QuillAudits

AUDIT REPORT

October 2025

For

Aventus

Table of Content

Executive Summary

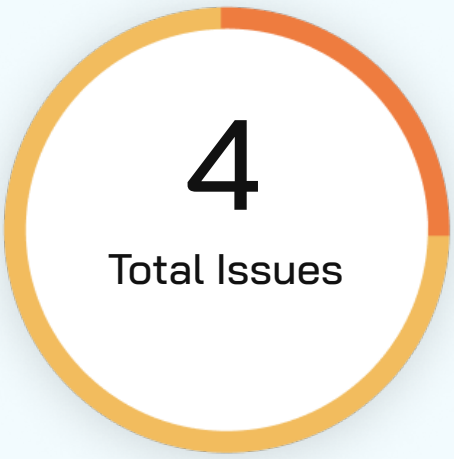
Project Name	AVNBridge
Protocol Type	Bridge
Project URL	https://aventus.io/
Overview	AVNBridge contract is a cross-chain bridge between Ethereum (T1) and the Aventus Network (T2), facilitating the secure transfer of ETH, ERC20, and ERC777 tokens while managing a set of validators called "authors." It allows the addition and removal of authors, the publication of T2 state roots to Ethereum, and the lifting or lowering of assets across chains. Security is enforced through EIP-712 signatures, quorum-based confirmations, and protections against replayed or duplicate transactions. Additionally, the contract is upgradeable via UUPS, owned through OwnableUpgradeable,and supports ERC777 tokens using the ERC1820 registry
Audit Scope	The scope of this Audit was to analyze the AVNBridge Smart Contracts for quality, security, and correctness.
Source Code Link	https://github.com/Aventus-Network-Services/avn-bridge
Branch	Main
Commit Hash	578d69535997d10de431f4c33c125060a81e594c
Contracts in Scope	AVNBridge.sol
Language	Solidity
Blockchain	Ethereum
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	08th October - 24th October
Updated Code Received	24th October
Review 2	24th October - 27th October

Fixed In	61476d2b3fdcdf01ee1ba9786a34e6086d92841a

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0 (0.0%)
High	0 (0.0%)
Medium	1 (25.0%)
Low	3 (75.0%)
Informational	0 (0.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	1	0
Partially Resolved	0	0	0	1	0
Resolved	0	0	1	1	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Inconsistent Address Check in rotateT1() Allows Overwriting Existing Author	Medium	Resolved
2	Centralization Risk Issue	Low	Partially Resolved
3	Recipient Can Indefinitely Lock It's Own Funds in Lower Claims Without Expiry	Low	Acknowledged
4	Unrotatable Author if address(0) is Registered During Initialization	Low	Resolved

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls

- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Medium Severity Issues

Inconsistent Address Check in rotateT1() Allows Overwriting Existing Author

Resolved

Path

AVNBridge.sol

Function Name

`rotateT1()`

Description

The rotateT1() function lacks validation to prevent rotating to an address that is already in use by another author. This inconsistency with other functions in the codebase can lead to corrupted state where multiple author IDs map to the same address.
Vulnerability Details
Missing Validation
The rotateT1() function only checks if the new address is address(0), but does not verify if the address is already registered to another author:

```
function rotateT1(uint256[] calldata ids, address[] calldata newAddresses)
external onlyOwner {
    // …
    for (uint256 I; I < rotations; I++) {
      id = ids[I];
      newAddress = newAddresses[I];

      if (newAddress == address(0)) revert AddressIsZero(); // ✅ Checks for
zero address

      // ❌ Missing: if (t1AddressToId[newAddress] != 0) revert
T1AddressInUse(newAddress);

      oldAddress = idToT1Address[id];
      if (oldAddress == address(0)) revert NotAnAuthor();
      t1AddressToId[oldAddress] = 0;
      idToT1Address[id] = newAddress;
      t1AddressToId[newAddress] = id; // Overwrites existing mapping if address
in use
    }
  }
```

Inconsistency with Other Functions
Other functions in the contract properly validate against duplicate addresses. For example, in _initialiseAuthors():

```
function _initialiseAuthors(…) private {
    // …
    do {
      t1Address = t1Addresses[I];
      t1PubKey = abi.encode(t1PubKeysLHS[I], t1PubKeysRHS[I]);
      if (address(uint160(uint256(keccak256(t1PubKey)))) != t1Address) revert
AddressMismatch();
      if (t1AddressToId[t1Address] != 0) revert T1AddressInUse(t1Address); //
```

