Roadmap for Implementing an Anti-Cheat System in MTA:SA

1. Anti-Trigger Protections

To prevent cheaters from firing game events illegitimately, implement strict server-side validation for all events:

- **Define Event Access:** Determine which events should be callable by clients. For any sensitive server-only events (e.g. admin actions), disable remote triggering when registering them. Use `addEvent(eventName, false)` to make an event server-side only, so clients cannot call it at all [1]. For example, `addEvent("onServerSideOnlyEvent", false)` ensures no client can trigger that event externally [1]. (MTA's documentation emphasizes that *"It is very important to disable remote triggering ability in addEvent, to prevent calling server-side only events from client-side."* [2].)

- **Validate Event Origins:** For events that *are* meant to be triggered by clients (allowRemote set to true), always verify the caller. In each server-side event handler, use the implicit global variable `client` provided by MTA instead of trusting any event parameters like `source`. The `client` variable reliably identifies the player who triggered the event (it will be `nil` if the server triggered it). **Never trust** `source` or arguments for client identity, as they can be faked [3]. For example:

```lua
function onPlayerBuyItem(itemID)
    if not client then return false end  -- ensure this was triggered by a player
    local player = client  -- the real source of the event
    -- proceed with handling purchase for 'player'
end
addEvent("onPlayerBuyItem", true)
addEventHandler("onPlayerBuyItem", root, onPlayerBuyItem)
```

  In the above, `not client` guard ensures the event is ignored if it didn't come from a client. Always use `client` for security checks (e.g. permissions) instead of an event argument, since *"all parameters including source can be faked and should not be trusted. Global variable client can be trusted."* [3]. This prevents cheaters from spoofing another player or an admin as the event source.

- **Validate Event Data:** Rigorously check any data sent by the client through events. Define expected types, ranges, and relations for event parameters and enforce them in the handler. For example, if an event passes an element (like a vehicle or player) or a player name/ID, verify that it's a valid element and that it **belongs to the calling player** if applicable. You can cross-verify that an element parameter equals the `client` (for player-specific events) and is of the correct type. In fact, you may implement a general validation table for events. For instance: an anti-cheat could specify that an event's source element must equal the client and be a player (or allowed element type) [4]. If any check fails (type mismatch, value out of allowed range, etc.), immediately reject the event and treat it as a potential exploit. This ensures, for example, that a cheater can't trigger an event giving money to another player or spoof themselves as an admin – the server will catch that the `source` or data isn't consistent with the real `client` [4].

- **Permission Checks for Sensitive Events:** For admin or high-privilege operations triggered by clients, add explicit permission checks. Verify the `client`'s ACL group or privileges before executing the event logic [2]. For example, if an event is supposed to ban a player, ensure `client` has admin rights (`isObjectInACLGroup` or `hasObjectPermissionTo` for the ban function) before proceeding [2]. If the player lacks permission, cancel the action and log the attempt. This layered check prevents ordinary players (or imposters) from executing admin-only functionality.

- **Use Proper Event Design:** Wherever possible, design events such that clients don't need to send sensitive triggers at all. For instance, rather than trusting the client to initiate an "awardMoney" event, have the client send a request and let the server decide the amount and call the actual function. The less authority the client has, the harder it is to cheat. Keep critical game logic (health, currency, inventory changes, etc.) on the server side and only accept minimal input from clients (which you then validate).

- **Rate Limiting & Spam Protection:** Cheaters might abuse even legitimate events by spamming them (e.g. sending many triggers to cause lag or exploit timing). Implement rate limits or cooldowns on client-triggered events. For example, if an event is to open a GUI or perform an action, ignore or throttle repeated calls from the same client within a short time frame. This can be done via timers or counters server-side. While not directly about unauthorized triggers, it's an anti-cheat measure to prevent macro or bot abuse [5].

- **Logging and Fail-Safe:** For any event that fails these checks (e.g. wrong `client`, invalid data), take action. At minimum, log the incident on the server (with details like which player (serial or name) and what data was suspicious). You can use `outputDebugString` (which will log to the server console/log file) or write to a custom log file. Ideally, notify an admin or the anti-cheat system. Optionally, immediately kick or ban the player if the violation is severe (see section 7 on detection & logging for more). The key is to *never let an unauthorized or malformed event slip through silently*. Even non-critical events should be checked, as any unchecked path can be a vector for cheaters.

2. Anti-Dump Protections

Preventing players or external tools from **dumping your scripts or assets** (i.e. extracting them from the client's downloaded files or memory) is crucial for protecting your intellectual property and preventing reverse-engineering of your anti-cheat. MTA provides some built-in options to reduce dumping, and you can add custom measures:

- **Disable Client Caching of Code:** By default, when a client downloads a resource, the files (including `.lua` scripts) are stored in the client's cache directory. To hinder players from grabbing your Lua files, mark sensitive client scripts as **non-cacheable** in the resource's `meta.xml`. Use the `cache="false"` attribute on client and shared script entries. For example:

```
<script src="client.lua" type="client" cache="false" />
```

This tells MTA to load `client.lua` for the client *but not save it to disk* [6]. With `cache=false`, the script is sent over the network each time and only kept in memory during runtime. Most casual users will then **not have a readable copy** of your script lying around in their MTA cache. (Note: this has no effect on server-side scripts, as those aren't sent to clients anyway. Also, if a player had previously downloaded the resource with caching on, MTA won't retroactively delete it from their disk [7]. In practice, you should set `cache="false"` from the start or change the resource name to force a fresh download if you enable this later.)

- **Lua Script Compilation (Bytecode):** Even with no caching, a determined attacker could use memory dumping tools. So, **obfuscate and compile** your Lua scripts before release. MTA supports compiled scripts (often .luac files) that run on the client. Use the official MTA Lua compiler (available via the luac.mtasa.com API or the provided `luac_mta` tool) to compile your `.lua` files into bytecode. Enable the highest obfuscation level (level 3) for maximum protection [8]. This will mangle variable names and make the bytecode harder to decompile. The compiled output can be used in place of raw source. For example, compile `client.lua` into `client.luac` with obfuscation; then ship the `client.luac` (or rename it) and reference that in the meta. Compiled bytecode is not human-readable, and while not impossible to reverse, it's **significantly harder** to steal or tamper with your logic. According to MTA's documentation, using the Luac compiler with obfuscation makes things "slightly harder" for malicious actors [9]. In practice, a well-obfuscated bytecode (especially combined with encryption, next) will deter all but the most expert hackers. *(MTA's Luac compiler with obfuscation level 3 has been noted to produce code that is extremely difficult to decompile [8].)*

- **Custom Script Encryption:** For an additional layer, encrypt your compiled scripts. Simply put, even if someone manages to dump the compiled bytecode from memory or disk, it will be encrypted gibberish without the decryption key. You can use Lua-based encryption provided by MTA (such as the Tiny Encryption Algorithm, **TEA**, or AES) to protect the files. MTA offers functions like `teaEncode`/`teaDecode` and a generic `encodeString`/`decodeString` (which supports `"tea"` and `"aes128"` algorithms) for this purpose [10] [11]. The strategy is: **before distribution**, take your compiled script file and run it through an encryption function with a secret key. This produces an encrypted file (for example, `client.enc`) that you include in the resource instead of the plain `.luac`. (We will design a loader for it in the next section.) By doing this, even if a player somehow accesses the file, they cannot decode it without the key.

