



Auditing Report

Hardening Blockchain Security with Formal Methods

FOR



Aleo Oracle



Veridise Inc.
Oct. 23, 2025

► **Prepared For:**

Venture23 Inc.

<https://www.venture23.xyz/>

► **Prepared By:**

Petr Susil

Jon Stephens

► **Contact Us:**

contact@veridise.com

► **Version History:**

Oct. 23, 2025 V2

Sep. 11, 2025 V1

Contents

Contents	iii
1 Executive Summary	1
2 Project Dashboard	6
3 Security Assessment Goals and Scope	7
3.1 Security Assessment Goals	7
3.2 Security Assessment Methodology & Scope	8
3.3 Classification of Vulnerabilities	9
4 Trust Model	10
4.1 Operational Assumptions	10
4.1.1 Operational Assumptions	10
4.1.2 Systemic Assumptions	10
4.2 Privileged Components and Roles	11
5 Vulnerability Report	14
5.1 Detailed Description of Issues	16
5.1.1 V-ORC-VUL-001: Missing rate limits can enable oracle manipulation and DoS	16
5.1.2 V-ORC-VUL-002: Integrity of timestamps	18
5.1.3 V-ORC-VUL-003: HTTP response splitting / header injection vulnerability	19
5.1.4 V-ORC-VUL-004: Outlier-insensitive VWAP propagates exchange errors	21
5.1.5 V-ORC-VUL-005: SGX Attestation Does Not Cover Aleo Address or Signature	23
5.1.6 V-ORC-VUL-006: Long term exposure of Aleo private key	25
5.1.7 V-ORC-VUL-007: Insecure PCCS configuration undermines SGX quote verification	27
5.1.8 V-ORC-VUL-008: Handling of SWHardeningNeeded status	29
5.1.9 V-ORC-VUL-009: Dropped exchange information risks price accuracy	30
5.1.10 V-ORC-VUL-010: Missing Enclave Key Rotation and Attestation Enforcement Enables Long-Term Key Misuse	32
5.1.11 V-ORC-VUL-011: Missing quote verification in notarization backend	34
5.1.12 V-ORC-VUL-012: Timing Variations in Scalar Multiplication of Private Key	37
5.1.13 V-ORC-VUL-013: GetPriceFeed May Hang on Goroutine Failure	39
5.1.14 V-ORC-VUL-014: LVI mitigations	40
5.1.15 V-ORC-VUL-015: Contract assumes fixed-length attestation data leaving string attestation data unsupported	43
5.1.16 V-ORC-VUL-016: Unbounded request bodies in verification APIs enable memory exhaustion	44
5.1.17 V-ORC-VUL-017: Encoded price feed proof data cannot be decoded	45
5.1.18 V-ORC-VUL-018: Silent Truncation of HTML Responses via LimitReader Can Lead to Incomplete Attestation Data	48

5.1.19	V-ORC-VUL-019: Duplicate Exchange Prices Can Skew Volume-Weighted Average	50
5.1.20	V-ORC-VUL-020: TLS Interception Risk	52
5.1.21	V-ORC-VUL-021: Open redirect risk in outbound HTTP requests introduces new attack vectors	54
5.1.22	V-ORC-VUL-022: Channel Blocking and Goroutine Leak in executeRequest	55
5.1.23	V-ORC-VUL-023: Missing keep-alive hardening of notarization server	57
5.1.24	V-ORC-VUL-024: Spectre, Meltdown, ZombieLoad, Downfall mitigation	58
5.1.25	V-ORC-VUL-025: Silent Attestation Loss Due to Front-Loaded Report Errors	60
5.1.26	V-ORC-VUL-026: Field Name Mismatch	62
5.1.27	V-ORC-VUL-027: Partial Failure Masking in GetEnclavesInfo	64
5.1.28	V-ORC-VUL-028: Silent Truncation of >128-bit Integers in bigIntStrToBytes	66
5.1.29	V-ORC-VUL-029: Hardcoded Aleo API Request Path Reduces Maintainability	67
5.1.30	V-ORC-VUL-030: Unsynchronized Access to AleoContextManager State Causes Race Conditions	68
5.1.31	V-ORC-VUL-031: Logging is not integrated with Gramine's secure logging	69
5.1.32	V-ORC-VUL-032: DNS domain hijacking	70
5.1.33	V-ORC-VUL-033: Defensive Programming Gaps	71
5.1.34	V-ORC-VUL-034: Outdated versions for dependencies (internal&external)	76
5.1.35	V-ORC-VUL-035: Process-scoped mutex guards a system-wide attestation interface	78
5.1.36	V-ORC-VUL-036: Uncanceled verification requests in verifyReports	79
5.1.37	V-ORC-VUL-037: Repeated code in context cancellation and missing return	80
5.1.38	V-ORC-VUL-038: Debug mode is not rejected in verification logic, rejected only on-chain	81
5.1.39	V-ORC-VUL-039: Dangerous mutation of fields	83
5.1.40	V-ORC-VUL-040: Computational integrity of weighted average	85
5.1.41	V-ORC-VUL-041: Standardize SGX enclave info serialization across backend and SDK	87
5.1.42	V-ORC-VUL-042: DecodeQuoteHandler calls WriteHeader before setting Content-Type	89
5.1.43	V-ORC-VUL-043: Go SDK leaks connections when response read fails	90
5.1.44	V-ORC-VUL-044: Missing Cardinality Checks for PcrValuesTarget	92
5.1.45	V-ORC-VUL-045: Missing cache-control header permits stale attestation responses	93
5.1.46	V-ORC-VUL-046: Multiple Typos and Inconsistencies Across Codebase	94
5.1.47	V-ORC-VUL-047: Case-Sensitive String Comparisons in HTTP and URL Handling	95
5.1.48	V-ORC-VUL-048: HTTP success detection is limited to status code 200	97
5.1.49	V-ORC-VUL-049: SGXReport struct field order mismatch with Intel SGX specification	98
5.1.50	V-ORC-VUL-050: Missing rate limiting on SGX attestation endpoints	100
5.1.51	V-ORC-VUL-051: Strict Content-Type check rejects valid JSON requests	102

5.1.52 V-ORC-VUL-052: Misconverted HTTP status codes in metrics label . . .	103
Glossary	104

From Aug. 4, 2025 to Sep. 3, 2025, Venture23 Inc. engaged Veridise to conduct a security assessment of their Aleo Oracle. The security assessment covered the off-chain logic of the Aleo Oracle spread across six repositories. The main components included an Intel SGX notarization service built on [Gramine](#) that attests to user queries, a verification service used to determine the validity of an Intel SGX or AWS nitro notarization and a SDK that can be used to interact with these services. Veridise conducted the assessment over 8 person-weeks, with 2 security analysts reviewing the project over 4 weeks on the commits listed in the project dashboard. The review strategy involved a thorough code review of the program source code performed by Veridise security analysts.

Project Summary. The security assessment covered the [notarization backend](#), [verification backend](#), [aleo-oracle-encoding repository](#), [aleo-utils-go repository](#), and [Go/JS SDKs](#). While the overall oracle framework also includes on-chain Aleo smart contracts, those components fall under a separate report and are not included in this assessment*. This audit focuses specifically on the off-chain infrastructure and the guarantees it provides. The off-chain infrastructure consists of three main components: a notarization service, a verification service and a SDK used to interact with these services.

The main functionality of the notarization service is to provide trusted attestations to requested user data from whitelisted exchanges. The notarization service is run within a [TEE](#) and holds a secret aleo key both of which attest to produced data. This allows attestations to be audited by ensuring that it was produced by a TEE running a whitelisted application and by a trusted entity. The notarization service allows specific data to be requested from an exchange by providing a URL along with instructions of how to extract the requested data (e.g. with a JSON selector). The exchange will make the request on behalf of the user and attest to the response with a TEE report and a signature using the included private key. In addition to user request notarizations, the notarization service is also used to construct a price feed by aggregating prices from whitelisted exchanges with a volume-weighted average price (VWAP) computation. Similar to the user notarizations, the price feed calculation is attested to by the notarization service in a TEE report and by signing data that includes the price with the private Aleo key. The attestations produced by the notarization service can then be consumed by the verification service or submitted to the on-chain contracts to update the Aleo oracle program.

The on-chain contracts can be used to verify the Aleo signature produced by the notarization service, but it cannot verify the TEE reports. As such, Venture23 Inc. developers also provide a verification service that can be used to verify the integrity of the TEE reports so that an alarm can be raised if an invalid attestation is accepted on-chain. More specifically, the verification service consumes an attestation from the notarization service and ensures that the report was generated by a whitelisted enclave with the proper configuration.

*The audit report for these components can be found on Veridise website at <https://veridise.com/audits-archive/>

Users may interact with both the notarization service and verification service using a SDK written in either Go or Typescript. These services query the endpoints provided by configured notarization and verification services so that they may produce attestations, verify attestations and query information about the services.

Code Assessment. The Aleo Oracle developers provided the source code of the Aleo Oracle oracles for the code review. The source code was initially created by a separate development team and was subsequently extended and modified by Venture23.

The notarization backend includes a dedicated docs directory covering architecture, deployment, APIs, and error codes. Its Go sources are well commented, and many packages provide function-level docstrings explaining routing, request handling, constants, and errors, which improves readability and maintenance. In contrast, most other backends and utilities offer only brief READMEs.

To facilitate the Veridise security analysts' understanding of the code, the Aleo Oracle developers shared documentation for the Oracle Program[†], met with the Veridise analysts weekly to answer questions regarding the project, and provided an initial walkthrough of the codebase.

The source code contained a test suite, which the Veridise security analysts noted demonstrated comprehensive unit coverage in the notarization backend.

Summary of Issues Detected. The security assessment uncovered 52 issues, 5 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, [V-ORC-VUL-001](#) found that the attestation endpoint has no throttling or budgeting, allowing attackers to spam requests, trigger exchange bans, or selectively disable exchanges to skew feeds. [V-ORC-VUL-002](#) showed that reliance on the host clock makes timestamps manipulable, where a shifted clock can freeze updates, accept stale data, or distort smart contract logic. [V-ORC-VUL-005](#) noted that SGX quotes do not bind the Aleo address or signature, enabling impersonation and key-substitution attacks by combining a valid quote with another keypair. [V-ORC-VUL-004](#) reported that VWAP lacks outlier filters or weight caps, so a single bad exchange with large volume can dominate pricing and repeat past oracle failure modes. Finally, [V-ORC-VUL-003](#) observed that user-supplied headers are forwarded without checks, allowing malicious headers to alter outbound requests while remaining masked in notarized data, enabling hidden redirects or injections. The Veridise analysts also identified 14 medium-severity issues, including [V-ORC-VUL-006](#) which describes how long-lived Aleo private keys are stored as immutable Go strings and copied into unmanaged WASM memory, leaving multiple plaintext buffers that cannot be wiped and raising the risk of side-channel extraction. [V-ORC-VUL-008](#) showed that attestation reports with `SWHardeningNeeded` indicate missing microcode mitigations; without verifying `LFENCE` or an LVI-hardened toolchain, enclaves remain vulnerable to transient execution attacks. [V-ORC-VUL-010](#) highlighted that enclave signing keys lack enforced rotation or expiry, leaving old or compromised keys valid indefinitely. 12 low-severity issues, 13 warnings, and 8 informational findings were also identified during the course of the review.

[†] Documentation for the Oracle program is available at <https://aleo-oracle-docs.surge.sh/guide/>

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Aleo Oracle.

Private Key Protections. The oracle's notarization service relies on a secret private key held within the enclave to sign attestations and thereby prove that an attestation was provided by a whitelisted service. Should this private key be leaked, arbitrary attestations (including prices for price feeds) may be signed and check in on-chain. While an alarm would hopefully be raised if the TEE report were validated, a careful attacker could avoid even this as mentioned in [V-ORC-VUL-005](#). It is therefore imperative that the private key be protected by mitigating issues that could result in a private key leak such as [V-ORC-VUL-006](#), [V-ORC-VUL-010](#), and [V-ORC-VUL-012](#). Additionally the Veridise analysts would strongly recommend implementing a system that could rapidly rotate private keys by killing a notarization service, blacklisting the associated key on-chain, launching a new notarization service and whitelisting the new key on-chain. They also recommend rotating keys regularly with explicit expirations that are enforced on-chain and would blacklist a private key either after a certain time period (e.g. a month) or after a certain number attestations.

Price Calculation Manipulation. While the oracle can attest to data retrieved from user queries, the main functionality appears to be resolving price feed queries. Price feeds are resolved by querying prices from whitelisted exchanges and aggregating responses by computing a volume-weighted average price. Using a VWAP can help defend against price manipulations but the Veridise analysts noted that manipulation is still possible in issues such as [V-ORC-VUL-001](#), [V-ORC-VUL-004](#) and [V-ORC-VUL-009](#). To protect against these issues, the Veridise analysts strongly recommend validating the freshness of returned prices and adding methods of identifying outliers that may skew the prices. Additionally, the Veridise analysts would recommend requiring prices from more than 2 exchanges, and add methods to ban exchanges that repeatedly provide price outliers. Finally, the analysts recommend implementing a tracking/pausing service that will monitor returned prices, validate them against more mature oracles and pause the oracle if prices ever deviate too much from the trusted oracles.

Architectural Modifications. During the course of the review, the Veridise analysts had several architectural suggestions to improve the security of the oracle. First, they suggest separating the arbitrary attestation service from the price feed service as recommended in [V-ORC-VUL-005](#). The specific attestation service provides users much more fine-grain control over which exchanges are queried and could be used to ban the service from querying specific exchanges. This could therefore impact the price feed computation logic and potentially allow oracle manipulations. By separating the services, it would be more difficult for malicious users to control which exchanges are queried. Second, as suggested in [V-ORC-VUL-005](#), the analysts recommend that the TEE reports also include information about the Aleo signing key either by including the public key or signature in the report. Currently the TEE report and Aleo signature are separate in that the TEE report is not signed by the Aleo private key and the TEE report does not identify the Aleo signing key. As a result, returned notarizations can be manipulated by combining the TEE report and signature from two different sources as long as the attestation data matches, allowing falsified notarizations to appear valid even to the verification service. Finally, during one of our meetings the developers indicated that they are considering requiring a signature from the verification service. The Veridise analysts note that this would improve security of the project, but only under certain circumstances. Specifically, the Aleo signing process should not be moved into the verification service but rather the verification service should add an additional signature and both signatures should be verified on-chain. This is because

the notarization signature currently provides a method of identifying the notarization service which would be lost otherwise. While the verification service would still confirm that the correct application is running, this would not be sufficient to protect against enclave environment manipulations such as those brought up in [V-ORC-VUL-002](#), [V-ORC-VUL-020](#), [V-ORC-VUL-024](#). Additionally, moving the key would, at best, result in a similar level of security as a single key compromise would allow a malicious user to create arbitrary attestations.

SGX Hardening. The notarization service executes within a Trusted Execution Environment (TEE), specifically leveraging the Gramine framework to support deployment inside Intel SGX enclaves with minimal source modifications. Enclave authenticity is attested by the EGo component, which additionally consults the Provisioning Certificate Caching Service (PCCS) to evaluate the current status of platform countermeasures against documented weaknesses. Nevertheless, the presence of an SGX enclave does not obviate all risks: certain classes of attacks, including side-channel leakage arising from speculative execution and fault injection, fall outside the formal SGX threat model. An adversary who successfully mounted such an attack could extract the Aleo private key, thereby enabling the forgery of notarizations and subverting the system's integrity guarantees. Consequently, it is critical to ensure that both kernel-level and userspace-level defenses are correctly deployed and continuously maintained. Effective mitigation requires a layered approach. The operating system kernel must enable appropriate hardening features, which remain disabled by default due to their performance impact. The compiler must introduce defenses against speculative execution, such as serialization barriers or retpolines, ensuring that speculative leakage paths are closed. Additionally, Intel advisories must be actively monitored to confirm that kernel protections and compiler-based mitigations remain aligned with current guidance. Finally, the PCCS instance itself must be kept up to date, since outdated provisioning or attestation data may lead to reliance on obsolete countermeasures.

Improve Testing. As mentioned above, the Veridise analysts assessed portions of the protocol to be insufficiently tested. To strengthen assurance, the testing strategy should emphasize how infrastructure handles oracle data under adverse conditions. This involves simulating API throttling and quota exhaustion, introducing artificial delays or dropped responses from providers, and verifying that retry and backoff logic maintains availability without cascading failures. Fuzzing should target the data pipeline by injecting malformed, duplicated, or extreme values to ensure they are filtered before reaching contracts. End-to-end simulations can combine sharp market swings with forced rate-limit responses to confirm monitoring and failover mechanisms activate properly. This approach strengthens confidence that external feeds remain reliable even under pressure.

Additional Security Reviews. Issues such as [V-ORC-VUL-001](#), [V-ORC-VUL-004](#), [V-ORC-VUL-009](#) and [V-ORC-VUL-005](#) relate to missing or insufficient protections against oracle manipulation attacks. Due to the number and size of the changes required to properly address these issues, the Veridise analysts recommend that the developers perform additional security reviews after fixing these issues. These reviews should focus specifically on the protections against oracle manipulation attacks and forgeries.

Disclaimer. Given the complexity of the Aleo Oracle, the size of the code base, and the scope of the proposed changes, the Veridise team cannot make guarantees about the absence of high/critical issues in the protocol beyond the 5 identified. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report

should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Table 2.1: Application Summary.

Name	Version	Type	Platform
Notarization Backend	137dbd0c	Go, Rust	Gramine SGX
Verification Backend	9f6285dd	Go	—
Utils	78a47dde	Go	—
Encoding	d4385824	Go	—
SDK Go	0a82cfaf	Go	—
SDK JS	c150794f	Typescript	—

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Aug. 4–Sep. 3, 2025	Manual & Tools	2	8 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	0
High-Severity Issues	4	4	3
Medium-Severity Issues	14	14	10
Low-Severity Issues	12	12	10
Warning-Severity Issues	13	13	8
Informational-Severity Issues	8	8	6
TOTAL	52	52	37

Table 2.4: Category Breakdown.

Name	Number
Data Validation	20
Logic Error	6
Maintainability	6
Denial of Service	5
Cryptographic Vulnerability	4
Usability Issue	4
SGX attestation	2
Transient Execution Injection	1
Side-channels	1
Serialization Error	1
Race Condition	1
Fault injection	1



3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Aleo Oracle's source code. During the assessment, the security analysts aimed to answer questions such as:

Price Oracle

- ▶ How does the system avoid single-source dependence and build resilience against manipulation through aggregation or cross-checking?
- ▶ What policy bounds constrain outputs, and how are exceptions reviewed, logged, and reverted?
- ▶ How are timestamps sourced, bound, and enforced so that freshness and anti-replay guarantees hold?
- ▶ Are verifier and service outcomes unambiguous and cryptographically checked, with robust success/failure signaling and error provenance?
- ▶ What rate limits, quotas, and backpressure protect oracle and attestation endpoints from DoS/exhaustion and stale or replayed responses?
- ▶ How are whitelists, domain lists, and retry/HTTP clients configured, authenticated, versioned, and audited to prevent bypass or unauthorized changes?
- ▶ How is authenticity and routing integrity assured from source to enclave, and what mitigations exist for MITM, SSRF, DNS, and BGP manipulation?
- ▶ How are all external inputs canonicalized, validated, and sanitized before use, including strict schemas and rejection of malformed or unexpected data?
- ▶ How are spoofed identities, header injection, parsing inconsistencies, and protocol-level attacks prevented across all request/response paths?

SGX / Enclave Security, Operational Security Key Management

- ▶ To what extent is the enclave protected against side-channels, timing leakage, and fault/-voltage/speculation injections, including constant-time crypto and platform/compiler mitigations?
- ▶ Does the attestation pipeline reliably prove the right code, in the right environment, with debug/test enclaves and outdated microcode rejected by design?
- ▶ Are SGX/CPU microcode, firmware, DCAP components, OS settings, and build flags current and continuously enforced, with drift detection and clear upgrade procedures?
- ▶ Given SGX limits, what guarantees and compensating controls ensure computational and memory integrity?
- ▶ How are all cryptographic keys generated, stored, rotated, retired, and recovered—and who/what is authorized to perform those actions?
- ▶ What operational and governance controls prevent central points of failure or unilateral key changes?
- ▶ How does the code base embed disaster recovery capabilities, and what guarantees exist for continuity under critical failures?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment a thorough review of the source code by human experts.

