NEW!

How to crack a Discord Hook

Using IDA Pro to crack Rivals Hook

Writeup by Rajat Balwani with the help of Ryuga Balwani

21-01-2026

PART 1

INDEX

TOPICS	PAGES
The Overview	Page 01
The Hooking Mechanics	Page 02
The Toolkit	Page 03
The Theoretical Foundation	Page 04
The "Entry" Phase	Page 05
The Deep Dive	Page 06
The Analysis	Page 07
The Strategy	Page 08
The Implementation	Page 09
The Conclusion	Page 10

THE OVERVIEW

A Deep Dive into Reverse Engineering with IDA Pro

This writeup focuses on exploring the Rivals Discord hook to understand how its license verification works and why it fails. Rivals is intentionally low-quality and largely copied from leaked hook source code. This makes it a convenient target for educational reverse engineering rather than an example of genuine protection. The aim is not to defeat a complex system, but to show how weak, keyauth license checks break down with even minimal scrutiny.

The scope of this writeup is limited to the basics of reverse engineering. Rivals has no meaningful protections, such as notable obfuscation, integrity checks, or serious anti-debug or anti-tamper measures. Therefore, advanced techniques are unnecessary. Basic static analysis is enough to find the license-related logic by inspecting strings, imports, and simple control flow. Dynamic analysis is then used to confirm assumptions at runtime, observe when the license checks run, and see how functionality is restricted or disabled.

By combining these two approaches at a basic level, the writeup highlights common mistakes found in rushed or poorly made Discord hooks. These mistakes include trusting the client, depending on predictable checks, and assuming users won't look beyond surface-level logic. Rivals is chosen specifically because of its flaws. These flaws become evident upon examination, making it suitable for learning fundamental concepts without unnecessary complexity. The developer of this hook (Mercy) is a pure skid & of-course a newgen.

The ultimate goal is to understand how these systems are typically set up, why poor implementations fail, and how basic reverse engineering techniques can bypass them when no real protection is in place.

THE HOOKING MECHANICS

Basic introduction on Function Hooking

What is a Hook?

A hook is a technique used to interrupt and redirect function calls within a running program. Instead of letting a function execute normally, the execution is redirected so that custom code runs before, after, or instead of the original logic. This usually involves patching function entry points, changing function pointers, or detouring execution flow. Hooks are powerful because they work at key decision points. Change the behavior of a function, and the program follows that new path.

The Discord Ecosystem (Audio-Level Hooks)

In the context of Discord voice, hooks work directly within the client's audio pipeline. Discord uses Opus for audio encoding and decoding, and PCM audio is processed locally before being sent to voice servers. Discord hooks intercept internal functions that handle audio capture, PCM processing, Opus encoding/decoding, and voice packet construction.

By hooking these functions, a hook can inject custom audio data, change PCM values, adjust frame sizes or bitrates, or swap Discord's Opus encoder with a custom one. This is how audio-focused hooks accomplish effects like extreme loudness, distortion, or synthetic audio—by taking over the audio path before packets are created and sent.

Why Analyze Rivals Hook?

Rivals Hook is weak and largely copied, making it good for basic reverse engineering. it is made by an Indian skidder "Mercy", making it easier to reverse engineer.

THE TOOLKIT

IDA Pro: Why It’s the Industry Standard (Disassembler vs. Decompiler)

IDA Pro is seen as the standard in reverse engineering because it provides reliable, analyst-controlled insights into how machine code works. At its core, IDA is a disassembler. It converts raw binary instructions into readable assembly, reconstructs control flow, and shows how code paths connect across a program. This low-level visibility is crucial when analyzing real software, especially for features like hooks, license checks, and obfuscated logic.

A decompiler, like Hex-Rays’ decompiler add-on, generates high-level pseudocode that resembles C. This is helpful for understanding, but decompilers tend to make assumptions about types, structures, and control logic. They can be incorrect or misleading, especially with complex or non-standard code. Professionals use decompiled output as a reference, not as the definitive source.

IDA’s disassembler, on the other hand, provides precise details about every instruction and branch, showing the exact commands the CPU will run. This makes it better for understanding protections, bypassing checks, and reconstructing logic in a reliable way.

IDA is also very extensible, scriptable, and supports many instruction sets. Its robustness and long history explain why it continues to be the preferred tool for serious reverse engineering.

IDA Pro is a paid program, but here is a link for you:
https://getintopc.com/softwares/utilities/hex-rays-ida-pro-2025-free-download/

THE THEORETICAL FOUNDATION

Reverse Engineering Basics

Assembly Language Primer (x86/x64)

At the lowest level, the code you're analyzing consists of CPU instructions. For this project, only a few are important:

MOV: This instruction copies data between registers and memory. It appears everywhere—in initializing values, passing arguments, and storing results.

CALL: This instruction transfers execution to a function and pushes a return address. License checks usually begin with a CALL to a verification routine.

JMP / Jcc: These instructions change control flow. Conditional jumps (JE, JNE, JZ, JNZ, etc.) are essential because they determine success or failure paths.

CMP / TEST: These instructions compare values and set CPU flags. They don't change data; instead, they prepare the condition for a jump that follows.

RET: This instruction returns from a function. It's often paired with a return value in a register (EAX/RAX), which checks rely on.

