t3rn - Guardian

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **December 11th, 2023 - December 22nd, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	12/21/2023
0.2	Draft Review	12/22/2023
1.0	Remediation Plan	02/12/2024
1.1	Remediation Plan Review	03/07/2024
1.2	Remediation Plan Review	03/07/2024

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Ferran Celades	Halborn	Ferran.Celades@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

t3rn engaged Halborn to conduct a security assessment on their smart contracts beginning on December 11th, 2023 and ending on December 22nd, 2023 . The security assessment was scoped to the smart contracts provided in the t3rn/guardian GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

In summary, Halborn identified some security risks that were successfully addressed by the t3rn team. Here are the key findings and recommendations:

- Issue with Quorum Calculation in Constructor: A flaw was identified in the quorum calculation for single committee members, resulting in a quorum value of zero. It is recommended to implement a conditional check to ensure correct quorum setting.
- Permanent Committee Hash Equality in receiveAttestationBatch: Discovered a logic issue that causes currentCommitteeHash and nextCommitteeHash to become permanently equal, thereby locking the committee update mechanism. A two-step committee update process with proper validation is suggested.
- Redundant Balance Check in LocalExchange: Noted a redundant balance check in the LocalExchange contract. Removing the user.balance >= amount check for native token transactions is recommended to align with Ethereum's balance deduction mechanism.
- Asset Validation Missing in RemoteOrder: Detected the absence of asset validation in the orderMemoryData function of the RemoteOrder contract. Advised including a check to validate rewardAsset against asset, ensuring consistency across different functions.
- Mismatched Functionality in claimRefund and claimPayout: Uncovered an inconsistency between claimRefund and claimPayout due to

the withdrawFromVault function. Modifying withdrawFromVault to an internal function with additional parameters to distinctly handle refunds and payouts is recommended.

- Vulnerability in Delegated Calls in EscrowGMP: Found a critical issue with the use of delegatecall in the EscrowGMP contract. Proposed replacing delegatecall with call and implementing a whitelist of trusted contracts for enhanced security.
- Missing Validity Check in storeEscrowCallOrder: Identified a missing check for existing EscrowCall data in storeEscrowCallOrder, posing a risk of data overwriting. Adding a validity check within the function is recommended.
- Potential ERC20 Token Exploit in LocalExchange.localOrder: Noted a potential vulnerability in handling ERC20 token rewards in LocalExchange.localOrder. Transferring reward tokens at the order creation stage to mitigate this risk is suggested.

Each of these findings has been thoroughly investigated, with specific recommendations provided to address the identified issues. Implementing these changes will significantly enhance the overall security and reliability of the smart contracts in question.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose
- Smart contract manual code review and walkthrough

- Graphing out functionality and contract logic/connectivity/functions (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes
- Manual testing by custom scripts
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Brownie, Remix IDE, Foundry)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

EXECUTIVE OVERVIEW

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

The security assessment was scoped to the following smart contracts:

- attestationsVerifierProofs.sol
- TRNToken.sol
- localExchange.sol
- t3USD.sol
- t3DotToken.sol
- ERC20Mock.sol
- escrowGMP.sol
- t3SOLToken.sol
- remoteOrder.sol
- t3BTCToken.sol

COMMIT ID: bd784cde3b37773b062288925b2015a0c8a9806b

OUT-OF-SCOPE:

- Third-party libraries and dependencies.
- Economic attacks.

REMEDIATION COMMIT IDs:

- 8609f3d41577f694d6a8b266e6e7f351c23c8691
- 69eb56161b5e8f2005da9831db743bb9eab94116
- 27b50011bdd991c913b3b44b40e2dee2fea54061
- cb47d8f4c7ed4027a101ceb79c4f9f7effe60daa
- e89b63190c6c6a842d10f9ddc6a5ac34f1e782ac
- dbf474829647e9ee14be13453314aa63b1bd7555
- 54543ff374738c3448792415a77d8ba60789c382
- 077d90b34310d79f54d7af9e7f7bfb1d1f463798
- d4d47859f34a1752e4a39e7f79bdc29ba6ec5f05

OUT-OF-SCOPE IN REMEDIATION PLAN:

- New features after/within the remediation.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	3	0	2	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) REENTRANCY RISK IN executeLocalOrder WITH ERC677/ERC223 TOKENS	Critical (10)	SOLVED - 01/09/2024
(HAL-02) DELEGATECALL TO UNTRUSTED CONTRACT IN ESCROWGMP	Critical (10)	SOLVED - 12/13/2023
(HAL-03) BYPASSING CLAIM REFUND WAIT PERIOD	Critical (10)	SOLVED - 01/09/2024
(HAL-04) ASSET VALIDATION MISSING	High (7.5)	SOLVED - 01/31/2024
(HAL-05) POTENTIAL ERC20 TOKEN EXPLOIT	High (7.5)	SOLVED - 01/31/2024
(HAL-06) QUORUM CALCULATION ISSUE IN CONSTRUCTOR WITH SINGLE COMMITTEE MEMBER	High (7.0)	SOLVED - 01/09/2024
(HAL-07) MISSING VALIDITY CHECK	Low (2.1)	SOLVED - 01/09/2024
(HAL-08) REDUNDANT BALANCE CHECK	Low (2.5)	SOLVED - 01/09/2024
(HAL-09) PERMANENT COMMITTEE HASH LOCK	Informational (1.3)	SOLVED - 01/09/2024

FINDINGS & TECH DETAILS

4.1 (HAL-01) REENTRANCY RISK IN executeLocalOrder WITH ERC677/ERC223 TOKENS - CRITICAL(10)

Description:

In the executeLocalOrder function of the LocalExchange contract, there exists a potential reentrancy risk when dealing with ERC677, ERC223, or similar tokens. These token standards implement a callback mechanism that is triggered when tokens are transferred to a contract. If the recipient of a transfer in executeLocalOrder is a contract, this callback could be exploited to re-enter the executeLocalOrder function.

This reentrancy risk is particularly concerning because if the allowance for the token is greater than the amount required for a single order, an attacker could potentially trigger the same order multiple times. The current implementation sets localOrders[local_order_id] = false after the transfer, which leaves a window for reentrancy attacks.

BVSS:

AO:A/AC:L/AX:M/C:N/I:C/A:N/D:C/Y:C/R:N/S:U (10)

Recommendation:

To mitigate this reentrancy risk, it is recommended to employ the cause-effect pattern by setting localOrders[local_order_id] = false immediately after the order existence check and before any external calls or token transfers. This change ensures that even if a reentrancy occurs, the state change (order execution marking) has already been done, preventing repeated execution of the same order.