Scope. The scope of this security assessment is limited to the following locations:

- ▶ <https://github.com/venture23-aleo/aleo-utils-go>
 - /build.sh
 - /optimize_wasm.sh
 - /session.go
 - /wrapper.go
 - /cmd/*
 - /src/*
- ▶ <https://github.com/venture23-aleo/aleo-oracle-encoding>
 - /encoding.go
 - /positionRecorder/recorder.go
- ▶ <https://github.com/venture23-aleo/aleo-oracle-notarization-backend>
 - /cmd/server/main.go
 - /internal/*
 - * EXCLUDING *_test.go
 - * EXCLUDING /internal/config/config.json
- ▶ <https://github.com/venture23-aleo/oracle-verification-backend>
 - /main.go
 - /nitro-build.sh
 - /api/*
 - /attestation/*
 - * EXCLUDING /attestation/decoding_test.go
 - * EXCLUDING /attestation/nitro/*
 - /config/config.go
 - /contract/contract.go
 - /pccs/setup.sh
 - /u128/u128.go
- ▶ <https://github.com/venture23-aleo/aleo-oracle-sdk-go>
 - /client.go
 - /configs.go
 - /defaults.go
 - /dns.go
 - /info.go
 - /nooplogger.go
 - /notarize.go
 - /random.go
 - /request.go
- ▶ <https://github.com/venture23-aleo/aleo-oracle-sdk-js>

- /src/•

Methodology. Veridise security analysts reviewed the reports of previous audits for Aleo Oracle, inspected the provided tests, and read the Aleo Oracle documentation. They then began a manual review of the codebase. During the security assessment, the Veridise security analysts regularly met with the Aleo Oracle developers to ask questions about the code and to report new findings.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s)
	- OR -
Very Likely	Requires a small set of users to perform an action
	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user
	- OR -
Very Bad	Affects a very small number of people and requires aid to fix
	Affects a large number of people and requires aid to fix
	- OR -
Protocol Breaking	Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own



4.1 Operational Assumptions

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Aleo Oracle.

4.1.1 Operational Assumptions

- ▶ **Insider attack:** The operators and developers are trusted. Segregation of duties is enforced.
- ▶ **Configuration governance:** Runtime configurations and environment variables are correctly injected by a trusted operator.
- ▶ **Host OS integrity and patching:** The kernel, firmware, and SGX microcode remain current and uncompromised.
- ▶ **Enclave integrity:** Buffer overflows and RCE are out of scope.
- ▶ **Trusted host bindings:** [wazero](#) host functions are securely implemented and do not expose undefined behavior.
- ▶ **SGX verification:** The [EGo](#) verifier is assumed to be bug-free.
- ▶ **Honest verifier:** The OS, filesystem, and network stack used by the verifier do not alter or suppress verification data.
- ▶ **Integrity of proxied TLS data:** Proxies correctly set X-Real-IP, X-Forwarded-For, and related headers for the notarization service. Other data relayed by TLS proxies remains unchanged.
- ▶ **Trusted domain whitelisting:** Configured whitelisted domains remain stable and are not redirected to untrusted sources in the future.
- ▶ **Secure network termination:** TLS endpoints and proxies are configured with trusted certificates and follow good practices for protecting TLS private keys.
- ▶ **Verifier diligence:** Relying parties validate quotes, signatures, and certificate chains.

4.1.2 Systemic Assumptions

- ▶ **Dependency supply-chain security:** [Gramine](#), [wazero](#) modules, runtimes, and third-party libraries are properly vetted, version-pinned, and free of malicious behavior.
- ▶ **Faithful quote verification logic:** Parsing, signature validation, and revocation handling for SGX quotes are correct and complete.
- ▶ **Protected in-enclave key management:** The Aleo private key generated at startup remains bound to the enclave.
- ▶ **Cryptographic primitives:** Poseidon8 hashing and Schnorr signature generation are implemented correctly.
- ▶ **Defenses against memory exploits:** Enclave runtime and application code enforce memory safety, preventing buffer overflows, stack smashing, and return-oriented programming (ROP).

4.2 Privileged Components and Roles

Authority is divided between on-chain and off-chain components as follows.

Off-chain Notarization The notarization backend runs inside an SGX enclave, generating a fresh key pair on each start and producing and signing oracle reports, confirming that external data is bound to enclave-generated keys.

Off-chain Verification The verification service cross-checks reported enclave measurements against both reproducible builds and values retrieved from the Aleo contract, providing verification of TEE attestation.

SDK The SDK exposes default notarization and verification endpoints but lets integrators supply custom ones.

On-chain Contracts (out-of-scope) This service manages the interface between enclave keys and blockchain state. It enforces which keys are trusted on-chain and ensures that only authorized identities are permitted.

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. The roles are grouped based on two characteristics: privilege-level and time-sensitivity. *Highly-privileged* roles may have a critical impact on the protocol if compromised, while *limited-authority* roles have a negative, but manageable impact if compromised. Time-sensitive *emergency* roles may be required to perform actions quickly based on real-time monitoring, while *non-emergency* roles perform actions like deployments and configurations which can be planned several hours or days in advance.

During the review, Veridise analysts assumed that the role operators perform their responsibilities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- ▶ Highly-privileged, emergency roles:
 - None
- ▶ Highly-privileged, non-emergency roles:
 - Server Administrator
 - Aleo Smart Contract Admin
- ▶ Limited-authority, emergency roles:
 - None
- ▶ Limited-authority, non-emergency roles:
 - Logs and Metrics Analyst

Operational Recommendations. Highly-privileged, non-emergency operations that are performed on-chain should be operated by a multi-sig contract or decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly-privileged, emergency operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

Full validation of operational security practices is beyond the scope of this review. The service provider should ensure

- ▶ **CI/CD and Build Integrity** All enclave builds should be performed within a dedicated and isolated build environment. The build process must be deterministic and reproducible, enabling third parties to independently verify binary equivalence against the published source code.
- ▶ **Third-party libraries:** Known bugs in library dependencies are addressed through timely updates and enclave rebuilds.
- ▶ **Separation of Duties and Four-Eyes Control** Any approval process related to Aleo address registration for price-oracle updates shall adhere to the four-eyes principle. At least two independent and authorized individuals must jointly approve such actions to prevent unilateral decision-making.
- ▶ **Authorization of Aleo addresses** Authorization services and associated cryptographic material must be managed under strict security controls. Based on service guarantees and usage of SGX technology, Aleo keys for authorization of notarization addresses should be persisted within an SGX enclave or other secure hardware, with a securely stored offline archival copy maintained exclusively for disaster recovery scenarios.
- ▶ **Minimization of Manual Interaction** Manual intervention in Aleo address approvals should be restricted to the greatest extent possible. When unavoidable, approvals must be executed by a designated technical user, with a well-defined “break-glass” procedure in place to handle exceptional circumstances.
- ▶ **Independent Verification of Attestation** Client systems should operate an independent verification service, ensuring that trust is not centralized in a single entity. The verification process must validate all fields of the attestation response, thereby providing comprehensive due diligence and mitigating the risk of acceptance of malformed or malicious attestations.

Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- ▶ Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- ▶ Using separate keys for each separate function.
- ▶ Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- ▶ Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- ▶ Validating that no party has control over multiple multi-sig keys.
- ▶ Performing regularly scheduled key rotations for high-frequency operations.
- ▶ Securely storing physical, non-digital backups for critical keys.
- ▶ Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.

- ▶ Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5

Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-ORC-VUL-001	Missing rate limits can enable oracle . . .	Critical	Partially Fixed
V-ORC-VUL-002	Integrity of timestamps	High	Fixed
V-ORC-VUL-003	HTTP response splitting / header injection . . .	High	Fixed
V-ORC-VUL-004	Outlier-insensitive VWAP propagates . . .	High	Fixed
V-ORC-VUL-005	SGX Attestation Does Not Cover Aleo . . .	High	Acknowledged
V-ORC-VUL-006	Long term exposure of Aleo private key	Medium	Fixed
V-ORC-VUL-007	Insecure PCCS configuration undermines . . .	Medium	Fixed
V-ORC-VUL-008	Handling of SWHardeningNeeded status	Medium	Fixed
V-ORC-VUL-009	Dropped exchange information risks price . . .	Medium	Fixed
V-ORC-VUL-010	Missing Enclave Key Rotation and . . .	Medium	Acknowledged
V-ORC-VUL-011	Missing quote verification in notarization . . .	Medium	Partially Fixed
V-ORC-VUL-012	Timing Variations in Scalar Multiplication . . .	Medium	Fixed
V-ORC-VUL-013	GetPriceFeed May Hang on Goroutine Failure	Medium	Fixed
V-ORC-VUL-014	LVI mitigations	Medium	Acknowledged
V-ORC-VUL-015	Contract assumes fixed-length attestation . . .	Medium	Acknowledged
V-ORC-VUL-016	Unbounded request bodies in verification . . .	Medium	Fixed
V-ORC-VUL-017	Encoded price feed proof data cannot be . . .	Medium	Fixed
V-ORC-VUL-018	Silent Truncation of HTML Responses via . . .	Medium	Fixed
V-ORC-VUL-019	Duplicate Exchange Prices Can Skew . . .	Medium	Fixed
V-ORC-VUL-020	TLS Interception Risk	Low	Partially Fixed
V-ORC-VUL-021	Open redirect risk in outbound HTTP . . .	Low	Acknowledged
V-ORC-VUL-022	Channel Blocking and Goroutine Leak in . . .	Low	Fixed
V-ORC-VUL-023	Missing keep-alive hardening of . . .	Low	Fixed
V-ORC-VUL-024	Spectre, Meltdown, ZombieLoad, . . .	Low	Fixed
V-ORC-VUL-025	Silent Attestation Loss Due to Front- . . .	Low	Fixed
V-ORC-VUL-026	Field Name Mismatch	Low	Fixed
V-ORC-VUL-027	Partial Failure Masking in GetEnclavesInfo	Low	Fixed
V-ORC-VUL-028	Silent Truncation of >128-bit Integers in . . .	Low	Fixed
V-ORC-VUL-029	Hardcoded Aleo API Request Path . . .	Low	Fixed
V-ORC-VUL-030	Unsynchronized Access to . . .	Low	Fixed
V-ORC-VUL-031	Logging is not integrated with Gramine's . . .	Low	Fixed
V-ORC-VUL-032	DNS domain hijacking	Warning	Acknowledged
V-ORC-VUL-033	Defensive Programming Gaps	Warning	Fixed
V-ORC-VUL-034	Outdated versions for dependencies . . .	Warning	Fixed
V-ORC-VUL-035	Process-scoped mutex guards a . . .	Warning	Acknowledged
V-ORC-VUL-036	Uncanceled verification requests in . . .	Warning	Acknowledged

V-ORC-VUL-037	Repeated code in context cancellation and ...	Warning	Fixed
V-ORC-VUL-038	Debug mode is not rejected in verification ...	Warning	Fixed
V-ORC-VUL-039	Dangerous mutation of fields	Warning	Fixed
V-ORC-VUL-040	Computational integrity of weighted average	Warning	Acknowledged
V-ORC-VUL-041	Standardize SGX enclave info serialization ...	Warning	Fixed
V-ORC-VUL-042	DecodeQuoteHandler calls WriteHeader ...	Warning	Fixed
V-ORC-VUL-043	Go SDK leaks connections when response ...	Warning	Fixed
V-ORC-VUL-044	Missing Cardinality Checks for ...	Warning	Acknowledged
V-ORC-VUL-045	Missing cache-control header permits stale ...	Info	Fixed
V-ORC-VUL-046	Multiple Typos and Inconsistencies Across ...	Info	Fixed
V-ORC-VUL-047	Case-Sensitive String Comparisons in ...	Info	Fixed
V-ORC-VUL-048	HTTP success detection is limited to status ...	Info	Acknowledged
V-ORC-VUL-049	SGXReport struct field order mismatch ...	Info	Fixed
V-ORC-VUL-050	Missing rate limiting on SGX attestation ...	Info	Acknowledged
V-ORC-VUL-051	Strict Content-Type check rejects valid ...	Info	Fixed
V-ORC-VUL-052	Misconverted HTTP status codes in ...	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-ORC-VUL-001: Missing rate limits can enable oracle manipulation and DoS

Severity	Critical	Commit	N/A
Type	Denial of Service	Status	Partially Fixed
Location(s)	aleo-oracle-notarization-backend/[...]/attestation.go		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/37		

Description The service exposes the attestation endpoint without any server-side request throttling or upstream budgeting. As implemented, any external caller can rapidly issue repeated requests, causing the backend to query an exchanges' REST APIs at a pace that exceeds their limits. Additionally, query responses used to report rate limit violations are ignored. Major exchanges explicitly enforce rate limits and will return HTTP 429 (Too Many Requests) and-if the client continues-escalate to automated IP bans. For example:

1. Binance will return HTTP 418 with ban durations scaling from minutes to days when a rate limit is exceeded; they also send a Retry-After header that clients are expected to honor, and limits are enforced per IP, not per API key.
2. Coinbase similarly throttles public endpoints by IP (10 rps, bursts to 15) and returns HTTP 429 upon limit violations .

Without server-side controls, hostile or misconfigured callers can (a) exhaust the oracle's own capacity, and (b) cause upstream exchanges to throttle or ban your egress IPs, disrupting notarization services for all users.

Impact Missing rate limits can allow a malicious user to DoS the oracle by spamming the attestation service at a rate that will result in bans from the underlying exchanges. For example, repeated 429s followed by 418 bans on Binance can block your egress IPs for up to **3 days**, halting all REST data pulls that aren't already on websockets.

Additionally, such techniques could allow a malicious user to select which exchanges are queried by a price feed. To do so, they can spam specific notarization requests for data from the exchanges that they would like to disable until the notarization service returns an error. After disabling the desired exchanges they can still query the price feed as long as the minimum number of exchanges remain. At the time of auditing this appears to be 2 and so if one of these two exchanges can be influenced or is misreporting data, this could have a significant impact on the prices reported by the price feed.

Recommendations Implement rate limiting and upstream protection at three layers:

1. **Ingress rate limiting (per client/IP):** Add a token-bucket or leaky-bucket limiter at your HTTP edge (e.g., API gateway, reverse proxy, or framework middleware). Enforce per-IP and per-API-key ceilings plus short-window burst caps. Return 429 with a sane Retry-After.
2. **Upstream budgeting (per exchange):**

- ▶ Maintain **per-exchange request budgets** aligned with official docs (e.g., Coinbase 10 rps public; Binance mixed weight limits).
- ▶ **Honor backoff signals:** On 429/418, read and obey Retry-After; stop retry storms. For Binance, track X-MBX-USED-WEIGHT-*/order-count headers to stay under dynamic limits; clamp concurrency and queue excess requests.
- ▶ Prefer **websocket market data** subscriptions (ticker/order-book) and cache results to reduce REST pressure (Binance recommendation).

3. Architecture & resiliency:

- ▶ Consider separating the service that allows users to query specific exchanges from the price feed service as allowing specific queries provides attackers with more fine-grained control over the system.
- ▶ Consider raising the number of minimum exchanges so if one exchange is compromised, it will have a smaller impact on the returned price.

Developer response For now, we will restrict access to the Notarization Backend to a single client using mTLS to ensure the exchange rate limit is not exceeded. At a later stage, we will implement proper rate limiting for the exchange endpoints.

5.1.2 V-ORC-VUL-002: Integrity of timestamps

Severity	High	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-contracts/[...]/vlink_oracle_v0001.leo:459 ▶ aleo-oracle-notarization-backend/internal/services/ <ul style="list-style-type: none"> • attestation/quote_preparation.go:288 • dataextraction/price_feed.go:293 		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/2, f723d80		

Description The price oracle component enforces **monotonic timestamp updates** to prevent outdated or replayed price data. However, this mechanism implicitly trusts the local system clock. If the system time is misconfigured or tampered with, the oracle may incorrectly:

- ▶ Reject valid price updates if timestamps appear non-increasing.
- ▶ Accept stale or manipulated price data if the local clock is set backwards or forwards inappropriately.

For example, an attacker with access to the host machine (or a misconfigured deployment) could reset the clock to an earlier time. Subsequent updates from honest data sources would be rejected by the oracle, halting price feed updates and degrading the accuracy or availability of downstream components that rely on this data.

Impact A manipulated local clock can lead to:

- ▶ **Denial of Service (DoS):** Fresh, valid oracle updates are discarded, freezing the price at outdated values.
- ▶ **Incorrect price acceptance:** If the clock is fast-forwarded, future-dated updates may be accepted too early.
- ▶ **Market manipulation:** Smart contracts using this oracle (e.g., for collateral valuation or trading limits) could behave incorrectly due to inaccurate time-gating.

This introduces a central fragility that can be exploited to disrupt price-dependent logic in lending, AMMs, or liquidation systems.

Recommendation Decouple oracle time validation from local system time. Use an **authenticated, external time source**—such as a Network Time Security (NTS) pool or signed timestamps from a trusted oracle feed.

Developer response The developers followed the recommendations and fixed the issue.

5.1.3 V-ORC-VUL-003: HTTP response splitting / header injection vulnerability

Severity	High	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/internal/ ▶ constants/constants.go:4-54 ▶ services/dataextraction/data_extractor.go:94		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/4 , https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/4 , b092a7f, d4b4e46		

Description The ExtractDataFromTargetURL function allows arbitrary user-supplied HTTP headers to be injected into outbound requests. These headers are forwarded without filtering or validation, allowing attackers to introduce untrusted values into the HTTP request sent to the target server. This behavior is implemented in makeHTTPRequestToTarget, where headers are set as follows:

```

1 for key, value := range attestationRequest.RequestHeaders {
2     req.Header.Set(key, value)
3 }

```

While the application defines an AllowedHeaders list, this list is only used during response processing to mask sensitive headers-it does not restrict which headers are forwarded in outbound requests. The notarization service hashes the entire encoded request, including headers, using PrepareOracleRequestHash and PrepareOracleTimestampedRequestHash. Any change in headers-either in value or presence-alters the final hash. However, headers not on the allowlist are masked with "*****" during encoding, meaning their true values do not contribute to the notarized hash.

This behavior introduces a visibility gap. If an attacker injects a non-allowlisted header with malicious content, its influence on the outbound request remains effective, but the masked version will obscure this content in the encoded and hashed request. Thus, verifiers or clients cannot detect such tampering.

For example, consider the following CRLF injection payload:

```

1 Authorization: legitValue%0d%0aX-Forwarded-Host: https://attacker.com
2 Authorization: legitValue%0d%0aContent-Type: text/html%0d%0a%0d%0a<html><meta http-
  equiv="refresh" content="0;url=https://attacker.com"></html>

```

Since Authorization is not in the allowlist, it will be masked in the notarized data while still altering the HTTP response at the application layer. This disconnect enables undetectable response-splitting and content injection attacks.

Impact Malicious headers not on the allowlist can modify the outbound request and influence the server's response, while remaining hidden in the notarized output. This breaks the integrity guarantees of the notarization process-verifiers cannot detect that the response was shaped by attacker-controlled input. Specifically, attackers can inject hidden redirects. Because these

headers are masked in the notarized data, downstream consumers cannot verify the authenticity or trustworthiness of the response content.

Recommendation

- ▶ Restrict outbound header forwarding to a validated `OutboundAllowedHeaders`. Only headers explicitly listed should be included in outbound
- ▶ Enforce strict validation of header keys and values at the application layer, even if the transport layer performs sanitization. This includes rejecting any headers containing newline or carriage return characters before request construction. This ensures that headers used for application logic or response handling cannot be manipulated in a way that is invisible to verifiers.
- ▶ Develop test coverage to detect header injection vulnerability.

Developer response The developers followed the recommendations and fixed the issue.

5.1.4 V-ORC-VUL-004: Outlier-insensitive VWAP propagates exchange errors

Severity	High	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/internal/[...]/price_feed.go		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/39 , https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/39 , 14705eb, 6cd7b3e		

Description When a user requests attestation to a price feed, the notarization service aggregates prices from multiple exchanges using the CalculateVolumeWeightedAverage function shown below which computes a volume weighted average price.

```

1 func CalculateVolumeWeightedAverage(prices []ExchangePrice) (float64, float64, int)
2 {
3     if len(prices) == 0 {
4         return 0, 0, 0
5     }
6
7     var totalVolume float64
8     var weightedSum float64
9     exchanges := make(map[string]bool)
10
11     for _, price := range prices {
12         if price.Price > 0 && price.Volume > 0 {
13             weightedSum += price.Price * price.Volume
14             totalVolume += price.Volume
15             if _, exists := exchanges[price.Exchange]; !exists {
16                 exchanges[price.Exchange] = true
17             }
18         }
19     }
20
21     if totalVolume == 0 {
22         return 0, 0, 0
23     }
24
25     volumeWeightedAvg := weightedSum / totalVolume
26     return volumeWeightedAvg, totalVolume, len(exchanges)
27 }
```

After computing the average price, the number of queried exchanges is checked to ensure that a minimum number of exchanges participated in the average computation (at the time of auditing, the minimum is 2). Otherwise, no additional outlier filtering, or dispersion checks are performed to determine if the resulting price had an outsized influence by a single exchange. As a result, an erroneous or adversarial quote—especially one paired with a large (or misreported) volume—can dominate the VWAP and be accepted so long as it doesn't drop the exchange count below the required minimum number of exchanges.