- **Prevent Memory Dumping:** MTA's own anti-cheat has some measures against common cheat tools, but as a script developer you can do a few things as well. One trick is to load and execute code dynamically rather than storing it plainly. For instance, you can opt *not to list the main client script in meta at all*. Instead, send it via a server trigger when the player joins. MTA's event system can transfer data securely: *"triggerClientEvent encrypts data before sending (encryption is enabled by default in mtaserver.conf) and the script content is also encrypted, so it's a double layer of encryption."* [12] This means you could keep the actual client logic on the server side (or in an encrypted file not auto-downloaded), and upon player connect, send the code to the client through an event, then execute it with `loadstring` on the client. Because the data is encrypted in transit (and you can encrypt the content yourself too), this method leaves nothing permanent on the client's disk – the script lives only in memory. The downside is a slight delay on join and needing to resend on each reconnect, but it massively reduces the chance of dumping. We will cover using `loadstring` in section 4, but note here that this is an effective anti-dump approach (the server essentially streams the code as needed).

- **Self-Deletion:** Another simple anti-dump trick is to delete files on the client after they are loaded. For example, at the end of a client script's execution, call `fileDelete("client.lua")` on its own file (or any other sensitive files in the resource). MTA allows a client-side script to delete files in its resource directory. By doing this, you remove the file from the client's storage as soon as it's read. One community recommendation was: *"Just use fileDelete in all your client-side files then nobody can steal the file"* [13] . In practice, if `cache="false"` was set, the file isn't on disk anyway; but if not, or if you have other assets, this ensures a shorter exposure window. **Caveat:** A savvy cheater running a memory dump or hooking function calls might still catch the content at load time, and using `fileDelete` does not stop a running script from being copied out of memory. However, it does prevent novice users from simply digging into the cache folder or using basic dumper tools to find your files. It's a "low-hanging-fruit" protection.

- **Hide Event Signatures:** Cheaters might try to discover event names or functions to exploit by analyzing public client code. If your resource's event names or function names are guessable (e.g., `giveMoneyToPlayer`), an attacker could attempt to call them. To mitigate this, use unobvious names or even randomized event names. For example, you could register an event with a name that includes a secret token or random string unique to each session or client. Only your scripts would know the correct name. This security-through-obscurity technique means even if a cheater tries `triggerServerEvent("giveMoneyToPlayer", ...)` it will fail because the real event is named `"giveMoneyToPlayer_ABC123"` for that session. While not foolproof (the name can be extracted from memory if the hacker is determined), it adds another hurdle against trivial event triggering. If you implement this, ensure the server communicates the dynamic event name to the client securely (e.g., first message on join).

- **Keep Server Logic Secret:** This is more of a general practice: the **best way to prevent script dumping is not to send the script in the first place**. Wherever possible, keep code on the server. Use client scripts only for the necessary client-side visuals and input handling. All important calculations and game rules should be on the server. This way, even if a client dumps everything they have, they still lack the core logic (making it harder to create a working cheat). Also, never send secrets (like encryption keys, admin passwords, or special triggers) to the client. The client is a potentially compromised environment; treat it as such by sending only what is absolutely required.

By combining caching restrictions, compilation, encryption, and dynamic code techniques, you create multiple layers of defense against script or asset dumping. A user who somehow bypasses one layer will still have others to deal with, making it extremely difficult to extract usable code or assets from your server.

3. Encryption and Obfuscation of Scripts and Assets

This section details how to convert all your resource files (especially Lua scripts) into encrypted, non-human-readable forms, using **Lua-based techniques** (no external languages, in compliance with MTA's scripting environment). The goal is to protect your resource files by both **obfuscating** their content and **encrypting** them, so that even if someone intercepts or obtains the file, they cannot easily understand or use it.

Step 1: Compile Lua Scripts to Bytecode
Begin by compiling your client-side Lua scripts into Lua bytecode. Use MTA's official Lua compiler for this, as it ensures compatibility with MTA's runtime. You have two primary options:

- **Online Compilation:** Use the MTA Lua Compiler web service. You can upload your `.lua` file and get a compiled version. The API allows setting `compile=1` and `obfuscate=3` (and turning off debug info) to get a fully obfuscated result [14] [15]. For example, a curl command:

  ```
  curl -X POST -F compile=1 -F debug=0 -F obfuscate=3 -F
  "luasource=@client.lua" https://luac.mtasa.com > client.luac
  ```

  This returns `client.luac` which is the compiled bytecode of your script with maximum obfuscation. Level 3 obfuscation will rename variables and add junk instructions, etc., making reverse-engineering very hard [8].

- **Offline Compilation:** Alternatively, download the `luac_mta` tool (available for Windows and Linux) from MTA. This tool is a drop-in replacement for the standard Lua compiler but produces output compatible with MTA (with the same obfuscation levels as the web service). For example on Windows:

  ```
  luac_mta.exe -e3 -o client.luac client.lua
  ```

  (`-e3` sets obfuscation to "even more"). This produces the same `client.luac`.

After this step, **verify that the compiled file runs** in MTA (e.g., replace the script in meta with the .luac version and test on a local server). MTA can execute compiled scripts seamlessly. The original source is no longer needed client-side. A compiled script by itself is already safer – it's not plain text – but we will strengthen it further.

Step 2: Encrypt the Compiled Scripts (Custom Format)
Next, apply encryption to the compiled bytecode file. We will create a **custom** `.enc` **format** (or you can choose any extension, but ".enc" for encrypted is intuitive) to indicate the file is encrypted. The encryption will use **Lua on the server side** (or an offline tool script) and then Lua on client side to decode. Here's how to design it:

1. **Choose an Encryption Algorithm:** MTA's built-in options are TEA (128-bit key) or AES-128 (CTR mode) for symmetric encryption, both via `encodeString/decodeString` or the older `teaEncode/teaDecode` functions [11]. TEA is simple and fast; AES is more secure. For our purposes, TEA with a strong key is usually sufficient and easier to use. For example, we might use TEA with a 16-character key. (We avoid any external C/C++ encryption – everything is done in Lua script using MTA's provided libraries.)

