Solidity SDK Smart Contract Audit

Wormhole, 8 January 2026

hello@iosiro.com

Contents

1. Introduction

iosiro was commissioned by Wormhole to perform a smart contract audit of their Solidity SDK library. Two auditors conducted the audit between 09 December 2025 and 06 January 2026, over 11 audit days.

Overview

The assessment focused on several core library and utility components intended to be consumed by other projects. Integrators should exercise caution when consuming these components and ensure that the security and functional considerations documented in the comments (particularly around unchecked parsing and hashing helpers) are strictly adhered to.

The audit identified one critical issue and seven informational findings. All issues were either resolved through code changes or closed as acceptable design trade-offs, and there were no open findings at the conclusion of the engagement.

Most findings were related to design and robustness concerns rather than exploitable vulnerabilities that would directly put funds at risk. With these considerations addressed, the reviewed library components are well-positioned for safe integration, provided that the documented usage constraints are respected by downstream consumers.

	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Resolved	1	0	0	0	4
Closed	0	0	0	0	3

Scope

The assessment focused on the source files listed below, with all other files considered out of scope. Any out-of-scope code interacting with the assessed code was presumed to operate correctly without introducing functional or security vulnerabilities.

- **Project name:** wormhole-solidity-sdk
- **Initial audit commit:** 97ca219
- **Final review commit:** b4a904e
- **Files:**
  - src/libraries/BytesParsing.sol
  - src/libraries/VaaLib.sol
  - src/libraries/ReplayProtection.sol
  - src/proxy

- src/utils/Keccak.sol
- src/libraries/PermitParsing.sol
- src/libraries/Percentage.sol

A specification is available in the Specification section of this report.

2. Disclaimer

This report aims to provide an overview of the assessed smart contracts' risk exposure and a guide to improving their security posture by addressing identified issues. The audit, limited to specific source code at the time of review, sought to:

- Identify potential security flaws.
- Verify that the smart contracts' functionality aligns with their documentation.

Off-chain components, such as backend web application code, keeper functionality, and deployment scripts, were explicitly not in scope of this audit.

Given the unregulated nature and ease of cryptocurrency transfers, operations involving these assets face a high risk from cyber attacks. Maintaining the highest security level is crucial, necessitating a proactive and adaptive approach that accounts for the experimental and rapidly evolving nature of blockchain technology. To encourage secure code development, developers should:

- Integrate security throughout the development lifecycle.
- Employ defensive programming to mitigate the risks posed by unexpected events.
- Adhere to current best practices wherever possible.

3. Methodology

The audit was conducted using the techniques described below.

Code review	The source code was manually inspected to identify potential security flaws. Code review is a useful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high-risk areas of the system.
Dynamic analysis	The contracts were compiled, deployed, and tested in a test environment, both manually and through the test suite provided. Dynamic analysis was used to identify additional edge cases, confirm that the code was functional, and validate the reported issues.
Automated analysis	Automated tooling was used to detect the presence of various types of security vulnerabilities. Static analysis results were reviewed manually, any false positives were removed. Any true positive results are included in this report.

4. Audit findings

The table below provides an overview of the audit's findings. Detailed write-ups are provided below.

ID	Issue	Risk	Status
IO-WRM-SDK-001	Incorrect offset used	Critical	Resolved
IO-WRM-SDK-002	Rename return variable	Informational	Resolved
IO-WRM-SDK-003	Unnecessary cast	Informational	Resolved
IO-WRM-SDK-004	Missing warnings for Keccak functions	Informational	Resolved
IO-WRM-SDK-005	Unused Constants	Informational	Resolved
IO-WRM-SDK-006	Sequence-based replay protection griefing	Informational	Closed
IO-WRM-SDK-007	Recalculate envelope size	Informational	Closed
IO-WRM-SDK-008	Free memory space access assumptions	Informational	Closed

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

Critical risk The issue could result in the theft of funds from the contract or its users.

High risk The issue could result in the loss of funds for the contract owner or its users.

Medium risk The issue resulted in the code being dysfunctional or the specification being implemented incorrectly.

Low risk A design or best practice issue that could affect the ordinary functioning of the contract.

