ChatGPT

FiveM Anti-Cheat Development: Comprehensive Guide for NexusGuard

Introduction

FiveM is a modification framework for GTA V that lets developers run custom multiplayer servers with extensive scripting and modding capabilities. This flexibility creates a **unique attack surface** for cheating: FiveM supports multiple scripting runtimes (Lua primarily, but also JavaScript and C#) and uses a built-in Chromium browser instance (CEF) for in-game UIs (NUI) [1]. As a result, a robust anti-cheat for FiveM must handle exploits spanning **client scripts, server events, and even UI/web interfaces** [1] [2]. The goal of the NexusGuard project is to build a completely proprietary anti-cheat that is **comprehensive** (covering all common and advanced cheats) and **widely compatible** (usable on any server regardless of frameworks or permission systems). This guide provides a detailed overview of FiveM development basics, known cheats and hacks, and best practices to design an innovative, **multi-layered anti-cheat** solution.

FiveM Server Development Basics

FiveM Architecture and Scripting Environment

FiveM uses a client-server model: **server scripts** run on the dedicated server (using the "FXServer"), and **client scripts** run on each player's game client. A collection of scripts and assets is packaged as a *resource*. Each resource has an `fxmanifest.lua` (resource manifest) that defines which files run on server, client, or both. FiveM natively supports **Lua** and **JavaScript** for scripting on both server and client side (Node.js on the server, and a JS runtime for UI), as well as **C#** (via the .NET/Mono runtime) for scripting [3].

- **Client-side scripts** can call *GTA V native functions* (game engine API calls) to manipulate game state (e.g. spawn vehicles, change player health, etc.), and can trigger events to communicate with the server. They also handle rendering and UI (including the **NUI** browser for HTML/JS UIs).
- **Server-side scripts** typically handle game logic, database interactions, and authoritative decisions. They can trigger events to send data to clients or listen for client events.

Natives: FiveM exposes GTA V's native functions (hundreds of game engine functions like `CreateVehicle`, `SetEntityHealth`, etc.) to scripts. These are powerful but must be used carefully – many cheats abuse natives (e.g. calling `GiveWeaponToPed` to spawn guns) [4] [5]. It's wise to wrap native calls with safety checks. (NexusGuard includes a *Natives Wrapper* that catches errors and provides consistent return values, ensuring a bad native call won't crash the script [6] [7].)

Events: FiveM's event system allows communication between client and server. For example, `TriggerServerEvent("eventName", data)` from client calls a server handler for "eventName". Many server frameworks (like ESX, QBCore, etc.) rely on events for gameplay (e.g. paying a player, giving an item). However, *any client can call any event* by name, which is a major vulnerability if not secured. Cheaters exploit this by calling events with fabricated data (e.g. giving themselves money by triggering a payment event)

[8]. **Never trust client input** – server events must validate every parameter and ensure the caller is allowed that action [9]. We will cover event security in depth later.

Permission Systems and Compatibility

FiveM has an inbuilt ACL (Access Control List) known as **ACE & Principals** that many admin tools (like vMenu, EasyAdmin) use for permissions [10]. Additionally, popular server frameworks have their own role/permission systems (e.g. groups in ESX/QB, vRP user roles, Discord-based perms, etc.). **NexusGuard should operate independently of any specific permission system.** This means the anti-cheat should function on *any server* out-of-the-box, without requiring a particular admin mod or framework.

However, it's useful to allow **integration with permissions** for certain features: for example, server owners may want to designate trusted admins or modes where some cheats (like noclip or teleport) are allowed. NexusGuard can support this by checking either Ace principals or a configurable list of trusted player IDs. The key is to make this optional and configurable. By default, the anti-cheat should apply to everyone, but server owners can whitelist specific people or groups (e.g. an "admin" Ace permission) to prevent false positives when administrators use moderator tools. This ensures any server can adopt NexusGuard regardless of how they manage permissions, and simply toggle compatibility settings if needed.

Resource Independence and Ease of Use

To be widely usable, NexusGuard should be a **self-contained resource** that doesn't conflict with others. It should not rely on hooking into a specific framework's code. Instead, it can detect framework usage at runtime and adapt if necessary. For example, if ESX is present, the anti-cheat might watch for known ESX events being misused (like `esx_pizza:pay` exploit) [11], but if not present, those checks harmlessly do nothing. Similarly, it can use FiveM's global states (like player connecting events) rather than framework-specific player lists. The installation should be as simple as dropping the NexusGuard folder in the server's resources and adding `start NexusGuard` in *server.cfg*. Configuration files can allow tuning of which detections are active, logging preferences, and integration points (like Discord webhooks for alerts, or enabling framework-specific protections). By focusing on **server-side authority** and generic FiveM APIs, the anti-cheat remains *framework-agnostic*.

Multi-Language Strategy for Anti-Cheat Development

FiveM allows anti-cheat logic to be implemented in Lua, JavaScript, and C# (and even C++ for external modules). Each has strengths and weaknesses, so an **optimal solution uses a mix of languages** where appropriate.

- **Lua:** The dominant language for FiveM scripts, used by most resources. Lua is lightweight, easy to learn, and runs using the highly optimized LuaJIT interpreter. Lua has tight integration with FiveM's native functions and game events [12]. For NexusGuard, Lua is ideal for core client-side and server-side logic that needs to interface with other resources. For example, **event handlers, basic state validation, and quick checks** can be in Lua so that they can easily interact with existing Lua-based frameworks and resources [13]. However, Lua is also the primary target of cheat executors (malicious programs that inject Lua code into the client at runtime) [14] [12]. This means Lua-based anti-cheat components must be written securely (e.g. avoid leaving global variables that cheats can tamper

with, and be aware that anything running in Lua on the client can potentially be observed or altered by an injector).

- **JavaScript:** JavaScript in FiveM is typically used for two purposes: **NUI (UI front-end)** and **server-side (Node.js)**. On the client, JS runs in the browser environment (CEF) to create interactive UIs. An anti-cheat can use JS in NUI to, for example, detect if a cheat tries to inject UI elements or open the dev console. Its reach is limited to UI, but it could watch for signs of known mod menu interfaces or anomalous changes in the DOM that indicate an injected menu [15]. On the server side, using Node.js can be powerful for certain tasks – for instance, if NexusGuard needs to perform **complex computations or use external libraries** (the NPM ecosystem) for things like machine learning analysis of player behavior, a Node.js module could handle that [15]. Performance of JS vs Lua in FiveM is debated, and JS is not as commonly used for core game logic, but it's available. In sum, JS is great for **web integrations, UI, and leveraging external services** within the anti-cheat.

