SCRIPT

Bash Script to Automate Ubuntu Security Patch Audits

This guide shows how to build a Bash script that audits Ubuntu security patches by checking installed packages against available security updates, with safe defaults, validation, and exportable results.

Operating Systems - July 04, 2026

Introduction

Security patch audits often fail operationally because the review process is manual, inconsistent, or too slow to support change windows and compliance evidence. On Ubuntu systems, that means teams may know updates exist, but not whether the machine is missing security fixes, which packages are affected, or how to capture results in a repeatable way.

In this guide, you will build a practical Bash script to automate an Ubuntu security patch audit. The script will check whether security updates are available, list the affected packages, support a safe dry-run mode by default, and produce machine-readable output you can use for reporting or follow-up. You will also see what the script does not do, what must be verified before production use, and how to validate the results without changing the system.

Script language and scope

This solution uses Bash and standard Ubuntu package-management tools. It is intentionally narrow in scope: it audits the local system for pending security updates and reports what is missing. It does not install packages, restart services, patch remotely, or open tickets.

For broader system hygiene, pair this audit with your normal access-control and firewall review process. If you also need to validate host exposure before patching, a configuration review such as How to Configure UFW Firewall Rules on Ubuntu Server can help confirm whether the machine is restricted appropriately before maintenance work begins.

What the script does and does not do

Does

- Checks whether the local Ubuntu host has security updates available.
- Filters package results to security-relevant updates when repository metadata supports that distinction.
- Runs in safe, read-only mode by default.
- Validates input parameters before doing any work.
- Emits human-readable output and optional JSON output for automation.
- Exits with clear status codes so schedulers and pipelines can interpret the result.

Does not

- Install or upgrade packages.
- Reboot the machine.
- Verify kernel live patching status.
- Aggregate results from multiple hosts.
- Guarantee that every security fix is tagged perfectly by every repository or mirror configuration.

That last point matters: the script depends on what package metadata is available on the host. You should verify repository health, update freshness, and your Ubuntu release support status before trusting the audit as a compliance source.

Assumptions, requirements, and permissions

Assumptions

- The host is Ubuntu and uses apt/apt-get.
- Security repository metadata is available through the configured package sources.
- The machine can reach its package mirrors or an internal mirror.
- You want local audit output, not fleet orchestration.

Requirements

- Bash 4+.
- apt-get, apt-cache, grep, awk, sed, sort, mktemp, and flock if you want to prevent concurrent runs.
- python3 if you want JSON formatting in the example below. The script can still operate without JSON, but the example includes it because machine-readable output is useful in automation.
- Read access to package metadata, which is normally available to non-root users for audit-only checks, though some environments restrict package cache access.

Permissions

- Read-only audit mode: usually works as a standard user if local package metadata is readable.
- Full package cache refresh: may require sudo if you choose to update package lists before the audit.

- Do not run an audit script that changes package state unless you explicitly intend to patch.

If you are using this as a compliance or security evidence workflow, document which account ran the audit, the repository state at the time, and whether caches were refreshed immediately beforehand.

Bash script

The script below is a safe skeleton suitable for production hardening. It defaults to audit-only behavior and uses explicit parameters for anything that could alter system state.

How the script works

The script follows a simple operational flow:

- Parse and validate command-line arguments.
- Check that required commands exist.
- Optionally refresh package lists in a controlled way.
- Query pending upgrades from the local apt cache.
- Filter the output to security-relevant entries when requested.
- Return either text or JSON.
- Exit with a status code that tells automation whether updates are pending.

That structure is intentional. In security audit work, the biggest practical failure is not the query itself but the inability to tell whether the script completed cleanly, found a problem, or silently failed because a dependency was missing.

Expected input and output

Input

The script accepts only command-line flags. It does not read files or require an inventory feed.

Typical inputs:

- No arguments: audit-only, text output.
- --refresh-cache: refresh package metadata first.
- --format json: return JSON for pipelines or log collectors.
- --all-updates: report all pending updates instead of only security-related ones.

Example command

Example output

Text mode output might look like this:

The exact package names, source labels, and formatting will vary by Ubuntu release and repository configuration.

What to change before production use

The script is deliberately conservative, but you should still harden it for your environment before using it as an audit control.

Verify the security filter

The sample filter uses repository text markers such as security, Ubuntu-Security, or security.ubuntu.com. That is a practical starting point, but you must verify that your mirrors and release naming conventions expose security updates in a way the filter can detect. If your environment uses internal mirrors or apt proxies, the source string may differ.

