PatchMon PATCHMON BLOG

Managing Patches Across Mixed Linux Distributions: A Practical Guide

How to keep Ubuntu, RHEL, Alpine, Arch, and FreeBSD servers patched when your fleet runs five different package managers - without losing your mind.

AUTHOR

PatchMon Team

PUBLISHED

19 March 2026

CATEGORIES

Tutorials, Security

READ ONLINE

https://patchmon.net/blog/managing-patches-mixed-linux-distributions

Contents

You open your terminal on a Monday morning and SSH into your fleet. There are 14 Ubuntu 22.04 servers running your main application, 6 Rocky Linux 9 boxes inherited from the team that got absorbed last year, a pair of CentOS 7 machines that "nobody touch because billing depends on them," 40-something Alpine containers across three Kubernetes clusters, one Arch Linux box that a developer set up for "testing" two years ago and somehow became production, and a FreeBSD firewall that predates your employment.

Every single one of them needs patches. Each one uses a different package manager with different flags, different output formats, different ideas about what constitutes a "security update," and different opinions about whether a reboot is necessary afterward.

This is not a hypothetical scenario. This is Tuesday.

This guide is for the sysadmin who just inherited that fleet, or the one who has been managing it for years and is tired of the duct tape. We will walk through the real differences between package managers, show you what a DIY approach looks like (and where it falls apart), and lay out a strategy that actually holds together at scale.

How Mixed Environments Happen

Nobody sits down on day one and says, "Let's run five different operating systems in production." Mixed environments happen through completely normal, completely unavoidable circumstances:

Acquisitions and mergers. Company A runs Ubuntu. Company B runs RHEL. Now you run both.

Team autonomy. The database team standardized on Rocky Linux. The web team likes Debian. The security team deployed a FreeBSD appliance. Nobody coordinated.

Legacy systems. CentOS 7 hit end-of-life in June 2024, but the billing system on it works and nobody wants to touch the migration.

Containers. Alpine is the default base image for half the Docker ecosystem. Your hosts might be Ubuntu, but your containers are Alpine.

Specialized workloads. FreeBSD for firewalls and network appliances. Arch for that one developer's pet project that became load-bearing infrastructure.

Vendor requirements. Some commercial software is only certified on RHEL. Your monitoring stack requires Ubuntu. Your storage appliance runs its own Linux.

The reality is that heterogeneous environments are the norm, not the exception. The question is not how to avoid them but how to manage them without losing your sanity or your weekends.

Package Manager Deep Dive

Before you can manage patches across distributions, you need to understand what each package manager actually does and how they differ. This is not a surface-level comparison. These differences will bite you when you try to automate.

Comparison Table

Feature	APT (Debian/Ubuntu)	DNF/YUM (RHEL/Rocky/Alma)	APK (Alpine)	Pac (Ar
Check for updates	`apt update && apt list --upgradable`	`dnf check-update`	`apk update && apk list -u`	`pa && Qu`
Apply all updates	`apt upgrade -y`	`dnf update -y`	`apk upgrade`	`pa --n`
Security-only updates	`apt upgrade -y -o Dir::Etc::sourcelist=/etc/apt/security.sources` or `unattended-upgrades`	`dnf update --security -y`	Not natively supported	Not sup
List security advisories	`apt list --upgradable` (limited)	`dnf updateinfo list security`	Parse `secfixes` in APKBUILD	`ar` (thi
Automatic updates	`unattended-upgrades`	`dnf-automatic`	`apk upgrade` via cron	Not rec
Exit code on updates available	`apt list --upgradable` returns 0 either way	`dnf check-update` returns 100	`apk list -u` returns 0 either way	`pa` retu nor
Rollback support	Snapshot-based only	`dnf history undo`	Snapshot-based only	`do` pac

Read that exit code row carefully. Three different package managers, three different conventions for "are there updates available?" This single inconsistency has broken more automation scripts than any other difference.

