// Security Assessment

12.09.2025 - 12.26.2025

Vault, Cosmos & EVM Contracts

Provenance

Vault, Cosmos & EVM Contracts - Provenance

Prepared by: HALBORN

Last Updated 01/01/2026

Date of Engagement: December 9th, 2025 - December 26th, 2025

Summary

0% ⓘ OF ALL REPORTED

FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
15	0	0	0	9	6

1. Introduction

`Provenance Blockchain` engaged Halborn to conduct a comprehensive security assessment of the `Provenance Blockchain` beginning on December 9th, 2025, and concluding on December 26th, 2025. The scope of this assessment was limited to the smart contracts and Cosmos-SDK Vault module provided to the Halborn team. Commit hashes, scope boundaries, and additional technical details are documented in the Scope section of this report.

`Provenance Blockchain` is a public, permissionless, proof-of-stake blockchain, purpose-built to modernize financial infrastructure. It is more than just a general-purpose ledger; it's an integrated ecosystem designed for finance.

TABLE OF CONTENTS

2. Assessment Summary

`Halborn` assigned a full-time security engineer to perform a comprehensive review of the contracts. The engineer is a blockchain and smart contract and blockchain security expert with extensive experience in penetration testing, vulnerability research, and auditing across multiple blockchain ecosystems.

The purpose of this assessment was to:

- Identify potential security issues and vulnerabilities within the EVM and Cosmos Vault smart contracts along with the Integration of a specific Cosmos-SDK vault module.
- Ensure that all contract components function as intended under expected and edge-case conditions.

In summary, `Halborn` identified several areas for improvement to minimize both the likelihood and potential impact of security risks, which should be addressed by the `Provenance Blockchain team`. The primary recommendations include:

- `Rejecting SwapOut requests where the computed redeemable assets are zero (return a clear "amount too small" error), or treat zero-asset conversions as a recoverable failure in payout processing and refund shares instead of burning.`
- `Introducing a separate per-block scan budget that counts skipped paused entries, or move paused-vault jobs out of the due-walk path.`
- `Removing the hardcoded uylds.fcc 1:1 path and always use Marker NAV for conversions.`

baseapp validatebasic

7.15 Missing zero-amount check in burnswapinreceipt

3. Test Approach And Methodology

Halborn performed a combination of manual code review and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is essential to uncover flaws in logic, process, and implementation, automated testing techniques enhance coverage of smart contracts and can quickly identify issues that do not follow security best practices.

The following phases and associated tools were used throughout the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual code review and walkthrough of the smart contracts to identify potential logic issues.
- Manual testing of all core functions, including deposit, withdraw, vault creation, etc., to validate expected behavior and identify edge-case vulnerabilities.
- Local testing to simulate contract interactions and validate functional and security assumptions.
- Fuzz testing with the golang's integrated Fuzzer.

4. RISK METHODOLOGY

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

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

successful exploit.

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

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

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

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

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

ATTACK COMPLEXITY (AX):

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

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE

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

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

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

INTEGRITY (I):

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

AVAILABILITY (A):

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

DEPOSIT (D):

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

YIELD (Y):

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

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

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

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can

be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

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

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

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

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9

Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORIES

(a) Repository: nuva-evm-contracts

(b) Assessed Commit ID: 6f48f08

(c) Items in scope:

- contracts/CustomToken.sol
- contracts/Depositor.sol
- contracts/DepositorFactory.sol
- contracts/TokenFactory.sol
- contracts/Withdrawal.sol
- contracts/WithdrawalFactory.sol

Out-of-Scope: Third party dependencies and economic attacks.

(a) Repository: vault

(b) Assessed Commit ID: a7d0f0e

(c) Items in scope:

- vault-main/keeper/msg_server.go
- vault-main/module.go
- vault-main/types/msgs.go
- vault-main/keeper/vault.go
- vault-main/keeper/reconcile.go
- vault-main/keeper/query_server.go
- vault-main/types/events.go
- vault-main/types/vault.go
- vault-main/queue/pending_swapout.go
- vault-main/keeper/payout.go
- vault-main/keeper/valuation_engine.go
- vault-main/interest/interest.go
- vault-main/keeper/genesis.go
- vault-main/keeper/keeper.go

- vault-main/queue/payout_timeout.go
- vault-main/utils/shares.go
- vault-main/utils/query/query.go
- vault-main/keeper/state.go
- vault-main/types/expected_keepers.go
- vault-main/utils/accounts.go
- vault-main/keeper/queue.go
- vault-main/types/codec.go
- vault-main/types/keys.go
- vault-main/utils/slices.go
- vault-main/keeper/abci.go
- vault-main/types/payout.go
- vault-main/utils/math.go
- vault-main/types/errors.go
- vault-main/types/genesis.go
- vault-main/utils/tools.go

Out-of-Scope: Third party dependencies and economic attacks.