- **C#:** FiveM supports C# scripts via a Mono runtime. C# offers a robust, object-oriented environment with strong typing and access to the .NET libraries [12]. In an anti-cheat context, C# can be advantageous for building **more complex or performance-intensive server-side systems**. For example, **GoblinAC** is a FiveM anti-cheat that used C# to implement a secure event proxy, encrypting and validating client events [16]. C# could be used in NexusGuard to create modules that handle things like **secure communication (encryption of events)** or **advanced analysis** in a structured way [17]. C# code is compiled to IL, so client-side C# is a bit harder for Lua injectors to tamper with or read (though not impossible – determined attackers can decompile .NET assemblies) [14] [18]. One potential role for C# is a **trigger protection system**: e.g., having all client TriggerServerEvents go through a C# handler that checks a signature or token – similar to GoblinAC's approach [16]. Another use is handling **concurrency and data structures** more efficiently on the server (C# might handle large sets of player data or run heavy logic more smoothly than Lua). Keep in mind C# scripts need to be compiled and loaded, and not all FiveM server hosts enable C# by default (the server admin must ensure the C# runtime is enabled). Still, including C# for critical components can improve security and possibly performance for NexusGuard.

- **C++:** Regular FiveM resources typically don't use C++, but C++ is the language FiveM itself and many external cheats are written in. If one were to extend anti-cheat beyond the scripting layer, C++ could be used to create a **custom client module or even a separate program** to perform low-level tasks. For example, a **memory scanner or kernel-level driver** to detect cheats would likely be in C++ for speed and access to system APIs [19] [20]. Integrating C++ directly into FiveM is non-trivial (it might require a custom build of FiveM or a carefully injected DLL, which could break EULA or trigger false positives in existing anti-cheats). Given NexusGuard's scope as a resource, C++ might be considered for **external tools** (like a companion process that monitors the game) or highly specialized tasks that scripting can't handle. However, this significantly raises complexity and maintenance burden. It's generally best to maximize what can be done in Lua/C#/JS, and only consider C++ if absolutely necessary for critical performance or deep system access [20].

Hybrid Approach: In practice, a hybrid language strategy is recommended. Lua will handle the bulk of in-game integration (ensuring the anti-cheat can plug into any server easily), C# modules can bolster security for event validation or heavy computations, and JavaScript (Node) can be used for any supplementary services (like sending data to a web dashboard, or performing off-game analytics) [21] [22]. This way, NexusGuard plays to each language's strengths. Crucially, no matter the language, the anti-cheat must

implement the same core principles of not trusting the client and validating actions on the server. The **language used is less important than the architecture** – but using multiple languages can enhance efficiency and security when done thoughtfully. For example, one might implement a **Lua hook** that catches every `TriggerServerEvent` on the client and forwards it to a C# routine for verification before actually invoking the target event (combining Lua's easy integration with C#'s robust handling). The bottom line: **use Lua for integration, C# for muscle, JS for connectivity**, and ensure they all work together seamlessly. (NexusGuard's modular design can reflect this, e.g. separate Lua files for detectors, a C# assembly for critical checks, etc., all orchestrated together.)

Common Cheating Methods in FiveM

Understanding how cheaters operate is key to building effective countermeasures. FiveM inherits many cheats from GTA V modding and introduces new ones through its infrastructure. Below are the **most common and "lethal" cheats/hacks** seen in FiveM, along with how they work and how they can be detected or prevented:

- **Aimbots:** Software that automatically aims and shoots with superhuman accuracy. An aimbot typically reads memory to find other players' coordinates or intercepts game functions related to aiming. It then programmatically moves the player's aim to target enemies instantly. *Detection:* Because aimbots affect how a player aims, detection usually relies on **behavioral analysis** – e.g., tracking impossible aiming patterns like perfectly snapping to heads or extremely low reaction times [23]. Servers can also analyze hit data (many more headshots than normal, or 180-degree flick shots with inhuman consistency). A legitimate player will have natural variation in aim, whereas an aimbot might produce smoothing or snapping that can be statistically identified [23]. An anti-cheat can run periodic server-side checks on player hit accuracy and reaction metrics to flag suspiciously consistent performance.

- **Wallhacks / ESP (Extra Sensory Perception):** These reveal players or objects through walls, usually by drawing outlines or info on the cheater's screen. In FiveM, this is often done by reading positions of other entities from memory (or less commonly, intercepting network data) to know where everyone is [24]. *Detection:* It's hard to directly detect if a player has an ESP, since it's a client-side visual. However, one approach is **behavioral**: catch players who consistently react to things they can't see (e.g., a player pre-aiming or following another through a wall without any in-game way to know their position) [24]. Another approach is memory integrity – some anti-cheats try to **scan the client memory** for signatures of known ESP hacks or check if game memory related to entity lists is being accessed by an external program [24]. FiveM server itself can't scan a client's memory (that would require an AC client or driver), but NexusGuard can implement server-side tricks like **ghost players/objects**: e.g., spawn an invisible dummy entity and see if a supposed "legit" player somehow notices or reacts to it.

- **Mod Menus:** These are GUI menus injected into the game that give cheaters an entire suite of cheats at their fingertips. A typical mod menu can toggle godmode, teleportation, spawn weapons/vehicles, give money, kick players, etc., via a nice interface. They often work by injecting custom scripts (Lua or even DLLs) into the game. *Detection:* Detecting the presence of a mod menu itself is challenging – many are unique or obfuscated. Some methods include **signature scanning** for known menu code or UI elements, checking for suspicious resource names (if the menu runs as a resource), or even capturing screenshots from the client and scanning for the menu GUI (very

intrusive and unreliable) [25]. A more practical approach is to detect **actions coming from menus**: e.g., if a player suddenly triggers a bunch of events or calls natives in rapid succession that normal gameplay wouldn't (spawning 50 vehicles, turning invincible, etc.), that could indicate a mod menu in use. Many menus also hook certain common natives; NexusGuard could monitor calls to critical natives to see if they're invoked in abnormal ways (more on that below). In short, **it's often easier to detect what a mod menu *does* rather than the menu itself**. Layered defenses are wise: have some checks for known menu signatures or unusual resource activity, *and* robust detection for the cheats that menus enable [25] [4].