The revised sequence in executeLocalOrder should be:

1. Check if the order exists.

2. Mark the order as executed (localOrders[local_order_id] = false).
3. Proceed with the token transfer and other function logic.

Implementing this change will protect against reentrancy attacks in scenarios involving tokens with callback mechanisms, thereby enhancing the security of the executeLocalOrder function within the LocalExchange contract.

Remediation Plan:

SOLVED: The issue was solved as suggested on https://github.com/t3rn/guardian/pull/178/commits/8609f3d41577f694d6a8b266e6e7f351c23c8691

4.2 (HAL-02) DELEGATECALL TO UNTRUSTED CONTRACT IN ESCROWGMP - CRITICAL(10)

Description:

The EscrowGMP smart contract contains a critical vulnerability in its implementation of commitEscrowCall and revertEscrowCall functions. These functions utilize delegatecall to execute code from an untrusted external contract within the context of the EscrowGMP contract. This is dangerous as it allows the external contract to execute any code with the privileges of EscrowGMP, including modifying its state and accessing its funds.

The current implementation attempts to mitigate risks by checking for self-destruct calls through isSelfDestructCall. However, this method is flawed. It checks for a specific method signature, assuming that a self-destruct operation would always be invoked through a function with a recognizable signature. In reality, the SELFDESTRUCT opcode can be included in any function, regardless of its name or signature. Thus, an attacker can include the SELFDESTRUCT opcode in a function with a different name and bypass the check.

BVSS:

AO:A/AC:L/AX:M/C:N/I:C/A:C/D:M/Y:M/R:N/S:U (10)

Recommendation:

To mitigate this vulnerability, it is recommended to replace delegatecall with call in both the commitEscrowCall and revertEscrowCall functions. This change will ensure that the executed code operates in the context of the calling contract rather than EscrowGMP. Thus, even if the called contract contains malicious code, it will not have direct access to modify the state or access the funds of EscrowGMP.

Additionally, consider implementing a whitelist of trusted contracts or a mechanism to verify the trustworthiness of the contract being called. This approach can add an extra layer of security by ensuring that only vetted and safe contracts can interact with EscrowGMP.

It's important to note that the decision between using call and maintaining delegatecall should be aligned with the intended design and functionality of the contract. If the design requires preserving the context of EscrowGMP, then rigorous checks and a system to ensure the safety of the external code must be implemented.

Remediation Plan:

SOLVED: The issue was solved as suggested on https://github.com/t3rn/guardian/commit/69eb56161b5e8f2005da9831db743bb9eab94116

4.3 (HAL-03) BYPASSING CLAIM REFUND WAIT PERIOD - CRITICAL(10)

Description:

In the RemoteOrder contract, the functions claimRefund and claimPayout demonstrate a critical inconsistency in their functionality. This issue arises due to the withdrawFromVault function, which both claimRefund and claimPayout call. The withdrawFromVault function utilizes an || (logical OR) operator in its condition checks, which does not differentiate between a refund or a payout. This design flaw allows for a scenario where an ID marked for refund could still process a payout and vice versa.

Furthermore, withdrawFromVault is publicly accessible, allowing direct calls that bypass the intended logic of claimRefund and claimPayout. This opens a vulnerability where withdrawals can be made without adhering to the specific conditions meant for refunds or payouts.

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:N/Y:L/R:N/S:C (10)

Recommendation:

To address this vulnerability, several modifications are recommended:

1. **Change withdrawFromVault to Internal**: Restricting the visibility of withdrawFromVault to internal will prevent direct public calls, ensuring that withdrawals can only occur through the intended claimRefund or claimPayout functions.
2. **Differentiate Refund and Payout Logic**: Modify withdrawFromVault to accept an additional parameter or flag that indicates whether the operation is a refund or a payout. This modification allows for distinct handling of each case.

3. **Separate Hash Checks**: Implement separate hash checks within withdrawFromVault for refunds and payouts, based on the additional parameter/flag. This approach will ensure that an ID marked for refund cannot be used for a payout and vice versa.

4. **Adjust claimRefund and claimPayout Accordingly**: Modify claimRefund and claimPayout to pass the appropriate flag or parameter to withdrawFromVault, aligning with their respective intended functionalities.

These changes will enforce a clear distinction between refunds and payouts, ensuring that each process adheres strictly to its defined conditions and enhancing the overall security and integrity of the RemoteOrder contract's operations.

Remediation Plan:

SOLVED: The issue was solved by splitting the functionality into two different functions on https://github.com/t3rn/guardian/pull/178/commits/27b50011bdd991c913b3b44b40e2dee2fea54061

4.4 (HAL-04) ASSET VALIDATION MISSING - HIGH (7.5)

Description:

The RemoteOrder contract has a vulnerability in its orderMemoryData function, which is publicly accessible and does not validate the rewardAsset against the asset. While the remoteBridgeAsset function, which internally calls orderMemoryData, correctly checks supportedBridgeAssetsHereToThere to validate bridge assets, orderMemoryData lacks this validation. This discrepancy can lead to potential issues when orderMemoryData is called directly, bypassing the checks implemented in remoteBridgeAsset.

In the remoteBridgeAsset function, the asset and rewardAsset are intended to be equivalent, as indicated by the abi.encode call, which uses the same assetHere for both asset and rewardAsset. However, when orderMemoryData is called directly, there is no such guarantee, potentially leading to inconsistencies or unintended behavior.

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (7.5)

Recommendation:

To mitigate this vulnerability, the orderMemoryData function should be modified to include validation for the rewardAsset similar to the checks done in remoteBridgeAsset. This can be achieved by incorporating the supportedBridgeAssetsHereToThere mapping within orderMemoryData to ensure that the rewardAsset matches the expected asset.

The proposed changes include:

1. **Asset Validation in orderMemoryData**: Add a check in orderMemoryData to validate that the rewardAsset is a supported bridge asset corresponding to the given asset. This can be done

by comparing the rewardAsset with the value obtained from supportedBridgeAssetsHereToThere for the specified asset.

2. **Restrict Direct Access if Necessary**: If orderMemoryData is not intended for public use, consider changing its visibility to internal to prevent direct calls that bypass the validation logic in remoteBridgeAsset.

By implementing these changes, the contract will ensure consistency in asset handling across different functions, enhancing the security and reliability of the RemoteOrder contract's operations.

