SHERLOCK

Security Review For Nuva Labs

Collaborative Audit Prepared For: **Nuva Labs**
Lead Security Expert(s): **defsec**
Kirkeelee
Date Audited: **November 21 - December 5, 2025**

Introduction

NUVA unlocks global access to the world’s best real-world assets (RWAs). With no minimums and no lockups, users of NUVA earn yield from institutional‑grade RWAs and have the ability to user their tokens freely across DeFi. NUVA was co-created by Animoca Brands and NU Blockchain Technologies, and is stewarded by the NUVA Foundation for progressive decentralization. Learn more at NUVA.finance.

Scope

Repository: ProvLabs/nuva-cosmos-contracts

Audited Commit: 76ec0b36a276540a626973ec7c5243852a4d0749

Final Commit: 001d23527c8d29b05f0b91d49cc96c21577ab13e

Files:

- src/bin/schema.rs
- src/contract.rs
- src/helpers.rs
- src/lib.rs
- src/msg.rs
- src/state.rs

Repository: ProvLabs/nuva-evm-contracts

Audited Commit: 9da04e4841fab051e5c338ff5ddef831fa7488c6

Final Commit: f4f53b2f1a2434cad5f9572c17ad6e6619eca34b

Files:

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

Repository: ProvLabs/vault

Audited Commit: 5e6a36f859a09c3c240f38bf2a3b09b454608e47

Final Commit: 7fa6c5a16ae4cfe8599281b5eefbe8c1daa66fa9

Files:

- interest/interest.go
- keeper/abci.go
- keeper/genesis.go
- keeper/keeper.go
- keeper/msg_server.go
- keeper/payout.go
- keeper/query_server.go
- keeper/queue.go
- keeper/reconcile.go
- keeper/state.go
- keeper/valuation_engine.go
- keeper/vault.go
- module.go
- queue/payout_timeout.go
- queue/pending_swap_out.go
- types/codec.go
- types/genesis.go
- types/keys.go
- types/msgs.go
- types/payout.go
- types/vault.go
- utils/accounts.go
- utils/math.go
- utils/query/query.go
- utils/shares.go
- utils/slices.go
- utils/tools.go

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
1	12	23

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: `TokenFactory` transfer/allowance functions are non-functional [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/41

Summary

The `TokenFactory` contract contains five token utility functions (`safeTransferToken()`, `safeTransferFromToken()`, `safeIncreaseAllowance()`, `safeDecreaseAllowance()`, and `forceApproveToken()`) that are fundamentally broken. These functions operate on the factory contract's own token balances and allowances rather than the caller's, making them completely non-functional for any practical use case.

Vulnerability Detail

the TokenFactory contract implements several ERC20 token utility functions that appear to be intended for user convenience. However, these functions have a critical design flaw:

1. **safeTransferToken()** - Calls `IERC20(token).safeTransfer()` which attempts to transfer from the factory's balance, not the caller's
2. **safeIncreaseAllowance()** - Increases the factory's allowance to a spender, not the caller's allowance
3. **safeDecreaseAllowance()** - Decreases the factory's allowance, not the caller's
4. **forceApproveToken()** - Sets the factory's approval, not the caller's

The factory contract is not designed to hold tokens or manage allowances. It's purely a deployment contract for creating new CustomToken instances. Therefore:

- The factory's token balance is always 0 (no mechanism exists to send tokens to it)
- The factory's allowances are meaningless (it never needs to approve spenders)
- Users calling these functions expecting to manage their own tokens will experience transaction failures or no-ops

The only function that could theoretically work is `safeTransferFromToken()`, but it requires the caller to have pre-approved the factory contract, which makes no sense in the context of a token factory.

Impact

- Users calling `safeTransferToken()` expecting to transfer their tokens will fail with ”insufficient balance” or the transaction will silently do nothing.

Code Snippet

TokenFactory.sol:60-103

```
/**
 * @notice Safely transfers tokens from the sender to the specified address.
 * @param token The address of the token to transfer.
 * @param to The address to which the tokens will be transferred.
 * @param amount The amount of tokens to transfer.
 */
function safeTransferToken(address token, address to, uint256 amount) external {
    IERC20(token).safeTransfer(to, amount);
    //  Transfers from factory's balance (always 0)
    // Should be: IERC20(token).safeTransferFrom(msg.sender, to, amount);
}

/**
 * @notice Safely transfers tokens from the specified address to the specified
 ↪  address.
 * @param token The address of the token to transfer.
 * @param from The address from which the tokens will be transferred.
 * @param to The address to which the tokens will be transferred.
 * @param amount The amount of tokens to transfer.
 */
function safeTransferFromToken(address token, address from, address to, uint256
 ↪  amount) external {
    IERC20(token).safeTransferFrom(from, to, amount);
    //  Requires 'from' to have approved factory contract (why would they?)
}

/**
 * @notice Safely increases the allowance for a spender to transfer tokens on
 ↪  behalf of the sender.
 * @param token The address of the token to transfer.
 * @param spender The address of the spender.
 * @param addedValue The amount of tokens to add to the allowance.
 */
function safeIncreaseAllowance(address token, address spender, uint256 addedValue)
 ↪  external {
    IERC20(token).safeIncreaseAllowance(spender, addedValue);
    //  Increases factory's allowance, not caller's allowance
}

/**
 * @notice Safely decreases the allowance for a spender to transfer tokens on
 ↪  behalf of the sender.
 * @param token The address of the token to transfer.
 * @param spender The address of the spender.
 * @param subtractedValue The amount of tokens to subtract from the allowance.
 */
```

```
function safeDecreaseAllowance(address token, address spender, uint256
 ↪  subtractedValue) external {
    IERC20(token).safeDecreaseAllowance(spender, subtractedValue);
    //  Decreases factory's allowance, not caller's allowance
}

/**
 * @notice Forces the approval of a spender to transfer tokens on behalf of the
  ↪  sender.
 * @param token The address of the token to transfer.
 * @param spender The address of the spender.
 * @param amount The amount of tokens to approve.
 */
function forceApproveToken(address token, address spender, uint256 amount) external
 ↪  {
    IERC20(token).forceApprove(spender, amount);
    //  Approves from factory's perspective, not caller's
}
```

Tool Used

Manual Review

Recommendation

Consider removing all non-functional functions.

Discussion

scirner22

Corrective action here should be to totally remove these functions - they are unused.

defsec

Fix is confirmed on the https://github.com/ProvLabs/nuva-evm-contracts/commit/f4f53b2f1a2434cad5f9572c17ad6e6619eca34b.

Issue M-1: Missing access control on factory contract creation functions [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/40

Summary

The three factory contracts (`DepositorFactory`, `WithdrawalFactory`, and `TokenFactory`) all lack proper access control on their respective creation functions (`createDepositor()`, `createWithdrawal()`, and `createToken()`).

Vulnerability Detail

All three factory contracts inherit from OpenZeppelin's `Ownable` contract but fail to implement the `onlyOwner` modifier on their critical creation functions:

1. DepositorFactory.createDepositor()

- Anyone can create depositor clones with arbitrary AML signer addresses
- No validation that the caller is authorized to create depositors
- The attacker controls the `_amlSignerAddress` parameter during initialization

2. WithdrawalFactory.createWithdrawal()

- Anyone can create withdrawal clones with arbitrary AML signers and burn user addresses
- Both `_amlSignerAddress` and `_burnUser` parameters can be attacker-controlled
- No authorization check before contract deployment

3. TokenFactory.createToken()

- Anyone can create unlimited custom ERC20 tokens
- No restrictions on token names, symbols, or creation frequency
- Causes unbounded growth of the `allTokens` array

The root cause is that these functions are marked as `external` without any access control modifiers, despite the factories inheriting from `Ownable`.