Informational An improvement related to best practice or a suboptimal design pattern.

In addition to a risk rating, each issue is assigned a status:

Open The issue remained present in the code as of the final commit reviewed and may still pose a risk.

Resolved The issue was identified during the audit and has since been satisfactorily addressed, removing the risk it posed.

Closed The issue was identified during the audit and acknowledged by the developers as an acceptable risk without actioning any change.

IO-WRM-SDK-001

Incorrect offset used

Critical | Resolved | VaaLib.sol#L430

In `VaaLib::decodeVaaBodyStructMem()`, an offset of 0 was passed to `VaaLib::decodeVaaBodyMemUnchecked()`. This was incorrect because the decode function expects an offset at the end of the header and beginning of the envelope. Given an offset of 0, the function attempts to decode the envelope from within the header, producing incorrect data.

Recommendation

The correct approach, as used in other functions, is to calculate the offset that skips the header by calling `VaaLib::skipVaaHeaderMemUnchecked(encodedVaa, 0)` and pass that offset to `VaaLib::decodeVaaBodyMemUnchecked()`.

Client response

This issue was resolved in 109135d, where `VaaLib::decodeVaaBodyStructMem()` was updated to call `VaaLib::decodeVaaBodyMem()`, which handles the offset calculation correctly.

IO-WRM-SDK-002

Rename return variable

Informational | Resolved | VaaLib.sol#L1089

The returned offset from `VaaLib::decodeGuardianSignatureMemUnchecked()` should be renamed from `envelopeOffset` to `newOffset`, similar to `VaaLib::decodeGuardianSignatureCdUnchecked()`. This is because this function will likely be called in a loop, meaning only the last call would actually be the `envelopeOffset`.

Client response

The return variable was renamed in 109135d.

IO-WRM-SDK-003

Unnecessary cast

Informational | Resolved | BytesParsing.sol#L381-L384

A cast from `uint8` to `uint256` was used in `BytesParsing::asBoolCdUnchecked()`. This may be unnecessary as booleans are equivalent to `uint8`. As such, the final `val` value could be assigned to `ret` directly.

Client response

The cast was used to clear the upper 31 bytes. However, an improvement was implemented in abf9706 to reorder the cleaning step to avoid two extra 0xff masking operations.

IO-WRM-SDK-004

Missing warnings for Keccak functions

Informational | Resolved | Keccak.sol#L14

A warning explaining the risks of unchecked functions, similar to the format at the top of the `BytesParsing.sol` library, was not present for the Keccak free functions. This was especially true for the `Keccak::keccak256SliceUnchecked()` function since an invalid `offset` or `length` value would hash invalid data, creating an invalid hash.

Client response

A comment with the necessary warnings was added above the function in 57ad49b.

IO-WRM-SDK-005

Unused Constants

Informational	Resolved	VaaLib.sol#L249, VaaLib.sol#L263, VaaLib.sol#L286, Percentage.sol#L26, Percentage.sol#L29

The following constants were unused in `VaaLib.sol`. These could be removed unless the intention is to allow contracts that derive from this library access to these constants:

- `MULTISIG_SIGNATURE_ARRAY_OFFSET`
- `MULTISIG_SIGNATURE_V_OFFSET`
- `SCHNORR_ENVELOPE_OFFSET`

Similarly, the following constants were unused in `Percentage.sol`.

- `BYTE_SIZE`
- `EXPONENT_BASE`

Client response

`EXPONENT_BASE`, which had no utility, was removed in 0fd4257. The rest of the constants were kept for reference for consuming contracts.

IO-WRM-SDK-006

Sequence-based replay protection griefing

Informational | Closed | ReplayProtection.sol#L65

When determining the base slot for a given emitter address and chain, the values are XORed instead of encoded and packed. Due to XOR involution, an attacker could attempt to derive an emitter address that satisfies `attacker_emitter == target_emitter XOR target_chain_id XOR attacker_chain_id`. This would allow them to access the same base slot and set the replay protection flags for sequence numbers not yet reached by the valid emitter.

While theoretically possible, it would require the attacker to be able to deploy an emitter at a specific address on their attack chain. Currently, it is infeasible to do this on existing chains.

