verichains

SECURITY AUDIT OF

AVAIL BRIDGE

Public Report

Jul 26, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.

EXECUTIVE SUMMARY

This Security Audit Report prepared by Verichains Lab on Jul 26, 2024. We would like to thank Avail for trusting Verichains Lab, delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Avail Bridge. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the source code, along with some recommendations.

TABLE OF CONTENTS


verichains

1. MANAGEMENT SUMMARY

1.1. About Avail Bridge

Avail is the unification layer for Web3. Avail aims to accelerate the unification of Web3 using its cutting-edge modular technology stack that combines data availability, aggregation, and shared security.

To achieve this, Avail has been building out its Unification Layer, a unifying tech stack starting with the foundational Data Availability (DA) layer, the Nexus unification layer, and Fusion, an additive security layer. Avail stands to anchor the whole Web3 ecosystem with a scalable, cutting-edge data availability layer, harnessing validity proofs that leverage KZG polynomial commitments to ensure immediate and reliable data availability, enabling rollups to grow, connect, stay secure, and adapt.

With the Avail Bridge, users can bridge AVAIL tokens to and from Ethereum. This can be expanded to include other tokens with further runtime upgrades. However, to allow for a generalized interface that can be expanded into different use cases, an arbitrary message bridge on Ethereum is ideal.

1.2. Audit Scope

In this particular project, a timebox approach was used to define the consulting effort. This means that **Verichains Lab** allotted a prearranged amount of time to identify and document vulnerabilities. Because of this, there is no guarantee that the project has discovered all possible vulnerabilities and risks.

Furthermore, the security check is only an immediate evaluation of the situation at the time the check was performed. An evaluation of future security levels or possible future risks or vulnerabilities may not be derived from it.

This audit focused on identifying security flaws in code and the design of the Avail Bridge. It was conducted on the source code provided by Avail team.

The following repositories were made available in the course of the review:

Repository	Commit
https://github.com/availproject/avail	9d88dbf
https://github.com/availproject/avail-core	993fae8
https://github.com/availproject/contracts	b8a0fef

1.2.1. Detailed scope

The detailed audit scope is LIMITED ONLY to the following files and directories with the corresponding commit hashes:

- *https://github.com/availproject/avail/tree/9d88dbf/pallets/vector*
- *https://github.com/availproject/avail/tree/9d88dbf/patricia-merkle-trie*
- *https://github.com/availproject/avail/tree/9d88dbf/base/src/header_extension*
- *https://github.com/availproject/avail-core/blob/993fae8/core/src/data_proof.rs*
- *https://github.com/availproject/avail-core/blob/993fae8/core/src/data_proof/message.rs*
- *https://github.com/availproject/contracts/tree/b8a0fef/*

Out of scope

The following two repositories are also under review to support the audit. However, they are considered out of scope for this audit:

- *https://github.com/succinctlabs/telepathyx*
- *https://github.com/succinctlabs/vectorx*

1.3. Audit Methodology

Our security audit process includes four steps:

- Mechanism Design is reviewed to look for any potential problems.
- Source codes are scanned/tested for commonly known and more specific vulnerabilities using public and our in-house security analysis tool.
- Manual audit of the codes for security issues. The source code is manually analyzed to look for any potential problems.
- Set up a testing environment to debug/analyze found issues and verify our attack PoCs if necessary.

1.3.1. Audit process

Below are overall processes for the Audit service:

Step	Assignee	Description
Step 1: Handle the resource	Avail	Avail provides the source code and related documents to Verichains for the audit process.
Step 2: Test & Audit	Verichains	Verichains performs the test and review process, handing to Avail the detailed reports about the found bugs, vulnerabilities result come with suggestions how to resolve the problem.

Step	Assignee	Description
Step 3: Bug fixes	Avail	Avail has time to check the report, release updates for all reported issues.
Step 4: Verification	Verichains	Verichains will double-check all the fixes, patches which related to the reported issues are fixed or not.
Step 5: Publish reports	Verichains	Verichains will create final public reports for the project.

