RustDesk: Automatisierte Geräte-Bereinigung (Device Cleaner)

Dieses Python-Skript dient der automatisierten Wartung des RustDesk Server Pro. Es verhindert, dass das **Gerätelimit der Lizenz** durch nicht mehr genutzte oder einmalige Support-Sitzungen blockiert wird.

Hintergrund & Problemstellung

Bei der Nutzung von RustDesk Server Pro (insbesondere mit Standard-Lizenzen) tritt häufig ein Problem auf, das auch in der **GitHub Discussion #182** thematisiert wird:

> **Das Problem: "Quick Support" Lizenz-Verbrauch**
>
> Jedes Gerät, das sich mit dem Server verbindet, verbraucht einen Lizenz-Slot. Dies gilt auch für **Quick-Support-Kunden**, denen nur einmalig geholfen wird (z.B. via "Nur Ausführen" des Clients ohne Installation).
>
> - Diese Geräte verbleiben oft als "ungruppierte" Einträge im Server.
> - Die Geräteliste füllt sich schnell mit **inaktiven Einträgen**.
> - Das **Lizenzlimit wird erreicht**, und neue (legitime) Geräte können nicht mehr registriert werden.
> - Der Admin muss diese veralteten Datensätze normalerweise mühsam manuell löschen.

Referenz: GitHub Discussion #182: Delete devices via API

Die Lösung

Dieses Skript automatisiert den Bereinigungsprozess. Es identifiziert Geräte, die **keiner Gruppe zugewiesen** sind oder seit **X Tagen offline** sind, und entfernt diese aus der Lizenzverwaltung.

Da die RustDesk API ein direktes Löschen nicht erlaubt, führt das Skript den notwendigen zweistufigen Prozess automatisch durch:

1. **Disable:** Das Gerät wird deaktiviert.
2. **Delete:** Das Gerät wird endgültig gelöscht und der Lizenz-Slot wird frei.

Funktionen

- **Lizenz-Optimierung:** Hält Slots für aktive Geräte frei.
- **Filterung:** Zielt spezifisch auf "Einmal-Kunden" (ohne Gruppe) oder Altgeräte (Offline > X Tage).
- **Safety First:** Standardmäßig ist ein `DRY_RUN` (Simulationsmodus) aktiv.
- **Logging:** Protokolliert Aktionen in `rustdesk_cleanup.log`.

Installation & Vorbereitung

Das Skript benötigt Python 3 und die `requests` Bibliothek.

1. Abhängigkeiten installieren (Debian/Ubuntu):

```
sudo apt update && sudo apt install python3 python3-pip
pip3 install requests
```

2. Skript ablegen:

Erstelle eine Datei, z.B. unter `/opt/scripts/rustdesk_cleaner.py` und füge den Python-Code (siehe Abschnitt unten) ein.

3. Konfiguration anpassen:

Öffne das Skript und passe den oberen Konfigurationsbereich an deine Umgebung an:

```
API_URL = "https://your-rustdesk-server.com"
API_TOKEN = "your-bearer-token-here"
```

> **Hinweis:** Den API Token erhältst du im RustDesk Web-Interface unter Einstellungen -> API.

Nutzung

Manueller Testlauf (Dry Run)

Standardmäßig führt das Skript keine Änderungen durch (Simulation). Um zu sehen, welche Geräte gelöscht würden:

```
python3 rustdesk_cleaner.py delete
```

Manuelle Ausführung (Ernstfall)

Um die Geräte wirklich zu löschen, muss das Flag `--no_dry_run` gesetzt werden. Dies ist der Befehl, um z.B. alle Geräte ohne Gruppe endgültig zu entfernen:

```
python3 rustdesk_cleaner.py delete --no_dry_run
```

Weitere Parameter

Das Skript akzeptiert diverse Argumente zur Steuerung:

Parameter	Beschreibung
`--offline_days X`	Löscht Geräte, die länger als X Tage offline sind.
`--no_group`	(Standard) Löscht Geräte, die keiner Gruppe zugewiesen sind (ideal für Quick Support).
`--only_disable`	Deaktiviert die Geräte nur, löscht sie aber nicht.
`--yes`	Bestätigt die Sicherheitsabfrage automatisch (für Skripte notwendig).