2. **Write an Encryption Script (Server-side or offline):** This script will read the compiled file and output an encrypted file. You can integrate this into a build process or even have the server perform it on resource start (but doing it offline or once is better to avoid performance cost each start). For example, using TEA:

```lua
-- This is a one-time encryption step (not sent to clients)
local file = fileOpen("client.luac")         -- open compiled script
local data = fileRead(file, fileGetSize(file))
fileClose(file)
local key = "MySecretKey12345"               -- 16-char key for TEA
local encryptedData = teaEncode(data, key) -- encrypt the bytecode data [16]
[10]
local encFile = fileCreate("client.enc")   -- create the encrypted file
fileWrite(encFile, encryptedData)
fileClose(encFile)
```

   This produces `client.enc` which contains binary encrypted data. (If the data might contain zeros or non-printable bytes, ensure you handle that properly; using base64 encoding on the data before TEA or after TEA is an option, but MTA's `teaEncode` can handle raw bytes directly as shown above.) The key should be a hard-to-guess string. **Never use a trivial key like "password"** – use a random mix of characters. At runtime, the same key will be used to decrypt.

3. **Adjust the Resource to use `.enc`:** Remove the original `client.lua` and even the `client.luac` from the downloadable files. Instead, add the encrypted file to the meta. For example:

```xml
<script src="loader.lua" type="client" />       <!-- a small loader script
-->
<file src="client.enc" type="client" />         <!-- encrypted script data
-->
```

   The loader script (discussed in Step 3 below) will be a tiny client Lua file responsible for reading `client.enc`, decrypting it, and executing it. By doing this, **the actual game logic never appears in plaintext or even as straightforward bytecode on the client**. It's stored encrypted. Note: the encrypted file is listed as a <file>, not a <script>, because it's not directly executable by MTA – it's just data for us to use. Also, since it's not marked as a script, you can still benefit from `cache="false"` on the loader to avoid even caching the loader (though the loader is small and not sensitive by itself aside from the key). Keep in mind: *"Don't put in the original files."* – only include the protected versions in your meta and resource package [17]. Anyone inspecting the resource will only see encrypted blobs instead of source code.

4. **Encrypt Other Assets:** Use a similar approach for other asset types in the `resources` folder: models (`.dff`), textures (`.txd`), images (`.png/.jpg`), sound files, config files, etc. Any file that you don't want players to freely access or that could give them an advantage if tampered with should be protected. For each file, choose an encryption (you can use the same TEA key or different

keys for different file types). MTA's `encodeString`/`decodeString` can handle binary data via base64 or directly for known formats. For instance, to encrypt a model file `car.dff`, you could:

```lua
local f = fileOpen("car.dff")
local dffData = fileRead(f, fileGetSize(f))
fileClose(f)
local enc = teaEncode(dffData, key)
fileWrite(fileCreate("car.dff.enc"), enc)
```

Then you'd include `car.dff.enc` as a <file> in meta, instead of the original. The client will later decrypt and use `engineLoadDFF` on the data. Similarly, you can encrypt texture files (perhaps as `.txd.enc`) and any others. We'll handle how to load these in section 5, but essentially it's the same pattern applied across all resource files.

At this point, all your scripts are compiled and encrypted, and your assets are encrypted. The resource that you distribute or use on the server no longer contains easily readable code or raw assets. Everything is in a proprietary, custom format known only to your loader/decryption code.

It's worth noting some pros/cons of this approach (which mirror what community-developed tools have noted):

- **Pros:** Your scripts are in binary form (harder to reverse) and additionally encrypted with a secret key. Without that key, even if someone grabs the file, it's useless. Also, by doing decryption in Lua at runtime, you avoid needing any external DLL or module (maintaining portability and keeping within MTA's allowed methods). This approach does not require any server calls during gameplay for decryption, so after the initial load it runs offline on the client (no decryption query to server needed) [18]. If done right, the decrypter (loader) script itself can be compiled and marked not to cache, meaning hackers can't easily snag the key from it either [19]. A strong encryption (like AES or TEA with a long random key) is effectively unbreakable by brute force in any reasonable time [20], so your content is very secure.

- **Cons:** There is a performance cost to decrypting content on the client. However, TEA and AES are quite fast for the sizes of data typical in game scripts and assets, and you can decrypt once at startup. Another con is complexity: you must manage keys securely (don't accidentally include them in public repositories; consider different keys per resource or updating keys periodically). Also, if a skilled individual somehow obtains both the encrypted file *and* your loader script (with the key), they could still decode the content. Our goal is to make that as hard as possible (hence compiling the loader too). No method is 100% foolproof, but these layers of obfuscation and encryption are as close as it gets within MTA's scripting abilities.

Step 3: Develop Runtime Loaders for Encrypted Files
Encryption is only half the battle – now you need to *use* those encrypted files in the game. This means writing Lua code that, at runtime, will **read, decrypt, and load** the content of each protected file. This

loader code must of course have access to the encryption key. Here we implement the secure loading mechanism for each type of asset:

- **Loader for Encrypted Lua Script:** Create a small client-side script (we called it `loader.lua` above) whose job is to load the main encrypted script. This loader will be the only client code in plain (though you can also compile it for extra safety). For example:

```lua
-- loader.lua (client-side, runs on resource start)
local key = "MySecretKey12345"  -- same key used in encryption step
local file = fileOpen("client.enc")
if file then
    local encData = fileRead(file, fileGetSize(file))
    fileClose(file)
    local code = teaDecode(encData, key)         -- decrypt the bytecode [21]
    if code then
        local func = loadstring(code)           -- load the Lua chunk from
decrypted bytes
        if func then
            func()                              -- execute the loaded chunk
        else
            outputDebugString("Decrypted code couldn't load", 1)
        end
    else
        outputDebugString("Failed to decrypt client.enc (key mismatch or
corrupt file)", 1)
    end
else
    outputDebugString("Encrypted script file not found", 1)
end
```

  Let's break down what this does. It opens `client.enc`, reads all bytes, then uses `teaDecode` with the known secret key to get the original code back [21]. The result `code` should be the exact Lua bytecode that was compiled, just as a string of bytes. We then use `loadstring` (or `load` in newer Lua) on that string. Because it's Lua bytecode, `loadstring` will produce a function that is the compiled script ready to run. We then call that function to actually execute the script in the client's environment. At that point, it's as if the original `client.lua` were running, except it was decoded on the fly. The `loader.lua` can then self-delete if you want (though since it contains the key in memory, that key could theoretically be found by memory inspection; to mitigate that, you could obfuscate the key itself in code or construct it at runtime from pieces).

