

Beating XLoader at Speed: Generative AI as a Force Multiplier for Reverse Engineering

: 11/3/2025



Research by: Alexey Bukhteyev

Key takeaways

- XLoader remains one of the most challenging malware families to analyze. Its code decrypts only at runtime and is protected by multiple layers of encryption, each locked with a different key hidden somewhere else in the binary. Even sandboxes are no help: evasions block malicious branches, and the real C2 (command and control) domains are buried among dozens of fakes. With new versions released faster than researchers can investigate, analysis is almost always a (losing) race against time.

- Generative AI flips the balance. Instead of spending days on painstaking manual analysis and writing decryption routines and reverse-engineering scripts by hand, researchers can now use AI to examine complex functions, identify algorithms, and generate working tools in just hours, accelerating the path to decrypted code, strings, and IoCs.
- Check Point Research (CPR) demonstrated a new way to use ChatGPT for malware analysis directly from the web interface. By exporting IDA data and analyzing it in the ChatGPT cloud, we showed that deep static reverse engineering with AI is possible without relying on Model Context Protocol (MCP) or a live disassembler session. This approach not only removes the dependency on local heavy tooling, but also makes the results reproducible, easier to share, and more collaborative across research teams.
- In this research we use a fresh XLoader 8.0 sample to demonstrate how cloud-based static analysis with ChatGPT can be combined with MCP for runtime key extraction and live debugging validation. We documented the time spent on key tasks and included real prompt examples, showing how the workflow progresses from unpacking a fully encrypted binary to recovering hidden C2 domains.

Introduction

XLoader is a widely observed malicious loader with information-stealing capabilities. It first surfaced in 2020 as a rebrand of the FormBook code base, a well-known and capable information stealer, and has since undergone substantial hardening and feature growth. In addition to the Windows variant, its developers also marketed a [macOS build](#), though it appears far less prevalent in the wild.

XLoader is a prime example of malware that is extremely difficult to analyze. It combines several layers of protection: customized encryption with additional mixing steps, encrypted blocks disguised as valid but meaningless assembly code, obfuscated API calls, injections into system processes, and a wide set of sandbox evasion techniques. In addition, XLoader encrypts its network traffic, and hides real C2 addresses among dozens of decoys and fake domains.

An important feature of XLoader is its ongoing development. The authors release new versions regularly, changing internal mechanisms beyond recognition and adding new anti-analysis methods. As a result, previous research quickly becomes outdated. In earlier versions, extracting the configuration required pulling out a few keys using intricate algorithms. At the same time, obtaining the decrypted data only required peeling off two layers of obfuscation and encryption. Version 5 introduced a built-in packer, and in versions 6 and 7 analysts had to work through dozens of chained functions that decrypt each other, extracting intermediate keys at every stage. For someone new to XLoader, the entry barrier has become very high: on top of the analysis itself, extra time is needed for onboarding. By the time one research cycle is completed, the next iteration of the malware may already be out – and if there are significant changes, another time-consuming investigation is required.

When we began this research, XLoader version 8.0 had just been discovered. It seemed the XLoader developers were winning the race. But with the rise of generative models, we asked ourselves: can AI change the rules of the game and help us analyze such complex malware more quickly? To explore this, we applied generative AI assistance in two ways: by directly integrating with our analysis tools through a live

MCP connection, and by leveraging ChatGPT's project and file-upload capabilities to work from exported data. Each approach turned out to have distinct benefits, and together they allow us to solve reverse engineering tasks more effectively.

In this study, we focus on the second approach and show how ChatGPT without MCP can be effectively used for reverse engineering tasks, using one of the latest XLoader samples as an example.

Motivation

To defend against XLoader, it is critical to extract up-to-date Indicators of Compromise (IoCs) from each new version — real C2 domains and URLs, cryptographic keys, and version identifiers. These IoCs feed into detection signatures and help track active campaigns. The primary way to obtain IoCs is by extracting and decrypting the malware's configuration data from samples.

The challenge is that XLoader's constantly shifting tactics break automated extraction tools and scripts almost as soon as they're developed. The malware authors frequently tweak encryption schemes and packing methods specifically to thwart these efforts. An automated config extractor that worked yesterday might fail today, meaning each major version demands a fresh reverse-engineering cycle.

Sandboxes offer little relief:

- **Aggressive evasion:** XLoader checks for signs of virtual machines and analysis tools. If it detects them, the malicious branch may never run at all.
- **Just-in-time decryption:** Critical functions and data remain encrypted in memory until just before use and revert to encrypted form shortly afterward. A sandbox might never capture them in their decrypted state. Even if you manage to dump the process memory in a sandbox, you often end up with an incoherent snapshot: pieces of encrypted and decrypted data jumbled together, missing whatever wasn't in memory at that exact time.
- **Useless memory dumps:** When trying to get a memory dump at the exact moment of calling certain API functions, such as `NtResumeThread` or `NtAllocateVirtualMemory`, you only get almost completely encrypted code.
- **C2 camouflage:** The malware's real command-and-control domains are obscured among many decoys. Captured network traffic is usually incomplete and contains a lot of noise.

In short, a sandbox does not solve the problem. It does not provide a reproducible dump or a complete set of IoCs.

The most reliable method is still static analysis: unpack everything, function by function, decrypt the config, and extract the IoCs. The downside is that doing this manually for each new version is slow and painstaking. This is where we hoped generative AI could act as a force multiplier.

Two approaches to AI-assisted analysis

In recent months, many reverse engineers began integrating LLMs with IDA Pro via the Model Context Protocol (MCP) to create an AI-assisted workflow. This agentic approach allows a model to interface directly with the disassembler and debugger, but it has its own practical challenges. For example, some MCP client setups lack certain ChatGPT interface features (like Projects or file uploads), and they still rely on maintaining a live IDA session and stable connection.

We explored two complementary workflows to apply GPT-5 to unraveling XLoader:

- **Live MCP integration:** Using MCP, we gave LLM direct access to our analysis tools (IDA Pro, x64dbg, and a VMware instance). This allowed the AI to query the disassembler, inspect memory, and even control the debugger in real time.
- **“Offline” data pipeline with ChatGPT:** We exported the IDA Pro database (disassembly, decompilations, strings, etc.) and the malware binary to the ChatGPT environment. We then asked ChatGPT to perform static analysis on this data and generate, refine, and execute its own Python scripts against the binary, all within its cloud sandbox. No live connection to our tools was needed.

Each approach has its own strengths. MCP offers an agentic, interactive workflow, whereas the offline pipeline provides a self-contained analysis that’s easy to share and reproduce. These approaches aren’t mutually exclusive — you can use both, picking the appropriate tool for each task.

MCP: Agentic analysis with live tools

The idea of hooking an LLM into IDA isn’t new. For example, researchers at Cisco Talos demonstrated an IDA integration with an LLM acting as a “[reverse engineering sidekick](#)”. In our setup, we used MCP to bridge ChatGPT with IDA Pro and also interface with the x64dbg debugger and a VMware virtual machine. This gave LLM a live window into the malware’s execution.

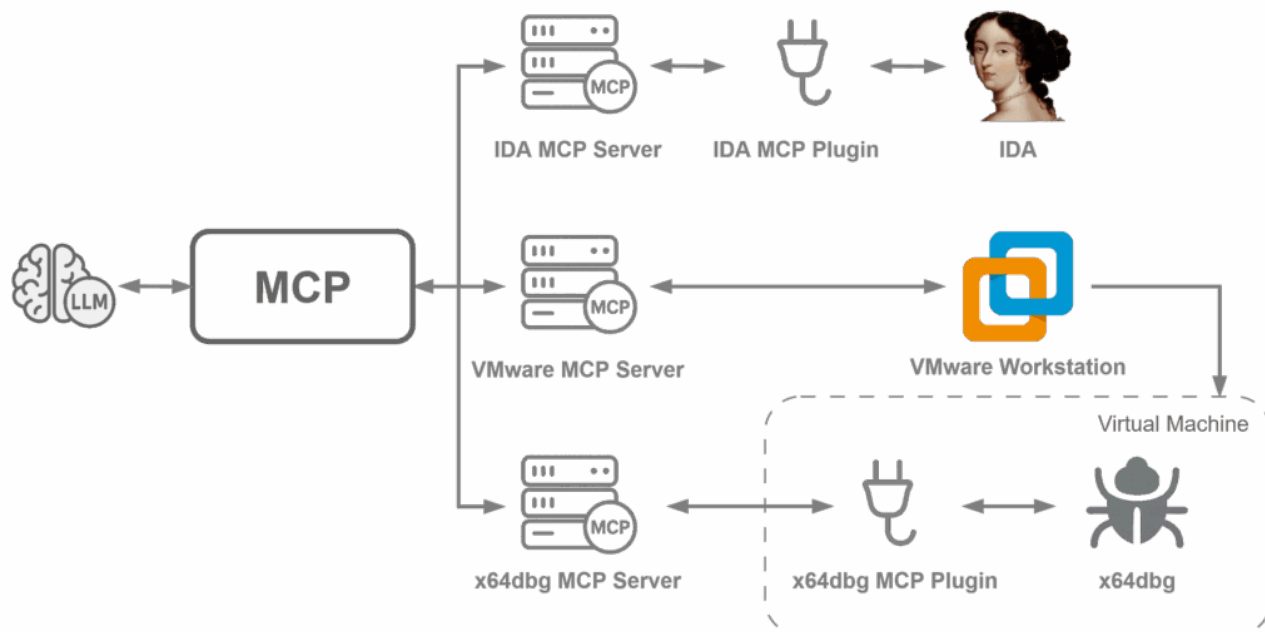


Figure 1 – Integration of an LLM with the reverse engineering environment through MCP.

This live integration, in addition to static analysis and annotating IDA database, enabled the AI to perform these actions:

- **Pull live data on demand.** Set a breakpoint at a critical function to grab a decrypted buffer or a cryptographic key from memory at runtime.
- **Perform “experiment and observe” cycles.** Hypothesize about what a function does, then substitute real runtime data from the debugger and compare the output, adjusting the analysis based on the results.
- **Assist with unpacking in real time.** If the sample self-decrypts or unpacks code, work through those routines, dumping out intermediate values or decrypted code as soon as they appear.

However, the MCP approach isn't without drawbacks:

- **Setup and resource requirements:** This requires a running instance of IDA Pro and other tools. The analyst's machine effectively becomes part of the loop, and that environment must remain up and stable.
- **Single-task focus:** Standard IDA doesn't support multiple independent analyses in one interface. If we want to work on two samples in parallel with AI assistance, we need two separate IDA sessions and MCP connections.
- **Network dependency:** The workflow hinges on a reliable internet connection. A drop in connectivity or an MCP glitch can disrupt the analysis mid-stream.
- **Limited ChatGPT UI features:** When using custom MCP clients with API-based access, we can't utilize some of the conveniences of ChatGPT's own interface, such as long-term project history or easy file management.

For many scenarios, these issues are manageable and the benefits of live interaction outweigh the hassles. Some solutions, such as the [MCP SuperAssistant](#) browser extension, reduce friction by bringing the ChatGPT interface and MCP connectivity together. Recently, ChatGPT introduced a [Developer Mode](#) that can use MCP directly, without third-party plugins. Regardless of whether you use a plugin or the built-in mode, the workflow still depends on a live MCP session tied to a running toolchain and stable connection.

If any of the requirements listed above are difficult to fulfill, for example, you can't keep IDA running constantly, or you need to easily share analysis progress with a colleague who doesn't have the same setup, then a different approach might be preferable. That's why we developed the “offline” data pipeline as an alternative.

Offline IDA export pipeline: reverse engineering with AI in the cloud

Our second approach ditches the live connection entirely. Here AI acts as a self-reliant analyst working from a full static snapshot of the sample.

The workflow is straightforward: we exported everything we could from our IDA Pro database into a structured format (JSON and text). This includes the disassembly and decompiler output of every function,

the list of cross-references, the readable strings, and even the original binary itself. We uploaded the .zip file to ChatGPT.

For example, our export bundle included these files:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

ida_export.zip

- |— meta.json # basic info (sample name, hashes, image base, etc.)
- |— index.json # lookup tables mapping names/EAs to function indices
- |— functions.jsonl # NDJSON: disassembly, xrefs, bytes, prototypes, etc
- |— strings.jsonl # list of strings in the binary and their references
- |— data.jsonl # globals, arrays, named data references
- |— decomp/ # decompiled pseudocode for functions (if available)
- | |— func_or_sub_XXXXXXXXX.c
- | |— func_or_sub_YYYYYYYYY.c
- | |— ...
- |— sample.bin # the malware sample itself

ida_export.zip |— meta.json # basic info (sample name, hashes, image base, etc.) |— index.json # lookup tables mapping names/EAs to function indices |— functions.jsonl # NDJSON: disassembly, xrefs, bytes, prototypes, etc |— strings.jsonl # list of strings in the binary and their references |— data.jsonl # globals, arrays, named data references |— decomp/ # decompiled pseudocode for functions (if available) | |— func_or_sub_XXXXXXXXX.c | |— func_or_sub_YYYYYYYYY.c | |— ... |— sample.bin # the malware sample itself

ida_export.zip

— meta.json	# basic info (sample name, hashes, image base, etc.)
— index.json	# lookup tables mapping names/EAs to function indices
— functions.jsonl	# NDJSON: disassembly, xrefs, bytes, prototypes, etc
— strings.jsonl	# list of strings in the binary and their references

```

└─ data.jsonl          # globals, arrays, named data references
└─ decomp/             # decompiled pseudocode for functions (if available)
  └─ func_or_sub_XXXXXXX.c
  └─ func_or_sub_YYYYYYYY.c
  └─ ...
└─ sample.bin          # the malware sample itself

```

In practice, it is better to upload the archive to a ChatGPT project. Files attached only in the chat can disappear after a session restart, while files in a project stay available for the whole engagement, and can be reused in different chats.

We also wrote an initial prompt explaining how the data is organized and how the AI should format its outputs (for example, proposing new function names and comments in a machine-readable JSON that we could import back into IDA). Essentially, we taught the AI how to read the phonebook we gave it, and how we wanted its notes recorded.

Below is an approximation of our prompt:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

You are my reverse-engineering copilot.

I will upload a ZIP produced by an IDA Pro 9 exporter. It contains:

- meta.json

- index.json

```
```json
```

```
{
```

```
"by_name": { "<funcName>": "0x40XXXXXX", ... },
```

```
"by_ea": { "0x40XXXXXX": <line_index_in_functions_jsonl>, ... }
```

```
}
```

```
...
```

- functions.jsonl (NDJSON; one function per line, with mnemonics/operands already plain text)

```

```json

{

"ea": "0x40XXXXXX",

"name": "func_or_sub_XXXXXXXXXX",

"prototype": "int __cdecl ...", // if available

"ranges": [{"0xstart", "0xend"}], // function address range(s)

"xrefs_in": [{"0x...", "0x..."}], // callers (function start)

"xrefs_out": [{"ea": "0x...", "name": "..."}, ...], // callees (from call sites)

"comments": [{"ea": "0x...", "kind": "...", "text": "..."}],

"bb": [{"start": "0x...", "end": "0x...", "succ": [{"0x..."}], ...}], // basic blocks

"insn": [

{"ea": "0x...", "bytes": "8BEC", "mnem": "mov", "opstr": "...", "size": 2, "cmt": null},

...

],

"bytes_concat": "...", // all function bytes hex, no spaces

"decomp_path": "decomp/<name>_<EA>.c", // if Hex-Rays available

}

...