Decide whether to refresh caches

Refreshing the apt cache makes results more current, but it also depends on network reachability and may require elevated privileges. If your audit process runs from cron or a configuration management job, decide whether the cache refresh belongs in the script or in a separate, controlled precheck.

Align exit codes with automation

The example returns 1 when updates are found. That is useful for schedulers, but some monitoring systems interpret any non-zero exit as a hard failure. If that is your case, keep the status code logic and map it to an alerting rule that understands ?updates available? versus ?script error.?

Replace local-only assumptions if needed

If you need fleet reporting, do not bolt a server inventory layer into this script unless you are ready to handle authentication, transport security, and result normalization. Keep the script local and let your orchestration platform collect the output.

Safe execution behavior and dry-run defaults

This script is read-only by default. That is the right default for security audits because the purpose is evidence, not remediation.

Safe behavior in this script includes:

- No package installation.
- No service restarts.
- No direct changes to system configuration.

- No outbound API calls.
- No ticket creation.
- Optional cache refresh only when explicitly requested.

If you later extend this script to trigger notifications, write to an external system, or open a case, keep the safe default pattern: dry-run first, explicit opt-in for active actions, and strong input validation around every external dependency.

Validation and testing steps

Before production use, validate the script in at least three stages.

1. Syntax check

This confirms there are no parsing errors.

2. ShellCheck review

ShellCheck will flag quoting issues, unreachable branches, and common Bash mistakes.

3. Controlled execution

Run it on a non-production Ubuntu system that matches the target release as closely as possible.

Check for these outcomes:

- The script exits 0 when no updates are pending.
- The script exits 1 when pending security updates are found.
- The script exits 2 for missing commands or invalid arguments.
- The text and JSON formats both parse cleanly.

- The output matches your repository setup and release expectations.

If you are using the audit as part of a broader host-hardening workflow, document the evidence in the same way you would for firewall controls or patch baselines. For example, a security review may also reference host exposure controls like those described in How to Configure UFW Firewall Rules on Ubuntu Server, but keep the patch audit focused on update status only.

Security and privacy notes

Patch audit output can reveal software versions, installed packages, and repository naming details. That is operationally useful, but it is still sensitive in many environments.

Keep the following in mind:

- Restrict access to audit logs and exported JSON.
- Avoid sending results to unsecured channels.
- Treat package version data as inventory information.
- Do not expose internal mirror hostnames unless necessary.
- If you wrap the script in a centralized scheduler, ensure credentials and repository tokens are not echoed to logs.

If your audit process uses sudo -n apt-get update, confirm that passwordless privilege is limited to the intended command and account. Broad sudo rights undermine the control you are trying to prove.

Cleanup and rollback guidance

Because the script is read-only, rollback is usually simple: remove the file and any scheduler entry you created.

Cleanup

- Delete the script from the host if it is no longer needed.

- Remove any cron job, systemd timer, or CI schedule that invokes it.
- Archive or rotate audit logs according to your retention policy.

Rollback

If a production validation reveals that the script?s security filter misclassifies updates, rollback means:

- Disable the job.
- Revert to manual auditing or a known-good version of the script.
- Adjust the repository filter logic.
- Re-run the audit in a controlled environment before re-enabling automation.

Because the script does not modify packages, there is no package-state rollback required. That is another reason to keep the audit and the remediation workflows separate.

Common mistakes

Mistake	Why it matters	Better approach
Treating every apt warning as a security finding	Non-security noise can create false positives and desensitize responders	Filter the
Running without validating dependencies	Missing tools can produce partial output or a silent failure path	Check commands up front w
Refreshing package lists implicitly	Hidden network dependency makes runs inconsistent and harder to troubleshoot	Make cache refre
Using non-zero exit codes without documentation	Monitoring tools may treat ?updates available? as a job failure	Document the exit-c
Assuming JSON output is automatically valid	Malformed output breaks log pipelines and downstream parsing	Generate JSON throug
Extending the script to patch systems directly	Audit and remediation have different change-control and rollback requirements	Keep th

Final takeaway

A Bash-based Ubuntu security patch audit works well when it stays narrowly focused: verify the local package state, report pending security updates clearly, keep dry-run behavior by default, and make every dependency and exit code explicit before you trust it in production.

Use this guidance together with Windows 11 hardening checklist to connect the workflow with related operational context already available on the site.