A few notes: It's a good idea to handle errors (as shown with debug messages) — e.g., if decryption returns nil (perhaps someone tampered with the file or used a wrong key) — and optionally take action (like alerting the server that the file was unexpectedly not decrypted properly, since that could indicate tampering). Also, using `cache="false"` on this loader script ensures it isn't saved on disk, protecting the key further [22]. Additionally, you could compile this loader too (so that the key string isn't plainly visible). If compiled and not cached, even if a hacker dumps the loader from memory, it's obfuscated bytecode containing the key, not straight text.

- **Loader for Encrypted Models/Textures:** For other file types, you will use their respective load functions that accept raw data. MTA's engine functions often allow you to pass a chunk of data instead of a filename. For example, `engineLoadDFF` and `engineLoadTXD` can take a raw data string for the model/texture data instead of a path [23]. The process for each encrypted asset is:
- Read the encrypted file using `fileOpen` / `fileRead` on the client.
- Decrypt the data with the same key (using `teaDecode` or `decodeString`).
- Pass the decrypted binary to the appropriate loader function.

For instance, for a DFF model:

```
local f = fileOpen("car.dff.enc")
local encData = fileRead(f, fileGetSize(f))
fileClose(f)
local dffData = teaDecode(encData, key)             -- decrypt model data
if dffData then
    local dff = engineLoadDFF(dffData)              -- load model from raw data
bytes [24]
    engineReplaceModel(dff, 420)                    -- replace model ID 420
(example)
end
```

Similarly for a TXD texture dictionary:

```
local f = fileOpen("car.txd.enc")
local txdData = teaDecode(fileRead(f, fileGetSize(f)), key)
fileClose(f)
if txdData then
    local txd = engineLoadTXD(txdData)
-- engineLoadTXD can take raw TXD data
    engineImportTXD(txd, 420)
end
```

And for general images (e.g., if you have a custom .png for GUI that you encrypted), you can decrypt the bytes and use `dxCreateTexture` with the raw pixel data string [25]. For example:

```
local encImg = fileOpen("map.png.enc")
local imgData = teaDecode(fileRead(encImg, fileGetSize(encImg)), key)
fileClose(encImg)
if imgData then
    local texture = dxCreateTexture(imgData, "argb")   -- create texture from
raw PNG data
```

```
    -- now use dxDrawImage or others with this texture
end
```

The key point is MTA provides flexible loading functions that accept raw data, enabling us to feed decrypted content directly without ever writing it to disk in plain form. The technique is consistent: open file, decrypt with key, load resource from memory.

- **Loader for Encrypted Config/Data Files:** If you have configuration files or other data (perhaps an XML or JSON that you don't want exposed or altered), you can also encrypt them and load similarly. For example, if you had a JSON config, you could encrypt the JSON string. Then on client, decrypt it and use `fromJSON` to parse it. Or if it's an XML, you might actually consider not sending it at all (server can just send needed config values). But if needed, decrypt and use MTA's XML functions on the decoded string (MTA doesn't parse XML from string directly, but you could write a simple parser or use Lua's loadstring if you convert XML to Lua table, etc.). This is an advanced scenario – often it's simpler to keep such data server-side or embed it in the compiled script.

After writing these loader routines, your `meta.xml` should load the loader scripts on client startup (and those will internally load everything else needed). Test each loader thoroughly: ensure that a client successfully decrypts and uses the assets (you should see models replaced, etc., if the keys and process are correct).

Summary of this Encryption/Obfuscation Phase: All client-side code and assets are now either compiled to bytecode, encrypted, or both. They are loaded through custom Lua logic at runtime. This fulfills the requirement of converting scripts into a custom format (`.enc`) with proprietary encoding, and using Lua (and MTA's support libraries) to encrypt/decrypt content. By using compiled bytecode and encryption, we've achieved a robust obfuscation: **even if someone opens the resource files, they see binary junk, not your game logic or original assets.**

Finally, note that you should keep backup copies of your original scripts and unencrypted assets securely (not in the deployed resource). The compiled/encrypted versions can't be recovered to original form easily – which is the point – so maintain your source code in a safe place. Only distribute the protected versions.

4. Custom Format for Secure Script Loading (`.enc` files)

Designing a *custom format* for your scripts means you define how the script is stored and retrieved such that only your program knows how to execute it. In our case, the "format" is basically the encrypted bytecode with a known key. We've already essentially done the design in the previous section, but let's lay it out clearly as a roadmap item:

- **Format Definition:** We choose a file extension like `.enc` to denote an encrypted file. The content of this file is the raw output of a symmetric encryption algorithm applied to either the source or compiled Lua code. In our implementation, `.enc` contains the TEA-encrypted compiled bytecode of a Lua script. You could add your own metadata or structure (for example, a custom header, or splitting the file into sections), but that's generally unnecessary. A simple format where the entire file is just an encrypted blob is sufficient.

- **Encryption Keys Management:** The security of the custom format relies on a secret key. You need to decide how to store or generate this key. Options:

- Hardcode the key (or parts of it) in the client loader script. Our example did this (a plaintext key string in `loader.lua`). This is easy but somewhat risky if someone were to dump that memory. To obfuscate it, you can break the key into pieces or perform operations to derive it so it's not plainly visible. For instance, store it as a math expression or as char codes, etc., inside the compiled loader. Since the loader itself can be compiled, the key won't be in plain text form in the resource file.

- Do not store the key on the client at all: instead, have the server send the key at runtime (like via `triggerClientEvent`) just before decryption is needed. This way, the key lives in client memory only fleetingly. Remember, MTA's event data is encrypted in transit [12], so sending the key from server to client is secure against eavesdropping. The client can then use it to decrypt. You might combine this with caching: e.g., the encrypted script file stays on the client's disk between sessions, but the key is sent each time they join (if they don't have the key, the file is useless). This approach is very secure – even if someone obtains the `.enc` file, they can't run it without the key from the server. It does add complexity: you'll need to trigger the key send and have the client wait for it before decoding. But it's a viable design for a truly secure custom format. (One could even implement a challenge-response so the key is different each session, but that may be overkill.)

- **Runtime Interpretation:** Define how the encrypted script is loaded. We already wrote the loader code which *is* the interpretation logic: "read bytes -> decrypt -> loadstring -> execute". That is the core of our custom format's interpreter. When implementing, ensure you handle errors gracefully (for example, if the file is missing or corrupted, the code should not crash the client). Perhaps have the loader report back to the server if it fails to load (so you know if someone's files are not loading, which could hint at tampering).