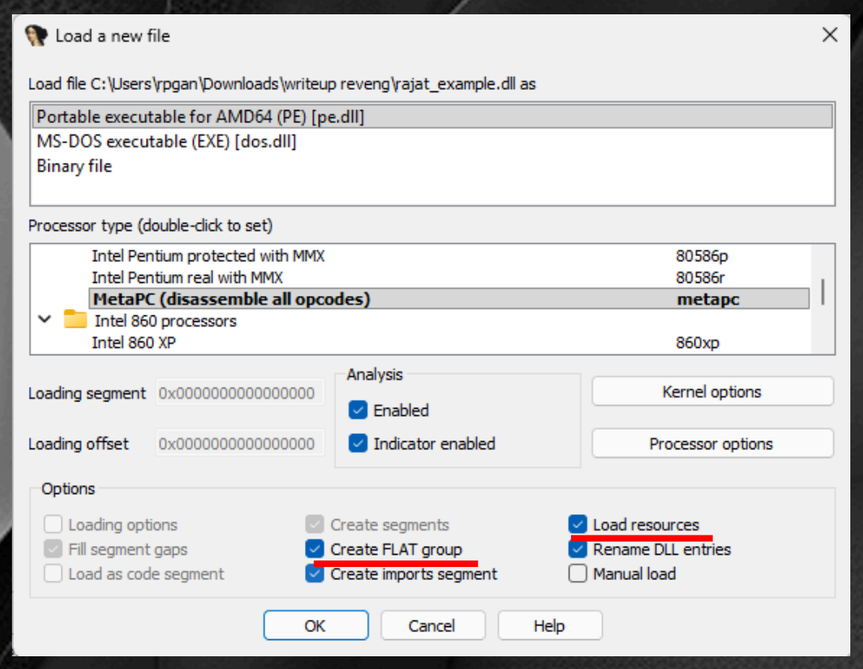
In practice, a license check involves: compute something, compare it, and jump based on the result. Once you grasp this pattern, most protections become clear.

Static vs. Dynamic Analysis

Static analysis means examining the binary without running it. You inspect instructions, strings, cross-references, and control flow to understand the program's structure. This helps you identify license-related functions, decision points, and failure or success paths.

Dynamic analysis means running the program while observing it—using a debugger, breakpoints, or logging—to see its behavior at runtime. This confirms assumptions about which functions are called, what values are returned, and which branches are taken.

For this write-up, static analysis is the main focus, so IDA Pro is the primary tool. Rivals have no strong protection, so there's no need to rely heavily on runtime techniques. IDA simplifies the mapping of control flow, spotting CMP/JMP patterns, and identifying where execution can be redirected. Dynamic analysis is used solely to validate findings, not to discover them.



THE "ENTRY" PHASE

Initial Reconnaissance

Loading the Binary in IDA Pro

When you load the binary into IDA Pro, the first step is to select the correct file format and processor architecture. For a Windows DLL or EXE, IDA should recognize it as a Portable Executable (PE) for AMD64, with the processor set to MetaPC. This setup is necessary for accurate x86/x64 disassembly. Choosing the wrong processor will break the analysis right away, so it's important to confirm this first.
Two loader options are essential and should always be turned on. First, you must check Create FLAT group. This forces IDA to represent the program using a flat memory model, which matches how modern Windows binaries actually run. A flat memory layout makes it easier to analyze control flow, track addresses, and understand function relationships. This is especially important when following execution paths related to hooks or license checks.
Second, you need to enable Load resources so IDA can parse the .rsrc section. Hooks and modified binaries often store useful data there, such as strings, configuration blobs, embedded payloads, or license-related information. Ignoring resources can lead to missing important logic or clues before the analysis even begins.

Once loading is complete, IDA generates import and export segments and starts auto-analysis. This is where the initial assessment begins.

String Analysis

One of the first manual tasks is to check the Strings window. Searching for clear indicators like "Success," "Error," "License," "Invalid," or "Discord" points to code paths related to validation, feature gating, or hook initialization. By cross-referencing these strings, you can often find the functions that determine whether the hook will enable itself. SHIFT + F12 is the default shortcut for viewing strings.

Import / Export Tables

Next, the import table shows which Windows APIs the hook depends on. Functions like InternetOpen, InternetReadFile, GetSystemTime, GetTickCount, or hardware-identification APIs suggest license checks, online verification, or time-based restrictions. Exports, if available, reveal entry points meant for external calls, which can also expose initialization or verification routines.

THE DEEP DIVE

Identifying the "Gatekeeper"

Finding the Main Logic (Strings → Xrefs)

Once you find a relevant string like "License Invalid" or "Invalid key," it acts as an anchor point. In IDA, you don’t guess; you trace backward. Viewing cross-references (Xrefs) to the string shows which functions reference it. One of these references will be the function that handles the failure case. From there, you trace up the call stack: identify where that function gets called from, and what condition led there. This process helps you move from a user-facing error message back to the actual license decision logic.

In less reliable software like Rivals, this path is usually short. The string is often referenced directly in the verification function or in a simple wrapper around it, making the core logic easy to find.

Graph Mode Analysis

Once you enter the suspected function, Graph View makes the logic clear. Instead of reading linear assembly, IDA displays basic blocks connected by branches. You can quickly spot decision points: comparisons, conditional jumps, success paths, and failure paths. One branch leads to normal hook initialization, while the other leads to the error string or an early exit. This visual layout helps you understand how the software “decides” if the license is valid.

For beginners, Graph Mode eliminates guesswork. You see the flow instead of having to mentally reconstruct it.

Identifying the Critical Jump

At the core of the license check, there is usually a `CMP` or `TEST` followed by a conditional jump like JZ, JNZ, JE, or JNE. This jump acts as a gate. One direction means “license valid,” and the other means “fail.” In Rivals, this jump directly decides whether the hook enables itself or exits with an error.

