PatchMon PATCHMON BLOG

The Sysadmin's Guide to Patch Management Without Enterprise Bloat

You manage 50 servers, not 50,000. Here's how to build a patch management workflow that doesn't require a dedicated team, a Java application server, or a six-figure budget.

AUTHOR

PatchMon Team

PUBLISHED

18 March 2026

CATEGORIES

Tutorials, Security

READ ONLINE

https://patchmon.net/blog/patch-management-without-enterprise-bloat

Contents

1. What Enterprise Patch Management Looks Like (And Why It Exists)
    - Red Hat Satellite
    - SUSE Manager
    - ManageEngine Patch Manager Plus
    - Ivanti (formerly Shavlik, formerly LANDESK, formerly...)
6. What Small-to-Mid Teams Actually Need
7. The Progression of Approaches
    - Level 0: SSH and Memory
    - Level 1: Scripts + Cron + Email
    - Level 2: Ansible / Puppet / Salt Playbooks
    - Level 3: Dedicated Lightweight Tools
12. What to Look for in a Lightweight Patch Management Tool
13. Building a Workflow That Fits a Small Team
    - 1. Establish Continuous Visibility
    - 2. Classify Your Servers
    - 3. Define "Good Enough" SLAs
    - 4. Automate the Boring Parts
    - 5. Keep Records Without Drowning in Process
    - 6. Handle the Exceptions
20. The Real Goal

You have a problem. Not a complicated problem, either. You manage somewhere between 10 and 500 Linux servers, and you need to keep them patched. Security updates need to go out. You need to know what is running where. When the auditor asks "how do you manage patching?", you need a better answer than "I SSH into them."

So you google "patch management tool."

And every single result on the first page wants you to deploy a dedicated server with 20GB of RAM, configure a PostgreSQL cluster, set up content synchronization, purchase a per-node subscription, and block out a three-day maintenance window just to get the tool itself running.

You don't need a forklift to move a box of books. But apparently nobody told the enterprise software vendors.

What Enterprise Patch Management Looks Like (And Why It Exists)

Let's be fair before we get frustrated. Enterprise patch management tools exist because enterprise problems are real. When you manage 10,000 nodes across multiple data centers with change advisory boards and regulatory frameworks that require six months of audit logs in a specific format, you need something built for that scale. The tools below are genuinely good at what they do. The problem is not that they exist. The problem is that they are the only things that show up when a sysadmin with 40 servers goes looking for help.

Red Hat Satellite

Satellite is the gold standard for RHEL fleet management. It handles content management, provisioning, configuration, and patching across thousands of hosts. It is also a beast to deploy and operate.

Under the hood, Satellite runs Foreman, Katello, Pulp (content management), Candlepin (subscription management), and PostgreSQL. Red Hat's own documentation recommends a minimum of 20GB RAM and 4 CPU cores for the Satellite server alone, and that is before you add Capsule servers for each network segment. You need a valid Red Hat subscription for every managed host. The initial sync of RHEL content repositories can take hours and consume hundreds of gigabytes of disk.

If you are an all-RHEL shop with 2,000+ nodes and a dedicated platform team, Satellite is excellent. If you manage 80 mixed-distro servers and you are also the person who fixes the printer, Satellite is going to consume your entire quarter just getting it stood up.

SUSE Manager

SUSE Manager is architecturally similar to Satellite (it is also built on open-source components, historically Spacewalk, now Uyuni). It handles SUSE, RHEL, and Ubuntu to varying degrees. The deployment story is comparable: dedicated server, significant resource requirements, content synchronization, and an assumption that you have a team to operate it.

It is a solid tool. It is also clearly designed for organizations with "Infrastructure Team" in the org chart, not "the sysadmin."

ManageEngine Patch Manager Plus

ManageEngine takes a different approach: it is a Windows-based application backed by SQL Server (or the bundled PostgreSQL). Licensing runs anywhere from $10,000 to $25,000 per year depending on node count and features. The product is capable, but it brings a Windows-centric worldview to what is often a Linux problem. The agent-based architecture is reasonable, but the management server requiring Windows is a hard sell for teams that do not run Windows infrastructure at all.

