Celestia Arbitrum Nitro

Security Assessment

May 29th, 2024 — Prepared by OtterSec

James Wang james.wang@osec.io

Table of Contents

01 — Executive Summary

Overview

Celestia Organization engaged OtterSec to assess the `celestia-arbitrum` program. This assessment was conducted between March 25th and April 10th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 14 findings throughout this audit engagement.

We identified several critical vulnerabilities. One involves the incorrect validation of the data span for Nitro batches, where the code compares the span with the edge length of the data square instead of its total area (OS-CAN-ADV-00). Another issue is a panic in the functionality for fetching the next operation code when the Nitro machine is in a halted state (OS-CAN-ADV-01). Additionally, the current implementation misinterprets proofs for delayed messages, either failing to create proofs for delayed messages or rejecting valid proofs altogether (OS-CAN-ADV-02).

Furthermore, we highlighted an unhandled deserialization error within the payload recovery functionality, which ignores an error when unmarshalling the Celestia blob pointer (OS-CAN-ADV-07) and the lack of a time-out mechanism which may result in the code hanging indefinitely, waiting for Celestia to respond during data storage (OS-CAN-ADV-05).

We also made recommendations for the removal of several redundant or unused code instances (OS-CAN-SUG-01) and emphasized the need to adhere to coding best practices (OS-CAN-SUG-02). Moreover, we advised against utilizing similar flag values to avoid potential conflicts with Celestia message flags due to overlapping bit fields (OS-CAN-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/celestiaorg. This audit was performed on the `nitro` and `nitro-contracts`, against commits 211cc39 and 9362c3a respectively, specifically focusing on code relevant to supporting Celestia as an data availability layer for Arbitrum Nitro.

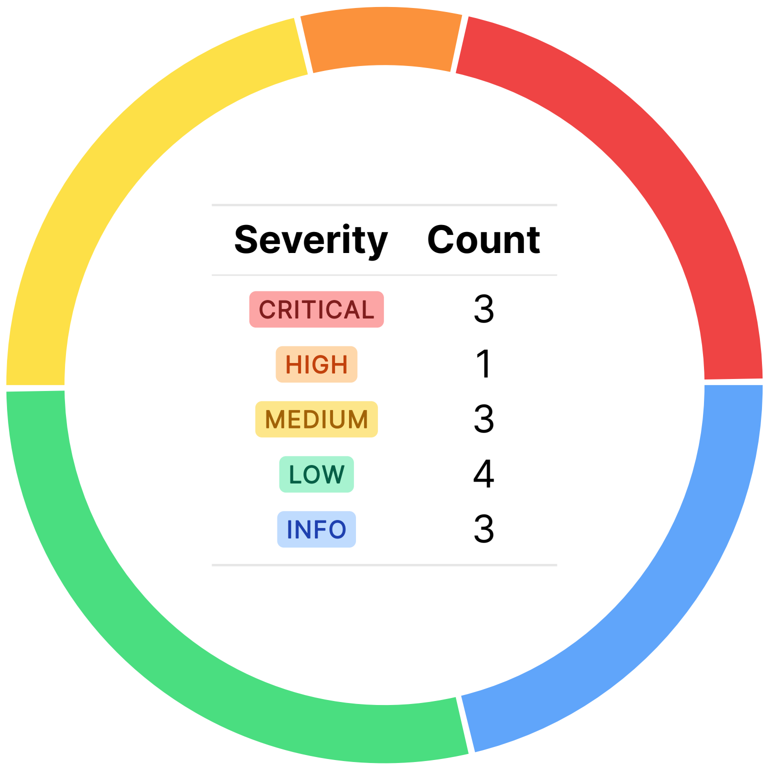
A brief description of the programs is as follows:

Name	Description
celestia-arbitrum	Celestia's fork of Nitro supports using Celestia as the data availability solution. The code also includes modifications to nitro-contracts to support proving data settled on Celestia during interactive challenges.

