Reverse Engineering a Malicious Minecraft Forge Mod: An Analysis of Java-Based Info Stealer

Jack Vega

November 2025

Abstract

This paper presents a reverse-engineering analysis of a malicious Java application posed as a Minecraft Forge mod. Static analysis shows that the package functions exclusively as a credential-harvesting and exfiltration utility. The investigation highlights how malicious actors exploit familiar distribution formats to disguise standalone infostealers.

1 Introduction

Community modification frameworks such as Minecraft Forge extend the creative reach of players and developers alike. Unfortunately, the same ecosystem that fosters open collaboration also enables the distribution of malicious software through deceptive mod packages. Over the past several years, a recurring threat has emerged: infostealers presented as simple "utility" or "enhancement" mods.

This paper documents a defensive analysis of one representative example, intended to demonstrate research methodology and to promote awareness. The sample examined here is not an authentic gameplay extension but a credential-harvesting Java application disguised as a Forge mod. All analysis was conducted with ethical intent, and was disclosed through the proper reporting channels.

2 Safety Controls

Prior to analysis, precautionary measures were taken to eliminate any risk of accidental execution. The sample was renamed with the `.malware` extension and handled exclusively in a controlled, non-executable environment. No behavioral testing was performed; instead, the

investigation relied entirely on static inspection of the decompiled bytecode and associated metadata.

3 Methodology

The investigation followed a structured analytical workflow modeled on standard malware research practice:

1. **Identification** - Initial discovery of the sample on public repositories such as GitHub and collection of general metadata, including file type, language (Java), and claimed functionality.
2. **Decompilation and Static Analysis** - Use of safe tooling to read compiled Java bytecode into a human-legible form for annotation and tracing.
3. **Architectural Mapping** - Identify relationships among functions, network routines, and local file interactions.

Dynamic analysis and sandbox execution were intentionally excluded to avoid engaging the malware as static analysis is sufficient.

4 Analytical Tools

Java Decompiler (JD-GUI).[1] A lightweight graphical decompiler capable of displaying class hierarchies and readable method approximations from compiled bytecode. This tool enabled direct examination of the program's logic and identification of suspicious function calls without requiring code execution.

5 Identification

The sample was discovered on a public GitHub repository claiming to distribute a utility mod for the Minecraft server *Hypixel SkyBlock*. Several indicators visible on the repository page itself suggested suspicious intent before any binary inspection was performed:

- **Implausible or unethical functionality.** The repository description advertised the mod as a "duping" tool for in-game advantage, implying capabilities that violate

[1] `https://java-decompiler.github.io/`

server rules and legitimate gameplay mechanics. Such claims are strong indicators of potentially malicious or deceptive software as this area of modding is unregulated and can be full of bad actors.

- **Lack of available source code.** The repository contained only a compiled release (a single JAR file) with no corresponding source files or build instructions. Legitimate Forge mods commonly distribute open or inspectable source for transparency or community contribution.
- **Unusual distribution format.** Although not present here, similar suspicious repositories often provide archives (`.zip`) instead of standard mod packages (`.jar`), or mix multiple file types inconsistent with typical Forge workflows.
- **Language or framework inconsistency.** In some cases, the advertised mod may be for a game that is modded in language A, but the repo uses language B.
- **New or low-activity repository.** The project had minimal development history, few or no commits, and recent creation timing, characteristics consistent with short-lived malware staging accounts.

These preliminary warnings justified further offline analysis of the distributed binary to determine whether its function aligned with its stated purpose.

Upon download, the sample was immediately identified as a Java archive (.jar) package. This observation justified the use of JD-GUI for reverse engineering. When opened in JD-GUI, the file displayed no obfuscation, allowing full review of the class names and logic flow. With the code legible, the analysis could proceed directly to inspection of its behavior and structure.

6 Analysis

Static examination revealed that the program contained a single class performing all operations associated with initialization and exfiltration. It implemented no Minecraft gameplay logic, only routines for data capture and network transmission. Comments within the code and function naming conventions further confirmed the program's purpose as a credential-stealing agent rather than a legitimate mod.

7 Anatomy of the Malicious Workflow

Analysis of the decompiled source revealed three discrete phases of operation. Figure 1 illustrates the conceptual data flow.

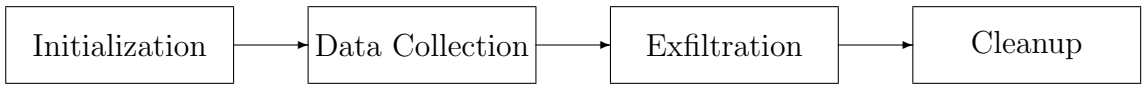

Figure 1: Simplified operational flow of the analyzed sample.

Phase 1: Initialization. Upon loading, the program retrieved configuration text from a remote paste-hosting service. The file included a line containing a webhook URL, which was parsed and stored in memory for future data transmission.

Phase 2: Data Collection. Methods within the main class enumerated directories associated with the Minecraft launcher and auxiliary clients, reading files that resembled account or session data. Each discovery was serialized into a temporary plaintext file within the folder `user-data/`.

Phase 3: Exfiltration and Cleanup. A routine then constructed a multipart HTTP POST request containing all gathered artifacts along with system information such as username, operating system, and IP address. Following successful transmission, the staged files were immediately deleted to minimize evidence on disk.

8 Behavioral Signatures and Heuristics

Even without dynamic execution, several persistent indicators emerged during static analysis that facilitate automated detection:

- Use of `HttpURLConnection` within the Forge mod initializer.
- Creation—and subsequent deletion—of temporary files in a self-made directory (e.g., `user-data/`).
- References to online text-hosting services used for remote configuration.

- Inconsistent or deceptive metadata, such as non-descriptive or misleading mod identifiers.
- Embedded JSON payloads and tags like `@everyone`, signaling potential Discord webhook abuse.

Together, these traits form a recognizable behavioral signature for detecting malicious Forge-style JARs.

9 Discussion: Why This Approach Persists

Producing a credential stealer disguised as a Forge mod requires minimal technical skill yet achieves significant reach. Because Forge automatically executes classes annotated with `@Mod`, a malicious mod inherits a built-in execution path without reliance on external loaders or privilege escalation. The community's open trust model and the prevalence of mod-sharing platforms make this an ideal vector for social engineering.

Threat actors exploit:

1. User assumptions that all minecraft mods are safe.
2. The association of open-source hosting with safety.

These factors collectively sustain the success of such deceptive packages within the Minecraft ecosystem.

10 Conclusion

This defensive analysis of a malicious Java application posing as a Minecraft Forge mod revealed a straightforward yet effective information-stealing design. The sample contained no authentic mod features, functioning solely as a data harvester with remote exfiltration capability. Recognizing these deceptive structures aids researchers and developers in identifying and neutralizing similar threats before deployment.

Ultimately, the paper illustrates that even minimal Java programs can violate user trust when distributed under familiar community formats. Awareness, source verification, and improved code-review enforcement remain key to reducing exposure in open-mod ecosystems.