It also tries to be everything: patch management, software deployment, a compliance dashboard, and half of an endpoint management platform. You came here to patch servers. You do not need a software deployment pipeline for Microsoft Office.

Ivanti (formerly Shavlik, formerly LANDESK, formerly...)

Ivanti is the enterprise endpoint management platform that has absorbed roughly half the patch management companies that have ever existed. It is powerful. It is also the kind of tool where "deployment" is a professional services engagement measured in months and billed accordingly. Annual licensing for a mid-size deployment can easily exceed $50,000. The platform does everything from patch management to IT asset management to service desk workflows.

If you need all of that, great. If you need to patch 50 servers and move on with your day, this is like hiring a general contractor to hang a picture frame.

What Small-to-Mid Teams Actually Need

Here is the thing: the actual requirements for patching a small-to-mid fleet are not complicated. You need:

> **An inventory of what is installed across all your hosts.** Which servers are running what OS, what kernel, what packages. This should be automatic and current, not a spreadsheet you update when you remember.

Visibility into what needs updating. Especially security updates. A dashboard, a report, an API endpoint: something that answers "what is vulnerable right now?" without requiring you to SSH into 50 machines.

Dry-run capability. Before you run `apt upgrade` or `dnf update` on production, you want to know what is going to change. Which packages, which versions, are there any held-back packages or dependency conflicts.

Approval workflows for production systems. Maybe your dev servers auto-patch on Tuesday night. But production needs someone to review the change list and click "approve" before anything happens.

An audit trail. When did each server last patch? What changed? Who approved it? This is not optional if you have any compliance requirements, and it is incredibly useful even if you do not.

Alerting when something goes wrong. A patch fails, a reboot is needed, a service did not come back up; you need to know, ideally before your users tell you.

What you do NOT need:

A service desk

A CMDB

An ITSM workflow engine

A "digital transformation platform"

A Java application server

A content delivery network for RPM repositories

A vendor who wants to schedule a "discovery call" before telling you the price

The gap between what small teams need and what enterprise vendors sell is enormous. And it is not getting smaller.

The Progression of Approaches

Most sysadmins do not wake up one morning and decide to evaluate patch management platforms. They evolve through a series of increasingly sophisticated (and increasingly broken) approaches until the pain gets bad enough to look for a real tool.

Level 0: SSH and Memory

```
ssh server12
sudo apt update && sudo apt upgrade -y
# Done. Probably. I think I did server11 yesterday.
# Or was that last week?
```

This is where everyone starts. It works when you have 5 servers. It works less well when you have 15. It completely falls apart when you have 50, because you cannot remember which servers you have already patched, whether you rebooted the ones that needed it, or whether that one server in the corner is still running a kernel from 2024.

The failure mode here is not dramatic. Nothing explodes. You just slowly accumulate unpatched servers until one of them gets compromised, or an auditor asks for your patching records and you realize you do not have any.

Level 1: Scripts + Cron + Email

```
#!/bin/bash
# patch-check.sh: runs weekly via cron
# Totally reliable. Has never broken. (It broke 3 months ago.)

SERVERS="server01 server02 server03 server04"

for server in $SERVERS; do
  echo "=== $server ===" >> /tmp/patch-report.txt
  ssh "$server" "apt list --upgradable 2>/dev/null" >> /tmp/patch-report.txt
done

mail -s "Weekly Patch Report" sysadmin@company.com < /tmp/patch-report.txt
```

This is the "I automated it" phase. You write a script that SSHes into each server, checks for available updates, and emails you a report. Maybe you have a second script that actually applies patches. You feel good about this for roughly six weeks.

Then the script silently breaks because someone changed an SSH key, or the mail relay stopped working, or you added 10 new servers and forgot to update the server list. The script has been failing for three months and nobody noticed because the absence of an email is indistinguishable from "everything is up to date."