- **Example Custom Format Workflow:**

- **Packaging**: Developer compiles `gameplay.lua` to `gameplay.luac`, then encrypts it with key `K` to produce `gameplay.enc`.
- **Distribution**: Include `gameplay.enc` in the resource and a `loader` script that knows `K`.
- **Execution**: Client joins, MTA downloads `loader` and `gameplay.enc`. On resource start, `loader` runs, reads `gameplay.enc`, decrypts with `K`, and does `loadstring`. The `gameplay` code is now live in memory.

- **Security**: If the player tries to open `gameplay.enc` outside the game, it's meaningless without `K`. If they intercept the network download of it, it's encrypted. If they search their disk, there's no `gameplay.lua` to find. Only a heavily obfuscated loader (which itself can be secured).

- **Use of `luac` and binary loading**: Our custom format leverages the fact that Lua can load bytecode. The `loadstring` function will accept the bytecode string because it starts with the expected Lua binary chunk signature. This is important: if you ever accidentally tried to `loadstring` encrypted data or text data, it would fail. Only after correct decryption do we get a proper Lua chunk. Keep the encryption aligned (for instance, if using AES in CTR mode via `encodeString('aes128', ...)`, note that an IV is used – you'd need to store the IV or use a fixed IV for decrypt to succeed [26]. TEA doesn't use IVs, making it straightforward here).

- **Testing the Custom Format:** Always test that your `.enc` file can round-trip: encrypt -> decrypt -> run. On a development server, you could include both original and encrypted and have a command to compare outputs of running each, but ultimately you'll remove the original. A common mistake might be mismatched keys or using the wrong algorithm for decode. If the loader prints "Failed to decrypt" or the code doesn't execute, double-check that the same key and algorithm are in use on both sides.

In summary, the custom format (like `client.enc`) is essentially an **MTA-specific packaged script**: compiled and encrypted Lua code that only your loader can understand. This satisfies the goal of designing a protected format for scripts which can be securely loaded at runtime. The concept can be extended to any file type as needed.

By this stage, your anti-cheat system has robust measures for code protection – but an anti-cheat is not only about hiding code. In the next sections, we focus on ensuring *all* resources are protected and on packaging, deployment, and active detection of cheating behavior.

5. Full Resource Folder Protection

An anti-cheat system must shield **all components** of your resource, not just scripts. Cheaters could exploit or steal any client-side asset: images might reveal secret information or be altered for advantage, models could be extracted, etc. This step-by-step plan covers protecting every relevant file in the `resources` folder using the techniques above, ensuring a holistic security of the resource package:

- **Protect Scripts:** *(We have largely done this already, but to recap in context.)* All client and shared Lua scripts should be compiled and/or encrypted. Remove any plain-text `.lua` files from the delivered resource. Only include compiled `.luac` or encrypted `.enc` script files along with the small loaders. Use `cache="false"` for any script that must remain in memory only. This makes it *"hard for those having bad intentions... to obtain client code"* [9]. By doing so, you close off the easiest route of cheat developers who look for logic flaws in your Lua – most will be discouraged when they see nothing but obfuscated code.

- **Protect Config/Data Files:** If your resource has configuration files (e.g., `.xml` files, `.map` files for maps, or `.json`/`.ini` data), decide how to secure them. Server-side configs (like settings that never go to clients) can remain as is (since clients can't read server files). But any config sent to clients (for example, a map file describing spawn points or pickups) can be vulnerable. Cheaters might parse a map file to find hidden objects or sensitive info. To protect map or XML files, you have a few options:

- *Embed into Script*: Convert the config into Lua tables within a script, then compile/encrypt it like any other code. Essentially treat config data as code constants. This way it gets same protection as code. The downside is you lose the ease of editing that an external file provided.
- *Encrypt and Load*: You can encrypt the text of the config file and have the client decrypt and parse it. For instance, if you have `config.xml`, you could encrypt its content and include as `config.xml.enc`. On client, decrypt to a string and then use Lua's XML functions to parse. MTA doesn't have a direct function to parse XML from string, but you can write a simple XML parser or use Lua patterns to extract needed info. Alternatively, consider converting XML to a Lua script (like a

series of createObject calls for a map) and then treat it as a script (embedding map info in code, which you compile).

- *Rely on Server*: The ultimate protection for something like a map is to not send it at all. You could load the map server-side (since server has the raw file) and then spawn objects via server-side script for the clients (MTA will sync objects to clients created with `createObject` on server). This requires more work to simulate what the .map would do, but it keeps the actual map file private.

For simplicity, using encryption on data files is a straightforward route. For example, if you have a list of secret locations in a text file, encrypt it and at client runtime decrypt and use it in code (thus players can't just open the text file to get all the answers).

- **Protect Images/Textures:** If your game uses custom images (for GUI, maps, etc.), these image files (.png, .dds, etc.) get downloaded to the client. A malicious user could open them outside the game to glean information (maybe an easter egg map or a puzzle solution image) or to redistribute your artwork. By encrypting them (as we described: store as `.png.enc` and decode at runtime), you prevent casual browsing. Only your game will decrypt and use them via `dxCreateTexture`. There is a note that reading images as binary and creating textures can sometimes have pitfalls (like needing proper format flags) [27] , but as long as you pass the correct format to `dxCreateTexture` (or use DDS which is basically raw), it should work.
  Community advice in MTA for this scenario: *"encrypt them using teaEncode and then on the client decrypt them and pass the decrypted data to engineLoadDFF and so on."* [28] – this was about models, but it applies to any asset. Indeed, one user released a system to encrypt *models, shaders, images*, indicating the method can cover all those file types [29] .

- **Protect Models and Sounds:** Models (`.dff`, `.txd`) we covered with encryption and using `engineLoad*` on the fly. This is a known and recommended method in the community to prevent theft of custom models. The code snippet we included earlier (encrypt on server, decrypt and `engineReplaceModel` on client) is exactly how to implement it [30] [31] . Do the same for any custom sound files (`.mp3` or `.wav`): while MTA doesn't have a direct "loadSound from bytes" function, you can work around it by writing the decrypted bytes to a temp file and playing it, or by using the **playSound** from URL with a data URI (if MTA supports data URIs – that might not be the case, so a temp file might be needed). However, sound files are less commonly secured due to complexity; if your sounds are not critical secrets, you might skip encryption on those. But if you have voice lines or secret messages you don't want extracted, consider it.

- **Resource File Rights & ACL:** Another aspect of full protection is ensuring that no other resource or unauthorized server admin can open or dump your files. By default, resources cannot read each other's files. Only resources with the appropriate ACL rights (like the "Admin" or "Console" resource with general.FileRead rights) could attempt to open another resource's files. Make sure your server's ACL is configured so that no malicious resource can start and read your resource's files. Generally, do not give sensitive rights to resources you don't trust. This is more server administration than scripting, but it's worth mentioning: your anti-cheat resource itself should probably be given Admin ACL (since it may need to ban players or use certain functions), but normal game mode resources should not have file access to others. This prevents a scenario where an insider could install a resource that snoops your files from the server side.

