Rustyjackd Raspberry Pi Diagnostic Test Blueprint

A single-script (.sh) test plan for security, correctness, and reliability
(Project: watchdog / Rustyjack)

Generated: 2026-01-10 (Europe/Dublin)

Executive summary

This document is a blueprint for a comprehensive on-device diagnostic suite for **rustyjackd** on a Raspberry Pi. The goal is to produce a single **bash** script that can be run after your install script finishes, and that will: (1) validate systemd hardening and OS-level containment, (2) exercise the daemon's Unix-domain-socket (UDS) RPC protocol, (3) verify authorization boundaries, (4) stress reliability/timeout behavior, and (5) emit a machine-readable summary plus human-friendly logs describing *what failed and why*.

Important safety principle: by default, the diagnostic suite should run in **SAFE mode** (read-only and non-disruptive). Potentially disruptive tests (WiFi connect/disconnect, hotspot start/stop, reboot/shutdown, mounting devices, reverse shells, MITM, DNS spoofing, etc.) should be gated behind an explicit **--dangerous** flag and ideally a second "I know what I'm doing" prompt.

System under test: daemon shape and RPC protocol

On the Pi, **rustyjackd** is a systemd service that listens on a UDS (default `/run/rustyjack/rustyjackd.sock`) and implements a framed JSON protocol:

```
Frame format (rustyjack-ipc/src/wire.rs):
- 4-byte big-endian length prefix (u32)
- followed by UTF-8 JSON payload of that length

Handshake (rustyjack-daemon/src/server.rs):
1) Client sends ClientHello JSON frame:
   { "protocol_version": 1, "client_name": "...", "client_version": "...", "supports": [] }
2) Daemon replies with HelloAck JSON frame:
   { "protocol_version": 1, "daemon_version": "...", "features": [...], "max_frame": 1048576,
     "authz": { "uid": ..., "gid": ..., "role": "operator|admin|read_only" } }

Requests:
- Client sends RequestEnvelope:
  { "v": 1, "request_id": <u64>, "endpoint": "<snake_case>", "body": { "type": "<CamelCase>", "data":
- Daemon replies ResponseEnvelope:
  { "v": 1, "request_id": <u64>, "body": { "type": "Ok|Err|Event", "data": ... } }

Protocol-robustness protections in server.rs:
- handshake timeout: 2s
- read/write timeouts: configurable via env (defaults 5s)
- MAX_PROTOCOL_VIOLATIONS = 3 (then connection is dropped)
```

Authorization is derived from Linux peer credentials (SO_PEERCRED) and group membership. Linux documents that SO_PEERCRED returns the peer credentials that were in effect at connect time (including PID/UID/GID).

Your diagnostics should therefore test both (a) filesystem permissions on the socket path and (b) daemon-side tier checks, because a weakness in either can yield privilege escalation.

Where the interesting logic lives (files & functions)

These are the hotspots your test script is implicitly validating (and the places you'll look first when tests fail):

```
Core protocol & authorization
- rustyjack-daemon/src/server.rs
  - handle_connection(): handshake, frame parsing, protocol violation tracking
  - read_frame()/write_frame(): 4-byte length-prefixed framing
  - auth checks: required_tier(), tier_allows(), required_tier_for_jobkind()

- rustyjack-daemon/src/auth.rs
  - peer_credentials(): SO_PEERCRED extraction
  - authorization_for_peer(): group-based auth (reads /proc/<pid>/status and /etc/group)
  - required_tier(): endpoint -> tier matrix

API surface (request/response types)
- rustyjack-ipc/src/types.rs
  - Endpoint enum (snake_case serialized)
  - RequestBody enum (adjacently tagged: {type, data})
  - ResponseBody / ResponseOk / DaemonEvent

Business logic dispatch
- rustyjack-daemon/src/dispatch.rs
  - handle_request(): match on RequestBody and call core ops / state ops
  - CoreDispatch (legacy): CommandDispatch (parses rustyjack_core::Commands)

Runtime configuration
- rustyjack-daemon/src/config.rs
  - socket path, max_frame, timeouts, group names, root path
- rustyjackd.service (systemd unit)
  - sandboxing/hardening directives, capabilities, runtime dirs
```

