ZKSECURITY

Audit of Venture23 Aleo Oracle

Date: August 11th, 2025

Introduction

On August 11th, 2025, zkSecurity was engaged to perform a security audit of Venture23 Aleo Oracle. The specific code to review was extracted from various public repositories of Venture23 (https://github.com/venture23-aleo). The audit lasted 3 weeks with 2 consultants.

Scope

The scope of the audit included the following repositories with the respective commit hash:

- **aleo-oracle-notarization-backend**: `137dbd0cd9b2cc8e0f6a46142407b875127b07ff`
- **oracle-verification-backend**: `9f6285dd70472bf893c66b919963aa696614cd19`
- **aleo-oracle-encoding**: `d4385824268369695 9759ac7fc300479dffa1120`
- **aleo-utils-go**: `78a47ddecc7a7e34a769b90b94f2d85053ce7ff6`
- **aleo-oracle-sdk-js**: `c150794f24e98874c4972d81aa9e1b1cb6c577b1`
- **aleo-oracle-sdk-go**: `0a82cfaf43c5b33eea4d25550b58d4590a96f5f7`
- **aleo-oracle-contracts**: `d709d0eb0b24af0a725e1287eeb742b1d76b0f31`

For **aleo-oracle-contracts**, only the file `vlink_oracle_v0001.aleo` was included in scope. Components related to **AWS Nitro** in the repositories were considered **out-of-scope** and not reviewed during this audit, as the assessment focused solely on Intel SGX–based components.

Methodologies

The security assessment employed a systematic two-phase approach designed to evaluate both the overall system architecture and individual component security properties. The methodology was structured to understand the complete trust model and identify potential attack vectors across the entire oracle pipeline, using the price feed workflow as a primary reference implementation to map relationships between components and understand the trust boundaries, data transformation points, and critical dependencies.

The first phase focused on the core infrastructure components, examining the notarization backend, verification backend, and supporting utilities. This analysis evaluated the security properties of the protocol design, assessing whether the composition of various parties achieves the intended security guarantees through proper cryptographic proofs and attestation mechanisms. The team studied how the split-verification architecture addresses Aleo's platform limitations while maintaining security properties, with particular attention to

trust relationships between components and potential failure modes in the verification chain.

The second phase conducted detailed analysis of the SDK and client-side components, examining how applications integrate with the oracle infrastructure and identifying potential vulnerabilities in the client interaction layer. This included analysis of the Go and JavaScript SDKs, smart contract interfaces, and the overall client experience to ensure that security properties are maintained throughout the complete user interaction flow.

Throughout both phases, each component was examined for implementation vulnerabilities, configuration weaknesses, and operational risks focusing on three primary threat categories: integrity of the attestation process, confidentiality of cryptographic keys involved in the system, and potential availability issues that could disrupt oracle operations. For identified potential issues, the team engaged in collaborative discussions with the client to understand the intended behavior and operational constraints. Where appropriate, proof-of-concept demonstrations were developed to confirm the exploitability and impact of discovered vulnerabilities. All findings were then classified using a severity framework that considers both the likelihood of exploitation and the potential impact on the oracle's security properties, enabling prioritized remediation efforts.

Strategic Recommendations

The Aleo Oracle system demonstrates a well-engineered approach to bridging trusted execution environments with blockchain-based data verification, but its current architecture reveals fundamental trust assumptions that can potentially compromise the security of the oracle. The most significant architectural limitation stems from Aleo's inability to perform native ECDSA signature verification, which forces critical SGX attestation validation to occur off-chain in the verification backend. This design creates a trust dependency that impacts the security model, as the smart contract can only validate Aleo signatures while trusting that proper SGX verification has already occurred.

The current split-verification architecture represents a pragmatic compromise given Aleo's limitations, but it introduces several systemic risks. The verification backend becomes a single point of trust and failure, potentially allowing compromised or malicious verifiers to inject invalid attestations into the system. Additionally, the off-chain verification process lacks the transparency and auditability that blockchain-based systems typically provide, making it difficult to detect or prove verification malfeasance after the fact.

Although the likelihood of such attacks is relatively low (it requires an attacker's control of an Aleo private key corresponding to a trusted public key in the contract), the impact is very high, since it breaks the fundamental trust premise of the system. In order to strengthen the architectural foundations of the oracle system, development should focus on reducing and eventually eliminating these trust assumptions. Short term, is advised to review the architecture such that verifiers sign the outcome of a given verification and contracts can check this against a list of trusted verifiers. Medium term, the contract

should implement ECDSA verification and the system should be able to work without the verification backend, as it is already acknowledged in the project documentation.

Overview

Architecture

The Aleo Oracle system provides cryptographically verifiable external data to Aleo smart contracts using Trusted Execution Environments (TEEs). The architecture addresses Aleo's current limitations while aiming at maintaining strong security guarantees through hardware-backed attestation.