```

- decomp/*.c // optional

// Optional:

- strings.jsonl // readable strings with code xrefs

- data.jsonl // named globals/arrays

- data_index.json

```

```json

```

```

{

```



```
"by_name": { "g_DomainKeys": "0x40YYYYYY", "var_X": "0x40ZZZZZZ", ... }
}
...
```

- sample.bad // malware sample binary

## On upload (INIT)

1) Parse meta.json & index.json.

2) Stream functions.jsonl just enough to build fast lookups by EA and by name, and to count functions; do NOT eagerly load all decomp/\*.c.

3) Reply with an INIT REPORT:

- file\_name, imagebase, hashes (MD5/SHA256/CRC32), compiler (if present)

- total function count and number with decomp\_path

- confirm you'll use Canvas artifacts for tracking changes (see below)

...

## Live suggestions (function-level, stored on Canvas)

Keep human-readable **suggestions.json** (full JSON, not JSONL) with only proposed renames/comments (no auto-apply).

Schema:

```
```json
```

```
{
```

```
"meta": { "file_name": "<from meta.json>", "imagebase": "0XXXXXXXXX", "input_sha256": "<SHA256>" },
```

```
"changes": [
```

```
{
```

```
"ea": "0XXXXXXXXX", // function start EA (required)
```

```
"name": "sub_XXXXXXXX", // current name (optional precondition)
```

```
"new_name": "ai_better_name", // MUST start with "ai_"
```

"comments": [// only new/changed comments (optional)

{ "kind": "func" | "func_rep" | "anterior" | "repeatable",

"text": "...", "ea": "0xYYYYYYYY", "mode": "set" | "append" }

]

}

]

}

...

...

You are my reverse-engineering copilot. I will upload a ZIP produced by an IDA Pro 9 exporter. It contains: - meta.json - index.json ``json { "by_name": { "<funcName>": "0x40XXXXXX", ... }, "by_ea": { "0x40XXXXXX": <line_index_in_functions_jsonl>, ... } } `` - functions.jsonl (NDJSON; one function per line, with mnemonics/operands already plain text) ``json { "ea": "0x40XXXXXX", "name": "func_or_sub_XXXXXXXX", "prototype": "int __cdecl ...", // if available "ranges": [["0xstart", "0xend"]], // function address range(s) "xrefs_in": ["0x...", "0x..."], // callers (function start) "xrefs_out": [{ "ea": "0x...", "name": "...", ... }, ...], // callees (from call sites) "comments": [{ "ea": "0x...", "kind": "...", "text": "...", ... }, ...], "bb": [{ "start": "0x...", "end": "0x...", "succ": ["0x...", ...], ... }, ...], // basic blocks "insn": [{ "ea": "0x...", "bytes": "8BEC", "mnem": "mov", "opstr": "...", "size": 2, "cmt": null }, ...], "bytes_concat": "...", // all function bytes hex, no spaces "decomp_path": "decomp/<name>_<EA>.c", // if Hex-Rays available } `` - decomp/*.c // optional // Optional: - strings.jsonl // readable strings with code xrefs - data.jsonl // named globals/arrays - data_index.json ``json { "by_name": { "g_DomainKeys": "0x40YYYYYY", "var_X": "0x40ZZZZZZ", ... } } `` - sample.bad // malware sample binary ## On upload (INIT) 1) Parse meta.json & index.json. 2) Stream functions.jsonl just enough to build fast lookups by EA and by name, and to count functions; do NOT eagerly load all decomp/*.c. 3) Reply with an INIT REPORT: - file_name, imagebase, hashes (MD5/SHA256/CRC32), compiler (if present) - total function count and number with decomp_path - confirm you'll use Canvas artifacts for tracking changes (see below) ... ## Live suggestions (function-level, stored on Canvas) Keep human-readable **suggestions.json** (full JSON, not JSONL) with only proposed renames/comments (no auto-apply). Schema: ``json { "meta": { "file_name": "<from meta.json>", "imagebase": "0xXXXXXXXX", "input_sha256": "<SHA256>", "changes": [{ "ea": "0xXXXXXXXX", // function start EA (required) "name": "sub_XXXXXXXX", // current name (optional precondition) "new_name": "ai_better_name", // MUST start with "ai_" "comments": [// only new/changed comments (optional) { "kind": "func" | "func_rep" | "anterior" | "repeatable", "text": "...", "ea": "0xYYYYYYYY", "mode": "set" | "append" }] } }] } `` ...

You are my reverse-engineering copilot.

I will upload a ZIP produced by an IDA Pro 9 exporter. It contains:

```

- meta.json
- index.json
```json
{
 "by_name": { "<funcName>": "0x40XXXXXX", ... },
 "by_ea": { "0x40XXXXXX": <line_index_in_functions_jsonl>, ... }
}
...

- functions.jsonl (NDJSON; one function per line, with mnemonics/operands
already plain text)
```json
{
  "ea": "0x40XXXXXX",
  "name": "func_or_sub_XXXXXXXX",
  "prototype": "int __cdecl ...", // if available
  "ranges": [{"0xstart","0xend"}], // function address range(s)
  "xrefs_in": [{"0x...","0x..."}], // callers (function start)
  "xrefs_out": [{"ea":"0x...","name":"..."},...], // callees (from call sites)
  "comments": [{"ea":"0x...","kind":"...","text":"..."}],
  "bb": [{"start":"0x...","end":"0x...","succ":["0x..."]},...], // basic
blocks
  "insn": [

{"ea":"0x...","bytes":"8BEC","mnem":"mov","opstr":"...","size":2,"cmt":null},
  ...
],
  "bytes_concat": "....", // all function bytes hex,
no spaces
  "decomp_path": "decomp/<name>_<EA>.c", // if Hex-Rays available
}
...

- decomp/*.c // optional

// Optional:
- strings.jsonl // readable strings with
code xrefs
- data.jsonl // named globals/arrays
- data_index.json
```json
{
 "by_name": { "g_DomainKeys": "0x40YYYYYY", "var_X": "0x40ZZZZZZ", ... }

```

```

 }
 ...

- sample.bad // malware sample binary

On upload (INIT)
1) Parse meta.json & index.json.
2) Stream functions.jsonl just enough to build fast lookups by EA and by
 name, and to count functions; do NOT eagerly load all decomp/*.c.
3) Reply with an INIT REPORT:
 - file_name, imagebase, hashes (MD5/SHA256/CRC32), compiler (if present)
 - total function count and number with decomp_path
 - confirm you'll use Canvas artifacts for tracking changes (see below)

...

Live suggestions (function-level, stored on Canvas)
Keep human-readable **suggestions.json** (full JSON, not JSONL) with only
proposed renames/comments (no auto-apply).

Schema:
```json
{
  "meta": { "file_name": "<from meta.json>", "imagebase": "0XXXXXXXX",
"input_sha256": "<SHA256>" },
  "changes": [
    {
      "ea": "0XXXXXXXX",           // function start EA (required)
      "name": "sub_XXXXXXXX",      // current name (optional precondition)
      "new_name": "ai_better_name", // MUST start with "ai_"
      "comments": [               // only new/changed comments (optional)
        { "kind": "func"|"func_rep"|"anterior"|"repeatable",
          "text": "...", "ea": "0YYYYYYYY", "mode": "set"|"append" }
      ]
    }
  ]
}
```
...

```

After it was set up, this pipeline allowed ChatGPT to perform deep static analysis entirely within its own environment. We could ask it to find cryptographic algorithms, trace complex control flows, or even write and

execute a Python script to decrypt some data from sample.bin. Many such tasks can be done without any new information from us – the AI works off the data we provided, verifying its logic by running Python scripts as needed. If there is an error, it fixes the script and reruns the tests, repeating this until the result converges. Compared to our previous approach, all these steps (analysis, code, test, correction) run in a single loop without dozens of local MCP calls. Naturally, this works well when using GPT-5 in the “thinking” mode.

This approach had several clear benefits:

- **No persistent local session needed:** If our IDA crashed or we closed our laptop, it didn’t matter as ChatGPT could access everything it needed from the cloud. We didn’t need to babysit a live connection.
- **Easily repeatable and shareable:** Because the entire state of the analysis was captured in our export, anyone with the archive and the prompt could reproduce the analysis. We can even run multiple ChatGPT sessions on the same data in parallel (to explore different questions or use different prompt strategies) without interference.
- **Better use of ChatGPT’s features:** Inside the ChatGPT interface, we can take advantage of file uploads, persistent chat history, and the editable canvas.
- **Collaboration-friendly:** Several researchers can work on the same sample independently and later merge their results via sharing suggestions.json file, without the need to diff IDBs.
- **Safe script execution:** ChatGPT can prototype and test analysis scripts directly on the “live” sample in a secure cloud environment and deliver a working output to the analyst.
- **Extremely fast onboarding:** No need to set up an MCP server or configure complex integrations. The exported data and results can even be shared with colleagues who don’t have IDA installed and cannot open your IDB.
- **Broad applicability:** The concept isn’t tied to x86 binaries or IDA. It can be adapted to virtually any platform or technology stack. For example, GoLang, .NET, or JavaScript samples can all be analyzed in the same way. The main challenge is properly preparing the data and providing a tailored prompt that explains how to work with it. The analysis process remains the same.

That said, the offline approach isn’t a universal magic wand. There were cases where we still needed to resort to actual debugging (and therefore MCP), for example, to confirm a guessed key or to dump something that our static analysis missed. In addition, while analyzing other malware families, we encountered situations where continuous work in IDA was required, involving constant modifications to the live database. Previously, we would have needed to export the database after every iteration of changes. In this instance, the MCP-based approach turned out to be a better i.e. more convenient alternative.

## What went wrong and how we fixed it

Unsurprisingly, using an AI with an offline IDA export wasn’t without hiccups. We encountered a few issues with AI’s performance and solved them by adding strict rules to the prompt.

- **Guessing missing values.**

Sometimes the model tried to invent missing data, for example, encryption keys that were computed dynamically at runtime. To prevent such “hallucinations”, we enforced an evidence-first rule: every numeric value and every algorithm must be backed by a quote from the export (functions.jsonl, decomp/\*.c, or data.jsonl) with the exact EA address. If the data is not there, the model must produce a not-found report that explains where it looked and why nothing was found.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

## Provenance & no-fabrication

- Any *\*specific\** numeric/structural claim (modulus, key length, magic multipliers like 0x66666667, loop bounds) MUST be backed by direct

evidence from the uploaded data:

- Quote the exact line(s) from `functions.jsonl` (insn/mnem/opstr/bytes) or from `decomp/\*.c`, and cite EA(s).

- If the claim is not literally present, mark it **\*\*UNPROVEN\*\*** and offer a concrete verification plan.

- If you revise a claim, explicitly state what changed and show the new

evidence quote. No silent edits.

## Provenance & no-fabrication - Any *\*specific\** numeric/structural claim (modulus, key length, magic multipliers like 0x66666667, loop bounds) MUST be backed by direct evidence from the uploaded data: - Quote the exact line(s) from `functions.jsonl` (insn/mnem/opstr/bytes) or from `decomp/\*.c`, and cite EA(s). - If the claim is not literally present, mark it **\*\*UNPROVEN\*\*** and offer a concrete verification plan. - If you revise a claim, explicitly state what changed and show the new evidence quote. No silent edits.

## Provenance & no-fabrication

- Any *\*specific\** numeric/structural claim (modulus, key length, magic multipliers like 0x66666667, loop bounds) MUST be backed by direct evidence from the uploaded data:

- Quote the exact line(s) from `functions.jsonl` (insn/mnem/opstr/bytes) or from `decomp/\*.c`, and cite EA(s).

- If the claim is not literally present, mark it **\*\*UNPROVEN\*\*** and

offer a concrete verification plan.

- If you revise a claim, explicitly state what changed and show the new evidence quote. No silent edits.

- **Shaping output to match expectations.**

For example, a string-decryption routine was expected to return printable text, but due to a mistake in extracting the key, the output was corrupted. To make the output “look right,” the model applied Base64. We banned any cosmetic transformations (such as Base64) used just to make results look valid. Instead, the model must find the actual error in the keys or in the algorithm and rerun the tests until the output is correct.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

Verification contract

\* Define acceptance criteria from the task (properties/invariants).

\* Run self-checks (lengths, wrap-around, bounds monotonicity, step counts, round-trip where applicable).

\* Do NOT transform outputs to “look right”; if a check fails, proceed to the recovery loop.

Verification contract \* Define acceptance criteria from the task (properties/invariants). \* Run self-checks (lengths, wrap-around, bounds monotonicity, step counts, round-trip where applicable). \* Do NOT transform outputs to “look right”; if a check fails, proceed to the recovery loop.

Verification contract

\* Define acceptance criteria from the task (properties/invariants).

\* Run self-checks (lengths, wrap-around, bounds monotonicity, step counts, round-trip where applicable).

\* Do NOT transform outputs to “look right”; if a check fails, proceed to the recovery loop.

- **Asking the user for data that is already in the archive.**

Early on, the model sometimes requested data we had already provided. We fixed this with a local-first rule: search in the archive files first. It should produce a not-found report only if the data is truly missing.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

`## Local-first data usage`

- Treat the uploaded dataset as the primary source of truth.
- Before requesting any bytes/strings/keys from the user, attempt to obtain them from the uploaded files
- Never ask the user for blobs that are present in data.jsonl/strings.jsonl or are trivially recoverable from functions.jsonl.
- Only if a needed EA/function is absent from the snapshot, say so and propose next steps (e.g., MCP call).

`## Local-first data usage - Treat the uploaded dataset as the primary source of truth. - Before requesting any bytes/strings/keys from the user, attempt to obtain them from the uploaded files - Never ask the user for blobs that are present in data.jsonl/strings.jsonl or are trivially recoverable from functions.jsonl. - Only if a needed EA/function is absent from the snapshot, say so and propose next steps (e.g., MCP call).`

`## Local-first data usage`

- Treat the uploaded dataset as the primary source of truth.
- Before requesting any bytes/strings/keys from the user, attempt to obtain them from the uploaded files
- Never ask the user for blobs that are present in data.jsonl/strings.jsonl or are trivially recoverable from functions.jsonl.
- Only if a needed EA/function is absent from the snapshot, say so and propose next steps (e.g., MCP call).

With these precautions in place, our AI “assistant” became a reliable analyst for the static portions of the work. In the next sections, we show how it performed on the real challenges within XLoader 8.0, such as decrypting the payload and API resolution and working with occasional MCP-powered dynamic checks.

## GPT-5 in Practice: Analyzing XLoader’s Built-in Crypter

When working with older ChatGPT models such as o3, getting the right result required splitting the task into many small steps and explicitly telling the model what to do, down to pointing out exact code addresses and



the algorithms to apply. Without this level of detail, the output was unpredictable. This approach was closer to “text-based programming” and required deep engagement on our side.

With GPT-5, however, we can pose broader and more abstract tasks. Below we show an example of XLoader’s built-in crypter analysis with a mixed approach: using the IDA export as the main data source, and MCP+x64dbg for result verification.

For this task we took a recently discovered XLoader sample with SHA256: **77db3fdccda60b00dd6610656f7fc001948cdcf410efe8d571df91dd84ae53e1**. For the entire process we used GPT-5 in the “Thinking” mode.

After we gave the AI-assistant the instructions for processing the data, we received a short report:

```
Here's your INIT REPORT ✓

• file_name: sample.bad
• imagebase: 0x00400000
• hashes:
 • MD5: C1315FDEA3AE5C485EE19590FEC9771C
 • SHA256: 77DB3FDCCDA60B00DD6610656F7FC001948CDCF410EFE8D571DF91DD84AE53E1
 • CRC32: 2ABBEAFC
• compiler (best-effort): 129
• IDA Pro version: 9.1

Functions

• Total functions: 26 (streamed from functions.jsonl)
• With decomp_path : 26 (all have Hex-Rays output)