Designing the single-script diagnostic harness

The recommended implementation is a single **bash** entrypoint that generates a run directory, then executes suites. For UDS RPC it is simplest to embed a small helper in **python3** (ships on most Raspberry Pi images) to handle framing, JSON encoding/decoding, and structured results. This keeps the harness a single file while still being readable.

```
Proposed output layout (created per run):
  /var/tmp/rustyjackd-diag/<timestamp>/
    diag.log                # human log (bash)
    summary.json            # machine summary (json)
    sysinfo.txt             # uname, os-release, cpuinfo, memory, df
    systemd/
      unit.txt              # systemctl cat/show
      security.txt          # systemd-analyze security
      journal.txt           # journalctl -u rustyjackd --since <start>
    rpc/
      handshake.json
      requests/
        0001_health.json
        ...
      responses/
        0001_health.json
        ...
```

Prerequisites and tool dependencies

Minimum: bash, coreutils, python3, systemctl, journalctl, ss/ip. Optional but helpful: jq, socat, timeout, lsof, iptables/nft, iw.

The harness should self-check dependencies at startup and downgrade gracefully (e.g., if jq is missing, keep raw JSON).

RPC helper: how the .sh script should call daemon APIs

```
In bash, avoid trying to craft big-endian length prefixes by hand.
Instead, embed python:

rj_rpc <body_type> <json_data_or_null> [--socket PATH] [--timeout-ms N]

Python steps:
1) connect Unix socket
2) send ClientHello frame
3) read HelloAck frame (capture authz.role and features)
4) send RequestEnvelope with:
   - v: 1
   - request_id: incrementing u64
   - endpoint: snake_case(body_type) (must match daemon’s endpoint_for_body check)
   - body: { "type": body_type, "data": data }
5) read ResponseEnvelope, print JSON to stdout, exit non-zero if body.type == "Err"

Your bash harness stores request/response JSON in rpc/requests and rpc/responses.
```

Endpoint authorization matrix

Derived from `rustyjack-daemon/src/auth.rs::required_tier`. Use this to assert allow/deny behavior when connecting as different users.

Endpoint (snake_case)	RequestBody.type	Required tier
health	Health	ReadOnly
version	Version	ReadOnly
status	Status	ReadOnly
job_start	JobStart	Operator
job_status	JobStatus	Operator
job_cancel	JobCancel	Operator
core_dispatch	CoreDispatch	Operator
status_command	StatusCommand	ReadOnly
wifi_command	WifiCommand	Operator
ethernet_command	EthernetCommand	Operator
loot_command	LootCommand	Operator
notify_command	NotifyCommand	Operator
system_command	SystemCommand	Operator
hardware_command	HardwareCommand	Operator
dns_spoof_command	DnsSpoofCommand	Operator
mitm_command	MitmCommand	Operator
reverse_command	ReverseCommand	Operator
hotspot_command	HotspotCommand	Operator
scan_command	ScanCommand	Operator
bridge_command	BridgeCommand	Operator
process_command	ProcessCommand	Operator
system_status_get	SystemStatusGet	ReadOnly
disk_usage_get	DiskUsageGet	ReadOnly
system_reboot	SystemReboot	Admin
system_shutdown	SystemShutdown	Admin
system_sync	SystemSync	Admin
hostname_randomize_now	HostnameRandomizeNow	Admin
block_devices_list	BlockDevicesList	ReadOnly
system_logs_get	SystemLogsGet	Operator
active_interface_get	ActiveInterfaceGet	ReadOnly
active_interface_clear	ActiveInterfaceClear	Operator
interface_status_get	InterfaceStatusGet	ReadOnly