Once you identify this branch, the entire protection becomes clear conceptually. You’ve found the single instruction that determines success versus failure—the critical point. Understanding this is the goal of this section, not brute force or patching.

THE ANALYSIS

Analyzing the Pseudocode

Decompilation (Hex-Rays Pseudocode)

IDA’s decompiler produces C-style pseudocode, which makes it easier to understand low-level logic. You can invoke it by pressing F5 on the currently selected function. This doesn't replace disassembly, it offers a convenient view on top of it.

Variable Renaming

By default, IDA assigns generic names like v1, v2, result, and sub_140012340. Keeping these names slows down your work and obscures intent. Renaming is essential for analysis.

As you figure out what a variable or function actually means, you can rename it:

v3 becomes is_authorized

sub_14001A2F0 changes to verify_license

v7 is now server_response

After renaming, the pseudocode no longer appears as random logic; it starts to resemble an actual program. This process helps you create a mental model of what the software does. In simple hooks like Rivals, proper renaming often reveals the “protection” clearly.

Logic Mapping

After decompilation and renaming, break the logic down step by step. For Rivals Hook, this typically involves tracing how a value is generated or received (such as a hardcoded key, time value, or basic server response), how it’s compared, and the boolean result it creates.

Most skidded hooks use very simple logic: a fixed comparison, a weak checksum, or a basic server handshake that blindly trusts the return value. Once you map this out, you can see exactly where the success or failure decision occurs and how little keeps the hook from being “locked” or “unlocked.”

At this stage, the protection is no longer mysterious; it simply involves control flow.

THE PATCHING STRATEGY

Multiple Patching Strategies

The “NOP” Approach

The NOP approach is a simple method for patching. A NOP (0x90) instruction does nothing and just moves to the next instruction. Conceptually, this means disabling logic by replacing important instructions, like a conditional branch or a function call, with instructions that don't have any impact. As a result, the program "falls through" as if the check never happened. From an analysis viewpoint, this shows how weak a protection can be when removing just a few instructions can make the whole safeguard vanish.

The “Inverse” Approach

The inverse approach focuses on the decision itself. Most license logic comes down to a comparison followed by a conditional jump—success in one direction and failure in the other. By flipping the condition conceptually, such as treating a failure condition as success, the execution is forced down the “valid” path every time. This teaches an important lesson: when protection relies on a single branch instruction, the whole system depends on one boolean decision.

Code Injection / “Hooking the Hook”

Instead of changing the binary directly, another set of techniques involves intercepting execution from the outside. This is often done by injecting a custom DLL that hooks into the hook’s own functions at runtime. Rather than altering instructions on disk, the execution is redirected in memory so that critical functions return controlled values.

Whether this is needed depends on complexity. For weak, client-side-only logic like Rivals, injection is usually excessive. But conceptually, it demonstrates the same main point: once you control the execution flow, it doesn’t matter how the software is meant to behave—only what it is instructed to do.

THE IMPLEMENTATION

Process & Checking

Applying the Patch (Conceptual Overview)

Once the critical decision point is identified, meaning the conditional branch or check, “applying a patch” simply means changing the execution flow so the failure path is never taken. This can happen by modifying instructions in the binary or by changing behavior at runtime. Tools like disassemblers and hex editors allow analysts to experiment by changing instructions and seeing how the control flow shifts. In weak software, even small changes at a single decision point can show just how fragile the protection is.

The main takeaway is not the patch itself, but the lesson: when enforcement happens on the client-side and focuses on one branch, control flow equals control.

Testing the Result

Testing focuses on checking behavior, not just confirming “it didn’t crash.” After changes, the analyst looks to see if the hook reaches code paths that were previously restricted, such as initialization routines, audio interception logic, or feature enablement. If the program behaves the same way as a “licensed” state without triggering error handling, the analysis is confirmed.

This step emphasizes that reverse engineering is about confirming hypotheses, not just making random changes to the bytes.

Troubleshooting Crashes or Failures

If changes lead to crashes, it usually signals that assumptions were incorrect. Common reasons include:

- Breaking control flow by removing necessary instructions
- Incorrect stack or register state
- Hidden integrity checks that catch unexpected execution paths
- Simple anti-debug or anti-tamper logic reacting to changes

When this happens, the right response isn’t brute force; it’s to re-evaluate the logic, comprehend why the code exists, and adjust the understanding accordingly. Even simple software can fail if you take out logic without recognizing its dependencies.

The point of this section is to show that reverse engineering involves reasoning and validation, not just patching things until they work.

THE CONCLUSION

Conclusion & Cracking Mercy’s Hook

This write-up described a basic reverse engineering workflow using Rivals Hook as a weak target. The process began with loading the binary into IDA Pro, making sure to use a flat memory model and load resources for full visibility. String analysis and import inspection quickly identified license-related code paths. Cross-references connected user-facing error strings to the functions that handled authorization decisions. Graph View made the decision flow clear, showing the single conditional branch that determined success or failure. Decompilation and renaming variables clarified the intent, making vague logic easy to read and understand. The key takeaway is straightforward: when protection depends on the client side, is linear, and is poorly designed, basic static analysis can fully uncover and conceptually defeat it.