Table 1. Audit process

1.3.2. Vulnerability severities

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the functioning; creates a critical risk; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the code with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the code with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 2. Severity levels

1.4. Disclaimer

Avail acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Avail understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Avail agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Avail will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Avail, the final report will be considered fully accepted by the Avail without the signature.

2. AUDIT RESULT

2.1. Overview

From a high-level overview, the audit scope can be divided into three main parts:

- *Vector Pallet*: This component is responsible for syncing the `finalized_header_root`, `execution_state_root`, and `sync_committee_poseidon` from the Ethereum Beacon Chain. With the corresponding block data from Beacon Chain, this pallet can handle AVAIL token transfers and arbitrary messages sent from Ethereum.
- *Data Root Upgrade*: In this upgrade, the `blob_root` and `bridge_root` will be merged to form a single `data_root` for the Avail block. A range of data roots can be used to construct the data root commitment, which will be synced to the VectorX contract. These commitments will subsequently be used to verify cross-chain message proofs submitted to the `AvailBridge` contract on Ethereum.
- *AvailBridge contract*: This is the main contract responsible for handling cross-chain messages and token transfers from Avail to Ethereum. The bridge messages are checked against the bridge Merkle tree and the `dataRootCommitment` from the `VectorX` contract.

2.1.1. Vector pallet

To support cross-chain messaging to and from Ethereum, the Vector pallet is implemented to simulate the LightClient contract from Telepathy. Telepathy, at its core, is an interoperability protocol for Ethereum that is secured by verifying the signatures of Ethereum validators on-chain.

Telepathy implements two main circuits using the Circom language, which uses the Groth16 proving system to generate zk-SNARK proofs:

- *Step*: This circuit verifies that at least two-thirds of the sync committee have provided valid signatures for the block header before adding it to the Light Client.
- *Rotate*: This circuit verifies the cryptographic commitment of the new sync committee when it is changed periodically.

The detailed implementation of these circuits can be found in the Telepathy repository (which is used to verify the Vector pallet - Light Client implementation for Avail Bridge):

- *https://github.com/succinctlabs/telepathyx/blob/2f6822b/circuits/step.circom*
- *https://github.com/succinctlabs/telepathyx/blob/2f6822b/circuits/sync_committee.circom*

Sending messages

The message sent from Avail to the Ethereum is submitted as an extrinsic `send_message` and based on the provided parameters validation is done. This message is filtered with `HeaderExtensionDataFilter` and used as data for a `data_root` construction.

Receiving messages

The message that is emitted from the smart contract (sent via storage) on the Ethereum network is reflected in the Avail bridge pallet in terms of execution state root.

Once the message has been emitted on the Ethereum network, the operator using zero-knowledge proves that the header is correct by submitting extrinsic as a `step` call of the `fulfill_call` and the `execution_state_root` gets updated as well as other important fields for the particular `slot`.

Avail implementation for the cross-chain message handling is done in via the `execute` extrinsic function. This function will check all relevant preconditions and execute a message with a given proof that is validated against the bridge pallet execution state root from the storage. The provided proofs will be checked using `patricia-merkle-trie` which checks and validates account and storage proofs.

2.1.2. Data root upgrade

The data root upgrade is designed to support the cross-chain messaging feature from Avail to Ethereum and can be expanded to any other EVM-compatible chains. In this upgrade, the `dataRoot` for an Avail block is constructed as follows:

$$dataRoot = \texttt{keccak256}(\begin{cases} \texttt{0x0000000000000000000000000000000000} & \text{when } \emptyset \\ blobRoot & \text{otherwise} \end{cases}, \begin{cases} \texttt{0x0000000000000000000000000000000000} & \text{when} \\ bridgeRoot & \text{otherwise} \end{cases})$$

The `blobRoot` is the root of the data submission Merkle tree (or blob tree), where empty nodes are filled with zero-hash values to the next power of two nodes, forming a perfect binary tree. This perfect binary structure is required by the Merkle proof verification algorithm on the Ethereum side, as the algorithm only operates correctly on perfect binary Merkle trees (where the number of leaves must be `2**n`). Each leaf of the blob Merkle tree represents a `keccak256` hash of the data submitted by the `DACall::submit_data` extrinsic from the Avail chain.