- **Resource/Item Spawning Exploits:** FiveM servers rely on events for giving items, money, or spawning cars. Cheaters abuse this by directly calling server events that grant rewards without doing the work. For example, a cheater might call an event like `esx_pizza:pay` with a huge amount parameter to give themselves money, or trigger a weapon-give event to get guns [11]. If those events don't have server-side checks, the server will blindly give the reward. *Prevention:* The **solution is entirely server-side** – ensure **server validation** for any such event. The server should *never* trust the client to decide "I delivered a pizza, pay me $1000"; instead, the server should track the job progress and only pay if conditions are met [8]. In practice, server developers must fix insecure scripts as a form of "server hardening" [26]. NexusGuard can assist by **monitoring event usage**: flag or block attempts to call known vulnerable events with improper parameters, and use **honeypot events** (dummy events that legitimate scripts never call) to catch cheaters (e.g., an event named like a common exploit). For instance, one could register handlers for events named after popular framework functions that your server doesn't actually use – if someone triggers "es_admin:banPlayer" on a server that doesn't have that admin resource, it's clearly a malicious call and that user can be banned immediately [27]. In summary, strict server-side permission checks and event whitelisting/blacklisting are crucial [4].

- **Teleportation:** Instantly moving a player to another location without normal travel. Teleports can be achieved by writing to player coordinate memory or calling a native like `SetEntityCoords` on the client. *Detection:* FiveM provides server events like `entityCreating` and state data that can help. A server-authoritative approach will validate movement – for example, **track the distance and time between player position updates**. If a player's position changes by 1000 meters in one tick, that's impossible under normal conditions and indicates a teleport [28]. The anti-cheat can auto-correct or rubber-band the player back and flag them. Another check is to **monitor native calls** on the client: if possible, intercept `SetEntityCoords` calls and block ones that are not allowed (though doing that purely in script can be tricky). A simpler server check: enforce maximum speed limits – if a player moved faster than any vehicle or on-foot speed could achieve, suspect teleport or speed hack [29]. NexusGuard's server module can periodically compare each player's last known position to their current one and calculate velocity; extremely high velocity or bizarre position changes trigger a teleport flag.

- **Speed Hacks:** Making the player move faster than normal (without a teleport jump). This can be done via altering game speed, modifying run/swim speed multipliers, or physics manipulation. *Detection:* Similar to teleport, **server-side movement validation** works: track velocity and acceleration. If someone is sprinting at 50 m/s (far above normal human run speed) or accelerating instantly to top speed, they're likely speed hacking [29]. Also, if the client reports a tick where they moved further than possible given game mechanics, it's a red flag. Speed hack can also manifest as vehicle speed exploits – those can be caught by comparing to vehicle's known max speed. The anti-

cheat can either correct the position or continuously apply a slowdown. On client side, one might detect if certain natives like `SetRunSprintMultiplierForPlayer` are called with out-of-range values (if accessible via a native call monitor).

- **God Mode (Invincibility):** Making a player unkillable (health never decreases, or they constantly regenerate). Cheaters achieve this by either freezing the health value in memory, intercepting damage events, or repeatedly calling natives like `SetPlayerInvincible(player, true)` in a loop [30]. *Detection:* The server can periodically check each player's health and armor values. If a player's health never goes below a threshold despite taking damage events, something is wrong. For example, if five direct hits from a rifle do 0 damage, that player could be in god mode. FiveM has a native `GetPlayerInvincible(player)` which might reflect invincibility status – an anti-cheat can call this on clients to see if it returns true unexpectedly [31] [16]. Also, if health is exactly 200 (max) constantly in situations it shouldn't be, that's a sign. Another angle: monitor **damage events** – FiveM's `weaponDamageEvent` gives the server a chance to modify or cancel damage. If a player is legitimately hit but their health doesn't change, you could catch that. Many anti-cheats simply slay or kick players who appear invincible after warnings. Logging every instance of a player surviving lethal damage can help differentiate lag vs cheat. Some cheats use **semi-god mode** (e.g. setting player's ped to *essential* mode where they revive automatically); those can be caught by checking if a player goes into an injured state or not at expected times. In short, **server-side health validation** and checking invincibility natives can detect god mode [30].

- **Noclip / Flying / No Collision:** The cheater can pass through walls and fly around freely. This is done by disabling collision on their player or toggling a noclip native, often provided by mod menus. *Detection:* The server can detect noclip by unusual movement patterns: e.g., a player moving in a straight line through obstacles with no collisions reported, or Z-coordinate changes that don't match jumping or falling (flying) [28]. Another clue is if a player's position intersects solid objects or they're deep underground or high above the map with no vehicle. FiveM's `GetEntityCollisionDisabled` native might indicate if an entity's collision is off [32]. If available, NexusGuard could call something like that for each player to see if their collision is unexpectedly disabled (though that might be unreliable). Usually, the anti-cheat will rely on **positional sanity checks** – if you're inside a building that normally has no interior, or in coordinates outside the normal map bounds, or your vertical movement is continuous with no falling gravity, you might be noclipping. Pair this with velocity checks (e.g. if moving too fast in vertical axis with no gravity effect).

- **Super Jump / Enhanced Physics:** Cheaters can modify game physics values to jump extremely high or not take fall damage. FiveM has natives that detect if a player is using super jump (`IS_PLAYER_SUPER_JUMP` in GTA natives) [33]. *Detection:* The client or server can call such natives to see if enabled [33]. Also, abnormally high Z-velocity or surviving huge falls could indicate physics cheats. Anti-cheat can simply reset these multipliers if detected or flag the player.

- **Infinite Ammo / Rapid Fire:** Giving unlimited ammo or removing reload times, achieved by using natives like `SetPedInfiniteAmmoClip` or altering weapon data. *Detection:* If a player fires far more bullets than a weapon's magazine size without reloading, that's a sign (e.g. 100 shots from a 30-round clip) – server can observe reload events or ammo count if synced. Rapid fire (increasing fire rate) can be caught if the time between shots is impossibly low; the server event `weaponDamageEvent` might show an unusual rate of damage from one player. Also, check if a player is constantly at max ammo.

- **Explosion/Chaos Spam:** Some cheats spawn massive explosions or fire effects to kill others or lag the server. FiveM triggers a server `explosionEvent` for each explosion, with data on who caused it and what type. Anti-cheat can **listen to** `explosionEvent` and decide to cancel it or punish if it's not legitimate (e.g. a player not authorized to use explosives causing many explosions in a short time) [5] . If someone throws one grenade, it's fine; if they trigger 20 explosions in a second all over, that's a modder. Explosion spam often comes from mod menus (the infamous "atomic bomb" options), which you can curtail by limiting explosion events per player per minute and by checking if the explosion origin is near that player (cheats sometimes spawn remote explosions far away to grief others).

These are the core cheats that **must be prevented or detected**, as they give unfair advantages or ruin the game. NexusGuard's design will layer multiple checks to catch these behaviors. It's worth noting that **many of these cheats can be mitigated by strong server-side logic**: if the server actively **validates game state changes** (position, health, inventory), most client hacks (teleport, godmode, give money) simply won't "stick" – the server will reject the change [34] [8] . Still, client-side detection is needed for things like aimbot or ESP which don't always directly violate server rules but can be inferred by behavior.