Remediation Plan:

SOLVED: The issue was solved on https://github.com/t3rn/guardian/pull/178/commits/cb47d8f4c7ed4027a101ceb79c4f9f7effe60daa

4.5 (HAL-05) POTENTIAL ERC20 TOKEN EXPLOIT - HIGH (7.5)

Description:

The LocalExchange contract's localOrder function presents a potential vulnerability concerning ERC20 token rewards. The function does not transfer the ERC20 rewardToken at the time of order creation, relying instead on the allowance mechanism. This design choice can lead to an exploit scenario, particularly when a user removes their allowance for a token after placing an order but before the order is executed. The steps for this exploit scenario are as follows:

1. **Order Placement**: A user places an order using localOrder, specifying an ERC20 rewardToken and setting a sufficient allowance for LocalExchange to transfer the reward amount.
2. **Allowance Removal**: The user then removes the allowance for the rewardToken after placing the order.
3. **New Order Placement**: The user places a new order with the same rewardToken without cancelling the previous order.
4. **Execution by Attacker**: An attacker tracks non-cancelled orders with enough initial allowance and triggers the execution of these orders.
5. **Unintended Execution**: The initial user might believe that by removing the allowance, they have effectively cancelled their previous orders. However, due to the initial sufficient allowance, the contract will still execute these orders.

This scenario creates a risk where the user's ERC20 tokens could be unexpectedly transferred as rewards for old orders they assumed were cancelled.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:M/Y:M/R:N/S:U (7.5)

Recommendation:

To mitigate this vulnerability, it is recommended to modify the localOrder function to transfer the ERC20 rewardToken at the time of order creation, similar to the handling of native tokens. This approach ensures that the reward tokens are securely held within the LocalExchange contract until the order is executed or refunded. Consequently, the claimRefund function should also be adjusted to return these tokens to the user if the order is not executed within the specified timeout period.

The proposed changes include:

1. **Transfer Reward Tokens on Order Creation**: Modify localOrder to transfer the specified rewardToken amount from the user to the LocalExchange contract.
2. **Refund Tokens on Claim**: Adjust claimRefund to transfer back the rewardToken amount to the user if the order times out without being executed.

These modifications will prevent users from unintentionally executing old orders due to allowance manipulations and ensure a more secure and predictable order management process within the LocalExchange contract.

Remediation Plan:

SOLVED: The issue was solved on https://github.com/t3rn/guardian/pull/178/commits/e89b63190c6c6a842d10f9ddc6a5ac34f1e782ac

4.6 (HAL-06) QUORUM CALCULATION ISSUE IN CONSTRUCTOR WITH SINGLE COMMITTEE MEMBER - HIGH (7.0)

Description:

The issue arises in the constructor of a contract (not specified but inferred from the context), where a `quorum` is calculated as 2/3 of the `initialCommittee.length`. When the `initialCommittee` contains only one member, this calculation results in a `quorum` of 0. This creates a significant vulnerability in scenarios requiring committee member attestations, as no valid signatures would be needed to meet the quorum requirements. This same issue arises in the `updateCommitteeSize` function.

Despite this, any proofs relying on the quorum would still fail, as the expected number of signatures would not be met. This discrepancy between the quorum calculation and the actual proof validation logic can lead to unexpected behavior and potential security risks.

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:M/D:N/Y:N/R:P/S:C (7.0)

Recommendation:

To mitigate this vulnerability, the constructor logic and `updateCommitteeSize` should be modified to handle edge cases where the `initialCommittee` has a minimal number of members. Specifically, for a committee of one member, the quorum should be set to 1 instead of 0. The constructor logic could be updated with a conditional check to handle this case appropriately.

The revised constructor could include the following logic:

Listing 1

```
if (initialCommittee.length == 1) {
    quorum = 1;
} else {
    quorum = initialCommittee.length * 2 / 3;
}
```

This change ensures that the quorum calculation aligns with the intended functionality of requiring a majority of the committee members' attestations for validation. It prevents the scenario where no signatures are required for a quorum, thereby maintaining the integrity of the attestation process.

Remediation Plan:

SOLVED: The issue was solved as suggested on https://github.com/t3rn/guardian/pull/178/commits/dbf474829647e9ee14be13453314aa63b1bd7555

4.7 (HAL-07) MISSING VALIDITY CHECK - LOW (2.1)

Description:

The storeEscrowCallOrder function in the EscrowGMP contract lacks a critical check to verify if an EscrowCall with the given id already exists. This function is responsible for storing EscrowCall data associated with a unique identifier (id). Without a check to determine if the id is already in use, there is a risk that existing EscrowCall data could be overwritten. This could lead to potential data integrity issues, where a malicious or erroneous call could replace valid EscrowCall data.

The current implementation does not provide clarity on who is responsible for calling storeEscrowCallOrder, and without context, it's difficult to assess the potential impact fully. However, the ability to overwrite existing EscrowCall data without any validation poses a risk of unauthorized manipulation of contract data.

BVSS:

AO:S/AC:L/AX:L/C:N/I:C/A:N/D:L/Y:N/R:N/S:U (2.1)

Recommendation:

To address this vulnerability, it is recommended to implement a validity check within the storeEscrowCallOrder function. This can be achieved by adding a boolean flag or a validity status within the EscrowCall struct to indicate whether the data is valid or has been set. Alternatively, a separate mapping can be used to track whether an id has already been associated with an EscrowCall.

A possible implementation is as follows:

1. **Add a Validity Flag**: Extend the EscrowCall struct to include a boolean flag, such as isValid, which indicates whether the data is

valid or initialized.

2. **Check Validity on Insertion**: Modify storeEscrowCallOrder to first check if the EscrowCall for the given id is already marked as valid. If it is, reject the operation to prevent overwriting existing data.

3. **Maintain a Separate Mapping**: Alternatively, create a new mapping (e.g., mapping(bytes32 => bool)public escrowCallExists) that tracks whether an id has been used. Before storing a new EscrowCall, check this mapping to ensure the id is not already in use.

By implementing these checks, the contract can prevent unauthorized overwriting of existing data, thus preserving the integrity and intended functionality of the EscrowGMP contract.

Remediation Plan:

SOLVED: The issue was solved as suggested on https://github.com/t3rn/guardian/pull/178/commits/54543ff374738c3448792415a77d8ba60789c382

4.8 (HAL-08) REDUNDANT BALANCE CHECK - LOW (2.5)

Description:

The ensureBalanceAndAllowance modifier in the LocalExchange contract includes a redundant balance check when dealing with native tokens (where token == address(0)). This check, user.balance >= amount, is intended to ensure the user (typically msg.sender) has enough native token balance to proceed with the transaction. However, in scenarios where msg.sender is the user and msg.value is the amount being transferred, this check is not only unnecessary but also incorrectly implemented.

