection of objects, and when these structures are stored in memory, the arrangement of their elements is crucial. Unlike individual data types, structures do not visibly manifest in memory dumps. However, by examining the memory layout of a structure, we can observe important differences in both the order and alignment of its elements.

```
struct MyStruct {
    unsigned char ubyteValue;
    char byteValue;
    unsigned short uwordValue;
    short wordValue;
    unsigned int udwordValue;
    int dwordValue;
    unsigned long long ulongLongValue;
    long long longLongValue;
    float floatValue;
};

MyStruct* m = 0;
printf("Offsets: %d,%d,%d,%d,%d,%d,%d,%d,%d\n",
    &m->ubyteValue, &m->byteValue,
    &m->uwordValue, &m->wordValue,
    &m->udwordValue, &m->dwordValue,
    &m->ulongLongValue, &m->longLongValue,
    &m->floatValue);
```

Listing 3: Example Structure Element Order and Alignment

- Explanation

  This code declares a structure named MyStruct containing various data types.

  A pointer m is created and assigned to point to an instance of MyStruct at memory address 0.

  The printf() function is then used to print the offsets of each member of the structure relative to the start of the structure.

- Observations

  The offsets printed represent the distance between the start of the structure and each member.

  Members are ordered in memory exactly as they were defined in the code.

  The alignment of structure members is not like globally scoped variables; instead, it is determined by the compiler's alignment rules.

Structure members are aligned on addresses divisible by either the struct member alignment or the size of the member, whichever is smaller.

If padding is necessary to achieve alignment, it will be inserted by the compiler to ensure proper alignment of structure members.

4. There's other types such as Unions, Classes and VF Tables, but they wouldn't be of such importance in the development of a simple cheating engine.

A.1 Detailed Explanation and Examples

1. Obtaining the Game's Process Identifier [1]

   To interact with a game's memory, you first need to obtain its Process Identifier (PID), a unique number assigned to each active process in the system. There are two primary methods to acquire the PID:

   (a) Using the Game Window's Handle: If the game has a visible window, you can retrieve its PID by calling GetWindowThreadProcessId(). This function requires the window's handle as input and outputs the corresponding PID. Here's how you can do it:

   ```
   HWND myWindow = FindWindow(NULL, "Title of the game
       window here");
   DWORD PID;
   GetWindowThreadProcessId(myWindow, &PID);
   ```

   Listing 4: Obtaining Game's Process Identifier

   In this example, FindWindow() locates the game window based on its title, and GetWindowThreadProcessId() retrieves the corresponding PID.

   (b) Enumerating Processes: If the game is not windowed or the window name is unpredictable, you can enumerate all processes and search for the game's binary name. This approach involves using functions from tlhelp32.h like CreateToolhelp32Snapshot(), Process32First(), and Process32Next().

```
#include <tlhelp32.h>

PROCESSENTRY32 entry;
entry.dwSize = sizeof(PROCESSENTRY32);
HANDLE snapshot = CreateToolhelp32Snapshot(
    TH32CS_SNAPPROCESS,
NULL);

if (Process32First(snapshot, &entry) == TRUE) {
    while (Process32Next(snapshot, &entry) == TRUE) {
        wstring binPath = entry.szExeFile;
        if (binPath.find(L"game.exe") != wstring::
            npos) {
             printf("Game PID is %d\n", entry.
                 th32ProcessID);
             break;
        }
    }
}

CloseHandle(snapshot);
```

Listing 5: Fetching a Game's PID without the Window Name

In this code, CreateToolhelp32Snapshot() creates a snapshot of all processes, Process32First() initializes the iterator, and Process32Next() iterates over the processes. The PID of the game process is obtained by searching for the game's binary name, such as ”game.exe”.