This is the same failure mode seen in the 2019 Synthetix oracle incident: one upstream API intermittently reported a KRW price 1000* too high; because only **two** feeds were live for KRW at the time, the oracle **averaged the two** and propagated the bad price on-chain. Synthetix's post-mortem explicitly notes that outlier protection failed because too few independent feeds were available, causing the average to accept the bad quote.

Impact

- ▶ **Skewed or manipulated prices:** A single faulty/malicious venue can drag the VWAP far from fair value, particularly if it reports outsized volume, leading to incorrect prices for downstream consumers (quoting, risk, liquidations).
- ▶ **Quorum fragility:** During partial outages, a small number of required exchanges allows a bad feed to outweigh good ones (exactly as in the Synthetix case), converting transient upstream errors into protocol-level failures

Recommendation

- ▶ **Pre-aggregation outlier filtering:** Compute a robust center (e.g., median) and remove or winsorize quotes beyond tight bands (e.g., median +/- k·MAD or an IQR rule) before any averaging.
- ▶ **Weight guards:** Cap per-exchange influence (e.g., <=40-50% of total weight) and sanity-check cross-venue dispersion (max/min ratio, %-deviation from median) with alerts. Note that if exchange influence is capped, also consider filtering out exchanges with unreasonably small volumes.
- ▶ **Testing & telemetry:** Add unit tests that inject a 50-100* outlier (with large volume) and assert the final price is unchanged within a tight tolerance; monitor live dispersion and reject updates when dispersion exceeds thresholds.

Developer response The developers followed the recommendations and fixed the issue.

5.1.5 V-ORC-VUL-005: SGX Attestation Does Not Cover Aleo Address or Signature

Severity	High	Commit	N/A
Type	Cryptographic Vulnerability	Status	Acknowledged
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-notarization-backend/internal/ <ul style="list-style-type: none"> • aleoutil/aleo_util.go:45-70 • services/attestation/ <ul style="list-style-type: none"> * oracle.go:64-74, 103-119, 221-246 * quote_preparation.go:242-289 • sgx/quote.go:61-109 ▶ oracle-verification-backend/ <ul style="list-style-type: none"> • api/handlers/verify.go:62-78, 108-129, 116-125 • attestation/ <ul style="list-style-type: none"> * attestation.go:85-116 * sgx/sgx.go:12-25 		
Confirmed Fix At	N/A		

Description The Aleo key lifecycle design relies on an SGX enclave to generate and use a private Aleo key internally. The enclave outputs AttestationResponse which includes OracleData.

```

1 quotePrepData, err := attestation.PrepareDataForQuoteGeneration(extractDataResult.
    StatusCode, extractDataResult.AttestationData, uint64(timestamp),
    attestationRequest)
2 ...
3 quote, err := sgx.GenerateQuote(quotePrepData.AttestationHash)
4 ...
5 oracleData, err := attestation.BuildCompleteOracleData(quotePrepData, quote)
6 ...
7 response := &attestation.AttestationResponse{
8     ...
9     AttestationReport:    base64.StdEncoding.EncodeToString(quote),
10    OracleData:            *oracleData,
11 }

```

The oracleData contain attested quote together with aleo signature and aleo public key (OracleData.Address).

```

1 oracleReport, err := PrepareOracleReport(quote)
2 ...
3 oracleData := &OracleData{
4     UserData:            string(quotePrepData.UserData),
5     EncodedPositions:    *quotePrepData.EncodedPositions,
6     EncodedRequest:      string(encodedRequest),
7     RequestHash:         requestHashString,
8     TimestampedRequestHash: timestampedHash,
9     Report:              string(oracleReport),
10    Signature:            signature,
11    Address:              aleoContext.GetPublicKey(),
12 }

```

The `fieldOracleData.Address` is retrieved from `AleoContext.GetPublicKey()` but it can be tampered with *after* receiving the enclave's SGX quote and report, meaning this address is not authenticated or cryptographically bound to the attestation. Nothing prevents an attacker from combining:

- ▶ A valid SGX quote from one enclave (proving some genuine enclave was used)
- ▶ A signature and Aleo address from a *different* keypair (generated in debugged enclave or outside SGX)

Because the SGX quote does not cover or hash the Aleo address, and the on-chain verifier checks only the Aleo signature-not the SGX quote-there is no binding between the attested enclave and the signing key.

This creates an implicit trust link that is not enforced by cryptographic evidence.

Example scenario:

1. Attacker runs enclave A to get a valid SGX quote (with correct MRENCLAVE).
2. Attacker separately generates Aleo key B and signs data with it.
3. Attacker constructs `OracleData` using:
 - ▶ SGX quote from enclave A
 - ▶ Aleo address and signature from key B

Because the Aleo signature matches key B, the signature verification passes. But the SGX quote has no cryptographic tie to key B. If verifiers whitelist key B based on the quote from enclave A, they are accepting data signed by a key not provably tied to the attested enclave.

Impact This enables key-substitution and impersonation attacks. An attacker can produce seemingly valid oracle data using any Aleo key, bypassing the assumption that the data was signed by the specific enclave verified in the SGX quote.

Since downstream systems rely solely on Aleo signature checks on-chain, the lack of cryptographic linkage between the quote and the address allows attackers to substitute keys and redirect trust. The off-chain verification backend does not enforce non-debug mode, which additionally increases the risk of whitelisting a key from enclave running in debug mode.

Recommendation

1. Bind the public key to the quote: embed a hash of the public key in `ReportData` before generating the quote.
2. Bind messages to the attestation and protect against later mix-and-match attacks or replayed quotes:
 - ▶ compute `M = H("ALEO/POP/v1" || quote_hash || aleo_address || mrenclave || debug || timestamp [|| payload_hash])`
 - ▶ `sig_aleo = SchnorrSign(sk_aleo, M)`
 - ▶ Include `sig_aleo` alongside the SGX quote and Aleo address in oracle responses.

Developer response We will be fixing it in the future releases.

5.1.6 V-ORC-VUL-006: Long term exposure of Aleo private key

Severity	Medium	Commit	N/A
Type	Cryptographic Vulnerability	Status	Fixed
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-notarization-backend/[...]/aleo_util.go:20, 60 ▶ aleo-utils-go/session.go:25, 89-98, 470 		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-utils-go/pull/2		

Description The AleoContext struct stores a long-lived Aleo private key as a Go string:

```

1 type AleoContext struct {
2     Session    aleoUtils.Session // The Aleo session.
3     privateKey string           // The Aleo private key.
4     PublicKey  string           // The Aleo public key.
5     Close      func()           // The Aleo close function.
6 }
```

Although all memory is SGX-protected through Gramine-meaning it is hardware-encrypted and shielded from the host-this protection only applies to **out-of-enclave threats**. **In-enclave threats**, such as side-channel remain relevant and are exacerbated by unsafe in-memory key handling.

Converting the generated private key from []byte to string (`key := string(privKey)`) **duplicates the secret into an immutable, garbage-collected buffer that cannot be wiped**. The key is then re-copied into WASM memory for signing:

```

1 s.mod.Memory().Write(uint32(privateKeyPtr[0]), []byte(key))
```

This call creates another copy of the key in the WebAssembly module's heap. **WASM memory is outside Go's control, so wiping must be manual**. Unless explicitly overwritten, this memory persists until deallocation, leaving residual plaintext in enclave memory.

While the key never leaves the SGX enclave, persistent plaintext buffers can still be targets for **in-enclave attacks**-including cache timing, speculative execution - due to their extended lifetime and unmanaged lifecycle. **The presence of multiple unmanaged copies significantly increases the surface area and temporal window available for side-channel exploitation**, making it easier for an attacker to locate and extract sensitive material within enclave memory.

Impact Storing long-lived keys as Go strings and copying them into unmanaged WASM memory increases in-enclave exposure by:

- ▶ Preventing secure zeroization (Go strings are immutable)
- ▶ Leaving residual plaintext in multiple heap locations (Go + WASM)
- ▶ Extending the memory lifetime of sensitive data beyond intended use
- ▶ Increasing the likelihood of successful side-channel attacks or faults within the enclave due to redundant unmanaged key copies

Recommendation

- ▶ Avoid storing key material as `string`; retain it as `[]byte` in a memory-zeroable container
- ▶ Immediately overwrite WASM memory used for key operations after use
- ▶ Refactor `AleoContext` to avoid long-lived storage of secret keys-pass ephemeral buffers to signing routines only
- ▶ Treat enclave memory as hostile from a lifecycle perspective: enforce explicit teardown of secrets
- ▶ Flush CPU caches at key boundaries using architecture-specific instruction `CLFLUSH` to reduce side-channel residue.
- ▶ Wrap sensitive buffers in secure containers like `memguard` to allocate protected, lockable memory regions

Developer response The developers implemented the recommendation given in the issue.

5.1.7 V-ORC-VUL-007: Insecure PCCS configuration undermines SGX quote verification

Severity	Medium	Commit	N/A
Type	SGX attestation	Status	Fixed
Location(s)	oracle-verification-backend/ ▶ api/handlers/decode_quote.go:39 ▶ attestation/sgx/sgx.go:13 ▶ pccs/setup.sh:11, 40 ▶ sgx_default_qcnl.conf:8		
Confirmed Fix At	https://github.com/venture23-aleo/oracle-verification-backend/pull/10 , d1f7898		

Description The setup script (setup.sh) for the Platform Certificate Caching Service (PCCS) grants world-readable permissions (chmod 644) to the private key generated for HTTPS communication. Additionally, PCCS is configured to listen on all network interfaces (0.0.0.0) by default. The associated configuration file (sgx_default_qcnl.conf) disables TLS certificate verification by setting use_secure_cert: false.

Both SGX quote verification code paths -DecodeQuoteHandler and VerifySgxReport- call `verifyRemoteReport` function depend on this insecure PCCS setup to fetch Intel-provided attestation collateral. The function `verifyRemoteReport` performs a call to `oe_verify_evidence` which reports **"not up-to-date"** (returns `OE_TCB_LEVEL_INVALID`) when the Intel SGX **collateral** (fetched via PCCS or Intel's service) says the platform's current **TCB level** isn't acceptable.

Because TLS verification is disabled and the private key is readable by any user on the system, an attacker on the same host or network can:

- ▶ Intercept and spoof PCCS responses by performing a man-in-the-middle (MitM) attack.
- ▶ Read the private key and impersonate the PCCS server or decrypt intercepted HTTPS traffic.

Impact A network or local attacker can inject forged attestation collateral or tamper with responses from PCCS.

This allows them to trick EGo verifier into accepting fake or revoked quotes, which breaks the authenticity and integrity guarantees of SGX attestation collateral, undermining the trust model of SGX quote verification.

Recommendation

- ▶ Change private key permissions to be accessible only by the PCCS service user (e.g., `chmod 600`, owned by `pccs:pccs`).
- ▶ Bind PCCS to localhost or a specific internal interface, not 0.0.0.0.
- ▶ Set `use_secure_cert: true` in `sgx_default_qcnl.conf` to enforce TLS certificate validation.
- ▶ Consider deploying PCCS behind a reverse proxy with strict mutual TLS policies and hardened key management.

Developer response The developers implemented the recommendations given in the issue.

5.1.8 V-ORC-VUL-008: Handling of SWHardeningNeeded status

Severity	Medium	Commit	N/A
Type	SGX attestation	Status	Fixed
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-notarization-backend/internal/sgx/sgx.go:13 ▶ oracle-verification-backend/api/[...]/decode_quote.go:39 		
Confirmed Fix At	https://github.com/venture23-aleo/oracle-verification-backend/pull/29,6aa090c		

Description Intel's SGX platform may return the TCB status `SWHardeningNeeded` during remote attestation, indicating that microcode mitigations for certain transient execution vulnerabilities or speculative side-channels are insufficient. Instead, the enclave software must implement specific hardening techniques (such as `LFENCE`, `retpoline`, or the Intel LVI mitigation toolchain) to ensure security.

This is distinct from a status of `UpToDate`, which guarantees both microcode and software-level security. OpenEnclave (OE) and EGo treat `SWHardeningNeeded` as a **valid TCB status**. However, this shifts the burden of verifying whether mitigations are implemented onto the relying party or operator.

In Go-based systems using EGo, the relevant status appears as `report.TCBStatus == tcbstatus.SWHardeningNeeded`. If this is the case, the relying party must inspect `report.TCBAdvisories` to check for the list of Intel SA advisories and ensure all necessary mitigations are implemented and allowed by policy.

Impact Accepting attestation reports with `SWHardeningNeeded` without verifying that software mitigations are actually implemented may lead to enclaves being falsely treated as secure despite being vulnerable to known microarchitectural attacks. This opens the door for confidentiality breaches or code execution attacks within the enclave.

Recommendation When handling attestation reports where `TCBStatus == SWHardeningNeeded`:

- ▶ Always validate that `report.TCBAdvisories` are present and match an explicitly allowlisted set of advisories.
- ▶ Ensure the enclave binary was built using the required mitigations (e.g., compiler flags or mitigation toolchains) for all advisories in the report.
- ▶ Ensure kernel microcode implements required mitigations and ensure they are enabled at system startup.
- ▶ Only accept such attestation reports if all criteria above are satisfied. Otherwise, treat the enclave as insecure and reject the attestation.

Developer response The developers implemented recommendations and track TCB advisories to accept only those allowed by internal policy.

5.1.9 V-ORC-VUL-009: Dropped exchange information risks price accuracy

Severity	Medium	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/[...]/exchange_parser.go		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/32 , 5664a67		

Description Across all exchange adapters, two pieces of information are dropped that most exchanges return and that are critical for correctness:

1. **Returned symbol/product ID** - needed to verify the payload actually corresponds to the requested market (guards against exchange-side errors and local misconfiguration).
2. **Server-provided timestamp** - needed to detect and reject stale data before aggregation.

For example, Binance's 24-hour ticker endpoint returns both a symbol and a time corresponding to the latest price measurement which can be used to validate the market and assess staleness. Similarly, Coinbase Exchange's REST ticker for a given pair returns a time field in the response, providing a precise timestamp for the quoted price.

Because the current structs do not retain these fields, the system cannot (a) assert that the response matches the intended trading pair, or (b) enforce freshness thresholds. This is especially important when parsing responses from exchanges that can return multiple prices such as the Bybit response parser shown below as currently it is assumed that the first response corresponds to the requested price. This makes it easier for exchange hiccups, misrouted results, misconfiguration, or clock drift to propagate into the volume-weighted average.

```

1 func parseBybitResponse(data []byte) (price, volume float64, err *appErrors.AppError)
2 {
3     var bybitResponse BybitResponse
4     if err := json.Unmarshal(data, &bybitResponse); err != nil {
5         logger.Error("Error unmarshalling data: ", "exchange", "bybit", "error", err)
6         return 0, 0, appErrors.ErrDecodingExchangeResponse
7     }
8
9     list := bybitResponse.Result.List
10    if len(list) == 0 {
11        logger.Error("No data in response", "exchange", "bybit")
12        return 0, 0, appErrors.ErrMissingDataInResponse
13    }
14
15    item := list[0]
16
17    price, parseErr := strconv.ParseFloat(item.Price, 64)
18    if parseErr != nil {
19        logger.Error("Error parsing price: ", "exchange", "bybit", "error", parseErr)
20        return 0, 0, appErrors.ErrParsingPrice
21    }
22
23    volume, parseErr = strconv.ParseFloat(item.Volume, 64)
24    if parseErr != nil {

```

```
24 |         logger.Error("Error parsing volume: ", "exchange", "bybit", "error", parseErr
25 |     )
26 |     return 0, 0, appErrors.ErrParsingVolume
27 | }
28 | return price, volume, nil
29 | }
```

Impact

- ▶ **Mismatched market acceptance:** If an exchange returns a list or an incorrect market (e.g., due to misconfiguration or upstream bug), the aggregator may accept and weight a price for the wrong symbol because the returned symbol/product_id is not verified.
- ▶ **Stale data in aggregation:** Without using exchange timestamps, prices that are minutes or hours old can be included, skewing VWAP and increasing the chance of propagating bad data during incidents.

Recommendations Extract the symbol and timestamp from an exchange response when available and perform the following validation:

1. Ensure that the response corresponds to the requested symbol or product ID.
2. Ensure freshness by converting exchange timestamps to a unified clock and dropping old responses.

Developer response The developers implemented the recommended fix. Note that the comparison timestamp uses local machine time but this should be change in the "integrity of timestamps" fix..

5.1.10 V-ORC-VUL-010: Missing Enclave Key Rotation and Attestation Enforcement Enables Long-Term Key Misuse

Severity	Medium	Commit	N/A
Type	Cryptographic Vulnerability	Status	Acknowledged
Location(s)	aleo-oracle-notarization-backend/internal/[...]/aleo_util.go		
Confirmed Fix At	N/A		

Description The protocol currently lacks enforced enclave key rotation and binding of keys to trusted attestation, enabling potential misuse of long-lived keys without detection.

Enclave keys (used for signing oracle responses) are not associated with metadata such as `created_at`, `expires_at`, or `previous_key_signature`. There is no enforced rotation policy or validation of a key's freshness through attestation. The absence of expiry and rotation mechanisms allows a key generated years ago to remain valid indefinitely—even if the underlying SGX enclave becomes vulnerable over time.

In trusted execution environments like Intel SGX, long-lived enclaves represent a growing risk. As new vulnerabilities in SGX hardware, microcode, or firmware are discovered (e.g., Foreshadow, Plundervolt), enclaves compiled and provisioned under older threat models may become exploitable. Without mandatory key rotation or attestation renewal, an attacker who compromises an outdated enclave can continue signing data with an old key that is still accepted by the smart contract.

Additionally, if enclave public keys are whitelisted manually (e.g., via governance vote or static registry), this introduces a governance-based trust model that bypasses technical attestation guarantees. A malicious or misconfigured enclave could be whitelisted without producing an SGX quote, or an attacker could trick governance into re-whitelisting a previously leaked key. This undermines the integrity of SGX-based trust, as manually whitelisted keys may not correspond to authentic or uncompromised enclaves.

Impact This enables long-term misuse of stale or compromised enclave keys. If an enclave is compromised due to newly discovered SGX vulnerabilities, its associated key can still sign values indefinitely unless a rotation or attestation check is enforced.

Manual whitelisting further weakens the security model by allowing keys to bypass attestation entirely. This introduces trust dependencies on governance or operator diligence, rather than verifiable hardware-backed guarantees.

Consequences include corrupted oracle data, manipulated financial positions, or theft of funds from downstream protocols relying on the compromised input.

Recommendation To mitigate the risks of long-lived enclave keys, manual whitelisting, and evolving SGX vulnerabilities, adopt a lifecycle management approach aligned with **NIST Special Publication 800-57: Recommendation for Key Management**:

1. Enforce key lifetimes (NIST Key Usage Periods)

- ▶ Define a maximum validity window for each enclave key (`expires_at`), consistent with NIST's and Intel SGX recommended cryptoperiods (e.g., 30 days).
- ▶ Enforce expiration within the enclave and on-chain, denying signing after `expires_at` and remove the key from whitelist.

2. Implement automated key rotation (NIST Key Establishment & Transition)

- ▶ Add a `rotate_keys()` to generate a successor key, sign it with the current key, and update lineage metadata (`previous_key_signature`).
- ▶ Trigger rotation automatically before expiry.
- ▶ Separate keys for data signing and key rotation to align with NIST SP 800-57 principles, even though both keys require attestation binding and expiry enforcement. During bootstrapping, whitelist only the key rotation key - hence, data signing key will be bound to fresh SGX attestation.

3. Bind keys to trusted attestation (NIST Key Authenticity Requirements)

- ▶ Require off-chain verification of enclave measurement (`MRENCLAVE`) and store a hash of the attestation alongside the key on-chain.

4. Contract-level enforcement (NIST Key Registration & Usage Control)

- ▶ Replace manual whitelisting with `rotateKey(new_key, signature, expiry, attestation_hash)`.
- ▶ Validate that:
 - `new_key` is signed by `old_key`
 - `expiry` is within acceptable NIST cryptoperiod limits
 - `attestation_hash` matches a valid enclave quote

5. Audit and monitoring (NIST Key Accountability)

- ▶ Log every signing operation with key ID and timestamp.
- ▶ Alert if a key is used post-expiry, without a valid attestation, anomalous frequency of outlier data.