When a user sends native tokens to a contract, the amount (msg.value) is automatically deducted from the user's balance before the contract code is executed. Therefore, the current check effectively requires the user to have twice the necessary amount: once in msg.value and the same amount still in their balance. This implementation results in a higher and incorrect balance requirement for users.

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

To address this issue, the ensureBalanceAndAllowance modifier should be modified to remove the user.balance >= amount check for native token transactions. The correct check in this context is to ensure that the msg.value matches the amount required for the transaction, which is already being done with require(msg.value == amount, "Mismatched deposit execution amount of native").

The modified modifier would look like this:

Listing 2

```
modifier ensureBalanceAndAllowance(
    address token,
    address user,
    uint256 amount
) {
    // Ensure the user has enough balance and allowance for the token.
    if (token == address(0)) {
        require(msg.value == amount, "Mismatched deposit execution amount of native");
    } else {
        require(IERC20(token).balanceOf(user) >= amount, "Insufficient user balance");
        require(IERC20(token).allowance(user, address(this)) >= amount, "Insufficient user allowance");
    }
    _;
}
```

This change ensures that the contract correctly checks the amount sent for native token transactions without imposing an unnecessarily high balance requirement on the user.

Remediation Plan:

SOLVED: The issue was solved as suggested on https://github.com/t3rn/guardian/pull/178/commits/077d90b34310d79f54d7af9e7f7bfb1d1f463798

4.9 (HAL-09) PERMANENT COMMITTEE HASH LOCK - INFORMATIONAL (1.3)

Description:

The receiveAttestationBatch function in the given contract contains a logic flaw in handling committee updates. When batch.maybeNextCommittee.length > 0, the function sets both currentCommitteeHash and nextCommitteeHash to impliedNextCommitteeHash. The condition checks if the nextCommitteeHash equals impliedNextCommitteeHash before updating, which ensures that nextCommitteeHash is set correctly the first time. However, subsequent calls to the function will always find currentCommitteeHash and nextCommitteeHash to be equal, effectively locking the committee update mechanism. This results in a scenario where currentCommitteeHash and nextCommitteeHash will perpetually remain the same after the first committee change, barring any future updates to the committee.

This issue could have significant implications for the contract's functionality, particularly in systems relying on committee consensus or periodic updates to the committee composition.

BVSS:

AO:S/AC:L/AX:L/C:N/I:C/A:C/D:L/Y:N/R:P/S:U (1.3)

Recommendation:

To resolve this issue, the contract needs a mechanism to update the nextCommitteeHash independently of the currentCommitteeHash, along with a validation process for any new committee proposed in the attestation batch. The recommended changes include:

1. **Separate Update Mechanisms**: Implement distinct processes for updating the currentCommitteeHash and nextCommitteeHash. Ensure that

nextCommitteeHash can be updated independently, allowing for future committee transitions.

2. **Committee Validation**: Introduce a validation process for any new committee hash proposed. This could involve signature verification or other cryptographic methods to ensure the legitimacy and integrity of the new committee.
3. **Conditional Logic Adjustment**: Modify the conditional logic in receiveAttestationBatch to prevent setting both hashes to the same value inadvertently. Ensure that the update of one does not automatically lead to the update of the other unless explicitly intended and validated.

By implementing these changes, the contract will maintain the flexibility to transition between committees securely and as intended, ensuring the ongoing integrity and functionality of the attestation process.

Remediation Plan:

SOLVED: The issue was solved by storing the next committee hash based on the batch data https://github.com/t3rn/guardian/pull/178/commits/d4d47859f34a1752e4a39e7f79bdc29ba6ec5f05

REVIEW NOTES

5.1 TRN / t3USD / t3SOL / t3DOT / t3BTC

- Standard ERC20 token with mint functionality restricted to only owner.
- The owner is the deployer of the contract.

5.2 EscrowGMP

- The owner is the deployer of the contract.
- `assignAttesters` and `assignOrderer` are restricted to the owner. They enforce the callers of functions using the `onlyAttesters` and `onlyOrderer` modifiers to the set address.
- `storeRemoteOrderPayload` does check for the id already set under `remotePaymentsPayloadHash`. If not set, it allows storing the hash. Otherwise, the store is skipped and `false` returned. It is limited to `onlyOrderer`.
- `commitRemoteBeneficiaryPayload` will store under a given `sfxId` on the `remotePaymentsPayloadHash` mapping a new hash of the previous hash and the beneficiary abi encoded. Hash chaining the set hash via `storeRemoteOrderPayload` and the beneficiary address. Limited to `onlyAttesters`.
- The `revertRemoteOrderPayload` function does the same as the `commitRemoteBeneficiaryPayload` function but using `address(0)` instead. Limited to `onlyAttesters`.
- The `storeEscrowCallOrder` function does check for functions with a signature of `selfdestruct(address)` but this doesn't prevent from having any function with different signature and a selfdestruct opcode. The call is then stored under `escrowCalls[id] = call;`.
- `claimRefund` can be executed after 128 blocks have passed from the creation of the order. If `ETH` tokens are used, then the reward amount is refunded.

5.3 LocalExchange

- ensureBalanceAndAllowance should not check for user.balance >= amount as msg.value is already deducted.
- localOrder does allow creating multiple orders for the same block. If all parameters, token, amount, rewardToken, reward are the same the transaction will revert. You can select the corresponding local_order_id as long as you use the same parameters. There is a low probability risk of hash collision using a malicious rewardToken that would lead to the same hash id.
- executeLocalOrder will fetch based on the parameters the order id and transfer from the caller the amount and reward the caller with the reward amount using different tokens for the order and reward.

5.4 RemoteOrder

- The owner is the deployer, and also sets the escrowGMP address.
- assignAttesters is restricted to the owner. It enforces the onlyAttesters modifier.
- The orderMemoryData function will generate a unique and none re-usable id per user/block. storeRemoteOrderPayload will prevent re-usage. There is a lot of data that is not taken into consideration and hashed.
- The remoteBridgeAsset function, does call the orderMemoryData without giving the mgs.value. However, it is not required as the latter is a public function called on the same flow.
- The bidFifo allows the first caller for a new/un-used sfxId to be set as the orderWinners.
- claimRefund will calculate the id based on parameters and trigger a withdrawFromVault. The withdraw happens to the sender, not the actual creator of the order.
- The withdrawFromVault function, looks like the hashing chaining could be manipulated as the second hash value does have type confusion, it can be either an address or an amount. However, the withdrawFromVault function does perform a hash of the base

paymentPayloadHash (that uses amount). This means, it is not possible to forge neither an address that matches the amount nor an amount that matches and address.