Once the PID of a game is known, a handle to the process itself can be obtained using the OpenProcess() API function. This handle is crucial for performing memory operations such as reading from and writing to game memory. Here's how you can work with OpenProcess():

(a) Function Prototype - The OpenProcess() function prototype is as follows:

```
HANDLE OpenProcess(DWORD DesiredAccess, BOOL
    InheritHandle, DWORD ProcessId);
```

Listing 6: OpenProcess Function Prototype

(b) Desired Access Flags: The first parameter, DesiredAccess, expects one or a combination of process access flags. Common flags used in game hacking include:

   * PROCESS_VM_OPERATION: Allows operations like memory allocation and protection.
   * PROCESS_VM_READ: Allows reading from process memory.
   * PROCESS_VM_WRITE: Allows writing to process memory.
   * PROCESS_CREATE_THREAD: Allows creating remote threads.
   * PROCESS_ALL_ACCESS: Grants all possible access rights.

(c) Example Call - Here's an example call to OpenProcess() with access permissions for reading from and writing to memory:

```
DWORD PID = getGamePID(); // Fetch the PID of the game
HANDLE process = OpenProcess(
    PROCESS_VM_OPERATION |
    PROCESS_VM_READ |
    PROCESS_VM_WRITE,
    FALSE,
    PID
);

if (process == INVALID_HANDLE_VALUE) {
    printf("Failed to open PID %d, error code %d", PID, GetLastError());
}
```

Listing 7: Fetching Game PID and Opening Process Handle

(d) Error Handling - Ensure to handle errors properly, especially if the OpenProcess() call fails. You can use GetLastError() to retrieve the error code.

(e) Closing the Handle - Once you're done with the handle, remember to close it using CloseHandle():

Now that you've obtained a handle to the game process, you're ready to manipulate its memory to achieve desired game hacks. Let's delve into the next steps for accessing and modifying game memory.

2. Accessing Memory [1]

   (a) To manipulate a game's memory, you need to utilize the Windows API functions ReadProcessMemory() and WriteProcessMemory(). These functions allow you to read from and write to the memory of another process, such as a game, enabling external manipulation of the game's state.

   The prototypes for ReadProcessMemory() and WriteProcessMemory() are similar

```
BOOL ReadProcessMemory(
    HANDLE Process, LPVOID Address,
    LPVOID Buffer, DWORD Size,
    DWORD *NumberOfBytesRead
);

BOOL WriteProcessMemory(
    HANDLE Process, LPVOID Address,
    LPCVOID Buffer, DWORD Size,
    DWORD *NumberOfBytesWritten
);
```

Listing 8: ReadProcessMemory and WriteProcessMemory

   Both functions require a process handle (HANDLE Process) and a target memory address (LPVOID Address). When reading from memory, the Buffer parameter should point to an object to hold the read data, while when writing to memory, Buffer should point to the data to be written. The Size parameter specifies the size of the buffer in bytes. The final parameter (NumberOfBytesRead for ReadProcessMemory() and NumberOfBytesWritten for WriteProcessMemory()) is optional and is used to return the number of bytes accessed.

(b) To illustrate how these functions are used, consider the following code snippet:

```c
DWORD val;
ReadProcessMemory(proc, adr, &val, sizeof(DWORD), 0);
printf("Current mem value is %d\n", val);
val++;
WriteProcessMemory(proc, adr, &val, sizeof(DWORD), 0);
ReadProcessMemory(proc, adr, &val, sizeof(DWORD), 0);
printf("New mem value is confirmed as %d\n", val);
```

Listing 9: Memory writing and reading

(c) Before executing code like this in a program, you need to obtain the PID (proc) of the game process and the memory address (adr) you want to read from or write to. The ReadProcessMemory() function retrieves a value from memory and stores it in val. Then, val is incremented and the modified value is written back to memory using WriteProcessMemory(). Finally, ReadProcessMemory() is called again to confirm the new memory value. Note that val is not a buffer; however, passing &val as the Buffer parameter works because it can serve as a pointer to any static memory structure, as long as its size matches the Size parameter.

Of course those methods are only the most basic examples that can be used for the creation of a simple cheating engine. And of course, games might have some protection mechanisms in place to try and prevent you from editing the memory. In the next section, themes like Memory protection and ASLR is are going to be the primary focus.