Indexing done

• Parsed meta.json & index.json .
• Streamed functions.jsonl to count functions and confirm decomp_path presence.
• Fast lookups available by name and by EA (from index.json ; ready to pivot into functions.jsonl lines as needed).
```

**Figure 2** – IDA export initial report.

Next, we deliberately formulated the tasks for the AI assistant as if we knew nothing at all about the sample under analysis, assuming this would reflect the actions of someone unfamiliar with XLoader.

The first prompt was written in the most abstract way possible:

Plain text

Copy to clipboard

Open code in new window

Perform an initial analysis of the sample starting from the entry point and provide a short report.

Perform an initial analysis of the sample starting from the entry point and provide a short report.

Perform an initial analysis of the sample starting from the entry point and provide a short report.

Processing this simple prompt took 8 minutes and 46 seconds. As a result, our assistant correctly identified the RC4 implementation and concluded that the sample was packed. It is worth noting that, based only on the data available to the model, it suggested that the sample looked similar to XLoader. At the same time, there was nothing in the archive or in the initial prompt that explicitly pointed to this.

### What this means (short)

- The EP is a stage-0 loader: it derives a key, RC4-decrypts tiny command/strings, resolves exports manually, then marks memory RX and jumps into the newly unpacked/decoded region. The heavy use of opaque arithmetic and stringless resolution is consistent with XLoader stubs.

**Figure 3** – Initial analysis report: Entry point analysis.

In addition, it detected API call obfuscation. While the assistant was not fully able to deobfuscate all API calls during the quick triage, in some cases it inferred the function being called from the context and its signature.

### Memory permissions flip (VirtualProtectEx-like pattern)

- Later, a resolved function in `[ebp+var_60]` is called with 5 args: `(-1 /*current process*/, &addr, &size, 0x40 /*PAGE_EXECUTE_READWRITE*/, &oldProt)` — the classic `VirtualProtectEx` signature shape.
- Proof (call site):

perl

Copy Edit

```
0x00401898 mov eax, [ebp+var_60]
0x004018A9 push ecx ; &oldProt
0x004018AA push 40h ; flNewProtect
0x004018AC push ecx ; &size
0x004018B3 push ecx ; &addr
0x004018BA push 0FFFFFFFh ; hProcess = -1
0x004018BC call eax
```

(exact bytes show at EAs 0x004018A9–0x004018BC).

- Status: API name is UNPROVEN (no cleartext strings), but the 5-arg shape and constants strongly indicate `VirtualProtectEx`.

**Figure 4** – Initial analysis report: Presumed call to the `VirtualProtectEx` function.

It also successfully identified the point where execution is handed over to the decrypted code.

### Hand-off to decrypted/unpacked payload

- Control eventually transfers via additional resolved pointers: `call eax` (above), `call [ebp+var_60]` (protect), then later `call esi` at `0x00401A89`, consistent with tail-jumping into the unpacked module.

**Figure 5** – Initial analysis report: Call to the original entry point in the decrypted code.

At this stage, our priority was to reach the payload as quickly as possible. We therefore focused on this goal by first asking the assistant to find all cryptographic function calls, and then to analyze how exactly the payload was decrypted.

We found out that the main payload block goes through two rounds of RC4: first, an RC4 decryption of the entire buffer, and then a second pass in 256-byte chunks using a different key.

### Memory permissions flip (VirtualProtectEx-like pattern)

- Later, a resolved function in `[ebp+var_60]` is called with 5 args: `(-1 /*current process*/, &addr, &size, 0x40 /*PAGE_EXECUTE_READWRITE*/, &oldProt)` — the classic `VirtualProtectEx` signature shape.
- Proof (call site):

```
perl
0x00401898 mov eax, [ebp+var_60]
0x004018A9 push ecx ; &oldProt
0x004018AA push 40h ; flNewProtect
0x004018AC push ecx ; &size
0x004018B3 push ecx ; &addr
0x004018BA push 0FFFFFFFh ; hProcess = -1
0x004018BC call eax
```

(exact bytes show at EAs `0x004018A9–0x004018BC`).
- Status: API name is **UNPROVEN** (no cleartext strings), but the 5-arg shape and constants strongly indicate `VirtualProtectEx`.

**Figure 6** – Initial analysis report: Description of the two rounds of RC4 encryption.

In addition, the assistant managed to collect:

- The virtual address of the original entry point in the decrypted code (`0x00430CB3`).
- The offset of the encrypted block in the binary (`0x3143`).
- The encrypted blob size (`0x44A00`).

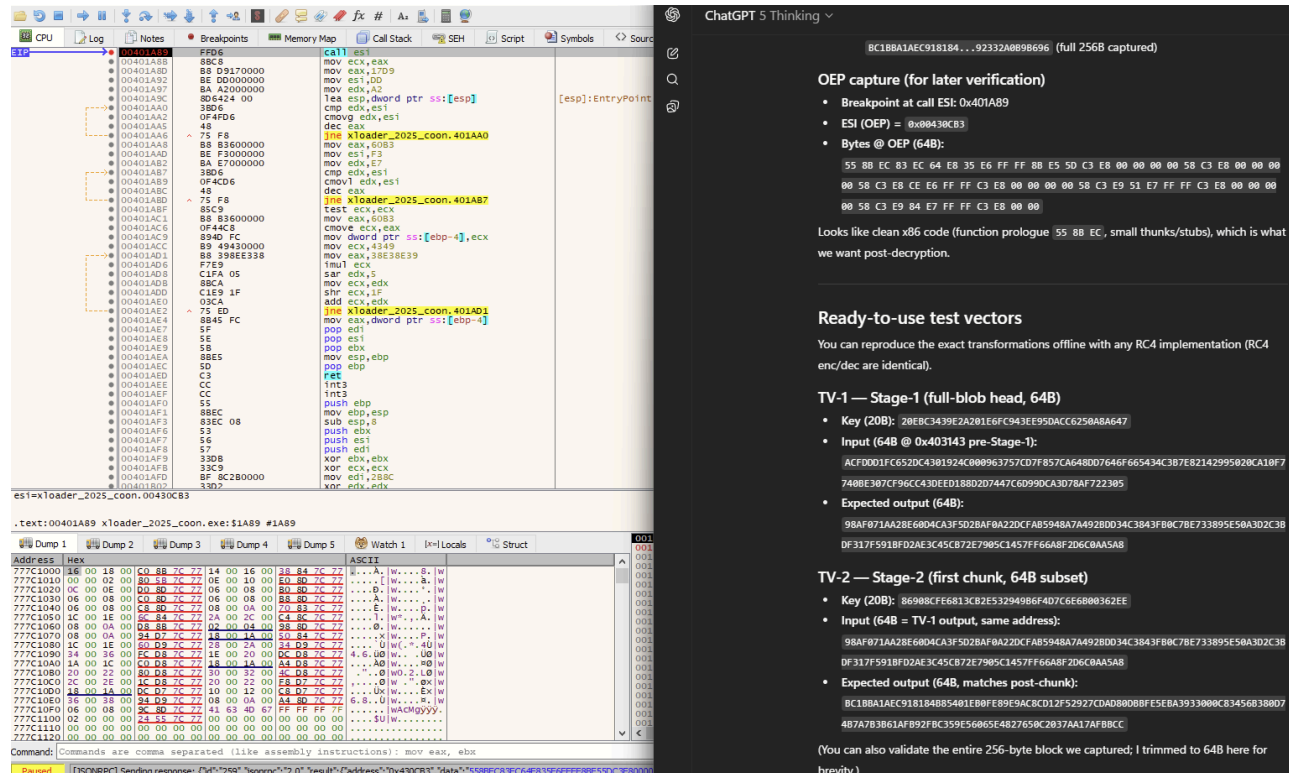
The next step was to obtain the real-time keys and verify the result. At this point, we turned to MCP. In one of the steps, we also asked the assistant to read a section of decrypted data to validate the correctness of the static decryption.

As a result, it obtained the following keys:

Stage-1 Key 20EBC3439E2A201E6FC943EE95DACC6250A8A647

Stage-2 Key 86908CFE6813CB2E532949B6F4D7C6E6B00362EE

It also obtained a section of decrypted code:



**Figure 7 – AI-controlled debugging in x86dbg.**

After the final keys were read from memory, we asked the assistant to identify where and by which algorithms the keys were generated, and to verify the analysis was correct using the real-time data obtained in the previous step.

### Plain text

Copy to clipboard

[Open code in new window](#)

## EnlighterJS 3 Syntax Highlighter

Find how the Stage-1 and Stage-2 RC4 keys are calculated: source key material and algorithms. Please note that ALL the required data is available to you in IDA export.

Check your assumptions using the captured realtime values. Start with the Stage-1 key.

Find how the Stage-1 and Stage-2 RC4 keys are calculated: source key material and algorithms. Please note that ALL the required data is available to you in IDA export. Check your assumptions using the captured realtime values. Start with the Stage-1 key.

Find how the Stage-1 and Stage-2 RC4 keys are calculated: source key material and algorithms. Please note that ALL the required data is available to you in IDA export.  
Check your assumptions using the captured realtime values. Start with the Stage-1 key.

In the end, AI produced a working script that unpacked the analyzed sample. Unfortunately, the script was not universally applicable, as the patterns it used to locate the keys were tightly bound to this particular sample. As a result, it failed when we tried to apply it to samples from other versions, requiring further manual fine-tuning.

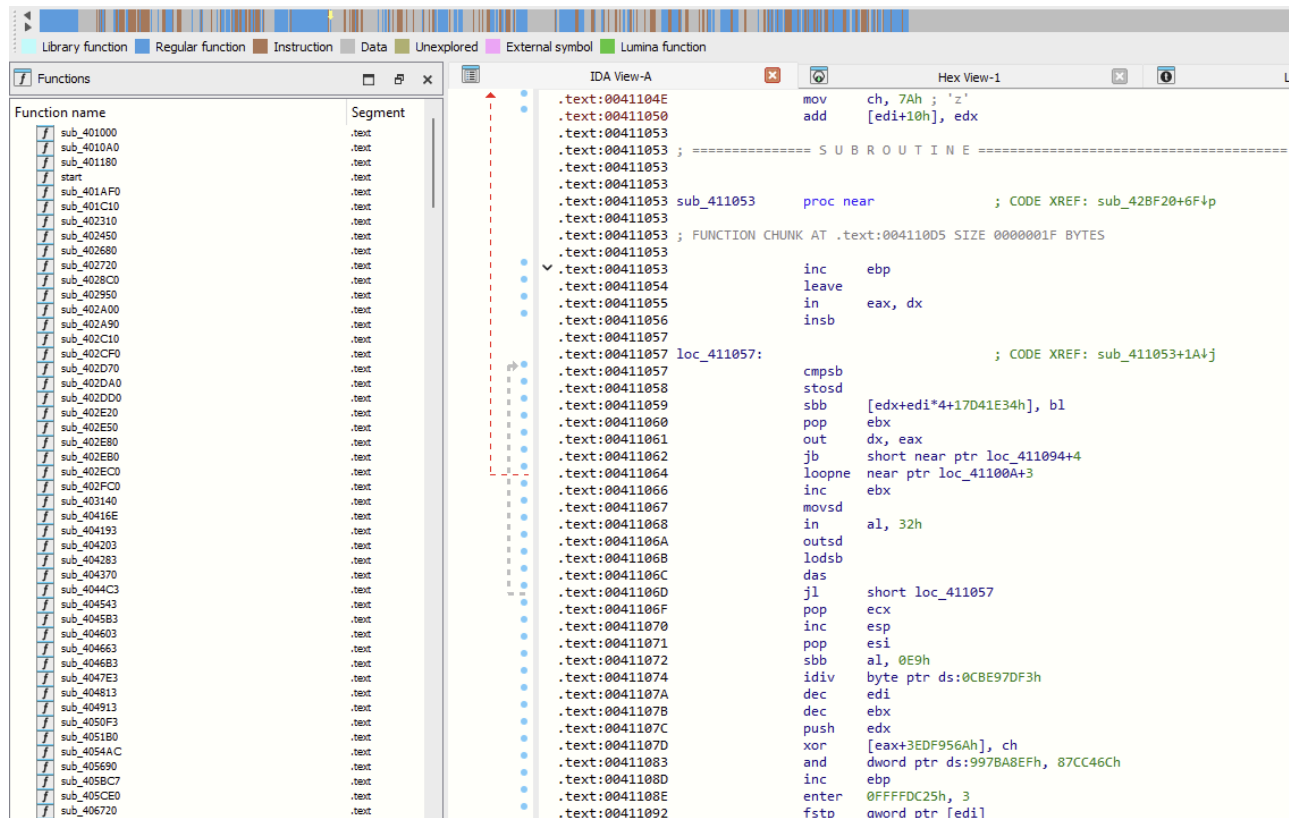
Excluding the final step of creating a generic unpacker, the entire analysis took about 40 minutes and required 39 MCP calls. The table below lists the prompts we used and the time spent on each analysis step.

| Prompt                                                                                                                                                                                                                                                                       | Time consumed | Number of MCP calls |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|---------------------|
| Initial prompt (instructions).                                                                                                                                                                                                                                               | 37s           | 0                   |
| Perform an initial analysis of the sample starting from the entry point and provide a short report.                                                                                                                                                                          | 8m 46s        | 0                   |
| Find and carefully inspect all calls to the found cryptographic functions.                                                                                                                                                                                                   | 4m 13s        | 0                   |
| Analyze main payload blob decryption. Make sure you inspected all touches of the encrypted blob across different functions.                                                                                                                                                  | 4m 13s        | 0                   |
| Set breakpoints on required addresses and capture the required data. Also, set a breakpoint before calling the OEP and capture a small decrypted block at OEP for using it later for the decryption verification. Before start, please provide a plan.                       | 9m 7s         | 39                  |
| Find how the Stage-1 and Stage-2 RC4 keys are calculated: source key material and algorithms. Please note that ALL the required data is available to you in IDA export. Check your assumptions using the captured realtime values. Start with the Stage-1 key.               | 6m 56s        | 0                   |
| Move to Stage-2 key derivation.                                                                                                                                                                                                                                              | 4m 40s        | 0                   |
| Implement an offline reproducer for Stage-1 and Stage-2 keys. Then implement a complete static decryptor that works directly with sample.bin: extracts the encrypted payload from the binary and decrypts it. Verify it works correctly using the captured data (OEP bytes). | 1m 13s        | 0                   |
| Total                                                                                                                                                                                                                                                                        | 39m 8s        |                     |

## Analysis of the unpacked sample

After manually creating a function at address `0x00430CB3` (original entry point: OEP) which we named `oep_start`, we opened the unpacked sample in IDA and applied the export script again. We also created a new project to analyze the unpacked sample.

Even before starting a deep dive, it was clear that IDA failed to recognize a large portion of the code, and many of the identified functions did not look valid.



**Figure 8** – Unpacked XLoader sample in IDA.

This may indicate that the code is obfuscated in some way, or that the functions are encrypted. In fact, we know that XLoader uses on-the-fly function decryption, as we mentioned in the introduction. For some functions, multi-layer encryption is applied.

For the sake of the experiment, we wanted to see if the AI assistant could determine all this on its own, without our guidance. We started the analysis with the same type of prompt we used when analyzing the packed sample.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

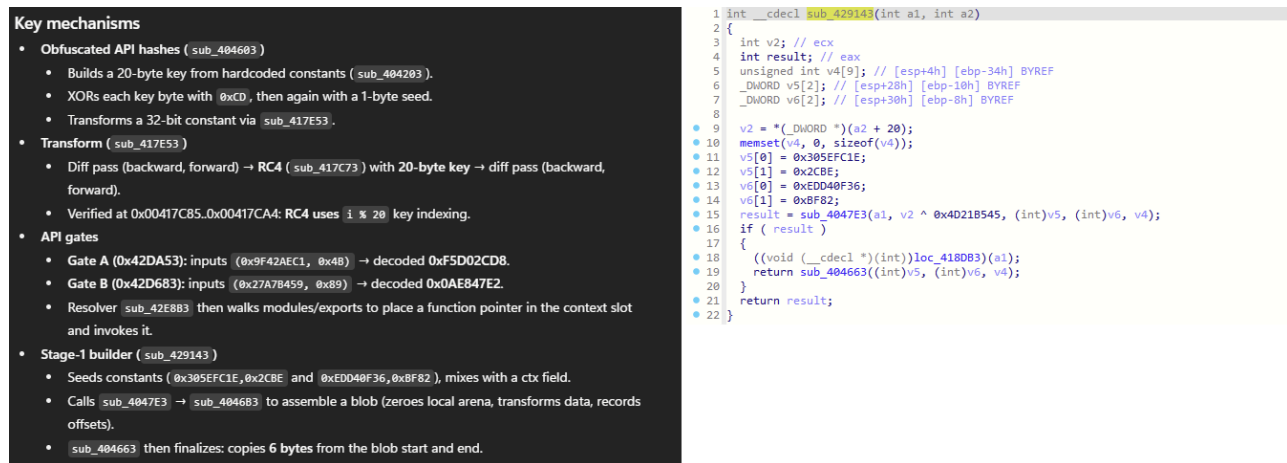
Perform an initial analysis of the sample starting from the `oep\_start` (`0x00430CB3`) and provide a short report.

Perform an initial analysis of the sample starting from the `oep\_start` (`0x00430CB3`) and provide a short report.

Perform an initial analysis of the sample starting from the `oep\_start` (`0x00430CB3`) and provide a short report.

After initial analysis, we identified:

- Use of obfuscated API calls.
- Use of RC4 encryption with additional modifications before and after using RC4.
- Some stage-1 builder (a decryptor of the encrypted function).



**Figure 9** – Triage report and the “Stage-1 builder” function (a function decryptor stub).

## Function decryption scheme I

Next, we asked the assistant to focus on the logic around the so-called **stage-1 builder** (sub\_429143) and to locate cross-references of the functions involved. The AI assistant identified 90 similar functions. These functions derive 6-byte head and tail markers, use those markers to locate a target region in memory, overwrite the markers with six NOP instructions (90 90 90 90 90 90), transform the region, and then transfer execution to a hardcoded address unique to each wrapper.



## 0) Moving parts (names → original)

- Wrapper (per-call): `ai_dec_call_418DB3` (orig `sub_429143`)
- Begin (thin wrapper): `ai_encfunc_begin` (orig `sub_4047E3`)
- Core locate+decrypt: `ai_encfunc_locate_decrypt` (orig `sub_4046B3`)
- Finalize (re-encrypt + edge copy): `ai_encfunc_reencrypt` (orig `sub_404663`)
- Base key filler (20B): `ai_encfunc_key_base` (orig `sub_404193`)
- Transforms:
  - `ai_xform_sub_rc4_sub` (orig `sub_417E53`) → used to decrypt
  - `ai_xform_add_rc4_add` (orig `sub_417ED3`) → used to re-encrypt
  - (RC4 core: `ai_rc4_20`, orig `sub_417C73`; 20-byte key via `i % 20`)

**Figure 10** – Results of the function encryption scheme analysis, together with the renamed functions from the analysis.

The assistant also implemented inline Python snippets and decrypted one of the encrypted functions, providing us with all the data and the keys, as well as the part of the decrypted code:

## 4) Putting it all together (worked example: `ai_dec_call_418DB3`)

- Wrapper keys
  - wrapper XOR: `0x4D21B545` (at `0x0042917C`)
  - seed source: `*(a2+20)`; in this call chain set to `0xBE61C366` → seed = `0xF3407623`
- Core keys
  - mix constant: `0x6CFC3E60` (at `0x0040471D`)
  - base key DWORDs (from `ai_encfunc_key_base`):  
`[28D252AC, ECD04511, AB5815D4, 9AB5C4CE, BA404E31]`
  - effective 20-byte key = base  $\oplus$  (seed  $\wedge$  `6CFC3E60`)  
→ (as 5 LE dwords): `ef1a6eb7 520d6c73 975de434 8d8c0905 7206fc25`
- Markers
  - head raw: `1e fc 5e 30 be 2c` → head enc (via A): `4d b6 6e 1f 71 7b`
  - tail raw: `36 0f d4 ed 82 bf` → tail enc (via A): `d1 21 28 62 35 e8`
- Locate + decrypt
  - found head in `/.text` at RVA `0x00018DBD`, tail at RVA `0x000192DE`
  - decrypted body: RVA `0x00018DC3`, 1307 bytes
  - first bytes of plaintext:  
`33 f6 68 b4 02 00 00 8d 85 08 fd ff ff 56 50 89 ...`
- Re-encrypt  
`ai_encfunc_reencrypt` applies Transform B to the same `[base,size]` and copies 6 bytes after the body to buffer #1 and before the body to buffer #2.

**Figure 11** – Report on the successful decryption of one of the functions (`0x00418DB3`).



Interestingly, in this case, the use of MCP wasn't even necessary, as the validity of the extracted keys can be easily verified by AI: if it's possible to locate the start and end markers of the code after decryption, it means the keys and the algorithm were recovered correctly. Additionally, we can see that the decrypted data doesn't appear to be random (it contains sequences like *00 00* and *ff ff*), which suggests the function was indeed decrypted correctly.

The AI performed very well in reimplementing the algorithms, including the modified RC4 with additional tweaks, as well as in locating the keys within the provided sample. It also successfully implemented functions for detecting 6-byte markers.

However, it was unable to fully implement a universal script capable of decrypting all functions without human assistance. The issue arose in locating all the XOR modifiers required to construct the 20-byte effective RC4 key.

The challenge lies in the fact that the effective RC4 key is derived by XOR-ing its 4-byte components with a 4-byte modifier, which is unique for each encrypted function and is calculated this way:

$$seed\_external \wedge seed\_internal \wedge 0x6CFC3E60$$

