INTELLIGENCE BRIEFING
Security Command Center
TLP:CLEAR
2026-06-16 19:21 UTC

Vulnerable OpenSSL Bundled in Python cryptography Package Wheels (GHSA-537c-gmf6-5ccf)

CVE VULNERABILITY | HIGH

SCC Item ID	SCC-CVE-2026-0313
Type	CVE Vulnerability
Severity	HIGH
Affected Products	cryptography (PyPI), specific affected versions not extractable from provided data; see OSV advisory GHSA-537c-gmf6-5ccf for version range
Published	2026-06-15T20:12:27Z
Discovery Source	Osv

Executive Summary

The Python 'cryptography' package, widely used in enterprise software and cloud applications, bundles OpenSSL directly inside its pre-built installation artifacts. Because OpenSSL is embedded rather than pulled from the operating system, standard OS-level security patching does not remediate the exposure. Development and security teams must update the Python package itself. Organizations using this library in any application that handles encrypted communications, credentials, or sensitive data may be running a vulnerable cryptographic dependency without knowing it.

Technical Analysis

GHSA-537c-gmf6-5ccf documents a supply chain risk in the 'cryptography' PyPI package (pyca/cryptography). Pre-built wheel artifacts distributed via pip bundle a vendored OpenSSL binary at build time. When that bundled OpenSSL version carries known vulnerabilities, every application installing the affected wheel inherits the flaw. Critically, this bypasses OS-level remediation: a fully patched Linux or Windows host running a vulnerable wheel still contains the vulnerable OpenSSL binary within the Python package's file tree. The underlying vulnerability class is CWE-1104 (Use of Unmaintained Third-Party Components). MITRE ATT&CK maps this pattern to T1195.001 (Supply Chain Compromise: Compromise Software Dependencies and Development Tools). Consult the OSV advisory at https://osv.dev/vulnerability/GHSA-537c-gmf6-5ccf for the authoritative version range and upstream CVE references. No CVSS score was available at the time of publication; severity is assessed as high based on advisory classification and supply-chain impact. EPSS data was not available.

Action Checklist

1. Step 1: Discovery, Identify all applications and services in your environment that install 'cryptography' via pip. Run 'pip list' or query your dependency manifests (requirements.txt, pyproject.toml, Pipfile.lock) across all environments, including CI/CD pipelines and container images, to enumerate affected deployments. Prioritize internet-facing services first. Reference: CIS 2.1, Establish and Maintain a Software Inventory.
2. Step 2: Detection, Determine whether installed wheels contain the vulnerable bundled OpenSSL by checking the installed version of 'cryptography' against the affected version range in GHSA-537c-gmf6-5ccf. Use 'pip show cryptography' in each environment. For containerized workloads, scan images using a software composition analysis tool. Check your SBOM if one exists; if not, this gap is itself a finding. Reference: CIS 7.1, Establish and Maintain a Vulnerability Management Process; NIST SI-4, Information System Monitoring.
3. Step 3: Eradication, Upgrade the 'cryptography' package to the version specified in GHSA-537c-gmf6-5ccf as the fixed release. Run 'pip install --upgrade cryptography' and pin the remediated version in all dependency manifests. Rebuild and redeploy all container images that bundled the affected wheel. Do not rely on OS-level OpenSSL patching; only upgrading the Python package remediates the embedded binary. Reference: CIS 7.3, Perform Automated Operating System Patch Management; CIS 7.4, Perform Automated Application Patch Management.
4. Step 4: Recovery, After upgrading, re-run 'pip show cryptography' and verify the installed version matches the patched release. Re-scan affected container images with your SCA tool to confirm the vulnerable wheel is no longer present. Monitor application logs for cryptographic errors or unexpected TLS failures that may indicate a misconfiguration introduced during upgrade. Reference: NIST AU-6, Audit Record Review, Analysis, and Reporting.
5. Step 5: Post-Incident, Establish or expand your software bill of materials (SBOM) process to capture vendored and bundled dependencies, not just direct package declarations. Integrate SCA scanning into CI/CD pipelines to catch future bundled-dependency vulnerabilities before deployment. Review your vulnerability management policy to ensure it explicitly covers transitive and vendored dependencies. Reference: CIS 2.1, Establish and Maintain a Software Inventory; CIS 7.2, Establish and Maintain a Remediation Process; NIST AC-6, Least Privilege.