Endpoint (snake_case)	RequestBody.type	Required tier
wifi_capabilities_get	WifiCapabilitiesGet	ReadOnly
hotspot_warnings_get	HotspotWarningsGet	Operator
hotspot_diagnostics_get	HotspotDiagnosticsGet	Operator
hotspot_clients_list	HotspotClientsList	Operator
gpio_diagnostics_get	GpioDiagnosticsGet	Operator
wifi_interfaces_list	WifiInterfacesList	ReadOnly
wifi_disconnect	WifiDisconnect	Operator
wifi_scan_start	WifiScanStart	Operator
wifi_connect_start	WifiConnectStart	Operator
hotspot_start	HotspotStart	Operator
hotspot_stop	HotspotStop	Operator
portal_start	PortalStart	Operator
portal_stop	PortalStop	Operator
portal_status	PortalStatus	ReadOnly
mount_list	MountList	ReadOnly
mount_start	MountStart	Operator
unmount_start	UnmountStart	Operator
set_active_interface	SetActiveInterface	Operator
hotplug_notify	HotplugNotify	Operator
log_tail_get	LogTailGet	Operator
logging_config_get	LoggingConfigGet	ReadOnly
logging_config_set	LoggingConfigSet	Admin

Test suites

Each suite is designed to be runnable independently, but the “single script” will run them in order. For each test, capture: start/end time, command, stdout/stderr, return code, and a short failure diagnosis.

Suite A: Installation and service sanity

Purpose: Confirm the daemon binary, systemd unit, runtime directories, and socket are present and consistent with configuration.

Steps:

```
1) systemctl is-enabled rustyjackd
2) systemctl is-active rustyjackd
3) systemctl show rustyjackd -p MainPID,ExecStart,User,Group,Environment
4) systemctl cat rustyjackd
5) ls -ld /run/rustyjack /var/lib/rustyjack
6) stat /run/rustyjack/rustyjackd.sock (type, mode, owner, group)
7) journalctl -u rustyjackd -b --no-pager | tail -n 200
```

Expected pass criteria:

```
- Service is active; MainPID is non-zero
- Socket exists and is a Unix socket
- RuntimeDirectory permissions are 0770 (or tighter), not world-accessible
- Journal shows "rustyjackd ready" and no crash-loop
```

Notes:

Fail fast here; all subsequent RPC tests assume the socket is reachable.

Suite B: systemd hardening posture (static + live)

Purpose: Validate sandboxing directives and produce a hardening score/report for audit trails.

Steps:

```
1) systemd-analyze security rustyjackd.service
2) systemctl show rustyjackd -p CapabilityBoundingSet,AmbientCapabilities,NoNewPrivileges,ProtectSyst
3) Read /proc/<MainPID>/status (CapEff, CapBnd, NoNewPrivs)
```

Expected pass criteria:

```
- systemd-analyze security runs and is saved to systemd/security.txt
- NoNewPrivileges is enabled (or intentionally justified)
- CapEff does not contain unexpected capabilities beyond what unit declares
- ProtectSystem=strict with explicit ReadWritePaths for required dirs
```

Notes:

Use systemd-analyze security as the primary “one number” posture check. The output is advisory but useful for regression detection.

Suite C: UDS permissions and group boundary checks

Purpose: Confirm the OS-level and daemon-level boundaries match your intended trust model (root/admin/operator/read-only).

Steps:

```
1) Ensure groups exist: rustyjack, rustyjack-admin
2) Create ephemeral users (cleanup on exit):
   - rjdiag_ro: in no rustyjack groups
```

```
   - rjdiag_op: in group rustyjack
   - rjdiag_admin: in groups rustyjack and rustyjack-admin
3) As each user:
   - attempt to connect to socket
   - perform Health + Version RPC
   - read HelloAck.authz.role and assert expected tier
4) Attempt an Operator-only endpoint as rjdiag_ro and expect:
   - connect may fail at filesystem permissions OR
   - daemon may accept but must return Forbidden
```

Expected pass criteria:

```
- rjdiag_admin: role=Admin
- rjdiag_op:    role=Operator
- rjdiag_ro:    role=ReadOnly (or cannot connect at all)
- Admin-only endpoints (e.g., logging_config_set) are Forbidden for Operator/ReadOnly
```

Suite D: Protocol robustness (negative tests)

Purpose: Ensure the daemon rejects malformed frames/JSON, enforces max_frame, and disconnects on repeated protocol violations without crashing.