Similarly, the `bridgeRoot` is the root of the bridge-message Merkle tree, which is also a perfect binary tree. Each leaf of this tree represents the ABI encoded bridge message submitted by the `VectorCall::send_message` extrinsic from the Avail chain. Depending on the `messageType`, the bridge message can either be a cross-chain arbitrary message or a token transfer. Only `AVAIL` token is supported in the current version.

2.1.3. AvailBridge contract

The `receiveMessage` function of the `AvailBridge` contract is responsible for handling cross-chain messages from the Avail chain. The validity of the message is ensured by the `MerkleProofInput`, with the ABI-encoded message as the `bridgeLeaf`. This `bridgeLeaf` is verified against the `bridgeRoot`.

The `bridgeRoot`, along with the `blobRoot`, combines to form the `dataRoot`, which is finally verified against the `dataRootCommitment` from the VectorX contract.

2.2. Findings

This section contains a detailed analysis of all the vulnerabilities which were discovered by our team during the audit process.

Avail team has updated the code, according to Verichains's draft report.

#	Issue	Severity	Status
1	U256 conversion can cause panics	LOW	Fixed
2	Avoid using default values instead of throwing errors	INFORMATIVE	Fixed
3	Duplicated condition check in `decode_proof` function	INFORMATIVE	Fixed
4	Unused functions	INFORMATIVE	Fixed
5	Hardcoded values in `filter_vector_call` function	INFORMATIVE	Acknowledged

2.2.1. U256 conversion can cause panics LOW

Affected files:

- pallets/vector/src/lib.rs
- pallets/vector/src/state.rs

In the `set_poseidon_hash` function, the `U256::from` function is used to convert a vector of `u8` to `U256`. The function `parse_rotate_output` also uses a similar function `U256::from_big_endian`. These functions can panic if the length of the input vector is greater than 32.

```
// File: pallets/vector/src/lib.rs#608
#[pallet::call_index(4)]
#[pallet::weight(T::WeightInfo::set_poseidon_hash())]
pub fn set_poseidon_hash(
```

```
    origin: OriginFor<T>,
    #[pallet::compact] period: u64,
    poseidon_hash: BoundedVec<u8, ConstU32<200>>,
) -> DispatchResultWithPostInfo {
    ensure_root(origin)?;

    let hash = U256::from(poseidon_hash.to_vec().as_slice()); // AUDIT: CAN CAUSE PANIC

    SyncCommitteePoseidons::<T>::insert(period, hash);
    Self::deposit_event(Event::SyncCommitteeUpdated { period, root: hash });
    Ok(().into())
}

// File: pallets/vector/src/state.rs#143
pub fn parse_rotate_output(output: Vec<u8>) -> U256 {
    U256::from_big_endian(output.as_slice()) // AUDIT: CAN CAUSE PANIC
}
```

RECOMMENDATION

It is recommended to check the length of the input vector before converting it to `U256` to avoid panics. These functions should return errors if the length of the input vector exceeds 32.

UPDATES

- **Jul 26, 2024**: This issue has been acknowledged and fixed by Avail team.