Advanced and Emerging Exploits

In addition to the common cheats above, FiveM servers face **more sophisticated attacks** that require advanced countermeasures:

- **Lua Script Injection (Executors):** Since Lua is the main scripting language, hackers have developed external tools called *executors* that can inject and run arbitrary Lua code in the client's context. This means a cheater can execute any game native or trigger any event call as if it were part of a script [35] . This is extremely powerful (essentially giving the cheater a live console to the game). *Detection:* Catching injected Lua is very hard because from the server's perspective, it's just the client doing things (e.g. if they call an event via injected code, it looks like any normal event trigger). Some known approaches: **Global environment checks** – many Lua menus leave traces like global variables or certain function hooks. Anti-cheats like "Valkyrie" tried checking the `_G` global table for known cheat variables, but cheat makers quickly learned to nil out or rename those, making this check unreliable [36] [25] . Another approach is **function integrity** – if you can, detect if critical native functions have been overridden by Lua (some executors monkey-patch natives). FiveM doesn't provide an official way to list all loaded client scripts or check their source, so it's cat-and-mouse. One promising angle is to use a **client-side watchdog**: include a script that periodically verifies the anti-cheat's own Lua files haven't been altered and perhaps checks for the presence of foreign resources. Even so, determined injectors can bypass many such measures. This is why **server-side validation is crucial**: even if a cheater runs Lua code to, say, give themselves a weapon, the server can still decide "player X isn't supposed to have that gun now" and remove or block it. In summary, Lua injection is a known vector and cannot be completely prevented by scripts alone – it often requires a combination of heuristic monitoring and robust server rules to minimize impact [35] .

- **NUI/CEF Exploits:** The in-game browser (CEF) that displays UIs can be exploited if a server's UI code isn't secure. For instance, an attacker might find an XSS (cross-site scripting) vulnerability in a menu (perhaps a poorly sanitized player name in a scoreboard UI) and inject malicious JS [14] [12] . That JS could call FiveM's `window.invokeNative` to execute game natives or use the **NUI callback** system to trigger client events. Essentially, the cheater might hijack a legitimate UI to run cheat

code. *Mitigation:* This is more on the **secure coding practices** side – ensure any UI HTML/JS is sanitized and cannot be easily hijacked [14] [15]. NexusGuard can include a **NUI monitor** that checks if any unexpected URLs are loaded or if known dangerous JS calls are made from UI. But given the sandbox, there's limited reach – likely, this is a lower priority unless specific vulnerabilities are known. At minimum, document guidelines for server developers to not expose sensitive natives to UI and to validate inputs in any UI forms.

- **Memory Manipulation & External Tools:** Hackers can also use external memory editors or inject DLLs to directly modify the game (just as in single-player modding). This can enable any cheat from godmode to spawning objects, even injecting custom code. FiveM itself doesn't come with a kernel-level anti-cheat like Battleye, so some external cheats can operate without immediate detection. *Detection:* Without a kernel driver, detection is mostly **behavioral or signature-based**. For example, an anti-cheat could run in an external program and scan GTA V's memory for known cheat code patterns or monitor if certain game functions are hooked [37]. However, doing this as a resource is not possible; it'd require a separate anti-cheat client program (which is beyond the scope of a script-based solution and edges into what paid anti-cheats do). **NexusGuard's approach** should be to catch the *effects* of memory hacks (like impossible stats changes) and possibly use **native function hooking** in script to detect anomalies. For instance, intercept calls to natives that **should only be called by legitimate scripts** – e.g., if a non-admin player tries to call `AddExplosion` via an injected thread, a client-side hook could detect that and report it [4] [38]. FiveM unfortunately doesn't let scripts truly *block* a native call by another script, but you can sometimes override or preempt things with higher resource priority. In any case, heavy memory cheats often manifest in obvious ways (explosions, teleport, etc.) which server authority will handle. Truly stealthy memory hacks (like reading data for ESP) are nearly impossible to detect without a specialized anti-cheat client. Recognizing that limitation, we rely on catching consequences (like the ESP user reacting unrealistically, or unusual memory access patterns if we have a way to check those).

- **Network Protocol Exploits:** FiveM uses networking to sync players; an attacker could try to exploit the network messages themselves. For example, sending malformed packets or trying to replay encrypted messages. FiveM's network is fairly robust and most cheats don't go this route (since it's easier to just exploit the game directly), but we consider it. *Mitigation:* Ensure **encryption and integrity** of any custom data NexusGuard sends. If analyzing network traffic, look for anomalies like a client sending data that doesn't match their declared state (though this blends into server validation). Some advanced anti-cheats monitor network timing or volume for patterns of tool use (like a cheat might spam certain packet types); this is very advanced and typically not needed for FiveM as long as server logic is strong [39]. The main point: by enforcing proper server checks, even if a malicious client crafts a custom packet, it shouldn't grant them anything unless the server explicitly allows that action.

- **Anti-Anti-Cheat Measures:** We should also note that cheat developers often implement features to disable or evade anti-cheats. For example, a cheat might search for common anti-cheat resource names or behaviors and disable them, or hook functions like `TriggerServerEvent` to filter out the anti-cheat's calls. They might also use **anti-debugging and obfuscation** so that if you tried to attach a debugger to the game to see the cheat, it hides itself [40] [41]. NexusGuard, being proprietary, can maintain some secrecy – avoid obvious naming (don't literally call it "anticheat.lua" in the resource; use a less conspicuous name so basic cheats might not target it). Also, employing **code obfuscation** on the clientside scripts or C# DLL can make it harder for cheat makers to reverse-

engineer NexusGuard [42]. Techniques include minifying/encrypting Lua code, using tools to obfuscate .NET assemblies, etc. Be aware though: heavy obfuscation can make debugging your own code challenging and may affect performance slightly. It's a balance, but for a truly robust solution, some obfuscation/anti-tamper is recommended so that the anti-cheat itself isn't easily turned off by the next mod menu update [42] [43].

In summary, FiveM cheat threats range from simple (e.g. one-click godmode from a public mod menu) to highly advanced (custom executors and memory hacks). **NexusGuard must employ a defense in depth: multiple detection layers to cover this spectrum [44].** Now that we have enumerated the threats, we can discuss how to counter them through smart anti-cheat design.

Core Anti-Cheat Techniques and Best Practices

Building the "most robust" anti-cheat requires combining **server-side authoritative design** with targeted client-side detection and a smart understanding of FiveM's internals. Below are the core techniques NexusGuard will employ:

Server-Authoritative Design & Validation

Making the **server the source of truth** is the single most effective way to prevent many cheats. In a server-authoritative model, the server does not blindly accept any action from the client; instead, it verifies that action against game rules and either approves or rejects it [34]. This philosophy can nullify whole classes of cheats because even if a hacker alters something on their client, it won't have an effect unless the server allows it.