A more robust approach would be to derive the base slot using `abi.encodePacked` instead of using XOR.

Client response

The team acknowledged this as a theoretical design consideration and opted to leave the implementation unchanged.

IO-WRM-SDK-007

Recalculate envelope size

Informational | Closed | VaaLib.sol#L312-L313

`ENVELOPE_SIZE` could be calculated based on the sum of its individual element sizes. While the current calculation is correct, it is based on an assumption that the `ENVELOPE_TIMESTAMP_OFFSET` will always be zero. Should this change in a future refactor, the `ENVELOPE_SIZE` may become incorrect, and this could be missed in that refactor. Furthermore, calculating it based on the sum of element sizes aligns with the convention used in calculating other `*_SIZE` constants in the library, such as `MULTISIG_GUARDIAN_SIGNATURE_SIZE`.

Client response

The team acknowledged this as a maintainability consideration and opted to leave the implementation unchanged.

IO-WRM-SDK-008

Free memory space access assumptions

Informational	Closed	BytesParsing.sol#L82, Keccak.sol#L32

Multiple functions in the SDK access free memory space directly without moving the pointer beyond the data or clearing the data. Hence, for any operations that use this memory space, the data cannot be assumed to be padded with zeros. The offset and size values used should match the data exactly to avoid padding with junk data.

In a similar vein, any data calculated in the free memory space should be returned to the call stack for use and not accessed through the free memory space to prevent corruption.

Client response

The team acknowledged this as an integration consideration and opted to leave the implementation unchanged.

5. Specification

The following section outlines the SDK's intended functionality at a high level, based on its implementation in the codebase. The assessed components are library and utility building blocks intended to be consumed by other projects. They prioritize gas efficiency (including the use of inline assembly where appropriate), which places additional responsibility on downstream integrators to satisfy the documented preconditions, such as providing correct offsets and lengths to unchecked parsing and hashing helpers.

BytesParsing

`BytesParsing.sol` provides the SDK's low-level byte decoding primitives and serves as the foundation for the higher-level parsing libraries. The implementation prioritizes gas efficiency, using inline assembly where appropriate, and exposes distinct variants for both calldata and memory inputs.

In general, functions are offered in four forms: calldata (`Cd`) and memory (`Mem`) variants, each with a bounds-checked version and an `Unchecked` version. The checked variants protect only against out-of-bounds reads of the provided buffer; they do not protect against arithmetic overflow when computing offsets and lengths.

The intended high-efficiency integration pattern is to parse from calldata where possible and, when using unchecked helpers, to validate correctness at the end by performing a single full-consumption check (e.g., via `checkLength`) so the decoder consumes the payload exactly. Downstream consumers remain responsible for ensuring offsets and lengths are sensibly bounded, including validating user-controlled inputs, and for preferring encoding formats that constrain attacker-controlled length fields.

VaaLib

`VaaLib.sol` provides gas-efficient VAA parsing utilities built on top of `BytesParsing`. The library is designed to decode VAAs with minimal overhead and, where possible, to return decoded components directly on the stack rather than requiring memory allocation for a full struct.

`VaaLib` exposes whole-VAA helpers for use cases such as skipping headers and decoding envelopes/bodies, as well as functions that parse individual VAA components and advance offsets to support iterative decoding, for use cases such as decoding guardian signatures in a loop.

Function variants and naming conventions

Data location variants	functions are generally provided in calldata (`Cd`) and memory (`Mem`) forms.
Struct-return variants	many decoders have a "struct flavor" that returns the decoded values in an associated in-memory struct, in addition to the stack-returning variants.
Parameter naming	`encodedVaa` and `encodedAttestation` are used when the input is expected to contain a single full VAA/attestation; `encoded` is used for partial inputs or buffers containing multiple items.
Unchecked suffix	`Unchecked` denotes the absence of bounds checks - it is not Solidity's `unchecked` keyword. Arithmetic is performed using unchecked operations as overflows are not expected under the VAA format's constraints, while underflow conditions are explicitly checked where relevant.
`Vaa` tag in function names	function names include `Vaa` to improve clarity and reduce the risk of name collisions when used with `using ... for bytes`.