✅ `Checks for duplicate`

```
      _activateAuthor(_addNewAuthor(t1Address, t2PubKeys[I]));
      // …
    } while (I < numAuth);
  }
```

Impact

Accidentally or otherwise rotates to an address that's already in use leading to multiple issue: Broken bidirectional mapping, Loss of access points to a different ID, state inconsistency e.t.c. High

Likelihood

Low, only admin can make this error

Recommendation

Add the same duplicate address check used in other functions:

```
function rotateT1(uint256[] calldata ids, address[] calldata newAddresses)
external onlyOwner {
    uint256 rotations = ids.length;
    if (rotations != newAddresses.length) revert MissingKeys();

    uint256 id;
    address newAddress;
    address oldAddress;

    for (uint256 I; I < rotations; I++) {
      id = ids[I];
      newAddress = newAddresses[I];
      if (newAddress == address(0)) revert AddressIsZero();
+     if (t1AddressToId[newAddress] != 0) revert T1AddressInUse(newAddress); //
```

✅ `Add duplicate check`

```
      oldAddress = idToT1Address[id];
      if (oldAddress == address(0)) revert NotAnAuthor();
      t1AddressToId[oldAddress] = 0;
      idToT1Address[id] = newAddress;
      t1AddressToId[newAddress] = id;
    }
  }
```

Low Severity Issues

Centralization Risk Issue

Partially Resolved

Path

AVNBridge.sol

Description

The contract owner has unilateral control over critical bridge functions through toggleAuthors(), toggleLifting(), and toggleLowering(). A compromised or malicious owner can instantly disable all bridge operations without any timelock, multi-signature requirement, or delay mechanism, effectively freezing user funds and halting the bridge permanently.

```
function toggleAuthors(bool state) external onlyOwner {
    authorsEnabled = state;
    emit LogAuthorsEnabled(state);
}

function toggleLifting(bool state) external onlyOwner {
    liftingEnabled = state;
    emit LogLiftingEnabled(state);
}

function toggleLowering(bool state) external onlyOwner {
    loweringEnabled = state;
    emit LogLoweringEnabled(state);
}
```

Impact

Permanent Fund Lockup: Users who have lifted tokens to T2 cannot lower them back to T1 if loweringEnabled is set to false. Their funds become permanently inaccessible on T1.
DoS on All Bridge Operations: All cross-chain transfers can be halted instantly:

- Users cannot lift new tokens (reverts with LiftDisabled)
- Users cannot claim existing lower proofs (reverts with LowerDisabled)
- Authors cannot perform governance actions (reverts with AuthorsDisabled)

No Recovery Mechanism: The contract provides no automatic recovery or maximum pause duration. If the owner key is lost or the owner becomes uncooperative, the bridge is permanently bricked.
Financial Loss: Users with funds in-transit or already on T2 face complete loss of access to their T1 assets.

Likelihood

Low

Recommendation

Replace single owner with multi-signature wallet (3-of-5 Gnosis Safe) requiring multiple approvals for critical operations.

Set long-term goal to transition ownership to decentralized governance (DAO) for community-controlled bridge parameters.

AVNBridge Team's Comment

The owner is already a multisig wallet. For the past few years the bridge has been owned by this multisig: 0x79eb961929357650c8aE7E3f00684636db9A1241.
Recently, ownership was transferred to the new Aventus Community Safe: 0xdEC28A1398bF876eCDB9C59B6022964565F7522D.

Recipient Can Indefinitely Lock It's Own Funds in Lower Claims Without Expiry

Acknowledged

Function Name

`_releaseFunds()`

Description

The claimLower() function lacks an expiry mechanism, allowing malicious recipients to indefinitely prevent its fund withdrawal when receiving native token. This creates a griefing vector where it's funds remain kept in the contract and in the protocol accounting for as long as the recipient wishes with no way for other parties to force execution or reclaim them.