- Both claimPayoutBatch and claimRefundBatch will make sure that if the same rewardAssets is used, they will merge rewardAmounts.

5.5 AttestationsVerifierProofs

- During constructor, the initialCommittee and nextCommittee hash are set using the parameters array via implyCommitteeRoot. The quorum is stablished to be 2/3 of the initialCommittee length. **However, if the initialCommittee size is 1, quorum will be zero.**
- recoverCurrentSigners will return all signers that verify a given signature hash. It will also be checking for banned addresses on an array.
- In the receiveSingleAttestation the hash is obtained from a user controlled parameters. However, the signatures are verified against that hash, which means that an attacker cannot forge custom attestersAsLeaves that validate via multiProofVerifyCalldata the currentCommitteeHash.
- For the receiveAttestationBatch function, even if an attacker hand-crafted all the batch data, and the impliedNextCommitteeHash == nextCommitteeHash check bypassed by providing a valid maybeNextCommittee that would create the same hash as the constructor. It wouldn't be possible to give a valid attestersAsLeaves to multiProofVerifyCalldata.
- implyCommitteeRoot does create a root hash from a list of addresses, being the leaves of the tree. No sibling hash is passed using multiProofProof.

AUTOMATED TESTING

6.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
The security team assessed all findings identified by the Slither software, however, findings with severity Information and Optimization are not included in the below results for the sake of report readability.

Results:

Slither results for t3rn - guardian	
Finding	**Impact**
RemoteOrder.withdrawFromVault(bytes32,address,uint256) (contracts/remoteOrder.sol#179-191) sends eth to arbitrary user Dangerous calls: - address(msg.sender).transfer(amount) (contracts/remoteOrder.sol#187)	High
LocalExchange.executeLocalOrder(uint256,address,address,uint256,address,uint256) (contracts/localExchange.sol#68-105) uses arbitrary from in transferFrom: IERC20(rewardToken).safeTransferFrom(user,msg.sender,reward) (contracts/localExchange.sol#97)	High

AUTOMATED TESTING

Finding	Impact
Reentrancy in LocalExchange.executeLocalOrder(uint256,address,address,uint256,address,uint256) (contracts/localExchange.sol#68-105): External calls: - IERC20(token).safeTransferFrom(msg.sender,user,amount) (contracts/localExchange.sol#92) - IERC20(rewardToken).safeTransferFrom(user,msg.sender,reward) (contracts/localExchange.sol#97) External calls sending eth: - address(user).transfer(amount) (contracts/localExchange.sol#89) - address(msg.sender).transfer(reward) (contracts/localExchange.sol#99) State variables written after the call(s): - localOrders[local_order_id] = false (contracts/localExchange.sol#103)LocalExchange.localOrders (contracts/localExchange.sol#21) can be used in cross function reentrancies: - LocalExchange.claimRefund(uint256,address,uint256,address,uint256) (contracts/localExchange.sol#107-126) - LocalExchange.executeLocalOrder(uint256,address,address,uint256,address,uint256) (contracts/localExchange.sol#68-105) - LocalExchange.localOrder(address,uint256,address,uint256) (contracts/localExchange.sol#56-66) - LocalExchange.localOrders (contracts/localExchange.sol#21)	High
RemoteOrder.withdrawFromVault(bytes32,address,uint256) (contracts/remoteOrder.sol#179-191) sends eth to arbitrary user Dangerous calls: - address(msg.sender).transfer(amount) (contracts/remoteOrder.sol#187)	High
RemoteOrder.withdrawFromVault(bytes32,address,uint256) (contracts/remoteOrder.sol#179-191) uses a dangerous strict equality: - require(bool,string)(paymentHash == calculatedWithdrawHash || paymentHash == calculatedRefundHash,Payload for payment not matching) (contracts/remoteOrder.sol#184)	Medium

Finding	Impact
Reentrancy in AttestationsVerifierProofs.receiveSingleAttestation(bytes,bytes4,uint32,bytes[],bytes32[],bool[]) (contracts/attestationsVerifierProofs.sol#148-178):External calls: - decodeAndProcessPayload(messageGMPPayload) (contracts/attestationsVerifierProofs.sol#174) - escrowGMP.commitRemoteBeneficiaryPayload(sfxId,destination) (contracts/attestationsVerifierProofs.sol#301) - escrowGMP.revertRemoteOrderPayload(sfxId_scope_0) (contracts/attestationsVerifierProofs.sol#309) - escrowGMP.commitEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#320) - escrowGMP.revertEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#323) State variables written after the call(s): - committedGMPMessagesMap[messageHash] = true (contracts/attestationsVerifierProofs.sol#176)AttestationsVerifierProofs.committedGMPMessagesMap (contracts/attestationsVerifierProofs.sol#66) can be used in cross function reentrancies: - AttestationsVerifierProofs.committedGMPMessagesMap (contracts/attestationsVerifierProofs.sol#66) - AttestationsVerifierProofs.isAttestationApplied(bytes32) (contracts/attestationsVerifierProofs.sol#233-235) - AttestationsVerifierProofs.receiveAttestationBatch(bytes,bytes,bytes[],bytes32[],bool[]) (contracts/attestationsVerifierProofs.sol#181-230) - AttestationsVerifierProofs.receiveSingleAttestation(bytes,bytes4,uint32,bytes[],bytes32[],bool[]) (contracts/attestationsVerifierProofs.sol#148-178)	Medium