Anti-Tamper Mechanics: What Could Have Been Improved
Rivals struggles because it relies solely on trust in the client and clearly lays out its logic. Developers have known methods to make this more challenging. Commercial protections like VMProtect virtualize key code paths, so the original logic does not appear as simple x86/x64 instructions. Obfuscation techniques such as control-flow flattening, opaque predicates, and string encryption increase the difficulty of analysis by disrupting the patterns reverse engineers depend on. Basic integrity checks can spot modified code paths, and simple anti-debug techniques can hinder naive dynamic analysis.

More importantly, real enforcement should not depend on a single conditional jump. Distributing checks, continuously validating state, and avoiding clear branches for success or failure makes bypassing harder. Even then, client-side protections ultimately serve as delays, not guarantees.

Rivals shows the opposite: no obfuscation, no integrity checks, and no resistance. That’s why the analysis remains basic and the crack is easy to achieve from a reverse engineering perspective.

The next few pages is where we’ll be live cracking mercy’s hook (rivals) using the methods explained.

CRACKING RIVALS

Cracking Mercy’s Hook (Applying the analysis)

This section shifts from theory to practical use. Everything we've covered so far —assembly basics, control flow analysis, string tracing, graph mode, and logic mapping—now comes together in one clear example. Rivals is selected because it offers no real protection, which makes it perfect for showing how these techniques lead to actual results.

The focus here is not on finding something new but on applying what we already know. The license system in Rivals has been fully mapped: where the decision is made, what condition affects it, and how that decision controls the hook’s functionality. With this understanding, the “crack” becomes a simple result of control flow manipulation instead of a complicated exploit.

This section describes the logic at a basic level:

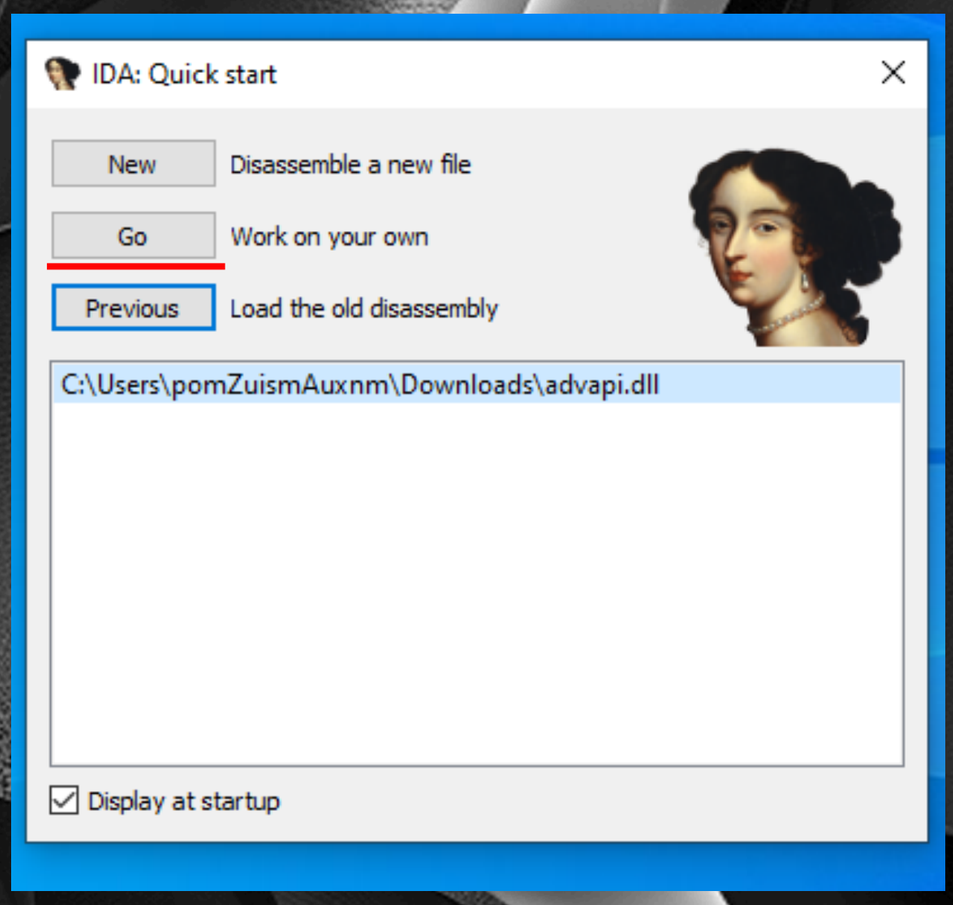
- Identifying the exact decision point that grants authorization
- Showing how execution differs between success and failure paths
- Demonstrating that changing this decision forces the program into a constant “authorized” state
- Verifying that the hook starts and works without triggering the original restrictions

No advanced reversing, no special tools, and no clever tricks are needed. Rivals fails because its protection is weak, centralized, and entirely client-side. Once the decision logic is revealed, the result is certain.

The main purpose of this page is educational: to show that reverse engineering is about understanding, not brute force. When protections are poorly designed, even simple analysis can break them down. Rivals isn’t cracked through attacks —it’s cracked because it was never protected in the first place.