APT (Debian, Ubuntu)

APT is the most common package manager you will encounter in mixed environments because Ubuntu dominates cloud server deployments. The key things to know:

```
# Refresh package lists
apt update

# List available upgrades
apt list --upgradable

# Apply all upgrades
apt upgrade -y

# Apply only security updates (Ubuntu with unattended-upgrades)
unattended-upgrades --dry-run -d

# Check what unattended-upgrades would install
unattended-upgrades -v --dry-run
```

Ubuntu separates security updates into their own repository (`-security` suffix). You can target these specifically. The `unattended-upgrades` package is the standard tool for automated security patching, configured in `/etc/apt/apt.conf.d/50unattended-upgrades`:

```
Unattended-Upgrade::Allowed-Origins {
  "${distro_id}:${distro_codename}-security";
};
Unattended-Upgrade::Automatic-Reboot "false";
Unattended-Upgrade::Mail "admin@example.com";
```

Watch out for: Debian does not use the same repository naming convention as Ubuntu. If you write automation that targets `jammy-security`, it will not work on Debian `bookworm`. Debian uses `bookworm-security` and the repository structure is slightly different.

DNF/YUM (RHEL, CentOS, Rocky Linux, AlmaLinux, Fedora)

The RHEL family has the most mature security advisory system of any Linux distribution. DNF (which replaced YUM in RHEL 8) has first-class support for filtering updates by advisory type:

```
# Check for all available updates
dnf check-update

# List security advisories
dnf updateinfo list security

# Show details of a specific advisory
dnf updateinfo info RHSA-2026:0142

# Apply only security updates
dnf update --security -y

# Apply a specific advisory
dnf update --advisory=RHSA-2026:0142 -y
```

The `dnf-automatic` service handles automated patching:

```
# /etc/dnf/automatic.conf
[commands]
upgrade_type = security
apply_updates = yes

[emitters]
emit_via = email
email_from = root@localhost
email_to = admin@example.com
```

```
systemctl enable --now dnf-automatic-install.timer
```

Watch out for: CentOS 7 uses `yum`, not `dnf`. The commands are similar but not identical. CentOS 7 also reached end-of-life in June 2024, which means no more security updates from the official repositories. If you still have CentOS 7 in production, your patching problem is actually a migration problem.

Also note: `dnf check-update` returns exit code 100 when updates are available and 0 when there are none. This is the opposite of what most people expect, and it will cause your shell scripts to exit prematurely if you have `set -e` enabled.

APK (Alpine Linux)

Alpine is everywhere in containers and nowhere in traditional server deployments. Its package manager is fast and minimal, like everything else in Alpine:

```
# Refresh package lists
apk update

# List available upgrades
apk list -u

# Upgrade all packages
apk upgrade

# Upgrade a specific package
apk add --upgrade openssl
```

The security update problem: Alpine does not have a dedicated security update channel. There is no `apk upgrade --security`. Alpine tracks security fixes in `secfixes` metadata within APKBUILD files, but the package manager itself does not expose this as a filter.

To identify security-relevant updates, you have a few options:

```
# Check for known vulnerabilities in installed packages
# (requires the alpine-sdk or checking secfixes database manually)
apk audit --system

# Or check the Alpine security tracker
# https://security.alpinelinux.org/
```

In practice, most teams running Alpine containers simply rebuild images with `apk upgrade` on a regular schedule and redeploy. The container model makes this more tractable than in-place patching.

Watch out for: Alpine uses musl libc instead of glibc. This occasionally means that CVEs affecting glibc do not apply to Alpine, and vice versa. Your vulnerability scanner may not account for this correctly.

Pacman (Arch Linux)

If you have Arch in production, you already know what you signed up for. Arch is a rolling release distribution with no concept of "security updates" as a separate category:

```
# Sync package database and upgrade all packages
pacman -Syu

# Check for available updates without installing
pacman -Qu

# Install the arch-audit tool for CVE tracking
pacman -S arch-audit

# Check installed packages against known CVEs
arch-audit
```

The `arch-audit` tool queries the Arch Linux security tracker and tells you which installed packages have known vulnerabilities:

```
$ arch-audit
Package openssl is affected by CVE-2026-XXXX. Medium risk.
Package linux is affected by CVE-2026-YYYY. High risk.
```

Watch out for: Arch does partial upgrades poorly. Running `pacman -Sy package` (sync and install one package) without upgrading everything can break your system due to shared library mismatches. In Arch, it is all or nothing. This makes maintenance windows more complex because you cannot cherry-pick security patches.