Core Data Flow: The core data-flow can be summarized as follows: A TEE Backend receives a request to notarize the response of a given URL (i.e. BTC to USD price according to a given exchange or list of exchanges). This response is notarized in two ways: an Aleo private key (that is generated in the TEE enclave) signs the attestation report, which is also signed by the TEE hardware. A client uses this attestation to update the state of an Aleo contract. Given that Aleo cannot directly verify the hardware attestation (ECDSA signatures), the client calls a verifier backend that checks the TEE signature. The contract then checks the Aleo signature (verifying also that the verification public key is in a list of allowed keys).

The system consists of five primary components:

1. *Notarization Backend* (`aleo-oracle-notarization-backend`): Runs inside Intel SGX enclaves using Gramine LibOS to fetch external data and generate cryptographically signed attestation reports. The TEE isolation ensures data integrity and prevents tampering during the critical data acquisition phase.

2. *Verification Backend* (`oracle-verification-backend`): Validates SGX attestation reports and ECDSA signatures outside the TEE environment. This component serves as a bridge until Aleo supports native ECDSA verification, converting complex cryptographic proofs into simpler Aleo-compatible signatures.

3. *Smart Contract* (`aleo-oracle-contracts/programs/vlink_oracle_v0001.leo`): Implements on-chain verification and data storage with blockchain-enforced security. The contract manages enclave measurements, authorized keys, and final data validation using Aleo's native signature schemes.

4. *SDKs* (`aleo-oracle-sdk-go`, `aleo-oracle-sdk-js`): Client libraries (Go/JavaScript) that provide application integration points, handling network communication and request formatting. These can be orchestrated by automated infrastructure such as

cronjobs (e.g., the `aleo-oracle-gateway` service) to manage periodic price feed updates.

5. *Supporting Libraries* (`aleo-oracle-encoding`, `aleo-utils-go`): Including encoding utilities for Leo language compatibility and WASM wrappers for Aleo cryptographic functions.

BTC Price Attestation Example

Consider a typical Bitcoin price update flow:

1. *Data Acquisition*: Upon request by a client, the notarization backend, running within an SGX enclave, makes HTTPS requests to multiple cryptocurrency exchanges (e.g., Binance, CoinGecko) to fetch current BTC/USD prices.

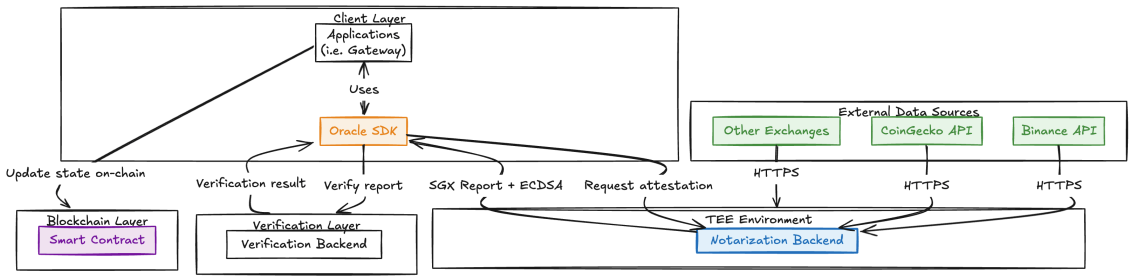
2. *Processing & Signing*: The enclave aggregates the price data, applies any configured transformations (averaging, precision scaling), and signs the result using its private key along with SGX attestation evidence proving the computation occurred within a genuine Intel SGX environment.

3. *Attestation Submission*: Automated infrastructure (a client, such as the gateway service) retrieves the signed price attestation and submits it to the verification backend for validation.

4. *External Verification*: The verification backend validates the SGX attestation report, verifies the ECDSA signature and returns the result of the verification process.

5. *On-Chain Storage*: The verified price attestation is submitted to the Aleo smart contract, which performs final validation checks and stores the BTC price for use by other Aleo applications.

This entire flow aims at ensuring that price data cannot be manipulated by compromised infrastructure, as the critical computation and signing occurs within the hardware-protected SGX enclave.



Gramine Library OS

The notarization backend uses the Gramine framework to run the service inside an Intel SGX enclave for processing attestation requests. Gramine is a lightweight Library Operating

System that runs largely unmodified Linux applications (including Go binaries) inside SGX. It provides several useful capabilities for the oracle system:

- **POSIX compatibility**: Enables standard networking, file I/O, and common cryptographic libraries to work inside SGX without application rewrites.
- **Secure channels**: TLS libraries run *inside* the enclave; for inbound use cases Gramine offers RA-TLS helpers to bind an SGX quote to a TLS session. For outbound calls, the service performs normal HTTPS from within the enclave boundary.
- **Attestation integration**: Provides APIs and tooling (e.g., RA-TLS) to obtain SGX quotes and expose measurements; the application is responsible for binding keys, source identifiers, and freshness into attested payloads.

By using Gramine, the notarization backend can reuse existing HTTP clients, JSON parsers, and crypto libraries while benefiting from SGX's hardware-backed isolation and remote attestation. This reduces development complexity compared to the native SGX SDK, with the trade-off of a larger trusted computing base (the LibOS becomes part of the enclave TCB).