Key areas to implement server-side validation:
- **Movement & Position:** The server should track player positions and movements. By checking the difference between updates, the server can detect teleports or speed hacks and correct them [29]. For example, impose a maximum distance a player can travel within a given time (based on known vehicle speeds). If a player exceeds that, either clamp their position or snap them back and flag them. Also check z-coordinate changes: if someone goes from ground level to a high altitude in one tick (and they don't have a vehicle or parachute that explains it), that's invalid. Modern FiveM servers with OneSync can do more rigorous entity control server-side. Use the `entityCreating` event to intercept any entity spawn (vehicle, object, ped) – the server can decide if it's allowed or not [4] [45]. For instance, prevent a player from spawning a tank if they're not permitted, etc.

- **Combat & Damage:** Validate hits and kills. If your server uses custom damage events (FiveM has `weaponDamageEvent` and `gameEventTriggered` for some combat events), leverage those. Ensure that if player A "killed" player B, player B truly had low health and the weapon was capable, etc. You might implement a system where the server roughly simulates or at least checks that damage is plausible – e.g., no killing a fully armored player with one pistol shot unless it's a headshot. For aimbot detection server-side, maintain stats on shot accuracy, headshot rates, etc., and flag outliers [23]. While the client anti-cheat might directly observe suspicious aim movements, the server can still gather stats to identify blatant aimbots.

- **Inventory & Economy:** *Never let the client decide any economic outcome.* Money, items, experience – all should be granted by the server. If the client says "I earned $100", the server should calculate that independently. Validate any `TriggerServerEvent` that relates to giving rewards [8]. For example, if a client triggers `giveMoney` with an amount, check server-side: are they allowed that money? Did they perform the task, and is the amount within expected bounds? If not, reject or adjust it. This prevents money exploits and duplication glitches. Many cheats will try to spam money-giving events – implement rate limits on how often a player can trigger certain actions (maybe you only allow one "job complete" event per minute, etc.). **Transaction logging** is useful: keep a log of all money/item gains with reasons, so if economy irregularities appear, you can trace them.

- **Health & State:** The server can periodically enforce health/armor limits. For instance, if your game mode doesn't allow full armor unless a player buys armor, but suddenly a player's armor is maxed with no record of that event, set it back to previous value or zero. When a player is hit, ensure their health drops appropriately. If `GetEntityHealth(ped)` on server shows they should be dead but they're still alive, you can force kill them or flag godmode [30]. Also check for impossible states – e.g., if a player's ped is in ragdoll or down state but health is not decreasing as it should, or they never ragdoll at all from big explosions, these are suspicious.

- **Event Sanitization:** For every event handler on the server, **validate inputs**. If an event passes an ID of an entity or player, ensure that entity actually exists and the source player has the rights to interact with it. If an event is supposed to only be fired by the framework, consider implementing an **identifier or token**. One advanced method: include a hidden parameter or a dynamic token that the client scripts know but a random cheater calling the event wouldn't. For example, when the server starts, it generates a random token for "giveMoneyEvent" and the client script pulls that from the server. The client then calls `TriggerServerEvent("giveMoneyEvent", token, amount)`. If the token doesn't match, the server ignores it – a basic cheater likely wouldn't know the token. There are community implementations of this idea (like **Salty Tokenizer**) which add unguessable tokens to events [46] [47]. According to one developer, this makes it so even if a hacker dumps your client scripts, they can't easily replay the events because the token isn't present or is obfuscated [46]. NexusGuard can integrate a **security token system** like this to protect critical server events from unauthorized calls (with tokens rotating or unique per session to avoid leaks). Always remember the guiding rule: **never trust the client** for anything game-critical [9].

Implementing a server-authoritative approach does increase the server's workload and complexity (the server essentially double-checks everything) [48]. But the trade-off is worth it for security. It can prevent cheats *before* they actually affect other players, essentially eliminating the need to "detect" some cheats because they simply cannot succeed. For example, a teleport hack that tries to move a player will immediately be undone by the server's authority on position [49]. This spares you from having to ban the player purely on suspicion – the cheat just doesn't work effectively, and you can then warn/ban them calmly knowing they didn't ruin gameplay in the meantime.

Event Monitoring and Exploit Prevention

As discussed, FiveM events are a common vehicle for cheats, so robust event monitoring is a must:

- **Logging and Alerting:** NexusGuard should log every significant `TriggerServerEvent` call with the source, event name, and perhaps parameters (be careful not to log sensitive info though). By

analyzing logs, you can find patterns (e.g., a certain event being spammed) and retroactively catch exploits. Real-time, the anti-cheat can flag unusual event usage – for example, if a single player triggers an event 100 times in a minute when normally it triggers maybe 5 times, raise an alert or kick them (this could catch rapid-fire exploits, spamming give-money, etc.).

- **Event Blacklist/Honeypots:** Maintain a list of known malicious event names that should *never* be called by normal players. For example, events that give admin powers or known from cheats ("redEngine:whatever" was a known cheat trigger in the past, etc.). If those are triggered, instantly ban or kick. Also use **honeypot events** as mentioned: register dummy handlers for events commonly used by cheats that your server doesn't use [27]. Anyone calling them is certainly using a generic cheat trying everything. This is a low-effort, high-reward tactic. Just be mindful to update these if FiveM introduces new native events (to avoid false positives as noted in the forum post) [50]. One can also implement a "ban on any unknown event" mechanism, but that's risky due to updates. A safer approach is curated honeypots.

- **Secure Event Proxy:** This is an advanced technique where you **don't call FiveM events directly for sensitive actions**, but route them through an anti-cheat proxy. For example, instead of `TriggerServerEvent("esx_pizza:pay", amount)` from client, the client script calls `NexusGuard.TriggerSecure("esx_pizza:pay", amount)`. NexusGuard's client component could append a token and call a different server event, and on the server NexusGuard verifies and then internally calls the real handler. This way, if a hacker tries to call `esx_pizza:pay` directly, it won't have the right token or context and gets blocked [46]. **GoblinAC** essentially did this by encrypting event payloads and requiring the events be sent through their system [16]. It's a bit of work to integrate (especially across all resources), but one can focus on the *most critical events* (like money giving, job rewards, etc.). If making it generic, perhaps provide a wrapper export like `exports.NexusGuard:SecureEvent(eventName, params...)` that resource authors can use to secure their events without worrying about the details.