FreeBSD pkg and freebsd-update

FreeBSD is not Linux, but it shows up in mixed environments often enough to warrant inclusion. FreeBSD has two separate update mechanisms:

```
# Update the package repository catalog
pkg update

# Check for known vulnerabilities in installed packages
pkg audit -F

# Upgrade all packages
pkg upgrade -y

# For base system updates (kernel, userland)
freebsd-update fetch
freebsd-update install
```

The `pkg audit` command is one of the best vulnerability checking tools across any platform. It checks installed packages against the VuXML (Vulnerability and eXposure Markup Language) database and gives you clear, actionable output:

```
$ pkg audit -F
openssl-3.0.12 is vulnerable:
  OpenSSL: Buffer overread in DTLS
  CVE: CVE-2026-XXXX
  WWW: https://vuxml.FreeBSD.org/freebsd/xxxxxxxx.html
```

Watch out for: `freebsd-update` only works for RELEASE versions, not CURRENT or STABLE. If you are running FreeBSD from source, you need to manage base system updates through `svn` or `git` and recompile.

Security Update Identification: The Real Problem

The biggest challenge in multi-distro patch management is not running the update command. It is answering the question: "Which of my servers have unpatched security vulnerabilities right now?"

Each distribution handles security metadata differently:

Ubuntu/Debian: Security updates come from a separate repository. You can identify them by source. Reasonably reliable.

RHEL/Rocky/Alma: Full advisory system with RHSA (security), RHBA (bug fix), and RHEA (enhancement) classifications. The gold standard.

Alpine: No native security classification. You have to cross-reference the secfixes database or the Alpine security tracker yourself.

Arch: No classification at all. Everything is just "an update." Use `arch-audit` as a separate step.

FreeBSD: `pkg audit` gives you vulnerability data, but it is separate from the upgrade process. You know what is vulnerable, but there is no "upgrade only vulnerable packages" command.

This inconsistency is the fundamental reason why managing a mixed fleet with shell scripts becomes painful. You are not just running different commands; you are working with fundamentally different information models.

The DIY Approach (And Where It Breaks)

Every sysadmin's first instinct is to write a script. Here is what a multi-distro patching script looks like in its early stages:

```
#!/bin/bash
# multi-patch.sh - Check for updates across different distros

set -euo pipefail

check_updates() {
  local host=$1
  local os
  os=$(ssh "$host" "cat /etc/os-release 2>/dev/null | grep ^ID= | cut -d= -f2 | tr
-d '\"'")

  case "$os" in
    ubuntu|debian)
    ssh "$host" "apt update -qq 2>/dev/null && apt list --upgradable 2>/dev/null"
    ;;
    rocky|almalinux|rhel|centos)
    # Remember: exit code 100 means updates available
    ssh "$host" "dnf check-update 2>/dev/null" || true
    ;;
    alpine)
    ssh "$host" "apk update -q && apk list -u 2>/dev/null"
    ;;
    arch)
    ssh "$host" "pacman -Sy -q && pacman -Qu 2>/dev/null"
    ;;
    *)
    echo "Unknown OS on $host: $os"
    ;;
  esac
}

# Read hosts from a file
while IFS= read -r host; do
  echo "=== $host ==="
  check_updates "$host"
  echo ""
done < /etc/patch-hosts.txt
```

This works on your laptop. Then you deploy it and discover:

Output parsing is a nightmare. `apt list --upgradable` outputs `package/repo version [upgradable from: old_version]`. `dnf check-update` outputs `package.arch version repo`. `apk list -u` outputs `package-version description`. Good luck writing a consistent parser.