- **Exclude Unneeded Files:** Review everything in the `resources` folder – if something is not needed by the client, do not send it. For example, large reference data that only the server uses can be marked server-only or omitted from meta so clients never download it. Less exposure = less risk. Also, consider packaging your resource (MTA allows resources to be zipped as `.zip` with the meta and files inside). If you do use a single zip for the resource, it's slightly harder for a user to casually browse (though they can still unzip it, but at least you don't have loose files scattered). Regardless, with encryption, even if they unzip, they see `.enc` files.

In essence, **apply the same encryption/obfuscation pipeline to every client-bound file** in the resource: scripts, models, textures, maps, configs, etc. By the end, the client's cache (or memory) contains nothing decipherable: all important files are either not present, or present in encrypted form that only your program can decode. This comprehensive approach ensures that cheaters cannot easily tamper with or extract any piece of your content for unfair advantage or leak.

(As an example of full protection: one could create a tool that bulk-encrypts every file in the resource with TEA and generates a corresponding loader script. In fact, projects like "NandoCrypt" have automated much of this process for models and other files [32] [33]. Adopting such tools or the techniques they use can save development time and ensure no file is left unprotected.)

6. Secure Packaging and Deployment

With all these security measures in place, it's important to package and deploy your resource correctly, otherwise the protections might fail. This phase of the roadmap deals with how you **bundle, distribute, update, and maintain** the anti-cheat protected resource. It ensures that from development to production use, the integrity of the system is preserved.

- **Integrate Protection into Build Process:** It's wise to automate the compilation and encryption steps so that no human error causes a sensitive file to be left in plain form. For example, you might have a build script that when run, compiles all .lua to .luac (using the MTA compiler) and then runs a Lua script (server-side or external) to encrypt those .luac files and any assets. The script can also update the `meta.xml` entries automatically to point to the encrypted files (there are community tools that do exactly this, replacing meta entries with .luac versions [34]). By automating, you ensure consistency. Set up a routine like: "before deploying new version, run `build_secure.bat`" which outputs a ready-to-deploy folder with only safe files.

- **Do Not Distribute Secrets or Source:** When you deploy the resource to your live server (or share it with others), double-check that no plaintext secrets are included. This seems obvious, but sometimes developers accidentally leave a `.lua` in the resource archive or a backup file. Go through the package: it should contain **no** `.lua` **or unobfuscated text** except maybe the meta and a readme. The meta.xml itself might reveal some info (like resource names or event names). There's not much you can do about that except naming things unobtrusively. But ensure keys are not in meta (they shouldn't be). Also, **don't include the compilation/encryption tool or scripts in the public resource** – those should remain on your side. Only the runtime components (loaders, encrypted files) go out.

- **Resource Versioning & Cache:** MTA clients cache resource files by default (unless told not to). If you update your resource (say you change some code and thus produce a new .enc file), clients might not automatically fetch the new file if the resource name/version is unchanged and caching is enabled. To force updates, you have a few strategies:

- Increase the resource version (if you use the optional `version` attribute in meta or change the resource name slightly).
- Simply have the client redownload by either instructing them to clear cache or by the server using the `uploadFile` mechanism if available (not commonly used).

- A reliable way is just to **restart the resource** on the server after updates. MTA's HTTP download system will see the file has changed (since the resource was restarted and presumably the script will have a new hash) and it will fetch the new one. As one community member noted, *"You would have to restart the resource after the encryption so that the HTTP cache is updated and players download the new encrypted versions."* [35] . So each time you deploy changes, do a resource restart (or stop server -> update files -> start server). This ensures no one is running an outdated mix of old/new files which could cause decrypt failures or security holes if, say, the key changed.

- **Secure Transfer:** When players download the resource, MTA uses an HTTP server on the game server to serve files. Ensure this transfer is not hijacked. MTA's default is plain HTTP (not encrypted) for file download, but since your files are encrypted anyway, it's not critical if someone intercepts them – they still can't decode them without keys. However, to prevent tampering (improbable, but if someone could man-in-the-middle the connection and swap your files), you might consider enabling **HTTPS** for resource downloads if you run a custom web server for that. This is an edge case, as typically one would need direct network MITM to do it. MTA doesn't natively provide HTTPS for file serve as far as I know. The built-in anti-cheat will reject modified files (since the server will verify the file hash as part of resource integrity checks). So this is generally not a worry.

- **Key Management:** Treat encryption keys like passwords. Only give them to those who absolutely need them (for example, if you have co-developers). If working with a team, consider that each developer can work with the unencrypted source, but the key should not be stored in a public repository. One recommendation is: *"If working on a project with multiple people or a shared repository, don't share the secret key used to encrypt the files."* [36] . You might have each dev encrypt with their own key for testing, and only at release time use the real key. Or use environment variables/config that are not committed to hold the key for the build script. The idea is to minimize the chance the key leaks, because if it does, all that encryption can be undone by the person with the key. If a key is compromised, you should change it and re-encrypt assets as soon as possible in an update.

- **Packaging the Anti-Cheat Resource:** If your anti-cheat system spans multiple resources (for example, a separate "anticheat" resource plus the game mode resource), ensure all are protected similarly. The anti-cheat resource itself might contain sensitive code (like detection algorithms) that you want to hide. Use the same practices on it (compile it, encrypt if needed). This can be meta because if it's a server-side only anticheat (some logic to monitor players), you might not need to send anything to client beyond perhaps a small client detector script. But if you do send client scripts (like a cheat scanner that runs on client), secure them too.

- **Distribution to Other Servers (if applicable):** If you plan to let others use your anti-cheat system (maybe as a resource you share), consider providing it in a pre-compiled form only. This is essentially what you're doing anyway. You might include documentation of how to use it but not expose the code. Also, if distributing externally, be mindful that once it's out, someone could try to break it – so really make sure the above steps are done thoroughly. Perhaps include a license note and a warning that attempting to crack it is against terms (though that might not stop anyone, it sets an expectation).

- **Testing and Debugging:** One challenge with heavily obfuscated code is debugging issues. During development, test with the protections *off* until things work, then apply encryption and test again. It's easier to debug with readable code. But after deploying encrypted code, if something goes wrong, you may need to add debug prints or logs in the loader or in the server to understand what happened (e.g., if a decrypt failed). Always keep an unprotected version for yourself to replicate bugs.