A.2 Alternative Options and Justifications

Games can sometimes have protective mechanisms in place to try and stop hackers from interfering with their game. Some of the important ones are Memory Protection and ASLR for singleplayer games. [1]

1. Memory Protection

   When memory is allocated by a game (or any program), it is placed in a page. In x86 Windows, pages are chunks of 4,096 bytes that store data. Because all memory must be within a page, the minimal allocation unit is 4,096 bytes. The operating system can place memory chunks smaller than 4,096 bytes as a subset of an existing page that has enough uncommit- ted space, in a newly allocated page, or across two contiguous pages that have the same attributes

   (a) Understanding the memory protection attributes in x86 Windows is crucial for effective memory manipulation. Let's delve deeper into this topic:

Table 2: Memory Protection Types

Protection Type	**Value**	**Read**	**Write**	**Execute**
PAGE_NOACCESS	0x01	No	No	No
PAGE_READONLY	0x02	Yes	No	No
PAGE_READWRITE	0x04	Yes	No	No
PAGE_WRITECOPY	0x08	Yes	Yes	No
PAGE_EXECUTE	0x10	No	No	Yes
PAGE_EXECUTE_READ	0x20	Yes	No	Yes
PAGE_EXECUTE_READWRITE	0x40	Yes	Yes	Yes
PAGE_EXECUTE_WRITECOPY	0x80	Yes	Yes	Yes
PAGE_GUARD	0x100	No	No	No

Explanation: Each protection type defines the permissions for reading, writing, and executing memory.

Understanding permissions is a crucial part of game hacking as it allows you to know the permission you have, for example, if a page is PAGE_READONLY, programs can read the memory but cannot write to it. Constant data like strings and certain program metadata are typically stored with PAGE_READONLY protection. Assembly code is stored on pages protected with PAGE_EXECUTE_READ.

(b) Changing Memory Protection

When hacking a game, you may encounter situations where you need to access memory in ways prohibited by its protection. In such cases, it's crucial to be able to modify memory protection settings. Fortunately, the Windows API offers the VirtualProtectEx() function for this purpose. Here's its prototype:

```
BOOL VirtualProtectEx(
    HANDLE Process, LPVOID Address,
    DWORD Size, DWORD NewProtect,
    PDWORD OldProtect
);
```

Listing 10: Changing memory protection

The parameters Process, Address, and Size are similar to those in the ReadProcessMemory() and WriteProcessMemory() functions. NewProtect specifies the new protection flags for the memory, while OldProtect can optionally store the previous protection flags. Lastly, it's a good practice to always restore the original protection after making changes to the game so it doesn't detect anything suspicious.

2. Address Space Layout Randomization (ASLR)

ASLR is a security feature implemented in modern Windows systems to enhance system security by randomizing the base addresses of executables and DLLs. This prevents attackers from reliably predicting the memory addresses of system components, making it more difficult to exploit memory-related vulnerabilities.

(a) A simple method to disable ASLR would be to use a editbin command, but that is not possible in production as normal users cannot be expected to disable the security feature. That's why, a dynamic rebasing mechanism can be implemented to bypass ASLR at runtime

To dynamically rebase addresses at runtime, a function like the following can be used:

```
DWORD rebase(DWORD address, DWORD newBase) {
    DWORD diff = address - 0x400000 //0x400000 works only for WIndowsXP;
    return diff + newBase;
}
```

Listing 11: Dynamic Rebase Function

This function effectively neutralizes ASLR by adjusting addresses relative to a known base address (newBase), making memory manipulation consistent across different runs of the application.

To obtain the base address (newBase) dynamically, the GetModuleHandle() function can be utilized in conjunction with CreateRemoteThread() to execute code in the target process and retrieve the base address. Here's how it's done:

* Process Injection: Injecting code into the target process to retrieve information about its memory layout, including the base address.
* Memory Scanning: Scanning the process's memory for known patterns or signatures to identify relevant data structures, which can indirectly reveal the base address.
* Code Reuse: Leveraging existing code within the target process to perform operations that disclose memory information.
* Symbol Resolution: Utilizing debugging or symbol resolution techniques to resolve addresses dynamically at runtime.

This approach effectively bypasses ASLR by dynamically determining the base address of the target process at runtime, ensuring consistent memory manipulation across different executions.