IR / Forensic Enrichment

Triage Priority	URGENT
Escalation Criteria	Escalate to CISO and legal/compliance if discovery confirms the vulnerable cryptography package was installed in any application processing PII, PHI, or financial credentials during the exposure window, as the embedded OpenSSL vulnerability may constitute a cryptographic failure requiring breach notification assessment under GDPR, HIPAA, or PCI-DSS; also escalate if SCA scanning reveals the package is present in production internet-facing services and no compensating TLS termination control (WAF, load balancer) exists upstream.

Recovery Notes	After upgrading the cryptography package across all environments, verify that no application is silently falling back to a degraded TLS configuration by monitoring Python application error logs for `ssl.SSLError` and `cryptography.exceptions.InternalError` tracebacks for a minimum of 72 hours post-deployment, as API surface changes between cryptography versions occasionally break callers relying on deprecated OpenSSL bindings. Re-run SCA scans against all rebuilt container images using their new digests to confirm the vulnerable wheel is absent at the layer level, not merely overwritten in the top layer. Update your vulnerability management SLA documentation to explicitly classify bundled-dependency advisories (GHSA-class) as equivalent in urgency to direct CVE disclosures, since OS-level patching provides zero remediation for this class of vulnerability.
Forensic Artifacts	Installed cryptography wheel dist-info RECORD file at $(pip show cryptography | grep Location | awk '{print $2}')/cryptography-.dist-info/RECORD — contains SHA256 hashes of all files in the wheel including the bundled OpenSSL shared object, providing cryptographic proof of which binary version was installed at time of discovery | Bundled OpenSSL shared library on disk — located inside the cryptography package directory as a .so file (e.g., _openssl.abi3.so or a Rust-compiled binding); run `strings ` to extract the embedded OpenSSL version string and `sha256sum ` to fingerprint the binary for comparison against known-vulnerable wheel hashes published in GHSA-537c-gmf6-5ccf | Python process memory (live systems only, capture before any upgrade or restart) — a full memory dump of any long-running Python process using the cryptography library may contain in-memory TLS session keys, certificate material, or plaintext from recently decrypted data handled by the vulnerable OpenSSL binary; capture with `procdump` (Linux: `gcore `) before service restart | Application TLS handshake and error logs — for web frameworks (Django, Flask, FastAPI) using cryptography for HTTPS or mTLS, collect access logs and error logs covering the full exposure window; look for anomalous TLS negotiation failures, unexpected cipher suite downgrades, or client certificate errors that could indicate active exploitation of the bundled OpenSSL vulnerability | CI/CD pipeline build logs and container image layer history — run `docker history --no-trunc ` and retrieve build logs from your pipeline for all images built during the exposure window; these establish which image digests contain the vulnerable cryptography wheel and whether any attacker-controlled dependency injection could have substituted a malicious wheel during the pip install step

Per-Action IR Details

Step 1: Containment — Identify all applications and services in your environment that install 'cryptography' via pip. Run 'pip list' or query your dependency manifests (requirements.txt, pyproject.toml, Pipfile.lock) across all environments, including CI/CD pipelines and container images, to enumerate affected deployments. Prioritize internet-facing services first. Reference: CIS 2.1 — Establish and Maintain a Software Inventory.

NIST Phase: Containment

Reference: NIST 800-61r3 §3.3 — Containment Strategy: scope the affected population before taking remediation action to prevent incomplete remediation and to prioritize blast radius.

Controls: CIS 2.1 (Establish and Maintain a Software Inventory), CIS 1.1 (Establish and Maintain Detailed Enterprise Asset Inventory)

Compensating: For teams without an enterprise software inventory platform, run the following across all hosts and CI environments: `pip list --format=json | python3 -c "import sys,json; [print(p['name'],p['version']) for p in json.load(sys.stdin) if 'cryptography' in p['name'].lower()]"`. For container images, use `grype` (free, Anchore) or `syft` to generate an SBOM from each image layer: `syft -o json | jq '.artifacts[] | select(.name=="cryptography")'`. Enumerate Pipfile.lock and pyproject.toml files recursively: `find / -name 'Pipfile.lock' -o -name 'pyproject.toml' -o -name 'requirements*.txt' 2>/dev/null | xargs grep -l cryptography`.