Impact

- Attackers create malicious depositors with their own AML signers, bypassing compliance checks to steal user deposits.

- Rogue depositors for popular token pairs trick users into depositing funds that are then stolen.

Tool Used

Manual Review

Recommendation

Add the `onlyOwner` modifier to all factory creation functions to restrict access to owner.

Discussion

scirner22

Thank you. This will be corrected.

defsec

Fix is confirmed on the https://github.com/ProvLabs/nuva-evm-contracts/commit/f4f53b2f1a2434cad5f9572c17ad6e6619eca34b.

Issue M-2: No mechanism to recover accumulated dust [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/43

Summary

The contract accumulates ”dust” (small residual token amounts from rounding during partial burns) in its internal storage but provides no administrative function to withdraw these tokens. The contract includes tracking and query functionality for dust but completely lacks any execute message to recover it, resulting in tokens becoming permanently locked in the contract as dust accumulates over time.

Vulnerability Detail

The vault proxy contract is designed to handle cross-chain token bridging through a receipt system. When burning receipts, users can specify a `burn_amount` that is less than the full `receipt.coin.amount`, creating ”dust” that represents the difference.

The contract never imports or uses `BankMsg`, `CosmosMsg`, or any token transfer functionality except through the vault module burn operation. There is literally no code path to move tokens from the contract to an external address.

Impact

No recovery mechanism means permanent loss of protocol funds.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/main/nuva-cosmos-contracts/src/contract.rs#L217

Tool Used

Manual Review

Recommendation

Implement a `WithdrawDust` execute message that allows the `bridge_admin` to recover accumulated dust and send it to a specified recipient address.

Discussion

scirner22

Our plan was to add that functionality and migrate the contract at some point in time when it was needed. Right now it's not clear what we would want to do with the dust and based on expected value it would require 10 million swap-ins in order to accumulate $5 dollars of value in dust.

Are there downsides to this approach?

defsec

Thanks for the explanation. I agree that the expected dust value is low in the short term. Migrations introduce operational overhead however It should be fine.

defsec

Marked as a fixed with migration.

scirner22

@defsec it seems this is the last outstanding issue on the dashboard? Are we in agreeance that at some point a legal decision will be made on what we're allowed to do with this dust and at that point we will implement a fix and migrate any instantiated contracts. Until then we're fine with letting the dust accumulate.

defsec

Hi @scirner22 , agreed, thank you!

Issue M-3: AML signature lacks chain id binding [RE-SOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/51

Summary

The AML message hash in the `Withdrawal` contract does not include `block.chainid`, allowing AML signatures to be replayed across different chains. While `address(this)` is included (as `_destinationAddress`), the missing chain ID creates a cross-chain replay vulnerability.

Vulnerability Detail

The `getMessageHash` function in `Withdrawal.sol` creates a hash from:

```
function getMessageHash(
    address sender,
    uint256 _amount,
    address _destinationAddress,  // @audit This is address(this)
    uint256 _deadline
) private view returns (bytes32) {
    return
        keccak256(
            abi.encodePacked(
                sender,
                address(shareToken),
                paymentToken,
                _amount,
                _destinationAddress,  // @audit address(this) IS included
                _deadline
                // @audit block.chainid is MISSING
            )
        );
}
```

Impact

AML signatures can be replayed on different chains.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/main/nuva-evm-contracts/contracts/Withdrawal.sol#L348

Tool Used

Manual Review

Recommendation

Include `block.chainid` in the message hash:

```
function getMessageHash(
    address sender,
    uint256 _amount,
    address _destinationAddress,
    uint256 _deadline
) private view returns (bytes32) {
    return
        keccak256(
            abi.encodePacked(
                block.chainid,
                sender,
                address(shareToken),
                paymentToken,
                _amount,
                _destinationAddress,
                _deadline
            )
        );
}
```

Alternative: Use EIP-712 Standard

For better security and standardization, consider using EIP-712 typed data signing:

```
bytes32 DOMAIN_SEPARATOR = keccak256(
    abi.encode(
        keccak256("EIP712Domain(string name,string version,uint256 chainId,address
         ↪  verifyingContract)"),
        keccak256("Withdrawal"),
        keccak256("1"),
        block.chainid,
        address(this)
    )
);
```

```
function getMessageHash(...) private view returns (bytes32) {
    return keccak256(
        abi.encodePacked(
            "\x19\x01",  // EIP-712 prefix
            DOMAIN_SEPARATOR,
            keccak256(abi.encode(
                keccak256("Withdraw(address sender,address shareToken,address
                ↪  paymentToken,uint256 amount,address destination,uint256
                ↪  deadline)"),
                sender,
                address(shareToken),
                paymentToken,
                _amount,
                _destinationAddress,
                _deadline
            ))
        )
    );
}
```

Discussion

scirner22

Great suggestion!

We shall add `address(this)` and `block.chainid` to all aml checks.

scirner22

@defsec do you agree that with the EIP-712 addition the following should suffice?

```
bytes32 DOMAIN_SEPARATOR = keccak256(
    abi.encode(
        keccak256("EIP712Domain(string name,string version,uint256 chainId,address
        ↪  verifyingContract)"),
        keccak256("Withdrawal"),
        keccak256("1"),
        block.chainid,
        address(this)
    )
);

function getMessageHash(...) private view returns (bytes32) {
    return keccak256(
        abi.encodePacked(
            "\x19\x01",  // EIP-712 prefix
            DOMAIN_SEPARATOR,
            keccak256(abi.encode(
                keccak256("Withdraw(address sender,uint256 amount,uint256
                 ↪  deadline)"),
                sender,
                _amount,
                _deadline
            ))
        )
    );
}
```

scirner22

The thought being that shareToken and paymentToken are superfluous once we include the verifyingContract address (`address(this)`).

defsec

Hi @scirner22 , Have a great day!

Yes, including `block.chainid` and `address(this)` in the `DOMAIN_SEPARATOR` **does** bind the signature to the specific chain and contract, so from that perspective it's sufficient for preventing cross-chain and cross-contract replay.

However, note one thing:

Your updated typed struct:

```
keccak256("Withdraw(address sender,uint256 amount,uint256 deadline)")
```

no longer includes `shareToken`, `paymentToken`, or `destinationAddress` (which in your current implementation is `address(this)`). If those fields were intentionally part of the authorization scope before, removing them changes what the signer is attesting to.

So the EIP-712 setup is correct from a *chain-binding* standpoint, but you should verify whether those omitted fields are still required for signature integrity.

defsec

Oh I see, sorry missed last message. Are we expecting any different payment token on our end?

scirner22

No, the shareToken and paymentToken are set during contract initialization and cannot change. They were only being used as a way to prevent replay across different Withdrawal contracts, but with `address(this)` and `block.chainid` it seems like we are better covered.

defsec

Got it, than that makes sense! I was thinking about multiple token deployments If we need to offchain tracking. However, address(this) covers that

defsec

Fix is confirmed on the https://github.com/ProvLabs/nuva-evm-contracts/commit/f4f53b2f1a2434cad5f9572c17ad6e6619eca34b.

Issue M-4: Griefing attack via residual dust and incorrect initial mint condition [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/52

Summary

The vault has a bug in how it calculates shares for the ”first” depositor. It checks if the vault has zero assets to decide if it is empty. However, if a user withdraws all their funds, small rounding errors (”dust”) can leave a tiny amount of assets behind while the total shares go to zero. This causes the next depositor to be treated incorrectly, resulting in them receiving far fewer shares than they should.