Beispiel: Alte Geräte löschen Löschen von veralteten Datensätzen, die länger als 90 Tage offline waren:

```
python3 rustdesk_cleaner.py delete --offline_days 90 --no_dry_run
```

Automatisierung (Cronjob)

Um das Skript regelmäßig (z.B. jede Nacht um 03:00 Uhr) laufen zu lassen, kann ein Cronjob eingerichtet werden.

1. Crontab öffnen: `crontab -e`
2. Zeile hinzufügen:

```
# RustDesk Cleanup: Löscht Geräte ohne Gruppe täglich um 03:00 Uhr
0 3 * * * /usr/bin/python3 /opt/scripts/rustdesk_cleaner.py delete --no_dry_run --yes >>
/var/log/rustdesk_cron.log 2>&1
```

> **Wichtig:** Das Flag `--yes` ist im Cronjob notwendig, um Sicherheitsabfragen bei der Massenlöschung automatisch zu bestätigen.

Quellcode (Python)

Source Code on GitHub

```
#!/usr/bin/env python3

"""
RustDesk Device Cleaner
=======================

This script manages devices on a RustDesk API server. It can view,
disable, enable, or delete devices based on filters such as group
membership or offline duration.

Prerequisites & Installation:
-----------------------------
1. Install Python 3 (if not already present):
   - Linux (Ubuntu/Debian): sudo apt update && sudo apt install python3 python3-pip
   - Windows: Download from python.org
2. Install required 'requests' library:
```

```
   pip install requests

How it works:
-------------
1. The script fetches all devices from the server via API.
2. It filters the list (e.g., 'No Group' or 'X days offline').
3. It performs the requested action (view, disable, enable, delete).
4. For 'delete' operations, a device MUST be DISABLED first (MANDATORY RustDesk API
   requirement), then it is DELETED.
5. All actions are logged to the file 'rustdesk_cleanup.log'.

How to create a Cronjob:
-----------------------
1. Open your crontab editor:
   crontab -e
2. Add a line at the end of the file.
   Format: [minute] [hour] [day] [month] [day_of_week] [command]

   Example (Delete ungrouped devices every day at 01:00 AM):
   0 1 * * * /usr/bin/python3 /absolute/path/to/rustdesk_cleaner.py delete

3. Save and exit. The script will now run automatically.

Important Override Parameters (CLI):
-----------------------------------
--offline_days X : Overrides config with X days.
--no_group       : Forces filtering for ungrouped clients.
--dry_run        : Forces simulation mode.
--no_dry_run     : Forces execution even if DRY_RUN=True is set in the script.
--yes            : Confirms actions immediately (required for cron).
--only_disable   : Only disables clients without deleting them.
--disable_before_delete: (Default: True) Ensures mandatory disabling before deletion.
"""

import requests
import argparse
import logging
from datetime import datetime, timedelta

# --- CONFIGURATION ---
DRY_RUN = True           # True = Simulation only | False = Real deletion/disabling
```

```
AUTO_CONFIRM = True      # True = Automatically confirm multiple devices (required for cron)
API_URL = "https://your-rustdesk-server.com"
API_TOKEN = "your-bearer-token-here"

# Filter Options (Default values for automated runs)
DELETE_UNGROUPED = True  # True = Only target devices without a group
OFFLINE_DAYS = None      # None = Disable filter | Integer (e.g. 3) = Only devices offline for
X days

# Process Options
# NOTE: RustDesk requires devices to be DISABLED before they can be DELETED.
DISABLE_BEFORE_DELETE = True  # Must stay True for successful deletion in RustDesk
ONLY_DISABLE = False          # Set to True to only disable clients without deleting them
# ---------------------



def view(
    url,
    token,
    id=None,
    device_name=None,
    user_name=None,
    group_name=None,
    device_group_name=None,
    offline_days=None,
    no_group=False,
):
    """
    Fetches and filters devices from the RustDesk server.

    Filters:
    - offline_days: Only includes devices offline for at least X days.
    - no_group: Only includes devices that are not assigned to any group.
    """
    headers = {"Authorization": f"Bearer {token}"}
    pageSize = 100
    params = {
        "id": id,
        "device_name": device_name,
        "user_name": user_name,
        "group_name": group_name,
```

```
    "device_group_name": device_group_name,
}

# Add wildcards to search parameters if not already present
params = {
    k: "%" + v + "%" if (v != "-" and "%" not in v) else v
    for k, v in params.items()
    if v is not None
}
params["pageSize"] = pageSize

devices = []
current = 0

# Paginated API requests
while True:
    current += 1
    params["current"] = current
    response = requests.get(f"{url}/api/devices", headers=headers, params=params)
    if response.status_code != 200:
        print(f"Error: HTTP {response.status_code} - {response.text}")
        exit(1)