ReplayProtection

`ReplayProtection.sol` provides reusable replay protection primitives intended to prevent a VAA (or message) from being processed more than once. Two replay protection approaches are supported:

- Sequence-based replay protection (per chain + emitter)
    - Tracks replay status per `(emitterChainId, emitterAddress)` using the VAA sequence number.
    - Must only be used for VAAs published with the finalized consistency level (the default case).
    - Implemented via a bitmap to reduce gas cost by avoiding writes to clean storage slots wherever possible.
- Hash-based replay protection
    - Can be used for all consistency levels.
    - Less efficient than sequence-based replay protection because it always writes to a clean storage slot.

- Uses the canonical VAA hash. Note that this is not what CoreBridge returns in `VM.hash` – see the warning box at the top of `VaaLib.sol` for details.

Proxy contracts

The `src/proxy` components provide a slimmed-down, opinionated implementation of the EIP-1967 proxy pattern (based on the reference implementation) intended for downstream consumers that require upgradeability or call forwarding.

Provision is included for consumers to upgrade from an OpenZeppelin proxy implementation. However, care must be taken during such a migration to ensure that the EIP-1967 proxy admin and implementation storage locations are migrated correctly, as incorrect slot handling can lead to loss of upgrade control or unexpected behavior.

Keccak utilities

`Keccak.sol` provides keccak256 helper functions, including slice-based hashing utilities intended to reduce overhead when hashing sub-regions of an encoded payload. These helpers are designed for performance and include unchecked variants; callers must ensure the `offset` and `length` parameters exactly match the intended data region.

PermitParsing

`PermitParsing.sol` standardizes the encoding and decoding of permit payloads so that consuming contracts can acquire tokens via one of the supported permit mechanisms. It supports:

- ERC-2612 permit (`IERC20Permit.permit`)
- Uniswap Permit2 “permit” (approval) (`IAllowanceTransfer.permit`)
- Uniswap Permit2 “transfer” (`ISignatureTransfer.permitTransferFrom`)

The library defines a consistent on-wire format for each mechanism:

- **ERC-2612 Permit**: `value` (uint256), `deadline` (uint256), `r` (bytes32), `s` (bytes32), `v` (uint8)
- **Permit2 Permit**: `amount` (uint160), `expiration` (uint48), `nonce` (uint48), `sigDeadline` (uint256), `signature` (bytes; 65-byte packed r,s,v)
- **Permit2 Transfer**: `amount` (uint256), `nonce` (uint256), `sigDeadline` (uint256), `signature` (bytes; 65-byte packed r,s,v)

Decode functions are provided in 2×2 flavors: calldata (`Cd`) and memory (`Mem`) inputs, each returning either individual stack values or an associated in-memory struct

(`Struct`). As with `BytesParsing`, an `Unchecked` suffix indicates that bounds checks are omitted and unchecked offset arithmetic is used.

The primary decode entry points are `decodePermit`, `decodePermit2Permit`, and `decodePermit2Transfer`. Encoding helpers (`encode*`) exist primarily to support testing.

Percentage

`Percentage.sol` provides a compact representation of percentages using a `uint16` encoding intended to reduce storage and calldata overhead in consuming projects.

Percentages are represented with up to four decimal digits of precision, or up to a maximum of 1000%. The encoding uses a 14-bit mantissa and a 2-bit decimal exponent:

The `uint16` value is split into:

- a 14-bit **mantissa** (m) (the significant digits), and
- a 2-bit **decimal exponent** (e) (a base-10 scale factor).

The represented percentage is:

$$value(\%) = \frac{m}{10^{(1+e)}}$$

The exponent shifts the decimal point downwards, enabling a practical range from sub-0.001% values up to 1000.00%. Due to the decimal exponent scheme, some percentages may admit multiple valid representations; consuming code should treat the value as the decoded percentage rather than relying on a unique canonical encoding.

6. Test coverage report

All tests and fuzz tests executed successfully and were well-written. It was not possible to compile a coverage report because Foundry does not compile fuzz tests successfully when running coverage.