Vulnerability Detail

The vault employs two distinct formulas for share calculation:

Initial Mint: Used when the vault is empty. It sets the initial exchange rate using a fixed scalar (`ShareScalar` = 1,000,000). Formula: Shares = Deposit * 1,000,000

Subsequent Mint: Used for later deposits. It includes `Virtual Shares` and `Virtual Assets` to mitigate inflation attacks. Formula: Shares = Deposit * (TotalShares + 1,000,000) / (TotalAssets + 1)

The vulnerability exists in the condition used to select the formula: `if totalAssets.IsZero() { ... }`

An attacker can exploit this by depositing into a fresh vault and then redeeming 100% of their shares. Due to floor arithmetic in the redemption formula (Payout = floor(Shares * TotalAssets / TotalShares)), the payout may be slightly less than the total assets held if the vault has accrued any value or simply due to integer division properties. For example, if the vault holds 101 assets and the attacker owns 100% of the shares, the payout might be 100.

This leaves the vault in a state where `TotalShares` is 0 but `TotalAssets` is 1 (the residual dust). When a new victim deposits 100 units, the condition totalAssets.IsZero() evaluates to false because of the dust. The system falls back to the Subsequent Mint logic: Shares = 100 * (0 + 1,000,000) / (1 + 1), which simplifies to 100 * 1,000,000 / 2, resulting in 50,000,000 shares.

Since `TotalShares` is 0, the vault is effectively empty or reset, and the user should trigger the Initial Mint logic, receiving 100,000,000 shares. Instead, the victim receives 50% of the expected shares. While they technically own 100% of the shares (and thus 100% of the assets), the exchange rate is permanently halved (500,000 shares/asset instead of 1,000,000).

Impact

The first depositor after a vault reset receives a significantly worse exchange rate than intended, constituting a griefing attack. The vault's exchange rate is permanently deviated from the intended ShareScalar, causing state corruption.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/6d949a344913469816bbcd75e1e05c557364bf4b/vault/utils/shares.go#L58-L68

Tool Used

Manual Review

Recommendation

Ideally both `totalShares` and `totalAssets` should be checked to see if it is the first deposit.

Discussion

nullpointer0x00

I believe this one would not be possible to execute, see: https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/48allowbreak #issuecomment-3571775660 ...

Kirkeelee

@nullpointer0x00 Yes, it is invalid if direct transfers are impossible. Closing it.

Kirkeelee

@nullpointer0x00 re-opened the issue as ”dust” left behind can cause the same effect as a donation in an empty vault.

Issue M-5: Unpausing vault does not re-enable interest accrual period [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/57

Summary

The `UnpauseVault` function does not reset `PeriodStart` or `PeriodTimeout` when unpausing a vault. This causes interest to be calculated for the entire duration from the old `PeriodStart` to the current block time, including the paused period.

Vulnerability Detail

hen a vault is paused, `reconcileVaultInterest` returns early (no-op) and `PeriodStart/PeriodTimeout` remain at their old values. When the vault is unpaused, `UnpauseVault` sets `Paused = false` but does NOT reset `PeriodStart` or `PeriodTimeout`.

The next time `reconcileVaultInterest` is called (e.g., on `SwapIn`), it checks if `currentBlockTime > vault.PeriodStart` (old timestamp). If true, it calls `PerformVaultInterestTransfer`, which calculates `periodDuration = currentBlockTime - vault.PeriodStart`. This duration includes the entire paused period, causing interest to be incorrectly calculated for time when the vault was paused.

Impact

Interest is calculated for the entire paused period, leading to significant over-accrual for long pauses.

Code Snippet

UnpauseVault Missing Period Reset (vault/keeper/msg_server.go:497-532):

```
func (k msgServer) UnpauseVault(goCtx context.Context, msg
 ↪  *types.MsgUnpauseVaultRequest) (*types.MsgUnpauseVaultResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)

    vaultAddr := sdk.MustAccAddressFromBech32(msg.VaultAddress)
    vault, err := k.GetVault(ctx, vaultAddr)
    // ... error handling ...

    vault.PausedBalance = sdk.Coin{}
    vault.Paused = false
    vault.PausedReason = ""
    //  Missing: vault.PeriodStart = ctx.BlockTime().Unix()
    //  Missing: vault.PeriodTimeout = 0
    if err := k.SetVaultAccount(ctx, vault); err != nil {
        return nil, fmt.Errorf("failed to set vault account: %w", err)
    }

    // ... rest of function ...
}
```

Interest Calculation Including Paused Period (vault/keeper/reconcile.go:90-130):

```
func (k *Keeper) PerformVaultInterestTransfer(ctx sdk.Context, vault
 ↪  *types.VaultAccount) error {
    currentBlockTime := ctx.BlockTime().Unix()

    if currentBlockTime <= vault.PeriodStart {
        return nil
    }

    //  periodDuration includes the paused period
    periodDuration := currentBlockTime - vault.PeriodStart

    // ...
    interestEarned, err := interest.CalculateInterestEarned(principal,
     ↪  vault.CurrentInterestRate, periodDuration)
    // ...
}
```

Tool Used

Manual Review

Recommendation

Consider resetting the interest accrual period after unpausing.

Discussion

nullpointer0x00

This one is the highest on my list to solve. In the mean time, if we want to pause a vault for a long period of time that we do not want to pay interest for, we will have to manually set the interest to 0.0 then pause it. We can do that as a multi msg tx. Then do the reverse on unpause. I'm happy you found this.

defsec

Fixed with https://github.com/ProvLabs/vault/pull/139/files.

Issue M-6: Critical logic error in `_doDeposit` allows infinite bridge minting via self-transfer [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/59

Summary

The `_doDeposit` function transfers the deposited stablecoins to the `_destinationAddress` specified by the caller. By setting this address to their own wallet, a user can perform a self-transfer that emits a valid Deposit event without locking any funds. This allows an attacker to trick the off-chain bridge into minting Nuva Tokens (shares) while retaining their original stablecoins.

Vulnerability Detail

The `_doDeposit` function takes the `_destinationAddress` input and uses it as the target for the ERC20 transfer of the deposit token.

```
function _doDeposit(uint256 _amount, address _destinationAddress) private {
        if (_amount == 0) {
            revert InvalidAmount();
        }
        if (_destinationAddress == address(0)) {
            revert InvalidAddress("destination");
        }

        depositToken.safeTransferFrom(msg.sender, _destinationAddress, _amount);

        emit Deposit(msg.sender, _amount, address(depositToken), shareToken,
        ↪   _destinationAddress);
    }
```

The attack begins with an attacker holding a balance of stablecoins, for example 100 USDC. The attacker calls the deposit function and specifies their own wallet address as the destination address. The contract executes the transferFrom function, moving 100 USDC from the attacker to the attacker. Consequently, the attacker's balance remains unchanged and no funds are locked in the contract. However, the contract emits a valid Deposit event. The off-chain bridge observes this event and, believing a valid deposit occurred, mints corresponding Nuva Tokens(shares) to the attacker provided `_destinationAddress`. The attacker has now effectively double-spent their funds, retaining the original stablecoins while receiving the shares. This process can be repeated indefinitely.

Impact

This vulnerability allows for infinite minting of Nuva Tokens on the `_destinationAddress` without any backing assets. This leads to immediate protocol insolvency as the minted shares have no underlying value locked in the bridge.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/6d949a344913469816bbcd75e1e05c557364bf4b/nuva-evm-contracts/contracts/Depositor.sol#L175-L186

Tool Used

Manual Review

Recommendation

Modify `_doDeposit` to transfer the tokens to a protocol controlled address to enforce custody/locking. The `_destinationAddress` should only be used in the event log to indicate where the shares should be minted.

Discussion

scirner22

In practice it wouldn't really be attackable like this because the destinationAddress must go to a specific place (cross chain bridge address) which locks and mints the tokens on Provenance. The offchain system reads the event emitted here and combines it with mint events on Provenance in order to advance. Also, the destinationAddress is present in the amlSignature and the amlSigner address is set during the initialization step. The attack vector here would be if our FE allowed the user to craft a request in which it attested to an invalid destinationAddress. That would not result in the user getting minted nuTOKEN shares, but rather just send the funds to an address that might or might not be in their control.