Developer response We will be fixing it in the future releases.

5.1.11 V-ORC-VUL-011: Missing quote verification in notarization backend

Severity	Medium	Commit	N/A
Type	Data Validation	Status	Partially Fixed
Location(s)	aleo-oracle-notarization-backend/ ▶ deployment/native/[...]/generate-manifest-template.sh:59 ▶ internal/api/handler/attestation.go		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/6,04afc9d		

Description The notarization backend generates SGX quotes and constructs signed OracleData responses without validating their authenticity. In the GenerateAttestationReport handler, the backend calls `sgx.GenerateQuote()` and immediately passes the output into `attestation.BuildCompleteOracleData()`. No intermediate verification ensures the quote was produced by a genuine SGX-capable CPU. Instead, `BuildCompleteOracleData()` simply formats the input, appends the Aleo address, and signs it - effectively trusting host-controlled input.

The backend also continues to start and serve requests even when SGX support is missing or misconfigured. If `/dev/attestation/quote` is absent, unreadable, or invalid, the backend does not fail fast. It continues running, only returning errors at attestation request time. This "lazy failure" model permits indefinite operation in an untrusted or non-SGX environment.

Missing safeguards include:

- ▶ No validation of report/quote length or structure (truncated or malformed data accepted).
- ▶ Raw device file access (`os.ReadFile`, `os.WriteFile`) without protection against symlink or device substitution.
- ▶ Blind trust in `/dev/attestation/quote` contents, rejecting only empty files.
- ▶ Manifest specifies `remote_attestation = "dcap"`, but DCAP collateral and signature verification are not performed.

Example:

```

1 quote := sgx.GenerateQuote() // Reads unvalidated data from /dev
  /attestation
2 oracleData := attestation.BuildCompleteOracleData(quote) // Signs and returns it
  without verifying source

```

Impact A malicious or compromised host OS can inject arbitrary SGX quotes, which the backend will sign as if they were genuine. This enables attackers to forge attestations, impersonate enclaves, and register arbitrary Aleo addresses under the guise of SGX-backed trust.

Because the backend runs without enforcing SGX availability, it can provide meaningless or spoofable attestation services. Reliance on unguarded file operations also enables path substitution attacks (e.g., symlinking `/dev/attestation/quote` to attacker-controlled data).

Recommendation To strengthen the attestation backend and eliminate the risks described, we recommend implementing the following verifications:

1. Startup Enforcement

- ▶ Fail if SGX device files are missing, unreadable, or contain invalid data.
- ▶ Do not continue running in non-SGX environments or debug mode for production deployment.

2. Quote Verification (DCAP)

- ▶ Ensure `/dev/attestation/attestation_type` contains `dcap`
- ▶ After generating a quote, validate its structure, freshness, and authenticity.
- ▶ Perform Intel DCAP collateral and signature verification before packaging the quote in `OracleData`.
- ▶ Intel DCAP Quick Install Guide

3. CPU Feature Flags (hardware proofs)

Validate that the host CPU advertises the required mitigation flags via `/proc/cpuinfo` or CPUID MSRs:

- ▶ `ibrs`, `ibpb`, `stibp` (Spectre v2 defenses) [2]
- ▶ `ssbd` (Speculative Store Bypass Disable) [2]
- ▶ `md_clear` (MDS buffer clearing) [3]
- ▶ `flush_l1d` (L1D cache flush support) [4]
- ▶ `arch_capabilities` MSR bits (e.g. `MDS_NO`, `TAA_NO`, `SSB_NO`) [5]

4. Kernel Vulnerability Status (`/sys/devices/system/cpu/vulnerabilities/*`)

Require all vulnerability status files to report an active mitigation, not "Vulnerable":

- ▶ `spectre_v2` -> "Mitigation: IBRS/IBPB" or "Mitigation: Retpolines" [6]
- ▶ `l1tf` -> "Mitigation: PTE Inversion" or "Mitigation: flush" [4]
- ▶ `mds` -> "Mitigation: Clear CPU buffers" [7]
- ▶ `tsx_async_abort` -> "Mitigation: TSX disabled" or equivalent [3]

5. Kernel Command-Line Flags (`/proc/cmdline`)

Enforce boot parameters that enable hardening:

- ▶ `spectre_v2=on` or `spectre_v2=retpoline,ibrs` [6]
- ▶ `l1tf=full,flush` [4]
- ▶ `mds=full` [7]
- ▶ `tsx=off` or `tsx_async_abort=full` [3]
- ▶ `spec_store_bypass_disable=on` [8]
- ▶ `nosmt` or `stibp=always-on` [4]

1. Intel DCAP Quick Install Guide

2. Intel Speculative Execution Side Channel Mitigations Whitepaper

3. Linux TAA (TSX Async Abort) Mitigation Documentation

4. Linux L1TF Mitigation Documentation

5. Intel 64 Architecture Processor Topology & MSRs (ARCH_CAPABILITIES etc.)

6. Linux Spectre Mitigation Documentation

7. Linux MDS (Microarchitectural Data Sampling) Mitigation Documentation

8. Linux Speculative Store Bypass Mitigation Documentation

Developer response The SDK we're using performs both notarization and verification within a single attestation request. Since this process is encapsulated by the SDK, we won't be performing separate validation of the platform configuration inside the enclave. The SDK's built-in verification ensures that we will know whether the enclave-generated quote is valid or not.

5.1.12 V-ORC-VUL-012: Timing Variations in Scalar Multiplication of Private Key

Severity	Medium	Commit	N/A
Type	Cryptographic Vulnerability	Status	Fixed
Location(s)	aleo-utils-go/ ▶ Cargo.toml:1-14 ▶ src/sign.rs:80-101		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/43		

Description The `PrivateKey::sign` method in Aleo's `snarkVM` library (at `console/account/src/signature/sign.rs`) implements a Schnorr-style signature scheme. A key part of this process involves computing the scalar multiplication of a secret scalar (either the ephemeral nonce or the private key scalar) with the curve's generator point. This is implemented via `Network::g_scalar_multiply()` ([sign](#), [g_scalar_multiply](#)) using a non-constant-time algorithm:

```

1 GENERATOR_G
2   .iter()
3   .zip_eq(&scalar.to_bits_le())
4   .filter_map(|(base, bit)| match bit {
5       true => Some(base),
6       false => None,
7   })
8   .sum()

```

This uses the scalar's bits to conditionally include base points in a sum based on their bit value. The number of additions (and which are executed) depends directly on the bit pattern of the secret scalar.

Impact This implementation introduces secret-dependent branching and variable execution time, which violates constant-time principles. An attacker with access to fine-grained timing or microarchitectural leakage (e.g. cache-timing or branch-prediction side channels) could:

- ▶ Infer the Hamming weight of the scalar (number of 1-bits)
- ▶ Learn partial information about individual bits through branching behavior
- ▶ Potentially extract the private signing key over multiple observations

This compromises the security of Aleo's Schnorr signatures, especially in scenarios where signatures are produced on demand and the environment is shared with untrusted code.

Recommendation

- ▶ As a temporary solution, automate key management with periodic key rotation to limit the window of opportunity for a potential attacker. Enforce key rotation after 10^3 - 10^4 signatures to ensure the attacker cannot collect enough traces from side-channel analysis.
- ▶ Replace the scalar multiplication with a constant-time algorithm.

- ▶ Use of a Montgomery ladder or fixed-window scalar multiplication, ensuring the same sequence of operations regardless of secret scalar bits
- ▶ Avoid branching on secret data or using constructs like `filter_map` that conditionally execute logic
- ▶ Use constant-time conditional select functions to always process each base point, regardless of bit value
- ▶ If a constant time version of the algorithm cannot be used, insert a random delay before serving results back to the user. Note that this increases the difficulty of measuring remote timing channels but does not prevent on-device channels.

Developer response The developers implemented the recommendation to add a random delay to the computation.

5.1.13 V-ORC-VUL-013: GetPriceFeed May Hang on Goroutine Failure

Severity	Medium	Commit	N/A
Type	Logic Error	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/[...]/ price_feed.go:249-272		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/34 , 8a9effe		

Description GetPriceFeed reads from the results channel in a loop that assumes every spawned goroutine will send exactly one fetchResult. It uses:

```

1 for i := 0; i < totalTradingPairs; i++ {
2     result := <-results
3     // ...
4 }
```

This fixed-count receive loop blocks until it has read totalTradingPairs items. If any goroutine exits early (due to a network error, configuration mismatch, panic, or context cancellation) and never sends to results, the loop waits indefinitely on <-results. Consequently:

- ▶ The function never returns, effectively deadlocking the caller.
- ▶ exchangePrices remains partially filled, or empty, because the loop never completes.
- ▶ Goroutines that finished successfully may be wasted, while the calling service stalls, consuming resources.

Impact

- ▶ **Deadlock:** The main function stalls indefinitely if even one goroutine fails to send a result.
- ▶ **Resource exhaustion:** Blocking calls accumulate, degrading service performance under load.

Recommendation Redesign result collection to avoid assuming a fixed number of responses. Options include:

- ▶ Use a sync.WaitGroup to track completion and close the results channel when all goroutines are done. Loop over range results instead of fixed i < N.
- ▶ Wrap each goroutine with a defer recover() block to prevent unhandled panics from terminating the routine silently.
- ▶ Enforce timeout logic to break the wait in case of missing results.

Developer response The developers implemented the recommended fix.

5.1.14 V-ORC-VUL-014: LVI mitigations

Severity	Medium	Commit	N/A
Type	Transient Execution Injection	Status	Acknowledged
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-notarization-backend/[...]/aleo-oracle-notarization-backend.manifest.template ▶ aleo-utils-go/wrapper.go:75-76 		
Confirmed Fix At	N/A		

Description A Pure Go application is running inside a Gramine SGX enclave, with the Aleo signing algorithm executed through the Wazero WebAssembly runtime:

```

1 func NewWrapper() {
2     ...
3     runtimeConfig := wazero.NewRuntimeConfigCompiler()
4     runtime := wazero.NewRuntimeWithConfig(context.Background(), runtimeConfig)
5     ...
6 }
7
8 func newAleoContext() {
9     wrapper, closeFn, _ := aleoUtils.NewWrapper()
10    ...
11    s, _ := wrapper.NewSession()
12    ...
13    privKey, address, _ := s.NewPrivateKey()
14    ...
15 }
16
17 s.Sign(privKey, message)

```

This configuration enables Wazero's compiler backend, which performs ahead-of-time (AOT) compilation at runtime. That is, when a WebAssembly module is instantiated, Wazero compiles it to native machine code and then executes it - offering higher performance than interpretation. Although this is often described as AOT, the compilation occurs within the enclave at runtime, and the resulting native code is placed in memory regions mapped with `PROT_EXEC`.

Importantly, Wazero does enforce W^X (write XOR execute) - memory is writable *or* executable, but never both at the same time - which prevents traditional RWX mappings.

However, this still presents a security risk in SGX enclaves. Because Wazero generates executable code dynamically, it expands the attack surface for transient execution vulnerabilities, especially Load Value Injection (LVI). LVI attacks allow an untrusted OS or hypervisor to inject data into faulting loads within the enclave and observe speculative execution behavior. The presence of dynamically generated executable code increases the likelihood of exploitable gadget sequences being available to attackers.

- ▶ Wazero's compiler is not hardened against Load Value Injection (LVI), a class of transient execution attacks.
- ▶ Go's native toolchain provides no support for LVI mitigations.

- ▶ Gramine's default configuration may permit executable memory mappings (e.g., via `mmap`), which can be exploited by speculative execution attacks in the presence of JIT-compiled code.

LVI enables an untrusted OS or hypervisor to inject malicious values into faulting memory loads in the enclave. Combined with the dynamic code generation of Wazero's JIT, this increases the enclave's attack surface and creates a viable vector for speculative data exfiltration, including private keys used in Aleo signing.

Impact An attacker with OS-level privileges could exploit LVI in combination with speculative execution side-channels to extract the Aleo private key or other enclave secrets. This risk is heightened by the presence of dynamically generated executable code pages, which expand the available gadget space for constructing speculative execution chains.

Specifically:

- ▶ Secrets used in the WASM module (e.g., signing keys) can be leaked by injecting values into faulting enclave loads.
- ▶ The use of Wazero's JIT compiler enables code reuse or code injection gadgets via executable memory mappings (RWX), facilitating speculative exploitation.
- ▶ The lack of compiler-based LVI mitigations in Go or Wazero increases exposure to these attack classes.

Recommendation

1. Use LVI-Resistant Hardware (Ice Lake or newer): Prefer deploying to SGX-enabled CPUs that include hardware mitigations for LVI. Ice Lake and later platforms mitigate LVI at the silicon level, significantly [reducing attack surface](#).

2. Enforce W^X Memory Policy in Gramine: Set Gramine manifest to prevent mapping pages as both writable and executable. This blocks Wazero's AOT JIT engine from generating runtime code and eliminates dynamic executable memory.

To support this, switch to Wazero's interpreter backend:

```
1 r := wazero.NewRuntimeWithConfig(ctx, wazero.NewRuntimeConfigInterpreter())
```

3. Use LLVM-Based Runtimes With LVI Mitigations: If interpreter performance is insufficient, consider replacing Wazero with an LLVM-backed WASM runtime such as WAMR or WasmEdge, which allow AOT compilation with compiler-based mitigations (e.g., `-mlvi-cfi`, `-mlvi-hardening`, LTO, stack protectors).

4. Use TinyGo + LLVM for WASM Generation: For WASM binaries compiled from Go, use TinyGo with LLVM as the backend. This enables integration with compiler-level LVI mitigations unavailable in the standard Go toolchain.

Developer response We are running the notarization backend on an Azure VM from the DCsv3 series, which is equipped with Intel(R) Xeon(R) Platinum 8370C (Ice Lake) processors.

5.1.15 V-ORC-VUL-015: Contract assumes fixed-length attestation data leaving string attestation data unsupported

Severity	Medium	Commit	N/A
Type	Data Validation	Status	Acknowledged
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-contracts/[...]/vlink_oracle_v0001.leo:220-254 ▶ aleo-oracle-notarization-backend/[...]/encoding.go:345 		
Confirmed Fix At	N/A		

Description The backend and smart contract components of the protocol generate different Poseidon hashes for the same input due to inconsistent canonicalization of the attestation field. Specifically:

- ▶ The **backend** zeroes out the entire attestation segment (based on its actual length from metadata) and the timestamp block before hashing the buffer.
- ▶ The **contract**, however, zeroes only the first 16 bytes of attestation (f2) and the timestamp field (f3), leaving any additional attestation blocks untouched.

This inconsistency occurs in `PrepareEncodedRequestProof` on the backend and `get_request_hash` on the smart contract. If the attestation string exceeds 16 bytes (e.g., user-provided data nearing the 3KB limit), the backend's hash and the contract's hash will differ. The contract uses the hash to validate the integrity of the user data; thus, a mismatch causes false rejection of valid proofs.

Impact This bug results in **false negatives** during proof verification. Any proof involving an attestation longer than 16 bytes will be deterministically rejected by the contract despite being valid according to backend logic. This can prevent users from successfully submitting valid attestations, effectively breaking protocol functionality in production.

Recommendation Update the contract to read the attestation length from metadata and zero the full span of the attestation region before hashing.

Developer response We will only supports price feed in the aleo program which is based on the float encoding.

5.1.16 V-ORC-VUL-016: Unbounded request bodies in verification APIs enable memory exhaustion

Severity	Medium	Commit	N/A
Type	Denial of Service	Status	Fixed
Location(s)	oracle-verification-backend/api/handlers/verify.go:77		
Confirmed Fix At	https://github.com/venture23-aleo/oracle-verification-backend/pull/20 , 0cc4d7d		

Description Several HTTP handlers in the verification API-verifyHandler.ServeHTTP, DecodeProofDataHandler, and DecodeQuoteHandler-read the full body of the incoming request using io.ReadAll without any size restrictions.

```
1 body, err := io.ReadAll(req.Body)
2
```

This pattern allows clients to send arbitrarily large payloads, which are buffered into memory before any validation or decoding occurs. Since the body is read entirely into memory, the memory usage scales linearly with the request size and is not constrained by HTTP-level protections.

A malicious client can exploit this by issuing oversized HTTP POST requests or many concurrent requests with large bodies, exhausting memory and destabilizing the service.

Impact This is a denial-of-service (DoS) vector. A single large request or a burst of concurrent large requests can cause:

- ▶ High memory consumption leading to OOM (Out-of-Memory) crashes.
- ▶ Service unavailability or degraded performance for legitimate users.

As the verification service is used to validate the authenticity of the SGX quotes, denial of service attacks on the verification service could be used to delay the identification of oracle notarization compromises.

Recommendation Enforce strict upper bounds on the HTTP body size using io.LimitReader. Return HTTP status 413 Request Entity Too Large if the size limit is exceeded before decoding begins.

Developer response The developers implemented the suggested fix.

5.1.17 V-ORC-VUL-017: Encoded price feed proof data cannot be decoded

Severity	Medium	Commit	N/A
Type	Logic Error	Status	Fixed
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-notarization-backend/[...]/quote_preparation.go:69-77 ▶ oracle-verification-backend/attestation/ <ul style="list-style-type: none"> • attestation.go:85-120 • decoding.go:58-77 		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/45		

Description During the attestation process, the notarization backend returns information about a request and the retrieved information. Part of this information, referred to as the user data proof, corresponds to an encoded array that includes important information such as the timestamp, request URL, and request selector. Since this information is encoded using a custom format, the verification service provides an API to decode this information as shown below.

```

1 func DecodeProofData(buf []byte) (*DecodedProofData, error) {
2     if len(buf) < encoding.TARGET_ALIGNMENT*2 {
3         // the buffer doesn't even have a meta header, no need to try to parse
         anything
4         return nil, errors.New("too short to be encoded proof data")
5     }
6
7     // byte position for processing
8     pos := 0
9
10    header, err := encoding.DecodeMetaHeader(buf[:encoding.TARGET_ALIGNMENT*2])
11    if err != nil {
12        return nil, err
13    }
14
15    pos += encoding.TARGET_ALIGNMENT * 2
16
17    // int and float use the length of 255 in the header, they are always encoded as
    1 block.
18    // otherwise it's a string encoded as 256 blocks
19    attestationDataLen := header.AttestationDataLen
20    if header.AttestationDataLen == 255 {
21        attestationDataLen = encoding.TARGET_ALIGNMENT
22    }
23
24    ...
25
26    return &DecodedProofData{
27        ...
28    }, nil
29 }
```

Snippet 5.1: Logic used to decode the user data proof

The decoding logic, however, does not account for some specific manipulations performed to the user data proof when a request is made to a price feed. More specifically, if a request is made to a price feed then the first byte of the user data proof is overwritten with the price feed's identifier as shown below. Since this byte overlaps with the stored attestation data size, attempts to decode the user data proof will fail as the encoded size read above will not match the real size of the attestation data.

```

1 func PrepareOracleUserData(
2     statusCode int,
3     attestationData string,
4     timestamp uint64,
5     attestationRequest AttestationRequest,
6 ) (
7     userDataProof []byte,
8     userData []byte,
9     encodedPositions *encoding.ProofPositionalInfo,
10    err *appErrors.AppError,
11 ) {
12     ...
13
14     if common.IsPriceFeedURL(attestationRequest.Url) {
15         tokenID := common.GetTokenIDFromPriceFeedURL(attestationRequest.Url)
16         logger.Debug("Token ID: ", "tokenID", tokenID)
17         if tokenID == 0 {
18             logger.Error("Unsupported price feed URL: ", "url", attestationRequest.
19             Url)
20             return nil, nil, nil, appErrors.ErrUnsupportedPriceFeedURL
21         }
22         userDataProof[0] = byte(tokenID)
23     }
24     ...
25
26     // Step 5: Return the prepared data.
27     return userDataProof, userData, encodedPositions, nil
28 }

```

Snippet 5.2: Location where the first byte of the user data proof is overwritten if a request is made to a price feed

Impact Attempts to decode the user data proof via the embedded metadata, such as is performed in `DecodeProofData`, will fail. This can make it difficult for users to read information included in the user proof data and for external services to interact with the oracle if they require information stored in the user proof data. Additionally, overwriting the first byte with the token identifier of the price feed can result in errors if the first byte is used to determine if a price feed request is made. This is because the first byte is now overloaded so it is unclear if a price feed request was made if the first byte matches a price feed ID or if attestation data just happens to match the size of a price feed ID.