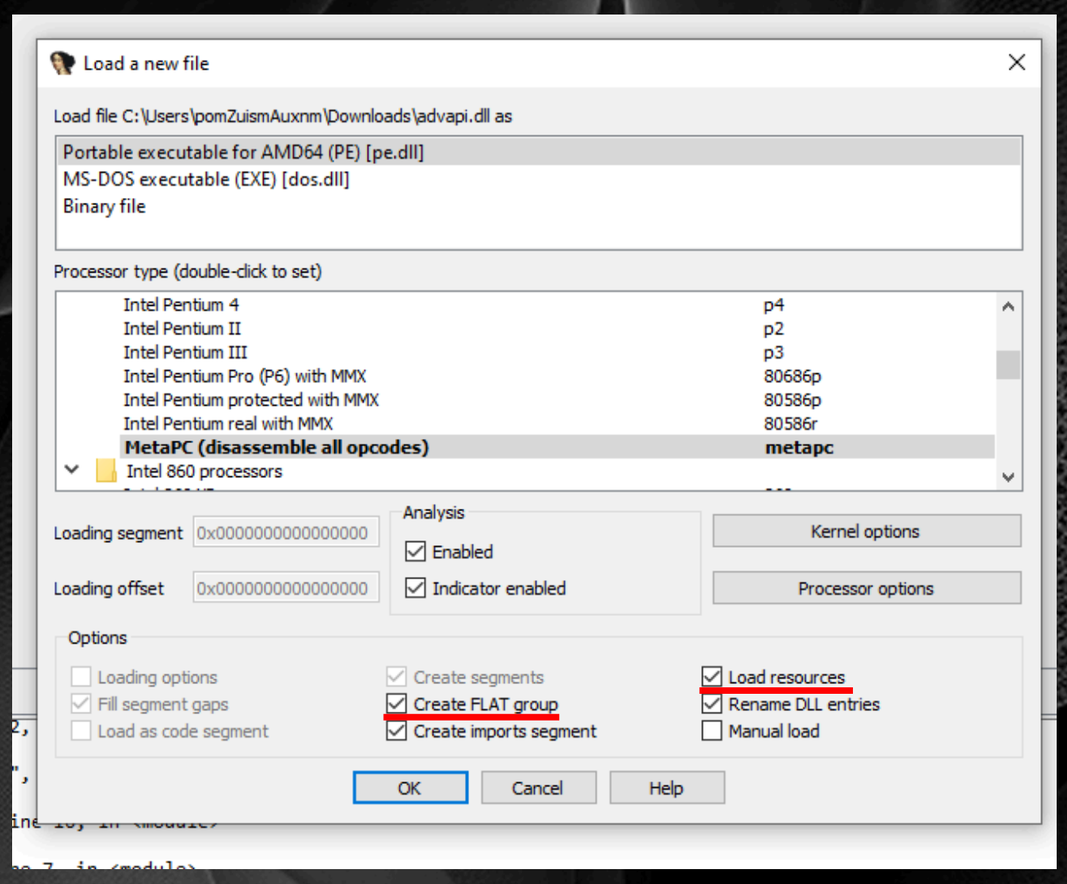
Opening the Binary in IDA Pro:

Start by launching IDA Pro 9.1. Once IDA is open, click the “Go” option (as shown in the image) and drag-and-drop the Rivals Hook DLL into the window. This is the standard entry point for loading a binary and letting IDA begin format detection.