Error handling across SSH is fragile. Network timeouts, host key changes, sudo prompts, locked package managers from unattended-upgrades already running. Each failure mode needs handling.

No audit trail. When your security team asks "when was CVE-2026-XXXX patched on prod-db-03?" you have nothing but grep through your terminal scrollback.

No rollback information. Something broke after patching. Which packages were updated? What were the previous versions? Hope you logged that.

FreeBSD is not in `/etc/os-release`. Your OS detection logic fails immediately. FreeBSD uses `freebsd-version` or `uname`.

Containers do not have SSH. Your 40 Alpine containers are not reachable via SSH. They need a completely different patching strategy (rebuild and redeploy).

Scaling problems. Running this sequentially across 60 hosts takes 20 minutes. Running it in parallel means interleaved output. Adding proper parallelism with output collection adds another 50 lines of bash that is hard to debug.

The script grows. It gains config files, logging, a database (often a CSV file), email notifications, and retry logic. After six months, it is 800 lines of bash that only one person understands, and that person is trying to take a vacation.

This is not a criticism. It is a description of the natural lifecycle of infrastructure automation. The script served its purpose, but the problem outgrew it.

Strategies for Keeping Your Sanity

Whether you use scripts, configuration management, or dedicated tooling, these principles apply:

1. Standardize Where Possible, Accommodate Where Necessary

You cannot eliminate every distribution from your fleet overnight. But you can stop the bleeding:

Pick a primary distribution for new deployments. Ubuntu LTS and Rocky Linux are the most common choices.

Containerize workloads that do not need a full OS. This consolidates your "patch surface" to fewer host OSes plus Alpine image rebuilds.

Set end-of-life dates for legacy systems. CentOS 7 should have a migration plan, not an indefinite exception.

2. Group Hosts by OS for Policy Assignment

Different distributions need different patching policies. Group your hosts explicitly:

```
# patch-groups.yaml
groups:
  ubuntu-prod:
  hosts: [web-01, web-02, web-03, api-01, api-02]
  os: ubuntu
  policy: security-weekly
  maintenance_window: "Sunday 02:00-06:00 UTC"

  rocky-prod:
  hosts: [db-01, db-02, batch-01]
  os: rocky
  policy: security-weekly
  maintenance_window: "Sunday 02:00-06:00 UTC"

  alpine-containers:
  clusters: [k8s-prod, k8s-staging]
  os: alpine
  policy: rebuild-weekly
  maintenance_window: "Saturday 22:00-02:00 UTC"

  legacy-centos:
  hosts: [billing-01, billing-02]
  os: centos7
  policy: manual-review
  note: "EOL - migration to Rocky 9 planned Q2 2026"

  freebsd-network:
  hosts: [fw-01]
  os: freebsd
  policy: monthly-manual
  maintenance_window: "First Saturday 03:00-05:00 UTC"
```

3. Maintenance Windows Per Group

Not all systems can be patched at the same time. Stagger your windows:

Staging first. Always. No exceptions.

Stateless application servers can be patched with rolling restarts during business hours if you have enough capacity.

Databases need coordinated windows with failover verification.

Network appliances (your FreeBSD firewall) get their own window because if the firewall goes down, nothing else matters.

4. Test Patches on Staging Before Production

This sounds obvious. In practice, it means maintaining staging environments that actually resemble production, including the OS distribution mix. If production runs Rocky Linux 9 and staging runs Ubuntu, your staging patching test tells you nothing about production.

5. Track and Report on Everything

At minimum, you need to answer these questions at any time:

> Which hosts have pending security updates right now?
> When was the last time each host was patched?
> Which CVEs are present in my environment?
> What changed during the last maintenance window?

If you cannot answer these questions, you do not have a patch management process. You have a patching habit.

