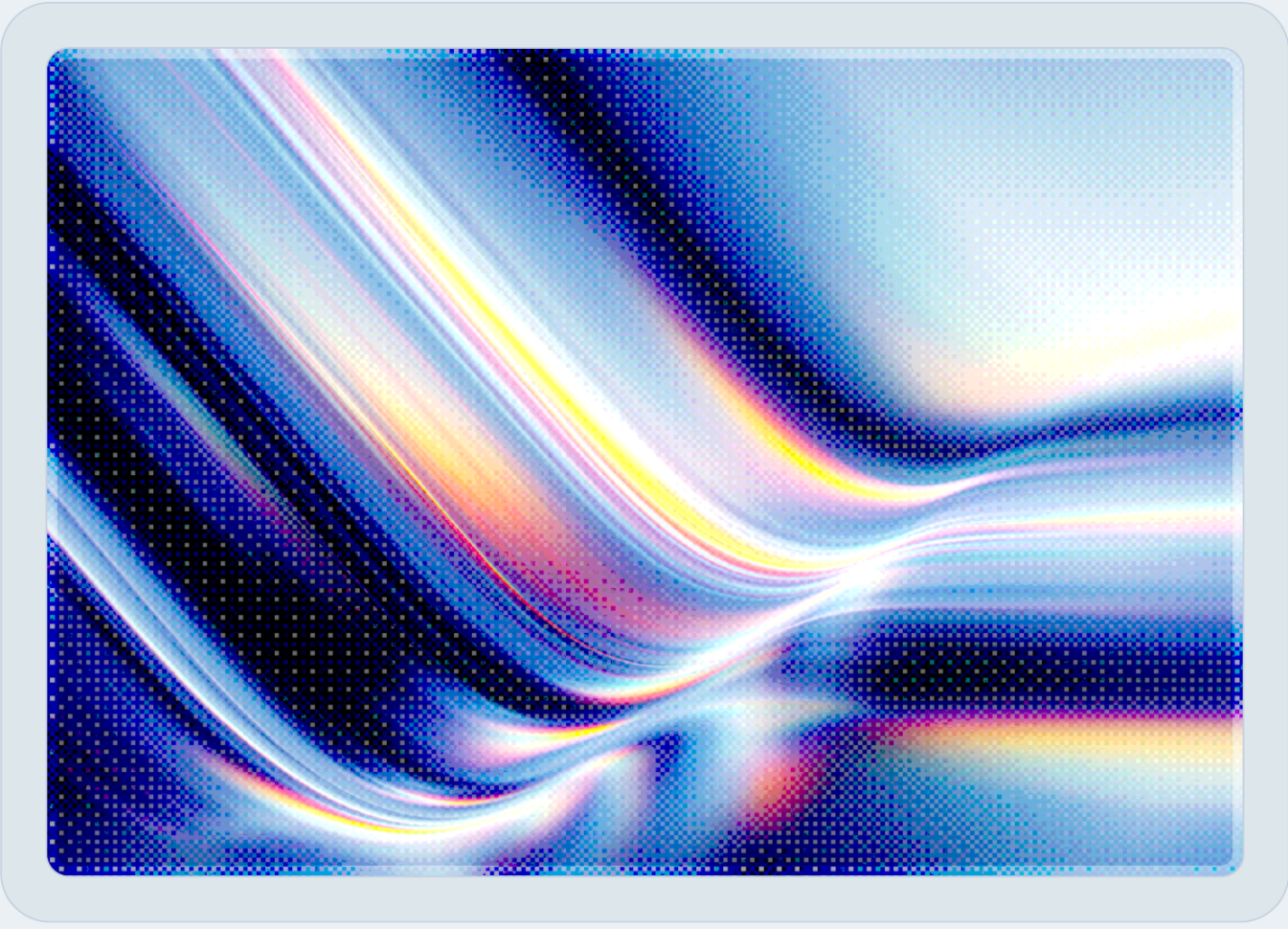
CSAI Foundation | Cloud Security Alliance