Recommendation Do not overwrite the first byte of the user data proof with the token ID of the price feed. Doing so compromises the ability to decode the data such as shown in `DecodeProofData` while not adding additional information to the user data proof's hash as the included URL already indicates that a particular price feed was requested. Additionally, any services relying solely on the first byte to determine the price feed can be compromised by selecting appropriately sized attestation data.

Developer response The developers implemented the recommended fix by instead modifying an unused byte of the user data proof.

5.1.18 V-ORC-VUL-018: Silent Truncation of HTML Responses via LimitReader Can Lead to Incomplete Attestation Data

Severity	Medium	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/internal/services/[...]/ ▶ html_parser.go:57 ▶ json_parser.go:60		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/26 , a788d1c		

Description The notarization oracle can attest to date embedded in an HTML page. To do so, elements from the HTML document are extracted from the response using the ExtractDataFromHTML function. The ExtractDataFromHTML function applies a size restriction to HTTP responses using io.LimitReader as shown below:

```
1 func ExtractDataFromHTML(ctx context.Context, attestationRequest attestation.  
    AttestationRequest) (ExtractDataResult, *appErrors.AppError) {  
2     ...  
3  
4     limitReader := io.LimitReader(resp.Body, constants.MaxResponseBodySize)  
5  
6     ...  
7 }
```

LimitReader enforces a hard cutoff by returning EOF once MaxResponseBodySize bytes are read. This results in **silent truncation** if the HTML document exceeds this size limit. Unlike JSON parsing (where truncation usually leads to invalid JSON and an explicit error), truncated HTML may still parse successfully. If the attestation-relevant element lies near or beyond the cutoff, the function may return incomplete or misleading attestation data without any indication of truncation.

Impact

- ▶ **Silent data loss:** Attestation data may be truncated at the size boundary while still appearing valid. This could cause attestation values in particular to be truncated silently.
- ▶ **Incorrect attestations:** Applications depending on full HTML content could incorrectly validate or reject proofs.
- ▶ **Reduced reliability:** Operators have no clear indication that the returned data was incomplete, introducing subtle integrity risks.

Recommendation Mitigation strategies include:

- ▶ **Explicit truncation handling:** Detect when resp.ContentLength > MaxResponseBodySize and return a clear error rather than truncated content.
- ▶ **Post-parse validation:** Ensure that critical selectors were fully captured by verifying document completeness before proceeding.

This approach ensures truncation results in a controlled failure rather than silent acceptance of incomplete data.

Developer response The developers followed the recommendations and fixed the issue.

5.1.19 V-ORC-VUL-019: Duplicate Exchange Prices Can Skew Volume-Weighted Average

Severity	Medium	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/[...]/ price_feed.go:172-197		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/36 , ac6e476		

Description The CalculateVolumeWeightedAverage function, shown below, computes a volume-weighted average price across exchanges. Currently, the function does not detect or handle **duplicate price entries** for the same exchange and symbol. If the input slice contains repeated entries, these will be included multiple times in the calculation, inflating both the weighted sum and total volume.

Since the function only tracks unique exchanges by name for counting, duplicate entries still affect the average price calculation. This could occur accidentally (e.g., due to repeated API responses) or intentionally (e.g., manipulated data feeds).

```

1 func CalculateVolumeWeightedAverage(prices []ExchangePrice) (float64, float64, int) {
2     if len(prices) == 0 {
3         return 0, 0, 0
4     }
5
6     var totalVolume float64
7     var weightedSum float64
8     exchanges := make(map[string]bool)
9
10    for _, price := range prices {
11        if price.Price > 0 && price.Volume > 0 {
12            weightedSum += price.Price * price.Volume
13            totalVolume += price.Volume
14            if _, exists := exchanges[price.Exchange]; !exists {
15                exchanges[price.Exchange] = true
16            }
17        }
18    }
19
20    if totalVolume == 0 {
21        return 0, 0, 0
22    }
23
24    volumeWeightedAvg := weightedSum / totalVolume
25    return volumeWeightedAvg, totalVolume, len(exchanges)
26 }

```

Snippet 5.3: Definition of the CalculateVolumeWeightedAverage function used to aggregate prices from exchanges

Impact

- ▶ **Price skewing:** Duplicate entries from a single exchange can bias the computed average towards that exchange's reported price.
- ▶ **Reduced reliability:** If an exchange has a data issue (e.g., repeatedly returning the same price), this error is amplified.
- ▶ **Silent failure mode:** The function provides no indication that duplicates influenced the calculation, making the skew hard to detect.

Recommendation Add explicit duplicate detection and filtering before aggregation by deduplicate price entries based on (`exchange`, `symbol`) or another unique identifier. This ensures that the volume-weighted average reflects a fair aggregation of prices across exchanges without silent skew.

Developer response The developers followed the recommendation and resolved the issue.

5.1.20 V-ORC-VUL-020: TLS Interception Risk

Severity	Low	Commit	N/A
Type	Data Validation	Status	Partially Fixed
Location(s)	aleo-oracle-notarization-backend/ <ul style="list-style-type: none">▶ deployment/<ul style="list-style-type: none">• docker/<ul style="list-style-type: none">* [...]/ aleo-oracle-notarization-backend.manifest.template:13-14* scripts/<ul style="list-style-type: none">· generate-manifest-template.sh:30-31· install-docker.sh:9• native/scripts/generate-manifest-template.sh:35-36▶ internal/httputil/retry_client.go:11		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/18 , b46ffc1		

Description In enclave-based systems, the OS and hypervisor are considered untrusted by design. A malicious or compromised OS/hypervisor can actively manipulate network-related system calls and enclave-external resources such as DNS resolution and TLS libraries.

The root CA store contains all default certificate authorities leaving opened a potential vulnerability.

An attacker with control of the OS or hypervisor can:

1. Intercept the enclave’s DNS requests and return an attacker-controlled IP for a legitimate domain.
2. Present a valid TLS certificate for the domain, obtained from a lenient but trusted Certificate Authority (CA).
3. Route enclave traffic to the attacker’s server, which completes the TLS handshake using the forged certificate.

This attack is feasible without breaking TLS cryptography or enclave isolation; it instead exploits weak trust boundaries around external identity resolution and certificate validation.

Impact This enables a man-in-the-middle (MITM) attack that breaks the confidentiality and integrity guarantees of the enclave.

- ▶ Injection of malicious data into data or price feeds.
- ▶ Undermining remote attestation-based trust.

Recommendation

- ▶ Implement DNS resolution and certificate validation *within the enclave* whenever possible.
- ▶ Avoid using host OS DNS and permissive CA trust stores - use minimal, enclave-verified CA roots.
- ▶ Pin specific server public keys or certificates inside enclave code to eliminate reliance on external CAs if possible.

Developer response For now, we will pin the root CA certificate and embed those files in the Gramine trusted files. At a later stage, we will also pin the exchange's public key.

5.1.21 V-ORC-VUL-021: Open redirect risk in outbound HTTP requests introduces new attack vectors

Severity	Low	Commit	N/A
Type	Data Validation	Status	Acknowledged
Location(s)	aleo-oracle-notarization-backend/internal/ ▶ httputil/retry_client.go:11-20 ▶ services/dataextraction/data_extractor.go:103-106		
Confirmed Fix At	N/A		

Description The attestation flow permits client-supplied URLs that the backend fetches and processes. While the system performs initial validation on URL schemes and domains (e.g., checking that the host is whitelisted), it fails to revalidate after redirects. The HTTP client used by the backend transparently follows HTTP 30x redirects, meaning an initially safe domain may redirect to a malicious or internal destination (e.g., AWS metadata IPs or internal APIs).

For instance, the attacker can use <https://www.google.com/url?q=https://evil.com> or <https://whitelisted/redirect?target=https://internal.service.local>

If `whitelisted.example.com` returns a 302 redirect to the internal URL, and the backend follows it without a secondary check, this results in an **open redirect vulnerability** bypassing domain allow-lists.

Additionally, this enables:

- ▶ Server-Side Request Forgery
- ▶ Exploit chaining, possibly in combination with parsing bugs or zero-days in HTTP libraries.

A malicious actor can cause the backend to retrieve attacker-controlled data, which may be interpreted as valid attestation material. If such data is cached, signed, or forwarded to clients, it undermines the integrity guarantees expected from the attestation process.

Impact An attacker can:

- ▶ Bypass domain allow-lists and fetch internal/private resources.
- ▶ Deliver crafted payloads to exploit backend libraries (e.g., via SSRF or parsing bugs), and potentially trigger remote code execution.
- ▶ Circumvent rate-limiting or access control by leveraging trusted domains.

Recommendation

- ▶ Disable automatic redirect following or implement a `CheckRedirect` callback that re-validates every redirect target against the allow-list.

Developer response We will only support price feed requests. We will exclude all the general whitelisted domains.

5.1.22 V-ORC-VUL-022: Channel Blocking and Goroutine Leak in executeRequest

Severity	Low	Commit	N/A
Type	Logic Error	Status	Fixed
Location(s)	aleo-oracle-sdk-go/request.go:98, 183, 296-302		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-sdk-go/pull/10		

Description The SDK launches a concurrent worker for each resolved backend address, where each worker attempts a request and writes successful responses to a shared channel (successInternalCh). This channel is a single-slot buffered channel meant to store only the first successful result.

Each worker calls executeRequestInternal, and upon success, writes to successInternalCh without checking whether the channel is already full. The outer function consumes only one message from the channel and exits after receiving the first success, triggering cancellation for all other workers.

However, due to concurrency and scheduling, multiple workers may successfully write to the channel before the cancellation propagates. The first success is consumed and triggers cancel(), but subsequent successes may attempt to enqueue again:

- ▶ The second successful response may write to the channel if it hasn't been canceled or the channel hasn't yet filled.
- ▶ Further successes block indefinitely trying to write to the full channel.
- ▶ Blocked workers never reach their wg.Done(), leading to a leaked goroutine and a hung wgInternal.Wait().

This creates a classic goroutine leak due to an unbounded blocking send on a shared channel without guaranteed consumption.

Impact Under load or slow cancellation propagation, this can lead to goroutine leaks that accumulate over time, potentially exhausting resources. It can degrade performance, delay clean shutdowns, or trigger out-of-memory conditions in high-traffic or long-lived services. Affected services may appear to hang waiting for wg.Wait() completion.

Recommendation Modify the success reporting logic in executeRequestInternal to use a non-blocking select when writing to successInternalCh. This prevents blocked sends if the channel is already full or canceled:

```

1 select {
2     case successInternalCh <- resp:
3     case <-ctx.Done():
4     default:
5 }

```

Ensure that all workers reliably call `wg.Done()` even if they fail to enqueue a success. Alternatively, consider designing the success reporting to use a bounded queue or cancellation-aware workgroup model that guarantees draining or cancellation propagation to unblock writers.

Developer response The developers implemented the recommended fix.

5.1.23 V-ORC-VUL-023: Missing keep-alive hardening of notarization server

Severity	Low	Commit	N/A
Type	Usability Issue	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/cmd/[...]/main.go:42-57		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/10 , 1a69ee4		

Description The `NewServer` function creates and returns two `*http.Server` instances but does not explicitly disable HTTP connection reuse (keep-alives). In Go's standard library, persistent connections are enabled by default, which means a client can send multiple HTTP requests over a single TCP connection unless this is disabled using `server.SetKeepAlivesEnabled(false)`.

When deployed behind certain reverse proxies (e.g., Nginx, HAProxy, AWS ALB) that may interpret and forward requests differently from Go's native HTTP parser, this setup can lead to HTTP request smuggling vulnerabilities. A mismatched understanding of request boundaries between frontend proxy and backend server allows an attacker to craft ambiguous HTTP requests that are partially processed by the proxy and interpreted differently by the backend.

Example scenario: A proxy forwards a request to the backend assuming the request ends at a certain byte boundary (based on headers like `Content-Length`), but the backend interprets additional data as a second request. If keep-alives are enabled, that second request may be processed under the wrong connection context, potentially bypassing access control or allowing cache poisoning.

Impact An attacker may exploit the mismatch in HTTP request parsing to:

- ▶ Smuggle unauthorized requests past authentication layers.
- ▶ Interfere with or poison backend state (e.g., via header manipulation or partial request injection).
- ▶ Execute privilege escalation or cross-user request hijacking in multi-tenant systems.
- ▶ Evade security monitoring tools that analyze frontend traffic only.

This class of vulnerability has been observed in real-world attacks targeting proxies with different parsing behaviors than application servers.

Recommendation Explicitly disable HTTP keep-alives (`server.SetKeepAlivesEnabled(false)`) for all `*http.Server` instances returned by `NewServer`, unless end-to-end request parsing alignment with all upstream proxies is guaranteed and tested. Disabling persistent connections ensures that leftover or smuggled bytes are not interpreted as a new request, eliminating the attack vector.

Developer response The keep-alive flag has been set to false in line with the recommendation.

5.1.24 V-ORC-VUL-024: Spectre, Meltdown, ZombieLoad, Downfall mitigation

Severity	Low	Commit	N/A
Type	Side-channels	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/ ▶ Makefile:194 ▶ deployment/ <ul style="list-style-type: none">• docker/<ul style="list-style-type: none">* Dockerfile:19* [...]/ aleo-oracle-notarization-backend.manifest.template:20* scripts/<ul style="list-style-type: none">· generate-manifest-template.sh:37· install-docker.sh• native/scripts/generate-manifest-template.sh:42		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/50 , 1bc2973		

Description The application executes Aleo signature operations within a Gramine SGX enclave using the Wazero WebAssembly runtime. The code initializes a JIT-compiled runtime with:

```
1 runtimeConfig := wazero.NewRuntimeConfigCompiler()
2 runtime := wazero.NewRuntimeWithConfig(context.Background(), runtimeConfig)
```

This configuration uses Wazero’s compiler-based backend inside the enclave. Gramine, by default, may permit executable memory regions (e.g., via mmap with PROT_EXEC), which are exploitable in speculative execution attacks such as Spectre, Meltdown (CVE-2017-5754), and Downfall (CVE-2022-40982).

In addition, Go’s compiler and toolchain do not apply hardware-level mitigations or constant-time protections against microarchitectural attacks. Since Wazero JIT compiles guest WASM code, it may generate executable memory pages within the enclave that are accessible by speculative execution paths.

If the Go host code or Wazero runtime initiates memory mappings with executable permissions, the combination of SGX + JIT + relaxed enclave configuration can expose attack surfaces to speculative execution side channels.

Impact Attackers with local system access (e.g., cloud co-tenants or local processes) may exploit microarchitectural leakage to extract enclave-resident secrets, such as Aleo private keys, through speculative execution attacks.

This includes:

- ▶ **Meltdown:** Leakage of privileged memory inside the enclave.
- ▶ **Downfall:** Exposure via speculative vector instructions (e.g., gather).
- ▶ **Spectre v2:** JIT-generated gadgets inside the enclave enabling register or memory inference via branch prediction.

The risk is amplified when enclave memory mappings are executable and the JIT runtime introduces dynamically generated code, increasing gadget availability.

Recommendation Go Compiler Options

- ▶ Build with [Spectre mitigations enabled](#):
 -gcflags=all=-spectre=all -asmflags=all=-spectre=all or
 -spectre=index,ret for targeted defenses.

Kernel and Microcode Updates

- ▶ Apply the latest kernel patches and Intel microcode for Meltdown and Downfall/GDS.
`apt-get install intel-microcode`
- ▶ Verify status via:
 - `grep -i vulnerable /sys/devices/system/cpu/vulnerabilities/*`
 - * for /sys/devices/system/cpu/vulnerabilities/l1tf / Mitigation: PTE Inversion; VMX: conditional cache flushes, SMT vulnerable
 - `use echo off | sudo tee /sys/devices/system/cpu/smt/control` or `kernel param nosmt`
 - `cat /sys/devices/system/cpu/vulnerabilities/meltdown`
 - * Expected: Mitigation: PTI, or Not affected. Alternative: Vulnerable
 - `cat /sys/devices/system/cpu/vulnerabilities/gds`
 - * Expected: Mitigation: Software; microcode, Alternative: SWHardeningNeeded
- ▶ On unsupported hardware, disable gather instructions. [This can be done by setting `gather_data_sampling="force"` or `clearcpuid=avx` on the kernel command-line.](#)
- ▶ EGo attestation results in SWHardeningNeeded, the CPU is likely be vulnerable to GDS/Downfall: Microcode update is present but Kernel/software-side mitigation is missing or disabled.

WASM Runtime Choices

- ▶ Favor interpreter mode or ahead-of-time compilation to avoid JIT-generated executable pages.

Developer response The developers followed the recommendations and resolve the issue.

5.1.25 V-ORC-VUL-025: Silent Attestation Loss Due to Front-Loaded Report Errors

Severity	Low	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-sdk-go/notarize.go:507-516 ▶ oracle-verification-backend/api/[...]/verify.go:106-131 		
Confirmed Fix At	https://github.com/venture23-aleo/oracle-verification-backend/pull/14 , https://github.com/venture23-aleo/aleo-oracle-sdk-go/pull/12		

Description In the aleo-oracle-sdk-go client code, the client only returns an error if *no* valid attestations are reported by the verification backend:

```

1 if len(result.ValidReports) == 0 {
2     return nil, errors.New(result.ErrorMessage)
3 }

```

However, in the oracle-verification-backend, the backend processes the reports sequentially and stops on the first decoding or verification error:

```

1 for i, v := range request.Reports {
2     reportBytes, err := base64.StdEncoding.DecodeString(v.AttestationReport)
3     if err != nil { ...; break }
4     ...
5     if err != nil { ...; break }
6     ...
7     if err != nil { ...; break }
8     validReports = append(validReports, i)
9 }

```

As a result:

- ▶ If the **first** report is malformed (e.g., invalid Base64), the backend stops and never inspects remaining reports. The client sees no ValidReports, and returns the backend's error.
- ▶ If some **early reports are valid** but a **later one is malformed**, the backend still stops, but returns the early ValidReports. The client interprets this as a success, even though some reports were skipped and an error was present.

This behavior can lead to partial verification while incorrectly reporting success to the user.

Example Scenario:

1. Client submits a batch of 5 attestation reports.
2. The first report is valid, the second is malformed.
3. Backend processes the first, adds it to ValidReports.
4. Backend fails on the second report and exits early.
5. Client sees len(ValidReports) == 1, and assumes verification succeeded.
6. Remaining 3 reports are never checked.

Impact

- ▶ **Partial Verification:** Malformed or malicious reports may go unnoticed if they occur after at least one valid entry.
- ▶ **Silent Dropping of Attestations:** Legitimate attestations may be skipped entirely if a malformed report precedes them.
- ▶ **Denial-of-Service via Input Corruption:** A single corrupted report (e.g., via transient network error or malicious injection) can prevent all subsequent attestations from being considered.
- ▶ **Security Blind Spot:** The client may falsely assume a full batch was verified, leading to incorrect notarization or false attestations.

Recommendation

- ▶ On the **backend**, replace break statements with continue so that all reports are processed regardless of earlier errors.
- ▶ Maintain a separate Errors list in the response to surface any failures explicitly alongside ValidReports.
- ▶ On the **client**, treat a response as valid only if **all** reports were verified successfully, or explicitly expose the partial verification state and associated errors to the caller.
- ▶ Consider performing pre-validation (e.g., Base64 decode) on the client side before submission to reduce malformed input cases.

Developer response The developers implemented the suggested fix.

5.1.26 V-ORC-VUL-026: Field Name Mismatch

Severity	Low	Commit	N/A
Type	Serialization Error	Status	Fixed
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-notarization-backend/[...]/attestation.go:60-72 ▶ aleo-oracle-sdk-go/notarize.go:533-545 ▶ aleo-oracle-sdk-js/src/types/attestation.ts:269-284 		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/16		

Description Both the Go and JS SDKs define functions to perform simple queries to the notarization service that checks if a given selector can be extracted. Both define types that are used to help parse the response from the "debug" notarization response but both use a field named extractedData for the decoded response, but the producer message uses attestationData. As a result, JSON decoding will not populate this field in either SDK.