Tooling Options for Multi-Distro Environments

There is no single tool that solves this perfectly, but here are the realistic options:

Ansible

Ansible's `package` module abstracts over multiple package managers. Combined with `ansible.builtin.apt`, `ansible.builtin.dnf`, and `ansible.builtin.apk`, you can write playbooks that handle different distributions:

```
- name: Apply security updates (Ubuntu/Debian)
  ansible.builtin.apt:
  upgrade: safe
  update_cache: yes
  when: ansible_os_family == "Debian"

- name: Apply security updates (RHEL family)
  ansible.builtin.dnf:
  name: "*"
  state: latest
  security: yes
  when: ansible_os_family == "RedHat"

- name: Apply all updates (Alpine)
  community.general.apk:
  upgrade: yes
  update_cache: yes
  when: ansible_os_family == "Alpine"
```

Pros: Flexible, widely understood, large community, good for orchestration. **Cons:** Ansible handles the execution but not the reporting, scheduling, or audit trail. You still need to build the monitoring and compliance layer yourself. Also, Ansible does not handle FreeBSD `freebsd-update` (base system updates) natively; you will need custom tasks for that.

Dedicated Patch Management Platforms

We are about to mention PatchMon. You are reading the PatchMon blog. We know you know. Let's all acknowledge the inherent bias and move on.

Tools specifically built for cross-platform patch management aim to solve the whole problem: detection, policy, execution, reporting, and compliance. PatchMon (https://patchmon.net) ships a single agent that understands APT, DNF, YUM, APK, Pacman, and FreeBSD pkg natively, with Windows Update monitoring today and Windows patch deployment on the roadmap. The agent handles the per-distribution logic locally and reports back in a normalized format, so you are not writing and maintaining the OS detection and output parsing yourself. You define patching policies by group and the platform handles scheduling, execution, and audit logging. It is available two ways: PatchMon Cloud (https://patchmon.net/pricing) as a managed SaaS (billed per-host, 14-day trial, no fleet minimum), or the AGPLv3 Community Edition self-hosted on infrastructure you manage.

Pros: Purpose-built for the problem, normalized cross-platform reporting, built-in audit trail. Cloud is per-host with no fleet minimum; Community has no PatchMon per-endpoint licence fee at all. **Cons:** Another tool to deploy and manage, agent-based (requires installation on each host). Newer project with a smaller community than Ansible or Foreman.

Spacewalk / Foreman / Katello

These are the traditional enterprise options, primarily aimed at RHEL-family distributions. Katello (built on Foreman and Pulp) gives you content management with errata and CVE tracking.

Pros: Deep integration with RHEL ecosystem, content management, errata-based patching. **Cons:** Heavy infrastructure requirements, RHEL-centric (limited support for other distributions), significant setup and maintenance burden.

The Hybrid Approach

In practice, most teams end up with a combination:

- Ansible or a patch management platform for host-level patching across Linux and FreeBSD
- Container image rebuild pipelines for Alpine (and other container base images)
- A reporting layer that aggregates patch status across all of the above

This is fine. The goal is coverage and visibility, not tool purity.

A Practical Workflow That Actually Works

Here is a concrete workflow for managing patches across a mixed fleet. Adapt it to your tools and scale.

Weekly Cycle

Monday: Scan and Assess

Run your update check across all hosts. With PatchMon, this is just opening your dashboard - every host reports its patch status automatically, across all distributions, with security updates flagged. If you are using Ansible instead, you need to gather facts manually:

```
# If using Ansible for assessment
ansible all -m package_facts -o
ansible all -m shell -a "cat /etc/os-release | grep PRETTY_NAME"
```

Review the results. Identify:

- Critical/high CVEs that need immediate attention
- Routine security updates for the weekly window
- Updates that require reboots (kernel, glibc, systemd, openssl)

Tuesday-Thursday: Stage and Test

Apply pending updates to staging environments:

```
# Staging Ubuntu hosts
ansible staging_ubuntu -m apt -a "upgrade=safe update_cache=yes" --become

# Staging Rocky hosts
ansible staging_rocky -m dnf -a "name=* state=latest security=yes" --become
```

Run your test suite against staging. Check application functionality, not just "did the server come back up." Pay specific attention to:

- TLS/SSL changes (OpenSSL updates)
- Database connectivity (library updates)
- Application startup (shared library changes)

Friday: Approve and Schedule

Based on staging results, approve the update batch for production. Document what is being applied and the expected impact.