2.2.2. Avoid using default values instead of throwing errors INFORMATIVE

Affected files:

- pallets/vector/src/lib.rs
- pallets/vector/src/verifier.rs

In the `VerifyingKeyJson::to_verifying_key` function, when errors occur while parsing the `gamma_abc_g1` field of the verifying key, the function uses a default value instead of throwing an error. This behavior may result in an incorrect verifying key being returned in some cases.

```
// File: pallets/vector/src/verifier.rs#111
let gamma_abc_g1: Vec<G1Affine> = self
    .ic
    .iter()
    .map(|coords| {
        G1Affine::from(G1Projective::new(
            Fq::from_str(&coords[0]).unwrap_or_default(), // AUDIT: THROW ERROR INSTEAD OF
USING DEFAULT VALUE
            Fq::from_str(&coords[1]).unwrap_or_default(),
            Fq::from_str(&coords[2]).unwrap_or_default(),
        ))
```

```
    })
    .collect();
```

In the `send_message` function, when the `extrinsic_index` function returns `None`, we should throw an error instead of using a default value, which results in an incorrect value for `tx_idx`. The same issue also occurs in the `fetch_curr_message_id` function.

```
// File: pallets/vector/src/lib.rs#581
pub fn send_message(
    origin: OriginFor<T>,
    message: Message,
    to: H256,
    #[pallet::compact] domain: u32,
) -> DispatchResultWithPostInfo {
    let who = ensure_signed(origin)?;

    let dispatch = Self::do_send_message(who, message, to, domain);
    if dispatch.is_err() {
        let _ = MemoryTemporaryStorage::update::<Vec<Compact<u32>>, _>(
            FAILED_SEND_MSG_ID.to_vec(),
            |failed| {
                let tx_idx =
                    <frame_system::Pallet<T>>::extrinsic_index().unwrap_or_default(); //
AUDIT: THROW ERROR INSTEAD OF USING DEFAULT VALUE
                failed.push(tx_idx.into());
                log::trace!(target: LOG_TARGET, "Send Message failed txs: {failed:?}");
            },
        );
    }

    dispatch
}

// File: pallets/vector/src/lib.rs#828
fn fetch_curr_message_id() -> u64 {
    let number = <frame_system::Pallet<T>>::block_number().saturated_into::<u32>();
    let tx_index = <frame_system::Pallet<T>>::extrinsic_index().unwrap_or_default(); //
AUDIT: THROW ERROR INSTEAD OF USING DEFAULT VALUE

    tx_uid(number, tx_index)
}
```

UPDATES

- **Jul 26, 2024**: This issue has been acknowledged and fixed by Avail team.

2.2.3. Duplicated condition check in `decode_proof` function INFORMATIVE

Affected files:

- pallets/vector/src/verifier.rs

In the decode_proof function, the condition `ethabi::Token::FixedArray(ar) = &t[1]` is checked twice. The second check can be removed to improve code readability.

```
// File: pallets/vector/src/verifier.rs#224
#[allow(clippy::type_complexity)]
pub fn decode_proof(
    proof: Vec<u8>,
) -> Result<(Vec<String>, Vec<Vec<String>>, Vec<String>), VerificationError> {
    // ...
        if let ethabi::Token::FixedArray(ar) = &t[1] { // AUDIT: DUPLICATED CODE
            if let ethabi::Token::FixedArray(arr) = &ar[0] {
                if let ethabi::Token::Uint(u) = &arr[0] {
                    b00 = u.to_string();
                }
                if let ethabi::Token::Uint(u) = &arr[1] {
                    b01 = u.to_string();
                }
            }

            if let ethabi::Token::FixedArray(ar) = &t[1] { // AUDIT: DUPLICATED CODE
                if let ethabi::Token::FixedArray(arr) = &ar[1] {
                    if let ethabi::Token::Uint(u) = &arr[0] {
                        b10 = u.to_string();
                    }
                    if let ethabi::Token::Uint(u) = &arr[1] {
                        b11 = u.to_string();
                    }
                }
            }
        }
    // ...
}
```

UPDATES

- **Jul 26, 2024**: This issue has been acknowledged and fixed by Avail team.

