Malware Analysis: A Case Study

Nandinii Yeleswarapu

Taking Apart Malware: What Reverse Engineering Taught Me About Writing Better Code

This project came out of my graduate Software Reverse Engineering course, a rare class where curiosity was encouraged more than memorization. We were given access to real-world malware samples and asked to dissect their behavior safely. My chosen sample was from the **Win32.LuckyCat** family, a strain known for its persistence tactics and command-and-control (C&C) disguises.

Setup

All analysis was done inside an isolated Windows 7 virtual machine to keep the work safe and reproducible. I used Ghidra for static disassembly and function tracing, Wireshark to capture and inspect network traffic, RegShot and Process Monitor to observe filesystem and registry changes, and a VirtualBox sandbox for controlled execution. The goal was never to "break" the sample but to map how it worked: how it persisted, how it talked to remote servers, and what tricks it used to avoid inspection.

Observed behavior and how I identified it

While analyzing the sample, I documented every behavior that stood out, ere's a summary of what I found and how each behavior was uncovered.

Why It Stuck With Me

Even though this wasn't a software engineering project, it changed how I think about code. Reverse engineering forces you to read other people's code, obfuscated, defensive, and often malicious, and still make sense of it. Security may seem far from building distributed backends, but the principles overlap: resilience, predictability, and understanding how code behaves under pressure. LuckyCat just happened to be the teacher that made those lessons stick.

Behavior	**Technique / Classification**	**How I found it**
C&C traffic disguised as POP3 / DNS	Network obfuscation, protocol mimicry	Packet captures with Wireshark; filtered by destination ports and payload patterns; correlation with timestamps from dynamic runs.
Registry keys for persistence	Persistence via autorun registry changes	Registry diffs with RegShot and live monitoring with Process Monitor to locate created keys and values.
Anti-debug checks and debugger detection	Anti-analysis / anti-debugging	Static analysis in Ghidra to find API calls (IsDebuggerPresent, CheckRemoteDebuggerPresent) and control-flow checks; confirmed during dynamic runs that code paths changed when a debugger was present.
File and process manipulation	Local footprint and lateral actions	Process Monitor traces showed file writes and spawned processes; cross-checked file hashes and timestamps after execution.
Command-and-control protocols	Encrypted or encoded payload exchanges	Combined Ghidra string analysis (to locate protocol markers) with Wireshark captures to reconstruct message formats and identify beacon intervals.

Table 1: Key behaviors observed in the Win32.LuckyCat sample, their classification, and how they were identified.