03 — Findings

Overall, we reported 14 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	3
HIGH	1
MEDIUM	3
LOW	4
INFO	3

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-CAN-ADV-00	CRITICAL	RESOLVED ✓	The current code incorrectly validates the data span for Nitro batches. It compares the span with the edge length of the data square instead of its total area.
OS-CAN-ADV-01	CRITICAL	RESOLVED ✓	If the Nitro machine is in a halted state (no more instructions to execute), the function that fetches the next instruction will return `None` resulting in a `panic` in `GetNextOpcode`.
OS-CAN-ADV-02	CRITICAL	RESOLVED ✓	The existing implementation misinterprets proofs when delayed messages are involved, failing to create proofs for delayed messages or rejecting valid proofs altogether.
OS-CAN-ADV-03	HIGH	RESOLVED ✓	`validateDaProof` may mistake regular proofs for ones with Celestia's data validity checks due to misinterpreting trailing data.
OS-CAN-ADV-04	MEDIUM	RESOLVED ✓	`validateDaProof` treats proofs with potentially unavailable data as empty batches, but it only removes the size information of the attached proof, rendering the untrusted data intact.
OS-CAN-ADV-05	MEDIUM	RESOLVED ✓	The current code may hang indefinitely waiting for Celestia to respond during data storage.
OS-CAN-ADV-06	MEDIUM	RESOLVED ✓	`GetProof` subtracts one from the number of shares when calculating the end block, which is exclusive in Celestia.
OS-CAN-ADV-07	LOW	RESOLVED ✓	`RecoverPayloadFromCelestiaBatch` ignores an error when unmarshalling the Celestia blob pointer.
OS-CAN-ADV-08	LOW	RESOLVED ✓	`filter` in `celestia` may miss relevant events due to Geth's filter limit.
OS-CAN-ADV-09	LOW	RESOLVED ✓	`filter` may miss events during dispute resolution due to a limited search window.
OS-CAN-ADV-10	LOW	RESOLVED ✓	`Write` in `NmtPreimageHasher` replaces data instead of appending it.

Incorrect Square Size Check CRITICAL OS-CAN-ADV-00

Description

`CelestiaBatchVerifier::verifyBatch` calls `DAVerifier.computeSquareSizeFromRowProof` to retrieve the size of the Original Data Square (ODS) based on the row proof. This `squareSize` represents the length of a single edge of the square. In the subsequent check, the contract extracts the `start` and `length` values from the batch data and calculates the end index. However, it compares this end index only with the `squareSize` (edge length). Thus, this check mistakenly assumes `squareSize` is the square's surface area when its the square's edge length.

>_ *BlobstreamVerifier.sol* solidity

```solidity
function verifyBatch(address _blobstream, bytes calldata _data) internal view returns (Result) {
    [...]

    // Calculate size of the Original Data Square (ODS)
    (uint256 squareSize, ) = DAVerifier.computeSquareSizeFromRowProof(rowProof);

    if (squareSize == 0) return Result.COUNTERFACTUAL_COMMITMENT;
    // Check that the start + length posted by the batch poster is not out of bounds
    // otherwise return counterfactual commitment
    // NOTE: a celestia batch has the start (8 bytes) and length (8 bytes) at index 8 - 24
    // we also substract 1 to account for the shares length including the start share
    // thus letting us correctly calculate the end index
    if ((uint64(bytes8(_data[8:16])) + uint64(bytes8(_data[16:24])) - 1) > squareSize)
            return Result.COUNTERFACTUAL_COMMITMENT;

    return Result.IN_BLOBSTREAM;
}
```

Thus, this check is flawed because the square's total area should limit a valid batch's data span, not just its edge length. This incorrect validation may result in the rejection of legitimate data batches. The current check would erroneously flag it as a counterfactual commitment if a batch's data span falls within the square's boundaries but exceeds the edge length.

Remediation

Modify the check such that it compares the calculated end index with the square's total area (`squareSize * squareSize`).

Patch

Fixed in b9b426d and 1b07741.

Panic On Opcode Retrieval CRITICAL OS-CAN-ADV-01

Description

The issue lies in the error handling behavior of `machine::GetNextOpcode`. `GetNextOpcode` retrieves the `opcode` of the next instruction that the Nitro machine is prepared to execute. It calls `arbitrator_get_opcode` internally, which in turn calls `get_next_instruction`. `get_next_instruction` checks if the machine is halted (`self.is_halted()`).