2.2.4. Unused functions INFORMATIVE

Affected files:

- pallets/vector/src/lib.rs

The `decode_message_data` function is not used in the codebase. It is recommended to remove the function to avoid confusion.

```
/// # TODO
/// - Remove `dead_code` here.
#[allow(dead_code)]
```

```
fn decode_message_data(data: Vec<u8>) -> Result<(H256, U256), DispatchError> {
    let decoded_data = ethabi::decode(
        &[
            ethabi::ParamType::FixedBytes(32),
            ethabi::ParamType::Uint(256),
        ],
        data.as_slice(),
    )
    .map_err(|_| Error::<T>::CannotDecodeData)?;
    ensure!(decoded_data.len() == 2, Error::<T>::CannotDecodeData);

    let asset_id_token = decoded_data.first().ok_or(Error::<T>::CannotDecodeData)?;
    let asset_id = asset_id_token
        .clone()
        .into_fixed_bytes()
        .ok_or(Error::<T>::CannotDecodeData)?;

    let asset = H256::from_slice(asset_id.as_slice());

    let amount_token = decoded_data.get(1).ok_or(Error::<T>::CannotDecodeData)?;
    let amount = amount_token
        .clone()
        .into_uint()
        .ok_or(Error::<T>::CannotDecodeData)?;

    Ok((asset, amount))
}
```

UPDATES

- **Jul 26, 2024**: This issue has been acknowledged and fixed by Avail team.

2.2.5. Hardcoded values in `filter_vector_call` function INFORMATIVE

Affected files:

- runtime/src/transaction_filter.rs

In the `filter_vector_call` function, the `AddressedMessage` struct is initialized with a hardcoded value of `1` for the `origin_domain` field. Using numeric values as constants can make the code less readable and harder to maintain.

```
fn filter_vector_call(
    failed_transactions: &[u32],
    caller: Option<&AccountId>,
    call: &VectorCall<Runtime>,
    block: u32,
    tx_index: usize,
) -> Option<ExtractedTxData> {
    // ...
```

```
    let from: [u8; 32] = *caller?.as_ref();
    let id = tx_uid(block, tx_index);
    let msg = AddressedMessage::new(message.clone(), H256(from), *to, 1, *domain, id); //
HARDCODE of ORIGIN DOMAIN
    let bridge_data = Some(BridgedData::new(tx_index, msg));
    Some(ExtractedTxData {
        bridge_data,
        ..Default::default()
    })
}
```

RECOMMENDATION

It is recommended to define these values as constants or enums to improve code readability.

UPDATES

- **Jul 26, 2024**: This issue has been acknowledged by Avail team.

2.3. Smart contracts findings

This section contains the findings related to the Avail Bridge smart contracts.

#	Issue	Severity	Status
1	Lack of Message Monitoring for Failed Transactions	LOW	Acknowledged
2	Add `_gap` Variable to Upgradable Contract	LOW	Fixed
3	Lack of Fee Charging	INFORMATIVE	Acknowledged
4	Restrict The Maximum Fee to Prevent User Mistakes	INFORMATIVE	Acknowledged
5	Require The `_bridge` Is Not The Zero Address	INFORMATIVE	Acknowledged
6	Ensure That The function `initialize` Is Called at Deployment	INFORMATIVE	Acknowledged
7	Emitting Events Following State Updates	INFORMATIVE	Acknowledged
8	Different leaf input format for `verifyBlobLeaf` and `verifyBridgeLeaf`	INFORMATIVE	Acknowledged

2.3.1. Lack of Message Monitoring for Failed Transactions LOW

Affected files:

- AvailBridge.sol

Failed transactions, attributed to various factors like insufficient gas or destination address denial, lack proper tracking mechanisms.

```
function receiveMessage(Message calldata message, MerkleProofInput calldata input)
        external
        whenNotPaused
        onlySupportedDomain(message.originDomain, message.destinationDomain)
        nonReentrant
{
    if (message.messageType != MESSAGE_TX_PREFIX) {
        revert InvalidMessage();
    }

    _checkBridgeLeaf(message, input);

    // downcast SCALE-encoded bytes to an Ethereum address
    address dest = address(bytes20(message.to));
    IMessageReceiver(dest).onAvailMessage(message.from, message.data);

    emit MessageReceived(message.from, dest, message.messageId);
}
```

RECOMMENDATION

Implement a mechanism that enables users to track and retry failed transactions, ensuring robustness in message handling and transaction reliability. We can refer to LayerZero as an example:

https://github.com/LayerZero-Labs/solidity-examples/blob/c04e7d2/contracts/lzApp/NonblockingLzApp.sol

UPDATES

- **Jul 22, 2024**: Since all messages are retry-able and can be executed in an async manner, the Avail team has decided not to fix this issue.