The lower claiming process has two key characteristics that enable this issue:

No expiry validation - Unlike other T2-initiated operations (addAuthor, removeAuthor, publishRoot) which include withinCallWindow(expiry) checks, claimLower() has no time restriction:

```
function claimLower(bytes calldata lowerProof) external whenLowerEnabled lock {
    (address token, uint256 amount, address recipient, uint32 lowerId) =
_extractLowerData(lowerProof);
    if (recipient == address(0)) revert AddressIsZero();
    _processLower(token, amount, recipient, lowerId, lowerProof);
    _releaseFunds(token, amount, recipient); // ❌ Can be blocked by recipient

    emit LogLowerClaimed(lowerId);
}
```

Payment pattern that can be rejected by the recipient:

```
function _releaseFunds(address token, uint256 amount, address recipient)
private {
    if (token == PSEUDO_ETH_ADDRESS) {
        (bool success, ) = payable(recipient).call{ value: amount }('');
        if (!success) revert PaymentFailed(); // ❌ Recipient can force this
revert
    }

}
```

Impact

A griefing vector where it's own funds remain kept in the contract and in the protocol accounting for as long as the recipient wishes with no way for other parties to force execution or reclaim them

Likelihood

Low. Attacker can only grief their own funds (Self-inflicted harm)

Recommendation

Document current behavior - If the current behavior is intentional, clearly document it. Otherwise, consider addressing it.

AVNBridge Team's Comment

A mechanism is being introduced in the next upgrade that allows the intended recipient of any unclaimed lower to revert it, returning the funds to the original sender. If the recipient address is invalid or unreachable, the contract owner will be able to step in and revert the lower, after a defined expiry period. This feature also requires an update to the T2 contract to include the sender address in the lower data and will be released once that change has been deployed.

Unrotatable Author if address(0) is Registered During Initialization

Resolved

Description

If address(0) is successfully registered as an author during initialization (either through a cryptographic collision or admin error), it becomes permanently unrotatable due to a check in rotateT1() that prevents rotation when oldAddress == address(0).

Root Cause
The _initialiseAuthors() function lacks an explicit check to prevent address(0) from being registered:

```
function _initialiseAuthors(
    address[] calldata t1Addresses,
    bytes32[] calldata t1PubKeysLHS,
    bytes32[] calldata t1PubKeysRHS,
    bytes32[] calldata t2PubKeys
  ) private {
    // …
    do {
      t1Address = t1Addresses[I];
      t1PubKey = abi.encode(t1PubKeysLHS[I], t1PubKeysRHS[I]);
      if (address(uint160(uint256(keccak256(t1PubKey)))) != t1Address) revert
AddressMismatch();
      if (t1AddressToId[t1Address] != 0) revert T1AddressInUse(t1Address); //
❌ Passes for address(0)
      _activateAuthor(_addNewAuthor(t1Address, t2PubKeys[I]));
      // …
    } while (I < numAuth);
  }
```

The check if (t1AddressToId[t1Address] != 0) will not revert for address(0) because:

Uninitialized mapping values default to 0
t1AddressToId[address(0)] returns 0
The condition 0 != 0 evaluates to false, so execution continues

Registration Process
If address(0) passes the address mismatch check, _addNewAuthor() will register it normally:

```
function _addNewAuthor(address t1Address, bytes32 t2PubKey) private returns
(uint256 id) {
    unchecked {
      id = nextAuthorId++;
    }
    // …
    idToT1Address[id] = t1Address;  // Sets idToT1Address[id] = address(0)
    t1AddressToId[t1Address] = id;  // Sets t1AddressToId[address(0)] = id
    // …
  }
```

Permanent Lock
Once address(0) is registered, it cannot be rotated due to this check in rotateT1():

```
function rotateT1(uint256[] calldata ids, address[] calldata newAddresses)
external onlyOwner {
    // …
    for (uint256 I; I < rotations; I++) {
      id = ids[I];
      newAddress = newAddresses[I];
      if (newAddress == address(0)) revert AddressIsZero();
      oldAddress = idToT1Address[id];
      if (oldAddress == address(0)) revert NotAnAuthor(); // ❌ Always reverts
      // …
    }
  }
```

The if (oldAddress == address(0)) check was likely intended to verify the author exists, but it creates an irrecoverable state when address(0) is the registered address.

Impact

slot becomes permanently locked and cannot be rotated, the system cannot remove or replace a compromised/invalid author entry
leading the unexpected behaviour and protocol disruption