>_ *validator/server_arb/machine.go* rust

```
func (m *ArbitratorMachine) GetNextOpcode() uint16 {
    defer runtime.KeepAlive(m)
    m.mutex.Lock()
    defer m.mutex.Unlock()

    return uint16(C.arbitrator_get_opcode(m.ptr))
}
```

Hence, if the Nitro machine is already in a halted state (no more instructions to execute), `get_next_instruction` will return `None`. As a result, the return value in `arbitrator_get_opcode` will be `None`. Thus, the function panics with the message `Failed to get next opcode for Machine`, disrupting the proof construction process for valid Nitro batches. If proof construction relies on retrieving the next `opcode`, a `panic` at this stage would prevent successful proof generation.

>_ *arbitrator/prover/src/lib.rs* rust

```
#[no_mangle]
pub unsafe extern "C" fn arbitrator_get_opcode(mach: *mut Machine) -> u16 {
    match (*mach).get_next_instruction() {
        Some(instruction) => return instruction.opcode.repr(),
        None => panic!("Failed to get next opcode for Machine"),
    }
}
```

Remediation

Skip instruction fetching when the `machine` is in the halted stated, and append a placeholder instruction instead.

Patch

Fixed in f6180a6 and 1c29c3a.

Proof Rejection On Delayed Messages CRITICAL OS-CAN-ADV-02

Description

The vulnerability lies in how `getDAProof` within `challenge_manager` handles proofs related to messages on the Arbitrum Nitro network. Specifically, the issue arises due to the introduction of delayed messages. Delayed messages are a feature in Arbitrum that allows sending messages directly to layer 1 (`L1`) after a certain delay. This functionality was not considered in the original implementation of `getDAProof`. The vulnerability stems from how the function extracts the batch number from the provided proof. `getDAProof` assumes the last eight bytes of the proof directly represent the batch number.

>_ *staker/challenge_manager.go* go

```go
func (m *ChallengeManager) getDAProof(ctx context.Context, proof []byte) ([]byte, error) {
    [...]
    if opCode == ReadInboxMessage {
        // remove opcode bytes
        proof = proof[:len(proof)-2]
        // Read the last 8 bytes as a uint64 to get our batch number
        batchNumBytes := proof[len(proof)-8:]
        [...]
    }
    [...]
}
```

However, with delayed messages, this assumption breaks down. The last eight bytes of delayed messages are delayed message numbers, which could not be interpreted as batch numbers, and subsequent usage of it to fetch batches would fail. Similarly, contracts relying on the layout of data in the proof may misinterpret the information (such as in `validateDaProof` in `OneStepProverHostIo`), resulting in the inability to construct proofs for delayed messages or rejection of benign proofs.

>_ *src/osp/OneStepProverHostIo.sol* solidity

```solidity
function validateDaProof(bytes calldata proof) internal view returns (uint256) {
    [...]
    if ([...]) {
        CelestiaBatchVerifier.Result result = CelestiaBatchVerifier.verifyBatch(
            BLOBSTREAM,
            proof[proof.length - daProofSize - 3:proof.length - 4]
        );
        [...]
    }
    return proofEnd;
}
```

Remediation

Include the message type within the proof and use it to skip fetching `DaProof` for delayed messages. The same flag can also be used within the OneStepProverHostIo contract to avoid incorrect attempts of `DaProof` verification.

Patch

Fixed in f6180a6.

Inconsistency In DA Proof Verification HIGH OS-CAN-ADV-03

Description

Celestia appends a `DA` proof to the end of a normal batch proof. The last four bytes of this appended `DA` proof represent the size of the `DA` proof itself. The contract utilizes this size to determine the starting position of the actual batch data within the combined proof. The vulnerability arises because the contract relies on the presence of the size field to identify a `DA` proof.