    response_json = response.json()
    if "error" in response_json:
        print(f"Error: {response_json['error']}")
        exit(1)

    data = response_json.get("data", [])

    # Apply custom filters locally
    for device in data:
        # 1. Filter by offline duration
        if offline_days is not None:
            last_online_str = device.get("last_online")
            if not last_online_str:
                continue
            last_online = datetime.strptime(
                last_online_str.split(".")[0], "%Y-%m-%dT%H:%M:%S"
            )
            if (datetime.utcnow() - last_online).days < offline_days:
```

```
                    continue

            # 2. Filter by group membership
            if no_group:
                # If a group name is present, the device is not 'ungrouped'
                if device.get("device_group_name"):
                    continue

            devices.append(device)

        total = response_json.get("total", 0)
        # Check if we've reached the end of the list
        if len(data) < pageSize or current * pageSize >= total:
            break

    return devices


def check(response):
    if response.status_code != 200:
        print(f"Error: HTTP {response.status_code} - {response.text}")
        exit(1)

    try:
        response_json = response.json()
        if "error" in response_json:
            print(f"Error: {response_json['error']}")
            exit(1)
        return response_json
    except ValueError:
        return response.text or "Success"


def disable(url, token, guid, id):
    """Sends a request to disable a device by its GUID."""
    print("Disable", id)
    headers = {"Authorization": f"Bearer {token}"}
    response = requests.post(f"{url}/api/devices/{guid}/disable", headers=headers)
    return check(response)
```

```
def enable(url, token, guid, id):
    """Sends a request to enable a device by its GUID."""
    print("Enable", id)
    headers = {"Authorization": f"Bearer {token}"}
    response = requests.post(f"{url}/api/devices/{guid}/enable", headers=headers)
    return check(response)


def delete(url, token, guid, id):
    """Sends a request to delete a device by its GUID."""
    print("Delete", id)
    headers = {"Authorization": f"Bearer {token}"}
    response = requests.delete(f"{url}/api/devices/{guid}", headers=headers)
    return check(response)


def assign(url, token, guid, id, type, value):
    print("assign", id, type, value)
    valid_types = [
        "ab",
        "strategy_name",
        "user_name",
        "device_group_name",
        "note",
        "device_username",
        "device_name",
    ]
    if type not in valid_types:
        print(f"Invalid type, it must be one of: {', '.join(valid_types)}")
        return
    data = {"type": type, "value": value}
    headers = {"Authorization": f"Bearer {token}"}
    response = requests.post(
        f"{url}/api/devices/{guid}/assign", headers=headers, json=data
    )
    return check(response)


def main():
    parser = argparse.ArgumentParser(description="Device manager")
    parser.add_argument(
```

```
        "command",
        choices=["view", "disable", "enable", "delete", "assign"],
        help="Command to execute",
    )
    parser.add_argument("--url", default=API_URL, help=f"URL of the API (Default from config: {API_URL})")
    parser.add_argument(
        "--token", default=API_TOKEN, help="Bearer token for authentication (Default from config: ***)"
    )
    parser.add_argument("--id", help="Device ID")
    parser.add_argument("--device_name", help="Device name")
    parser.add_argument("--user_name", help="User name")
    parser.add_argument("--group_name", help="User group name")
    parser.add_argument("--device_group_name", help="Device group name")
    parser.add_argument(
        "--assign_to",
        help="<type>=<value>, e.g. user_name=mike, strategy_name=test, device_group_name=group1, note=note1, device_username=username1, device_name=name1, ab=ab1, ab=ab1,tag1,alias1,password1,note1"
    )
    parser.add_argument(
        "--offline_days", type=int, default=OFFLINE_DAYS, help=f"Offline duration in days (Default from config: {OFFLINE_DAYS})"
    )
    parser.add_argument(
        "--no_group", action="store_true", default=DELETE_UNGROUPED, help=f"Only target devices with no device group (Default from config: {DELETE_UNGROUPED})"
    )
    parser.add_argument(
        "--dry_run", action="store_true", default=DRY_RUN, help=f"Simulate the operations (Current default: {DRY_RUN})"
    )
    # Allow explicit disabling if the constant is True
    parser.add_argument(
        "--no_dry_run", action="store_false", dest="dry_run", help="Force execution even if DRY_RUN is set to True in script"
    )
    parser.add_argument(