2.3.2. Add `_gap` Variable to Upgradable Contract LOW

Affected files:

- AvailAttestation.sol
- MessageReceiver.sol

Storage gaps are a convention for reserving storage slots in a base contract, allowing future versions of that contract to use up those slots without affecting the storage layout of child contracts.

RECOMMENDATION

We recommend adding `_gap` variable to the initialization contract, it will be helpful for the future developments. Following the guide by the Openzeppelin: *https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#storage-gaps*

UPDATES

- **Jul 22, 2024**: This issue has been acknowledged and fixed by Avail team.

2.3.3. Lack of Fee Charging INFORMATIVE

Affected files:

- AvailBridge.sol

Users are required to pay a fee when bridging their messages from the Ethereum chain to the Avail chain. However, users are exempt from fees if they transmit AVAIL tokens, ETH values, or any ERC20 tokens.

Notably, there is no documentation within the Avail bridge specifying any fees for these transactions.

```
function sendAVAIL(bytes32 recipient,
    uint256 amount
) external whenNotPaused checkDestAmt(recipient, amount) {
    uint256 id;
    unchecked {
        id = messageId++;
    }
    Message memory message = Message(
        TOKEN_TX_PREFIX,
        bytes32(bytes20(msg.sender)),
        recipient,
        ETH_DOMAIN,
        AVAIL_DOMAIN,
        abi.encode(bytes32(0), amount),
        uint64(id)
    );
    // store message hash to be retrieved later by our light client
    isSent[id] = keccak256(abi.encode(message));

    emit MessageSent(msg.sender, recipient, id);
```

```
    avail.burn(msg.sender, amount);
}
```

Similar to the `sendAVAIL` function above, the `sendETH` and `sendERC20` functions also do not charge fees.

RECOMMENDATION

Avail team, it is advised to provide clarity regarding any applicable fees when users transfer tokens cross-chain. This will ensure transparency and help users make informed decisions during the transaction process.

UPDATES

- **Jul 22, 2024**: This is intentional, no fees are charged on asset transfers. So the severity of this issue is downgraded from LOW to INFO.

2.3.4. Restrict The Maximum Fee to Prevent User Mistakes INFORMATIVE

Affected files:

- AvailBridge.sol

In cases where users mistake send significant ETH values, these values are unintentionally treated as fees.

```
function sendMessage(bytes32 recipient, bytes calldata data) external payable whenNotPaused
{
    uint256 length = data.length;
    if (length == 0 || length > MAX_DATA_LENGTH) {
        revert InvalidDataLength();
    }
    // ensure that fee is above minimum amount
    if (msg.value < getFee(length)) {
        revert FeeTooLow();
    }
    uint256 id;
    unchecked {
        id = messageId++;
    }
    fees += msg.value;
    Message memory message = Message(
        MESSAGE_TX_PREFIX, bytes32(bytes20(msg.sender)), recipient, ETH_DOMAIN,
AVAIL_DOMAIN, data, uint64(id)
    );
    // store message hash to be retrieved later by our light client
    isSent[id] = keccak256(abi.encode(message));
```

```
    emit MessageSent(msg.sender, recipient, id);
}
```

RECOMMENDATION

Enforce a requirement that `msg.value` must not exceed the maximum fee, such as being capped at 10% of the result obtained from the getFee function.