This surfaces the fact that the FE could have a bug and send to an incorrect destinationAddress, in which case the user's funds could be locked up (never recoverable). I'm going to implement a two step check here, add a destinationAddress admin which can add and remove an allowlist of destination addresses and then during deposit we verify the requested destinationAddress is contained in that list in order to proceed. In order for funds to go to the wrong place, the destinationAddress list admin and the Nuva FE would both need to incorrectly set the same bad address rather than either side providing bad data alone.

For more background, the destinationAddress is a third party bridge address which is relatively static, but can change a few times per year.

Kirkeelee

Thanks for the feedback. Downgraded to medium just to point out that proper verification is need for `_destinationAddress`.

scirner22

This should be fixed now since there was an introduction of a destinationAddress admin to manage an allow list of destination addresses?

Issue M-7: Genesis import missing `PayoutVerificationSet` restoration [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/60

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `InitGenesis` function imports `PayoutTimeoutQueue` entries from genesis state but completely ignores the `PayoutVerificationSet`. This means vaults pending verification on chain upgrade/restart will be lost from the verification queue.

Vulnerability Detail

In `genesis.go:55-77`, only `PayoutTimeoutQueue` and `PendingSwapOutQueue` are imported:

```go
// genesis.go:55-63
for _, entry := range genState.PayoutTimeoutQueue {
    addr, err := sdk.AccAddressFromBech32(entry.Addr)
    if err != nil {
        panic(fmt.Errorf("invalid address in timeout queue: %w", err))
    }
    if err := k.PayoutTimeoutQueue.Enqueue(ctx, int64(entry.Time), addr); err !=
    ↪  nil {
        panic(fmt.Errorf("failed to enqueue vault timeout for %s: %w", entry.Addr,
        ↪  err))
    }
}
// PendingSwapOutQueue is imported at line 75-77
// PayoutVerificationSet is never imported!
```

Similarly, `ExportGenesis` at `genesis.go:81-114` only exports `PayoutTimeoutQueue`:

```go
// genesis.go:91-102
err := k.PayoutTimeoutQueue.Walk(ctx, func(periodTimeout uint64, vaultAddr
↪  sdk.AccAddress) (stop bool, err error) {
    paymentTimeoutQueue = append(paymentTimeoutQueue, types.QueueEntry{
        Time: periodTimeout,
        Addr: vaultAddr.String(),
    })
    return false, nil
})
// PayoutVerificationSet is never exported!
```

Impact

On chain upgrade or genesis restart:

1. Vaults in the `PayoutVerificationSet` will be ”forgotten”
2. These vaults won't have their `handleReconciledVaults()` processing run in EndBlocker
3. Interest reconciliation for these vaults will be skipped
4. Vaults may need manual re-registration via interest rate update

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/6d949a344913469816bbcd75e1e05c557364bf4b/vault/keeper/genesis.go

Tool Used

Manual Review

Recommendation

1. Add `PayoutVerificationSet []string` to `GenesisState` proto definition.
2. Export all addresses from `PayoutVerificationSet` in `ExportGenesis`.
3. Import and restore `PayoutVerificationSet` in `InitGenesis`.

Discussion

nullpointer0x00

Adding this to our next milestone.

Issue M-8: `autoPauseVault` sets zero `PausedBalance` due to circular dependency [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/68

Summary

The `autoPauseVault` function sets `vault.Paused = true` before calling `GetTVVInUnderlyingAsset`. Since `GetTVVInUnderlyingAsset` returns `PausedBalance.Amount` when the vault is paused, and `PausedBalance` hasn't been set yet, it returns zero. This zero value is then stored as PausedBalance, creating a circular dependency that always results in zero PausedBalance.

Vulnerability Detail

The `autoPauseVault` function in `vault/keeper/vault.go` has an ordering issue. The function sets `vault.Paused = true` before calling GetTVVInUnderlyingAsset. However, `GetTVVInUnderlyingAsset` checks if the vault is paused and returns `PausedBalance.Amount`. Since PausedBalance hasn't been set yet, it returns zero. This zero value is then stored as `PausedBalance`, creating a circular dependency.

Additionally, if `GetTVVInUnderlyingAsset` fails due to NAV conversion errors, the error is logged but ignored, and zero is stored anyway.

Impact

Vault is always paused with zero PausedBalance when auto-paused, regardless of actual vault value. All TVV queries while paused return zero, showing incorrect vault value.

Code Snippet

```go
func (k *Keeper) autoPauseVault(ctx context.Context, vault *types.VaultAccount,
↪  reason string) {
    sdkCtx := sdk.UnwrapSDKContext(ctx)

    sdkCtx.Logger().Error(
        "Auto-pausing vault due to critical error",
        "vault_address", vault.GetAddress().String(),
        "reason", reason,
    )

    vault.Paused = true
    vault.PausedReason = reason
    tvv, err := k.GetTVVInUnderlyingAsset(sdkCtx, *vault)
    if err != nil {
        sdkCtx.Logger().Error("Failed to get TVV in underlying asset",
         ↪  "vault_address", vault.GetAddress().String(), "error", err)
    }

    vault.PausedBalance = sdk.Coin{Denom: vault.UnderlyingAsset, Amount: tvv}
    k.AuthKeeper.SetAccount(ctx, vault)

    k.emitEvent(sdkCtx, types.NewEventVaultPaused(vault.GetAddress().String(),
     ↪  vault.GetAddress().String(), reason, vault.PausedBalance))
}
```

Tool Used

Manual Review

Recommendation

Fix the ordering issue by getting TVV before setting `vault.Paused = true`.

Discussion

nullpointer0x00

Good catch. You are right about the ordering issue. Setting the Paused flag too early causes the zero balance.

That said, if this triggers, the vault is already in a critical state and needs an admin to fix it manually. The zero balance is just a display bug at that point, not a security risk.

I will fix it in the next release. I think it should be lower severity though.

Issue M-9: `handleVaultInterestTimeouts` dequeues timeout but does not re-enqueue on failure [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/73

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `handleVaultInterestTimeouts` function dequeues timeout entries before attempting interest reconciliation. If `PerformVaultInterestTransfer` fails, the function continues without re-enqueuing a timeout or resetting `PeriodStart`. While the vault can recover through user operations that trigger reconcileVaultInterest, automatic interest reconciliation is broken until recovery occurs, and if there's no user activity, the vault remains stuck without automatic reconciliation.

Vulnerability Detail

The `handleVaultInterestTimeouts` function processes vaults with expired interest periods:

1. The timeout entry is dequeued from `PayoutTimeoutQueue`
2. If `PerformVaultInterestTransfer` fails, the error is logged and execution continues
3. Only successful reconciliations are added to the reconciled list
4. `resetVaultInterestPeriods` is only called for vaults in the reconciled list.

Impact

Automatic interest reconciliation stops working if `PerformVaultInterestTransfer` fails in `handleVaultInterestTimeouts`. Interest will not accrue automatically until user activity triggers recovery or the issue is manually resolved. If there's no user activity, the vault remains stuck without automatic reconciliation indefinitely. PeriodStart remains stale, causing incorrect interest calculations if reconciliation is later triggered. The vault requires user activity or manual intervention to restore automatic interest reconciliation.