- **Performance Considerations:** Packaging also involves ensuring the encryption doesn't slow down load times too much. If you find that decrypting a large file (say a 10MB model file) is slow on client join, you might want to move that decryption to after join or in parts. You could even show a "Loading..." prompt while decrypting assets. Or split a huge file into chunks. Generally, small scripts decrypt in milliseconds, models/textures of a few MB might take a fraction of a second to a second. It's usually fine, but keep an eye on it. **Use asynchronous loading if needed** – MTA's `decodeString` supports an asynchronous callback mode [37], which you could use to avoid freezing the game during a long decode. This might be overkill, but it's an option.

- **Leverage MTA Anti-Cheat Config:** When deploying, also configure MTA's built-in anti-cheat settings (`mtaserver.conf` entries like <enablesd>, etc.) as appropriate. For example, enable Special Detections for known cheats if they don't interfere with your gameplay. MTA's AC can kick for memory hacks and known cheat tools; keeping it enabled complements your own system. The built-in AC won't conflict with your custom anti-cheat as long as you aren't doing forbidden things (like using disallowed functions). So ensure in packaging that your server's anti-cheat is on and not disabled unless necessary.

In short, **deployment is about diligence**: make sure only protected files go out, keys stay secret, and clients always get the latest secure version. With a solid packaging routine, your anti-cheat system will remain consistently effective through updates.

7. Detection and Logging Mechanisms

Up to now, we focused on preventive measures (obfuscation, encryption, triggers protection) that make cheating harder. Equally important is to **detect and respond** to any cheating or tampering that does occur. A robust anti-cheat system should actively monitor for suspicious behavior and keep logs of incidents for further analysis or automatic punishment. Here's a roadmap for implementing detection and logging in your MTA:SA server:

- **Event Violation Detection:** As described in section 1, whenever an event trigger from a client fails a security check (e.g., wrong `client`, disallowed parameter value, etc.), treat it as a cheat attempt.

Implement a unified function to handle these event violations. For example, if `processServerEventData` (like in the wiki example) returns false due to a security check fail, call a function `reportEventViolation(client, eventName, details)`. This function can:

- Log a clear message to server console/log: e.g. `"ANTI-CHEAT: Player John (Serial XYZ) attempted to call onServerGiveMoney with invalid source!"`.
- Record the event in a dedicated log file (you could create `logs/anticheat.log` and use `fileWrite` on the server to append entries).
- Optionally, notify online admins (e.g. via `outputChatBox` to staff or an admin panel) that a possible hack attempt was detected.

- Increment a counter on that player's record (could store in a table or database) for cheat attempts.

- **Automatic Punishments:** Decide on a policy for punishing cheat attempts. A common approach is escalating penalties:

- **Warn** on the first offense (maybe just log and maybe send a warning message to the player that cheating was detected).
- **Kick** on the second offense.
- **Ban** on subsequent or immediate ban for severe cases (like trying to trigger an admin-only event might be zero-tolerance ban).

You can automate this. For example, maintain a table of `attemptCount[player]`. Each time `reportEventViolation` is called, increment the count. If it reaches 2, kick the player (`kickPlayer`). If it reaches 3, ban the player (`banPlayer`) possibly permanently or for a significant time. Ensure you log these actions as well ("Player X was banned by anti-cheat for event hacking"). In the MTA community example, they set configurations like `punishPlayerOnDetect = true` and `punishmentBan = true` for serious detections, with a reason set (e.g., "Altering server event data") [38]. You can replicate that: when you call `banPlayer`, include a reason like "Anti-cheat: unauthorized trigger" so it's clear in ban lists. Using the admin console as the banner (which can be done by passing `false` for responsible player in `banPlayer` or using the console element) makes it an official action (the wiki example uses "Console" as punishedBy [39]).

- **File Tamper Detection:** Since you've encrypted files, direct tampering is unlikely (they'd just break the file). But you can still include integrity checks. One idea: after decrypting a script or asset, you could verify it matches an expected hash or watermark. For instance, you know the hash of the original script bytecode; you could store that (perhaps even inside the encrypted file as a small plaintext footer, or hardcoded in the loader). After decrypting, calculate `hash(decryptedData)` (MTA has an `hash` function for MD5/SHA1 [40]) and compare. If it doesn't match, that means the file was modified or corrupted. Immediately report this and do not execute it. This would detect if someone somehow altered the encrypted file (though if they can without the key, not likely, but maybe someone could swap it with another encrypted blob – it would fail to decrypt properly anyway). So this might be more paranoia than necessary. Still, adding a simple check doesn't hurt. For example:

```lua
local expectedHash = "<somevalue>"
if string.upper(hash("sha256", decryptedData)) ~= expectedHash then
    triggerServerEvent("onAntiCheatFileMismatch", resourceRoot,
expectedHash)
    return
end
```

On server, `onAntiCheatFileMismatch` could log the incident and maybe kick the player for having an unexpected file. Again, it's rare to see this triggered unless files corrupt, but it's a thought.

- **Runtime Behavior Monitoring:** Cheating isn't only about triggers and files. Players might use trainers to modify memory (speed hacks, health hacks) or use macro tools. MTA's built-in anti-cheat (AC) catches many of these (like health, ammo hacks, known trainers via AC # codes). You can supplement:

- For example, implement a **money sanity check**: if your game doesn't allow earning more than $1000 in a minute and suddenly a player's money jumps by $1,000,000, that's suspicious. Your server-side should normally control money, but if a client somehow gave themselves money (which *should* be impossible if server authoritative, but if there's an exploit), detect the anomaly and correct it.
- Another example: monitor position/teleport. If a player coordinates change too fast beyond allowed speed, flag it (could be a teleport hack). MTA AC might catch major ones, but subtle teleports might slip. You could keep last position and time, and if distance/time > some threshold (like moving 100 meters in 0.1s with no vehicle), then log that.
- **Projectile/explosion spam**: The MTA wiki anti-cheat example includes tracking projectiles created to prevent explosion cheats [41] [42]. Depending on your game, you might incorporate similar logic to detect if a player is creating objects or effects that they shouldn't (though MTA has events like onPlayerWeaponFire you can monitor).

The specifics depend on your game mode. Essentially, think of what a cheater would do in gameplay and add checks. Many of these checks are best done server-side (since the server can be trusted). Log any detected irregularities with context.

- **Add Debug Hooks (Advanced):** MTA provides `addDebugHook` which can catch certain internal events, like when a player triggers a Lua error or when certain low-level operations happen. This is more for debugging scripts, but it could potentially catch if someone is using something like `executeSQLQuery` in an unauthorized way or if a resource is throwing unusual errors that might come from injected code. This is quite advanced and not commonly needed, but mentionable if you want a comprehensive approach.