Notarization Backend

The notarization backend is the core component of the Aleo Oracle, responsible for processing attestation requests. It runs on the Gramine framework inside an Intel SGX enclave, fetching oracle data from external sources over TLS and generating an SGX quote to enable remote attestation.

According to the official architecture documentation in the repository, the workflow can be summarized as follows:

1. The backend runs inside Intel SGX enclave using Gramine. It generate enclave signing key, sign the manifest with the key, and loads the Gramine LibOS and the backend binary. This enclave will get a `MRENCLAVE` measurement hash, proving the integrity of the initial state of the enclave. Since the build is reproducible, anyone can independently verify the hash by rebuilding the backend via Docker.
2. On startup, the backend generates Aleo key pair. The private key remains inside the enclave, while the public key is registered on the Aleo smart contract.
3. Users submit requests to the backend. The backend verifies the URL whitelist and payload format, then fetches the target data to be attested via HTTPS.
4. The backend builds an oracle report containing the request details, timestamp, and attestation data, then encodes it in Aleo-specific format.
5. The backend produces two hashes using Poseidon8. First, it computes the `requestHash`, which is the hash of the Aleo-encoded request (with zeroed timestamp). Then, it derives the `timestampedRequestHash` by hashing the `requestHash`

together with the attestation timestamp. These hashes are later used in the SGX quote generation process.

6. The enclave generates an SGX report. The quoting enclave signs it along with the attestation hash and PCK chain, while the provisioning enclave assists with cryptographic key setup.
7. The Aleo-encoded oracle report is produced, hashed, and signed with the Aleo private key. The backend outputs both the attestation report and the Aleo oracle data for verification.

Verification Backend

The verification backend is responsible for validating SGX remote attestation reports produced by the notarization backend. Its sole function is to verify SGX reports, so it does not require a trusted execution environment (TEE) and can be operated by any party. The component leverages the Ego library for attestation verification and relies on a Platform Certificate Caching Service (PCCS) server to validate SGX reports.

At a high level, the verification backend operates as follows:

1. The backend loads the configuration containing the `uniqueId`, which corresponds to the `MRENCLAVE` of the enclave being verified.
2. Users submit attestation requests, which include the SGX report and Aleo-encoded data.
3. The backend uses Ego to verify the report, extracting metadata and enclave measurements. The attestation is accepted if the `uniqueId` matches the configured value.
4. The backend ensures integrity of the oracle data by recomputing the Poseidon8 hash and comparing it against the value embedded in the attestation. A match indicates that the data has not been tampered with.

Oracle SDK

The Aleo oracle provides an SDK to integrate its components into user applications. Currently, SDKs are available in TypeScript and Go. Under the hood, the SDK acts as an HTTP request wrapper, handling communication with both the notarization and verification backends.

The SDK exposes a `Client` class, which provides the following public methods:

- **Notarize**: The primary function to submit a request to the oracle.
- **EnclavesInfo**: Retrieves enclave information, including cryptographic details such as `uniqueId` and the Aleo public key.
- **GetAttestedRandom**: Requests a attested random number from the oracle.

- **TestSelector**: Allows testing of requests without performing attestation or verification.

Note that a single call to `Notarize` or `GetAttestedRandom` also triggers verification by sending the attestation to the verification backend.

Oracle Smart Contract

The Aleo Smart Contract (`vlink_oracle_v0001.aleo`) serves as the final trust anchor in the oracle system, providing cryptographically-enforced data storage and integrity guarantees on the blockchain. The contract maintains critical configuration state including authorized SGX enclave measurements (`UniqueID`) and a whitelist of allowed signing keys, with owner-only functions ensuring that only trusted TEE instances can contribute data to the system. When the SDK orchestrator calls the `set_data_sgx` function, the contract performs several validation steps: it verifies that the signing key is authorized, checks that the SGX `UniqueID` from the attestation report matches the stored trusted measurement, and validates that the provided data hash was correctly included in the TEE-signed report using Poseidon8 cryptographic hashing.

However, the contract cannot perform full SGX attestation verification due to Aleo's current limitation of not supporting ECDSA signature verification natively. Instead, it relies on the verification backend to have already validated the ECDSA signatures and SGX attestation reports before the data reaches the contract. The contract only performs Aleo-native signature verification on the final report hash that was signed by the verification backend, effectively creating a trust chain where SGX ECDSA validation occurs off-chain in the verification backend, and Aleo signature validation occurs on-chain in the contract. Once validation is complete, the contract stores the attested data with timestamps and implements versioning logic to maintain only the latest data per request type, ensuring that applications consuming oracle data have access to the most recent verified information while maintaining full cryptographic auditability of the trust chain from TEE to blockchain.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	aleo-oracle-notarization-backend	Insecure Source of Timestamp	High
#01	aleo-utils-go	Memory Capacity Mismatch in Rust-Go Boundary	High
#02	aleo-oracle-contracts	SGX Attestation Bypass via Compromised Aleo Private Keys	High
#03	aleo-oracle-notarization-backend	Excessive CA Trust Store	Medium
#04	aleo-oracle-encoding	Float64 Precision Loss in Price Feed Pipeline	Medium
#05	aleo-oracle-sdk-go	Multiple Runtime Panic Due To Missing Nil Checks	Medium
#06	oracle-verification-backend	Missing MRSIGNER and Debug Mode Validation in SGX Verification	Medium
#07	aleo-oracle-notarization-backend	Domain Whitelist Bypass via Unchecked URL Redirection	Low
#08	oracle-verification-backend	Incomplete TCB Status Validation	Low