Atomic Arch: eBPF Rootkit via AUR Supply Chain

Developer Credential Theft Across 1,500+ Compromised Packages

2026-06-15

AI-assisted Rapid Research



CSAI

CSA cloud security alliance®

Key Takeaways

- Beginning June 11, 2026, threat actors systematically adopted abandoned Arch User Repository (AUR) packages and modified their build scripts to deliver a Rust-compiled credential stealer paired with an optional eBPF kernel-level rootkit, a campaign named Atomic Arch by Sonatype [1][2].
- The first wave compromised approximately 408 packages [4]; a second wave within 24 hours —apparently in response to early takedown activity—substituted a different package (`js-digest`) delivered via the `bun` runtime, expanding the total to more than 1,500 packages carrying a CVSS score of 8.7 (tracked as Sonatype-2026-003775) [2][3].
- No vulnerability was exploited. The attackers inherited legitimacy by claiming ownership of orphaned packages through AUR's standard adoption workflow—requiring no privilege escalation, no code exploit, and no social engineering of the original maintainer [1][4].
- The credential stealer targets a notably comprehensive set of developer secrets, extending beyond consumer-credential targets to include source-control tokens, cloud provider keys, and secrets-management credentials: GitHub tokens, SSH keys, cloud provider credentials (AWS, GCP, Azure), container registry credentials, OpenAI API keys, browser session cookies, and Electron application data from Slack, Discord, and Microsoft Teams [1][2].
- Systems where the payload executed with root privileges must be treated as fully compromised and reinstalled from trusted media; within the running compromised OS, the eBPF rootkit conceals its own presence from the standard tools used to detect it [1][2].
- This incident highlights the need to extend Software Bill of Materials (SBOM) practices and CI/CD supply chain controls to cover community-sourced build environments and developer workstations, not only production application dependencies.

Background

The Arch User Repository and Its Trust Model

The Arch User Repository is a community-maintained collection of build scripts for Arch Linux, enabling users to compile and install software not available in Arch's official repositories. Each package is defined by a PKGBUILD—a shell script that specifies how to retrieve, build, and install a piece of software—along with optional `.install` hooks that execute during package installation. Unlike official Arch packages, AUR submissions receive no cryptographic signing from Arch maintainers and no automated code review. The model has historically relied on the vigilance of individual maintainers and the scrutiny of experienced users reviewing build scripts before installation.

A critical feature of this governance model is the orphaned-package adoption pathway. When a maintainer abandons a package, it is marked orphaned and any registered AUR account may claim stewardship. The adoption process is lightweight by design: a request is submitted, and once granted, the new maintainer receives immediate write access to the package's build scripts. There is no vetting of the new maintainer's identity, no delay period, no cryptographic continuity requirement linking the new identity to the prior one, and no notification surfaced to existing users that the package changed hands [4][5]. The design existed for years as a convenience mechanism; Atomic Arch demonstrated that it also constitutes an exploitable trust inheritance surface.

The Atomic Arch Campaign

On June 11, 2026, Sonatype researcher Eyad Hasan identified the first indicators of a coordinated supply chain campaign [2]. Threat actors had filed ownership requests for dozens of orphaned AUR packages and, once granted access, modified PKGBUILD files to embed post-install hook calls. The injected hook invoked `npm install atomic-lockfile minimist chalk` during the standard package build phase, causing any user running the affected AUR package to silently retrieve a malicious npm package alongside two legitimate-looking dependencies included for cover [1][2].

By the end of the day, community trackers had cataloged approximately 408 compromised packages in this first wave [4]. A second wave emerged on June 12, substituting a package called `js-digest` delivered via the `bun` runtime—an apparent operational pivot in response to early takedown activity. Across both waves, the total count of compromised packages exceeded 1,500 [3][4]. Arch Linux maintainers removed malicious commits and banned associated accounts, confirming that the attack

was limited to the AUR and that no official Arch repositories were breached. The malicious npm package `atomic-lockfile` was also reported to the npm registry for removal. Community response was coordinated in part through a mailing-list thread initiated by Arch maintainer Jonathan Grotelüschen [1].

Security Analysis

Attack Architecture and Payload Chain

The Atomic Arch attack chain is notable for its simplicity relative to its impact. The entire infection pathway involves no code vulnerability, no zero-day, and no privilege escalation to establish initial access. It requires only that a user install an affected AUR package while the malicious PKGBUILD is active—a routine operation for Arch users maintaining their systems.

The payload executes in three stages. First, the modified PKGBUILD runs an npm or bun command as a post-install hook, fetching the malicious package from the public npm registry. Second, the npm package's preinstall script executes a bundled Linux ELF binary named `deps`, compiled in Rust. Third, `deps` runs in the background, initiating credential harvesting and, if root privileges are available, loading the eBPF rootkit [1][2]. The decision to stage the payload through the npm registry rather than embedding it directly in the PKGBUILD reduced the malicious footprint visible in AUR's commit history, making detection harder for users performing a quick build-script review.