Evidence: This step does not alter live system state, so volatile capture is not a prerequisite. However, before any subsequent remediation action, preserve the current installed package state as a snapshot: run `pip list --format=freeze > pip_snapshot_$(hostname)_$(date +%Y%m%d%H%M%S).txt` on each affected host. For containerized workloads, capture the image digest and layer manifest (`docker inspect > docker_inspect_$(date +%Y%m%d%H%M%S).json`) to establish a pre-remediation baseline. These artifacts document which version of the bundled OpenSSL binary was present in the wheel at the time of discovery.

Step 2: Detection — Determine whether installed wheels contain the vulnerable bundled OpenSSL by checking the installed version of 'cryptography' against the affected version range in GHSA-537c-gmf6-5ccf. Use 'pip show cryptography' in each environment. For containerized workloads, scan images using a software composition analysis tool. Check your SBOM if one exists; if not, this gap is itself a finding. Reference: CIS 7.1 — Establish and Maintain a Vulnerability Management Process; NIST SI-4 is mapped to audit and monitoring but no directly mapped NIST control from the provided reference applies to SCA scanning — no mapped control for SCA tooling in provided data.

NIST Phase: Detection Analysis

Reference: NIST 800-61r3 §3.2 — Detection and Analysis: confirm the presence and scope of the vulnerable artifact, correlate across environments, and establish whether exploitation indicators exist alongside the vulnerable package.

Controls: CIS 7.1 (Establish and Maintain a Vulnerability Management Process), CIS 2.2 (Ensure Authorized Software is Currently Supported)

Compensating: Run `pip show cryptography` on each host and capture the 'Version' and 'Location' fields. To confirm which OpenSSL binary is bundled inside the installed wheel, inspect the wheel's shared library directly: `find $(python3 -c 'import cryptography, os; print(os.path.dirname(cryptography.__file__))') -name '*.so' | xargs strings | grep -E 'OpenSSL [0-9]+\.[0-9]+'`. For container images without a running instance, use `syft -o cyclonedx-json` (free) to extract the embedded OpenSSL version from the cryptography wheel layers. If no SBOM exists, document its absence as a separate finding under CIS 2.1.

Evidence: This is a detection step and does not alter live state. Capture the following before any upgrade action: (1) `pip show cryptography` output per host, noting the exact version string and install path; (2) the wheel's `.dist-info/RECORD` file (located at `$(pip show cryptography | grep Location | awk '{print $2}')/cryptography-*.dist-info/RECORD`) to identify the bundled OpenSSL shared object filename and hash; (3) for internet-facing services, capture active TLS connection state with `ss -tnp` or `netstat -tnp` to document which processes are currently using the vulnerable library for live encrypted sessions — these connections will be disrupted during upgrade.

Step 3: Eradication — Upgrade the 'cryptography' package to the version specified in GHSA-537c-gmf6-5ccf as the fixed release. Run 'pip install --upgrade cryptography' and pin the remediated version in all dependency manifests. Rebuild and redeploy all container images that bundled the affected wheel. Do not rely on OS-level OpenSSL patching; only upgrading the Python package remediates the embedded binary. Reference: CIS 7.3 — Perform Automated Operating System Patch Management; CIS 7.4 — Perform Automated Application Patch Management.

NIST Phase: Eradication

Reference: NIST 800-61r3 §3.4 — Eradication: remove the vulnerable artifact from all affected environments; for bundled-dependency vulnerabilities, eradication requires package-level replacement, not OS patch application.