Appendix B: Sub-question 2

In selecting the language for cheat engine development, the choice of C++ emerges as a natural consequence of its dominance in both game and bot development spheres. This decision is primarily driven by its exceptional capabilities in memory management, extensive documentation, and the abundance of rich functionalities and libraries it offers. In this appendix, we explore the rationale behind leveraging C++ for cheat engine development, focusing on its prowess in memory management, the wealth of available documentation, and the robust ecosystem of functionalities and libraries that empower developers in crafting sophisticated cheating mechanisms. [1]

B.1 Detailed Explanation and Examples

```cpp
#include <iostream>

int main() {
    unsigned char codeCave[20] = {
        0xFF, 0x74, 0x24, 0x04,     // PUSH DWORD PTR:[ESP+0x4]
        0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
        0xB8, 0x00, 0x00, 0x00, 0x00, // MOV EAX, 0x0
        0xFF, 0xD0,                 // CALL EAX
        0x83, 0xC4, 0x08,           // ADD ESP, 0x08
        0xC3                        // RETN
    };

    // Print shellcode bytes
    std::cout << "Shellcode Bytes:\n";
    for (int i = 0; i < sizeof(codeCave); ++i) {
        std::cout << "0x" << std::hex << static_cast<int>(codeCave[i]) << " ";
    }
    std::cout << "\n";

    return 0;
}
```

Listing 12: Generating Shellcode in C++

This code snippet showcases how C++ can be used to easily generate shellcode, a sequence of machine code instructions typically used in software exploitation. Here's what the code does:

- An array named `codeCave` is declared to hold the shellcode bytes.
- Each element of the array represents a machine code instruction.
- The shellcode performs a simple operation, pushing a value onto the stack, moving another value into the `EAX` register, calling the function pointed to by `EAX`, and then cleaning up the stack before returning.
- The shellcode bytes are printed to the console for verification.

This example illustrates how C++ enables developers to manipulate low-level byte sequences with ease, making it an ideal choice for crafting sophisticated cheat engine functionalities.

```
#include <iostream>

int main() {
    int health = 100;

    // Pointer to manipulate health value
    int* healthPtr = &health;

    // Decrease health
    *healthPtr -= 50;

    // Print updated health
    std::cout << "Updated health: " << health << std::endl;

    return 0;
}
```