```
seed_external = *(_DWORD *) (a2 + 4);
memset(v5, 0, sizeof(v5));
tag_head[0] = 0x97F438C0;
tag_head[1] = 0x2B54;
tag_tail[0] = 0xEB4EC47B;
tag_tail[1] = 0x8BD1;
if (!ai_encfunc_begin(a1, seed_external ^ seed_internal ^ 0xD83708DC, (int)tag_head, (int)tag_tail, (int)v5))
 return 0;
v3 = ai_encfunc_416203(a1);
ai_encfunc_decrypt(tag_head, tag_tail, v5);

ai_encfunc_key_base(effective_key + 3);
effective_key[3] ^= seed ^ 0x6CFC3E60;
effective_key[4] ^= seed ^ 0x6CFC3E60;
effective_key[5] ^= seed ^ 0x6CFC3E60;
effective_key[6] ^= seed ^ 0x6CFC3E60;
effective_key[7] ^= seed ^ 0x6CFC3E60;
ai_xform_sub_rc4_sub(tag_head, 6, effective_key + 3);
ai_xform_sub_rc4_sub(tag_tail, 6, effective_key + 3);
```

**Figure 12** – Calculation of the 4-byte XOR modifier for the key required to decrypt the functions.

While *seed\_internal* is always located within the wrapper function near the markers, the assistant was unable to implement a universal method for finding *seed\_external* (see Figure 13 below), as it could be placed in various locations within the calling function and might be deliberately mixed with other constants.

#### Limitations (honest & explicit)

- If a wrapper forms the seed from **heap/global state** (e.g., `mov ecx, [ecx+20] ; xor ecx, imm32`) and does **not** stage that source value in a local stack slot near the call, its **numeric seed is not present in code**. With **no emulation** and **no external metadata**, a *pure* static pass cannot reconstruct that seed — the script will **skip** those sites (but still extract the raw 6B markers).

**Figure 13** – Report on the limitations encountered while creating the function decryption script.

We had to manually modify the script to ensure it could correctly locate all external seeds. Additionally, we modified the rules for locating the remaining constants to make the script truly robust and capable of working with other samples as well. Therefore, the AI significantly reduced the time required for analysis and script development, but at this stage, it could not fully replace a human.

It is clear that the creators of XLoader deliberately complicated the key construction process by scattering crucial constants across multiple functions, to the extent that even AI was unable to develop an algorithm to

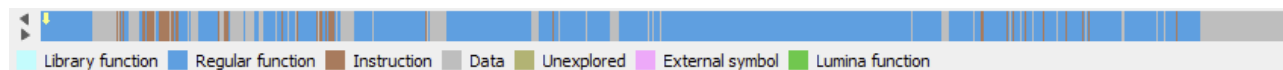
locate it. We are not disclosing how we derive the keys, so as not to give the XLoader developers any advantage.

Finally, after applying the script, we obtained 51 functions decrypted in the first pass. Many of the decrypted functions also contained similar calls to encrypted functions. Applying the script three times in a row, we got a total of 77 decrypted functions out of the 90 initially found.

```
.text:00416203 ai_encfunc_416203 proc near ;
.text:00416203
.text:00416203 var_364 = byte ptr -364h
.text:00416203 var_340 = byte ptr -340h
.text:00416203 var_138 = byte ptr -138h
.text:00416203 var_34 = byte ptr -34h
.text:00416203 var_24 = dword ptr -24h
.text:00416203 var_10 = byte ptr -10h
.text:00416203 arg_0 = dword ptr 8
.text:00416203 55 push ebp
.text:00416204 8B EC mov ebp, esp
.text:00416206 81 EC 64 03 00 00 sub esp, 364h
.text:0041620C 53 push ebx
.text:0041620D 56 push esi
.text:0041620E 90 nop
.text:0041620F 90 nop
.text:00416210 90 nop
.text:00416211 90 nop
.text:00416212 90 nop
.text:00416213 90 nop
.text:00416214 8B 75 08 mov esi, [ebp+arg_0]
.text:00416217 8D 45 CC lea eax, [ebp+var_34]
.text:0041621A 50 push eax
```

**Figure 14** – Decrypted function example with a patched 6-byte head marker (six NOP instructions).

After loading the resulting sample into IDA, we can see that a significant number of code blocks are still unrecognized:



**Figure 15** – Significant number of code blocks remain unrecognized by IDA.

During a quick review, we also identified several functions that still remain encrypted:

```
.text:00409F93 sub_409F93 proc near ; CODE XREF: .
.text:00409F93 55 push ebp
.text:00409F94 8B EC mov ebp, esp
.text:00409F96 81 EC 30 02 00 00 sub esp, 230h
.text:00409F9C 53 push ebx
.text:00409F9D 56 push esi
.text:00409FAE 57 push edi
.text:00409FAF 69 29 4F 1F 99 57 imul ebp, [ecx], 57991F4Fh
.text:00409FA5 F4 hlt
.text:00409FA5 sub_409F93 proc near ; CODE XREF: .
.text:00409FA5 ; -----
.text:00409FA6 28 D4 dw 00428h
.text:00409FAB 3D 55 24 F7 E3 FD 2E BA_ dd 0F724553Dh, 0BA2EFDE3h, 0003A0D7Dh,
.text:00409FB7 27 7A B1 49 E1 26 40 D6_ dd 49B17A27h, 0064026E1h, 00BEF15B4h,

.text:00411053 sub_411053 proc near ; CODE XREF:
.text:00411053 ; FUNCTION CHUNK AT .text:004110D5 SIZE 0000001F BYTE
.text:00411053 inc ebp
.text:00411054 C9 leave eax, dx
.text:00411055 ED in insb
.text:00411057 loc_411057: ; CODE XREF:
.text:00411057 A6 cmpsb stosd
.text:00411058 AB sbb [edx+edi*4+17D41E34h], b1
.text:00411059 18 9C BA 34 1E D4 17 pop ebx
.text:00411060 5B out dx, eax
.text:00411061 EF
```

**Figure 16** – Functions that remained encrypted after applying the decryption script.

It's also worth noting that in the scheme described above, encrypted functions are located using 6-byte sequences, which are replaced with six NOP instructions after decryption. This implies that the function must

have a valid, unencrypted prologue. At the same time, in Figure 16, on the right we can see an encrypted function that lacks a valid prologue. This likely indicates that a different decryption method was used.

## Function decryption scheme II

We recreated exporting the database and loaded it into ChatGPT. We initiated the analysis with the following prompt and uploaded the decryption log of the 77 functions:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

In the analyzed scheme, encrypted functions are located using 6-byte sequences, which are replaced with six NOP instructions after decryption.

This implies that the function must have a valid, unencrypted prologue. Some of the encrypted functions do not have a valid prologue (e.g., `sub\_407293`, `sub\_411053`, `sub\_415343`).

This likely indicates that a different method is used for it. Try to find it.

In the analyzed scheme, encrypted functions are located using 6-byte sequences, which are replaced with six NOP instructions after decryption. This implies that the function must have a valid, unencrypted prologue. Some of the encrypted functions do not have a valid prologue (e.g., `sub\_407293`, `sub\_411053`, `sub\_415343`). This likely indicates that a different method is used for it. Try to find it.

In the analyzed scheme, encrypted functions are located using 6-byte sequences, which are replaced with six NOP instructions after decryption.

This implies that the function must have a valid, unencrypted prologue. Some of the encrypted functions do not have a valid prologue (e.g., `sub\_407293`, `sub\_411053`, `sub\_415343`).

This likely indicates that a different method is used for it. Try to find it.

It's worth noting that we used a little trickery by pointing out the absence of a valid prologue in the prompt. Without this observation, the AI assistant was unable to identify the additional decryptors.

```
.text:00409F93 sub_409F93 proc near ; CODE XREF: .
.text:00409F93 55 push ebp
.text:00409F94 8B EC mov ebp, esp
.text:00409F96 81 EC 30 02 00 00 sub esp, 230h
.text:00409F9C push ebx
.text:00409F9D 56 push esi
.text:00409F9E 57 push edi
.text:00409F9F 69 29 4F 1F 99 57 imul ebp, [ecx], 57991F4Fh
.text:00409FA5 F4 hlt
.text:00409FA5 sub_409F93 endp
; -----
.text:00409FA6 2B D4 dw 0D428h
.text:00409FAB 3D 55 24 F7 E3 FD 2E BA_ dd 0F724553Dh, 0BA2EFD3h, 0003A0D7Dh,
.text:00409FBC 27 7A B1 49 E1 26 40 D6_ dd 49B17A27h, 0D64026E1h, 00BEF15B4h,

.text:00411053 sub_411053 proc near ; CODE XREF:
.text:00411053 ; FUNCTION CHUNK AT .text:00411005 SIZE 0000001F BYTE
.text:00411053 inc ebp
.text:00411054 C9 leave ebp
.text:00411054 C9 in eax, dx
.text:00411055 ED insb
.text:00411056 6C loc_411057: ; CODE XREF:
.text:00411057 A6 cmpsb
.text:00411058 AB stosd
.text:00411059 18 9C BA 34 1E D4 17 sbb [edx+edi*4+17D41E34h], bl
.text:00411059 58 pop ebx
.text:00411061 EF out dx, eax
```

**Figure 17** – Second decryption/patching scheme discovered.



Wrapper *0x00418dc3* decrypts function *0x0040d543* using:

- **Markers:** 9c da 5e e8, 0e e7 b2 36
- **Key-1:** 3f548513b8c7d376ec59d1a03e3313aaf6cd4262 (20B from sub\_404543, then XOR 0x36)
- **Key-2:** c2b2622cf0608327d4e542bc4ac3d2f709e092dc (calculated in the wrapper)

Additionally, a separate function may be used to handle the decryption of the second layer and the patching process.