Controls: CIS 7.4 (Perform Automated Application Patch Management), CIS 7.2 (Establish and Maintain a Remediation Process), CIS 2.2 (Ensure Authorized Software is Currently Supported)

Compensating: For teams without automated patch orchestration, execute the upgrade in each virtualenv or system Python environment: `pip install --upgrade cryptography==[FIXED_VERSION_FROM_GHSA-537c-gmf6-5ccf]`. Pin the version immediately in requirements files: `sed -i 's/cryptography.*/cryptography==[FIXED_VERSION]/' requirements.txt`. For container images, add a forced layer rebuild step to your Dockerfile: `RUN pip install --no-cache-dir cryptography==[FIXED_VERSION]` and rebuild with `docker build --no-cache`. Track remediation status per host in a simple spreadsheet logging hostname, environment (prod/staging/CI), previous version, upgraded

version, and timestamp.

Evidence: BEFORE upgrading any host, capture the following volatile and pre-change state: (1) active network connections using the cryptography library — run `lsof -p $(pgrep -f python) | grep -i ssl` or `ss -tnp` and record all established TLS sessions that will be interrupted; (2) running process list for all Python processes: `ps aux | grep python` — document PIDs, command lines, and associated users, as the upgrade may require service restarts that clear in-memory session state; (3) the existing wheel's bundled OpenSSL shared object hash: `sha256sum $(find /usr/lib/python3 -name '_openssl*.so' -o -name '_rust*.so' 2>/dev/null | head -5)` to document the pre-patch binary fingerprint for forensic baseline. Do not upgrade until these captures are complete and stored off-host.

Step 4: Recovery — After upgrading, re-run 'pip show cryptography' and verify the installed version matches the patched release. Re-scan affected container images with your SCA tool to confirm the vulnerable wheel is no longer present. Monitor application logs for cryptographic errors or unexpected TLS failures that may indicate a misconfiguration introduced during upgrade. Reference: NIST AU-6 — Audit Record Review, Analysis, and Reporting.

NIST Phase: Recovery

Reference: NIST 800-61r3 §3.5 — Recovery: verify that affected systems are returned to a known-good state and monitor for regression or newly introduced failure conditions before closing the incident.

Controls: AU-6 (Audit Record Review, Analysis, And Reporting), CIS 7.1 (Establish and Maintain a Vulnerability Management Process), AU-2 (Event Logging)

Compensating: Run the following post-upgrade verification sequence: (1) `pip show cryptography | grep Version` — confirm the version string matches the fixed release from GHSA-537c-gmf6-5ccf; (2) re-run the bundled OpenSSL string extraction: `find $(python3 -c 'import cryptography, os; print(os.path.dirname(cryptography.__file__))') -name '*.so' | xargs strings | grep -E 'OpenSSL [0-9]+\.[0-9]+'` and confirm the embedded version is no longer vulnerable; (3) tail application logs for SSL/TLS handshake errors: `grep -iE '(ssl|tls|handshake|certificate|openssl)' /var/log//*.log` for 24 hours post-upgrade; (4) for containerized workloads, re-run `syft -o cyclonedx-json` and diff against the pre-remediation SBOM.

Evidence: Recovery verification is non-destructive, but retain the following as closure evidence: (1) post-upgrade `pip show cryptography` output per host with timestamp; (2) SCA re-scan report confirming the vulnerable wheel hash is absent from all container image layers; (3) application TLS error log excerpts from the 24-hour monitoring window showing no cryptographic failures attributable to the upgrade — specifically watch for `ssl.SSLError`, `OpenSSL.SSL.Error`, or `cryptography.exceptions` tracebacks in Python application logs, which would indicate an API compatibility break introduced by the version change.

Step 5: Post-Incident — Establish or expand your software bill of materials (SBOM) process to capture vendored and bundled dependencies, not just direct package declarations. Integrate SCA scanning into CI/CD pipelines to catch future bundled-dependency vulnerabilities before deployment. Review your vulnerability management policy to ensure it explicitly covers transitive and vendored dependencies. Reference: CIS 2.1 — Establish and Maintain a Software Inventory; CIS 7.2 — Establish and Maintain a Remediation Process; NIST AC-6 — Least Privilege (scope reduction of affected package access).

NIST Phase: Post Incident

Reference: NIST 800-61r3 §4 — Post-Incident Activity: conduct lessons-learned review and implement process improvements to reduce time-to-detection and time-to-remediation for future bundled-dependency vulnerabilities.