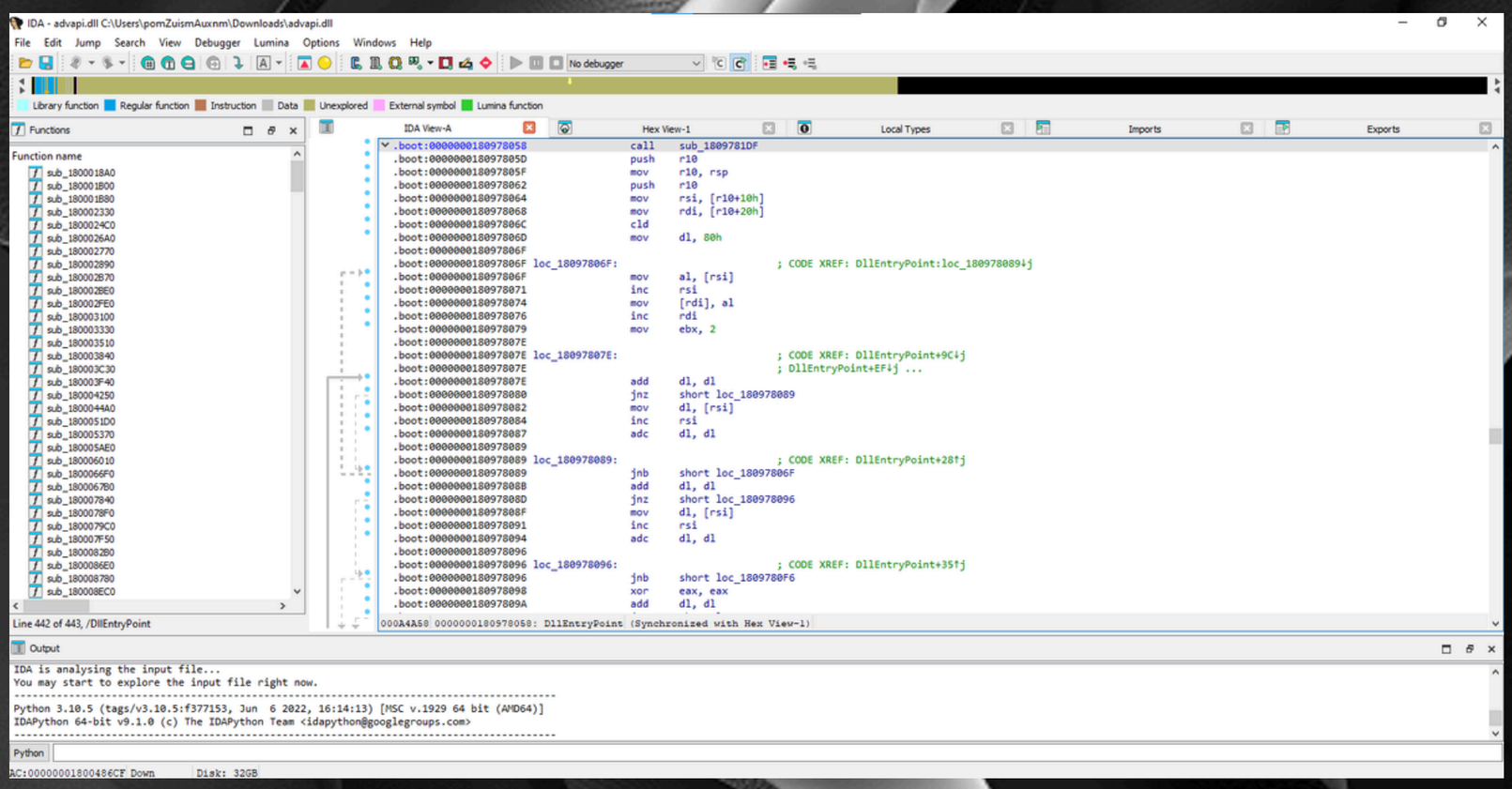


Loader Configuration Pane:

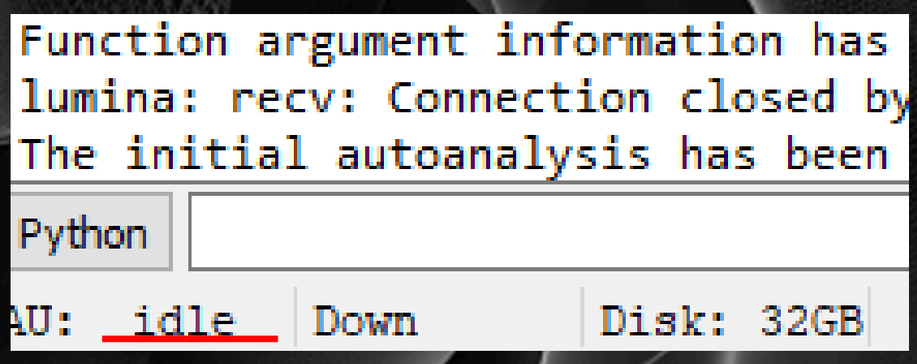
After dropping the file, a loader configuration pane appears. In this window, you must enable Create FLAT group and Load resources, exactly as shown in the image. The reasoning behind these options has already been covered earlier in the write-up, so this step is about applying that knowledge. Once both options are checked, click OK to continue.



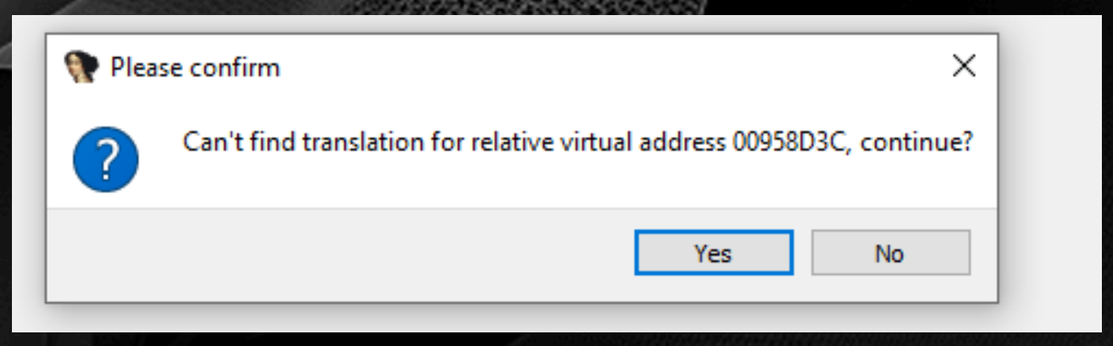
After this, the main IDA workspace opens. You'll see multiple panes such as the IDA View (disassembly), Functions window, Hex View, Imports, Exports, and other analysis windows, exactly as shown in the image. This layout is the primary working environment where all further analysis takes place, including static inspection, cross-references, graph mode, and decompilation.



At the bottom of the IDA window, the status bar indicates the state of analysis. When it shows "Idle", it means IDA has completed auto-analysis and the DLL is fully loaded and processed. This is the point at which manual analysis should begin. Working before the status reaches "Idle" can result in missing functions, incorrect control flow, or incomplete cross-references, so waiting for this state is mandatory before moving forward.
The rivals DLL is pretty small so the auto-analysis would be pretty quick.