(a) Repository: nuva-cosmos-contracts

(b) Assessed Commit ID: e999bef

(c) Items in scope:

- src/lib.rs
- src/contract.rs
- src/error.rs
- src/helpers.rs
- src/msg.rs
- src/state.rs
- src/tests.rs

Out-of-Scope: Third party dependencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM
0	0	0

LOW	INFORMATIONAL
9	6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - DUST SWAPOUT CAN BURN SHARES WHILE PAYING OUT ZERO ASSETS DUE TO FLOOR ROUNDING	LOW	PENDING
HAL-02 - PAUSED VAULT WITHDRAWAL BACKLOG CAUSES REPEATED ENDBLOCKER O(N) SCANS	LOW	PENDING
HAL-03 - RECEIPTID VALIDATION USES UNTRIMMED VALUE IN STORAGE	LOW	PENDING
HAL-04 - HARDCODED 1:1 PRICING FOR UYLDS.FCC BYPASSES NAV AND CAN CAUSE USER ARBITRAGE IF THE PEG DEVIATES	LOW	PENDING
HAL-05 - NAV PUBLISH CAN		

PANIC WHEN TOTALSHARES EXCEEDS UINT64	LOW	PENDING
HAL-06 - MIGRATION DOES NOT TRANSFER ACCUMULATED TOKENS	LOW	PENDING
HAL-07 - UNBOUNDED LOOP IN DESTINATION ADDRESS MANAGEMENT	LOW	PENDING
HAL-08 - MIGRATION DOES NOT PRESERVE DESTINATION ADDRESSES	LOW	PENDING
HAL-09 - BURNSWAPOUTRECEIPT AMOUNT PARAMETER NOT VALIDATED AGAINST RECEIPT	LOW	PENDING
HAL-10 - NO VALIDATION OF TO_ADDRESS PARAMETER	INFORMATIONAL	PENDING
HAL-11 - AML SIGNER ADDRESS CANNOT BE ROTATED	INFORMATIONAL	PENDING
HAL-12 - EMPTY MIGRATION HANDLER WITHOUT VERSION CHECKING	INFORMATIONAL	PENDING
HAL-13 - MISSING INPUT VALIDATION IN TOKEN CREATION	INFORMATIONAL	PENDING
HAL-14 - SWAPOUT ZERO-AMOUNT VALIDATION RELIES ON BASEAPP VALIDATEBASIC	INFORMATIONAL	PENDING

HAL-15 - MISSING ZERO-AMOUNT CHECK IN BURNSWAPINRECEIPT	INFORMATIONAL	PENDING

7. FINDINGS & TECH DETAILS

7.1 (HAL-01) DUST SWAPOUT CAN BURN SHARES WHILE PAYING OUT ZERO ASSETS DUE TO FLOOR ROUNDING

// LOW

Description

In the `Keeper.SwapOut` in `vault-main/keeper/vault.go` the request is enqueued even when the computed redeemable assets round down to zero. Later, `processSingleWithdrawal` in `vault-main/keeper/payout.go` will attempt payout using `sdk.NewCoins(assets)`. If `assets.Amount` is zero, `sdk.NewCoins` returns an empty coin set and `BankKeeper.SendCoins` treats it as valid and performs a no-op transfer, then the code proceeds to burn the user's escrowed shares. This creates a provable user-fund-loss footgun where a user can lose shares and receive zero assets.