ID	COMPONENT	NAME	RISK
#09	aleo-oracle-notarization-backend	SGX-Aleo Key Binding Vulnerability Enables Cross-Enclave Impersonation	**Low**
#0a	aleo-oracle-sdk-go, aleo-oracle-sdk-js	Unbounded JSON Response Parsing in Oracle SDKs	**Low**
#0b	*	Code Quality and Minor Corrections	**Informational**
#0c	aleo-oracle-notarization-backend	Unattested Response Body	**Informational**

#00 - Insecure Source of Timestamp

Severity: High **Location:** aleo-oracle-notarization-backend

Description.

The current implementation sources the timestamp in the SGX report from `time.Now().Unix()`, which in turn relies on the OS clock. This approach is insecure, because as documented in both the Intel whitepaper and the Gramine documentation, the host OS outside the enclave has full control over the date/time values.

As a simple proof of concept, when running the application inside an SGX enclave, the host can manipulate the OS clock easily. For example:

```
sudo timedatectl set-ntp false
sudo timedatectl set-time "2030-12-12 00:00:00"  # set to an arbitrary
future datetime
```

This causes the enclave to receives a host-controlled timestamp, not a trusted value, while still maintaining the attestation integrity. Note that Gramine **only** includes a safeguard against moving backwards in time: if the enclave fetches a timestamp earlier than the previous one, it will immediately abort and exit as can be seen in the source here.

The timestamp is used as critical data in the Aleo smart contract to ensures the freshness of the attestation. If an adversary can arbitrarily manipulate the timestamp, they could set it to a future value, causing data that should only reflect the present state to persist indefinitely within the contract, as seen in the following contract code:

```
// replace latest data if current data is newer
let latest_data: AttestedData = Mapping::get_or_use(sgx_attested_data,
request_hash, AttestedData { data: 0u128, attestation_timestamp: 0u128
});

if (attested_data.attestation_timestamp >
latest_data.attestation_timestamp) {
    Mapping::set(sgx_attested_data, request_hash, attested_data);
}
```

Recommendation.

The system should obtain trusted time information from an authenticated and verifiable source. Two possible approaches include:

- Leverage trusted date/time values from signed HTTP headers in the same request of the attested data (e.g., `Date` header from a TLS-secured response), where the

authenticity of the response is tied to the server's certificate chain. This is the simplest approach, but HTTP headers might not contain this data for all websites.

- Integrate a signed time protocol such as Roughtime or equivalent, which provides cryptographic assurance on the freshness and correctness of time data. This approach offers stronger guarantees, but requires additional trust assumption, infrastructure, and integration effort.

Client Response. The issue was fixed in https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/2 by integrating Roughtime server

#01 - Memory Capacity Mismatch in Rust-Go Boundary

Severity: High **Location:** aleo-utils-go

Description.

The aleo-utils-go library contains a critical memory safety vulnerability in the FFI boundary between Rust and Go code. The `dealloc` function in `memory.rs` assumes the Go caller tracks the exact capacity of Rust vectors, but this assumption is violated in multiple locations, most critically in the `HashMessage` function.

The vulnerability stems from a fundamental mismatch in how memory is allocated versus deallocated:

1. **Allocation (memory.rs:21-24)**: `Vec::with_capacity(capacity)` may over-allocate more memory than requested due to allocator optimizations, power-of-2 rounding, and alignment requirements.

2. **Deallocation (session.go:194,266,339,410)**: The Go code calls `s.deallocate.Call()` using output lengths instead of original allocation sizes. Critical instances include `FormatMessage` (line 194), `RecoverMessage` (line 266), and both hash functions (lines 339,410) where 16-byte hash outputs are used to deallocate much larger input allocations.

3. **Unsafe Reconstruction (memory.rs:29)**: `Vec::from_raw_parts(pointer.cast_mut(), 0, capacity)` creates a Vec with incorrect capacity, violating Rust's safety guarantees.

Impact.

This creates a deterministic vulnerability where:

- Large input messages (e.g., 1000 bytes) trigger allocation of 1000+ bytes
- Hash output length (16 bytes) is used for deallocation
- `Vec::from_raw_parts` reconstructs the Vec with wrong capacity (16 vs 1000+ bytes)
- According to Rust documentation, this is undefined behavior that can cause heap corruption

Dynamic memory analysis confirms this vulnerability causes measurable heap corruption. Testing with `HashMessage` operations using progressively larger inputs (64B to 16KB) demonstrated capacity mismatches ranging from 9.75x to 309.5x, with over 95% of allocations not properly freed due to size mismatches. Process memory monitoring showed +2MB growth per session and heap fragmentation exceeding 85% during stress testing.