This staging pattern echoes prior supply chain incidents in the npm ecosystem. The 2021 ua-parser-js compromise involved a popular npm package being briefly hijacked to deliver a cryptocurrency miner and credential stealer to downstream consumers [11]. The 2022 node-ipc incident similarly saw a widely-installed package modified to deliver a destructive payload to systems matching specific geographies [12]. Atomic Arch extends this technique across package ecosystem boundaries, using an AUR package as the trust anchor to pull from npm.

The Credential Stealer

The infostealer component of `deps` targets the full breadth of secrets typically present on a developer workstation. Browser credential databases—including Chromium, Chrome, Edge, and Brave—are harvested alongside session cookies that could grant authenticated access to web applications without requiring password reuse. Electron application data stores from Slack, Discord, Microsoft Teams, and Telegram are targeted for session tokens that grant access to organizational communication channels and any secrets shared through them [1][2].

Developer-specific targets reflect deliberate focus on organizational access rather than individual consumer credentials. GitHub personal access tokens and deploy keys grant write access to source repositories. npm authentication tokens allow publishing packages under a compromised identity—potentially enabling the attacker to extend the supply chain compromise further downstream. AWS, GCP, and Azure access keys provide potential entry into cloud infrastructure. HashiCorp Vault tokens may expose secrets management systems that distribute credentials to production workloads. Docker Hub and container registry credentials allow pushing modified container images. OpenAI API keys grant access to AI model APIs whose billing and usage can be exploited at the victim's expense. SSH private keys and shell command histories round out the harvest, with histories potentially exposing organizational topology, internal hostnames, and undocumented credentials passed on the command line [1][2].

Exfiltration routes data through a Tor onion service via a local loopback proxy, and uses the legitimate file-sharing service temp.sh for bulk data transfer [4]. The dual-channel design is operationally significant: security controls monitoring for Tor traffic may alert while missing the temp.sh exfiltration path, and controls focused only on known-malicious domains will miss both channels since temp.sh is a legitimate service.

The eBPF Rootkit

When `deps` executes with root privileges, it conditionally loads a kernel-level stealth component that exploits Linux's eBPF subsystem. The rootkit hooks the `getdents64()` system call, which is the kernel function invoked by directory listing operations [1][2]. By intercepting this call, the rootkit can silently filter its own artifacts—files and processes listed under `/proc/`—from directory-listing results. Standard user-space utilities including `ls`, `ps`, and `find`, as well as endpoint detection tools that rely on user-space system call results, will not display the malware's presence. eBPF-based detection tools such as Tetragon and Falco may retain independent visibility, but only if deployed before the rootkit loads. Three BPF maps are pinned into `/sys/fs/bpf/` under the names `hidden_pids`, `hidden_names`, and `hidden_inodes`, populated with the identifiers and names of the malware's own artifacts [1][2].

Persistence is established through systemd. When run as root, the malware copies itself to `/var/lib/` and registers a service unit in `/etc/systemd/system/` with `Restart=always`, ensuring survival across reboots. A non-root execution path also exists, deploying to the user's home directory with a per-user unit at `~/.config/systemd/user/` [1]. Because the eBPF rootkit conceals these systemd units and their associated files from standard inspection tools, in-kernel verification of cleanup is not possible once the rootkit has loaded. Within the compromised OS environment, the kernel's view of the filesystem cannot be trusted; only booting from external trusted

media restores an unmanipulated view of the system state. Security researchers and Arch maintainers uniformly advise full reinstallation for root-compromise scenarios precisely because in-place remediation cannot be reliably confirmed [1][2].

The eBPF subsystem presents a technically compelling rootkit foundation: programs run in kernel context, can hook a broad range of kernel functions, and require no kernel module loading—they can be loaded by any process with the `CAP_BPF` capability or, at lower kernel versions, with root. Unlike traditional loadable kernel module rootkits, eBPF programs do not modify kernel code and may evade integrity checks that look for unauthorized module loads. Defensive eBPF-based detection tools such as Tetragon and Falco can identify suspicious eBPF program loads, but only if deployed before the rootkit is active; post-compromise detection using these tools cannot be assumed reliable if the rootkit has already hidden the relevant process identifiers.