The types for the Go and JS SDKs can be found below:

```

1 type TestSelectorResponse struct {
2     // URL of the Notarization Backend the response came from.
3     EnclaveUrl string `json:"enclaveUrl"`
4
5     // Full response body received in the attestation target's response
6     ResponseBody string `json:"responseBody"`
7
8     // Status code of the attestation target's response
9     ResponseStatusCode int `json:"responseStatusCode"`
10
11     // Extracted data from ResponseBody using the provided selector
12     ExtractedData string `json:"extractedData"`
13 }

```

Snippet 5.4: Response extractor from the Go SDK

```

1 export type DebugRequestResponse = {
2     /**
3      * Full response body received in the attestation target's response.
4      */
5     responseBody: string;
6
7     /**
8      * Status code of the attestation target's response.
9      */
10    responseStatusCode: number;
11
12    /**
13     * Extracted data from 'responseBody' using provided selector.
14     */
15    extractedData: string;
16 };

```

Snippet 5.5: Response extractor from the JS SDK

These structs are used to parse messages from the following debug attestation response though:

```
1 type DebugAttestationResponse struct {  
2     ReportType string 'json:"reportType"' // The report type.  
3  
4     AttestationRequest AttestationRequest 'json:"attestationRequest"' // The  
5     attestation request.  
6  
7     AttestationTimestamp int64 'json:"timestamp"' // The attestation timestamp.  
8  
9     ResponseBody string 'json:"responseBody"' // The response body.  
10  
11     ResponseStatusCode int 'json:"responseStatusCode"' // The response status code.  
12  
13     AttestationData string 'json:"attestationData"' // The attestation data.  
14 }
```

Snippet 5.6: Go struct used to produce debug attestation responses

Impact Clients will receive empty/zero values for this field because the JSON key does not match. This can silently break downstream logic that relies on the attestation payload (e.g., validation, display, or persistence), and can lead to confusing "missing data" states during debugging.

Recommendation Align consumer field names and tags with the producer:

- ▶ **Go SDK:** Rename to `AttestationData` string json:"attestationData".
- ▶ **TS SDK:** Rename to `attestationData: string;`.
- ▶ Add unit tests that decode a sample payload containing `attestationData` and assert the field is populated in both SDKs.

Developer response The developers implemented the recommended fix by changing the `attestationData` field in the notarization service to `extractedData`.

5.1.27 V-ORC-VUL-027: Partial Failure Masking in GetEnclavesInfo

Severity	Low	Commit	N/A
Type	Usability Issue	Status	Fixed
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-sdk-go/info.go:204-230 ▶ aleo-oracle-sdk-js/src/client.ts:170-183 		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-sdk-go/pull/4		

Description GetEnclavesInfo only returns errors when **all** enclave requests fail as shown below. If at least one enclave responds, **all other errors are silently dropped**. For an API that reports enclave state, this behavior hides misconfigurations/outages and prevents operators from fixing issues. Note, that the Javascript SDK behaves similarly. Additionally, the current concurrency pattern can dereference nil and **panic** when a request fails.

```

1 func (c *Client) GetEnclavesInfo(options *EnclaveInfoOptions) ([]*EnclaveInfo, []
   error) {
2     ...
3
4     // result channels are closed before receiving
5     for _, ch := range resChanMap {
6         close(ch)
7     }
8     close(errChan)
9
10    if len(errChan) > 0 {
11        var reqErrors []error
12        for err := range errChan {
13            reqErrors = append(reqErrors, err)
14        }
15
16        // all request have failed
17        if len(reqErrors) == numServices {
18            return nil, reqErrors
19        }
20    }
21
22    var info []*EnclaveInfo
23    for enclaveUrl, resChan := range resChanMap {
24        enclaveInfo := <-resChan
25        // potential nil deref if request failed
26        enclaveInfo.EnclaveUrl = enclaveUrl
27        info = append(info, enclaveInfo)
28    }
29
30    // If not all requests failed, errors are ignored
31    return info, nil
32 }

```

Snippet 5.7: Snippet of the GetEnclavesInfo API that ignores errors unless all services error

Impact

- ▶ Observability loss: Operators won't see which enclaves errored; issues persist undetected.
- ▶ Incorrect/incomplete state: Callers may assume all enclaves are healthy based on partial data.
- ▶ Stability risk: Reading from a closed, empty resChan yields nil; setting EnclaveUrl on nil can panic.

Recommendations

1. Propagate partial errors. Return info and the collected errors when any request fails.
2. Nil-safety. Never dereference enclaveInfo without a nil check.

Developer response The developers implemented the recommended fix.

5.1.28 V-ORC-VUL-028: Silent Truncation of >128-bit Integers in bigIntStrToBytes

Severity	Low	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	oracle-verification-backend/contract/contract.go:16-35		
Confirmed Fix At	https://github.com/venture23-aleo/oracle-verification-backend/pull/12 , 9c4f7a2		

Description bigIntStrToBytes always emits **exactly 16 bytes** in little-endian order but never verifies that the parsed integer fits within 128 bits. If the input represents a value larger than 128 bits, the high-order bits are **silently truncated**. This can mask configuration/logic errors and, depending on usage, lead to subtle correctness or security issues.

```

1 func bigIntStrToBytes(strBigInt string) []byte {
2     num := new(big.Int)
3
4     num, ok := num.SetString(strBigInt, 10)
5     if !ok {
6         return nil
7     }
8
9     bytes := make([]byte, 16)
10
11     // Extract bytes in little-endian order
12     for i := 0; i < 16; i++ {
13         b := byte(num.Uint64() & 0xff)
14         bytes[i] = b
15
16         num.Rsh(num, 8)
17     }
18
19     return bytes
20 }
```

Snippet 5.8: The bigIntStrToBytes function used to convert an integer string to bytes

Impact Silent overflow: Inputs >128 bits are accepted but truncated, producing incorrect output without any signal to callers.

Recommendation After consuming 16 bytes, ensure `num == 0`, otherwise report an overflow.

Developer response The developers implemented the recommended fix.

5.1.29 V-ORC-VUL-029: Hardcoded Aleo API Request Path Reduces Maintainability

Severity	Low	Commit	N/A
Type	Denial of Service	Status	Fixed
Location(s)	oracle-verification-backend/[...]/contract.go:171-177, 190		
Confirmed Fix At	https://github.com/venture23-aleo/oracle-verification-backend/pull/16		

Description Both GetSgxUniqueIDAssert and GetNitroPcrValuesAssert functions construct Aleo API request URLs by hardcoding path elements as shown below:

```
1 requestUrl := apiBaseUrl + "/program/" + url.PathEscape(contractName) + "/mapping/  
   sgx_unique_id/0u8"  
2 requestUrl := apiBaseUrl + "/program/" + url.PathEscape(contractName) + "/mapping/  
   nitro_pcr_values/0u8"
```

This approach assumes the Aleo API structure (e.g., /program/{contract}/mapping/{key}/0u8) and the mapping names (sgx_unique_id, nitro_pcr_values) remain stable. If Aleo changes its endpoint format, naming conventions, or keying scheme, these functions will fail.

Unlike the notarization service, which uses configuration templates, these functions hardcode request paths directly in code, reducing flexibility and maintainability.

Impact This issue does not pose a direct security vulnerability but creates a **maintainability and availability risk**:

- ▶ Any upstream change in Aleo’s API or mapping schema will break the service.
- ▶ Recovery requires modifying and redeploying code, instead of adjusting configuration.
- ▶ This increases operational overhead and service downtime risk.

Recommendation Avoid hardcoding Aleo API request paths. Instead:

- ▶ Define request patterns (e.g., "/program/{contract}/mapping/{mapping}/{key}") as **configuration templates**.
- ▶ Load mapping names and keys (sgx_unique_id, nitro_pcr_values, 0u8) from config files or environment variables.
- ▶ Align this implementation with the notarization service’s configurable template approach.

This makes the service resilient to upstream changes and easier to maintain.

Developer response The developers implemented the recommended fix.

5.1.30 V-ORC-VUL-030: Unsynchronized Access to AleoContextManager State Causes Race Conditions

Severity	Low	Commit	N/A
Type	Race Condition	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/[...]/aleo_util.go:85-124		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/24 , 38c0f88		

Description The notarization service relies on embedded WASM code to interact with Aleo utilities. Access to this code is guarded by the AleoContextManager which includes a mutex presumably to ensure exclusive access to the shared WASM utilities. AleoContextManager reads and writes shared fields without consistent locking though.

- ▶ GetAleoContext() (both the package wrapper and the method) returns `m.context` and uses `m.initialized` **without acquiring** `mu`; only `once.Do` is used during first init (which does not protect later reads).
- ▶ InitAleoContext() publishes context and initialized **without** `mu`.
- ▶ ShutdownAleoContext() **modifies** `context/initialized` **with** `mu.Lock()`.

This asymmetric synchronization creates a race between unlocked reads and locked writes.

Impact

- ▶ **Data races and stale reads:** Callers can observe a context being torn down (e.g., `initialized=false` but non-nil context) or read nil after shutdown starts.
- ▶ **Intermittent failures:** Possible nil dereferences, signing with a closed session, and flaky behavior that only appears under load; the Go race detector would flag this.

Recommendation Adopt one consistent locking strategy when using the AleoContextManager.

Developer response The developers implemented the recommended fix.

5.1.31 V-ORC-VUL-031: Logging is not integrated with Gramine's secure logging

Severity	Low	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/internal/[...]/logger.go		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/22		

Description The project defines a custom logger using Go's `slog` package, initialized via `InitLogger` to emit logs in plaintext to `os.Stdout`. Logging is structured and includes HTTP middleware that injects request IDs and logs incoming requests and responses.

However, the current logging design does not meet Gramine's requirements for secure enclave environments due to three key issues:

1. **Bypass of Structured Logging for Fatal Errors** The main function uses `log.Fatalf` from Go's standard library instead of the configured structured logger. This emits unstructured output and bypasses centralized log handling.
2. **Logs Not Sent to a Secure Sink** All log output is sent to `os.Stdout`, which in Gramine is not a trusted or attested channel. There is no use of Gramine's secure logging facility (e.g., HMAC chaining or sealed logging mechanisms), so logs can be tampered with by an untrusted host OS.

Example of insecure logger initialization:

```

1 Logger = slog.New(
2     slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{Level: level}),
3 )

```

This writes plaintext logs directly to a host-controlled stream.

Impact In a Gramine SGX deployment, unprotected logging leads to:

- ▶ **Log tampering:** The host OS can modify or truncate logs without detection.
- ▶ **Loss of integrity and auditability:** There's no cryptographic linkage between log entries.
- ▶ **Unstructured panics:** `log.Fatalf` outputs bypass structured logging and are not integrity-protected.

Recommendation Replace all `log.Fatalf` calls with structured logger calls so that fatal events flow through the same secure pipeline. Implement a custom `slog.Handler` that writes records and adds a sequence number plus HMAC chaining for each entry. This ensures tamper-evidence: any truncation, modification, or reordering of logs is detectable during verification.

Developer response Developers have partially implemented the fixes for the current use case; tamper-proof logging is out of scope for the present requirements.

5.1.32 V-ORC-VUL-032: DNS domain hijacking

Severity	Warning	Commit	N/A
Type	Data Validation	Status	Acknowledged
Location(s)	aleo-oracle-notarization-backend/[...]/config.json:140		
Confirmed Fix At	N/A		

Description The GenerateAttestationReport function validates data sources by checking the hostname against a static allowlist. After validation, it fetches external data via ExtractDataFromTargetURL.

This validation relies solely on string comparison of hostnames. Currently, api.coinbasecloud.net resolves to NXDOMAIN, though it remains a registered domain. If the domain becomes available or is misconfigured, an attacker could serve forged data under a valid TLS certificate, which would be accepted as legitimate due to the lack of deeper verification.

Impact An attacker controlling a whitelisted but inactive domain can provide forged attestation data, potentially leading to incorrect on-chain decisions and financial loss.

Recommendation

- ▶ Remove inactive or unresolved domains from the allowlist.
- ▶ Monitor DNS and registration status for all allowlisted domains.
- ▶ Enforce TLS certificate pinning for attestation endpoints.

Developer response We will be only supporting price feed request and will remove all the whitelisted/allowed domains. Also we are pinning the root CA's certificate for the price feed exchanges API and validating it in a runtime.

5.1.33 V-ORC-VUL-033: Defensive Programming Gaps

Description Across the oracle repositories, there is defensive validation that it would be useful to perform to avoid potential issues such as nil pointer de-references and indexing into empty slices. These behaviors are widespread across key components and expose the application to panic-based runtime crashes under malformed or adversarial input.

aleo-oracle-encoding

- ▶ aleo-oracle-encoding/encoding.go:372-387
EncodeAttestationData and EncodeEncodingOptions dereference option pointers without checks, leading to panics if called with nil options.
 - ▶ aleo-oracle-encoding/encoding.go:478-496
EncodeAttestationData and EncodeEncodingOptions dereference option pointers without checks, leading to panics if called with nil options.
 - ▶ aleo-oracle-encoding/positionRecorder/recorder.go:37-39
NewPositionRecorder panics on odd blockSize values, abruptly terminating the application instead of returning an error.
-

aleo-oracle-notarization-backend

- ▶ aleo-oracle-notarization-backend/internal/aleoutil/aleo_util.go:35-37
AleoContext.Sign uses a.Session without confirming initialization, allowing a nil session dereference after failed setup or shutdown.
- ▶ aleo-oracle-notarization-backend/internal/services/attestation/encoding.go:33-57
prepareAttestationData dereferences encodingOptions directly, risking a panic on a nil pointer.
- ▶ aleo-oracle-notarization-backend/internal/services/attestation/quote_preparation.go:107-116
PrepareOracleEncodedRequest dereferences encodedPositions before ensuring it isn't nil.
- ▶ aleo-oracle-notarization-backend/internal/services/enclaveinfo/enclave_info.go:40-48
formatSGXReport assumes sgxReport is non-nil, accessing sgxReport.Body unconditionally.
- ▶ aleo-oracle-notarization-backend/internal/logger/logger.go:112 FromContext
returns nil when the global Logger is uninitialized yet callers like ErrorWithContext immediately invoke methods on its return value. Without checking for a nil logger, or guarding against a nil context.Context, these helpers can panic during logging operations.
- ▶ aleo-oracle-notarization-backend/internal/middleware/middleware.go:14 The Chain helper invokes each middleware function without validating the handler or middleware slice entries. Passing a nil handler or including a nil middleware causes a nil-function call and panics when the chain executes.
- ▶ aleo-oracle-notarization-backend/internal/metrics/collector.go:23 The Start method offers no protection against being invoked more than once. Each invocation

reassigns `stopChan`, and `Stop` always closes it. When `Start` runs twice, subsequent `Stop` calls attempt to close an already closed channel, triggering a runtime panic.

- ▶ `aleo-oracle-notarization-backend/internal/config/config.go:160` validates the configured `minExchangesRequired` threshold such that at least one exchange response must be required when querying price feeds. While the developers use 2 in their current configuration, the developers should consider raising the required value as 1 response could allow for easier manipulation of the price.
- ▶ `aleo-oracle-notarization-backend/internal/services/attestation/quote_preparation.go:212-225` assumes that the input request hash has a size of 16 bytes. It would be useful to validate the size of the request hash to protect against mistakes and identify breakages if the hash size changes.
- ▶ `aleo-oracle-notarization-backend/internal/services/dataextraction/price_feed.go:104` does not validate that `{symbol}` is included in the endpoint template. Similarly no validation of this is performed in the configuration validation logic. If `{symbol}` is not present due to misconfigurations or manipulation, this would result in the same endpoint being queried for every symbol.
- ▶ `aleo-oracle-notarization-backend/internal/sgx/quote.go:85` does not validate the size of the input data as it is assumed to be a hash. The invoked copy function will truncate the input data, however, so if the developers decide to add new information to the quote it could be silently stripped leading to the quote not including essential data.

aleo-oracle-sdk-go

- ▶ `aleo-oracle-sdk-go/request.go:33-67`
`getFullAddress` accesses backend fields without verifying the pointer, so a missing `CustomBackendConfig` triggers a panic.
- ▶ `aleo-oracle-sdk-go/request.go:214-233`
The `executeRequest` helper retrieves the request context's transport and immediately asserts it is an `*http.Transport`, copying and mutating the struct:

```

1 transport := ctx.Transport
2 transportRaw := *(transport.(*http.Transport))
3 transportRaw.DisableKeepAlives = true
4 transport = &transportRaw

```

Because `ctx.Transport` is declared as `http.RoundTripper`, any caller can supply a different implementation or even `nil`. The unchecked type assertion `transport.(*http.Transport)` will panic when `ctx.Transport` is `nil` or not an `*http.Transport`, terminating the request instead of returning an error.

aleo-utils-go

- ▶ `aleo-utils-go/session.go:77-108`
&

aleo-utils-go/session.go:173-181

NewPrivateKey and FormatMessage, which panics on empty slices: several WASM wrapper methods index the return value (

privKeyPtr

[0],

messagePtr[0]

- without checking length.

► aleo-utils-go/wrapper.go:103-129

The NewSession function instantiates a WASM module and populates an aleoWrapperSession by directly assigning exports such as new_private_key, get_address, and sign. The code does not check whether each mod.ExportedFunction call succeeds, so if the module fails to export one of the expected functions, the corresponding field in the session remains nil.

The session type stores these exports as api.Function fields without any guard against missing values.

oracle-verification-backend

► oracle-verification-backend/api/api.go:14-31

CreateApi assumes the configuration is non-nil. conf is dereferenced immediately to build targetPcrs and register handlers, so a nil conf would panic.

► oracle-verification-backend/api/api.go:26-28

CreateApi indexes conf.PcrValuesTarget[0..2] without slice bounds checks, risking an out-of-range panic when fewer PCR values are configured.

► oracle-verification-backend/api/handlers/info.go:60-63

infoHandler.ServeHTTP slices decoded identifiers ([0:16], [16:32]) while ignoring decoding errors, risking panic on malformed or short input.

► oracle-verification-backend/api/handlers/middleware.go:25-70

Context helpers call ctx.Value without nil guards. A nil context.Context would panic in GetContextLogger, GetContextRequestId, and GetContextHandlerName.

► oracle-verification-backend/attestation/attestation.go:85-99

VerifyReportData uses the resp pointer immediately, so a nil response causes a panic.

► oracle-verification-backend/attestation/attestation.go:92-96

VerifyReportData writes to dataBytes[0] without ensuring the slice has capacity, which panics if PrepareProofData returns an empty slice.

► oracle-verification-backend/attestation/attestation.go:112-116

VerifyReportData slices userData[:16] without validating length. Supplying fewer than 16 bytes causes a runtime panic.

► oracle-verification-backend/attestation/nitro/nitro.go:43-48

VerifyNitroReport panics if the global verifier isn't initialized instead of returning an error.

aleo-oracle-sdk-js

- ▶ `aleo-oracle-sdk-js/src/client.ts:382 settleAttestationResponses` calls `Object.keys(result.reason)` on every rejected Promise. If the rejection reason is `null`, `undefined`, or a non-object, this lookup throws a `TypeError`, halting attestation settlement instead of capturing the error gracefully.
-

aleo-oracle-sdk-go `aleo-oracle-sdk-go/notarize.go:618-622` If a notarization request fails, the corresponding response channel may be empty, and dereferencing `resp` without a `nil` check can cause a panic.

Impact Unchecked dereferences and slice indexing allow malformed or adversarial input to crash the application, resulting in denial-of-service. In protocols like notarization or attestation, this can be triggered remotely (e.g., via malformed requests), which may disrupt service availability or compromise trust assumptions. The severity is elevated when these operations occur in critical protocol paths.

Recommendation Refactor all panic-prone code paths to return errors instead of triggering runtime panics. Implement:

- ▶ Defensive checks for `nil` pointers before dereferencing.
- ▶ Length checks before slice indexing.
- ▶ Graceful fallback or error propagation for unexpected types or missing fields.

Developer response The developers implemented the recommended fixes.