Code Location

The `Keeper.SwapOut` showing it queues the withdrawal without checking `assets.Amount > 0` after conversion.

```
244 func (k *Keeper) SwapOut(ctx sdk.Context, vaultAddr, owne
245     // ... omitted checks ...
246     assets, err := k.ConvertSharesToRedeemCoin(ctx, *vaul
247     if err != nil {
248         return 0, fmt.Errorf("failed to calculate assets
249     }
250     // no guard to reject assets.Amount == 0
251     if err := k.checkPayoutRestrictions(ctx, vault, owner
252         return 0, err
253     }
254     if err := k.BankKeeper.SendCoins(ctx, owner, vault.Ge
255         return 0, fmt.Errorf("failed to escrow shares: %w
256     }
257     requestID, err := k.PendingSwapOutQueue.Enqueue(ctx,
258     // ...
259     return requestID, nil
260 }
```

The `processSingleWithdrawal` can payout zero coins (empty coin set) then burns shares.

```
func (k *Keeper) processSingleWithdrawal(ctx sdk.Context,
    // ...
    assets, err := k.ConvertSharesToRedeemCoin(ctx, vault
    if err != nil {
        return fmt.Errorf("failed to convert shares to re
    }
    if err := k.BankKeeper.SendCoins(markertypes.WithTran
        return err
    }
    // burns shares even if assets was a zero-amount coin
    if err := k.MarkerKeeper.BurnCoin(ctx, vaultAddr, req
        // ...
    }
    // ...
    return nil
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (3.4)

Recommendation

It is recommended to reject `SwapOut` requests where the computed redeemable assets are zero (return a clear “amount too small” error), or treat zero-asset conversions as a recoverable failure in payout processing and refund shares instead of burning.

7.2 (HAL-02) PAUSED VAULT WITHDRAWAL BACKLOG CAUSES REPEATED ENDBLOCKER O(N) SCANS

// LOW

Description

In the `processPendingSwapOuts` in `vault-main/keeper/payout.go`, due withdrawals for paused vaults are

skipped without being dequeued and without being counted against the per-block batch limit. If a vault becomes paused while

it has a large backlog of due withdrawals, every EndBlocker will walk and skip those same due entries again, causing repeated work proportional to the backlog size and degrading block processing time.

Code Location

The snippet from `processPendingSwapOuts` shows that paused vault entries are skipped and do not increment the processed batch counter.

```
22 func (k *Keeper) processPendingSwapOuts(ctx context.Conte
23     // ...
24     processed := 0
25     err := k.PendingSwapOutQueue.WalkDue(ctx, now, func(t
26         vault, ok := k.tryGetVault(sdkCtx, vaultAddr)
27         if ok && vault.Paused {
28             return false, nil
29         }
30         if processed == batchSize {
31             return true, nil
32         }
33         processed++
34         jobsToProcess = append(jobsToProcess, types.NewPa
35         return false, nil
36     })
37     // ...
38     return nil
39 }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (3.4)

Recommendation

It is recommended to Introduce a separate per-block scan budget that counts skipped paused entries, or move paused-vault jobs out of the due-walk path (e.g., bucket by vault and only revisit on unpause) so `EndBlocker` does not repeatedly iterate the same due set.

7.3 (HAL-03) RECEIPTID

VALIDATION USES UNTRIMMED VALUE IN STORAGE

// LOW

Description

The `ReceiptId` type conversion in the `try_from` function within `src/helpers.rs` validates the trimmed version of the input string but then stores the original untrimmed string. This means that if a user provides an ID with leading or trailing whitespace such as " myreceipt123", the validation checks are performed against "myreceipt123" but the storage key becomes " myreceipt123" with the spaces included. This creates a mismatch between what the contract validates and what it actually stores. The impact is significant because off-chain bridge relayer systems typically canonicalize and trim IDs before performing lookups. If a receipt was created with whitespace in the ID, the relayer will fail to find it when querying with the trimmed version, potentially causing bridged funds to become stuck. Additionally, this allows multiple receipts to exist that appear identical in user interfaces but are actually different storage keys, leading to confusion and potential accounting errors.

Code Location

```
52  impl TryFrom<String> for ReceiptId {
53      type Error = StdError;
54
55      fn try_from(desired: String) -> Result<Self, Self::Er
56          let value = desired.trim();
57          if value.is_empty() {
58              return Err(StdError::generic_err("receipt id
59          }
60
61          if value.len() < 8 || value.len() > 100 {
62              return Err(StdError::generic_err(
63                  "receipt id must be between 8 and 100 cha
64              ));
65          }
66
67          Ok(Self(desired))
68      }
69  }
```

CVSS

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (3.4)

Recommendation

It is recommended to change the final line of the function to store the trimmed value instead of the original.