>_ *src/osp/OneStepProverHostIo.sol* solidity

```solidity
function validateDaProof(bytes calldata proof) internal view returns (uint256) {
    [...]
    if (
        daProofSize < proof.length - 4 &&
        proof[proof.length - daProofSize - 4] & CELESTIA_MESSAGE_HEADER_FLAG != 0
    ) {
        CelestiaBatchVerifier.Result result = CelestiaBatchVerifier.verifyBatch(
            BLOBSTREAM,
            proof[proof.length - daProofSize - 3:proof.length - 4]
        );
        [...]
    }
    return proofEnd;
}
```

However, non-Celestia batches will not have a `DA` proof attached and the last four bytes of such proofs will be arbitrary data. Thus, in the worst case scenario, this may result in benign non-Celestia batches falling into the condition that determines if a `DA` proof is attached in `OneStepProverHostIoCelestiaMock::validateDaProof` and getting rejected.

Remediation

Since the format of a regular batch proof is fixed, the contract should directly calculate the offset of the batch data within the combined proof without needing the size information.

Patch

Fixed in f6180a6 and b9b426d.

Incorrect Proof Augmentation MEDIUM OS-CAN-ADV-04

Description

`validateDaProof` aims to treat proofs with counterfactual commitment verification results (indicating potentially unavailable data) as empty batches. This ensures the contract does not process proofs with potentially unreliable data. The function subtracts `daProofSize - 4` from `proofEnd` if the verification result is `COUNTERFACTUAL_COMMITMENT`. This aims to remove the `DA` proof itself (`size + data`) from the proof.

>_ src/osp/OneStepProverHostIo.sol solidity

```solidity
function validateDaProof(bytes calldata proof) internal view returns (uint256) {
    [...]
    if (
        daProofSize < proof.length - 4 &&
        proof[proof.length - daProofSize - 4] & CELESTIA_MESSAGE_HEADER_FLAG != 0
    ) {
        [...]
        // if its a counterfactual commitment, we replace the batch data with an empty batch
        if (result == CelestiaBatchVerifier.Result.COUNTERFACTUAL_COMMITMENT) {
            // this would slice the array into an empty batch
            proofEnd -= daProofSize - 4;
        }
        [...]
    }
    return proofEnd;
}
```

However, the subtraction only removes `daProofSize - 4` bytes instead of the correct `daProofSize + 4` bytes for both the 4-byte size field but and the actual `DA` proof data itself. This leaves the data portion of the counterfactual `DA` proof still attached to the main proof. Thus, even though the verification failed, the contract continues to process the main proof along with the untrusted data from the counterfactual `DA` proof.

Remediation

Ensure the subtraction accounts for the entire `DA` proof size, including both the size field and the data.

Patch

Fixed in b9b426d.

Missing RPC Timeout MEDIUM OS-CAN-ADV-05

Description

Currently `maybePostSequencerBatch` calls `b.celestiaWriter.Store` without a timeout for the `RPC` request. This means the function may hang indefinitely if the Celestia `RPC` server is unresponsive or extremely slow. There is a fallback mechanism to on-chain storage if Celestia storage fails. However, without a timeout, the code will never reach the fallback logic if Celestia `RPC` is stuck.

Remediation

Add a timeout, to ensure `maybePostSequencerBatch` does not block indefinitely waiting for Celestia.

Patch

Fixed in f6180a6.

Incorrect Celestia ProofShare Query Range MEDIUM OS-CAN-ADV-06

Description

There is an issue within `celestia::GetProof`, where it subtracts one from `blobPointer.SharesLength` before calling `c.Prover.Trpc.ProveShares`. This needs to be removed as `EndBlock` is exclusive in Celestia as stated in the documentation.

>_ *das/celestia/celestia.go* go

```go
func (c *CelestiaDA) GetProof(ctx context.Context, msg []byte) ([]byte, error) {
    if valid {
        sharesProof, err := c.Prover.Trpc.ProveShares(ctx, blobPointer.BlockHeight,
            ↪  blobPointer.Start, blobPointer.Start+blobPointer.SharesLength-1)
        [...]
    }
    [...]
}
```

Remediation

Remove the subtraction from one.

Patch

Fixed in f6180a6.