Listing 13: Memory Manipulation with Pointers

This example demonstrates how C++ can be easily used for direct manipulation of memory using pointers. We declare an integer variable `health` and a pointer `healthPtr` pointing to its memory address. By dereferencing the pointer (`*healthPtr`), we can directly modify the value of `health`. This capability is crucial for cheat engine development, where manipulating in-game values like health or ammunition is often necessary.

```cpp
#include <iostream>
#include <Windows.h>

// Original function to hook
int originalFunction(int value) {
    return value * 2;
}

// Hooked function
int hookedFunction(int value) {
    // Inject cheat code before calling original function
    std::cout << "Cheat injected! Value: " << value << std::endl;

    // Call original function
    return originalFunction(value);
}

int main() {
    // Replace original function with hooked function
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    DetourAttach(&(PVOID&)originalFunction, hookedFunction);
    DetourTransactionCommit();

    // Test hooked function
    int result = originalFunction(5);
    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

Listing 14: Function Hooking for Cheat Injection

This example demonstrates function hooking, a technique used in cheat engine development to inject custom code into game functions. We define an original function `originalFunction` and a hooked function `hookedFunction`. Using the Detours library on Windows, we replace the original function with the hooked function, allowing us to inject cheat code before calling the original function. This technique is invaluable for implementing cheats like aimbots or wallhacks by intercepting and modifying game behavior.

B.2 Alternative Options and Justifications

In the realm of cheat engine development, the selection of a programming language plays a pivotal role in shaping the effectiveness and efficiency of the final product. While there are various approaches to creating cheat engines, ranging from low-level memory manipulation to high-level scripting, the choice of programming language significantly influences the development process and the capabilities of the resulting cheat engine. Let's delve into some of the prominent languages considered for cheat engine development and their respective advantages:

Python

Python is one of the best languages used for developing cheating engines due to its extensive libraries and tools. [3]

Advantages

+ **Simplicity and Readability:** Python's clean syntax and readability make it an attractive choice for rapid prototyping and scripting.

+ **Extensive Ecosystem:** Python boasts a vast array of libraries and frameworks, facilitating rapid development and integration of various functionalities.

Considerations

- **Performance Overhead:** While Python excels in simplicity, its interpreted nature introduces performance overhead, which may be a concern for cheat engine tasks requiring high performance and real-time operations.

- **Limited Memory Access:** Python's high-level abstractions may limit direct memory access, potentially hindering the implementation of low-level memory manipulation cheats.

Rust

Rust is by far one of the best languages in terms of memory management, which makes it a good option for the creation of a cheating engine. [4]

Advantages

+ **Memory Safety:** Rust's ownership model ensures memory safety without sacrificing performance, making it suitable for secure cheat engine development.

+ **Concurrency Support:** Rust's features support safe concurrent programming, beneficial for cheat engine tasks involving parallelism.

Considerations

- **Learning Curve:** Rust's strict compiler and ownership rules may pose a steep learning curve for developers unfamiliar with the language.

- **Ecosystem Maturity:** While Rust shows promise, its ecosystem is still evolving, potentially lacking mature libraries for cheat engine development.

C

The original Cheat Engine was written in C which by itself makes this langauge to be a top candidate. [5]

Advantages

+ **Simplicity and Performance:** C's simplicity and efficiency make it suitable for low-level system programming tasks, including cheat engine development.

+ **Direct Memory Access:** C provides direct access to memory, enabling precise memory manipulation crucial for cheat engine tasks.

Considerations

- **Manual Memory Management:** C requires manual memory management, increasing the risk of memory leaks and errors if not handled properly.

- **Safety Concerns:** C's lack of modern safety features may make it prone to vulnerabilities, posing risks in cheat engine development.

- **Lack of libraries:** C doesn't have a lot of libraries in contrast to C++.

While each language offers unique advantages and trade-offs, C++ emerges as a preferred choice for cheat engine development due to its unparalleled capabilities in memory management, performance optimization, and low-level control. With its mature ecosystem, extensive documentation, and wide adoption in the gaming industry, C++ provides developers with the necessary tools and flexibility to craft sophisticated cheats effectively. Ultimately, the choice of programming language should align with the specific requirements and objectives of the cheat engine project, ensuring optimal performance, security, and maintainability.

Appendix C: Sub-question 3

Windows, with its larger user base and extensive documentation, is often the preferred platform for cheat engine development. The abundance of information and community resources available for Windows API makes it easier for developers to find guidance on various aspects of cheat engine creation. The Windows API is a comprehensive set of functions, protocols, and tools provided by Microsoft for interacting with the Windows operating system. It serves as the primary interface through which applications can communicate with the underlying OS, enabling developers to perform a wide range of tasks such as file manipulation, process control, memory management, and user interface creation.

The Windows API is indispensable for cheat engine development due to its extensive and powerful set of tools for interacting with the operating system and running processes. It allows developers to access and manipulate game processes, read and write memory, inject code, control system behavior, and simulate user inputs, all of which are fundamental capabilities needed to create effective and sophisticated cheats. The extensive documentation and large user base of Windows further facilitate the development process, making it easier for cheat developers to find the necessary resources and support. [1]

C.1 Detailed Explanation and Examples

```
HWND myWindow = FindWindow(NULL, "Title of the game window here");
DWORD PID;
GetWindowThreadProcessId(myWindow, &PID);
```

Listing 15: Finding a Window and Retrieving its Process ID using Windows API

1. In this example, the Windows API is used to find a window by its title and then retrieve the Process ID (PID) associated with that window.

    - **Finding Windows:** The `FindWindow` function allows the cheat engine to locate the specific game window by its title. This is essential for targeting the correct application.
    - **Retrieving Process Information:** The `GetWindowThreadProcessId` function retrieves the Process ID of the game window. This PID is crucial for subsequent operations, such as reading or writing memory, injecting code, or controlling the game's process.