- **Rate Limiting and Cooldowns:** For events that can legitimately be called but not too frequently (like a paycheck event, or a healing item use), implement server-side rate limits per player. If a player triggers something too fast, ignore subsequent calls or apply a cooldown. Cheaters often maximize their advantage by spamming actions faster than humanly possible, so this catches those scenarios while probably not affecting normal players (who can't physically click that fast). Example: a cheat might attempt to heal to full health by triggering a "useMedkit" event 50 times in a second. A rate limiter would drop those extra calls and flag it.

- **Parameter Validation:** Always scrutinize data coming from clients. If an event is supposed to send an integer 1-100 and someone sends 1000000 or a string, clearly that's malicious or a bug. Check data types and ranges. If an event expects the client's own player ID, check that the source matches the ID provided. If an event is to give an item, check that item is allowed. This overlaps with server-authoritative checks, but it's worth emphasizing from the event perspective: *all inputs are untrusted*. Use defensive programming: assume any client input could be an attack.

By comprehensively monitoring events, NexusGuard effectively **shuts down the most common FiveM exploits (trigger abuse)**. Many cheat menus rely on abusing poorly secured events, so this layer is critical. It should work *regardless of framework* – for example, if a server uses ESX, there's a known set of events to guard; if using QBCore, a different set; if custom, maybe the server dev can configure which events are

sensitive. Provide sensible defaults (like any event named `giveMoney`, `Paycheck`, `AddItem`, etc. gets extra scrutiny).

Player State Monitoring (Client-Side Detection)

While the server keeps the authoritative state, client-side scripts can be used to **double-check and detect anomalies in real time**. NexusGuard can have a client component that runs on every player's machine and watches for certain red flags locally:

- **Critical Native Function Hooks:** The client anti-cheat can hook or overwrite certain game natives in the Lua runtime to detect misuse. For example, one could override `CreateVehicle` on the client: if a script (possibly an injected cheat) calls it to spawn a vehicle, your override function could log that call (with what model and who called it) or prevent it if the player isn't allowed. Similar hooks could be set for `GiveWeaponToPed`, `SetEntityCoords`, `SetPlayerInvincible`, etc. – all functions that no normal resource should call arbitrarily for a regular player [4] [5]. A caution: many legitimate resources also create vehicles or give weapons (e.g. a garage script spawning your car). So the hook can't just block them all – it needs context. Perhaps maintain a whitelist of resources or scenarios allowed to call those natives. Or integrate with permissions: e.g., allow `SetPlayerInvincible` only if the player is an admin (maybe used for admin spectate mode). Logging is useful: if a cheat does something weird, you have a record. Note that a sufficiently advanced cheat could remove or bypass your hooks (since if they have full Lua access, they could manipulate the runtime). That's why this is a layer, not standalone. But hooking natives raises the bar for cheats – an inexperienced cheat might accidentally trigger your hook and get caught.

- **Suspicious Resource Detection:** The client can check the list of running resources and detect if any unapproved resource is present. For instance, if a mod menu injects itself as a resource (some cheat executors do this to utilize FiveM's scripting), you might see a resource with a weird name running. If NexusGuard sees a resource not in the authorized list (you can send the list of legitimate resources from server to client and compare), it can alert or even attempt to stop it. FiveM doesn't allow a script to stop another resource unless it has permission, but just knowing it's there is enough to flag the player. Similarly, detect if any known cheat process is running on the client machine (this is harder from within FiveM; it would require something like reading process list which normal scripts can't do for security reasons). So focus on in-game indicators.

- **Exploit Sensors:** These are like mini-checks on the client that look for signs of hacks. For example, a **velocity sensor**: if the client script notices your own player is moving absurdly fast (beyond a threshold), it can report "speed hack suspected" to the server. Or a **ped attributes sensor**: periodically check if `GetPlayerInvincible(player)` is true when it shouldn't be, or if your weapon damage multiplier is unnaturally high, etc., and report if so [31] [51]. The idea is that the client is closer to the action and can check some native states more frequently. But be careful – a cheat running on the client could fake those responses or disable the anti-cheat client script. That's why anything detected client-side should ideally be verified server-side or used as a secondary measure. Think of client anti-cheat as an alarm system that can ring a bell, while the server is the lock on the door. Both together are better.

- **UI/CEF Monitoring:** As mentioned, a JS script in the NUI context could monitor if any new unexpected UI gets opened (like if a mod menu HTML appears). It could also intercept messages to

NUI callbacks. For example, if the cheat tries to use `SendNUIMessage` or triggers a NUI callback event in an unusual way, the anti-cheat could notice. Since NUI is often used by mod menus for fancy interfaces, one trick is to create an invisible iframe or element and see if cheat scripts inject content into it. This is very experimental – not guaranteed – but it's part of being comprehensive. At minimum, ensure the anti-cheat's own UI (if any) is secure from injection [15] .

- **Local Behavior Analysis:** The client can also gather data on the player's inputs and aim that the server might not see with full fidelity (server sees results, but client can measure the exact timing). For instance, measure how quickly a player moves crosshair from one target to another; aimbots might have a distinct timing. This data can be sent to the server for analysis (we discuss ML/heuristics later). But note: trusting the client to send honest data about itself is tricky – a cheat could manipulate those metrics. Still, gross anomalies might slip through.

In summary, the client-side NexusGuard module is like a **sentry inside the client** that tries to catch cheats in the act. It will never be foolproof (a smart cheat can silence it), but it can catch less sophisticated cheaters or provide additional evidence. Moreover, if the anti-cheat client is obfuscated and updated regularly, cheat makers have to put effort into disabling it – which slows them down and may deter some.

Integration and Admin Tools

A complete anti-cheat should also consider **how it interacts with server admins and players**. Here are some best practices for integration:

- **Alerts and Logging:** When a violation is detected, NexusGuard should log it and optionally alert online admins in-game (or via Discord/webhook). For example, "NexusGuard: Player123 flagged for possible aimbot (headshot rate 95%)" or "Player456 triggered honeypot event esx_admin:banPlayer – likely using mod menu". These alerts help admins intervene. Provide configurable settings for what actions to auto-take (ban, kick, just log). Some server owners may prefer auto-ban on obvious hacks (like honeypot trigger), but only warn on statistical suspicions to avoid false bans. A Discord webhook integration is useful so that all cheat incidents are recorded externally with time, player, and details.