Severity	Warning	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-encoding/ <ul style="list-style-type: none"> • encoding.go:372-387, 478-496 • positionRecorder/recorder.go:37-39 ▶ aleo-oracle-notarization-backend/internal/ <ul style="list-style-type: none"> • aleoutil/aleo_util.go:35-37 • config/config.go:160 • logger/logger.go:112 • metrics/collector.go:23, 31 • middleware/middleware.go:14 • services/ <ul style="list-style-type: none"> * attestation/ <ul style="list-style-type: none"> · encoding.go:33-57 · quote_preparation.go:107-116, 212-225 * dataextraction/price_feed.go:104 * enclaveinfo/enclave_info.go:40-48 • sgx/quote.go:85 ▶ aleo-oracle-sdk-go/ <ul style="list-style-type: none"> • info.go:225 • notarize.go:514, 621 • random.go:80 • request.go:33-67, 214-233 ▶ aleo-oracle-sdk-js/src/client.ts:382 ▶ aleo-utils-go/ <ul style="list-style-type: none"> • session.go:77-108, 173-182 • wrapper.go:103-129 ▶ oracle-verification-backend/ <ul style="list-style-type: none"> • api/ <ul style="list-style-type: none"> * api.go:14-31 * handlers/ <ul style="list-style-type: none"> · info.go:60-75 · middleware.go:25-70 • attestation/ <ul style="list-style-type: none"> * attestation.go:85-116 * nitro/nitro.go:43-48 		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/46 , https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/54		

5.1.34 V-ORC-VUL-034: Outdated versions for dependencies (internal&external)

Severity	Warning	Commit	N/A
Type	Maintainability	Status	Fixed
Location(s)	<ul style="list-style-type: none">▶ aleo-oracle-notarization-backend/[...]/install-sgx-dcap-aesm.sh:18▶ aleo-utils-go/Cargo.toml:3▶ oracle-verification-backend/<ul style="list-style-type: none">• Dockerfile:15• go.mod:7• pccs/Dockerfile:36		
Confirmed Fix At	https://github.com/venture23-aleo/oracle-verification-backend/pull/27		

Description Multiple internal and external dependencies across the project are outdated, including critical components such as the EGo library, Intel SGX DCAP SDK, and supporting packages defined in Dockerfiles and package manifests:

- ▶ ego is downloaded at version 1.5.2 in the Dockerfile, and v1.6.1 in go.mod, while the latest is v1.7.2.
- ▶ aleo-utils is used at 1.0.0 while defined as 1.1.2.
- ▶ pccs_api_version is statically set to "3.0" across both Docker and native configurations, blocking adoption of DCAP SDK API 3.1.
- ▶ DCAP SDK version 1.19 is pinned, which is impacted by CVE-2023-0286, a high-severity vulnerability in OpenSSL (X.400 address type confusion).
- ▶ Unpinned installations via apt-get (libsgx-dcap-default-qpl, libsgx-dcap-ql) introduce implicit version upgrades and non-reproducible builds.

This dependency lag poses risks around operational stability, long-term support, and missed security patches. Post-1.19 DCAP releases introduce key features (e.g., TDX support, updated advisory handling), and mismatches between verifier and PCCS API versions may result in incorrect collateral parsing or failures.

Impact

- ▶ **Security exposure:** continued use of DCAP 1.19 inherits vulnerabilities such as CVE-2023-0286.
- ▶ **Stability risk:** newer PCCS or platform services may reject outdated collateral formats, leading to attestation failures.
- ▶ **Maintainability cost:** evolving upstream ecosystems (EGo, Intel SGX) may deprecate support for older APIs, increasing future upgrade friction.

Recommendation

- ▶ Upgrade to the latest supported versions for all internal and external dependencies:
 - Use ego v1.7.2 consistently across Docker and Go modules.

- Update aleo-utils usage to match its declared version (1.1.2) or upgrade both to a newer compatible release.
- Migrate from DCAP SDK 1.19 to 1.23, ensuring compatibility with PCCS API v3.1.

Developer response The developers implemented the recommended fix.

5.1.35 V-ORC-VUL-035: Process-scoped mutex guards a system-wide attestation interface

Severity	Warning	Commit	N/A
Type	Denial of Service	Status	Acknowledged
Location(s)	aleo-oracle-notarization-backend/internal/sgx/ <ul style="list-style-type: none">▶ quote.go:80▶ report.go:65		
Confirmed Fix At	N/A		

Description The SGX attestation code uses a `sync.Mutex` (`enclaveLock`) to serialize access to sensitive `/dev/attestation` file paths exposed through `gramineAttestationPaths`. This works correctly for serializing access within a single Go process. However, `sync.Mutex` does not provide cross-process synchronization. In multi-process or containerized deployments-such as multiple pods in a Kubernetes cluster or concurrent Go processes-this can lead to concurrent access to the attestation interface.

Impact This risk is mitigated when a deployment guarantees that only a single notarization service runs per virtual machine. In such a scenario, the `sync.Mutex` suffices to protect against intra-process race conditions, and no cross-process synchronization is needed.

This setup ensures that quote/report generation remains isolated and reliable, avoiding malformed SGX evidence or denial-of-service behavior.

Recommendation Enforce architectural deployment constraints to ensure only one notarization service is ever active per virtual machine. This avoids the need for cross-process locking mechanisms while ensuring safe access to the SGX attestation interface.

Developer response We will be deploying single instance of notarization backend per VM.

5.1.36 V-ORC-VUL-036: Uncanceled verification requests in verifyReports

Severity	Warning	Commit	N/A
Type	Usability Issue	Status	Acknowledged
Location(s)	aleo-oracle-sdk-js/src/client.ts:254-259		
Confirmed Fix At	N/A		

Description The `verifyReports` method in `client.ts` uses `Promise.any` to initiate verification requests to multiple backends and resolve with the first successful response:

```
1 response = await Promise.any(resolvedUrls.map(({ ip, path }) =>
2   fetch(this.#verifier, ip, path, fetchOptions)));
```

This approach improves responsiveness but introduces two critical issues:

1. **Resource waste:** The remaining fetch requests continue executing even after one has succeeded, consuming bandwidth and compute unnecessarily, and potentially leaving open sockets longer than needed.
2. **Silent divergence risk:** Only the first successful response is inspected. This means later responses from stricter or better-configured verification servers are silently discarded, even if they contradict the first response. A lenient or misconfigured verifier could return a false positive, allowing verification to succeed incorrectly.

For example, if `verifier-A` accepts malformed attestations and responds fastest, the client accepts it as valid. Later responses from `verifier-B` and `verifier-C` rejecting the attestations are ignored.

Impact An attacker could exploit this by targeting or deploying a lenient verifier that always approves verification. If this verifier responds first, verification would pass even in the presence of conflicting or invalid attestations. This undermines the integrity guarantees of the verification process and enables bypassing security policies enforced by stricter verifiers.

Recommendation Replace `Promise.any` with an approach that evaluates **all** responses before making a trust decision. For example:

- ▶ Send parallel requests to all backends.
- ▶ Collect and parse all responses.
- ▶ Apply quorum rules or require consensus among a threshold of backends before accepting verification.
- ▶ Optionally use `AbortController` to cancel remaining requests once sufficient valid responses are received.

Developer response We will map each domain to a single backend IP. In the SDK configuration, only one verification backend can be specified when creating the client.

5.1.37 V-ORC-VUL-037: Repeated code in context cancellation and missing return

Severity	Warning	Commit	N/A
Type	Maintainability	Status	Fixed
Location(s)	aleo-oracle-sdk-go/request.go:97-184		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-sdk-go/pull/8		

Description The function `executeRequestInternal` repeatedly checks for context cancellation using the pattern:

```
1 select {
2 case <-ctx.Done():
3     return
4 default:
5 }
```

This block appears multiple times within the function, but the final occurrence omits the `return` statement. This means that, in the event of context cancellation, the function may continue executing logic that should be aborted. Additionally, repeating the same `select` block across the function increases the risk of inconsistency and makes the code harder to maintain or audit.

Impact

- ▶ Execution may proceed after the context is cancelled, leading to extra network calls or partial work that must later be rolled back.
- ▶ Repetition increases maintenance cost and makes it easy to introduce inconsistencies-such as the missing return in the last block.

Recommendation Adopt a single `ctx.Err()` check at each decision point and propagate the error upward. This reduces duplication, clarifies intent, and ensures cancellation is handled consistently throughout the function.

```
1 if err := ctx.Err(); err != nil {
2     return
3 }
```

Developer response The developers implemented the recommended fix.

5.1.38 V-ORC-VUL-038: Debug mode is not rejected in verification logic, rejected only on-chain

Severity	Warning	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-contracts/artifacts/[...]/main.leo:283, 358-359 ▶ aleo-oracle-notarization-backend/[...]/enclave_info.go:55-61 ▶ oracle-verification-backend/ <ul style="list-style-type: none"> • README.md:40-47 • attestation/attestation.go:24-60, 25-40, 68-72 • config/config.go:16-27 • contract/contract.go:166-201 		
Confirmed Fix At	https://github.com/venture23-aleo/oracle-verification-backend/pull/6		

Description The system records SGX enclave debug mode status (Debug) in the SGXEnclaveInfo structure but fails to enforce any policy against it during enclave initialization or attestation. The initializeSGXEnclaveInfo function stores the Debug flag as-is and permits continued operation even when Debug == true. This permissiveness extends to the backend verification pipeline, where debug-mode reports are parsed, and attestation metadata is forwarded to the on-chain contract without enforcement.

Although the contract enforces enclave mode checks during the final SGX report verification - asserting enclave_flags & 2u128 == 0u128 to reject debug mode - upstream components (backend and notarization helpers) do not enforce or propagate this restriction.

```

1 function verify_sgx_report(
2     report_data: ReportData,
3     report: Report,
4     sig: signature,
5     pub_key: address
6 ) {
7     // https://github.com/openenclave/openenclave/blob/
8     // e9a0423e3a0b242bccbe0b5b576e88b640f88f85/include/openenclave/bits/sgx/sgxtypes.h#
9     // L1088
10    // verify enclave flags
11    // chunk 0 field 7 contains enclave flags
12    let enclave_flags: u128 = report.c0.f7;
13    assert_eq(enclave_flags & 1u128, 1u128); // enclave initated
14    assert_eq(enclave_flags & 2u128, 0u128); // enclave is not in debug mode
15    assert_eq(enclave_flags & 4u128, 4u128); // enclave is in 64-bit mode

```

This check is present on-chain but not mirrored in:

- ▶ initializeSGXEnclaveInfo, which allows debug enclaves
- ▶ Configuration parsing, which lacks a DebugAllowed policy flag
- ▶ Off-chain contract helpers like GetSgxUniqueIDAssert, which ignore the debug flag
- ▶ Notarization backends, which publish the Debug flag but don't enforce policies on it

Thus, debug enclaves can proceed through backend systems and reach contract interactions before being rejected.

Impact Allowing debug-mode enclaves in off-chain processing introduces the risk of:

- ▶ **Premature trust** in attestations lacking SGX protections
- ▶ **Replay or manipulation** of debug reports upstream of contract enforcement
- ▶ **Misaligned operator assumptions**, where backends appear to perform strict verification but allow insecure modes to pass

Debug enclaves disable key security guarantees such as sealing, anti-replay protection, and tamper resistance. If production systems use debug attestations (intentionally or by mistake), this could result in trusted operations (e.g., identity issuance, data sealing, voting) being manipulated.

Recommendation Enforce debug-mode policy at all layers of the verification stack:

1. **Configuration:** Introduce a `DebugAllowed` option (defaulting to false) in the configuration schema. Reject or warn on unrecognized or missing debug policy fields.
2. **Initialization:** Fail in `initializeSGXEnclaveInfo` when `Debug == true` and policy disallows it.
3. **Backend:** Modify attestation parsing and filtering logic to check `Debug` before propagating reports or metadata to contract helpers.
4. **Contract Helpers:** Extend `GetSgxUniqueIDAssert` and similar functions to require `Debug == false` unless explicitly overridden.
5. **Documentation:** Clearly document that debug enclaves are rejected by default. Operators must explicitly allow them (e.g., in dev/test environments).

Developer response The developers implemented the recommended fix.

5.1.39 V-ORC-VUL-039: Dangerous mutation of fields

Severity	Warning	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-sdk-go/ <ul style="list-style-type: none"> • client.go:46, 48, 57 • notarize.go:325, 387 • random.go:25-26 ▶ aleo-oracle-sdk-js/src/ <ul style="list-style-type: none"> • client.ts:56, 92 • defaults.ts:12 		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-sdk-go/pull/16		

Description Several components in the Go and TypeScript SDKs perform in-place mutation of shared default objects or caller-provided structures, leading to race conditions and state leakage.

In Go:

- ▶ notarize.go:325 replaces a nil options pointer with DEFAULT_NOTARIZATION_OPTIONS, then mutates it by injecting a new context.Context, inadvertently modifying global state across calls.
- ▶ notarize.go:387 mutates fields of the caller's AttestationRequest, altering the original request headers map. While this may match expected behavior, it introduces risks in concurrent usage.
- ▶ client.go:46 uses getFullAddress() during logging, which mutates the backend's .Port field when unset.
- ▶ client.go:48,57 assign slices or references to shared backend configurations, exposing them to subsequent mutations.

In TypeScript:

- ▶ client.ts:56 performs a shallow copy of the config object, but config.notarizer and config.verifier retain shared references and are mutated later.
- ▶ defaults.ts:12 exports mutable default objects, allowing any downstream consumer to alter shared state globally.

Impact Mutating shared or caller-owned state introduces risks of:

- ▶ **Cross-request contamination** - a canceled context, modified headers, or overwritten config can affect unrelated operations.
- ▶ **Race conditions** - concurrent goroutines using shared DEFAULT_* values or backend slices may observe or introduce inconsistent behavior.
- ▶ **Immutability violations** - modifying caller-supplied maps breaks conventional API contracts, leading to subtle bugs.
- ▶ **TypeScript side effects** - shallow-spread configs result in configuration leaks across components, and exported mutable constants introduce global state coupling.

These issues reduce reliability, hinder debugging, and violate concurrency safety expectations, particularly under multi-client or multi-threaded conditions.

Recommendation

► Go SDK

- At `notarize.go:325`, clone `DEFAULT_NOTARIZATION_OPTIONS` before mutation.
- At `notarize.go:387`, build headers on a copy of `AttestationRequest` to avoid modifying caller input.
- At `client.go:46`, refactor `getFullAddress()` to avoid mutating `backend.Port`.
- At `client.go:48,57`, clone backend slices or objects instead of referencing shared state directly.

► TypeScript SDK

- At `client.ts:56`, use deep copies for nested config fields:

```
1 { ...config, verifier: { ...config.verifier }, notarizer: { ...config.  
    notarizer } }
```

- At `defaults.ts:12`, freeze exported constants:

```
1 export const DEFAULTS = Object.freeze({ ... }) // or use 'as const'
```

Developer response The developers implemented the recommended fix.

5.1.40 V-ORC-VUL-040: Computational integrity of weighted average

Severity	Warning	Commit	N/A
Type	Fault injection	Status	Acknowledged
Location(s)	aleo-oracle-notarization-backend/[...]/price_feed.go:183		
Confirmed Fix At	N/A		

Description **Plundervolt** and **VoltPillager** are hardware-level attacks that exploit voltage manipulation to compromise the integrity of secure computations on CPUs, particularly those leveraging **Intel SGX** or **voltage-fault-tolerant environments**. These attacks fall under the broader category of **fault injection** attacks. **Plundervolt** leverages **undervolting** (reducing CPU voltage below stable levels) using Intel's **Dynamic Voltage and Frequency Scaling** interface. **VoltPillager** is a **hardware-based version** of Plundervolt that bypasses software mitigations, but requires **physical access** to the machine and works against systems that are otherwise hardened against Plundervolt.

The attacker may target computation within enclave and trigger random errors in computation of weighted average `weightedSum += price.Price * price.Volume`. Alternatively, the attacker can introduce faults during signature generation and use such error in cryptographic attack to derive private key.

Impact These attacks can **break confidentiality and integrity guarantees** of secure enclaves such as SGX. Successful exploitation allows an attacker to extract secrets, forge or computation results - without requiring software vulnerabilities inside the enclave logic. These issues are especially dangerous for decentralized systems or financial platforms that rely on **trusted hardware** for secure computation.

Recommendation

1. Verify PlunderVolt intel advisory in verification backend

```

1 const plundervoltAdvisory = "INTEL-SA-00289" // Plundervolt
2 for _, adv := range report.TCBAdvisories {
3     if strings.EqualFold(strings.TrimSpace(adv), plundervoltAdvisory) {
4         return fmt.Errorf("policy violation: advisory %s present",
5             plundervoltAdvisory)
6     }
7 }

```

2. Disable frequency scaling

- ▶ grub:
 - msr.allow_writes=off
 - intel_pstate=disable
- ▶ runtime:
 - rmmod msr; echo "blacklist msr" >> /etc/modprobe.d/blacklist.conf

3. Lockdown kernel into integrity mode (or stronger confidentiality mode)

```
1 echo integrity > /sys/kernel/security/lockdown
2 echo confidentiality > /sys/kernel/security/lockdown
```

4. Keep CPU voltage/frequency settings stable, avoiding rapid scaling down to lower voltages.

- ▶ grub: `cpufreq.default_governor=performance`
- ▶ runtime:

```
1 for cpu in /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor; do
2 echo performance > $cpu
3 done
```

5. In Azure, use **Confidential VMs** (DCsv3/DCE_v3) where undervolting is already disabled in firmware.
6. Only whitelist Aleo addresses from trusted parties that have implemented similar defenses and do not have physical access to the machine.

Developer response We are running our notarization backend in Azure DCsv3 series VM.

5.1.41 V-ORC-VUL-041: Standardize SGX enclave info serialization across backend and SDK

Severity	Warning	Commit	N/A
Type	Logic Error	Status	Fixed
Location(s)	<div><div>▶ aleo-oracle-notarization-backend/internal/services/</div><div><div>• attestation/attestation.go:166</div><div>• enclaveinfo/enclave_info.go:25-29</div></div><div>▶ aleo-oracle-sdk-go/info.go:25-29</div></div>		
Confirmed Fix At	<div>https: //github.com/venture23-aleo/aleo-oracle-sdk-go/pull/2</div>		

Description Across different parts of the codebase, the EnclaveInfo structure is defined with diverging field types. In aleo-oracle-notarization-backend/internal/services/enclaveinfo/enclave_info.go, the fields use a mix of uint16 for SecurityVersion and strings for identifiers (UniqueID, SignerID, ProductID).

```
1 SecurityVersion uint16      'json:"securityVersion"' // Security version of the
2 enclave. For SGX enclaves, this is the ISVSVN value.
3 UniqueID        string      'json:"uniqueId"'         // The unique ID for the enclave
4 . For SGX enclaves, this is the MRENCLAVE value.
5 SignerID        string      'json:"signerId"'         // The signer ID for the enclave
6 . For SGX enclaves, this is the MRSIGNER value.
7 ProductID       string      'json:"productId"'        // The Product ID for the
8 enclave. For SGX enclaves, this is the ISVPRDID value.
```

Conversely, aleo-oracle-sdk-go/info.go represents SecurityVersion as an uint and all identifiers as byte slices ([]byte). This inconsistency means that the same logical data is treated differently depending on the package, which can complicate JSON serialization, deserialization, and inter-module communication.

```
1 SecurityVersion uint        'json:"securityVersion"' // Security version of the
2 enclave. For SGX enclaves, this is the ISVSVN value.
3 UniqueID        []byte      'json:"uniqueId"'         // The unique ID for the enclave
4 . For SGX enclaves, this is the MRENCLAVE value.
5 SignerID        []byte      'json:"signerId"'         // The signer ID for the enclave
6 . For SGX enclaves, this is the MRSIGNER value.
7 ProductID       []byte      'json:"productId"'        // The Product ID for the
8 enclave. For SGX enclaves, this is the ISVPRDID value.
```

Impact Because the struct definitions vary, converting data between modules may require additional transformations that are easy to overlook or implement incorrectly. For example, comparing identifiers becomes more error-prone when one side expects a string while the other expects a byte array. These mismatches can lead to subtle logic errors-such as failed equality checks, improper validation, or misinterpreted attestation data-that are hard to trace and may surface only in production scenarios. Beyond functional issues, the divergence also reduces

maintainability: developers must remember which version of the struct is in play and ensure that any refactor or fix is duplicated across multiple definitions.

Recommendations Standardize the `EnclaveInfo` structure across packages so that all fields share consistent types and semantics. Choose the representation-strings or byte slices-that best fits downstream requirements (e.g., JSON readability vs. binary precision) and apply it uniformly. If binary data is necessary, consider using fixed-size arrays or encoding helper functions to keep serialization predictable. After unifying the definition, introduce tests that serialize and deserialize enclave information across boundaries to ensure compatibility. Clear, centralized documentation of the struct's purpose and expected field formats will further aid future contributors.

Developer response The developers implemented the recommended fix.

5.1.42 V-ORC-VUL-042: DecodeQuoteHandler calls WriteHeader before setting Content-Type

Severity	Warning	Commit	N/A
Type	Logic Error	Status	Fixed
Location(s)	oracle-verification-backend/api/[...]/ decode_quote.go:51-53		
Confirmed Fix At	https://github.com/venture23-aleo/ oracle-verification-backend/pull/8,13821ac		

Description The HTTP handler sends the response status using `w.WriteHeader(http.StatusOK)` **before** setting the Content-Type header:

```
1 w.WriteHeader(http.StatusOK)
2 w.Header().Set("Content-Type", "application/json")
3 w.Write(decodedQuote)
```

This ordering is incorrect in Go's `net/http` package. Specifically, once `WriteHeader` is called, it **immediately sends** the HTTP status line and any headers set at that point to the client. Internally, this triggers the `http.ResponseWriter` to flush its headers to the network. Any header modifications made after this—such as setting Content-Type—are silently ignored and will not be included in the actual response.