Finding	Impact
Reentrancy in AttestationsVerifierProofs.receiveAttestationBatch(bytes,bytes,bytes[],bytes32[],bool[]) (contracts/attestationsVerifierProofs.sol#181-230):External calls: - decodeAndProcessPayload(messageGMPPayload) (contracts/attestationsVerifierProofs.sol#222) - escrowGMP.commitRemoteBeneficiaryPayload(sfxId,destination) (contracts/attestationsVerifierProofs.sol#301) - escrowGMP.revertRemoteOrderPayload(sfxId_scope_0) (contracts/attestationsVerifierProofs.sol#309) - escrowGMP.commitEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#320) - escrowGMP.revertEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#323) State variables written after the call(s): - committedGMPMessagesMap[batchMessageHash] = true (contracts/attestationsVerifierProofs.sol#227)AttestationsVerifierProofs.committedGMPMessagesMap (contracts/attestationsVerifierProofs.sol#66) can be used in cross function reentrancies: - AttestationsVerifierProofs.committedGMPMessagesMap (contracts/attestationsVerifierProofs.sol#66) - AttestationsVerifierProofs.isAttestationApplied(bytes32) (contracts/attestationsVerifierProofs.sol#233-235) - AttestationsVerifierProofs.receiveAttestationBatch(bytes,bytes,bytes[],bytes32[],bool[]) (contracts/attestationsVerifierProofs.sol#181-230) - AttestationsVerifierProofs.receiveSingleAttestation(bytes,bytes4,uint32,bytes[],bytes32[],bool[]) (contracts/attestationsVerifierProofs.sol#148-178) - currentBatchIndex = batch.index (contracts/attestationsVerifierProofs.sol#225)AttestationsVerifierProofs.currentBatchIndex (contracts/attestationsVerifierProofs.sol#70) can be used in cross function reentrancies: - AttestationsVerifierProofs.constructor(address[],address[],uint256,EscrowGMP) (contracts/attestationsVerifierProofs.sol#92-108) - AttestationsVerifierProofs.currentBatchIndex (contracts/attestationsVerifierProofs.sol#70) - AttestationsVerifierProofs.overrideCurrentBatchIndex(uint256) (contracts/attestationsVerifierProofs.sol#118-120) - AttestationsVerifierProofs.receiveAttestationBatch(bytes,bytes,bytes[],bytes32[],bool[]) (contracts/attestationsVerifierProofs.sol#181-230) - AttestationsVerifierProofs.setBatchIndex(uint256) (contracts/attestationsVerifierProofs.sol#59-61)	Medium

AUTOMATED TESTING

Finding	Impact
AttestationsVerifierProofs.decodeAndProcessPayload(bytes) (contracts/attestationsVerifierProofs.sol#286-331) ignores return value by escrowGMP.revertRemoteOrderPayload(sfxId_scope_0) (contracts/attestationsVerifierProofs.sol#309)	Medium
AttestationsVerifierProofs.decodeAndProcessPayload(bytes) (contracts/attestationsVerifierProofs.sol#286-331) ignores return value by escrowGMP.revertEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#323)	Medium
AttestationsVerifierProofs.decodeAndProcessPayload(bytes) (contracts/attestationsVerifierProofs.sol#286-331) ignores return value by escrowGMP.commitEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#320)	Medium
AttestationsVerifierProofs.decodeAndProcessPayload(bytes) (contracts/attestationsVerifierProofs.sol#286-331) ignores return value by escrowGMP.commitRemoteBeneficiaryPayload(sfxId,destination) (contracts/attestationsVerifierProofs.sol#301)	Medium
LocalExchange.claimRefund(uint256,address,uint256,address,uint256) (contracts/localExchange.sol#107-126) ignores return value by IERC20(rewardToken).approve(address(this),0) (contracts/localExchange.sol#124)	Medium
RemoteOrder.withdrawFromVault(bytes32,address,uint256) (contracts/remoteOrder.sol#179-191) uses a dangerous strict equality: - require(bool,string)(paymentHash == calculatedWithdrawHash || paymentHash == calculatedRefundHash,Payload for payment not matching) (contracts/remoteOrder.sol#184)	Medium
AttestationsVerifierProofs.receiveSingleAttestation(bytes,bytes4,uint32,bytes[],bytes32[],bool[]).messageHash (contracts/attestationsVerifierProofs.sol#156) shadows: - AttestationsVerifierProofs.messageHash(AttestationsVerifierProofs.Batch) (contracts/attestationsVerifierProofs.sol#84-86) (function)	Low

Finding	Impact
AttestationsVerifierProofs.recoverCurrentSigners(bytes32,bytes[],address[]).leaves_scope_0 (contracts/attestationsVerifierProofs.sol#239) shadows: - AttestationsVerifierProofs.recoverCurrentSigners(bytes32,bytes[],address[]).leaves (contracts/attestationsVerifierProofs.sol#237) (return variable)	Low
AttestationsVerifierProofs.recoverSigner(bytes32,bytes).messageHash (contracts/attestationsVerifierProofs.sol#342) shadows: - AttestationsVerifierProofs.messageHash(AttestationsVerifierProofs.Batch) (contracts/attestationsVerifierProofs.sol#84-86) (function)	Low
EscrowGMP.assignOrderer(address) (contracts/escrowGMP.sol#43-45) should emit an event for: - orderer = _orderer (contracts/escrowGMP.sol#44)	Low
EscrowGMP.assignAttesters(address) (contracts/escrowGMP.sol#39-41) should emit an event for: - attesters = _attesters (contracts/escrowGMP.sol#40)	Low
RemoteOrder.assignAttesters(address) (contracts/remoteOrder.sol#80-82) should emit an event for: - attesters = _attesters (contracts/remoteOrder.sol#81)	Low
AttestationsVerifierProofs.setBatchIndex(uint256) (contracts/attestationsVerifierProofs.sol#59-61) should emit an event for: - currentBatchIndex = _batchIndex (contracts/attestationsVerifierProofs.sol#60)	Low
AttestationsVerifierProofs.overrideCurrentBatchIndex(uint256) (contracts/attestationsVerifierProofs.sol#118-120) should emit an event for: - currentBatchIndex = newBatchIndex (contracts/attestationsVerifierProofs.sol#119)	Low
AttestationsVerifierProofs.updateCommitteeSize(uint256) (contracts/attestationsVerifierProofs.sol#141-145) should emit an event for: - quorum = (committeeSize * 2) / 3 (contracts/attestationsVerifierProofs.sol#144)	Low
EscrowGMP.assignAttesters(address)._attesters (contracts/escrowGMP.sol#39) lacks a zero-check on : - attesters = _attesters (contracts/escrowGMP.sol#40)	Low