```
HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID)
    ;
```

Listing 16: Using OpenProcess to Access a Process by its PID

What `OpenProcess` Does

The `OpenProcess` function is used to obtain a handle to a specified process, given its process ID. This handle can then be used to perform various operations on the process, such as reading and writing memory, querying information, or even terminating the process. Here's a breakdown of what `OpenProcess` does:

- **Opens a Process:** The function opens the specified process using the provided process ID and the desired access level. This allows the caller to interact with the process in various ways depending on the access rights requested.
- **Returns a Handle:** If the function succeeds, it returns a handle to the process. This handle is essential for performing further operations on the process, such as memory manipulation or process control.
- **Access Rights:** The access rights specified in `DesiredAccess` determine what operations can be performed on the process. For example, `PROCESS_ALL_ACCESS` grants all possible rights, while `PROCESS_VM_READ` only allows reading the process's memory.

By using `OpenProcess`, cheat engine developers can gain the necessary access to a game's process to manipulate its memory, inject code, and perform other actions required to implement cheats effectively.

```
BOOL resultRead = ReadProcessMemory(hProcess, address, buffer, size, &bytesRead);
BOOL resultWrite = WriteProcessMemory(hProcess, address, buffer, size, &bytesWritten);
```

Listing 17: Reading from and Writing to Game Memory

What `ReadProcessMemory` Does

The `ReadProcessMemory` function is used to read memory from a specified process. Here's a breakdown of what `ReadProcessMemory` does:

- **Reads Memory:** The function reads data from an area of memory in a specified process.
- **Parameters:**
  - `HANDLE Process`: A handle to the process whose memory is being read. This handle must have been opened with the `PROCESS_VM_READ` access right.
  - `LPVOID Address`: A pointer to the base address in the specified process from which to read.
  - `LPVOID Buffer`: A pointer to a buffer that receives the contents from the address space of the specified process.
  - `DWORD Size`: The number of bytes to be read from the specified process.
  - `DWORD *NumberOfBytesRead`: A pointer to a variable that receives the number of bytes transferred into the buffer.
- **Returns:** The function returns a nonzero value if it succeeds; otherwise, it returns zero. If the function fails, the return value is zero and extended error information can be obtained by calling `GetLastError`.

By using `ReadProcessMemory`, cheat engine developers can read the memory of a game process to inspect or manipulate game variables, such as health, ammo, or scores.

What `WriteProcessMemory` Does

The `WriteProcessMemory` function is used to write data to an area of memory in a specified process. Here's a breakdown of what `WriteProcessMemory` does:

- **Writes Memory:** The function writes data to an area of memory in a specified process.
- **Parameters:**
  - `HANDLE Process`: A handle to the process whose memory is being modified. This handle must have been opened with the `PROCESS_VM_WRITE` and `PROCESS_VM_OPERATION` access rights.
  - `LPVOID Address`: A pointer to the base address in the specified process to which data is written.
  - `LPCVOID Buffer`: A pointer to the buffer that contains the data to be written to the address space of the specified process.
  - `DWORD Size`: The number of bytes to be written to the specified process.
  - `DWORD *NumberOfBytesWritten`: A pointer to a variable that receives the number of bytes transferred from the buffer.
- **Returns:** The function returns a nonzero value if it succeeds; otherwise, it returns zero. If the function fails, the return value is zero and extended error information can be obtained by calling `GetLastError`.

By using `WriteProcessMemory`, cheat engine developers can alter the memory of a game process to change game variables, enabling cheats like infinite health, unlimited ammo, or modified scores.

These are the most essential API functions that are crucial in the creation of a cheating engine. The methods are enough to create a simple cheating engine that opens the game memory, reads from it and writes to it.

C.2 Alternative Options and Justifications

While there isn't a direct equivalent to the Windows API for interacting with processes and memory, developers can utilize various libraries and system calls available in Linux. One common approach is to use the ptrace system call to trace and manipulate the execution of other processes. Additionally, memory reading and writing can be achieved by directly accessing /proc/¡pid¿/mem for reading and writing to the memory of a process.