```
unsigned int __cdecl ai_encfunc_decrypt_patch(int a1, int a2, int a3, unsigned int a4, int a5)
{
 unsigned int v5; // esi
 unsigned int v6; // eax
 _DWORD v8[2]; // [esp+8h] [ebp-8h] BYREF

 v5 = 0;
 v6 = 0;
 if (a4)
 {
 while (*(_DWORD *)(v6 + a1) != a3)
 {
 if (++v6 >= a4)
 goto LABEL_6;
 }
 v5 = v6;
 }
LABEL_6:
 ai_xform_sub_rc4_sub(a1, v5, a2);
 if (a5)
 {
 a4 = 0x90909090;
 v8[0] = 0xEC8B55;
 v8[1] = 0;
 ai_memcpy_bytes(a1 - 3, v8, 3);
 ai_memcpy_bytes(v5 + a1, &a4, 4);
 }
 return v5;
}
```

**Figure 21** – Function used to handle the decryption of the second layer and the patching process.

We identified three distinct function decryption schemes in XLoader:

- **Scheme 1** – Most common. In this scheme, encrypted functions already begin with a valid prologue. A global 20-byte base key is reused across all functions, but each function also has a unique 4-byte XOR modifier. The base key and modifier are combined to derive a per-function RC4 key. A modified RC4 routine is then used with this key to decrypt two 6-byte markers that define the boundaries of the encrypted block, and the same key is subsequently applied to the function body itself. Once decrypted, the two markers are overwritten with NOPs.
- **Scheme 2.** Uses 4-byte markers and two encryption layers. The first layer uses a 20-byte key produced by a dedicated key-builder function, then XOR-tweaked with a single byte. The second layer key is constructed from the 4-byte head marker concatenated with sixteen zero bytes. After decryption, the wrapper repairs the function by patching *55 8B EC* at the prologue and filling the tail with NOPs.

- **Scheme 3.** Similar to Scheme 2, but the second-layer key is not derived from the head marker. Instead, each wrapper embeds its own 20-byte constant (five DWORDs XORed with salt), which is used to decrypt the prefix up to a sentinel value before the same prologue/tail patching is applied.

It is worth noting that to implement universal static decryptors, we still had to break the task down into smaller steps ourselves: locating wrapper functions, extracting 20-byte keys, recovering 4-byte modifiers, and identifying and calculating marker positions. We combined them into a single decryptor only after confirming that each step worked reliably and produced the correct data for every function. At the same time, AI significantly reduced the time required to implement regular expressions (even though they had to be adjusted manually) as well as during the analysis and implementation of cryptographic algorithms.

With each decryption iteration we obtained a new batch of decrypted functions, some of which contained the keys required for decrypting additional functions. By applying all three decryptors sequentially over four iterations, we ultimately succeeded in decrypting 101 functions.

Unfortunately, it was not possible to accurately measure the time spent on this task, as it required a considerable amount of additional work and manual corrections. However, this stage turned out to be the most complex and time-consuming for both us and the LLM.

We also got *suggestions.json* with the suggested names for the analyzed functions. This is very useful, because we can keep this file for other analysis sessions and easily import it in the current IDB, or even in a new IDB (after decrypting the functions) without the need to diff it with the old database.

We now have a fully decrypted sample, which allows us to continue the analysis using the same methodology.

## API Calls deobfuscation

Now that we have a fully decrypted sample, we can apply the same approach to it. We first load *suggestions.json* and then perform an export. As stated earlier, even during the very first analysis of the sample (before we had obtained the decrypted functions) the assistant pointed out the presence of obfuscated API calls. The import table in this sample is empty and there are no plaintext strings that might contain library or function names.

As we created a new clean session by uploading the decrypted sample, we decided to test how reproducible the analysis results were.

Therefore, in the very first prompt, without providing any hints, we asked the assistant to identify the API call obfuscation mechanisms. As in the previous case, we specified that the analysis should begin at the OEP rather than at the *start* function, so that the AI assistant would not get bogged down analyzing the packer.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

The IAT is empty, no plaintext strings in the sample.

Determine how the sample resolves and invokes Windows APIs. Start the analysis from ``oep_start`` (`0x00430CB3``).

The IAT is empty, no plaintext strings in the sample. Determine how the sample resolves and invokes Windows APIs. Start the analysis from ``oep_start`` (`0x00430CB3``).

The IAT is empty, no plaintext strings in the sample.

Determine how the sample resolves and invokes Windows APIs. Start the analysis from ``oep_start`` (`0x00430CB3``).

Four minutes later, we received a description of the algorithm:

#### API resolution pipeline (hashed names + forwarders)

- **ID decoder:** `ai_apiid_decrypt` (`0x004045B3`, was `sub_4045B3`)  
Decrypts 32-bit obfuscated IDs (diff → RC4 → diff) into plaintext IDs used by the resolver. You'll see `push <obf>` → `call ai_apiid_decrypt` near each site before resolving.
- **Front-end resolver:** `ai_resolve_export` (`0x00425D43`, was `sub_425D43`)  
Validates PE (MZ/PE), then:
  - If a **name-hash** is supplied → calls `ai_find_export_by_hash` (`0x00426083`, was `sub_426083`).
  - Otherwise falls back to **string match** on export names.  
Handles **forwarders** transparently.  
**Proof:** hash equality helper `sub_425CC3` is hit at `0x00425E00`; calls into `ai_find_export_by_hash` at `0x00425E4E` / `0x00425F0A`; string compare at `0x00425E85`.
- **Hasher:** `ai_namehash_crc` (`0x00416EC3`, was `sub_416EC3`)  
CRC32-style, MSB-first over **lowercased** export names, finalized with `not eax`.  
**Proof:** table init via `sub_42F7D3` / `sub_416E83` at `0x00416EE4` / `0x00416EF0`; inner rolling update and final `not` at `0x00416F48`.  
(Behavior matches standard CRC32(MSB) with final NOT; lowercasing is required.)
- **Export table walk:** `ai_find_export_by_hash` (`0x00426083`, was `sub_426083`)  
Iterates Name → Ordinal → Address tables; for each name, computes/compares the hash and returns the target VA (forwarders re-resolved).  
**Proof:** name-index → ordinal → address at `0x004261DF-0x004261EB`; resolved pointer used with `call eax` at `0x00426240`.

**Figure 22** – Description of one of the API resolution algorithms.

Interestingly, during the first analysis (when part of the functions was still encrypted) and the second analysis, our AI assistant identified different functions responsible for API hash decryption. In the first case, it pointed to `sub_404603` (later renamed to `ai_apiid_decrypt_salt`), while in the second case it identified only `sub_4045B3` (`ai_apiid_decrypt`).

Next, we used the following simple prompt to generate a script for API call deobfuscation:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

Implement an IDAPython script for deobfuscating API calls. Annotate the resolver and every call site with Module!Function, original ID, and EA; IDA 9+. Log all deobfuscation attempts.

Implement an IDAPython script for deobfuscating API calls. Annotate the resolver and every call site with Module!Function, original ID, and EA; IDA 9+. Log all deobfuscation attempts.

Implement an IDAPython script for deobfuscating API calls. Annotate the resolver and every call site with Module!Function, original ID, and EA; IDA 9+. Log all deobfuscation attempts.

We got a script with the following functionality:

#### What it does (quick)

- **Decode IDs:** Finds `push imm32` → `call ai_apiid_decrypt (0x004045B3)` and decodes the 32-bit ID (diff → RC4-20 → diff).
- **Hash index:** Reads system exports and hashes function names with **CRC-32 MSB-first over lowercase** (stdcall `@N` stripped).
- **Map calls:** For each `ai_resolve_export (0x00425D43)` call, links to the **nearest preceding** decode and maps ID → `Module!Function`.
- **Annotate & log:**
  - Decoder & resolver sites get concise comments: `0x0BF -> 0xDEC (Name)`.
  - Console logs: `[DEC|RES] ea=... obf=... dec=... name=...`.

**Figure 23** – Description of the IDA Python script for API call deobfuscation.

As the assistant did not have access to IDA, we had to test the script manually. If there were errors, we sent the results back to the chat and asked for corrections. It took five iterations and about 20 additional minutes to obtain a fully working version.

Therefore, we also asked the assistant to analyze an alternative path:



Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

Analyze routine `sub\_404603` as an alternate API-hash decrypter. Recover its algorithm. Find call sites. Extend the IDAPython deobfuscator.

Analyze routine `sub\_404603` as an alternate API-hash decrypter. Recover its algorithm. Find call sites. Extend the IDAPython deobfuscator.

Analyze routine `sub\_404603` as an alternate API-hash decrypter. Recover its algorithm. Find call sites. Extend the IDAPython deobfuscator.

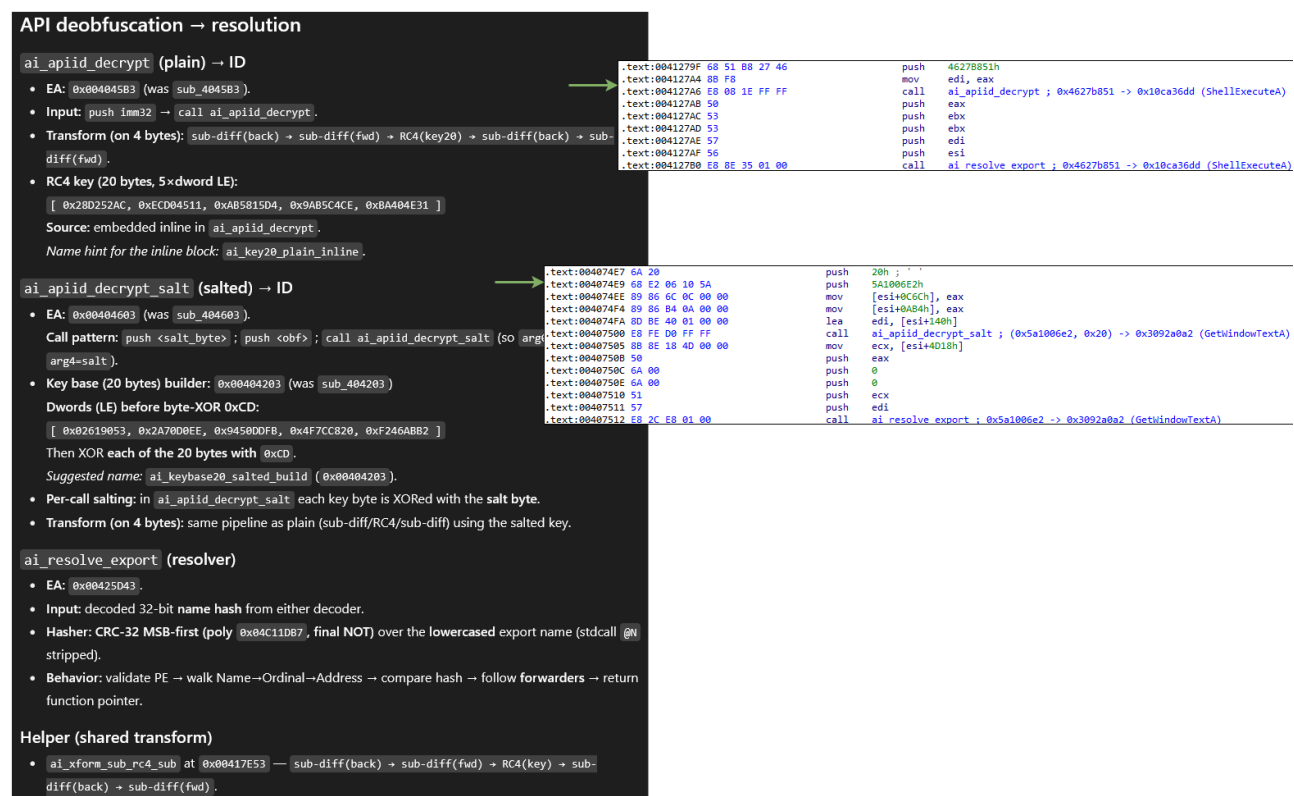
It took another five iterations (sending back errors and corrections) and 14 minutes before we obtained a fully functional script.

XLoader uses the same hash-decryption mechanism to look for sandbox artifacts, virtual machines, and processes typical of a researcher's environment. While fixing issues, we also added dictionary-based hash brute forcing (loading the wordlist from a separate file), which let us automatically annotate not only functions but also certain strings corresponding to specific evasion techniques:

```
ai_dec_func_45+124 call ai_apiid_decrypt_salt; (0x29dfa426, 0x91) -> 0xb6f4d5a8 (cwsx-)
ai_dec_func_45+148 call ai_apiid_decrypt_salt; (0x59987c31, 0x5) -> 0x67cea859 (wilbert-sc)
ai_dec_func_45+172 call ai_apiid_decrypt_salt; (0xf5ace332, 0x40) -> 0xc1626bff (xpamast-sc)
ai_dec_func_45+79 call ai_apiid_decrypt_salt; (0x0bd62d5d, 0x17) -> 0xed297eae (cuckoo)
ai_dec_func_45+A4 call ai_apiid_decrypt_salt; (0x9570af0e, 0x58) -> 0xa88492a6 (sandbox-)
ai_dec_func_45+CF call ai_apiid_decrypt_salt; (0x7840136e, 0x5b) -> 0xb21b057c (nmsandbox-)
ai_dec_func_45+FA call ai_apiid_decrypt_salt; (0xe5b09131, 0x6e) -> 0x70f35767 (xxxx-ox-)
ai_dec_func_46+126 call ai_apiid_decrypt_salt; (0xecc45a41, 0xe8) -> 0x5d4b4e66 (lavctestsuite\
ai_dec_func_46+36 call ai_apiid_decrypt_salt; (0xae6fb247, 0x5b) -> 0x6484bcb5 (cuckoo\
ai_dec_func_46+61 call ai_apiid_decrypt_salt; (0x719d9ebd, 0xff) -> 0x11fc2f72 (sandcastle\
ai_dec_func_46+8C call ai_apiid_decrypt_salt; (0xd86e258c, 0xf8) -> 0x2b44324f (laswsnx\
ai_dec_func_46+B4 call ai_apiid_decrypt_salt; (0xbd1c6722, 0xe) -> 0x9d70beea (sandbox\
ai_dec_func_46+DB call ai_apiid_decrypt_salt; (0x3eafee2a, 0xb8) -> 0x59adf952 (lsmmdir\
ai_dec_func_46+FF call ai_apiid_decrypt_salt; (0xd2514f0a, 0x2d) -> 0x172ac7b4 (samroot\
ai_dec_func_47+28 call ai_apiid_decrypt_salt; (0xa0c75f34, 0x39) -> 0xb4e1ae2 (ntdll.dll)
ai_dec_func_76+19D call ai_apiid_decrypt_salt; (0x41587c01, 0xd5) -> 0x8f5385d8 (CryptUnprotectData)
ai_dec_func_95+13A call ai_apiid_decrypt_salt; (0x81cfa733, 0x66) -> 0x42033885 (InternetOpenW)
ai_dec_func_95+155 call ai_apiid_decrypt_salt; (0x74bc21c3, 0x1) -> 0xc1368f28 (InternetConnectW)
ai_dec_func_95+170 call ai_apiid_decrypt_salt; (0x053cff70, 0x2d) -> 0xc0477b4a (HttpOpenRequestW)
ai_dec_func_95+191 call ai_apiid_decrypt_salt; (0x6818891d, 0xe4) -> 0xd3ec6064 (HttpSendRequestW)
ai_dec_func_95+1AC call ai_apiid_decrypt_salt; (0x926bd5e4, 0x7b) -> 0x233a695e (InternetReadFile)
ai_dec_func_95+1CA call ai_apiid_decrypt_salt; (0x2dfbb60b, 0x30) -> 0x550c512d (InternetQueryDataAvailable)
ai_dec_func_95+1E8 call ai_apiid_decrypt_salt; (0x4c50aeb7, 0x91) -> 0xd3afb325 (InternetCloseHandle)
ai_dec_func_98+138 call ai_apiid_decrypt_salt; (0x795e8895, 0xb6) -> 0x2902d074 (KiFastSystemCall)
ai_dec_func_98+B0 call ai_apiid_decrypt_salt; (0x795e8895, 0xb6) -> 0x2902d074 (KiFastSystemCall)
ai_dec_func_98+DB call ai_apiid_decrypt_salt; (0x795e8895, 0xb6) -> 0x2902d074 (KiFastSystemCall)
```

**Figure 24** – Deobfuscated API function and string identifiers.

As a bonus, we received a summary of how API resolution works, describing two different methods:



**Figure 25** – Description of the two methods for decrypting hashes and resolving API functions.

Overall, it took roughly one hour of the AI assistant's work to go from the first prompt to a fully functional API deobfuscation script. This figure does not include local testing or the time spent writing prompts. For this task, the human effort was minimal.

The table below summarizes the prompts we used and the time required for each step:

| Prompt                                                                                                                                                                                  | Time consumed | Notes                                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|--------------------------------------------------------------------------------------------------------------------------------|
| Initial prompt (instructions)                                                                                                                                                           | 1m 26s        |                                                                                                                                |
| The IAT is empty, with no plaintext strings in the sample.                                                                                                                              |               |                                                                                                                                |
| Determine how the sample resolves and invokes Windows APIs. Start the analysis from oep_start (0x00430CB3).                                                                             | 4m 28s        | Found only one path (ai_apiid_decrypt).                                                                                        |
| Implement an IDAPython script for deobfuscating API calls. Annotate the resolver and every call site with Module!Function, original ID, and EA; IDA 9+. Log all deobfuscation attempts. | 9m 44s        | The script did not work.                                                                                                       |
| Multiple                                                                                                                                                                                | 20m           | Local testing of the script and sending error reports, 5 iterations.                                                           |
| Analyze routine sub_404603 as an alternate API-hash decrypter. Recover its algorithm. Find call sites. Extend the IDAPython deobfuscator.                                               | 4m 49s        | Described the second API resolution path through ai_apiid_decrypt_salt (formerly sub_404603). The updated script did not work. |

| Prompt                                                                                                  | Time consumed | Notes                                                                                             |
|---------------------------------------------------------------------------------------------------------|---------------|---------------------------------------------------------------------------------------------------|
| <i>Multiple</i>                                                                                         | 14m 35s       | Local testing of the script, sending error reports and notes on incorrect behavior, 5 iterations. |
| Please provide a summary on two algorithms ai_apiid_decrypt_salt, ai_apiid_decrypt -> ai_resolve_export | 17s           |                                                                                                   |
| <i>Total time</i>                                                                                       | 55m 2s        |                                                                                                   |

## Additional protection of critical API calls

While reviewing the API call deobfuscation results, we noticed that some functions are invoked through an interesting wrapper which was originally hidden among the encrypted functions at address 0x0040AC93 (ai\_dec\_func\_16).

This function acts as a *secure-call trampoline*: it temporarily encrypts nearly the entire image before invoking a function pointer and then decrypts those same regions once the call returns. Only a tiny “island” (the space between the call-site’s return address and a marker) remains unencrypted by the per-call XOR so that execution can proceed.

Memory layout at call time (HEAD & TAIL encrypted, island plaintext)
Copy code