7.4 (HAL-04) HARDCODED 1:1 PRICING FOR UYLDS.FCC BYPASSES NAV AND CAN CAUSE USER ARBITRAGE IF THE PEG DEVIATES

// LOW

Description

In the UnitPriceFraction in `vault-main/keeper/valuation_engine.go`, the code hardcodes a 1:1 conversion whenever either the `vault.PaymentDenom` or `vault.UnderlyingAsset` equals "uylds.fcc". This bypasses the Marker module's NAV feeds entirely. If uylds.fcc ever deviates from parity with the other denom (even temporarily), normal users can deposit the weaker asset and mint shares as if it were worth 1:1, then redeem for more valuable assets, draining value from other vault users.

Code Location

This snippet is from UnitPriceFraction showing the `uylds.fcc` fast-path that forces 1:1 conversion.

```
45  func (k Keeper) UnitPriceFraction(ctx sdk.Context, srcDen
46      underlyingAsset := vault.UnderlyingAsset
47      if srcDenom == underlyingAsset {
48          return math.NewInt(1), math.NewInt(1), nil
49      }
50
51      // For now, if either the vault's underlying asset or
52      // we assume a 1:1 equivalence between the payment de
53      // See https://github.com/ProvLabs/vault/issues/73 fo
54      const uyldsFccDenom = "uylds.fcc"
55      if vault.PaymentDenom == uyldsFccDenom || underlyingA
```

```
56              return math.NewInt(1), math.NewInt(1), nil
57          }
58
59          fwd, errF := k.MarkerKeeper.GetNetAssetValue(ctx, src
60          rev, errR := k.MarkerKeeper.GetNetAssetValue(ctx, und
61          // ...
62      }
```

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (3.3)

Recommendation

It is recommended to remove the hardcoded uylds.fcc 1:1 path and always use Marker NAV for conversions, or gate the 1:1 override behind an explicit, monitored, and time-bounded mechanism that can be disabled immediately on any depeg signal.

7.5 (HAL-05) NAV PUBLISH CAN PANIC WHEN TOTALSHARES EXCEEDS UINT64

// LOW

Description

In `publishShareNav` in `vault-main/keeper/reconcile.go`, the code calls `vault.TotalShares.Amount.Uint64()` to populate the marker NAV Volume field. The `cosmossdk.io/math.Int.Uint64()` panics when the value does not fit in uint64, so a sufficiently large vault can trigger a panic during normal operations that call reconcileVaultInterest and publishShareNav, which can halt processing for that transaction and potentially crash the app depending on panic recovery.

Code Location

This `publishShareNav` where `TotalShares` is cast to uint64 for NAV publication.

```go
func (k *Keeper) publishShareNav(ctx sdk.Context, vault *
    vaultMarker, err := k.MarkerKeeper.GetMarker(ctx, vau
    if err != nil {
        return fmt.Errorf("failed to get principal marker
    }
    if !vault.TotalShares.IsPositive() {
        return nil
    }
    tvv, err := k.GetTVVInUnderlyingAsset(ctx, *vault)
    if err != nil {
        return fmt.Errorf("failed to get TVV: %w", err)
    }
    if !tvv.IsPositive() {
        return nil
    }

    k.MarkerKeeper.SetNetAssetValue(ctx, vaultMarker, mar
        Price:  sdk.NewCoin(vault.UnderlyingAsset, tvv),
        Volume: vault.TotalShares.Amount.Uint64(),
    }, types.ModuleName)
    return nil
}
```

BVSS

AO:A/AC:M/AX:H/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (2.2)

Recommendation

It is recommended that before calling `Uint64()`, check `vault.TotalShares.Amount.IsUint64()` and handle overflow explicitly (e.g., skip publishing NAV, cap Volume with a documented rule, or change representation so Volume cannot overflow).

7.6 (HAL-06) MIGRATION DOES NOT TRANSFER ACCUMULATED TOKENS

`// LOW`

Description

When the `migrateWithdrawal` function in `WithdrawalFactory.sol` is called to migrate a Withdrawal contract to a new implementation, any tokens that have accumulated in the old Withdrawal contract from users calling withdraw are left behind with no way to retrieve them. The

contract lacks any mechanism to transfer tokens from the old contract to the new one, rescue tokens after migration, or allow

the burner role to burn tokens from the old contract. This means tokens could become permanently inaccessible if a migration occurs while there is a balance in the old contract.