These results validate that the theoretical vulnerability manifests as concrete memory behavior degradation.

In the SGX enclave environment, this vulnerability poses risks to module stability and service availability through allocator corruption leading to crashes and re-keying operational overhead.

Recommendation.

It is recommended to implement one of these two approaches:

- Modify the `alloc` function to return both the pointer and actual allocated capacity (e.g., packed into a single u64 on wasm32). The `dealloc` function then receives the exact capacity that was actually allocated, eliminating any mismatch. This requires minimal Go code changes but fixes the root cause by making capacity explicit in the ABI.
- Store the actual Vec capacity in an 8-byte header immediately before the returned pointer. The `alloc` function allocates extra space for this header, while `dealloc` reads the header to determine the correct capacity. This approach requires no changes to Go calling code and eliminates capacity tracking entirely from the caller's perspective.

Client response. Client acknowledged the issue and fixed it in https://github.com/venture23-aleo/aleo-utils-go/pull/1/.

#02 - SGX Attestation Bypass via Compromised Aleo Private Keys

Severity: High **Location:** aleo-oracle-contracts

Description.

The smart contract's `set_data_sgx` function (`vlink_oracle_v0001.leo:318-374`) implements an indirect trust model where SGX attestation verification is bypassed at the contract level. The contract only verifies that the Aleo signature is valid and the signing key is in the `allowed_keys` mapping (line 358-359), without directly validating the SGX attestation proof. This creates a critical trust assumption: "if Aleo signature is valid ⟺ signature was produced inside trusted SGX enclave."

However, this assumption breaks if an Aleo private key is ever compromised through side-channel attacks, cloud infrastructure compromise, or other key extraction methods, or if the list of trusted public keys is maliciously manipulated.

Impact.

An attacker with a compromised private key corresponding to an allowed public key can completely bypass SGX attestation verification by:

1. Crafting arbitrary fake `report_data` and `report` structures
2. Signing the fake report with the compromised Aleo key
3. Calling `set_data_sgx` directly with the fabricated data

The contract will accept this fake data as authentic SGX-attested information since it only checks the Aleo signature validity, not the underlying SGX attestation proof. The `verify_sgx_report` function (lines 272-299) validates the report structure and signature but crucially assumes the signature comes from a legitimate SGX enclave, which is not guaranteed if the key is compromised.

This vulnerability is particularly concerning for cloud-deployed SGX enclaves where residual risks include hardware side-channels, hypervisor-level attacks, and cloud provider compromise scenarios. It also increase the risk of attacks in case the list of trusted public keys is maliciously manipulated (i.e. by an attack against the contract admin).

Recommendation.

To strengthen the oracle's security model, the system should implement a more robust verification architecture that explicitly validates SGX attestation rather than relying solely on Aleo signature verification. A trusted verifier model can address this limitation by deploying dedicated off-chain verifiers that perform comprehensive validation of both SGX

attestation proofs and accompanying Aleo signatures. These verifiers would then cryptographically sign their verification results, allowing the smart contract to verify verifier signatures against a maintained list of trusted verifier public keys. This approach creates an explicit chain of cryptographic evidence from SGX attestation through verification to on-chain storage.

Complementing this approach, the contract can implement a restricted caller model that limits `set_data_sgx` function calls exclusively to trusted client addresses. This architectural change ensures that all data submissions must pass through proper attestation verification channels, eliminating the possibility of bypassing SGX validation through direct contract calls with compromised Aleo keys.

As acknowledged in the project documentation, ideally the system should explore implementing direct SGX quote verification on-chain if technically feasible within Aleo's constraints. This would eliminate the indirect trust dependency entirely by bringing all cryptographic verification onto the blockchain itself.

Client response. Client has acknowledged the issue.

#03 - Excessive CA Trust Store

Severity: Medium **Location:** aleo-oracle-notarization-backend

Description.

The SGX enclave trusts the default Mozilla CA bundle from the `gramineproject/gramine:stable-jammy` base image, which includes approximately 145 root Certificate Authorities. The Gramine manifest template mounts the entire system CA trust store (`/etc/ssl/` and `/usr/lib/ssl/`) into the enclave, creating an unnecessarily broad attack surface for man-in-the-middle attacks against exchange APIs.

Impact.

If any of these 145 trusted CAs were compromised, an attacker could issue valid TLS certificates for target exchange domains (e.g., `api.binance.com`, `api.coinbase.com`) and perform MITM attacks against the oracle's data fetching operations. While CA compromise incidents are relatively rare, they have occurred historically and represent a significant risk given the financial nature of oracle data.

The current trust model treats all CAs equally, despite the fact that exchange APIs typically use certificates from a small number of well-established CAs. For example:

- Binance typically uses DigiCert certificates
- Coinbase uses Google Trust Services
- Other major exchanges have similarly predictable CA relationships

This broad trust store violates the principle of least privilege and unnecessarily exposes the oracle to supply chain attacks via the PKI ecosystem.

Recommendation.

Adopt per-domain CA pinning by restricting each whitelisted endpoint to the specific root CA it actually uses. For example, trust only DigiCert for Binance APIs and Google Trust Services for Coinbase. Build a minimal trust store that contains just those required roots instead of the full Mozilla bundle, and load it at runtime.