Unhandled Deserialization Error LOW OS-CAN-ADV-07

Description

`RecoverPayloadFromCelestiaBatch` in `inbox` ignores the error returned by `blobPointer.UnmarshalBinary(blobBytes)`. By ignoring the error, the function continues execution as if the unmarshalling was successful. If the unmarshalling fails due to corrupt data or an unexpected format, the `blobPointer` may be filled with garbage values. These garbage values may then be utilized in subsequent function calls, potentially resulting in unintended data retrieval or errors during the data validation steps.

>_ *arbstate/inbox.go* go

```go
func RecoverPayloadFromCelestiaBatch(
        ctx context.Context,
        batchNum uint64,
        sequencerMsg []byte,
        celestiaReader celestia.DataAvailabilityReader,
        preimages map[arbutil.PreimageType]map[common.Hash][]byte,
) ([]byte, error) {
        [...]
        blobPointer := celestia.BlobPointer{}
        blobBytes := buf.Bytes()
        blobPointer.UnmarshalBinary(blobBytes)
        [...]
}
```

Remediation

Ensure to check the error returned by `blobPointer.UnmarshalBinary`.

Patch

Fixed in 87a7d33.

Improper Ethereum Query Start Block Calculation LOW OS-CAN-ADV-08

Description

There is a potential issue with the way `celestia::filter` calculates the starting block for searching Ethereum events. `start` is initialized to zero. If `latestBlock` is less than 5000 (limit imposed by Geth), `start` remains zero. If `latestBlock` is greater than 5000, there is no initial adjustment to start. Geth limits filters to a maximum of 5000 blocks. Thus, if `latestBlock` is greater than 5000 and `start` remains zero, the filter will only search the most recent 5000 blocks. This may miss the relevant `DataCommitmentStored` event for the target `celestiaHeight` if it was submitted before the current `latestBlock - 5000`.

>_ das/celestia/celestia.go go

```go
func (c *CelestiaDA) filter(ctx context.Context, latestBlock uint64, celestiaHeight uint64,
    ↪ backwards bool) (*blobstreamx.BlobstreamXDataCommitmentStored, error) {
    // Geth has a default of 5000 block limit for filters
    start := uint64(0)
    if latestBlock < 5000 {
            start = 0
    }
    end := latestBlock
    for attempt := 0; attempt < 10; attempt++ {
        [...]
        if backwards {
            start -= 5000
            if end < 5000 {
                end = start + 10
            } else {
                end -= 5000
            }
        }
        [...]
    }
    return nil, fmt.Errorf("unable to find Data Commitment Stored event in Blobstream")
}
```

Remediation

Initialize `start` to `latestBlock - 5000` when `latestBlock` is greater than 5000. This ensures the filter always starts searching from a point that captures at least the last 5000 blocks. Additionally, validate that `start` is always greater than or equal to 5000 before subtracting from it in the backward search logic `(start -= 5000)`. This prevents potential negative values for `start`, which may result in invalid filter creation.

Patch

Fixed in f6180a6 and 1c29c3a.

Insufficient Proof Query Window LOW OS-CAN-ADV-09

Description

`filter` currently iterates ten times, with each iteration searching for events within a 5000 block window on the Ethereum blockchain. The Celestia network has a seven-day challenge period for dispute resolution. Assuming an average block time of twelve seconds on the Ethereum blockchain, seven days translate to roughly 43,200 blocks. With a 5000 block search window per iteration and ten iterations, the total scanned block range is 50,000 blocks (`10 iterations * 5000 blocks/iteration`).

>_ *das/celestia/celestia.go* go

```go
func (c *CelestiaDA) filter(ctx context.Context, latestBlock uint64, celestiaHeight uint64,
    ↪  backwards bool) (*blobstreamx.BlobstreamXDataCommitmentStored, error) {
    // Geth has a default of 5000 block limit for filters
    start := uint64(0)
    if latestBlock < 5000 {
            start = 0
    }
    end := latestBlock
    for attempt := 0; attempt < 10; attempt++ {
        [...]
    }
    return nil, fmt.Errorf("unable to find Data Commitment Stored event in Blobstream")
}
```

Thus, there is a possibility that the `DataCommitmentStored` event may fall outside the scanned block range, especially if Ethereum block production is slower than the assumed twelve seconds per block. This may result in the function failing to find the event and potentially delaying or throwing an error in the data verification process.