- **False Positive Reduction:** Design the anti-cheat rules to **minimize false positives**, as nothing annoys players more than being wrongly punished. Use context-aware checks: e.g., if a player's position jumps wildly but they just respawned or teleported via a legitimate script (perhaps a spectator teleport), don't flag that. Or when validating health, consider if there are safe zones or scripted invincibility phases (some games have godmode on spawn for a few seconds, etc.). NexusGuard can incorporate context from the server's game mode – perhaps allow an API for the game mode to inform the anti-cheat of special conditions (like "this player is in admin mode" or "this race event temporarily allows super speed"). Also implement **grace thresholds**: e.g., instead of banning immediately on one suspicious event, maybe mark the player and watch closely. If multiple different checks trip, then take action. This layered confirmation approach helps ensure you only punish real cheaters. The optimization prompt specifically mentioned avoiding false positives by considering state like falling or respawning for speed/teleport checks – implement exactly that (only flag teleport if not currently in a known teleport context, etc.) [52] .

- **Configurability:** Provide a config file where server owners can turn specific detections on/off and adjust sensitivity. For instance, they might disable a certain check if it conflicts with a custom script, or lower the sensitivity of aimbot detection if it was catching legit skilled players. Also allow configuring which identifiers to use for bans (some servers might want Steam ID, others license). Given the user wanted to avoid heavy database use, NexusGuard can simply use FiveM's built-in ban system or a lightweight JSON/YAML file to record bans, using multiple IDs (Rockstar license, Steam, Discord, IP) to catch evaders. **Ban evasion detection** means if a banned user comes back on a new Steam account but the same rockstar license or IP, the anti-cheat should recognize it. FiveM's `playerConnecting` event provides identifiers which can be cross-checked against a ban list. Storing a hash of their identifiers or using services like **Global Ban** lists (if available) can bolster this. But since we want proprietary, perhaps maintain our own list. Make sure to include hardware ID if FiveM offers one (FiveM has a concept of "GUID" or can get some machine ID, though not always reliable). Using a combination of identifiers makes it harder for a cheater to just make a new account and return. This can be done without a heavy database – a flat file or a web API call to a central NexusGuard server could suffice, depending on design.

- **Performance Considerations:** Anti-cheat checks can be CPU-intensive if done carelessly (e.g., scanning all players every frame for position changes, or memory scans). Optimize by running checks in **intervals** and spreading them out. For example, you don't need to check every player's coordinates every tick (which is 0.05s in FiveM). Maybe check 10 players per tick in a round-robin or do it twice a second for each player – that still catches teleports quickly without constant load. Use **adaptive checks**: if a player is already flagged or suspicious, you can put them under a microscope (check them more often) while relaxing checks for everyone else [53]. Conversely, if server population is high, ensure the loops scale (maybe skip some less critical checks or lower frequency dynamically). Lua is fast with LuaJIT, but still, don't do heavy math or huge loops unnecessarily on the server thread. Offload where possible: e.g., heavy log analysis or ML can be done by a Node.js module on a separate thread or even off-server.

- **Updates and Evolving Threats:** The cheat ecosystem evolves rapidly. New mod menus, new bypass techniques will come. Plan for **continuous updates** to NexusGuard. A modular design (breaking the code into components like `core.lua`, `detections.lua`, etc. as mentioned in the prompt) makes it easier to update specific parts [54]. Encourage community or admin feedback when a new cheat slips through so you can patch it. It's wise to implement a version check – e.g., NexusGuard can ping a version server or check a GitHub repo to see if an update is available, so server owners know to update (since it's proprietary, distribution might be controlled, but within the team at least keep track).

- **Ethical and Privacy Considerations:** Be mindful that some detection techniques (like screenshotting the client, or scanning processes) could raise privacy concerns. Always disclose to server owners (and by extension players) what the anti-cheat is doing. For example, if you implement a feature to take screenshots of a suspected cheater's game (some anti-cheats do that to catch visual menu), make sure this is known and maybe opt-in because it's sensitive. Focus on methods that impact only the cheater's experience (like scanning memory or monitoring their behavior) rather than anything that could leak personal info.

Advanced Detection Techniques and Enhancements

To truly build the "next-generation" anti-cheat, NexusGuard can incorporate advanced methodologies beyond the basics. These are more complex to implement but can significantly improve cheat detection:

- **Behavioral Analysis with Machine Learning:** Instead of only using fixed rules, NexusGuard can analyze player behavior data over time to spot subtle cheat indicators. For example, gather data on each player's aiming (reaction time, tracking smoothness), movement patterns, and decision-making [55]. Using machine learning, you could train a model on what "legit" gameplay looks like versus "cheating" (like aimbot usage). Metrics like **headshot ratio, kill-death patterns, hit accuracy, average movement speed** can feed into a model [56]. If a player suddenly exhibits superhuman consistency or impossible stats, the model flags them. This can catch aimbots or wallhackers who try to look semi-legit but still have patterns non-human players don't. Keep in mind: ML requires data – possibly from recorded sessions or known cheaters to train on. It also must be tuned to avoid false positives (a very skilled player might look suspiciously good; the model needs to not ban pro players!). A practical approach is use ML flags as supportive evidence rather than automatic bans at first [56] [57]. For example, "Player X is 99% likely an aimbot by model – mark them as high risk" and then admins can spectate or other detections can confirm.

- **Heuristic Code/Memory Analysis:** This refers to detecting cheats by their technical behavior rather than specific signatures. For instance, if a foreign module (DLL) is loaded into GTA V's process that isn't normally there, that's suspect. Or if certain Windows API calls (like ones to manipulate memory or input) are being made at a high rate by the game process, that could indicate a cheat injecting itself [58]. Implementing this in FiveM likely means an external watchdog or a very low-level client module. Some things can be done within limitations: for example, check the game's memory for anomalies if you had a C++ module. Without going that far, perhaps use **FiveM's native functions** to detect common signs: there is a native for checking if a trainer is present (though cheat devs circumvent those easily). Also, FiveM has built-in global state that sometimes reveals if something was network bypassed (OneSync has some entity ownership concept that can be validated). Another heuristic: if a client's game triggers heavy network traffic patterns not typical for their actions (like sending way more packets than others, or using high bandwidth constantly), that could hint at an exploit [39]. These are quite advanced and might not be initially in scope, but listing them for completeness. NexusGuard's architecture should leave room to plug in such modules later.