Weekend: Execute Production Window

Apply updates during the maintenance window, grouped by OS and criticality:

```
# Phase 1: Stateless application servers (rolling)
ansible prod_web_ubuntu -m apt -a "upgrade=safe update_cache=yes" --become -f 2

# Phase 2: API servers
ansible prod_api_rocky -m dnf -a "name=* state=latest security=yes" --become -f 1

# Phase 3: Database servers (one at a time, verify replication)
ansible prod_db_rocky -m dnf -a "name=* state=latest security=yes" --become -f 1

# Phase 4: Rebuild Alpine container images
# (Trigger CI/CD pipeline that rebuilds with apk upgrade)

# Phase 5: FreeBSD firewall (manual, SSH session)
ssh fw-01 "pkg update && pkg upgrade -y && freebsd-update fetch install"
```

Monday: Verify and Report

Confirm all hosts are patched. Re-run your assessment scan and compare against the pre-patch baseline. Generate a report for your security team showing:

- Hosts patched
- CVEs remediated
- Any hosts that failed or were skipped
- Outstanding items for next cycle

Handling Emergencies

When a critical CVE drops (think Log4Shell, XZ Utils, or the next one), the weekly cycle compresses:

Identify exposure. Which hosts run the affected package? Each package manager can tell you:

```
# Ubuntu/Debian
dpkg -l | grep packagename

# RHEL/Rocky
rpm -qa | grep packagename

# Alpine
apk info | grep packagename

# Arch
pacman -Q | grep packagename

# FreeBSD
pkg info | grep packagename
```

Check patch availability. Is the fix in the repos yet?

Test on staging. Even under pressure, 30 minutes of staging testing is worth it.

Deploy to production. Prioritize internet-facing systems.

Document everything. The incident report needs timestamps.

Wrapping Up

Managing patches across mixed Linux distributions is not a problem you solve once. It is a practice you maintain. The discipline matters - know what you are running, know what needs patching, test before you deploy, and keep records - but the tooling matters too. The right tool turns a multi-hour weekly chore into a glance at a dashboard.

The uncomfortable truth is that Ansible, as good as it is at orchestration, was not built for this. You end up writing playbooks to gather data, more playbooks to apply patches, scripts to parse the output into reports, and cron jobs to run all of the above. Then the FreeBSD box breaks the playbook. Then someone asks for an audit trail. You are building a patch management platform out of duct tape, and maintaining it becomes a job in itself.

That is exactly why we built PatchMon. One agent per host, regardless of whether it is running APT, DNF, APK, Pacman, or FreeBSD pkg. One dashboard showing the unified view. Patch policies with dry-run support on all tiers and approval workflows on Plus and above. Compliance scanning on the Max tier. An audit trail that does not require grep-ing through Ansible logs. The fastest path in is PatchMon Cloud (https://patchmon.net/pricing), our managed SaaS. If you'd rather keep everything on your own stack, the AGPLv3 Community Edition (https://patchmon.net/open-source) is free.

If you have 10 servers on a single distro, Ansible or `unattended-upgrades` might genuinely be enough. But if your fleet is mixed, growing, or subject to compliance requirements, you will eventually need something purpose-built. We would rather you start with PatchMon now than spend six months building a reporting layer on top of Ansible only to replace it later.

Your fleet did not become heterogeneous on purpose, but you can manage it on purpose. Start a PatchMon Cloud trial (https://patchmon.net/pricing): 14 days on real hosts, card required, cancel before day 14 and no money changes hands.

PatchMon

ABOUT PATCHMON

The open source Linux patch management platform

PatchMon gives sysadmins one dashboard for patching across Linux, FreeBSD, and Windows fleets. Run it as a managed SaaS on PatchMon Cloud (per-host billing, 14-day trial, no fleet minimum) or self-host the AGPLv3 Community Edition on your own infrastructure.

Start a trial: patchmon.net/pricing