Operationally, it is recommended to pair this with a light CA auditing process: periodically verify that each exchange hasn't rotated to a new certificate provider and refresh the pinned roots when they do (these changes are normally rare). This design reduces exposure from ca. 145 broadly trusted roots to a handful that are actually needed, while preserving operational flexibility.

Client response. Client has acknowledged the issue and fixed it in https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/18.

#04 - Float64 Precision Loss in Price Feed Pipeline

Severity: Medium **Location:** aleo-oracle-encoding

Description.

The price feed processing pipeline implements a conversion chain that introduces silent precision loss through IEEE-754 float64 arithmetic: `string → big.Float → ×10^precision → float64 → uint64 → bytes → Aleo u128`. The critical precision loss occurs at `encoding.go` lines 320-321 where `big.Float` values are converted to `float64` before final `uint64` conversion.

IEEE-754 float64 has inherent precision limitations that cause non-intuitive rounding behavior:

- Numbers like `123.456789012345` become `123.456789012344998...` when represented as float64
- Large integers lose precision (e.g., `9007199254740993` becomes `9007199254740992`)
- The mantissa has only 53 bits of precision, limiting exact integer representation to 2^{53}

Impact.

This precision loss may critically impact the accuracy of the attested data in some situations. While in the `price_feed` use case, current USD price feeds may mask this issue due to relatively small magnitudes and low precision requirements, the vulnerability becomes critical for:

- High-precision cryptocurrency values (e.g., satoshis, wei)
- Large-magnitude values approaching float64 limits
- Any scenario requiring exact decimal representation

The conversion from `big.Float` (arbitrary precision) to `float64` (limited precision) at line 320 constitutes an unnecessary precision bottleneck, as the subsequent `uint64` conversion could be performed directly from `big.Float` without intermediate float64 representation.

Recommendation.

Replace the floating point conversion path with a fixed point approach. Instead of converting `string` to `big.Float` to `float64` to `uint64`, parse the string into `big.Rat`, scale by ten to the power of the desired precision, perform a range check, then convert to `big.Int` and finally to `int64` or `uint64`. Remove the dependency on `float64` at lines 320 to 321 by using `big.Float.Int` or, preferably, operating with `big.Rat` directly so the pipeline remains entirely integer based once scaled.

Update the validation logic at lines 323 to 335 to compare `big.Int` values rather than `float64` results, and add explicit range checks to ensure the scaled amount fits within `uint64` before conversion. This preserves exact decimal precision end to end and avoids IEEE 754 rounding artifacts, reducing the risk of silent data corruption for high precision or large magnitude inputs.

Client Response. Client has acknowledged the issue and fixed it in PRs https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/30 and https://github.com/venture23-aleo/aleo-oracle-encoding/pull/4/files.

#05 - Multiple Runtime Panic Due To Missing Nil Checks

Severity: Medium **Location:** aleo-oracle-sdk-go

Description.

The Go SDK implements `executeRequest` as a generic helper responsible for dispatching HTTP requests to the notarization and verification backend services. The function then collects the successful response via a result channel called `resChan`, or aggregates errors from all attempts if none succeed into error channel called `errChan`.

But in the current implementation, there are multiple functions that called the `executeRequest` function without checking if `resChan` is `nil`, as seen in the one example below:

```
var info []*EnclaveInfo
for enclaveUrl, resChan := range resChanMap {
    enclaveInfo := <-resChan
    enclaveInfo.EnclaveUrl = enclaveUrl
    info = append(info, enclaveInfo)
}
```

This way, if `resChan` is `nil`, `enclaveInfo.EnclaveUrl` will trigger `nil` dereferencing. This causes any error captured in the `executeRequest` to crash the parent function and leads to runtime panic.

The following functions in `aleo-oracle-sdk-go` are affected:

- `GetAttestedRandom`
- `GetEnclavesInfo`
- `TestSelector`
- `verifyReports`

Recommendation.

Implement `nil` check for every variable that consume the result channel from `resChan`, as already implemented in the `createAttestation` function:

```
for enclaveUrl, resChan := range resChanMap {
    resp := <-resChan
    if resp != nil {
        resp.EnclaveUrl = enclaveUrl
        result = append(result, resp)
    }
}
```

Client Response. The issue was fixed in https://github.com/venture23-aleo/aleo-oracle-sdk-go/pull/4

#06 - Missing MRSIGNER and Debug Mode Validation in SGX Verification

Severity: Medium **Location:** oracle-verification-backend

Description.

The `VerifySgxReport` function (sgx.go:12-26) performs minimal validation by only checking the UniqueID (MRENCLAVE) against the expected value. Two critical security validations are missing that could allow malicious or compromised enclaves to bypass attestation controls:

Missing MRSIGNER Validation: The function does not validate the SignerID (MRSIGNER), which identifies the cryptographic key used to sign the enclave. Without this check, an attacker could create an enclave with identical code (same MRENCLAVE) but signed with a different key. While the contract's Aleo public key validation should catch unauthorized enclaves, MRSIGNER validation provides essential defense-in-depth protection against:

- Scenarios where Aleo private keys are compromised
- Cases where malicious public keys are mistakenly added to `allowed_keys`
- Supply chain attacks where legitimate enclave code is re-signed by unauthorized parties

The Docker build process shows that enclaves are signed with a specific private key (`/tmp/private-key.pem`), but this signing relationship is not enforced during verification.