Code Location

```
118 function migrateWithdrawal(
119     address _paymentTokenAddress,
120     address _shareTokenAddress,
121     address _amlSignerAddress
122 ) external onlyOwner returns (address newWithdrawalAddres
123     address oldWithdrawal = withdrawals[_paymentTokenAddr
124 
125     if (oldWithdrawal == address(0)) revert NoExistingWit
126 
127     newWithdrawalAddress = Clones.clone(implementation);
128 
129     Withdrawal(newWithdrawalAddress).initialize(
130         _shareTokenAddress,
131         _paymentTokenAddress,
132         _amlSignerAddress,
133         msg.sender
134     );
135 
136     withdrawals[_paymentTokenAddress][_shareTokenAddress]
137 
138     emit WithdrawalMigrated(_paymentTokenAddress, _shareT
139 }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (2.0)

Recommendation

It is recommended to add a `rescueTokens` function to the `Withdrawal` contract that allows the `BURN_ADMIN_ROLE` to transfer accumulated tokens to a specified recipient. This function should be called before or during migration to ensure no tokens are stranded. The function should transfer the entire balance of shareToken held by the contract to the designated recipient address.

7.7 (HAL-07) UNBOUNDED LOOP IN

DESTINATION ADDRESS

MANAGEMENT

// LOW

Description

The destination address management in `Depositor.sol` uses an array with unbounded loops for checking existence, adding, and removing addresses. The `isDestination`, `addDestinationAddress`, and `removeDestinationAddress` functions all iterate through the entire `destinationAddresses` array. While the code comments suggest this is efficient for small arrays under 5 elements, there is no enforcement of any limit. If a destination manager adds many addresses, whether maliciously or accidentally, these core functions could exceed block gas limits, causing deposits to fail and making it impossible to manage destination addresses.

Code Location

The `destinationAddresses.length` has no upper limit.

```
204 function isDestination(address _destination) public view
205     for (uint i = 0; i < destinationAddresses.length; i++
206         if (destinationAddresses[i] == _destination) {
207             return true;
208         }
209     }
210     return false;
211 }
212
213 function addDestinationAddress(address _destination) exte
214     if (_destination == address(0)) revert InvalidAddress
215
216     for (uint i = 0; i < destinationAddresses.length; i++
217         if (destinationAddresses[i] == _destination) {
218             emit DestinationAddressSkipped(_destination);
219             return;
220         }
221     }
222
223     destinationAddresses.push(_destination);
224     emit DestinationAddressAdded(_destination);
225 }
```

BVSS

AO:S/AC:L/AX:L/P:N/S:U/C:N/A:C/I:N/D:N/Y:N (2.0)

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (2.0)

Recommendation

It is recommended to replace the array-based lookup with a mapping for `O(1)` lookups. Maintain the array only for enumeration purposes if needed.

7.8 (HAL-08) MIGRATION DOES NOT PRESERVE DESTINATION ADDRESSES

// LOW

Description

When migrating a `Depositor` contract via the `migrateDepositor` function in `DepositorFactory.sol`, the new clone is initialized with an empty destination addresses array. All previously whitelisted destination addresses are lost in the process. This means deposits will fail immediately after migration until the destination manager manually re-adds all addresses. Users who have valid AML signatures may find their transactions reverting unexpectedly because their intended destination is no longer in the whitelist.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (2.0)

Recommendation

It is recommended to explicitly document this behavior clearly so administrators know to re-add destination addresses immediately after migration. Alternatively, add a `copyDestinationsFrom` function to the `Depositor` contract that allows the admin to copy all destination addresses from the old contract to the new one after initialization.

7.9 (HAL-09) BURNSWAPOUTRECEIPT AMOUNT PARAMETER NOT VALIDATED AGAINST RECEIPT

// LOW

Description

The `burn_swap_out_receipt` function in `src/contract.rs` accepts an amount parameter that is not validated against the actual receipt amount stored in the contract. The function loads the receipt, removes it from storage, and then sends the specified amount to the caller without checking if that amount exceeds what the receipt represents. Under the trusted-admin threat model this is not exploitable since only the `bridge_admin` can call this function, but it represents a missing check.

Code Location

```
346 fn burn_swap_out_receipt(
347     deps: DepsMut,
348     info: MessageInfo,
349     id: String,
350     amount: Uint128,
351 ) -> ContractResult<Response> {
352     cw_utils::nonpayable(&info)?;
353     helpers::must_be_bridge_admin(&deps.as_ref(), &info)?
354 
355     if amount.eq(&Uint128::zero()) {
356         return Err(ContractError::TransferAmountCantBeZer
357     }
358 
359     let receipt = SWAP_OUT_RECEIPTS.load(deps.storage, &i
360     SWAP_OUT_RECEIPTS.remove(deps.storage, &id);
361 
362     let msg = cosmwasm_std::BankMsg::Send {
363         to_address: info.sender.to_string(),
364         amount: vec![Coin {
365             amount,
366             denom: receipt.payment_denom,
367         }],
368     };
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (2.0)

Recommendation

It is recommended to add a validation check after loading the receipt to ensure the requested amount does not exceed `receipt.coin.amount`.