Code Snippet

```go
func (k *Keeper) handleVaultInterestTimeouts(ctx context.Context) error {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    now := sdkCtx.BlockTime().Unix()
```

```go
var keysToProcess []collections.Pair[uint64, sdk.AccAddress]
var depleted []*types.VaultAccount
var reconciled []*types.VaultAccount

err := k.PayoutTimeoutQueue.WalkDue(ctx, now, func(timeout uint64, addr
↪  sdk.AccAddress) (bool, error) {
    key := collections.Join(timeout, addr)
    vault, ok := k.tryGetVault(sdkCtx, addr)
    if ok && vault.Paused {
        return false, nil
    }
    keysToProcess = append(keysToProcess, key)
    return false, nil
})
if err != nil {
    return fmt.Errorf("walk failed: %w", err)
}

for _, key := range keysToProcess {
    timeout := key.K1()
    addr := key.K2()

    // Timeout is dequeued FIRST
    if err := k.PayoutTimeoutQueue.Dequeue(ctx, int64(timeout), addr); err !=
    ↪  nil {
        sdkCtx.Logger().Error("CRITICAL: failed to dequeue interest timeout,
        ↪  skipping", "vault", addr.String(), "err", err)
        continue
    }

    vault, ok := k.tryGetVault(sdkCtx, addr)
    if !ok {
        continue
    }

    periodDuration := int64(timeout) - vault.PeriodStart
    if periodDuration < 0 {
        periodDuration = now - vault.PeriodStart
    }

    canPay, err := k.CanPayoutDuration(sdkCtx, vault, periodDuration)
    if err != nil {
        sdkCtx.Logger().Error("failed to check payout ability", "vault",
        ↪  addr.String(), "err", err)
        continue  // Timeout dequeued but not re-enqueued
    }

    if !canPay {
        depleted = append(depleted, vault)
        continue
        }

        // If this fails, timeout is dequeued but not re-enqueued
        if err := k.PerformVaultInterestTransfer(sdkCtx, vault); err != nil {
            sdkCtx.Logger().Error("failed to reconcile interest", "vault",
            ↪  addr.String(), "err", err)
            continue  // Timeout dequeued but not re-enqueued, PeriodStart not reset
        }

        reconciled = append(reconciled, vault)
    }

    // Only called for successfully reconciled vaults
    k.resetVaultInterestPeriods(ctx, reconciled)
    k.handleDepletedVaults(ctx, depleted)
    return nil
}
```

Tool Used

Manual Review

Recommendation

Re-enqueue timeout or add vault to a retry mechanism if `PerformVaultInterestTransfer` fails.

Discussion

nullpointer0x00

Good catch. We originally thought the next user action would fix the state automatically. But you are right. If the vault is idle, it gets stuck. That is not ideal.

We need a retry mechanism and perhaps an emitted event might be helpful. I am adding this to the next release.

Issue M-10: Missing slippage protection in swap and bridge operations [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/83

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `SwapIn` and `SwapOut` functions execute conversions between `Assets` and `Shares` at the calculated Net Asset Value (NAV) without allowing the user to specify a minimum acceptable output amount. This exposes users to unlimited slippage due to transactions not being atomic or market volatility during the mandatory withdrawal delay (on redemption). This risk extends to cross-chain bridging operations, where users moving funds to EVM chains have no guarantee of the final exchange rate after the bridging delay.

Vulnerability Detail

In the `SwapIn` function, the number of shares to mint is calculated based on the current NAV at the moment of execution. Since there is no `minShares` parameter, a user's transaction can be front-run by a large deposit or interest update that alters the NAV, resulting in fewer shares than expected.

Similarly, `SwapOut` calculates the redemption amount based on the NAV at the time of execution (after `WithdrawalDelaySeconds`), not the time of request. The `PendingSwapOut` struct does not store a minimum acceptable amount. Consequently, if the vault's value decreases during the delay period, the user is forced to accept the lower payout.

This issue is compounded in cross-chain bridging scenarios. Users bridging shares to EVM chains rely on the NAV remaining stable during the cross-chain message propagation and processing time, but lack any mechanism to revert the transaction if the rate becomes unfavorable.

Impact

Users are vulnerable to sandwich attacks during deposits and unavoidable value loss during withdrawals due to market volatility during the delay period. In cross-chain scenarios, this can lead to significant discrepancies between the value sent and the value received on the destination chain.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/6d949a344913469816bbcd75e1e05c557364bf4b/vault/keeper/vault.go#L168-L224

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/6d949a344913469816bbcd75e1e05c557364bf4b/vault/keeper/vault.go#L243-L300

Tool Used

Manual Review

Recommendation

Update `MsgSwapIn` to include a `MinShares` field and enforce that the minted shares are greater than or equal to this minimum. Update `MsgSwapOut` to include a `MinRedeemAmount` field and store this value in the `PendingSwapOut` record. In the `EndBlocker` execution logic, verify that the calculated payout meets the `MinRedeemAmount`. If it does not, fail the payout and refund the shares to the user. Ensure bridge messages include slippage parameters to protect cross-chain transfers.

Discussion

nullpointer0x00

We have discussed similar parameters for next releases. I will add this to next milestone.

Issue M-11: Withdrawal queue griefing via batch limit [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/84

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The EndBlocker processes only 100 withdrawal requests per block (`MaxSwapOutBatchSize`). An attacker can flood the withdrawal queue with their own requests to delay legitimate users' withdrawals indefinitely.

Vulnerability Detail

The vault module processes pending swap-out requests in batches with a fixed limit of 100 requests per block. The queue is ordered by timestamp first, then by request ID. Since request IDs are sequential, an attacker who creates requests before a victim will have lower IDs and be processed first when requests share the same timestamp.

Batch Limit Implementation (`vault/keeper/abci.go`):

```
const (
    // MaxSwapOutBatchSize is the maximum number of pending swap-out requests
    // to process in a single EndBlocker invocation. This prevents a large queue
    // from consuming excessive block time and memory. This is a temporary value
    // and we will need to do more analysis on a proper batch size.
    MaxSwapOutBatchSize = 100  // Fixed limit
)
```

Queue Processing (`vault/keeper/payout.go`):

```
func (k *Keeper) processPendingSwapOuts(ctx context.Context, batchSize int) error {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    now := sdkCtx.BlockTime().Unix()
    var jobsToProcess []types.PayoutJob

    processed := 0
    err := k.PendingSwapOutQueue.WalkDue(ctx, now, func(timestamp int64, id uint64,
    ↪   vaultAddr sdk.AccAddress, req types.PendingSwapOut) (stop bool, err error) {
        vault, ok := k.tryGetVault(sdkCtx, vaultAddr)
        if ok && vault.Paused {
            return false, nil
        }
        if processed == batchSize {
            return true, nil  // Stops processing after 100 requests
        }
        processed++
        jobsToProcess = append(jobsToProcess, types.NewPayoutJob(timestamp, id,
         ↪  vaultAddr, req))
        return false, nil
    })
    // ... process jobs ...
}
```

Queue Ordering (`vault/queue/pending_swap_out.go`):

The queue key is `(timestamp, id, vault)`, ordered by timestamp first, then ID. Request IDs are sequential:

```
func (p *PendingSwapOutQueue) Enqueue(ctx context.Context, pendingTime int64, req
 ↪  *types.PendingSwapOut) (uint64, error) {
    // ...
    id, err := p.Sequence.Next(ctx)  // Sequential ID generation
    if err != nil {
        return 0, err
    }
    return id, p.IndexedMap.Set(ctx, collections.Join3(pendingTime, id, vault),
     ↪  *req)
}
```

The `SwapOut` function (`vault/keeper/vault.go`) does not enforce any limits on:

- Number of pending requests per user
- Total requests in queue
- Request creation rate

Impact

Legitimate users cannot withdraw funds in a timely manner.

Tool Used

Manual Review

Recommendation

Add a limit on pending withdrawals per user.

Discussion

nullpointer0x00

This one has been discussed for future releases. I'm adding it to next milestone.

Issue M-12: NAV staleness during cooldown [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/89

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Withdrawal amounts are calculated at request time (SwapOut) for validation purposes, but the actual payout amount is recalculated at execution time (processSingleWithdrawal) using the current NAV/TVV. The calculated amount at request time is not stored in the PendingSwapOut request, creating an issue where the payout amount can change between request and execution. This allows the payout amount to be manipulated through large deposits, NAV price updates, or interest reconciliation during the cooldown period, resulting in users receiving different amounts than what was validated at request time.