Steps:

```
1) Invalid hello JSON
2) hello.protocol_version mismatch (e.g., 999)
3) Empty frame (len=0) and too-large frame (len=max_frame+1)
4) Valid hello, then send invalid request JSON 3 times and confirm disconnect
5) Send endpoint/body mismatch and confirm BadRequest
6) Handshake timeout: open socket and never send hello
7) Read timeout: complete handshake, then remain idle > read_timeout and confirm Timeout error
```

Expected pass criteria:

```
- Daemon returns structured Err responses where applicable (BadRequest/IncompatibleProtocol/Timeout)
- After 3 protocol violations, daemon drops connection
- Service remains active (systemctl is-active) and does not leak file descriptors
```

Suite E: Safe functional smoke tests (read-only and non-disruptive)

Purpose: Exercise the minimum set of endpoints that should always be safe on a live Pi without changing network state.

Steps:

```
Call these endpoints and validate response schema + basic invariants:
- Health, Version, Status
- SystemStatusGet
- DiskUsageGet { path: "/" }
- BlockDevicesList (read-only)
- ActiveInterfaceGet
- InterfaceStatusGet { interface: <detected active if present> }
- WifiInterfacesList
- WifiCapabilitiesGet { interface: <detected wifi if present> }
- PortalStatus
- MountList
- LoggingConfigGet
```

Expected pass criteria:

```
- All endpoints return ResponseOk with expected fields
- No endpoint returns Internal errors under normal conditions
- Latency: Health/Version should be “fast” (< ~100ms locally); log actual timings
```

Suite F: Job subsystem reliability (safe jobs only by default)

Purpose: Validate job start/status/cancel flows, retention behavior, and state transitions.

Steps:

```
1) Start Noop job:
   JobStart { job: { kind: {type:"Noop"}, requested_by:"diag" } }
2) Poll JobStatus until state=completed/failed
3) Start Sleep job (e.g., 2s), poll status, then cancel and confirm state=cancelled
4) Attempt to start a dangerous job kind (SystemUpdate) when dangerous_ops is disabled and expect For
```

Expected pass criteria:

```
- JobStart returns job_id
- JobStatus returns coherent JobInfo, state transitions are monotonic
- Cancelled jobs do not continue running
- Dangerous jobs are correctly gated
```

Suite G: Logging and observability checks

Purpose: Verify you can retrieve meaningful diagnostics when something fails.

Steps:

```
1) journalctl -u rustyjackd --since <start> (save full log)
2) RPC: LogTailGet { component:"rustyjackd", max_lines:200 }
3) If UI is running as another service, optionally capture its journal too
4) Verify log directory permissions under root path (e.g., /var/lib/rustyjack/logs)
```

Expected pass criteria:

```
- LogTailGet returns lines and truncated flag is consistent
- On failures, the suite points to the specific request_id and log line context
```

Suite H: Stress and soak (non-destructive)

Purpose: Detect race conditions, timeouts, memory leaks, and file descriptor leaks under load.

Steps:

```
1) Burst RPC: 200 Health requests sequentially; record p50/p95 latency
2) Parallel RPC: spawn N clients (e.g., 25) concurrently issuing Health/Version
3) Connection churn: open/handshake/close loop (e.g., 500 times)
4) FD leak check: compare lsof/ /proc/<pid>/fd count before and after (if lsof present)
5) CPU/memory: sample /proc/<pid>/statm or ps RSS during stress
```

Expected pass criteria:

```
- No growth trend in fd count or RSS beyond small noise
- No crash or restart
- Error rates remain near zero under configured limits
```

Suite I: Security adversarial tests (explicit, reproducible)

Purpose: Probe for privilege escalation and authorization weaknesses in the group-based auth implementation.