- **Code Obfuscation & Anti-Tamper:** We touched on this in the exploits section. It's worth planning from the get-go how to protect NexusGuard's own code. If using C#, consider using an obfuscator on the compiled DLL (there are free ones and paid ones that rename methods, encrypt IL, etc.). For Lua, you can use Lua obfuscation tools which output a scrambled chunk that functions the same. Even a simple measure like removing readable identifiers and comments helps. Additionally, implement **integrity checks**: for example, the client script can have a known hash of itself and periodically verify none of its files were altered (though if a cheat can alter the script, it could alter the checker too – but if the cheat only hooks certain things, an internal integrity check might catch if the file or memory was changed) [43]. Anti-debug techniques can be used – e.g., detect if the game is running under a known debugger (some mod menus rely on being attached as a debug process). If detected, the anti-cheat can respond by e.g. crashing the game or disabling certain features (to prevent the cheat from easily analyzing it) [40]. Keep these measures subtle to avoid affecting normal players (who usually won't be debugging!). The goal is to **slow down cheat developers**. If they have to

spend days reverse-engineering your anti-cheat updates, that's time players are safer. Just be aware: nothing client-side can be 100% tamper-proof, but making it **difficult and annoying to crack** is a deterrent [59].

- **Memory Integrity Monitoring:** If at some point you incorporate a C++ module or if FiveM exposes certain low-level APIs, verifying game memory can catch cheats. For example, reading certain game variables to see if they were changed unexpectedly (like if health is locked at a value in memory while in game it shouldn't be). Another example: check if the function bytes of critical natives are unchanged (some cheats hook game functions). This is quite low-level (like scanning GTA V's memory addresses). Typically only a dedicated anti-cheat program or driver can do this, and doing it from a script context is not possible. So it might be beyond the initial scope. But as a concept: **memory scanning** for known cheat signatures and verifying module lists can catch injected DLLs [37]. For instance, if you know of a common cheat DLL name or certain byte patterns, a memory scan could find it. Keep this in mind if developing a separate AC client in the future.

- **Network Analysis:** Mentioned earlier, but advanced usage could be analyzing encrypted traffic timing or building a profile of normal network messages. For instance, if a cheat uses an exploit that requires sending a certain unusual packet, maybe you'd spot it. However, FiveM's network is mostly not your domain to analyze deeply (it's internal and encrypted). The best you can do is track events and states at the server script level, which we already cover with validations.

- **Global/Community Bans:** While NexusGuard is proprietary and standalone, you could consider integrating a community ban database or sharing data across servers (if multiple servers deploy NexusGuard). For example, if a cheater is caught and banned on Server A with evidence, that info (player identifiers, what they did) could be submitted to a central system. Then Server B running NexusGuard could pre-emptively ban or watch that player if they join, because they were known cheater. This helps catch **ban evaders** who server-hop. Of course, this requires server owners opt-in and trust the shared data. Also, legally, one must be careful with sharing identifiers (like Steam IDs) and ensure not to false-flag people. But many communities do share ban lists to keep known trolls out. Since the user specifically said a proprietary AC, we won't lean on external lists by default, but we can mention that **global ban integration** is an optional advanced feature.

- **Usability for Server Owners:** Lastly, an often overlooked aspect: making the anti-cheat easy to use and tune. Provide good documentation (the dataset/guide we're building is a start) so that server devs know how to configure NexusGuard for their specific server (e.g. toggling compatibility with popular frameworks, adjusting thresholds). Possibly include an **in-game menu or command** for admins to dynamically toggle the AC (like turning it to "lenient" or "strict" mode during events, or a command to manually trigger a scan on a suspicious player). Also, if false positives occur, have a way for admins to mark a player as safe temporarily. These quality-of-life features make admins more likely to keep the anti-cheat enabled rather than turning it off due to frustration.

Conclusion

Developing a robust FiveM anti-cheat like NexusGuard is a challenging but achievable task, requiring a **multi-faceted approach**. By combining *server-authoritative design* (never trusting the client's actions without verification) [9], *comprehensive monitoring* of events and player state, and *multi-language modules* for performance and security, we can cover the most common and devastating cheats. We've outlined how

to handle everything from simple godmode toggles to advanced injection exploits, and emphasized the need for **layered defenses** (no single technique will catch all cheats, but multiple layers dramatically increase security) [44].

NexusGuard should work on any FiveM server out-of-the-box, regardless of permission systems or frameworks, while allowing configuration to mesh with each server's unique setup. It leverages Lua for seamless integration, C# and JS for specialized tasks, and leaves room for expansion into cutting-edge techniques like behavioral analysis and memory integrity as needed. Importantly, it's a proprietary system – meaning the techniques can be kept private and obfuscated, making it harder for cheat developers to reverse-engineer and bypass [42] [43].

By following this guide, the AI (and by extension the development team) should gain the necessary context to write code that is compatible with FiveM's environment and *innovative* in its approach to anti-cheat. The result, NexusGuard, will be a **comprehensive anti-cheat framework** that not only catches the typical mod menu kiddies but also provides a foundation to combat more sophisticated threats. The key takeaways: **validate everything on the server, monitor everything on the client, assume attackers will adapt, and build your system to adapt in turn**. With diligent implementation and continuous improvement, NexusGuard can significantly elevate the security and fairness of any FiveM server [44] [34]. Good luck, and happy (secure) coding!

Sources: The insights and strategies above were compiled from a combination of FiveM documentation, community best practices, and NexusGuard's internal research on FiveM cheats and countermeasures [60] [34] [8]. This ensures that the guide stays up-to-date with current threats and solutions in FiveM server development and anti-cheat design.

---

[1] [2] [4] [5] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [28] [29] [30] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [48] [49] [55] [56] [57] [58] [59] [60] research.txt
https://github.com/Strategizing/NexusGuard/blob/b666aa634f69fc926bc47e03a348d7e058e562ca/NexusGuard/research.txt

[3] Master Essential Coding Skills for Custom Servers - FiveM Store
https://fivem-store.com/fivem-scripting-language-master-essential-coding-skills-for-custom-servers/

[6] [7] natives_wrapper.md
https://github.com/Strategizing/NexusGuard/blob/b666aa634f69fc926bc47e03a348d7e058e562ca/NexusGuard/docs/natives_wrapper.md

[8] [9] [27] [50] How hackers can exploit your servers and what to do about it - Server Discussion - Cfx.re Community
https://forum.cfx.re/t/how-hackers-can-exploit-your-servers-and-what-to-do-about-it/702213

[10] Permissions :: vMenu Documentation
https://docs.vespura.com/vmenu/permissions-ref/

[31] [32] [51] version_compat.lua
https://github.com/Strategizing/NexusGuard/blob/b666aa634f69fc926bc47e03a348d7e058e562ca/NexusGuard/shared/version_compat.lua

[46] [47] [Release][DEV] Server Event Security Tokens - Anticheat - Page 2 - FiveM Releases - Cfx.re Community
https://forum.cfx.re/t/release-dev-server-event-security-tokens-anticheat/139189?page=2

52 53 54 optimization_prompt.txt

https://github.com/Strategizing/NexusGuard/blob/b666aa634f69fc926bc47e03a348d7e058e562ca/NexusGuard/optimization_prompt.txt