7.10 (HAL-10) NO VALIDATION OF TO_ADDRESS PARAMETER

// INFORMATIONAL

Description

The `swap_in` and `swap_out` functions in `src/contract.rs` accept a `to_address` parameter representing the destination address on the target chain without any validation. Empty strings, excessively long strings, or malformed data can be stored.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (1.5)

Recommendation

It is recommended to add a validation to ensure `to_address` is non-empty and does not exceed a reasonable maximum length such as 256 characters.

7.11 (HAL-11) AML SIGNER ADDRESS CANNOT BE ROTATED

// INFORMATIONAL

Description

The `amlSigner` address in both `Depositor.sol` and `Withdrawal.sol` is set once during initialization and cannot be changed afterward. There is no setter function to update this value. This creates operational risk because if the AML signer private key is compromised, the entire contract must be migrated to use a new signer. Standard security practices recommend periodic key rotation, which is not possible with this design.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (1.0)

Recommendation

It is recommended to consider adding an `updateAmlSigner` function protected by an appropriate admin role.

7.12 (HAL-12) EMPTY MIGRATION HANDLER WITHOUT VERSION CHECKING

// INFORMATIONAL

Description

The migrate function in `src/contract.rs` has version checking commented out and performs no validation. While only the wasm-admin can trigger migrations, this allows migration from any version including newer ones, potentially enabling downgrade attacks.

Code Location

```
pub fn migrate(_deps: DepsMut, _env: Env, _msg: MigrateMsg) -> C
    // while we're in testnet phase, we may want to migrate with
    // cw2::ensure_from_older_version(deps.storage, CONTRACT_NAM

    Ok(Response::new().add_attribute("action", "migrate"))
}
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (1.0)

Recommendation

It is recommended to enable version checking before mainnet deployment by uncommenting the `cw2::ensure_from_older_version` call.

7.13 (HAL-13) MISSING INPUT VALIDATION IN TOKEN CREATION

// INFORMATIONAL

Description

The `createToken` function in `TokenFactory.sol` does not validate the input parameters for name, symbol, and decimals. Empty strings can be passed for the token name or symbol, and extreme decimal values such as 0 or 255 are allowed. While this does not create a direct security vulnerability, it allows the creation of tokens with unusual or potentially confusing configurations that may cause issues in external integrations or user interfaces.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (0.5)

Recommendation

It is recommended to add a validation to ensure the name and symbol strings are not empty and that the decimals value is within a reasonable range such as 0 to 18.

7.14 (HAL-14) SWAPOUT ZERO-AMOUNT VALIDATION RELIES ON BASEAPP VALIDATEBASIC

// INFORMATIONAL

Description

The `MsgSwapOutRequest.ValidateBasic` function in `vault-main/types/msgs.go` rejects zero amounts, and Cosmos SDK BaseApp calls ValidateBasic for messages that implement `sdk.HasValidateBasic`. However, `msgServer.SwapOut` in vault-main/keeper/msg_server.go and Keeper.SwapOut in vault-main/keeper/vault.go do not perform an explicit `shares.Amount > 0` check. This is not exploitable through normal transactions if ValidateBasic is always executed, but it is a fragile assumption for internal callers and future integrations, and a small guard would prevent accidental queueing/processing of zero-share requests if any path bypasses ValidateBasic.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to add explicit checks in `Keeper.SwapOut` and `Keeper.SwapIn` (and/or MsgServer handlers) to enforce shares/assets are positive, even though `BaseApp` calls `ValidateBasic`.

7.15 (HAL-15) MISSING ZERO-AMOUNT CHECK IN BURNSWAPINRECEIPT

// INFORMATIONAL

Description

The `burn_swap_in_receipt` function in `src/contract.rs` does not check if `burn_amount` is zero, unlike `burn_swap_out_receipt` which explicitly rejects zero amounts. This inconsistency could allow wasteful zero-amount burn operations. Since only `bridge_admin` can invoke this function, it is not exploitable without admin action.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to add a zero-amount check at the beginning of the function for consistency with `burn_swap_out_receipt`.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.