Steps:

```
Test I1: “PID disappears” group lookup fallback
- Run as a user NOT in rustyjack or rustyjack-admin.
- Connect to UDS, then fork:
  parent: connect(); fork(); parent exits immediately
  child: keeps socket open and performs an Operator-only call (e.g., SystemLogsGet or WifiScanStart)
```

```
- If daemon uses peer.pid from SO_PEERCRED (connect-time) and tries to read /proc/<pid>/status after
  group lookup can fail and code currently falls back to Operator.

Test I2: Endpoint tier enforcement
- For each Admin-only endpoint, assert Operator/ReadOnly receives Forbidden.

Test I3: Protocol abuse boundaries
- Attempt repeated oversized frames and ensure daemon doesn't OOM or wedge.
```

Expected pass criteria:

```
- I1 MUST NOT allow privilege escalation. If it does, treat as a critical vulnerability.
- Tier enforcement is consistent with required_tier() matrix.
```

Optional Suite J: Disruptive/“dangerous” functional tests

Only run with `--dangerous`. These tests can disrupt connectivity or modify the system. They are valuable for validation on a lab device but should not be the default.

```
Examples (gate behind flags):
- WiFiConnectStart / WifiDisconnect on a test SSID
- HotspotStart/Stop on a dedicated interface
- PortalStart/Stop with a test port
- MountStart/UnmountStart on a disposable USB device
- SystemSync (safe-ish); NEVER auto-run SystemReboot/SystemShutdown

All dangerous tests must include:
- Precondition snapshot
- Postcondition restoration (best effort)
- Explicit logs showing what changed
```

Pseudocode: bash harness + embedded python RPC

```
\
Bash skeleton (high level)

set -euo pipefail

RUN_ID="$(date +%Y%m%d-%H%M%S)"
OUT="/var/tmp/rustyjackd-diag/$RUN_ID"
mkdir -p "$OUT"/{systemd,rpc/requests,rpc/responses}

log()   { printf '%s %s\n' "$(date -Is)" "$*" | tee -a "$OUT/diag.log" ; }
fail()  { log "[FAIL] $*"; return 1; }
pass()  { log "[PASS] $*"; }

run_cmd() {
  local name="$1"; shift
  log "[CMD] $name :: $*"
  if "$@" >>"$OUT/diag.log" 2>&1; then pass "$name"; else fail "$name (rc=$?)"; fi
}

# rj_rpc wraps python and writes request/response files
rj_rpc() {
  local id="$1" body_type="$2" data_json="$3" user="${4:-root}"
  local req="$OUT/rpc/requests/${id}_${body_type}.json"
  local resp="$OUT/rpc/responses/${id}_${body_type}.json"

  printf '%s\n' "$data_json" > "$req"

  # run python as requested user; capture JSON response
  sudo -u "$user" python3 - "$SOCKET" "$body_type" "$req" > "$resp"
  # python exits non-zero on Err; harness records failure details
}

Test pattern (table-driven)
tests=(
  "0001 Health null root"
  "0002 Version null rjdiag_ro"
  "0003 LoggingConfigSet '{\"enabled\":true,\"level\":\"info\"}' rjdiag_op  # should fail (Forbidden)
)

for t in "${tests[@]}"; do
  read -r id bt data user <<<"$t"
  if rj_rpc "$id" "$bt" "$data" "$user"; then pass "$id $bt"; else fail "$id $bt"; fi
done
```

Python helper notes (inside the .sh file)

```
Key behaviors to implement:
- Encode frame: len(payload) as 4-byte big-endian + payload
- Decode frame: read 4 bytes, parse u32 big-endian, bounds-check, then read payload
- Use socket.settimeout() for connect/read/write timeouts
- Print a single JSON object to stdout:
  { "ok": true|false, "request_id": ..., "endpoint": ..., "body": ..., "raw": ... , "timing_ms": ... }

Also emit explicit error classification:
- connect_error (filesystem permissions)
- handshake_timeout, read_timeout, write_timeout
- protocol_error (bad JSON / mismatch)
- daemon_error (Err response with code/message)
```

References (external)

[1] systemd-analyze(1) man page (includes "systemd-analyze security"): man7.org
[2] systemd.exec(5) (ProtectSystem, ReadWritePaths, sandboxing directives): freedesktop.org
[3] unix(7) man page (SO_PEERCRED semantics): man7.org
[4] SUSE: "Securing systemd Services" (uses systemd-analyze security): documentation.suse.com
[5] ArchWiki: systemd/Sandboxing (overview and rationale): wiki.archlinux.org