Vulnerability Detail

The vault module uses a two-phase withdrawal process:

1. **Request Phase (SwapOut)**: User requests withdrawal, assets are calculated for validation, shares are escrowed, and the request is queued with a cooldown period.
2. **Execution Phase (processSingleWithdrawal)**: After the cooldown expires, the EndBlocker processes the queued withdrawal, recalculates assets using current TVV/NAV, and transfers the recalculated amount to the user.

The `PendingSwapOut` structure only stores the number of shares and redeem denom, not the expected asset amount. At execution time, `ConvertSharesToRedeemCoin` is called again using the current vault state, which may have changed during the cooldown period.

Impact

Users receive different payout amounts than what was calculated and validated at request time.

Code Snippet

Request Time Calculation (vault/keeper/vault.go):

```
assets, err := k.ConvertSharesToRedeemCoin(ctx, *vault, shares.Amount, redeemDenom)
if err != nil {
    return 0, fmt.Errorf("failed to calculate assets from shares: %w", err)
}

if err := k.checkPayoutRestrictions(ctx, vault, owner, assets); err != nil {
    return 0, err
}
// assets is not stored in PendingSwapOut
```

PendingSwapOut Structure (vault/proto/provlabs/vault/v1/vault.proto):

```
message PendingSwapOut {
  string owner = 1;
  string vault_address = 2;
  cosmos.base.v1beta1.Coin shares = 3;
  string redeem_denom = 4;
  // @audit No field for expected_assets or calculated_amount
}
```

Execution Time Recalculation (vault/keeper/payout.go):

```
if err := k.reconcileVaultInterest(ctx, &vault); err != nil {
    return fmt.Errorf("failed to reconcile vault interest: %w", err)
}

assets, err := k.ConvertSharesToRedeemCoin(ctx, vault, req.Shares.Amount,
↪   req.RedeemDenom)
// Recalculated using current TVV/NAV, may differ from request time
if err := k.BankKeeper.SendCoins(..., sdk.NewCoins(assets)); err != nil {
    return err
}
```

Tool Used

Manual Review

Recommendation

Store the expected asset amount in `PendingSwapOut` at request time and use it for payout at execution time. This ensures users receive the amount that was calculated and validated at request time.

Discussion

nullpointer0x00

This is expected behavior. When a user requests a swapout, their shares are escrowed in the vault during the cooldown period. However, the conversion rate is intentionally calculated at the time of execution, not the time of request.

This ensures the user receives the actual value of the shares at the moment of payout, including any interest accrued or NAV changes that occurred during the wait.

We have talked about adding slippage parameters in a future release. Also, allowing the cancel of pending swapouts.

Issue L-1: Use `Ownable2Step` instead of `Ownable` for factory contracts [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/42

Summary

The `DepositorFactory` and `WithdrawalFactory` contracts inherit from OpenZeppelin's `Ownable` contract, which uses a single-step ownership transfer process. This implementation is vulnerable to accidental permanent loss of ownership if the new owner address is incorrect or inaccessible. OpenZeppelin's `Ownable2Step` provides a safer two-step transfer process that requires the new owner to accept ownership.

Vulnerability Detail

Both factory contracts inherit from `Ownable`:

DepositorFactory.sol

```
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";

contract DepositorFactory is Ownable {
    constructor(address _implementation) Ownable(msg.sender) {
        // ...
    }
}
```

WithdrawalFactory.sol

```
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";

contract WithdrawalFactory is Ownable {
    constructor(address _implementation) Ownable(msg.sender) {
        // ...
    }
}
```

The standard `Ownable` contract uses `transferOwnership()` which immediately transfers ownership in a single transaction:

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _transferOwnership(newOwner);  // Immediate transfer
}
```

Impact

If ownership is transferred to an incorrect or inaccessible address, the protocol loses the ability to:

- Create new depositor/withdrawal contracts
- Migrate existing contracts to new implementations
- Update factory implementations
- Perform any administrative functions

Tool Used

Manual Review

Recommendation

Replace `Ownable` with `Ownable2Step` in both factory contracts.

Discussion

scirner22

Thank you. This will be corrected.

defsec

Fix is confirmed on the https://github.com/ProvLabs/nuva-evm-contracts/commit/f4f53b2f1a2434cad5f9572c17ad6e6619eca34b.

Issue L-2: Contract has no admin transfer mechanism [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/44

Summary

The contract sets `bridge_admin` during instantiation and provides no mechanism to transfer or update the admin address. If the admin key is lost, compromised, or needs to be rotated for security reasons, the contract becomes permanently locked with no way to recover administrative control over critical functions like `SetSwapInReceiptData` and `BurnSwapInReceipt`.

Vulnerability Detail

The `bridge_admin` address is set once during contract instantiation and stored in the contract state. There is no execute message to transfer or update this address:

```
// Instantiation (contract.rs:37-41)
let bridge_admin = if let Some(bridge_admin) = msg.bridge_admin {
    deps.api.addr_validate(&bridge_admin)?
} else {
    info.sender
};
let state = State::new(
    bridge_admin,
    vault,
    deps.api.addr_validate(&msg.vault_address)?,
)?;
STATE.save(deps.storage, &state)?;
```

Impact

If admin private key is lost, contract becomes permanently unusable.

Tool Used

Manual Review

Recommendation

Add a `TransferAdmin` execute message that allows current admin to transfer control.

Discussion

scirner22

Thank you. This will be added.

defsec

Fixed with https://github.com/ProvLabs/nuva-cosmos-contracts/commit/001d23527c8d29b05f0b91d49cc96c21577ab13e.

Issue L-3: CustomToken constructor contains redundant zero amount mint [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/45

Summary

The `CustomToken` constructor calls `_mint(_admin, 0)` which is a redundant operation that wastes gas. Minting zero tokens has no effect on token balances or state, but still consumes gas for the function call and event emission.

Vulnerability Detail

In `CustomToken.sol:37-47`, the constructor performs a zero-amount mint:

```
constructor(
    string memory _name,
    string memory _symbol,
    address _admin,
    uint8 _decimals
) ERC20(_name, _symbol) ERC20Permit(_name) {
    _customDecimals = _decimals;
    _grantRole(DEFAULT_ADMIN_ROLE, _admin);
    _grantRole(MINTER_ROLE, _admin);
    _mint(_admin, 0);  // @audit Redundant: minting zero tokens has no effect
}
```

Impact

Unnecessary gas consumption on every token deployment.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/main/nuva-evm-contracts/contracts/CustomToken.sol#L46

Tool Used

Manual Review

Recommendation

Remove the redundant zero-amount mint call.

Discussion

scirner22

Thank you. This will be removed.

defsec

Fix is confirmed on the https://github.com/ProvLabs/nuva-evm-contracts/commit/f4f53b2f1a2434cad5f9572c17ad6e6619eca34b.

Issue L-4: Receipt ID collision risk [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/46

Summary

Receipt IDs are user-provided strings without cryptographic uniqueness guarantees. A malicious or buggy off-chain system could create ID collisions.

Vulnerability Detail

The `swap_in` function accepts a user-provided `id` parameter:

```
fn swap_in(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    id: String,  // <-- User-provided
    to_address: String,
) -> ContractResult<Response> {
    // ...
    let payload = SwapInPayload {
        id: id.try_into()?,  // Only length validation
        to_address,
    };
    // ...
}
```

The only validation is in `ReceiptId::try_from()`:

```
impl TryFrom<String> for ReceiptId {
    type Error = StdError;

    fn try_from(desired: String) -> Result<Self, Self::Error> {
        let value = desired.trim();
        if value.is_empty() {
            return Err(StdError::generic_err("receipt id cannot be empty"));
        }

        if value.len() < 8 || value.len() > 36 {
            return Err(StdError::generic_err(
                "receipt id must be between 8 and 36 characters",
            ));
        }

        Ok(Self(desired))
    }
}
```

While the reply handler checks for duplicate IDs:

```
SWAP_IN_RECEIPTS.update(
    deps.storage,
    payload.id.as_ref(),
    |receipt| match receipt {
        Some(_) => Err(StdError::generic_err("receipt id already exists")),
        None => Ok(SwapInReceipt { /* ... */ }),
    },
)?;
```

This creates a front-running vulnerability where:

1. User A submits swap with ID "12345678"
2. User B front-runs with same ID
3. User A's transaction fails
4. User A loses gas and AML checks

Impact

Depends entirely on off-chain coordination.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/main/nuva-cosmos-contracts/src/contract.rs#L74

Tool Used

Manual Review

Recommendation

Generate receipt IDs on-chain using a counter or hash to guarantee uniqueness.

Discussion

scirner22

This id provides assurances that the offchain system does not attempt to reprocess events during partial failure states.

It seems like we can have the interface use `id` as it does now, but internally we can combine it with the sender's address so internally each sender has a unique set of id's they can use.

If you agree with this approach, we will create an issue.

defsec

Thanks for the clarification!

Using the user-provided id as an off-chain deduplication hint makes sense, and combining it internally with the sender's address would indeed eliminate the collision/front-running risk. That approach preserves the existing interface while ensuring that uniqueness is guaranteed at the contract level.

From a security perspective, namespacing receipt IDs by sender (e.g., hash(sender || id)) resolves the current class of issues:

- Prevents malicious users from front-running another user's id.
- Avoids cross-user collisions caused by off-chain retries or partial-failure scenarios.
- Maintains the off-chain system's ability to use the ID for replay-prevention and recovery logic.

scirner22

I thought about this more and instead I'm just going to require `swap_in` be called by the `bridge_admin`. This was not strictly required outside of solving this issue, but in practice this contract is meant to be the proxy for the bridge admin to execute so it makes sense to have all endpoints locked down to the bridge admin.

This also helps after solving the other issue `adding an update_bridge_admin() endpoint` because with my first proposed methodology of scoping the receipt_ids with the user's address, it would mean one bridge admin could not burn receipts for the previous bridge admin which is not the behavior we would want.

defsec

Fixed with https://github.com/ProvLabs/nuva-cosmos-contracts/commit/001d23527c8d29b05f0b91d49cc96c21577ab13e.

Issue L-5: Withdraw function actually deposits tokens [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/50

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `withdraw()` function in the `Withdrawal` contract does not actually withdraw tokens to the user. Instead, it transfers tokens FROM the user TO the contract, which is functionally a deposit/lock operation. This is misleading naming that contradicts the expected behavior of a ”withdraw” function.

Vulnerability Detail

In `Withdrawal.sol`, the `_doWithdraw()` function (called by `withdraw()` and `withdrawWithPermit()`) transfers tokens from the user to the contract:

```
function _doWithdraw(uint256 _amount) private {
    shareToken.safeTransferFrom(msg.sender, address(this), _amount);  // @audit
    ↪  Transfers TO contract
    emit Withdraw(msg.sender, _amount, address(shareToken), paymentToken);
}
```

Impact

- Users expect `withdraw()` to send tokens to them.
- Instead, tokens are locked in the contract.

Code Snippet

```
function _doWithdraw(uint256 _amount) private {
    shareToken.safeTransferFrom(msg.sender, address(this), _amount);  // @audit TO
    ↪  contract, not FROM
    emit Withdraw(msg.sender, _amount, address(shareToken), paymentToken);
}
```

Tool Used

Manual Review

Recommendation

Consider adding a comment or fix the function.

Discussion

scirner22

In this case the full withdrawal is a multi step/chain process that happens both on and off chain. This first step locks the tokens in this contract since the user is exchanging their nuva tokens for a stablecoin.

Nuva tokens get locked bridge burns eth nuva tokens for provenance nuva tokens nuva tokens are swapped for stablecoin payment token bridge withdraws stablecoin payment token to the user's eth wallet

defsec

Hi @scirner22 , Thank you so much for the response! I especially wanted to check if there's any confusion on the off-chain part, since the withdrawal event does not contain any ID or nonce parameter.

scirner22

Yes, thank you. This was something on my radar as an improvement for v2. We will be deciding which proper bridge we will integrate with and then this implementation will change to integrate with that, or we will continue our manual off chain bridging in which case we should be able to implement a receipt system in both Withdraw and Deposit so the off chain minter/burner operations are guarded against replay at the evm level.

Issue L-6: Vault state changes bypass validation via direct `AuthKeeper.SetAccount` [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/53

Summary

Several functions modify vault state and persist it using `k.AuthKeeper.SetAccount()` directly, bypassing the `SetVaultAccount()` wrapper that performs validation. This allows potentially invalid vault states to be persisted.

Vulnerability Detail

The `SetVaultAccount` function in `state.go` validates the vault before saving:

```
func (k *Keeper) SetVaultAccount(ctx sdk.Context, vault *types.VaultAccount) error {
    if err := vault.Validate(); err != nil {
        return err
    }
    k.AuthKeeper.SetAccount(ctx, vault)
    return nil
}
```

However, the following functions bypass this validation by calling `AuthKeeper.SetAccount` directly:

1. `processSingleWithdrawal` **(payout.go):**

```
vault.TotalShares, err = vault.TotalShares.SafeSub(req.Shares)
if err != nil {
    // ...
}
k.AuthKeeper.SetAccount(ctx, &vault)  // Direct call - no validation!
```

2. `UpdateInterestRates` **(reconcile.go):**

```
vault.CurrentInterestRate = currentRate
vault.DesiredInterestRate = desiredRate
k.AuthKeeper.SetAccount(ctx, vault)  // Direct call - no validation!
```

Note: `autoPauseVault` (vault.go) also uses direct `AuthKeeper.SetAccount`, but this is **intentional**, during critical error recovery, pausing the vault takes priority over validation.

Impact

Vault state changes could violate invariants checked by `Validate()`.

Code Snippet

```
// vault/keeper/payout.go
k.AuthKeeper.SetAccount(ctx, &vault)  // Should use SetVaultAccount

// vault/keeper/reconcile.go
k.AuthKeeper.SetAccount(ctx, vault)   // Should use SetVaultAccount
```

Tool Used

Manual Review

Recommendation

Replace direct `AuthKeeper.SetAccount` calls with `SetVaultAccount` in:

1. `processSingleWithdrawal` (payout.go:170)
2. `UpdateInterestRates` (reconcile.go:190)

Discussion

nullpointer0x00

The autoPause mechanism is used to pause the vault when a critical error happens that sets the vault into an invalid state. This is why I chose to skip the validation here. Pausing the vault is the most important thing to do here to prevent any other corruption.

defsec

Hi @nullpointer0x00 , Thank you so much for information, There are some other functions which is not using SetVaultAccount :

```
UpdateInterestRates()
processSingleWithdrawal()
```

Are they intended as well?

nullpointer0x00

Nope, those ones were missed. We will update those. Thanks!

defsec

Fixed with https://github.com/ProvLabs/vault/pull/147/files
allowbreak
#diff-456482902e57c6db3b9397fca5b013668dfa2914861ef8245fec8118ceb67983

Issue L-7: Withdrawal contract grants excessive admin privileges to burnUser [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/54

Summary

The `Withdrawal` contract's `initialize` function grants both `DEFAULT_ADMIN_ROLE` and `BURN_ROLE` to the `burnUser` address. This violates the principle of least privilege, as the `burnUser` only needs `BURN_ROLE` to perform token burns.

Vulnerability Detail

In OpenZeppelin's `AccessControlUpgradeable`, the `DEFAULT_ADMIN_ROLE` is a powerful role that can:

- Grant and revoke any role (including itself)
- Override access controls
- Potentially modify contract state if admin functions exist

The `Withdrawal` contract only has one function that requires role-based access: `burn()`, which uses `onlyRole(BURN_ROLE)`. There are no functions that require `DEFAULT_ADMIN_ROLE`, making the grant of admin privileges unnecessary.

Impact

If `burnUser` account is compromised, attacker has full admin control.

Code Snippet