Missing Debug Mode Check: The function does not examine the `report.Debug` flag to reject debug enclaves. Debug enclaves lack fundamental SGX security guarantees:

- Debug enclaves can be inspected and modified at runtime
- Attestation of debug enclaves provides no meaningful security assurance
- Production systems should never accept attestations from debug enclaves

An attacker could potentially run a debug version of the enclave to manipulate attested data while still producing valid SGX reports that pass the current verification logic.

Recommendation.

Enhance the SGX verification function with additional security checks:

1. **MRSIGNER Validation**: Add validation of `report.SignerID` against a trusted signer identifier derived from your Gramine RSA signing key. Store the expected MRSIGNER value in configuration and reject reports from enclaves signed by unauthorized keys.

2. **Debug Mode Rejection**: Add an explicit check for `report.Debug == false` and reject any reports from debug enclaves in production environments.

3. **Configuration Management**: Store trusted MRSIGNER values in the verification backend configuration alongside the existing UniqueID parameter.

Client response. Client has acknowledged this finding and partially addressed it in PR https://github.com/venture23-aleo/oracle-verification-backend/pull/6 by adding debug check validation. They will not add MRSIGNER validation for the time being.

#07 - Domain Whitelist Bypass via Unchecked URL Redirection

Severity: Low **Location:** aleo-oracle-notarization-backend

Description.

The application's oracle currently follows HTTP redirects automatically. This behavior introduces a security risk when the whitelisted domain is vulnerable to open redirects (e.g., `httpbin.org/redirect-to?url=https://evil.com`). This way, an attacker can abuse this to redirect requests from the trusted domain to an arbitrary external domain.

This could bypass the whitelist entirely, and in extreme cases, allow arbitrary Server-Side Request Forgery (SSRF), enabling access to internal services or sensitive metadata endpoints.

Recommendation.

If redirect following is necessary, enforce same-site redirects only by validating that the final redirect destination's scheme and domain exactly match the originally requested whitelisted domain. Reject any redirects to different domains, subdomains, or schemes.

Client Response. The client has acknowledged the issue and decided to exclude all whitelisted domains, focusing only on the price feed requests.

#08 - Incomplete TCB Status Validation

Severity: Low **Location:** oracle-verification-backend

Description.

In the Ego library, the `VerifyRemoteReport` function currently returns **OK** when the Intel SGX `TCBStatus` is `UpToDate` or `SWHardeningNeeded`. Meanwhile, the function returns `ErrTCBLevelInvalid` on the following other statuses:

- `ConfigurationNeeded`
- `ConfigurationAndSWHardeningNeeded`
- `OutOfDateConfiguration`
- `Revoked`

The current implementation of `VerifySgxReport` always accept **OK** and reject `ErrTCBLevelInvalid`, regardless of the `TCBStatus`:

```
func VerifySgxReport(reportBytes []byte, targetUniqueId string)
(*attestation.Report, error) {
    report, err := eclient.VerifyRemoteReport(reportBytes)
    if err != nil {
        return nil, err
    }
    ...
}
```

This means that statuses such as `ConfigurationNeeded` or `ConfigurationAndSWHardeningNeeded`, which might be acceptable under certain policies, are treated the same as invalid or insecure status. As a result, the application loses the ability to make fine-grained decisions about which `TCBStatus` values are acceptable depending on the security policy.

Recommendation.

Update the implementation to explicitly evaluate the returned `TCBStatus` depending on the clear security policy. If the policy desires maximum security, the implementation should only accept `UpToDate` on **OK**. Meanwhile, if the policy desires flexibility, the implementation may tolerate `ConfigurationNeeded` or `ConfigurationAndSWHardeningNeeded`.

Client Response. The issue was fixed in https://github.com/venture23-aleo/oracle-verification-backend/pull/18

#09 - SGX-Aleo Key Binding Vulnerability Enables Cross-Enclave Impersonation

Severity: Low **Location:** aleo-oracle-notarization-backend

Description.

The oracle system relies on two separate cryptographic keys: an SGX attestation key that signs the quote proving the measured enclave produced the report data, and an Aleo keypair generated inside the enclave whose public key serves as the on-chain address for signature verification. Currently, the SGX-attested payload does not cryptographically commit to the Aleo public key, while the smart contract (`vlink_oracle_v0001.leo:358-359`) only verifies that the provided signature matches a public key in the `allowed_keys` mapping.

Impact.

This separation enables cross-enclave impersonation: any holder of an allowed Aleo key can take SGX attestation data produced by another enclave, remove the original Aleo signature, and re-sign it with their own key before submitting to the contract. The SGX quote still attests to the authentic response data (maintaining data integrity), but attribution becomes malleable, allowing one enclave to claim credit for work performed by another.