UPDATES

- **Jul 22, 2024**: The assumption is that users will set the minimum possible fee when using the feature. The contract frontends take this into account and already have this built-in. Therefore, Avail team has decided not to fix this issue.

2.3.5. Require The `_bridge` Is Not The Zero Address INFORMATIVE

Affected files:

- Avail.sol

The contructor of the contract `Avail` does not validate zero addresses, potentially allowing for unintended operations or vulnerabilities.

We suggest adding a require statement to check that the address is not zero.

```
constructor(address _bridge) ERC20Permit("Avail") ERC20("Avail", "AVAIL") {
    // slither-disable-next-line missing-zero-check
    bridge = _bridge;
}
```

RECOMMENDATION

Implement a mechanism enabling users to retry failed transactions, ensuring robustness in message handling and transaction reliability.

UPDATES

- **Jul 22, 2024**: This issue has been acknowledged but not fixed.

2.3.6. Ensure That The function `initialize` Is Called at Deployment INFORMATIVE

Affected files:

- AvailBridge.sol

It is imperative to execute the initialize function during deployment to safeguard against potential initial hacks.

Initiating the function during deployment is crucial for establishing foundational security measures.

UPDATES

- **Jul 22, 2024**: This issue has been acknowledged but not fixed.

2.3.7. Emitting Events Following State Updates INFORMATIVE

Affected files:

- AvailBridge.sol

Functions mutating storage should emit events to facilitate off-chain monitoring.Events provide transparent, real-time insights into contract state changes, enabling stakeholders to track actions and detect anomalies.Without event emission, contract behavior becomes opaque, hindering monitoring and increasing the risk of errors or exploitation.

Following setters in the AvailBridge contract do not emit events: `updateVectorx`, `updateTokens`, `updateFeePerByte`, `updateFeeRecipient`.

We suggest the team should emit events for transparent monitoring.

UPDATES

- **Jul 22, 2024**: This issue has been acknowledged but not fixed.

2.3.8. Different leaf input format for `verifyBlobLeaf` and `verifyBridgeLeaf` functions INFORMATIVE

Affected files:

- AvailBridge.sol

The leaf input format for the two functions `verifyBlobLeaf` and `verifyBridgeLeaf` is different, which may cause confusion for users trying to use these functions directly. The `verifyBlobLeaf` function performs `keccak256` hashing on the leaf input before verifying the proof, while the `verifyBridgeLeaf` function implicitly requires the leaf to be `keccak256` hashed before calling the function.

```
function verifyBlobLeaf(MerkleProofInput calldata input) external view returns (bool) {
    if (input.blobRoot == 0x0) {
        revert BlobRootEmpty();
    }
    _checkDataRoot(input);
    // leaf must be keccak(blob)
    // we don't need to check that the leaf is non-zero because we hash the pre-image here
    return input.leafProof.verify(input.blobRoot, input.leafIndex,
```

```
keccak256(abi.encode(input.leaf)));
}

function verifyBridgeLeaf(MerkleProofInput calldata input) public view returns (bool) {
    if (input.bridgeRoot == 0x0) {
        revert BridgeRootEmpty();
    }
    _checkDataRoot(input);
    // leaf must be keccak(message)
    // we don't need to check that the leaf is non-zero because we check that the root is
non-zero
    return input.leafProof.verify(input.bridgeRoot, input.leafIndex, input.leaf);
}
```

RECOMMENDATION

Since these two functions have similar names and purposes, it is recommended to make the leaf input format consistent for both functions to avoid confusion.

UPDATES

- **Jul 22, 2024**: This is intentional as blob leaves are hashed twice, while messages are only hashed once, this is because they want to reveal the pre-image of both, so that blobs don't have to revealed and are completely distinct from messages. Therefore, Avail team has decided not to fix this issue.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*May 28, 2024*	Public Report	Verichains Lab
1.1	*Jul 22, 2024*	Public Report	Verichains Lab
1.2	*Jul 26, 2024*	Public Report	Verichains Lab

Table 3. Report versions history