```

vbnet

[base] [API call] [end = base+img]
|==== HEAD (*key[i%20]) =====|--- ISLAND (plain) ---|==== TAIL (*key[i%20]) =====|
 ^retaddr ^marker = 90 90 B0 A4 90

• HEAD = [base .. retaddr]; TAIL = [marker .. base+image_size]; ISLAND = (retaddr .. marker).
• During call: HEAD/TAIL are XOR'd with key[i%20]; ISLAND stays plain.
• Entry/exit: global seed-XOR flips on enter, flips back on return.

```

One-liner purpose

A secure-call trampoline that encrypts everything before the stub's return address and everything after a runtime marker to image end, calls the target function, then restores both regions and flips a global seed XOR at entry/exit — minimizing plaintext exposure during sensitive API invocations.

```

enc_seed = ctx->enc_seed;
size_280544 = ret_0x447e0(); // 280544 bytes
code_region_ptr = ctx->code_region_ptr;
memset(key, 0, 21);
ai_key20_from_seed(key, enc_seed);
if (!code_region_ptr || !argv || !argc)
 return 0xC0000002;
ai_xor_region_with_seed_key(ctx, enc_seed);
retaddr = ai_get_retaddr(); // ai_get_retaddr() -> 0x0040AD39
patch_marker = ai_find_patch_marker(retaddr, 0xA4u); // 90 90 B0 A4 90 -> 0x0040AEBA
if (!patch_marker)
 return 0xC0000002;
size = ai_get_retaddr() - ctx->code_region_ptr; // ai_get_retaddr() -> 0x0040AD58
for (i = 0; i < size; ++i)
 code_region_ptr[i] ^= key[i % 20];
if (patch_marker)
{
 size2 = size_280544 - (patch_marker - ctx->code_region_ptr);
 for (j = 0; j < size2; ++j)
 patch_marker[j] ^= key[j % 20];
}
n = argc;
do
{
 --n;
 while (n);
 res = fn(*argv);
 for (k = 0; k < size; ++k)
 code_region_ptr[k] ^= key[k % 20];
 if (patch_marker)
 {
 for (m = 0; m < size2; ++m)
 patch_marker[m] ^= key[m % 20];
 }
 ai_xor_region_with_seed_key(ctx, enc_seed);
 return res;
} while (1);
// .text:0040AEBA 90 nop
// .text:0040AEBB 90 nop
// .text:0040AEB0 B0 A4 mov al, 0A4h
// .text:0040AEBE 90 nop

```

**Figure 26** – “Secure-call trampoline” decompiled function explained by LLM.

What makes this mechanism notable? Because the function stays encrypted nearly the entire time, it is difficult to even detect its existence without static decryption. In memory dumps, it appears only in encrypted form.

At the same time, if some security software or a sandbox hooks API calls protected by this wrapper and tries to analyze or dump the process memory at the time of the call, the mechanism effectively shields the malicious code.

In total, 20 functions are protected this way, including NTAPI routines related to processes, threads, memory, and file operations, as well as several WinSock functions. The full list is shown in the image below:

| Address                                   | Text                                                     |
|-------------------------------------------|----------------------------------------------------------|
| ai_secure_call_socket+6D                  | call ai_secure_call_dec_func_16; socket                  |
| ai_secure_call_getaddrinfo+83             | call ai_secure_call_dec_func_16; getaddrinfo             |
| ai_secure_call_connect+6D                 | call ai_secure_call_dec_func_16; connect                 |
| ai_secure_call_send+73                    | call ai_secure_call_dec_func_16; send                    |
| ai_secure_call_recv+83                    | call ai_secure_call_dec_func_16; recv                    |
| ai_secure_call_NtOpenProcess+78           | call ai_secure_call_dec_func_16; NtOpenProcess           |
| ai_secure_call_NtOpenThread+78            | call ai_secure_call_dec_func_16; NtOpenThread            |
| ai_secure_call_NtGetContextThread+6C      | call ai_secure_call_dec_func_16; NtGetContextThread      |
| ai_secure_call_NtSetContextThread+6C      | call ai_secure_call_dec_func_16; NtSetContextThread      |
| ai_secure_call_NtSuspendThread+6C         | call ai_secure_call_dec_func_16; NtSuspendThread         |
| ai_secure_call_NtResumeThread+6F          | call ai_secure_call_dec_func_16; NtResumeThread          |
| ai_secure_call_NtQueueApcThread+7E        | call ai_secure_call_dec_func_16; NtQueueApcThread        |
| ai_secure_call_NtDelayExecution+6D        | call ai_secure_call_dec_func_16; NtDelayExecution        |
| ai_secure_call_NtCreateSection+8D         | call ai_secure_call_dec_func_16; NtCreateSection         |
| ai_secure_call_NtMapViewOfSection+A3      | call ai_secure_call_dec_func_16; NtMapViewOfSection      |
| ai_secure_call_NtCreateFile+A9            | call ai_secure_call_dec_func_16; NtCreateFile            |
| ai_secure_call_NtReadFile+99              | call ai_secure_call_dec_func_16; NtReadFile              |
| ai_secure_call_NtDeleteFile+66            | call ai_secure_call_dec_func_16; NtDeleteFile            |
| ai_secure_call_NtReadVirtualMemory+7E     | call ai_secure_call_dec_func_16; NtReadVirtualMemory     |
| ai_secure_call_NtAllocateVirtualMemory+84 | call ai_secure_call_dec_func_16; NtAllocateVirtualMemory |

**Figure 27** – The list of API calls protected by “secure-call trampoline.”

## String decryption

The sample contains no readable strings. At the same time, the code features a series of short routines that share the same skeleton: the prologue allocates a small stack frame and initializes a local array (its size varies by function). This is followed by a recurring trio of calls with the same order and identical argument signatures:

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> .text:00405773 55 .text:00405774 8B EC .text:00405776 83 EC 1C .text:00405779 53 .text:0040577A 56 .text:0040577B 8B 75 08 .text:0040577E 6A 19 .text:00405780 56 .text:00405781 C7 45 E4 C4 E6 91 FA .text:00405788 C7 45 E8 D4 B3 54 D8 .text:0040578F C7 45 EC AC 7F 1F BA .text:00405796 C7 45 F0 A0 F8 52 B3 .text:0040579D C7 45 F4 AD BA 9A B8 .text:004057A4 C7 45 F8 76 F1 B9 4D .text:004057AB 66 C7 45 FC 00 00 .text:004057B1 E8 CD 9F 02 00 .text:004057B6 6A 18 .text:004057B8 8D 45 E4 .text:004057BB 50 .text:004057BC 56 .text:004057BD E8 91 9F 02 00 .text:004057C2 6A 18 .text:004057C4 56 .text:004057C5 B3 DE .text:004057C7 E8 27 F9 FF FF .text:004057CC 83 C4 1C .text:004057CF 5E .text:004057D0 5B .text:004057D1 8B E5 .text:004057D3 5D .text:004057D4 C3 .text:004057D4 </pre> | <pre> push    ebp mov     ebp, esp sub     esp, 1Ch push    ebx push    esi mov     esi, [ebp+arg_0] push    19h push    esi mov     [ebp+var_1C], 0FA91E6C4h mov     [ebp+var_18], 0D85483D4h mov     [ebp+var_14], 0BA1F7FAC4h mov     [ebp+var_10], 0B352F8A0h mov     [ebp+var_C], 0B89ABAADh mov     [ebp+var_8], 4DB9F176h mov     [ebp+var_4], 0 call    sub_42F783 push    18h lea     eax, [ebp+var_1C] push    eax push    esi call    ai_memcpy_bytes push    18h push    esi mov     bl, 0DEh call    sub_4050F3 add     esp, 1Ch pop     esi pop     ebx mov     esp, ebp pop     ebp retn endp </pre> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Figure 28** – Example of an encrypted string in XLoader code.

Let's analyze one of these functions. Because we started a fresh session for this task, we instructed the AI assistant to trust the existing comments that briefly describe each function's behavior, so it doesn't reanalyze routines that were already covered.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

Analyze the functionality starting from `sub\_405773`. Recurse into its callees. Trust the comments in the disassembly.

Analyze the functionality starting from `sub\_405773`. Recurse into its callees. Trust the comments in the disassembly.

Analyze the functionality starting from `sub\_405773`. Recurse into its callees. Trust the comments in the disassembly.

As a result, we determined that this function decrypts a string using the algorithm implemented in one of the previously decrypted routines (ai\_dec\_func14). Note how the AI built the call graph and described each routine's behavior from a single short prompt, relying solely on the data in the previously prepared archive:

### What `sub_405773` does (string decrypt stub)

**Purpose (per code + prior notes):** build a 24-byte ciphertext on the stack, zero-terminate a 25-byte destination, copy the 24 bytes to `dst`, then decrypt in place using a 20-byte key derived from constants and salt 0xDE, via the dynamically-decrypted engine `ai_dec_func_14`.

#### Call graph (one level down)

nginx

Copy code

```
sub_405773 (dst)
├─ sub_42F783 ; zero out dst, len = 25
├─ ai_memcpy_bytes ; copy 24B ciphertext → dst
├─ sub_4050F3 (BL=0xDE) ; build key20 (^0x33 then ^BL), then
│ ├─ sub_404283 ; write base key bytes (20B), XOR each with 0x33
│ └─ sub_416F53 ; decrypt via engine loader:
│ └─ sub_42C343
│ ├─ ai_encfunc_locate_decrypt_wrapper(...)
│ └─ ai_dec_func_14(dst,len,key20) ; engine body (RC4 + byte-diff)
└─ ai_encfunc_recrypt(...)
```

**Figure 29** – Call graph and description of the string decryptor stub.

With a simple prompt, we readily obtain the encryption keys and the decrypted string.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

Decrypt the string from ``sub_405773``.

Decrypt the string from ``sub_405773``.

Decrypt the string from ``sub_405773``.

Decrypted string (from `sub_405773`):

`Q97vXR4iHn+C7yQMHTBY5Jk=`

- It's 24 bytes (NULL-terminated at len 25).
- Decoding Base64 → 17 bytes: `43 DE EF 5D 1E 22 1E 7F 82 EF 24 0C 1D 30 58 E4 99`.

### Provenance (short)

- **Ciphertext (24B)** built in `sub_405773` (`0x00405773`):  
dwords → `C4E691FA D4B354D8 AC7F1F8A A0F852B3 ADBA9AB8 76F1B94D` (LE), then copied to `dst` and zero-filled to 25. (See `decomp/sub_405773_00405773.c`.)
- **Key (20B)** from `sub_404283` then salted in `sub_4050F3` (`BL=0xDE`):  
base dwords `6F5F3E9A 63346B63 148BB354 8CE61520 04271C3E` → XOR `0x33` → XOR `0xDE`  
= `77 D3 B2 82 8E 86 D9 8E B9 5E 56 F9 CD F8 0B 61 D3 F1 CA E9`. (See `decomp/sub_404283_00404283.c` and `decomp/sub_4050F3_004050F3.c`.)
- **Transform** (`ai_dec_func_14`, `0x0040A7C3`):  
pre-RC4: backward sub-diff, then forward sub-diff → RC4 (20-byte key, KSA/PRGA) → post-RC4:  
backward sub-diff, then forward sub-diff. (See `decomp/ai_dec_func_14_0040A7C3.c` and insns at `0x0040A7E5..0x0040A800`, `0x0040A939..0x0040A94E`.)

Confidence: high.

**Figure 30** – String decryption result with derived key and ciphertext.

Now that we are confident the AI assistant understands how the encrypted strings are stored, knows how the key is derived, and already analyzed and correctly reimplemented the decryption algorithm (verifying it on real data), we can move on to implementing a script to decrypt the remaining strings.

This time we used a slightly more detailed prompt because we wanted specific information to appear both in the comments and in the console output:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

Implement an IDAPython script (IDA 9+) that decrypts strings. Requirements:

- Find and annotate every call to the decrypter function ``sub_4050F3`` with the decrypted string.
- For each string, output debug info: encrypted buffer, XOR tweak byte (``BL``), length, and decrypted bytes.
- Print all binary data in hex.

Implement an IDAPython script (IDA 9+) that decrypts strings. Requirements: - Find and annotate every call to the decrypter function `sub\_4050F3` with the decrypted string. - For each string, output debug info: encrypted buffer, XOR tweak byte (`BL`), length, and decrypted bytes. - Print all binary data in hex.

Implement an IDAPython script (IDA 9+) that decrypts strings. Requirements:

- Find and annotate every call to the decrypter function `sub\_4050F3` with the decrypted string.
- For each string, output debug info: encrypted buffer, XOR tweak byte (`BL`), length, and decrypted bytes.
- Print all binary data in hex.

This time we were lucky and immediately got a working script which we used to decrypt 175 strings:

```
[dec] Using decrypt function(s) at: 0x4050F3
[dec] Found 175 call site(s).
[dec] 0x4051a0 len=0x14 bl=0x0E7 enc=7509088B28A8A83A76F2AE83C1F55FF7CDC3992 dec=767A646D4662687737794F767864466B436D6F3D ascii="vzdmFbhw7yOvxdFkCmo="
[dec] 0x405207 len=0x18 bl=0x0C enc=878CF0EB38481B48B3E1B0C25B79F92D88D0C6530625428EB dec=31516F3162472F7862704932674759386C437A574277303D ascii="1Qo1bG/xbpI2gY8lCzhBw=="
[dec] 0x405270 len=0x14 bl=0x7F enc=A82A703D28CFD522F588FE3F527A693D08ACF74 dec=706C3467A53302F7141536872396C7058686C50 ascii="p13Fz50/qAShr9lpPk1P"
[dec] 0x4052d7 len=0x18 bl=0x40 enc=A0773A8F578580634EE2A3A579CE889F47D40758A43A71EE dec=486A4672F6754644734776F7457554D6332466461514034 ascii="HjGg/gTdg4wotWUMc2FdaQW4"
[dec] 0x405347 len=0x18 bl=0x70 enc=54E690CE0E08F60A22B69ADD8BF8324611F1078C254FB4 dec=44664961367034534D7075546C39707A383875707843733D ascii="DfIa6p45Mpu19pz88upxCs="
[dec] 0x4053b7 len=0x18 bl=0x38 enc=F06B4352AA314F813026E84666D7CAEC5C9A9AE786A4CAD dec=6A3570796351305A3677674A4E453562527367485A67303D ascii="j5pycQ8Z6wgJNE5bRgHZg=="
[dec] 0x40542e len=0x1c bl=0xC5 enc=DECC0A87EC3265729A875FFA56DC5982C1DEACDAF0A188C9C7C3E7AD dec=6B5641302F795728585271394F4E4A3244484F4B6348576D59433542 ascii="kvA0/yu+XRq9ONJ2DHOKchmMYCSB"
[dec] 0x40549e len=0x1c bl=0x9D enc=58727A858881153A12332E9AD2082A079BA18E2A806A17A4628879F dec=677149586C615076686C66716A2F724334357148436976757967303D ascii="gqIXIaPvhIfaj/rC45qHC1vuyg=="
[dec] 0x405500 len=0x14 bl=0x0A enc=33383F5B2F20748754E44AD0F08C0718101F138 dec=7A61782B48A57177743671394857697639416F3D ascii="zax+ZgwT6g9Khlv9Aco"
[dec] 0x405560 len=0x14 bl=0x96 enc=FD5024905F64A7163771F084D55227FAC16FA60 dec=566E727867A4A66757A5531414C3759527372686E ascii="Vnrxg3FuzU1AL7YRsrhn"

[dec] 0x40d054 len=0x4 bl=0x0F enc=478B10E1 dec=47455420 ascii="GET "
[dec] 0x40d093 len=0x3 bl=0x56 enc=7F91FC dec=505554 ascii="PUT"
[dec] 0x40d0d4 len=0x4 bl=0x8C enc=C19DC143 dec=504F5354 ascii="POST"
[dec] 0x40d11c len=0x7 bl=0x88 enc=E83AF2E81A6AC9 dec=4F5054494F4E53 ascii="OPTIONS"
[dec] 0x40d172 len=0xc bl=0xFC enc=E8F4C408B26107EC038D58FA dec=557365722D4167656E743A20 ascii="User-Agent: "
[dec] 0x40d1b4 len=0x4 bl=0x45 enc=59AF25CC dec=4158492D ascii="API-"
[dec] 0x40d1f4 len=0x4 bl=0x0F enc=98C42F1A dec=40532D57 ascii="MS-W"
[dec] 0x40d23c len=0xa bl=0x9E enc=08B705E4153E38111968 dec=5F333031204D6F766564 ascii="_301 Moved"
[dec] 0x40d28c len=0xa bl=0xEB enc=C0F3B9ED9A9615E8530A dec=5F33303220466F756E64 ascii="_302 Found"
[dec] 0x40d2e9 len=0xd bl=0x06 enc=EC7CEAB7D08526CBFB703C615 dec=496E7465726E65744F70656E41 ascii="InternetOpenA"
[dec] 0x40d349 len=0x10 bl=0x51 enc=C41FCE4C44C0243833012D913C72ACC dec=496E7465726E6574436F6E6E65637441 ascii="InternetConnectA"
[dec] 0x40d3a9 len=0x10 bl=0x18 enc=843A054568EF7CB681D56A802BD23852 dec=487474704F70656E5265717565737441 ascii="HttpOpenRequestA"
[dec] 0x40d409 len=0x10 bl=0x48 enc=0888512294B35E33C8D0478ED1A132CC dec=4874747053656E6565717565737441 ascii="HttpSendRequestA"
[dec] 0x40d469 len=0x10 bl=0x0C enc=B080E68A80E746F2E8919F2C2209205 dec=496E7465726E65745265615446696C65 ascii="InternetReadFile"
[dec] 0x40d4d1 len=0x13 bl=0xE9 enc=ASA7200B7F9895F8A87786294F0F25948BCA4 dec=496E7465726E6574436C6F736548616E646C65 ascii="InternetCloseHandle"
```

**Figure 31** – Decrypted strings in IDA.

The total time required for the analysis was about 20 minutes:

| Prompt                                                                                                                                                                                                                                                                                                         | Time consumed |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|
| Initial prompt (instructions)                                                                                                                                                                                                                                                                                  | 1m 26s        |
| Analyze the functionality starting from sub_405773. Recurse into its callees. Trust the comments in the disassembly.                                                                                                                                                                                           | 7m 45s        |
| Decrypt the string from sub_405773                                                                                                                                                                                                                                                                             | 5m 10s        |
| Implement an IDAPython script (IDA 9+) that decrypts strings. Requirements: Find and annotate every call to the decrypter function sub_4050F3 with the decrypted string. For each string, output debug info: encrypted buffer, XOR tweak byte (BL), length, and decrypted bytes. Print all binary data in hex. | 5m 30s        |
| Total time                                                                                                                                                                                                                                                                                                     | 19m 51s       |

We now have a decrypted sample with deobfuscated strings and API calls that we can analyze like a regular binary. However, XLoader is not that simple: some data remains encrypted even at this stage.

## Decrypting domain names



Extracting lists of Indicators of Compromise (IoCs) is always a critical task in malware analysis. Network indicators, such as domain names and URLs, are especially important because they help detect and classify malware through traffic analysis. That is why extracting domains is essential — even though some may be decoys or bait, or currently inactive but intended for later use.

Among the recovered strings, we see 64 Base64-encoded entries. Looking at the version history of XLoader, we find that starting from version 2.8 it began storing encrypted domain names in Base64 form. Without a doubt, these 64 Base64 strings represent domain names that we must decrypt. As early as version 4, XLoader added two additional layers of modified RC4 encryption with different keys, making the decryption more complicated. In later versions, this process became even more complex. In total, to reach the decrypted domain names we need to peel off at least five layers after first identifying where and how the keys are initialized: decrypt the functions that initialize encrypted strings, decrypt the strings themselves (which we already did earlier), base64-decode the results, and apply two more layers of decryption.

At the same time, obtaining the keys for each layer is the most difficult part, as the different pieces of data needed to generate them are scattered across multiple functions, making them hard to locate.

Before moving on, we updated our string deobfuscation script so that it also renames the functions responsible for retrieving decrypted strings and assigns a prefix *ai\_dec\_domain\_{NN}* to all functions handling Base64-encoded strings. We then exported the database to prepare for domain decryption.

All calls to the *ai\_dec\_domain\_{NN}* functions occur inside a single function, *ai\_dec\_func\_0(0x00404913)*:

```

v13[58] = 0x7A1BDE24;
v13[59] = 0x1819FF1D;
v13[60] = 0x51DB0D24;
v13[61] = 0x3B3CEA70;
v13[62] = 0x1D6DFF60;
v13[63] = 0xAAD2737F;
v13[64] = 0xF7C77C95;
memset(&v13[65], 0, 12);
sub_42F783((int)a3, 6);
*a3 = '/';
a3[5] = '/';
ai_memcpy_bytes((int)(a3 + 1), &v13[domain_index], 4);
return ai_dec_dispatch(a3 + 1, 4u, (int)v26);
}
else
{
 result = a1 - 1;
 if ((unsigned __int8)(domain_index - 1) <= 63u)
 {
 switch (domain_index)
 {
 case 1u:
 ai_dec_domain_00((int)a3);
 break;
 case 2u:
 ai_dec_domain_01((int)a3);
 break;
 case 3u:
 ai_dec_domain_02((int)a3);
 break;
 case 4u:
 ai_dec_domain_03((int)a3);
 break;
 case 5u:
 ai_dec_domain_04((int)a3);
 break;
 }
 }
}

```

**Figure 32** – Domain generation function.

We start the analysis of this function using the simplest prompt:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

Analyze `ai\_dec\_func\_0` (0x00404913)

Analyze `ai\_dec\_func\_0` (0x00404913)

Analyze `ai\_dec\_func\_0` (0x00404913)

As a result of our analysis, we obtained a detailed description showing that `ai_dec_func_0` (later renamed to `ai_dec_func_0_domain_tag_generate`) is XLoader's stage-1 domain builder. For a given domain

index (1..64), it pulls the matching seed string from *ai\_dec\_domain\_NN*, base64-decodes it, then runs a keyed RC4-with-diff transform whose key is a 20-byte secret stored at *ctx+0x23D0* (where *ctx* is a global structure that stores keys, function addresses, and other data), and byte-XORed with the domain index. The result is re-encoded to base64 and written to the output buffer. This is an intermediate artifact, not the final ASCII domain.

### What it does (in plain steps)

- Inputs:
  - a2**: context base; uses a 20-byte secret at offset +0x23D0 (9168).
  - a3**: output buffer (mutable).
  - a4** (U8): domain index.
  - a5**: mode selector.
- Fast-path / probe mode ( **a5 == 222** ):
  - Build a 20-byte key **K** from 5 dwords ( **v26[0..4]** ) then XOR each byte with **a4**.
  - Copy 4 bytes from a big dword table **v13** at index **a4** into **a3+1**, set **a3[0] = '/'**, **a3[5] = '/'**.
  - Run **sub\_416F53(a3+1, 4, K)** (a keyed transform/dispatch) and return.
- Normal mode ( **a5 != 222** and **1 ≤ a4 ≤ 0x40** ):
  - Call **ai\_dec\_domain\_{a4-1}(a3)** to materialize a per-index seed string (~20 bytes) into **a3**.
  - Decode that string into binary with **sub\_42C473(&tmp, a3, v12)**, which delegates to **ai\_dec\_func\_49** (custom base64-like decode).
  - Load secret **\*S = (a2+0x23D0)** and XOR every byte of **S** with **a4** → key **K**.
  - Keyed transform: **sub\_416F53(&tmp, len, K)** (wrapper that reaches **ai\_dec\_func\_14**).
  - Emit base64 of **tmp** back into **a3** via **sub\_42C3D3(a3, &tmp, len, v14)**, which calls **ai\_dec\_func\_48** (standard base64 encode).
  - Minor house-keeping: **sub\_42F953** (strlen), **sub\_42F783** (zero/resize helper).
- Else (index out of range): returns **a1-1** (the **ebx** param minus one).

**Figure 33** – Domain generation function analysis report.

A special “token” branch is enabled when *a5==222* is used for generating a path string for a URL. It emits a short pattern */<4-byte token>/* from a static table and applies the same keyed transform.

Next, we asked the assistant to reproduce the transformations implemented in *ai\_dec\_func\_0\_domain\_tag\_generate*. Because our script already retrieved the decrypted strings and added them as comments in the previous step, we requested that these strings also be used alongside the repeated string decryption:

Comments in the disassembly of *ai\_dec\_domain\_NN* contain base64-encoded encrypted domain names.

Plain text

Copy to clipboard

Open code in new window

Take the one from `ai\_dec\_domain\_01`, then transform it with `ai\_dec\_func\_0` and return the final string.

Take the one from `ai\_dec\_domain\_01`, then transform it with `ai\_dec\_func\_0` and return the final string.

Take the one from `ai\_dec\_domain\_01`, then transform it with `ai\_dec\_func\_0` and return the final string.

As a result, our assistant was unable to locate the key at `ctx+0x23D0` on its own:

If you can provide the 20-byte blob at `(ctx + 0x23D0)` (any format is fine — hex or raw), I'll finish the decryption and return the domain string.

**Figure 34** – AI was unable to locate the key.

That was not surprising, as none of the exported data contained references to the structure's fields and can only be located by the offset in the structure, or captured in the debugger.

Here's the result for `ai_dec_domain_01` → `ai_dec_func_0` (normal mode, index = `a4 = 2`):

- Seed from `ai_dec_domain_01` (24-byte blob) → RC4-decrypt (BL=220) → `1Qo1bG/xbpI2gGY81CzWBw==`
- Context key at `ctx + 0x23D0`: built from `ai_key20_seed_load`'s 20-byte seed, XOR `0x8F`, then XOR `a4 (2)` → key bytes (hex):  
`b3 59 d0 b8 05 47 84 80 46 08 04 d8 f6 17 de b6 8c 08 e8 ff`
- Decrypt that base64 with `ai_dec_func_0` (its `ai_dec_dispatch` / `ai_dec_func_14` does sub-diff(back) → sub-diff(fwd) → RC4(20-byte key) → sub-diff(back) → sub-diff(fwd), then base64-encodes):

**Final string (what `ai_dec_func_0` writes):**

```
ini
Qvm75Acm5NpYTbnYXdcvBw==
```

[Copy code](#)

**Figure 35** – Initialization of the RC4 key for decrypting the first layer.

We therefore had to manually locate where this key was initialized and provide that information to the AI assistant.

Plain text

Copy to clipboard

Open code in new window

You can find `ctx+0x23D0` initialization in `ai_dec_func_4` (`0x00407293`), also check `sub_404453`.

After the keys were extracted, we tried again:Now you have the required key.

Take the sting from `ai\_dec\_domain\_01`, then transform it with `ai\_dec\_func\_0` and return the final string.

You can find `ctx+0x23D0` initialization in `ai\_dec\_func\_4` (`0x00407293`), also check `sub\_404453`. After the keys were extracted, we tried again:Now you have the required key. Take the sting from `ai\_dec\_domain\_01`, then transform it with `ai\_dec\_func\_0` and return the final string.

You can find `ctx+0x23D0` initialization in `ai\_dec\_func\_4` (`0x00407293`), also check `sub\_404453`.

After the keys were extracted, we tried again:Now you have the required key. Take the sting from `ai\_dec\_domain\_01`, then transform it with `ai\_dec\_func\_0` and return the final string.

As a result, we obtained an intermediate value for the string returned by `ai_dec_domain_01` (`1Qo1bG/xbpI2gGY8lCzWBw==`) after the first decryption layer, which was re-encoded in Base64 as (`Qvm75Acm5NpYTbnYXdcvBw==`):

Here's the result for `ai_dec_domain_01` → `ai_dec_func_0` (normal mode, index = `a4 = 2`):

- Seed from `ai_dec_domain_01` (24-byte blob) → RC4-decrypt (BL=220) → `1Qo1bG/xbpI2gGY8lCzWBw==`
- Context key at `ctx + 0x23D0`: built from `ai_key20_seed_load`'s 20-byte seed, XOR `0x8F`, then XOR `a4` (2) → key bytes (hex):  
`b3 59 d0 b8 05 47 84 80 46 08 04 d8 f6 17 de b6 8c 08 e8 ff`
- Decrypt that base64 with `ai_dec_func_0` (its `ai_dec_dispatch` / `ai_dec_func_14` does sub-diff(back) → sub-diff(fwd) → RC4(20-byte key) → sub-diff(back) → sub-diff(fwd), then base64-encodes):

**Final string (what `ai_dec_func_0` writes):**