- **Logging Format and Storage:** Make your logs actionable. Include timestamps on each log entry (you can get date/time with `getRealTime()` and format it). For example: `[2025-11-29 22:31:00] [ANTI-CHEAT] JohnDoe (Serial ABC123) kicked for cheat attempt: vehicle speed hack`. If you have a lot of logging, consider using an external logging library or output to a database or Discord webhook for live alerts. But a simple file or console log is usually fine.

- **Review Logs Regularly:** The roadmap isn't just technical but also operational: designate someone (or yourself) to review the anti-cheat logs. You might catch patterns like a certain player trying different things, or multiple players from the same IP doing stuff, etc. Logs can also help improve the system – if you see a lot of false positives for a particular check, you can refine the logic or thresholds.

- **Player Reports & Manual Checks:** Even with automation, keep a command or interface for admins to manually trigger scans or checks on a player. For example, a command to force a client's anti-cheat component to run a scan (if you have any client-side detection like scanning for disallowed mods). While MTA's anti-cheat covers disallowed mods globally, you might have custom things to check (like ensuring they haven't modified certain files if you have custom handling). Logging can include things like suspicious chat messages or known cheat phrases. This goes beyond core anti-cheat, but a holistic approach often ties player reports (like if many people report a player, mark their logs for review or do a surprise check on them).

- **Protecting Anti-Cheat Itself:** Ensure that cheaters cannot easily disable your anti-cheat resource. For instance, if a cheater somehow gains admin on your server (unrelated to game cheat, but via social engineering or an exploit), they might stop your anti-cheat resource. To mitigate: use the ACL to restrict who can stop it (only console perhaps). You could also have a watchdog: a separate small resource that checks periodically if the anti-cheat resource is running and restarts it if not, and logs that incident (if it was stopped, that itself is suspicious). This might be overkill, but some servers do have self-healing scripts.

- **Use MTA's OnPlayerModInfo:** The default resource "acpanel" uses an event `onPlayerModInfo` which informs the server if a player has modified certain files (like GTA's game files). If your server is competitive, you might enable checks to ensure players aren't using modified game files (like a custom handling.cfg for super speeds). This isn't directly about your resource, but it's relevant to anti-cheat. If you enable those (via <client_file> settings in mtaserver.conf [43] [44]), log any hits (the event provides details on what file was different). Then you can decide to kick those players (since modded files could give unfair advantages like different map collisions etc.).

- **External Anti-Cheat Logging:** If desired, integrate with external systems. For example, log cheat attempts to a Discord channel (so admins get immediate alerts). There are MTA Discord bot resources that allow sending messages from the server to Discord via HTTP. This way, even if no admin is watching the console, someone gets notified of a cheat attempt or auto-ban.

To illustrate detection logging, consider an example scenario: A cheater tries to call a protected event "onServerAdminGiveMoney" with another player as source. Our anti-trigger code notices `client` is not an admin and that the source doesn't match `client`. The event handler calls `reportEventViolation`. We log:

```
[ANTI-CHEAT] 2025-11-29 22:35:10 – Player "EvilHacker"(ID 5, Serial 123456...)
attempted onServerAdminGiveMoney with spoofed source. Action: banned.
```

We then ban the player and perhaps broadcast that they were removed for cheating. This goes into our anticheat.log for future reference. If the same exploit was attempted by others, we'll see it in the log and know our system is catching it.

In summary, the detection and logging component turns your anti-cheat from a passive defense into an active one. It not only blocks cheats but also tracks and penalizes those attempts. The combination of real-time prevention (the earlier sections) and robust logging/punishment (this section) will strongly discourage cheaters: they not only fail to cheat, but they get caught in the act. Over time, you can adjust the sensitivity of detectors (to avoid false positives) and add new ones as new cheat methods are discovered. Always keep the logs – they are invaluable for improving your anti-cheat and for evidence if you need to justify bans. With these systems in place, your MTA:SA server will be significantly hardened against cheating and unauthorized modifications.

Sources: The strategies above are informed by best practices and community insights. MTA's own wiki and forums stress using built-in features like `addEvent`'s allowRemote parameter [1] and the `client` global [3] to prevent fake triggers. For file protection, techniques such as TEA encryption of files and on-the-fly decryption have been demonstrated in the community [30] [24]. The use of compiled scripts (luac) with obfuscation is officially supported and recommended [8]. Real projects like *NandoCrypt* have successfully implemented AES encryption for resources, compiling the decrypter so it "can't be uncompiled" and is not cached on client [19]. The approach to send scripts via `triggerClientEvent` under encryption was suggested to double-protect code in transit [12]. Additionally, forum advice confirms that encrypting models and other files and then decrypting in client memory is an effective way to prevent theft [28]. These techniques, combined with vigilant logging and use of MTA's anti-cheat flags [38], create a multi-layered anti-cheat system aligned with what experienced MTA developers recommend.

---

[1] [2] [3] [4] [6] [7] [8] [9] [38] [39] [41] [42] Script security - Multi Theft Auto: Wiki
https://wiki.multitheftauto.com/wiki/Script_security

[5] [43] [44] Anti-cheat guide - Multi Theft Auto: Wiki
https://wiki.multitheftauto.com/wiki/Anti-cheat_guide

[10] [16] [17] [21] [23] [24] [28] [30] [31] [35] Protecting files - Scripting - Multi Theft Auto: Forums
https://forum.multitheftauto.com/topic/90521-protecting-files/

[11] [26] [37] [40] DecodeString - Multi Theft Auto: Wiki
https://wiki.multitheftauto.com/wiki/DecodeString

[12] [13] script protection - Scripting - Multi Theft Auto: Forums
https://forum.multitheftauto.com/topic/66120-script-protection/

[14] [15] Lua compilation API - Multi Theft Auto: Wiki
https://wiki.multitheftauto.com/wiki/Lua_compilation_API

[18] [19] [20] [22] [32] [33] [36] GitHub - Fernando-A-Rocha/mta-nandocrypt: File Encryption Tool for Multi Theft Auto: San Andreas
https://github.com/Fernando-A-Rocha/mta-nandocrypt

[25] DxCreateTexture - Multi Theft Auto: Wiki
https://wiki.multitheftauto.com/wiki/DxCreateTexture

[27] Using dxCreateTexture with raw data makes textures blurry - Scripting
https://forum.multitheftauto.com/topic/139167-using-dxcreatetexture-with-raw-data-makes-textures-blurry/

[29] [HELP] engineLoadDFF using raw data - Scripting - Multi Theft Auto: Forums
https://forum.multitheftauto.com/topic/105682-help-engineloaddff-using-raw-data/

[34] brenodanyel/mta-compile-scripts - GitHub
https://github.com/brenodanyel/mta-compile-scripts