Likelihood

Low

Recommendation

Add explicit address(0) validation in _initialiseAuthors():

```
function _initialiseAuthors(
    address[] calldata t1Addresses,
    bytes32[] calldata t1PubKeysLHS,
    bytes32[] calldata t1PubKeysRHS,
    bytes32[] calldata t2PubKeys
  ) private {
    uint256 numAuth = t1Addresses.length;
    if (numAuth < MINIMUM_AUTHOR_SET) revert NotEnoughAuthors();
    if (t1PubKeysLHS.length != numAuth || t1PubKeysRHS.length != numAuth ||
t2PubKeys.length != numAuth) revert MissingKeys();

    bytes memory t1PubKey;
    address t1Address;
    uint256 I;

    do {
      t1Address = t1Addresses[I];
+     if (t1Address == address(0)) revert AddressIsZero(); // ✅ Explicit check
      t1PubKey = abi.encode(t1PubKeysLHS[I], t1PubKeysRHS[I]);
      if (address(uint160(uint256(keccak256(t1PubKey)))) != t1Address) revert
AddressMismatch();
      if (t1AddressToId[t1Address] != 0) revert T1AddressInUse(t1Address);
      _activateAuthor(_addNewAuthor(t1Address, t2PubKeys[I]));
      unchecked {
        ++I;
      }
    } while (I < numAuth);
  }
```

This prevents the possibility of the unrotatable author scenario entirely

Functional Tests

Some of the tests performed are mentioned below:

- Verified address rotation logic properly maps new T1 addresses to existing author IDs and reverts on invalid or zero addresses.
- Validated correct addition of new authors, proof verification, and state updates; reverts on duplicate or invalid keys.
- Confirmed author removal correctly decreases active count, respects minimum quorum, and emits accurate event data.
- Verified Merkle root publishing updates isPublishedRootHash and reverts on reuse or invalid confirmations.
- Confirmed ERC20 lift transfers tokens successfully and reverts on failed transfers, limit breaches, or disabled state.
- Tested ETH lift emits accurate event and rejects zero-value transactions.
- tokensReceived() – Verified ERC777 hook properly emits LogLifted, reverts on invalid sender, key length, or locked state.
- Validated lower proof parsing, confirmation counting, and accurate return of all verification flags and proof validity.
- Confirmed correct fund release and marking of proof usage; reverts on repeated claims or invalid proofs.
- Tested all transaction states (Succeeded, Pending, Failed) against expiry and txId usage conditions.
- Confirmed internal mappings and author counters update correctly for valid new entries.
- Confirmed duplicate lower prevention and proper invocation of confirmation verification.
- Verified ERC20, ERC777, and ETH fund releases handle fallback safely and revert on payment failures.
- Verified unique txId storage and reverts on reuse.
- Confirmed correct signature validation logic, activation of inactive authors, and revert on insufficient confirmations.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats
AVNBridge.sol	addAuthor	Every transaction gets a **unique, incrementing transaction ID(t2TxId) that can prevent reply attacks.** Malicious author monitors mempool and sees this pending transaction Malicious author front-runs by submitting addAuthor() with higher gas price to add a new author Suppose a user sets an expiry of 100 days while adding themselves as an author. After successfully adding, they decide a few hours later to remove themselves by calling removeAuthor(). What happens then
	removeAuthor	verifies signatures via _verifyConfirmations, ensures the author exists, checks quorum constraints, and updates isAuthor and authorIsActive states.
	publishRoot	Non-packed ABI encoding can cause collision
	lift	The function succeeds, but the user gets credit for less than intended, because the contract measures balance differences, not the input amount. This protects against fee tokens but can confuse users.

Contract	Function	Threats
AVNBridge.sol	addAuthor	Non-standard ERC-20 (USDT-like) that doesn't return bool: SafeERC20.safeTransferFrom handles this. Check where fundflow transfers and also check userflow.
	liftETH	Check any type of the re-entrancy attack possible Unlike ERC20, doesn't enforce T2_TOKEN_LIMIT
	tokensReceived	Check any type of the re-entrancy attack possible Check ERC 777 compliant
	claimLower	Check any type of the re-entrancy attack possible

Trust Assumptions

- Each t2TxId or transaction identifier is assumed to be unique across all activities (e.g., add/remove author, root publish).
- Off-chain components are assumed to correctly compute and provide valid root hashes and confirmations.
- The owner entity is trusted to manage author sets and configuration parameters responsibly
- Authors are assumed to behave honestly and provide valid signatures during consensus.
- Backend ensures timely synchronization between Tier 1 and Tier 2 within the defined call window.
- Incorrect Amount Emitted for Fee-on-Transfer ERC777 Tokens: assumes that the token
- integrated with the system is EIP-777 compliant, ensuring correct amount emission for fee-on-transfer tokens.

Closing Summary

In this report, we have considered the security of **AVNBridge**. We performed our audit according to the procedure described above.

1 issue of Medium severity and 3 issues of Low severity were found. **The AVNBridge team resolved two, partially resolved one and acknowledged the remaining issue.**

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

October 2025

For

Aventus

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com