Controls: CIS 2.1 (Establish and Maintain a Software Inventory), CIS 7.2 (Establish and Maintain a Remediation Process), CIS 7.1 (Establish and Maintain a Vulnerability Management Process), AU-11 (Audit Record Retention)

Compensating: For teams without an enterprise SBOM platform: (1) integrate `syft` into CI/CD as a free pipeline step to generate CycloneDX SBOMs on every build — `syft -o cyclonedx-json > sbom-$(date +%Y%m%d).json`; (2) add OSV-Scanner (free, Google) as a pre-merge gate: `osv-scanner --lockfile requirements.txt` will flag transitive and vendored dependency vulnerabilities including future cryptography/OpenSSL bundling issues; (3) add a Dependabot or Renovate (both free for public/private repos) configuration targeting Python ecosystems to auto-raise PRs when cryptography or similar packages publish patched releases; (4) document the SBOM gap identified in Step 2 as a formal finding in your vulnerability register with a remediation due date.

Evidence: No live system state is altered in this phase. Retain as lessons-learned artifacts: (1) the timeline delta between GHSA-537c-gmf6-5ccf publication date and your organization's first detection of affected deployments — this gap quantifies the SBOM and SCA coverage deficit that allowed bundled OpenSSL to persist undetected; (2) the full inventory of affected hosts, environments, and container image digests compiled during Step 1, preserved as the incident scope record; (3) pre- and post-remediation SBOM diffs demonstrating that the vulnerable cryptography wheel version and its embedded OpenSSL binary have been removed from all tracked environments.

Detection Guidance

Query all deployment environments for the installed version of the 'cryptography' package: run 'pip show cryptography' or 'pip list | grep cryptography' on hosts, in container images, and within CI/CD build agents. Cross-reference the installed version against the affected range in GHSA-537c-gmf6-5ccf. For container-based environments, scan images with a software composition analysis tool capable of detecting bundled native libraries inside Python wheels; standard OS-level package scanners will miss the embedded OpenSSL binary. If an SBOM exists, search it for 'openssl' entries sourced from Python wheel paths (typically under site-packages/cryptography/hazmat/bindings/). No network-based IOCs or behavioral indicators are available for this advisory; detection is inventory-based, not traffic-based. Reference: CIS 8.2, Collect Audit Logs; CIS 2.1, Establish and Maintain a Software Inventory.

Framework Mappings

MITRE-ATTACK

- **T1195.001** — Compromise Software Dependencies and Development Tools

OWASP-TOP10-2021

- **A06:2021** — Vulnerable and Outdated Components

NIST-800-53R5

- **SA-4** — Acquisition Process
- **SA-9** — External System Services
- **SR-2** — Supply Chain Risk Management Plan
- **SC-13** — Cryptographic Protection

CIS-V8

- **16.4** — Establish and Manage an Inventory of Third-Party Software Components
- **7.3** — Perform Automated Operating System Patch Management
- **7.4** — Perform Automated Application Patch Management
- **15.1** — Establish and Maintain an Inventory of Service Providers

ISO-27001-2022

- **A.8.8** — Management of technical vulnerabilities
- **A.5.21** — Managing information security in the ICT supply chain
- **A.8.24** — Use of cryptography

NIST-CSF-2

- **GV.SC-01** — Cybersecurity supply chain risk management program

SOC2-TSC

- **CC9.2** — Manages risks associated with vendors and business partners

HIPAA-SECURITY

- **164.312(e)(1)** — Transmission Security

MITRE ATT&CK Mapping

Technique ID	Technique Name	Tactic
T1195.001	Compromise Software Dependencies and Development Tools	Initial-Access

Sources

Source	URL	Tier
osv	https://osv.dev/vulnerability/GHSA-537c-gmf6-5ccf	T3
cryptography - Snyk Vulnerability Database	https://security.snyk.io/package/pip/cryptography	T3
cryptography - PyPI	https://pypi.org/project/cryptography/	T3
Six Malicious Python Packages in the PyPI Targeting Windows Users	https://unit42.paloaltonetworks.com/malicious-packages-in-pypi/	T3
pyca/cryptography: cryptography is a package designed to ... - GitHub	https://github.com/pyca/cryptography	T3

Generated 2026-06-16 19:21 UTC by TJS Security Command Center