Finding	Impact
RemoteOrder.withdrawFromVaultSkipGMPChecks(address,uint256,address).beneficiary (contracts/remoteOrder.sol#193) lacks a zero-check on : - address(beneficiary).transfer(amount) (contracts/remoteOrder.sol#195)	Low
RemoteOrder.assignAttesters(address)._attesters (contracts/remoteOrder.sol#80) lacks a zero-check on : - attesters = _attesters (contracts/remoteOrder.sol#81)	Low
EscrowGMP.assignOrderer(address)._orderer (contracts/escrowGMP.sol#43) lacks a zero-check on : - orderer = _orderer (contracts/escrowGMP.sol#44)	Low
RemoteOrder.withdrawFromVault(bytes32,address,uint256) (contracts/remoteOrder.sol#179-191) has external calls inside a loop: address(msg.sender).transfer(amount) (contracts/remoteOrder.sol#187)	Low
RemoteOrder.withdrawFromVault(bytes32,address,uint256) (contracts/remoteOrder.sol#179-191) has external calls inside a loop: escrowGMP.nullifyPayloadHash(sfxId) (contracts/remoteOrder.sol#185)	Low
RemoteOrder.withdrawFromVault(bytes32,address,uint256) (contracts/remoteOrder.sol#179-191) has external calls inside a loop: paymentHash = escrowGMP.getRemotePaymentPayloadHash(sfxId) (contracts/remoteOrder.sol#183)	Low
Address.functionCallWithValue(address,bytes,uint256,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#128-137) has external calls inside a loop: (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#135)	Low
Reentrancy in RemoteOrder.orderMemoryData(bytes) (contracts/remoteOrder.sol#97-114): External calls: - IERC20(rewardAsset).safeTransferFrom(msg.sender,address(this),maxReward) (contracts/remoteOrder.sol#109) - require(bool,string)(escrowGMP.storeRemoteOrderPayload(id,keccak256(bytes)(abi.encode(rewardAsset,maxReward))),Payload already stored) (contracts/remoteOrder.sol#111) Event emitted after the call(s): - RemoteOrderCreated(id,nonce,msg.sender,input) (contracts/remoteOrder.sol#113)	Low

Finding	Impact
Reentrancy in AttestationsVerifierProofs.receiveSingleAttestation(bytes,bytes4,uint32,bytes[],bytes32[],bool[]) (contracts/attestationsVerifierProofs.sol#148-178):External calls: - decodeAndProcessPayload(messageGMPPayload) (contracts/attestationsVerifierProofs.sol#174) - escrowGMP.commitRemoteBeneficiaryPayload(sfxId,destination) (contracts/attestationsVerifierProofs.sol#301) - escrowGMP.revertRemoteOrderPayload(sfxId_scope_0) (contracts/attestationsVerifierProofs.sol#309) - escrowGMP.commitEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#320) - escrowGMP.revertEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#323) Event emitted after the call(s): - CommitmentApplied(messageHash,msg.sender) (contracts/attestationsVerifierProofs.sol#177)	Low
Reentrancy in RemoteOrder.order(bytes4,uint32,bytes32,uint256,address,uint256,uint256) (contracts/remoteOrder.sol#134-140): External calls: - orderMemoryData(input) (contracts/remoteOrder.sol#137) - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#122)- (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#135)- IERC20(rewardAsset).safeTransferFrom(msg.sender,address(this),maxReward) (contracts/remoteOrder.sol#109) - require(bool,string)(escrowGMP.storeRemoteOrderPayload(id,keccak256(bytes)(abi.encode(rewardAsset,maxReward))),Payload already stored) (contracts/remoteOrder.sol#111) External calls sending eth: - orderMemoryData(input) (contracts/remoteOrder.sol#137) - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#135)Event emitted after the call(s): - OrderCreated(generateId(msg.sender,uint32(block.number)),destination,asset,targetAccount,amount,rewardAsset,insurance,maxReward,uint32(block.number)) (contracts/remoteOrder.sol#139)	Low

Finding	Impact
Reentrancy in AttestationsVerifierProofs.decodeAndProcessPayload(bytes) (contracts/attestationsVerifierProofs.sol#286-331):External calls: - escrowGMP.commitRemoteBeneficiaryPayload(sfxId,destination) (contracts/attestationsVerifierProofs.sol#301) - escrowGMP.revertRemoteOrderPayload(sfxId_scope_0) (contracts/attestationsVerifierProofs.sol#309) - escrowGMP.commitEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#320) - escrowGMP.revertEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#323) Event emitted after the call(s): - CallCommitApplied(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#321) - CallRevertApplied(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#324) - TransferCommitApplied(sfxId,destination) (contracts/attestationsVerifierProofs.sol#303) - TransferRevertApplied(sfxId_scope_0) (contracts/attestationsVerifierProofs.sol#311)	Low
Reentrancy in AttestationsVerifierProofs.receiveAttestationBatch(bytes,bytes,bytes[],bytes32[],bool[]) (contracts/attestationsVerifierProofs.sol#181-230):External calls: - decodeAndProcessPayload(messageGMPPayload) (contracts/attestationsVerifierProofs.sol#222) - escrowGMP.commitRemoteBeneficiaryPayload(sfxId,destination) (contracts/attestationsVerifierProofs.sol#301) - escrowGMP.revertRemoteOrderPayload(sfxId_scope_0) (contracts/attestationsVerifierProofs.sol#309) - escrowGMP.commitEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#320) - escrowGMP.revertEscrowCall(sfxId_scope_1) (contracts/attestationsVerifierProofs.sol#323) Event emitted after the call(s): - BatchApplied(batchMessageHash,msg.sender) (contracts/attestationsVerifierProofs.sol#229)	Low

Finding	Impact
TRN.constructor(string,string).name (contracts/TRNToken.sol#11) shadows: - ERC20.name() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#62-64) (function) - IERC20Metadata.name() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#17) (function)	Low
TRN.constructor(string,string).symbol (contracts/TRNToken.sol#11) shadows: - ERC20.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#70-72) (function) - IERC20Metadata.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#22) (function)	Low
LocalExchange.claimRefund(uint256,address,uint256,address,uint256).user (contracts/localExchange.sol#108) lacks a zero-check on : - address(user).transfer(reward) (contracts/localExchange.sol#121)	Low
Reentrancy in LocalExchange.executeLocalOrder(uint256,address,address,uint256,address,uint256) (contracts/localExchange.sol#68-105): External calls: - IERC20(token).safeTransferFrom(msg.sender,user,amount) (contracts/localExchange.sol#92) - IERC20(rewardToken).safeTransferFrom(user,msg.sender,reward) (contracts/localExchange.sol#97) External calls sending eth: - address(user).transfer(amount) (contracts/localExchange.sol#89) - address(msg.sender).transfer(reward) (contracts/localExchange.sol#99) Event emitted after the call(s): - OrderExecuted(user,msg.sender,token,amount) (contracts/localExchange.sol#104)	Low
t3USD.constructor(string,string).name (contracts/t3USD.sol#11) shadows: - ERC20.name() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#62-64) (function) - IERC20Metadata.name() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#17) (function)	Low