There is also no audit trail beyond your email archive, no approval workflow, no rollback capability, and no way to answer "which servers are currently unpatched?" without running the script again and waiting.

Level 2: Ansible / Puppet / Salt Playbooks

```
# ansible playbook: patch-servers.yml
---
- hosts: all_servers
  become: yes
  tasks:
 - name: Update all packages
  apt:
    upgrade: dist
    update_cache: yes
  when: ansible_os_family == "Debian"

 - name: Update all packages (RHEL)
  dnf:
    name: "*"
    state: latest
  when: ansible_os_family == "RedHat"

 - name: Check if reboot is required
  stat:
    path: /var/run/reboot-required
  register: reboot_required

 - name: Reboot if required
  reboot:
    msg: "Reboot for kernel update"
  when: reboot_required.stat.exists
```

Now we are getting somewhere. Configuration management tools are a genuine step up. Your server inventory is defined in code. Playbooks are version-controlled. Execution is logged. You can target groups of servers, run dry checks with `--check` mode, and build increasingly sophisticated workflows.

This is a perfectly valid approach, and many teams stop here permanently. If it works for you, there is no shame in it.

But there are real gaps:

No persistent dashboard. You know the state of your fleet at the moment you run the playbook. Five minutes later, you have no idea unless you run it again.

No continuous monitoring. If a new CVE drops at 2 AM, nothing happens until you manually trigger a check.

Audit trails require extra work. You can log Ansible output, but turning that into a compliance report means parsing text files or building something custom on top.

No approval workflows. You can gate playbook execution behind a CI/CD pipeline with manual approvals, but at that point you are building a patch management platform out of duct tape and GitHub Actions.

> **Alerting is DIY.** You need to bolt on monitoring for "did the patch run succeed? Did all services come back up?"

Configuration management is a building block, not a complete solution. It handles the "do the thing" part well. It does not handle the "know the current state," "approve before doing," or "prove we did it" parts without significant custom work.

Level 3: Dedicated Lightweight Tools

This is the category that barely existed five years ago and is slowly filling in. These are tools purpose-built for patch management that do not try to be an entire IT operations platform.

Foreman + Katello is the open-source upstream of Red Hat Satellite. You get much of Satellite's functionality without the subscription requirement, and it supports multiple Linux distributions. The trade-off is that you are deploying and maintaining the same complex stack (Pulp, Candlepin, PostgreSQL, the works) without Red Hat's support. It is lighter than Satellite in licensing cost, not in operational complexity. If you have the expertise and the time, it is a strong choice. If you are a one-person team, the initial deployment will test your patience.

Canonical Landscape manages Ubuntu systems well. If you are an all-Ubuntu shop, it is worth evaluating. It is tightly coupled to the Ubuntu ecosystem, which is either a feature or a limitation depending on your fleet composition.

PatchMon (yes, this is the part where we recommend our own product). We are self-aware enough to know how this looks. In the interest of honesty: you can run it two ways. PatchMon Cloud (https://patchmon.net/pricing) is the managed SaaS, per-host pricing with a 14-day trial on real hosts and no fleet minimum. The Community Edition (https://patchmon.net/open-source) is open source (AGPLv3), self-hosted on infrastructure you manage, and you can evaluate it in about five minutes with `docker compose up`. Either way, if it does not solve your problem, you have lost almost nothing. PatchMon takes a different approach to the deployment problem. The server is a single binary (or a Docker Compose stack: Postgres, Redis, and the application). The agent is a single binary you deploy to each managed host. There is no content synchronization, no Java application server, no multi-day installation process. You get a dashboard showing patch status across your fleet, security update detection, dry-run support, approval workflows (Plus tier and above), compliance scanning with OpenSCAP, CIS, and Docker Bench (Max tier), and an audit trail. It supports mixed environments: Debian, Ubuntu, RHEL, Rocky, Alma, Alpine, Arch, FreeBSD, and Windows. The trade-off is that it is a newer tool with a smaller ecosystem than Foreman, and it does not try to handle provisioning or full lifecycle management. It does patch management and does it without the overhead.