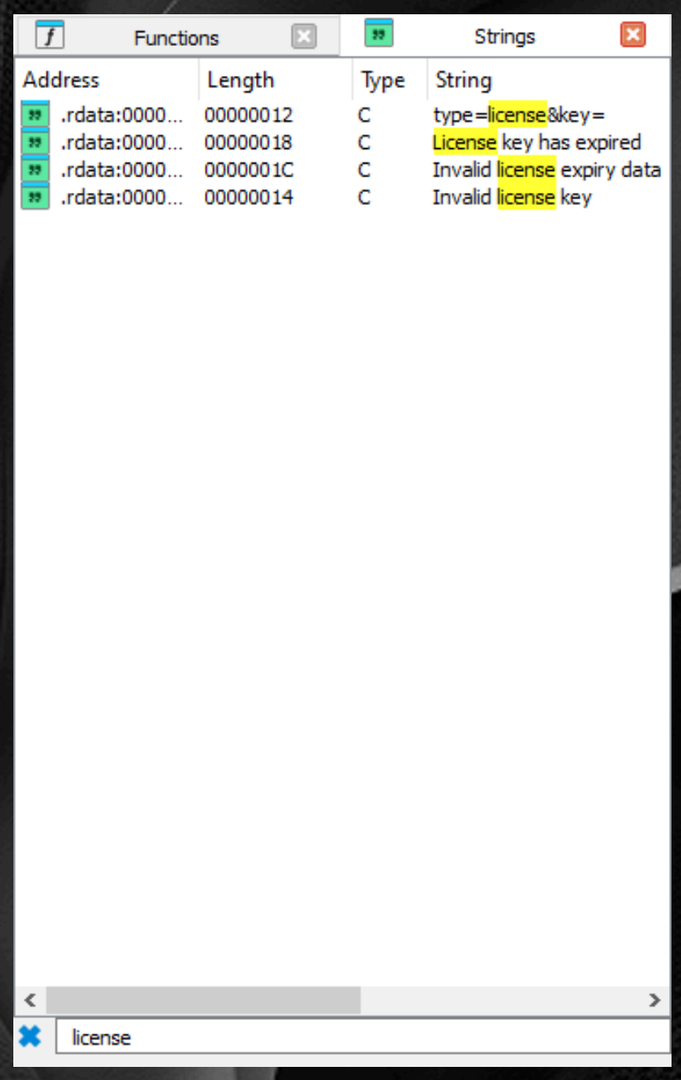
By the way, if any window like this shows up: "Can't find translation for relative virtual address" just click on yes, the likely cause to it is the program being packed / protected.

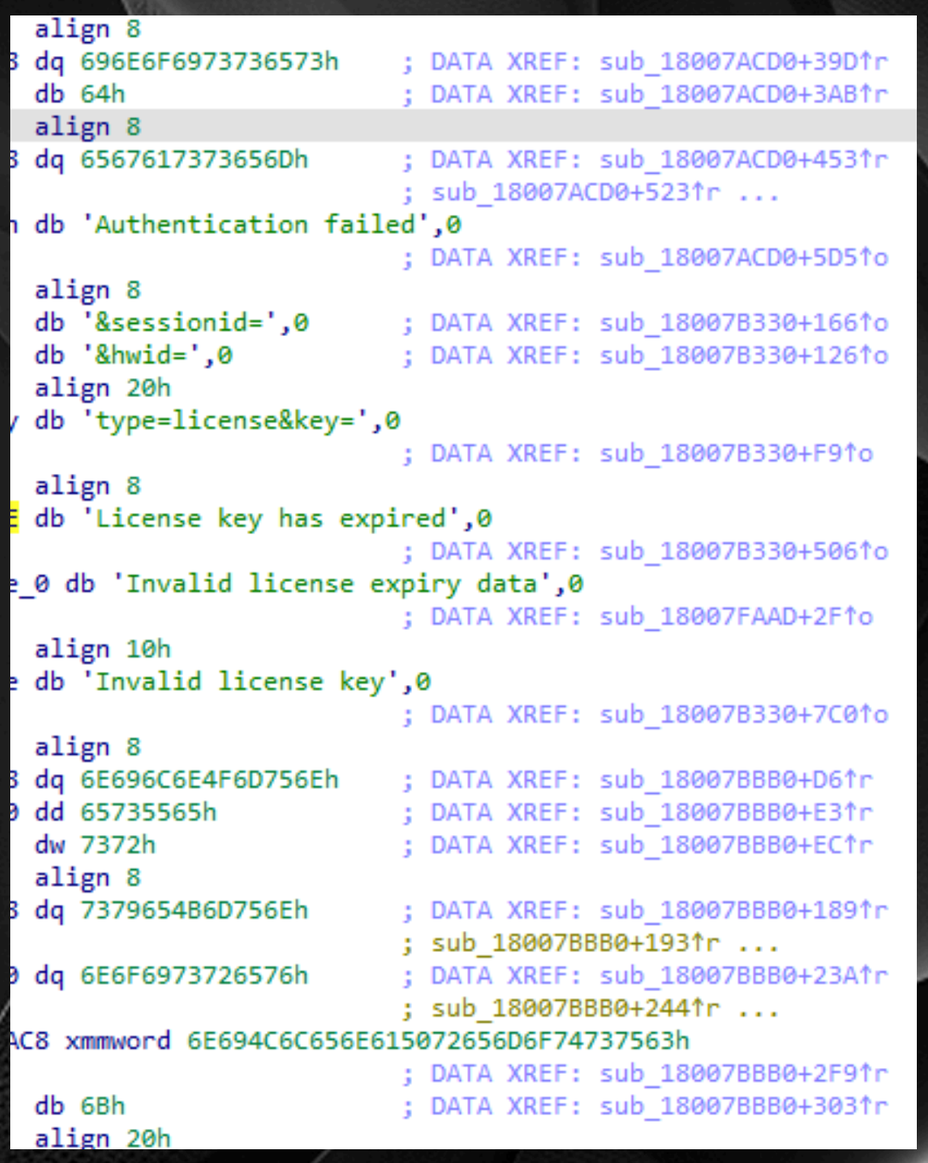


Now that the DLL is fully loaded and IDA's analysis is complete, the next step is string analysis. This is usually the fastest way to locate validation logic in weak binaries. Press Shift + F12 to open the Strings window. This view lists all human-readable strings extracted from the binary. Inside the Strings window, press Ctrl + F to search.