Because `w.Header().Set("Content-Type", "application/json")` is called **after** headers have already been sent, it has no effect. The server will still send the response body (in this case, `decodedQuote`), but without the Content-Type header indicating that the body is JSON.

As a result, clients may receive a valid HTTP 200 response with a JSON body, but without any Content-Type metadata. This can lead to incorrect parsing or rejection of the response by clients expecting a properly declared MIME type.

Impact Clients consuming the API may fail to interpret the response as JSON if the Content-Type header is missing. This can break compatibility with HTTP clients and could cause deserialization errors or incorrect rendering in browsers or tooling.

Recommendation Set all required headers—including Content-Type: application/json—**before** calling `w.WriteHeader(...)`. As a best practice, headers should always be finalized prior to writing the status code or response body.

Developer response The developers followed recommendation and moved writing the status after setting the headers.

5.1.43 V-ORC-VUL-043: Go SDK leaks connections when response read fails

Severity	Warning	Commit	N/A
Type	Logic Error	Status	Fixed
Location(s)	aleo-oracle-sdk-go/request.go:110-118		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-sdk-go/pull/6		

Description In the current HTTP request handling flow, connection leaks can occur under two distinct conditions due to incorrect placement of `resp.Body.Close()`:

1. **Cancellation before defer:** If the context is canceled (`ctx.Done()`) before reaching the `defer resp.Body.Close()` line, the function exits prematurely and leaves the HTTP response body unclosed. This results in an idle connection leak, which can exhaust connection pools under load.
2. **Early return on read failure:** When `io.ReadAll(resp.Body)` is called *before* deferring `resp.Body.Close()`, any error returned by `ReadAll` (e.g., network interruption or malformed response) will cause the function to return without closing the body. This also leaks the connection.

Additionally:

- **Unbounded memory risk:** The use of `io.ReadAll` without any size limit reads the entire response into memory, which can lead to memory exhaustion if a malicious or misconfigured server returns a large or infinite response.

Example

```

1 req, _ := http.NewRequestWithContext(ctx, "GET", url, nil)
2 resp, err := http.DefaultClient.Do(req)
3 if err != nil {
4     return err
5 }
6
7 if ctx.Err() != nil {
8     return ctx.Err() // resp.Body is not closed here
9 }
10
11 body, err := io.ReadAll(resp.Body) // read failure skips closure if err != nil {
12     return err
13 }
14 defer resp.Body.Close()

```

Impact

- **Resource exhaustion:** Leaked connections can saturate the HTTP client's connection pool, leading to stalled or failed requests across the system.
- **Denial of service:** Attackers could trigger this behavior to gradually drain the connection pool or memory by causing early exits (e.g., with slow or malformed responses).

- **Out-of-memory crash:** Unbounded `io.ReadAll` calls on large responses may exhaust heap memory, potentially crashing the process or triggering OOM killer behavior.

Recommendation

1. **Defer immediately:** Place `defer resp.Body.Close()` right after confirming `err == nil` from `http.Do`, regardless of whether the context is canceled or not.
2. **Bound response size:** Use `io.LimitReader` or similar mechanisms to cap the amount of data read from the response body.
3. **Context check:** Use context timeouts/deadlines properly, but do not conditionally skip cleanup logic (i.e., always close the body even if `ctx.Err()` is set).

Developer response The developers implemented the recommended fix.

5.1.44 V-ORC-VUL-044: Missing Cardinality Checks for PcrValuesTarget

Severity	Warning	Commit	N/A
Type	Data Validation	Status	Acknowledged
Location(s)	oracle-verification-backend/config/config.go:60-87		
Confirmed Fix At	N/A		

Description validateAndNormalizePcrValues verifies the **length of each PCR entry** but **never validates the size of the PCR array itself**. This allows configurations that are missing required PCRs or include extra/unexpected PCRs to pass validation as long as each individual value decodes to the expected byte length.

```
1 func validateAndNormalizePcrValues(conf *Configuration) error {
2     for pcrIdx, pcr := range conf.PcrValuesTarget {
3         var pcrBytes []byte
4         var err error
5
6         pcrBytes, err = hex.DecodeString(pcr)
7         isHex := err == nil
8
9         ...
10
11        if len(pcrBytes) != expectedPcrValueLength {
12            log.Printf("config: invalid Nitro PCR value: \"%s\"", pcr)
13            return fmt.Errorf("config \"PcrValuesTarget\" values must be %d bytes",
14                expectedPcrValueLength)
15        }
16    }
17
18    return nil
19 }
```

Snippet 5.9: The following snippet shows the validation of individual PCR entries but also shows a missing check on the overall size of the PCR values array

Impact

- **False sense of verification:** A config with only a subset of PCRs (e.g., omitting those that would fail comparison) can be accepted, weakening attestation/verification guarantees.
- **Misconfiguration:** Extra/unexpected entries can slip through, causing downstream verification to fail.

Recommendation Validate the size of the PCR values array by validating the size against an expected size (or group of sizes). We believe that it is likely intended for there to be 3 PCR values.

Developer response We will be only using Intel SGX.

5.1.45 V-ORC-VUL-045: Missing cache-control header permits stale attestation responses

Severity	Info	Commit	N/A
Type	Maintainability	Status	Fixed
Location(s)	oracle-verification-backend/api/[...]/decode_quote.go:52		
Confirmed Fix At	https://github.com/venture23-aleo/oracle-verification-backend/pull/4,811d4db		

Description The DecodeQuoteHandler returns decoded attestation data with the response header Content-Type: application/json, but it fails to set cache-related headers such as Cache-Control: no-cache Without these directives, HTTP intermediaries may store and later reuse successful responses. This omission permits the response to be cached unintentionally.

Impact Stale attestation data may be served from the cache, violating data freshness guarantees. This can result in users or systems acting on outdated verification results.

Recommendation Explicitly set the Cache-Control header to ensure that neither browsers nor intermediate caches store the response. For example, configure:

Cache-Control: no-store, no-cache, must-revalidate

Developer response The developers implemented the recommended fix.

5.1.46 V-ORC-VUL-046: Multiple Typos and Inconsistencies Across Codebase

Severity	Info	Commit	N/A
Type	Maintainability	Status	Fixed
Location(s)	<ul style="list-style-type: none">▶ aleo-oracle-notarization-backend/internal/<ul style="list-style-type: none">• constants/constants.go:11• errors/errors.go:95-155, 108▶ aleo-oracle-sdk-go/notarize.go:173▶ oracle-verification-backend/attestation/encoding.go:138		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-sdk-go/pull/14		

Description Several minor but impactful issues exist in the SDK that reduce clarity, create friction in development, and could cause functional errors in certain environments. These include:

1. **NotarizationOptions comment typo** - The comment misspells "optional" as "ptional," reducing readability and documentation quality.
2. **CORS header name mismatch** - The header string is defined as "Access-Control-Request-Header" instead of the correct "Access-Control-Request-Headers", which may break preflight checks.
3. **Misspelled error constants (ErrWritting)** - Multiple error constants incorrectly use the prefix ErrWritting instead of ErrWriting.
4. **Log message truncation in prepareProofDat** - A diagnostic log message drops the trailing "a" in prepareProofData, hurting discoverability in logs.
5. **Misaligned error code for ErrAleoContext** - This error is listed under the attestation error range (3000-3999) but incorrectly reuses code 7001.

Impact

- ▶ Developers may experience confusion due to typos in comments, identifiers, and logs.
- ▶ Incorrect CORS header could lead to failed cross-origin requests, creating harder-to-diagnose client issues.
- ▶ Misaligned error codes undermine the consistency of error taxonomy and monitoring.

Recommendations

- ▶ Fix typos in comments (optional), error constants (ErrWriting), and log messages (prepareProofData).
- ▶ Correct the CORS header string to Access-Control-Request-Headers.
- ▶ Reassign ErrAleoContext to an error code.

Developer response The developers implemented the recommended fix.

5.1.47 V-ORC-VUL-047: Case-Sensitive String Comparisons in HTTP and URL Handling

Severity	Info	Commit	N/A
Type	Usability Issue	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/internal/ <ul style="list-style-type: none">▶ common/common.go:17-21, 52-66, 82, 101▶ constants/constants.go:4-54▶ services/<ul style="list-style-type: none">• attestation/attestation.go:101-191, 166• dataextraction/price_feed.go:108		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/14		

Description The codebase contains multiple case-sensitive comparisons involving HTTP headers, URLs, schemes, and hostnames. Specifically:

- ▶ IsAcceptedHeader performs an exact match between a provided header and entries in constants.AllowedHeaders, failing to normalize casing or spacing. HTTP headers are case-insensitive by specification.
- ▶ IsPriceFeedURL, ExtractTokenFromPriceFeedURL, and GetTokenIDFromPriceFeedURL compare URLs directly to constant values, ignoring potential case variations in input.
- ▶ NormalizeURL and GetHostnameFromURL use strings.HasPrefix on schemes (e.g., "http://" and "https://") in a case-sensitive manner.
- ▶ IsTargetWhitelisted directly compares hostnames from input URLs to a whitelist without lowercasing them, even though domain names are case-insensitive.

Additionally, GenerateAttestationReport rejects requests using Content-Type: application/json due to case-sensitive header comparison, and the AttestationRequest.Validate method rejects valid inputs if casing does not match predefined constants exactly.

Impact Legitimate requests may be rejected, causing denial-of-service to clients that use alternate but valid casing (e.g., Content-Type: application/json vs application/JSON). This undermines interoperability and can block integrations with external systems. Case-sensitive URL and hostname comparisons introduce inconsistencies in whitelist enforcement and price feed detection.

Recommendation Normalize all string comparisons for HTTP headers, URLs, schemes, and hostnames to a consistent case (typically lowercase) prior to comparison. Specifically:

- ▶ Convert header names to lowercase in IsAcceptedHeader.
- ▶ Normalize hostnames and schemes (e.g., lowercasing, trimming whitespace) in NormalizeURL, GetHostnameFromURL, and IsTargetWhitelisted.
- ▶ Use case-insensitive comparison for IsPriceFeedURL and similar functions.
- ▶ Ensure validation logic in AttestationRequest.Validate accounts for case insensitivity in HTTP-related fields.

Developer response The developers implemented the recommended fix by normalizing attestation requests so that fields always have a specific case so that case-insensitive comparisons may be performed.

5.1.48 V-ORC-VUL-048: HTTP success detection is limited to status code 200

Severity	Info	Commit	N/A
Type	Maintainability	Status	Acknowledged
Location(s)	<ul style="list-style-type: none">▶ aleo-oracle-notarization-backend/[...]/json_parser.go:94▶ aleo-oracle-sdk-go/request.go:157▶ aleo-oracle-sdk-js/src/<ul style="list-style-type: none">• client.ts:261• fetch.ts:41• request.ts:90, 116		
Confirmed Fix At	N/A		

Description Multiple client-side modules across SDKs and platforms treat only HTTP status code 200 as a successful response, ignoring other valid 2xx codes such as 201 Created, 202 Accepted, or 204 No Content. This includes:

- ▶ JSON extraction logic that always returns 200 OK, even when the actual response differs.
- ▶ Go SDK aborting requests for any status other than 200.
- ▶ JavaScript client and helper functions treating non-200 statuses as errors.
- ▶ Custom fetch wrapper marking only status 200 as .ok, causing the response.ok property to fail on valid 2xx responses.

This hardcoded assumption violates the HTTP standard, which defines all 2xx codes as successful outcomes. It may lead to misinterpretation of server responses and unnecessary error handling on client side.

Impact Clients may reject or mislabel legitimate server responses as failures, resulting in:

- ▶ Broken integrations when APIs return 201 Created or 204 No Content as intended.
- ▶ User-facing errors or transaction aborts despite successful server-side operations.
- ▶ Inconsistent behavior across SDKs, complicating debugging and reliability.
- ▶ Potentially blocking protocol evolution if new endpoints adopt non-200 2xx responses.

Recommendation Refactor all response handling logic to treat any status code in the 2xx range as a successful outcome. Avoid hardcoded 200 checks. Instead:

- ▶ Use standard .ok flags or numeric range comparisons (status >= 200 && status < 300).
- ▶ Ensure any JSON-parsing or error-handling logic is conditioned appropriately to support all 2xx responses.
- ▶ Add integration tests covering a representative range of 2xx statuses to prevent regressions.

Developer response We will standardize on status code 200. For valid attestation requests, the notarization backend will always return 200 in responseStatusCode, regardless of the status code received from the attestation target URL. In addition, following the audit team’s suggestion, we have added a check in the Aleo program to verify that responseStatusCode equals 200.

5.1.49 V-ORC-VUL-049: SGXReport struct field order mismatch with Intel SGX specification

Severity	Info	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/internal/[...]/report.go:57		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/8		

Description According to Intel Software Guard Extensions Programming Reference, Section 2.15, the architectural definition of the REPORT structure is as follows:

Field Name	Offset (Bytes)	Size (Bytes)	Description
CPUSVN	0	16	The security version number of the processor.
MISCSELECT	16	4	SSA Frame specified extended feature set bit vector.
RESERVED	20	28	Must be zero.
ATTRIBUTES	48	16	The values of the attributes flags for the enclave. See Section 2
MRENCLAVE	64	32	The value of SECS.MRENCLAVE.
RESERVED	96	32	Reserved.
MRSIGNER	128	32	The value of SECS.MRSIGNER.
RESERVED	160	96	Zero.
ISVPRODID	256	2	Enclave PRODUCT ID.
ISVSVN	258	2	The security version number of the Enclave.
RESERVED	260	60	Zero.
REPORTDATA	320	64	A set of data used for communication between the enclave an
KEYID	384	32	Value for key wear-out protection.
MAC	416	16	The CMAC on the report using report key.

However, the user-provided definition deviates from this standard:

```
1 type SGXReport struct {
2     Body ReportBody
3     MAC [16]byte
4     KeyID [32]byte
5 }
```

This Go structure incorrectly orders the fields, placing MAC before KeyID. The ordering of fields is **semantically significant** in SGX attestation, especially for cryptographic integrity verification. Intel’s documentation explicitly specifies the layout for compatibility and correctness in enclave attestation flows.

Impact This discrepancy in field order can break the compatibility of the report structure with:

- ▶ Intel’s SGX quoting enclave and attestation service
- ▶ Any tooling that relies on the exact byte layout (e.g., for MAC verification)
- ▶ Parsing of REPORT blobs by other enclaves or verifiers

Specifically, MAC verification will fail because the keyed message authentication code is computed over the Body using the KeyID to derive the report key. If these fields are reordered, any cryptographic validation will be invalid due to incorrect byte offsets and layout expectations.

Recommendation Align the struct definition exactly with Intel's specification: Reorder the fields to match ReportBody, followed by KeyID[32], then MAC[16].

Developer response The developers implemented the recommended fix.

5.1.50 V-ORC-VUL-050: Missing rate limiting on SGX attestation endpoints

Severity	Info	Commit	N/A
Type	Denial of Service	Status	Acknowledged
Location(s)	<ul style="list-style-type: none"> ▶ aleo-oracle-notarization-backend/[...]/random_attestation.go:26 ▶ aleo-oracle-sdk-go/notarize.go:493 ▶ aleo-oracle-sdk-js/src/client.ts:240 ▶ oracle-verification-backend/api/api.go:28-31 		
Confirmed Fix At	N/A		

Description The backend services forming the oracle infrastructure—specifically `oracle-verification-backend` and `aleo-oracle-notarization-backend`—do not implement any form of request throttling or rate limiting on endpoints that trigger SGX-based computation. The following endpoints remain exposed and unprotected:

- ▶ `oracle-verification-backend`: `/verify`, `/decode`, `/decode_quote`, `/info`
- ▶ `aleo-oracle-notarization-backend`: `/notarize`, `/random`

These endpoints perform operations such as quote decoding or SGX local attestation generation. Even without full quote verification, local attestation generation incurs a **non-trivial cost** due to enclave transitions and limited EPC memory, especially under concurrent load.

Client SDKs in both Go and JavaScript are additionally designed to **broadcast requests to all known backend nodes simultaneously**:

- ▶ Go SDK: `executeRequest()` in `client.go/request.go`
- ▶ JS SDK: `requestBackendMesh()` in `src/request.ts`

These SDK calls trigger **parallel fan-out** without backoff or coordination, compounding the resource load on the backends. Under scaled or repeated use, this behavior amplifies system stress.

Impact Any client or attacker can repeatedly hit endpoints responsible for SGX local attestation and decoding, forcing the system to incur **enclave-side costs** (e.g., `EENTER`/`EEXIT` overheads, EPC page eviction). This opens the infrastructure to denial-of-service (DoS) risks. When combined with the aggressive SDK fan-out logic, even benign users can cause degraded performance, quote generation failures, or service disruption.

Recommendation

- ▶ **Backend:** Introduce IP-based rate limiting and exponential backoff for endpoints that invoke SGX instructions and enforce authentication.
- ▶ **SDKs:** Implement configurable concurrency caps and retry strategies in the Go and JS libraries to avoid flooding all backend IPs simultaneously.

Developer response We will restrict attestation requests to trusted clients by enforcing mTLS, apply rate limiting through Nginx, and map each domain to a single backend IP.

5.1.51 V-ORC-VUL-051: Strict Content-Type check rejects valid JSON requests

Severity	Info	Commit	N/A
Type	Data Validation	Status	Fixed
Location(s)	oracle-verification-backend/api/[...]/decode.go:54-56		
Confirmed Fix At	https://github.com/venture23-aleo/oracle-verification-backend/pull/2 , ec95858		

Description The request handlers validate the Content-Type header by comparing it for strict equality with "application/json". While this guards against malformed or unexpected media types, it rejects otherwise valid JSON requests that include parameters, such as "application/json; charset=utf-8". As a result, clients following common conventions-particularly those specifying character encoding-receive unnecessary errors.

Impact By disallowing JSON requests that include encoding parameters, the API becomes less interoperable. Clients built on standard libraries often append a charset parameter automatically; these requests would be rejected despite conforming to the JSON media type specification. This behavior can lead to failed integrations, frustrated users, and wasted troubleshooting time.

Recommendations

1. Relax the strict equality for "application/json".

Developer response The developers implemented the recommended fix.

5.1.52 V-ORC-VUL-052: Misconverted HTTP status codes in metrics label

Severity	Info	Commit	N/A
Type	Maintainability	Status	Fixed
Location(s)	aleo-oracle-notarization-backend/[...]/metrics.go:192		
Confirmed Fix At	https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/12 , bd76d79		

Description RecordHttpRequest converts an integer status code to a string using `string(rune(statusCode))`, which produces a single Unicode character instead of the numeric representation

Impact Metrics for HTTP status codes will be labeled with unintended characters, leading to misreported or conflated metrics and making debugging or monitoring difficult.

Recommendation Convert the status code to its decimal string form with `strconv.Itoa(statusCode)` and ensure the `strconv` package is imported.

Developer response The developers fixed conversion of status code according to recommendations.

EGo An open-source framework that enables Go applications to run inside Trusted Execution Environments (TEEs), such as Intel SGX. EGo provides a runtime and supporting tools that allow developers to build enclave-protected applications in Go with minimal changes to their existing codebases, combining memory safety guarantees of Go with the confidentiality and integrity protections of TEEs . [10](#)

Gramine An open-source library operating system designed to run unmodified applications inside Trusted Execution Environments (TEEs) such as Intel SGX. Gramine provides a lightweight framework that supplies the system call interface expected by applications while enforcing isolation and confidentiality guarantees from the underlying TEE . [1](#), [10](#)

LFENCE A memory-ordering instruction used in x86 architectures to serialize load operations. LFENCE ensures that all load instructions specified before it in program order are completed before any subsequent load instructions are executed. It is often used as a barrier to mitigate speculative execution attacks (such as Spectre) by preventing the CPU from speculatively reordering memory reads across the fence . [2](#)

TEE Trusted Execution Environment. A secure area within a main processor that ensures code and data loaded inside are protected with respect to confidentiality and integrity. TEEs provide guarantees that sensitive computations cannot be tampered with or observed, even by privileged software such as an operating system or hypervisor . [1](#)

wazero An open-source WebAssembly runtime written in Go. Wazero allows applications to run WebAssembly (Wasm) modules without needing CGO or external dependencies, making it lightweight and easily embeddable. It supports running unmodified Wasm modules in a secure, sandboxed environment directly inside Go programs . [10](#)