Mondoo and **Vulners** approach the problem from the vulnerability scanning angle rather than the patch deployment angle. They are strong on visibility ("here is what is vulnerable") but lighter on the workflow side ("now approve and deploy the fix").

The right choice depends on your specific environment, your team size, your compliance requirements, and honestly, how much time you want to spend setting up the tool versus using it.

What to Look for in a Lightweight Patch Management Tool

If you are evaluating tools in this space, here is a practical checklist. Not every tool will tick every box, but these are the things that matter for small-to-mid teams:

Deployment complexity. How long does it take to go from "I downloaded this" to "I can see my servers on a dashboard"? If the answer is measured in days, the tool was not built for your team size. If the answer is measured in minutes to hours, you are in the right category.

Resource requirements. Does the management server need 20GB of RAM and 200GB of disk for content sync? Or can it run on a 2GB VPS alongside your other tools? Small teams do not have spare hardware sitting around for a patch management server.

Multi-distribution support. Unless your fleet is 100% one distribution (and it probably is not), you need a tool that handles Debian and RHEL families at minimum. Bonus points for FreeBSD, SUSE, and other platforms.

Agent vs. agentless. Agent-based tools (a small daemon on each managed host) generally provide better real-time visibility and do not require SSH access from a central server. Agentless tools (SSH-based) are simpler to start with but scale less gracefully and require maintaining SSH credentials centrally. Either model works. Understand the trade-offs.

Visibility without action. Can you see the current state of your fleet at any time, without running a scan or playbook? A persistent, automatically-updated dashboard is a meaningful step up from "run the check and wait."

Dry-run support. Can you preview what a patch run will do before committing? This is non-negotiable for production systems.

Approval workflows. Even if your "workflow" is just "I look at the list and click approve," having that step recorded matters for compliance and for your own sanity at 2 AM when you are trying to remember whether you approved that kernel update.

Audit trail. Every patch action, every approval, every failure: logged, timestamped, and searchable. This is the thing you do not think you need until an auditor asks for it or until you are troubleshooting a post-patch incident.

Alerting and integration. Can the tool notify you when something needs attention? Email is fine. Slack or webhook integrations are better. If it only shows you information when you log into the dashboard, you will miss things.

Maintenance burden. How much ongoing effort does the tool itself require? Does it need regular content syncs? Database maintenance? Version upgrades that require downtime? The tool is supposed to reduce your workload, not add to it.

Cost. Open source is great. Reasonable per-node pricing is fine. If the vendor wants a "discovery call" before showing you pricing, the tool was not designed for a team of your size.

Building a Workflow That Fits a Small Team

Forget the enterprise ITIL framework for a moment. Here is a practical patch management workflow that a one-to-three person team can actually sustain:

1. Establish Continuous Visibility

First priority: know the current state of everything, all the time. Deploy your chosen tool's agent (or configure agentless scanning) across your entire fleet. The goal is a single dashboard where you can answer:

> How many servers have outstanding security updates?
>
> Which servers have not been patched in the last 30 days?
>
> Are any servers running end-of-life OS versions?

With PatchMon, this looks like deploying the agent binary to each host and pointing it at your server instance - the dashboard populates automatically within minutes. With Foreman, this means registering hosts and configuring content views (budget a day). With Ansible, this means writing a fact-gathering playbook, running it on a schedule, and building something to display the results (budget a weekend, then ongoing maintenance).

If you cannot answer "what is the current patch state of my fleet?" at any given moment, nothing else works. PatchMon gives you that answer on day one.

2. Classify Your Servers

Not all servers are equal. At minimum, define two tiers:

> **Auto-patch:** Dev servers, staging environments, internal tools, anything where a brief disruption is acceptable. These get patches automatically on a schedule.
>
> **Approval-required:** Production servers, customer-facing services, databases, anything where an unexpected reboot or package change could cause an outage. These get patches only after review and approval.