Finding	Impact
t3USD.constructor(string,string).symbol (contracts/t3USD.sol#11) shadows: - ERC20.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#70-72) (function) - IERC20Metadata.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#22) (function)	Low
t3DOT.constructor(string,string).symbol (contracts/t3DotToken.sol#11) shadows: - ERC20.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#70-72) (function) - IERC20Metadata.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#22) (function)	Low
t3DOT.constructor(string,string).name (contracts/t3DotToken.sol#11) shadows: - ERC20.name() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#62-64) (function) - IERC20Metadata.name() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#17) (function)	Low
ERC20Mock.constructor(string,string).symbol (contracts/ERC20Mock.sol#9) shadows: - ERC20.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#70-72) (function) - IERC20Metadata.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#22) (function)	Low
ERC20Mock.constructor(string,string).name (contracts/ERC20Mock.sol#9) shadows: - ERC20.name() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#62-64) (function) - IERC20Metadata.name() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#17) (function)	Low
EscrowGMP.assignOrderer(address) (contracts/escrowGMP.sol#43-45) should emit an event for: - orderer = _orderer (contracts/escrowGMP.sol#44)	Low
EscrowGMP.assignAttesters(address) (contracts/escrowGMP.sol#39-41) should emit an event for: - attesters = _attesters (contracts/escrowGMP.sol#40)	Low

Finding	Impact
EscrowGMP.assignAttesters(address)._attesters (contracts/escrowGMP.sol#39) lacks a zero-check on : - attesters = _attesters (contracts/escrowGMP.sol#40)	Low
EscrowGMP.assignOrderer(address)._orderer (contracts/escrowGMP.sol#43) lacks a zero-check on : - orderer = _orderer (contracts/escrowGMP.sol#44)	Low
t3SOL.constructor(string,string).symbol (contracts/t3SOLToken.sol#11) shadows: - ERC20.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#70-72) (function) - IERC20Metadata.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#22) (function)	Low
t3SOL.constructor(string,string).name (contracts/t3SOLToken.sol#11) shadows: - ERC20.name() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#62-64) (function) - IERC20Metadata.name() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#17) (function)	Low
EscrowGMP.assignOrderer(address) (contracts/escrowGMP.sol#43-45) should emit an event for: - orderer = _orderer (contracts/escrowGMP.sol#44)	Low
EscrowGMP.assignAttesters(address) (contracts/escrowGMP.sol#39-41) should emit an event for: - attesters = _attesters (contracts/escrowGMP.sol#40)	Low
RemoteOrder.assignAttesters(address) (contracts/remoteOrder.sol#80-82) should emit an event for: - attesters = _attesters (contracts/remoteOrder.sol#81)	Low
EscrowGMP.assignAttesters(address)._attesters (contracts/escrowGMP.sol#39) lacks a zero-check on : - attesters = _attesters (contracts/escrowGMP.sol#40)	Low
RemoteOrder.withdrawFromVaultSkipGMPChecks(address,uint256,address).beneficiary (contracts/remoteOrder.sol#193) lacks a zero-check on : - address(beneficiary).transfer(amount) (contracts/remoteOrder.sol#195)	Low
RemoteOrder.assignAttesters(address)._attesters (contracts/remoteOrder.sol#80) lacks a zero-check on : - attesters = _attesters (contracts/remoteOrder.sol#81)	Low

Finding	Impact
EscrowGMP.assignOrderer(address)._orderer (contracts/escrowGMP.sol#43) lacks a zero-check on : - orderer = _orderer (contracts/escrowGMP.sol#44)	Low
RemoteOrder.withdrawFromVault(bytes32,address,uint256) (contracts/remoteOrder.sol#179-191) has external calls inside a loop: address(msg.sender).transfer(amount) (contracts/remoteOrder.sol#187)	Low
RemoteOrder.withdrawFromVault(bytes32,address,uint256) (contracts/remoteOrder.sol#179-191) has external calls inside a loop: escrowGMP.nullifyPayloadHash(sfxId) (contracts/remoteOrder.sol#185)	Low
RemoteOrder.withdrawFromVault(bytes32,address,uint256) (contracts/remoteOrder.sol#179-191) has external calls inside a loop: paymentHash = escrowGMP.getRemotePaymentPayloadHash(sfxId) (contracts/remoteOrder.sol#183)	Low
Address.functionCallWithValue(address,bytes,uint256,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#128-137) has external calls inside a loop: (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#135)	Low
Reentrancy in RemoteOrder.orderMemoryData(bytes) (contracts/remoteOrder.sol#97-114): External calls: - IERC20(rewardAsset).safeTransferFrom(msg.sender,address(this),maxReward) (contracts/remoteOrder.sol#109) - require(bool,string)(escrowGMP.storeRemoteOrderPayload(id,keccak256(bytes)(abi.encode(rewardAsset,maxReward))),Payload already stored) (contracts/remoteOrder.sol#111) Event emitted after the call(s): - RemoteOrderCreated(id,nonce,msg.sender,input) (contracts/remoteOrder.sol#113)	Low

Finding	Impact
Reentrancy in RemoteOrder.order(bytes4,uint32,bytes32,uint256,address,uint256,uint256) (contracts/remoteOrder.sol#134-140): External calls: - orderMemoryData(input) (contracts/remoteOrder.sol#137) - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#122)- (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#135)- IERC20(rewardAsset).safeTransferFrom(msg.sender,address(this),maxReward) (contracts/remoteOrder.sol#109) - require(bool,string)(escrowGMP.storeRemoteOrderPayload(id,keccak256(bytes)(abi.encode(rewardAsset,maxReward))),Payload already stored) (contracts/remoteOrder.sol#111) External calls sending eth: - orderMemoryData(input) (contracts/remoteOrder.sol#137) - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#135)Event emitted after the call(s): - OrderCreated(generateId(msg.sender,uint32(block.number)),destination,asset,targetAccount,amount,rewardAsset,insurance,maxReward,uint32(block.number)) (contracts/remoteOrder.sol#139)	Low
t3BTC.constructor(string,string).name (contracts/t3BTCToken.sol#11) shadows: - ERC20.name() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#62-64) (function) - IERC20Metadata.name() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#17) (function)	Low
t3BTC.constructor(string,string).symbol (contracts/t3BTCToken.sol#11) shadows: - ERC20.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#70-72) (function) - IERC20Metadata.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#22) (function)	Low
End of table for t3rn - guardian	

Results summary:

The findings obtained as a result of the Slither scan were reviewed. The majority of Slither findings were determined false-positives.

THANK YOU FOR CHOOSING

HALBORN