```c
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t child_pid;
    child_pid = fork();

    if (child_pid == 0) {
        // Child process
        execl("/path/to/game", "game", NULL);
    } else {
        // Parent process
        wait(NULL); // Wait for child to start

        // Attach to child process
        ptrace(PTRACE_ATTACH, child_pid, NULL, NULL);
        wait(NULL); // Wait for child to stop

        // Perform cheats, e.g., modify memory

        // Detach from child process
        ptrace(PTRACE_DETACH, child_pid, NULL, NULL);
    }

    return 0;
}
```

Listing 18: Tracing and Manipulating Processes with `ptrace`

The `ptrace` system call is a powerful mechanism in Unix-like operating systems that allows a process (tracer) to observe and control the execution of another process (tracee). Here's how the provided code snippet works: [6]

- The program forks a child process using `fork()`.
- In the child process, it executes the game using `execl()`.
- In the parent process, it waits for the child to start using `wait(NULL)`.

- Once the child process starts, the parent process attaches to it using `ptrace(PTRACE_ATTACH, child_pid, NULL, NULL)`.

- After attaching, the parent process can perform cheats by manipulating the memory or altering the execution flow of the child process.

- Finally, the parent process detaches from the child process using `ptrace(PTRACE_DETACH, child_pid, NULL, NULL)`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define BUFF_SIZE 1024

int main() {
    int mem_fd;
    char buff[BUFF_SIZE];

    // Open /proc/<pid>/mem for reading and writing
    mem_fd = open("/proc/<pid>/mem", O_RDWR);

    // Read from memory
    lseek(mem_fd, <address>, SEEK_SET);
    read(mem_fd, buff, BUFF_SIZE);

    // Write to memory
    lseek(mem_fd, <address>, SEEK_SET);
    write(mem_fd, buff, BUFF_SIZE);

    close(mem_fd);

    return 0;
}
```

Listing 19: Reading and Writing Memory

This code snippet demonstrates how to read and write memory of a process using the ‘/proc/¡pid¿/mem‘ interface in Linux: [7]

- We open the file descriptor ‘mem_fd‘ by calling ‘open(”/proc/¡pid¿/mem”, O_RDWR)‘, which allows us to read and write to the memory of the process with the specified PID.

- Using ‘lseek‘, we move the file offset to the desired address in the process's memory.
- We then use ‘read‘ to read data from the memory into the buffer ‘buff‘.
- Similarly, we can use ‘write‘ to write data from the buffer ‘buff‘ to the memory at the specified address.
- Finally, we close the file descriptor ‘mem_fd‘.

By directly accessing ‘/proc/¡pid¿/mem‘, we can interact with the memory of a process, enabling us to inspect and modify its contents as needed.

Creating a cheating engine in Linux is just as feasible as building it in Windows. The only drawbacks I found are that Linux does not have that big of a gaming community, there's not as much information as there's on Windows out there. So for building a simple cheating engine, Windows is a better choice due to it's widespread in the gaming, comprehensive information and simplicity done by Windows API functions. Creating a cheating engine in Linux would require some knowledge of Linux before anything since it doesn't have Windows API type of library and it would require different approach as shown above.

Appendix D: Sub-question 4

start

Learning how anti cheats work is a crucial part if we want to create a robust cheating engine. For that reason, we are going to take a look at how two of the most popular anti cheats to date work, VAC and VANGUARD:

VAC (Valve Anti-Cheat)

Valve Anti-Cheat (VAC) is an anti-cheat system developed by Valve Corporation, the creators of popular gaming platforms such as Steam. VAC is designed to detect and prevent cheating in multiplayer games on the Steam platform, ensuring fair and competitive gameplay for all players. Since its inception, VAC has become one of the most widely used anti-cheat systems in the gaming industry, protecting a diverse range of online multiplayer games from cheating and exploitation. [8]

How VAC Works

VAC employs a combination of signature scanning, behavioral analysis, and delayed banning techniques to detect and deter cheating in online games. Here's how VAC operates:

1. Signature Scanning:

   (a) VAC continuously scans game processes for known cheat signatures, which are specific patterns or sequences of bytes associated with cheat software.

   (b) These signatures can include memory manipulation routines, injection methods, or other cheat functionalities commonly used by cheat developers.

   (c) When VAC detects a match with a known cheat signature, it flags the associated player account for further scrutiny.

2. Behavioral Analysis:

   (a) In addition to signature scanning, VAC monitors player behavior and gameplay patterns in real-time to identify suspicious or abnormal activities.

   (b) This includes analyzing movement patterns, aiming accuracy, reaction times, and other gameplay metrics to detect behaviors indicative of cheating.