```
ini
```

`Qvm75Acm5NpYTbnYXdcvBw==`

[Copy code](#)

**Figure 36** – Result of reproducing the domain generation function's behavior for index 2.

Because the string is still encrypted, and we do not yet know what happens to it next, we need to investigate further:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

The string you returned is still encrypted. Trace the complete transformation chain from ``ai_dec_func_0`` (case 2, string ``Qvm75Acm5NpYTbnYXdcbW==``) to the final ASCII domain.

Discover any remaining layers, locate and derive all required keys/parameters from the context/initializers, and cite function names with EAs used.

Output the final domain, and a concise step-by-step pipeline, print all keys/IVs as hex. If any value is missing, state exactly what it is and where to read it.

The string you returned is still encrypted. Trace the complete transformation chain from ``ai_dec_func_0`` (case 2, string ``Qvm75Acm5NpYTbnYXdcbW==``) to the final ASCII domain. Discover any remaining layers, locate and derive all required keys/parameters from the context/initializers, and cite function names with EAs used. Output the final domain, and a concise step-by-step pipeline, print all keys/IVs as hex. If any value is missing, state exactly what it is and where to read it.

The string you returned is still encrypted. Trace the complete transformation chain from ``ai_dec_func_0`` (case 2, string ``Qvm75Acm5NpYTbnYXdcbW==``) to the final ASCII domain.

Discover any remaining layers, locate and derive all required keys/parameters from the context/initializers, and cite function names with EAs used.

Output the final domain, and a concise step-by-step pipeline, print all keys/IVs as hex. If any value is missing, state exactly what it is and where to read it.

As a result, we discovered that the obtained Base64 string is decoded again and then decrypted by a second layer using a 20-byte key generated inside the function `ai_dec_func_11` (`0x004095F3`). This key is additionally XORed with `SALT_DWORD`. However, the initialization of `SALT_DWORD` is missing from `ai_dec_func_11`, and the assistant was therefore unable to retrieve it on its own.

**Stage-2 key (this is the one that produces the final ASCII domain)**

Initialized in `ai_dec_func_11` (0x004095F3). Five dwords are prepared and each is XOR'ed with a 32-bit salt read from a per-run context:

- Base dwords (before XOR with salt), as bytes (LE):

```
bash
0d 09 3b 60 | fc a7 27 17 | 91 3d c7 a0 | 12 66 e8 75 | be 67 db 99
```

- Then each 4-byte chunk is XOR'ed with `SALT_DWORD = *(_DWORD *) (ctx + 2808) + 9672`.

After this XOR, the 20 bytes are copied to `(ctx + 9352)` and used by `ai_dec_dispatch` to produce the final domain string.

There is no IV; the transform is the same keyed stream/byte-diff sequence used everywhere in this sample (`ai_dec_func_14`).

**What's still missing (exactly)**

To finish and output the final domain, I need **one thing**:

- `SALT_DWORD` (4 bytes, little-endian) from

```
SCSS
*(_DWORD *) (*(_DWORD *) (ctx + 2808) + 9672)
```

This is read at the top of `ai_dec_func_11` (0x004095F3) and XOR'ed into the 5 base dwords to form the 20-byte stage-2 key stored at `(ctx + 9352)`.

**If you prefer an equivalent input**

Instead of the salt, you can provide the 20 bytes currently at `(ctx + 9352)` after `ai_dec_func_11` runs. I can use that directly as the stage-2 key.

```
v2 = *(_DWORD *) (ctx + 2808);
key[0] = 0x603B090D;
key[1] = 0x1727A7FC;
key[2] = 0xA0C73D91;
key[3] = 0x75E86612;
key[4] = 0x99DB67BE;
v59 = 0;
v60 = 0;
v61 = 0;
*(_DWORD *) (ctx + 19708) = 7;
salt_dword = *(_DWORD *) (v2 + 0x25C8);
key[0] ^= salt_dword;
key[1] ^= salt_dword;
key[2] ^= salt_dword;
key[3] ^= salt_dword;
key[4] ^= salt_dword;
```

**Figure 37** – AI was unable to locate the Stage-2 key.

As in the previous case, we manually recovered the missing value. The complication was that instead of offset `0x25C8`, the base `0x2000` was used, with `0x5C8` added later, which made the search a bit more difficult.

We provided not only the address of the function where the initialization of `SALT_DWORD` was assumed to occur, but also the relevant code snippets:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

You can find `SALT_DWORD` in `'ai_dec_func_20'` (`'0x00411053'`). Please verify and continue:

```
.text:004111B3 81 C7 00 20 00 00 add edi, 2000h
```

...

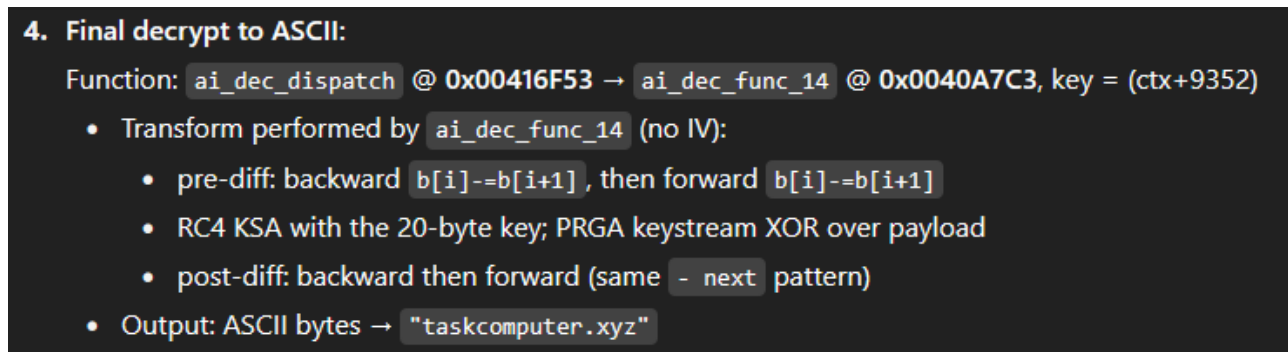
```
.text:0041182B C7 87 C8 05 00 00 00 A6 mov dword ptr [edi+5C8h], 0C6EA600h
```

You can find SALT\_DWORD in `ai\_dec\_func\_20` (`0x00411053`). Please verify and continue:  
.text:004111B3 81 C7 00 20 00 00 00 add edi, 2000h ... .text:0041182B C7 87 C8 05 00 00 00 A6 mov dword ptr [edi+5C8h], 0C6EA600h

You can find SALT\_DWORD in `ai\_dec\_func\_20` (`0x00411053`). Please verify and continue:

```
.text:004111B3 81 C7 00 20 00 00 00 add edi, 2000h
...
.text:0041182B C7 87 C8 05 00 00 00 A6 mov dword ptr [edi+5C8h], 0C6EA600h
```

During the analysis, our AI assistant confirmed that the key was correct and successfully decrypted the domain corresponding to *ai\_dec\_domain\_01*:



**Figure 38** – Result of successful domain decryption.

Now the AI assistant can automatically decrypt all the domains. Let's verify this by asking it to decrypt the first 16 domains:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

Decrypt domains from `ai\_dec\_domain\_00` to `ai\_dec\_domain\_15`. Output: table (index, src\_base64, final\_domain). Take per-function base64 from `ai\_dec\_domain\_NN` disassembly comments.

Decrypt domains from `ai\_dec\_domain\_00` to `ai\_dec\_domain\_15`. Output: table (index, src\_base64, final\_domain). Take per-function base64 from `ai\_dec\_domain\_NN` disassembly comments.

Decrypt domains from `ai\_dec\_domain\_00` to `ai\_dec\_domain\_15`. Output: table (index, src\_base64, final\_domain). Take per-function base64 from `ai\_dec\_domain\_NN` disassembly comments.



In the end, we obtained a table with the fully decrypted domain names:

| Index | Base64 string                    | Domain                   |
|-------|----------------------------------|--------------------------|
| 00    | vzdmFbhw7yOvxdFkCmo=             | taxi-in[.]online         |
| 01    | 1Qo1bG/xbpl2gGY8lCzWBw==         | taskcomputer[.]xyz       |
| 02    | pl3FzS0/qAShr9lpPkIP             | synergycdrop[.]xyz       |
| 03    | HjGg/gTdG4wotWUMc2FdaQM4         | royal-bet-king[.]xyz     |
| 04    | Dfla6p4SMpuTI9pz88upxCs=         | goldenspoon[.]click      |
| 05    | j5pycQ0Z6wgJNE5bRsgHZg==         | spark-stack[.]shop       |
| 06    | kVA0/yW+XRq9ONJ2DHOKcHWmYC5B     | lecerisierenfleur[.]net  |
| 07    | gqIXlaPvhlfqj/rC45qHCivuyg==     | shhiajtdaz9bhau[.]top    |
| 08    | zax+KZqwt6q9KWiv9Ao=             | allslotvip[.]vip         |
| 09    | VnrxcJfuzU1AL7YRsrhn             | debatevxtlm[.]sbs        |
| 10    | qjZURKL4uqdfIR6t/6ZRdBY/s9E/qw== | theexcelconundrum[.]info |
| 11    | LiJnz4B0FuWt3hyZKCg=             | runsociety[.]org         |
| 12    | L/EwBoA7HTg=                     | avvo[.]com               |
| 13    | zV9rEdBGpqvEo3v3cAnEkw==         | botbuilders[.]team       |
| 14    | vk923CIZnNuJkgaN4qXyZA==         | hawkingonsol[.]xyz       |
| 15    | Q97vXR4iHn+C7yQMHTBY5Jk=         | streamingsite[.]xyz      |

Let's recall that at the very beginning of the analysis we discovered a separate branch of `ai_dec_func_0` that activates when the last value equals 222. In this case, a 4-character tag is generated which later becomes part of the URL. Previously, this tag was the same for all domains and unique to a malware campaign. Now, however, each domain has its own tag.

We tried to decrypt the tags for the first 16 domains using the following prompt:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

Reproduce the output of ``ai_dec_func_0`` for `a5=222` and domain index 0..15.

Reproduce the output of ``ai_dec_func_0`` for `a5=222` and domain index 0..15.

Reproduce the output of ``ai_dec_func_0`` for `a5=222` and domain index 0..15.

In response to the prompt, the assistant provided a description of how the malware generates those 4-character tags, and the table containing the first 16 tags.

Here's the exact output `ai_dec_func_0_domain_tag_generate (0x00404913)` produces in **token mode** (`a5==222`). It writes `'/' + transform(4-byte token) + '/'`, where the 4 bytes come from its internal table and are transformed by `ai_dec_dispatch` → `ai_dec_func_14 (0x00416F53 → 0x0040A7C3)` with the per-call 20-byte key (`v26[] XOR domain_index`).

**Figure 39** – Description of the domain generation function in “token” mode.

| Index | Source bytes (hex, 4B LE) | Output bytes (hex, 6B) | Output string |
|-------|---------------------------|------------------------|---------------|
| 0     | 00 f6 76 09               | 2f 7a 62 75 75 2f      | /zbuu/        |
| 1     | 62 64 68 53               | 2f 6e 74 64 35 2f      | /ntd5/        |
| 2     | 2a bd 03 94               | 2f 78 65 72 73 2f      | /xers/        |
| 3     | 82 7e 4e 51               | 2f 6c 30 6d 6c 2f      | /l0ml/        |
| 4     | 0a 7c 87 3c               | 2f 38 67 6e 35 2f      | /8gn5/        |
| 5     | e1 5c f0 cb               | 2f 65 32 63 71 2f      | /e2cq/        |
| 6     | 27 21 aa e4               | 2f 33 67 6c 64 2f      | /3gld/        |
| 7     | ca ca b2 c7               | 2f 6b 6d 39 34 2f      | /km94/        |
| 8     | 51 7c 8a 6b               | 2f 7a 6b 33 32 2f      | /zk32/        |
| 9     | 30 d3 9f 6a               | 2f 7a 74 6b 72 2f      | /ztkr/        |
| 10    | c8 9a be 30               | 2f 35 67 6d 69 2f      | /5gmi/        |
| 11    | 2a 30 1c e3               | 2f 69 76 37 6e 2f      | /iv7n/        |
| 12    | 51 7b 47 41               | 2f 37 74 6e 61 2f      | /7tna/        |
| 13    | d1 d5 0d d2               | 2f 6f 67 61 67 2f      | /ogag/        |
| 14    | 8d 34 e0 6f               | 2f 71 30 73 66 2f      | /q0sf/        |
| 15    | df 4a 1e b6               | 2f 65 63 69 73 2f      | /ecis/        |

The table below summarizes the prompts we used, and the time required for each step:

| Prompt                                                                                                                                                                                                                                       | Time consumed | Notes                                |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|--------------------------------------|
| Initial prompt (instructions)                                                                                                                                                                                                                | 1m 31s        |                                      |
| Analyze <code>ai_dec_func_0 (0x00404913)</code>                                                                                                                                                                                              | 5m 22s        |                                      |
| Comments in the disassembly of <code>ai_dec_domain_NN</code> contain base64-encoded encrypted domain names. Take the one from <code>ai_dec_domain_01</code> , then transform it with <code>ai_dec_func_0</code> and return the final string. | 11m 33s       | Failed to obtain the layer 1 RC4 key |
| You can find <code>ctx+0x23D0</code> initialization in <code>ai_dec_func_4 (0x00407293)</code> , also check <code>sub_404453</code> .                                                                                                        | 1m 34s        |                                      |
| Now you have the required key. Take the sting from <code>ai_dec_domain_01</code> , then transform it with <code>ai_dec_func_0</code> and return the final string.                                                                            | 5m 14s        |                                      |
| The string you returned is still encrypted. Trace the complete transformation chain from <code>ai_dec_func_0</code> (case 2, string <code>Qvm75Acm5NpYTbnYXdcvBw==</code> ) to the final ASCII domain.                                       | 10m 2s        | Failed to obtain SALT_DOWRD          |
| Discover any remaining layers, locate and derive all required keys/parameters from the context/initializers,                                                                                                                                 |               |                                      |

and cite function names with EAs used.  
Output the final domain, and a concise step-by-step pipeline, print all keys/IVs as hex. If any value is missing, state exactly what it is and where to read it.  
You can find SALT\_DWORD in ai\_dec\_func\_20 (0x00411053). Please verify and continue:  
.text:004111B3 81 C7 00 20 00 00 add edi, 2000h 3m 20s  
...  
.text:0041182B C7 87 C8 05 00 00 00 A6 mov dword ptr [edi+5C8h], 0C6EA600h  
Decrypt domains from ai\_dec\_domain\_00 to ai\_dec\_domain\_15. Output: table (index, src\_base64, final\_domain). Take per-function base64 from ai\_dec\_domain\_NN disassembly comments. 5m 36s  
Reproduce the output of ai\_dec\_func\_0 for a5=222 and domain index 0..15. 3m 42s  
*Total time* 47m 54s

## Conclusion

From its initial appearance, XLoader has always been a moving target, with each new version raising the bar for security analysts and defenders. XLoader began as a two-layer puzzle but evolved into a maze of nested decryptors, scattered key material, and runtime-only code. For years, this meant that by the time researchers fully unraveled a sample, attackers were already one step ahead with the next version.

Generative AI changes this balance. Combining cloud-based analysis and occasional MCP-assisted runtime checks, we delegated a large part of the mechanical reverse engineering to LLM. Instead of spending hours rewriting decryption routines by hand, we asked our AI model to do it and received working prototypes in minutes.

The use of AI doesn't eliminate the need for human expertise. XLoader's most sophisticated protections, such as scattered key derivation logic and multi-layer function encryption, still require manual analysis and targeted adjustments. But the heavy lifting of triage, deobfuscation, and scripting can now be accelerated dramatically. What once took days can now be compressed into hours.

For defenders, this is more than a productivity boost. Faster turnaround means fresher IoCs, quicker detection updates, and a shorter window of opportunity for attackers. For researchers, it lowers the entry barrier to analyzing some of the most complex malware families in the wild.

Our research shows that with the right workflows, generative models can already serve as a force multiplier — helping security defenders keep pace with threats that were once considered prohibitively time-consuming to analyze.

However, it's too soon to declare victory, as we expect malware authors to adapt their techniques in response to AI-assisted analysis. And in turn, we'll need to come up with the next game-changer.

## Protection

Check Point [Threat Emulation](#) and [Harmony Endpoint](#) provide comprehensive coverage of attack tactics, file types, and operating systems and protect against the attacks and threats described in this report.

[GO UP](#)

[BACK TO ALL POSTS](#)