Structural Vulnerabilities Exploited

Atomic Arch exposes two structural weaknesses that are not unique to the AUR and affect the broader open-source package ecosystem. The first is trust inheritance without transparency: because AUR provides no user-visible signal when a package changes maintainers, users and automated update systems alike treat a post-adoption PKGBUILD as carrying the legitimacy of the package's entire prior history. Packages with years of accumulated users and integration into automated pipelines became delivery vehicles the moment an attacker submitted an adoption request [4][5].

The second structural weakness is the unconstrained install-hook execution surface. PKGBUILDs and npm preinstall and postinstall scripts execute arbitrary shell commands in the build context with no content policy, no sandboxing, and no review gate triggered by maintainer changes. The decision to stage the payload through an npm package rather than embedding it directly in the PKGBUILD further illustrates this: the attack crossed two package registries, each of which has its own partial controls, but neither applied any scrutiny to the cross-registry injection at the point of execution.

Recommendations

Immediate Actions

Organizations operating Arch-based developer workstations or self-hosted CI/CD runners should immediately audit all AUR packages installed or updated on or after June 11, 2026. The community detection tool published at github.com/lenucksi/aur-malware-check [6] cross-references installed

packages against known-compromised lists. Build cache directories, systemd unit paths (`/etc/systemd/system/` and `~/.config/systemd/user/`), and the eBPF pin directory `/sys/fs/bpf/` should be examined for evidence of the injection strings `npm install atomic-lockfile`, `bun install js-digest`, and `src/hooks/deps`. The SHA-256 hash of the known `deps` payload binary is `6144d433f8a0316869877b5f834c801251bbb936e5f1577c5680878c7443c98b` [4].

Credential rotation should be treated as mandatory for any host where a potentially compromised AUR package was installed during the exposure window, regardless of whether direct evidence of compromise is found. The full scope of rotation should include GitHub personal access tokens and deploy keys, npm tokens, AWS and cloud provider API keys, SSH private keys, HashiCorp Vault tokens, Docker registry credentials, OpenAI API keys, and browser profiles. Teams using shared CI runners should treat all secrets that transited those environments as exposed. For any host where the payload executed with root privileges, the guidance from Arch Linux security maintainers and Sonatype is explicit: reinstall from trusted media [1][2]. Within the running compromised OS, the kernel's view of the filesystem cannot be trusted, and no in-place cleanup procedure can be verified as complete without booting from external trusted media.

Short-Term Mitigations

Development teams dependent on AUR packages should adopt a policy of reviewing PKGBUILD files and `.install` hooks before building, with heightened scrutiny for packages that recently changed maintainers or that became active following a period of dormancy. The AUR package page displays the current maintainer and last-updated timestamp; cross-referencing against the package's history is a practical first filter. AUR helpers such as paru and yay support configuration options that present PKGBUILD diffs before installation, providing a manual review checkpoint for injected hook logic.

For CI/CD pipelines, restricting build environments to official Arch repositories or AUR packages pinned by content hash—rather than mutable package names resolved at build time—substantially reduces exposure to this class of attack. Containerized or ephemeral build environments that are fully destroyed after each run limit the persistence window even when a compromised package executes. Network egress policies restricting outbound connections from build hosts to known registries and artifact stores would reduce the exfiltration effectiveness of payloads that communicate with Tor or arbitrary file-sharing services.

Transitioning from long-lived credentials to short-lived, scoped identities reduces the value of any secrets exposed during a workstation compromise. GitHub App tokens scoped to specific repositories, cloud workload identity federation instead of access key pairs, and just-in-time credential issuance limit

how many systems a single compromised workstation can unlock. Secrets injected into build environments via dedicated secrets management systems rather than persisted in configuration files reduce the local secrets footprint that an infostealer can harvest.

Strategic Considerations

Atomic Arch argues for structural reform in how open-source package repositories handle maintainer transitions. The AUR's orphaned-package adoption pathway currently provides no consumer-facing signal that a package changed hands, no minimum review period before adoptions become active, and no cryptographic continuity mechanism tying a new maintainer to the history of the package. The Arch Linux project and analogous communities in the broader Linux packaging ecosystem should evaluate mechanisms that surface maintainer changes visibly to users, introduce a probation or review window for newly adopted packages, or require existing community members to vouch for new maintainers inheriting popular packages [4][5].

At the organizational level, the categories of credentials targeted by this attack—cloud provider keys, source control tokens, container registry credentials—are precisely the secrets that provide pathways from a compromised developer machine into production infrastructure. Organizations with mature software supply chain security programs should extend their scope to include developer workstations as a potential entry point to critical systems, applying the same rigor to workstation security posture as they would to internet-facing servers. Software Bill of Materials practices that currently cover application dependencies and container base images should be extended to include build-environment packages and developer toolchain components.