Before doing this, the program was run once with a random license key to observe its behavior. Since it returned an error message stating "Invalid license key", that string becomes an obvious entry point. Searching for the keyword "license" immediately narrows the scope of analysis.

In this case, the search returns four occurrences, as shown in the image. Each occurrence represents a potential reference to license-related logic.





Looking at the Xrefs for the license-related strings makes the situation clear very quickly. IDA shows that the strings "type=license&key=", "License key has expired", and "Invalid license key" are all referenced from the function sub_18007B330, which immediately marks it as the primary license-handling routine.

One of the strings is also referenced by sub_18007FAAD, so that function was checked as well. Opening it in the decompiler with F5 shows the following behavior: it zeroes a few bytes, writes the string "Invalid license expiry data" into a buffer, and returns 0. There is no validation logic, no branching, and no meaningful decision-making—just error state setup. In other words, this function is a dead-end error handler, not the actual license check.
Because of that, sub_18007FAAD can be safely ignored. It does not determine whether the license is valid; it only reacts after a failure has already occurred.

That leaves sub_18007B330 as the real point of interest. It is the only function that references multiple license-related strings and is therefore responsible for constructing the request, evaluating responses, or making the authorization decision. From here onward, analysis should focus entirely on sub_18007B330, as this is where the actual license logic lives and where the critical decision flow will be found.

Analyzing the Main License Function (sub_18007B330)

Opening sub_18007B330 in the decompiler immediately confirms it as the core license routine. It is large, stateful, and references every license-related string found during string analysis—clear proof this is the correct target.

At a high level, the function performs three tasks:

1. License Request Construction

The first portion builds a license request string by concatenating parameters such as:

- type=license&key=
- HWID
- session ID
- username
- owner ID

All values are pulled directly from internal structures (a1, a2) and combined into a single request buffer. This shows that Rivals relies entirely on an external license server and blindly trusts its response.

2. Response Handling

After sending the request, the function parses the response by searching for strings like "success":true, implying a JSON-like format. Authorization is determined purely by string comparisons—there is no cryptographic verification or signature checking.

This is a major weakness: license validity is reduced to the presence or absence of specific text.

3. Authorization State Control

The final section updates internal flags that control whether the hook is enabled.

- On failure, it emits messages such as “Invalid license key”, “License expired”, or “Failed to connect to authentication server” and disables the hook.
- On success, the same flags are set to allow normal execution.

Everything funnels into a small set of boolean branches. There is no defense-in-depth, no redundancy, and no integrity checks tying this logic to the rest of the binary.

```
MEMORY[0x9E30E](&v50);
v50 = &unk_180093890;
v57 = 0;
v58 = 0;
v4 = sub_18007C770(&v49, "type=license&key=");
v5 = a2;
if ( a2[3] > 0xFu )
  v5 = (_QWORD *)*a2;
v6 = sub_1800790A0(v4, v5, a2[2]);
v7 = sub_18007C770(v6, "&hwid=");
v8 = v46;
if ( v47 > 0xF )
  v8 = (_QWORD *)v46[0];
v9 = sub_1800790A0(v7, v8, v46[2]);
v10 = a1 + 88;
v11 = sub_18007C770(v9, "&sessionid=");
if ( (unsigned __int64)a1[91] > 0xF )
  v10 = (_QWORD *)*v10;
v12 = sub_1800790A0(v11, v10, a1[90]);
v13 = sub_18007C770(v12, "&name=");
v14 = a1;
if ( (unsigned __int64)a1[3] > 0xF )
  v14 = (__int64 *)*a1;
v15 = sub_1800790A0(v13, v14, a1[2]);
v16 = a1 + 4;
v17 = sub_18007C770(v15, "&ownerid=");
if ( (unsigned __int64)a1[7] > 0xF )
  v16 = (_QWORD *)*v16;
sub_1800790A0(v17, v16, a1[6]);
v41 = 0;
si128 = _mm_load_si128((const __m128i *)&xmmword_180093E50);
LOBYTE(v41) = 0;
LODWORD(v62) = 5;
v39 = 0;
v40.m128i_i64[0] = 0;
if ( (v58 & 0x22) == 2 || (v18 = *v55) == 0 )
```

Moving to the Patching Phase

At this point, the relevant control flow has been fully identified. We know which function governs the license state, which conditions determine success or failure, and which execution paths lead to the hook being disabled. With that understanding in place, the next step is patching—modifying the binary so execution consistently follows the authorized path.

Conceptually, patching is about altering program logic after analysis, not guessing or blindly changing bytes. The goal is to neutralize the decision point that enforces the license restriction, either by removing the failing path entirely or by forcing the program into a valid state regardless of input. Because Rivals lacks meaningful anti-tamper protections, this step is straightforward once the correct branch has been located.

It’s important to emphasize that patching is only possible because of the weaknesses identified earlier:

License validation is performed client-side.

Authorization depends on simple conditional checks.

No cryptographic integrity checks protect the code paths involved.

In the next section, these weaknesses are exploited by applying a minimal logic modification to the identified function, demonstrating how the hook’s restrictions can be bypassed once the decision-making branch is under the analyst’s control.

How to crack a Discord Hook

Presented to you by Rajat Balwani

PART 2 with Higher Security Hooks will be coming soon

.gg/balwani

22-01-2026