   (c) VAC compares player actions against established norms and thresholds, flagging players whose behavior deviates significantly from expected patterns.

3. Delayed Bans:

   (a) VAC often employs delayed banning techniques, where flagged accounts are not immediately banned but instead placed in a ”ban wave” queue.

   (b) This delay helps VAC gather additional data and evidence against suspected cheaters, ensuring more accurate and effective enforcement actions.

   (c) Once sufficient evidence is collected, VAC issues bans in waves, targeting multiple cheaters simultaneously to minimize disruption to gameplay.

Vanguard

Vanguard is an advanced anti-cheat system developed by Riot Games, the creators of popular titles such as Valorant and League of Legends. Designed to maintain fair and competitive gameplay, Vanguard is integrated into Riot's games to detect and prevent cheating, ensuring a level playing field for all players. Since its introduction, Vanguard has emerged as a formidable deterrent against cheating and exploitation in online multiplayer gaming environments. [9]

How Vanguard Works

Vanguard employs a multi-faceted approach to detect and deter cheating behavior in online games. Here's an overview of Vanguard's operational methods:

1. **Signature Scanning:** Vanguard continuously scans game processes for known cheat signatures, which are specific patterns or sequences of bytes associated with cheat software. These signatures encompass various cheat functionalities commonly used by cheat developers, including memory manipulation routines and injection methods.

2. **Behavioral Analysis:** In addition to signature scanning, Vanguard employs real-time monitoring of player behavior and gameplay patterns. By analyzing movement patterns, aiming accuracy, reaction times, and other gameplay metrics, Vanguard identifies suspicious or abnormal activities indicative of cheating behavior. Players whose behavior deviates significantly from expected norms are flagged for further scrutiny.

3. **Kernel-Level Protection:** Vanguard operates at the kernel level, providing enhanced protection against cheating at the system level. By monitoring and controlling system-level activities, Vanguard prevents unauthorized access to game processes and resources, making it harder for cheats to evade detection and manipulation.

4. **Proactive Detection Measures:** Vanguard implements proactive detection measures to identify and prevent cheating before it impacts gameplay. This includes preemptive checks during game initialization and continuous monitoring of system integrity to detect unauthorized modifications and exploits.

5. **Swift Enforcement Actions:** When Vanguard detects cheating behavior, it takes swift and decisive enforcement actions to maintain the integrity of the game environment. This may include issuing immediate bans or placing flagged accounts in a ban queue for further investigation, ensuring that cheaters are swiftly identified and removed from the game.

References

[1] Nick Cano. Game Hacking: Developing Autonomous bots for online games, 2016. 3, 5, 7, 12, 18, 21, 23, 26, 33

[2] Anti-Cheat. How does it work? : r/gamedev. https://www.reddit.com/r/gamedev/comments/87i3p1/anticheat_how_does_it_work, May 2024. [Online; accessed 25. May 2024]. 9

[3] Jeanextreme002. PyMemoryEditor. https://github.com/JeanExtreme002/PyMemoryEditor, May 2024. [Online; accessed 25. May 2024]. 30

[4] Writing our own Cheat Engine: Introduction | Lonami's Blog. https://lonami.dev/blog/woce-1, February 2021. [Online; accessed 25. May 2024]. 31

[5] cheat engine. cheat-engine. https://github.com/cheat-engine/cheat-engine, May 2024. [Online; accessed 25. May 2024]. 32

[6] ptrace(2) - Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html, April 2024. [Online; accessed 25. May 2024]. 37

[7] lseek(2) - Linux manual page. https://man7.org/linux/man-pages/man2/lseek.2.html, April 2024. [Online; accessed 25. May 2024]. 38

[8] Valve Anti-Cheat - UnKnoWnCheaTs Game Hacking Wiki. https://www.unknowncheats.me/wiki/Valve_Anti-Cheat, April 2023. [Online; accessed 25. May 2024]. 40

[9] The VALORANT Anti-Cheat: Vanguard - How it Works. https://www.gamechampions.com/en/blog/valorant-anti-cheat-vanguard, May 2024. [Online; accessed 25. May 2024]. 41