For example, if Enclave A fetches BTC price data and produces an SGX attestation, Enclave B can intercept this attestation, strip the Aleo signature, re-sign with its own allowed key, and submit the data as if it performed the original work. The contract cannot distinguish which enclave actually produced the attested data.

Recommendation.

Bind the Aleo public key into the SGX report data and verify this binding on-chain to cryptographically link TEE attestation with Aleo identity. This follows established patterns in SGX-TLS where certificate keys are bound to enclave attestation (https://arxiv.org/pdf/1801.05863). Modify the attestation process to include the Aleo public key in the data committed to by the SGX quote, then update the smart contract verification logic to check this binding alongside existing validations.

Client response. Client has acknowledged the issue.

#0a - Unbounded JSON Response Parsing in Oracle SDKs

Severity: Low **Location:** aleo-oracle-sdk-go, aleo-oracle-sdk-js

Description.

The Go and JavaScript Oracle SDKs perform unbounded JSON parsing of HTTP responses from backend services without implementing size limits or resource constraints. The Go SDK uses `json.Unmarshal` directly on response bodies in `request.go`, while the JavaScript SDK calls `resp.json()` in `request.ts:109`, both of which will attempt to parse arbitrarily large payloads into memory.

This creates a resource exhaustion vulnerability where compromised (or misconfigured, malfunctioning backend services) can return extremely large JSON responses that consume all available client memory and crash gateway processes.

Impact.

The vulnerability is particularly concerning in automated gateway deployments that manage periodic price feed updates, as a single oversized response could disrupt the entire oracle attestation pipeline. Attack vectors include compromised notarization or verification backends returning large payloads, network-level man-in-the-middle attacks replacing legitimate responses with malicious large payloads, or simple backend misconfigurations that accidentally generate oversized responses.

The issue affects all SDK operations that communicate with backend services, including attestation requests to notarization backends and verification requests to validation services. Since these operations are central to the oracle's functionality, memory exhaustion attacks could effectively disable oracle services by crashing gateway infrastructure responsible for maintaining continuous data feeds.

Recommendation.

Implement multi-layered protection against oversized responses by adding both pre-flight and streaming protections. For pre-flight protection, examine the `Content-Length` header when present and reject requests that exceed a reasonable threshold before attempting to read the response body. For streaming protection, implement a maximum read limit on the decompressed response body, typically 5-10 MB, and abort the request if this limit is exceeded during the read operation. Consider implementing configurable size limits to allow for different deployment environments while maintaining reasonable defaults that prevent memory exhaustion attacks. These protections should be applied consistently across all HTTP client operations in both SDKs to ensure comprehensive coverage against resource exhaustion vulnerabilities.

Client response. Client has acknowledged the issue and fixed it in PRs https://github.com/venture23-aleo/aleo-oracle-sdk-go/pull/6 and

https://github.com/venture23-aleo/aleo-oracle-sdk-js/pull/4.

#0b - Code Quality and Minor Corrections

Severity: Informational **Location:** *

Description.

During the audit, several minor code issues were identified that do not pose security risks but could lead to confusion or incorrect behavior:

1. In `aleo-oracle-notarization-backend/internal/api/handler/random_attestation.go` at line 60, the comparison should be `if max.Cmp(minMaxValue) < 0`. Currently, if `max = 2`, it triggers an error: `"Max must be greater than 1"`.
2. In `aleo-oracle-encoding/encoding.go` at line 798, the function incorrectly returns `ErrDecodingOptionalsInvalidContentTypeLength` instead of `ErrDecodingOptionalsInvalidBodyLength`.
3. In `aleo-oracle-notarization-backend/internal/services/attestation/encoding.go` at line 98, the variable `attesatationDataPositionInfo` contains a typo and should be renamed to `attestationDataPositionInfo`.

Client Response. The issue was fixed in https://github.com/venture23-aleo/aleo-oracle-encoding/pull/2 and https://github.com/venture23-aleo/aleo-oracle-notarization-backend/pull/20/

#0c - Unattested Response Body

Severity: Informational **Location:** aleo-oracle-notarization-backend

Description.

The notarization backend processes oracle requests and returns an attestation response that includes `responseBody`, representing the original HTTP response from which the `attestationData` is extracted. In the current implementation, the SGX quote's `REPORTDATA` field covers all the attestation response, excluding the `responseBody`.

This means that `responseBody` is not cryptographically bound to the attestation and, could be modified without affecting the attestation integrity. At present, this does not pose a direct security risk since `responseBody` is not used for critical verification, but its presence in the response could lead to confusion or misuse in future code changes.

Recommendation.

If there is a future need to rely on its integrity, consider including `responseBody` in the attested `REPORTDATA`.

Alternatively, make the trust boundary explicit by clearly documenting that `responseBody` is not protected by attestation and should not be used for security-sensitive purposes. If the field is unnecessary, it may be safer to omit it before forwarding to the verifier backend.

Client Response. The client has acknowledged the issue and decided to not include the response body in the quote generation.