In practice this often looks like:

```
Monday night:  Auto-patch dev/staging servers
Tuesday morning: Review results, check for issues
Wednesday night: Apply approved patches to production (batch 1)
Thursday morning: Verify production batch 1
Thursday night:  Apply approved patches to production (batch 2)
Friday:    Never patch on Friday. Ever.
```

Adjust the schedule to your environment, but the principle holds: let your non-critical servers be the canary. If an update breaks something, you find out on dev before it hits production.

3. Define "Good Enough" SLAs

You are not going to patch every CVE within 24 hours. That is fine. Define realistic targets:

Critical/High severity security updates: Applied within 7 days

Medium severity security updates: Applied within 30 days

Non-security updates: Applied within 60 days or at next maintenance window

Emergency (actively exploited zero-days): Applied within 24-48 hours via expedited process

Write these down. Share them with your team (even if your team is just you). Having defined targets turns patching from "I should probably do that" into "this has a deadline."

4. Automate the Boring Parts

The parts of patching that should be automated:

Checking for updates. This should happen daily, automatically. Your tool should be scanning every host and updating your dashboard without you lifting a finger.

Applying patches to auto-patch tier. Cron job, scheduled task, agent-driven, however your tool handles it. Dev servers should not wait for you to remember.

Notifications. New critical security updates, failed patch runs, servers that need reboots: these should come to you via Slack, email, or whatever your team monitors.

The parts that should NOT be automated (at least initially):

Production patching. Until you have high confidence in your process, keep a human in the loop for production changes.

Kernel updates that require reboots. These need coordination, especially for services without redundancy.

Major version upgrades. Going from PostgreSQL 16 to 17 is not a "patch." Do not let automation anywhere near it.

5. Keep Records Without Drowning in Process

Compliance does not have to mean "a 47-page change request for every patch Tuesday." At the lightweight end:

- Your tool's audit log covers what was patched, when, and by whom
- A brief entry in your team's changelog or Slack channel: "Applied March security updates to production cluster, all services verified"
- Screenshot or export of your dashboard showing current patch status

That is enough for most compliance frameworks at the small-to-mid scale. If your auditor wants more, they will tell you, and you can add process incrementally.

6. Handle the Exceptions

Every environment has them:

- The server running a legacy application that breaks if you update OpenSSL
- The vendor appliance that you are contractually prohibited from patching yourself
- The database server where you need to coordinate patching with application deployment windows

Document these exceptions explicitly. "Server X is excluded from automatic patching because [reason]. Compensating control: [what you do instead, e.g., network isolation, manual review quarterly]." Having this written down protects you when someone asks why that server is 90 days behind on patches.

The Real Goal

The point of all of this is not to achieve some theoretical state of perfect patch compliance. The point is to get from "I have no idea what is running on half my servers" to "I have continuous visibility, a reasonable process, and an audit trail" - without spending three months deploying an enterprise platform that was designed for a team ten times your size.

That is exactly the gap PatchMon was built to fill. You do not need a dedicated team to run it, a Java application server, an Oracle database, or a six-figure budget. The default path is PatchMon Cloud (https://patchmon.net/pricing), our managed SaaS: per-host pricing, 14-day trial on real hosts (card required, cancel before day 14 and you are not charged), no fleet minimum, and we run the control plane. If you'd rather self-host, the AGPLv3 Community Edition (https://patchmon.net/open-source) is free, runs on Docker Compose, and keeps all your data in your own Postgres.

If you are currently at Level 0 or Level 1, pick one and go. The Cloud trial takes less time than your next "let me SSH into all the servers and check" session, and you pay only for the hosts that actually check in. If the self-hosted route fits your stack better, the binaries and compose

files are waiting on GitHub.

The enterprise vendors will still be there if you ever grow into needing them. In the meantime, your 50 servers need patches and you have actual work to do.

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