        "--log_file", default="rustdesk_cleanup.log", help="Path to the log file (default: rustdesk_cleanup.log)"
```

```
    )

    parser.add_argument(
        "--yes", "-y", action="store_true", default=AUTO_CONFIRM, help=f"Confirm the operation
without prompting (Current default: {AUTO_CONFIRM})"
    )
    parser.add_argument(
        "--only_disable", action="store_true", default=ONLY_DISABLE, help=f"Only disable
clients, do not delete them (Current default: {ONLY_DISABLE})"
    )
    parser.add_argument(
        "--disable_before_delete", action="store_true", default=DISABLE_BEFORE_DELETE,
help=f"Ensure devices are disabled before deletion (Current default: {DISABLE_BEFORE_DELETE})"
    )

    args = parser.parse_args()

    # Configure logging
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s - %(levelname)s - %(message)s",
        handlers=[
            logging.FileHandler(args.log_file),
            logging.StreamHandler()
        ]
    )

    logger = logging.getLogger(__name__)

    while args.url.endswith("/"): args.url = args.url[:-1]

    devices = view(
        args.url,
        args.token,
        args.id,
        args.device_name,
        args.user_name,
        args.group_name,
        args.device_group_name,
        args.offline_days,
        args.no_group,
```

```
    )

    if args.command == "view":
        for device in devices:
            print(device)
    elif args.command in ["disable", "enable", "delete", "assign"]:
        # Safety check for multiple devices
        if len(devices) > 1 and not args.yes and not args.dry_run:
            logger.warning(f"Found {len(devices)} devices. Operation '{args.command}' requires
--yes or -y flag for multiple devices without interaction.")
            print(f"Found {len(devices)} devices. Use --yes or -y to confirm this operation in
a script.")
            return

        if args.command == "disable":
            for device in devices:
                if args.dry_run:
                    logger.info(f"[Dry Run] Would disable device: {device['id']} (GUID:
{device['guid']})")
                else:
                    response = disable(args.url, args.token, device["guid"], device["id"])
                    logger.info(f"Disabled device {device['id']}: {response}")
        elif args.command == "enable":
            for device in devices:
                if args.dry_run:
                    logger.info(f"[Dry Run] Would enable device: {device['id']} (GUID:
{device['guid']})")
                else:
                    response = enable(args.url, args.token, device["guid"], device["id"])
                    logger.info(f"Enabled device {device['id']}: {response}")
        elif args.command == "delete":
            for device in devices:
                if args.dry_run:
                    action = "disable" if args.only_disable else "disable and delete"
                    logger.info(f"[Dry Run] Would {action} device: {device['id']} (GUID:
{device['guid']})")
                else:
                    # MANDATORY RUSTDESK LOGIC: A client MUST be disabled before it can be
deleted.
                    logger.info(f"Processing device {device['id']}. Disabling first (required
for deletion)...")
```

```
                    disable_response = disable(args.url, args.token, device["guid"],
device["id"])
                    logger.info(f"Disable response for {device['id']}: {disable_response}")

                    if args.only_disable:
                        logger.info(f"ONLY_DISABLE is active. Skipping deletion for
{device['id']}.")
                    else:
                        # Proceeding to final deletion
                        logger.info(f"Proceeding to delete device {device['id']}...")
                        delete_response = delete(args.url, args.token, device["guid"],
device["id"])
                        logger.info(f"Delete response for {device['id']}: {delete_response}")
        elif args.command == "assign":
            if "=" not in args.assign_to:
                logger.error("Invalid assign_to format, it must be <type>=<value>")
                return
            type, value = args.assign_to.split("=", 1)
            for device in devices:
                if args.dry_run:
                    logger.info(f"[Dry Run] Would assign {type}={value} to device:
{device['id']}")
                else:
                    response = assign(
                        args.url, args.token, device["guid"], device["id"], type, value
                    )
                    logger.info(f"Assigned {type}={value} to {device['id']}: {response}")



if __name__ == "__main__":
    main()
```

Revision #3
Created 2026-02-14 14:29:20 UTC by Smok3y
Updated 2026-02-14 18:49:30 UTC by Smok3y