The eBPF rootkit component of this attack also warrants dedicated attention from teams assessing their detection capabilities. eBPF-based rootkits are not new in concept, but their deployment in a mass supply chain campaign—rather than targeted attacks—raises the baseline threat level for Linux workstation environments. Organizations should evaluate whether their endpoint detection stack has visibility into eBPF program loads and kernel hook activity, and whether those controls are deployed at the workstation tier as well as servers.

CSA Resource Alignment

This incident connects to several CSA frameworks and guidance areas that provide actionable structure for organizational response and longer-term posture improvement.

The Cloud Controls Matrix (CCM) addresses the governance dimensions of this attack most directly in its Application & Interface Security (AIS) domain, which requires organizations to assess and monitor third-party software components and to maintain current inventories of build-time and runtime dependencies [7]. The Supply Chain Management, Transparency and Accountability (STA) domain extends this to continuous third-party dependency monitoring and software provenance requirements. Atomic Arch demonstrates that these controls must reach community-sourced build tools and developer workstation packages—not only production application libraries or container images—to be effective against the actual threat surface.

CSA's MAESTRO framework for agentic AI threat modeling applies directly to this incident: the credential categories exfiltrated—OpenAI API keys, GitHub tokens for AI research repositories, and cloud provider credentials used to provision training infrastructure—map precisely to the access surfaces MAESTRO addresses. MAESTRO's Layer 1 (Foundation Model infrastructure) and Layer 4 (Agent Trust Boundaries) both address the integrity of build and deployment environments as a prerequisite for trustworthy AI system operation [8]. A compromised build host that exfiltrates API keys or pushes to model artifact storage undermines the trust model that agentic AI systems depend upon.

CSA's Zero Trust guidance addresses the workstation-tier risk directly. The implicit assumption that developer machines within a corporate network are trusted endpoints—and that credentials stored on them are adequately protected—is inconsistent with a Zero Trust posture. Short-lived credentials, hardware-backed key storage (FIDO2 and hardware security modules), and continuous device health attestation reduce the blast radius of workstation-level compromises regardless of the specific attack vector. NIST SP 800-207, the foundational Zero Trust architecture standard, and CSA's Zero Trust-aligned frameworks both emphasize that no device or user identity should be inherently trusted based on network location [9].

Finally, NIST's Secure Software Development Framework (SSDF) provides a standards-grounded framework for operationalizing supply chain security controls across the software development lifecycle [10]. SSDF practices under the "Protect the Software" category—including verifying the integrity of third-party components, using approved tools and toolchains, and auditing build environments—map directly to the preventive controls that would have limited the effectiveness of Atomic Arch against a prepared organization.

References

[1] L. Abrams. "Over 400 Arch Linux packages compromised to push rootkit, infostealer." BleepingComputer, June 12, 2026.

[2] Sonatype Research. "Atomic Arch: Attackers Hijack Trusted AUR Packages to Deliver Rootkit-Like Malware." Sonatype, June 2026.

[3] PrivacyGuides. "Around 1,500 AUR Packages Compromised with 'Rootkit-Like' Malware." PrivacyGuides.org, June 12, 2026.

[4] R. Lakshmanan. "Over 400 Arch Linux AUR Packages Hijacked to Deploy Infostealer and eBPF Rootkit." The Hacker News, June 2026.

[5] StepSecurity. "400+ AUR Packages Hijacked: What the 'Atomic Arch' Campaign Means for Supply-Chain Security." StepSecurity, June 2026.

[6] lenucksi. "aur-malware-check: Detection Tools for the June 2026 AUR Supply-Chain Attack." GitHub, June 2026.

[7] Cloud Security Alliance. "Cloud Controls Matrix v4." CSA, 2021.

[8] Cloud Security Alliance. "Agentic AI Threat Modeling Framework: MAESTRO." CSA, February 2025.

[9] NIST. "Special Publication 800-207: Zero Trust Architecture." NIST CSRC, August 2020.

[10] NIST. "Secure Software Development Framework (SSDF) SP 800-218." NIST CSRC, April 2022.

[11] BleepingComputer. "Popular NPM library hijacked to install password-stealers, miners." BleepingComputer, October 2021.

[12] L. Tal. "peacenotwar malicious module puts npm ecosystem at risk from node-ipc." Snyk, March 2022.