Remediation

Add one more iteration to ensure coverage of the seven day challenge phase under assumption where `L1` block production works properly.

Patch

Fixed in f6180a6.

Incorrect Preimage Handling In Hasher LOW

OS-CAN-ADV-10

Description

`nmt_hasher::Write` replaces the existing content of the `h.data` slice with the new data (`p`). Replacing the `h.data` slice with only the new data in each `Write` call essentially restarts the hashing process from scratch for every new data chunk. This means the final hash value would not reflect the complete set of data that was intended to be hashed.

>_ *das/celestia/tree/nmt_hasher.go* go

```go
func (h *NmtPreimageHasher) Write(p []byte) (n int, err error) {
    h.data = append(h.data[:0], p...) // Update the data slice with the new data
    return h.Hash.Write(p)
}
```

Currently `Write` is always preceded by a call to `reset`. `reset` clears the internal state of the underlying `hasher` and the data slice within `NmtPreimageHasher`. Therefore, in this usage pattern, replacing the data in `write` will not have a practical impact because the `hasher` starts fresh every time.

Remediation

Append the new data (`p`) to the existing data in `h.data` instead of replacing it.

Patch

Fixed in f6180a6.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-CAN-SUG-00	Both `SequencerInbox` and `das_reader` utilize the same flag (`0xc`) for Celestia messages, which conflicts with a Tree `DAS` message flag.
OS-CAN-SUG-01	Several redundant or un-utilized code instances may be removed.
OS-CAN-SUG-02	Suggestions to ensure adherence to coding best practices.

Overlapping Bit Fields OS-CAN-SUG-00

Description

Both `SequencerInbox` and `das_reader` utilize the same byte value (`0x0c`) for `CELESTIA_MESSAGE_HEADER_FLAG`. However, this flag is also utilized in `TREE_DAS_MESSAGE_HEADER_FLAG` in `SequencerInbox`. Thus, there is a collision between the flags of a Celestia message and a Tree `DAS` message.

Remediation

Avoid using `0xc` for `CELESTIA_MESSAGE_HEADER_FLAG`.

Patch

Fixed in a84e4fa.

Code Redundancy OS-CAN-SUG-01

Description

1. The proof for `ReadInboxMessage` contains various data elements to verify specific information. For non- `ReadInboxMessage` `opcodes` (`opcodes` that don't interact with the inbox), certain proof elements may be irrelevant and should be removed.
2. Within `proof`, remove the `CelestiaProof` structure as it is not utilized anywhere.

>_ *das/celestia/proof.go* go

```go
type CelestiaProof struct {
        namespaceNode      NamespaceNode
        binaryMerkleProof BinaryMerkleProof
        attestationProof  AttestationProof
}
```

Remediation

Remove the above-mentioned code.

Patch

Both issues fixed in f6180a6.

Code Maturity OS-CAN-SUG-02

Description

1. Within `Read` in `celestia` and `main` replace the current check (`if endRow > odsSize || startRow > odsSize`) with this: `endRow >= odsSize || startRow >= odsSize` as trying to access data at index `odsSize` would be out of bounds. Additionally the following check: `if startRow == endRow && startColumn > endColumn+1` may be replaced with this: `if startRow == endRow && startColumn > endColumn` in `Read` in `celestia`, as it makes more sense semantically.
2. Utilize `NamespaceSize` in `main::Read` instead of directly writing `29` to help improve maintainability and readability.

>_ *cmd/replay/main.go* go

```go
func (dasReader *PreimageCelestiaReader) Read(ctx context.Context, blobPointer
    ↪  *celestia.BlobPointer) ([]byte, *celestia.SquareData, error) {
    [...]
    for i, share := range shares {
        // trim extra namespace
        share := share[29:]
            if i == 0 {
                data = append(data, share[tree.NamespaceSize+5:]...)
                    continue
            }
            data = append(data, share[tree.NamespaceSize+1:]...)
    }
    [...]
}
```

Remediation

Implement the above-mentioned suggestions.

Patch

1. Fixed in f63c53e.
2. Fixed in f63c53e.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.