```
function initialize(
    address _shareTokenAddress,
    address _paymentTokenAddress,
    address _amlSignerAddress,
    address burnUser
) external initializer {
    __AccessControl_init();
    if (_shareTokenAddress == address(0)) {
        revert InvalidAddress("Invalid withdrawal token");
    }
    if (_paymentTokenAddress == address(0)) {
        revert InvalidAddress("Invalid payment token");
    }
    if (_amlSignerAddress == address(0)) {
        revert InvalidAddress("Invalid AML signer");
    }

    shareToken = ICustomToken(_shareTokenAddress);
    paymentToken = _paymentTokenAddress;
    amlSigner = _amlSignerAddress;
    _grantRole(DEFAULT_ADMIN_ROLE, burnUser);
    _grantRole(BURN_ROLE, burnUser);

    emit WithdrawalInitialized(_shareTokenAddress, _paymentTokenAddress,
    ↪   _amlSignerAddress);
}
```

Tool Used

Manual Review

Recommendation

Only grant `BURN_ROLE` to `burnUser`. If admin privileges are needed, they should be granted to a separate, dedicated admin address.

Discussion

scirner22

Correcting this by adding a `BURN_ADMIN_ROLE` in this contract and `MINTER_ADMIN_ROLE` in CustomToken and then using `_setRoleAdmin(..)` to allow those custom admin roles to manage the specific role types. Both admins are set to `msg.sender` so the contract creator can manage the associated role over time.

defsec

Fix is confirmed on the https://github.com/ProvLabs/nuva-evm-contracts/commit/f4f53b2f1a2434cad5f9572c17ad6e6619eca34b.

Issue L-8: Missing Defensive Allowlist Check in TVV Calculation [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/56

Summary

The `GetTVVInUnderlyingAsset` function iterates over all token balances held by the `PrincipalMarkerAddress` to calculate the Total Vault Value. While current marker access controls prevent unauthorized deposits of unsupported tokens, adding an explicit allowlist check for the underlying asset and payment denom provides robust defense-in-depth against future code changes or potential bypasses in the marker permission system.

Vulnerability Detail

Currently, `GetTVVInUnderlyingAsset` retrieves all balances via `k.BankKeeper.GetAllBalances(ctx, vault.PrincipalMarkerAddress())`. It iterates through every token found and attempts to convert it to the underlying asset value using `ToUnderlyingAssetAmount`.

```
balances := k.BankKeeper.GetAllBalances(ctx, vault.PrincipalMarkerAddress())
    total := math.ZeroInt()
    for _, balance := range balances {
        if balance.Denom == vault.TotalShares.Denom {
            continue
        }
        val, err := k.ToUnderlyingAssetAmount(ctx, vault, balance)
        if err != nil {
            return math.Int{}, err
        }
```

Although the `PrincipalMarkerAddress` is a restricted marker account that blocks unauthorized transfers from external users, relying solely on this external restriction creates a brittle dependency. If a future upgrade relaxes marker restrictions, or if a privileged governance operation accidentally sends unsupported tokens to this address, the TVV calculation could fail (causing a DoS) or be manipulated.

Impact

There is no immediate exploit vector because the Provenance Marker module restricts transfers to this address. However, the lack of validation leaves the vault vulnerable to state corruption if the marker protections fail or are bypassed in the future.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/6d949a344913469816bbcd75e1e05c557364bf4b/vault/keeper/valuation_engine.go#L137-L147

Tool Used

Manual Review

Recommendation

Add the following check:

```
for _, balance := range balances {
        if balance.Denom == vault.TotalShares.Denom {
            continue
        }

        // FIX: Strict Allowlist.
        // Only calculate value for the Underlying Asset and the configured Payment
        ↪  Denom.
        if balance.Denom != vault.UnderlyingAsset && balance.Denom !=
        ↪  vault.PaymentDenom {
            continue
        }

        val, err := k.ToUnderlyingAssetAmount(ctx, vault, balance)
```

Alternatively, `IsAcceptedDenom` function can be used which has the same logic.

Discussion

nullpointer0x00

Adding to next milestone.

defsec

Fixed with https://github.com/ProvLabs/vault/pull/145.

Issue L-9: Payment Denom NAV staleness [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/58

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `UnitPriceFraction` function selects between forward and reverse NAV based solely on `UpdatedBlockHeight`, without considering NAV age or staleness. An attacker can profit from outdated NAV prices.

Vulnerability Detail

In `vault/keeper/valuation_engine.go:72-80`:

```
useForward := false
switch {
case fwd != nil && rev == nil:
    useForward = true
case fwd == nil && rev != nil:
    useForward = false
default:
    useForward = fwd.UpdatedBlockHeight >= rev.UpdatedBlockHeight  // Only checks
    ↪  which is newer
}
```

There's no check for:

1. Maximum NAV age (e.g., must be updated within last 100 blocks)
2. Price deviation limits

Impact

Stale oracle prices allow attackers to extract value.

Tool Used

Manual Review

Recommendation

Add NAV freshness validation.

Discussion

nullpointer0x00

Yes, this is a concern. That is why we have our `uylds.fcc` short circuit currently in our pricing engine. With our first vaults this code will not be executed. The NAV system is something we are working on with provenance on fixing these issues.

This is very helpful input as we think these things through.

Issue L-10: Contracts do not account for fee-on-transfer tokens [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/62

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `Depositor` and `Withdrawal` contracts assume that the amount of tokens transferred equals the amount specified in function parameters. However, some ERC20 tokens (known as "fee-on-transfer" tokens) deduct a fee during transfers, meaning the recipient receives less tokens than the amount specified.

Vulnerability Detail

Fee-on-transfer tokens deduct a percentage or fixed fee during `transfer()` or `transferFrom()` operations. Examples include:

- PAXG (Paxos Gold) - takes a fee on transfer
- Some USDT implementations on certain chains
- Various rebasing tokens

When transferring 100 tokens of a fee-on-transfer token with a 1% fee:

- Sender's balance decreases by: 100 tokens
- Recipient's balance increases by: 99 tokens
- Fee is deducted: 1 token

Impact

Tokens are lost in the accounting gap between specified amount and received amount.

Code Snippet

```
function _doDeposit(uint256 _amount, address _destinationAddress) private {
    if (_amount == 0) {
        revert InvalidAmount();
    }
    if (_destinationAddress == address(0)) {
        revert InvalidAddress("destination");
    }
    // @audit Assumes _amount tokens are received
    depositToken.safeTransferFrom(msg.sender, _destinationAddress, _amount);

    // @audit Event shows incorrect amount if fee-on-transfer
    emit Deposit(msg.sender, _amount, address(depositToken), shareToken,
     ↪  _destinationAddress);
}
```

Tool Used

Manual Review

Recommendation

Modify functions to measure the actual balance before and after transfer.

Discussion

scirner22

Are we good with taking this forward as a long term issue? Our v1 deployment will be Ethereum and the only tokens across deposit/withdraw will be our CustomTokens and Circle's USDC. It seems like Circle does not implement such a fee in kind.

Issue L-11: `AutoCLI` positional arguments order mismatch with proto field definitions [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/63

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The AutoCLI configuration in `vault/module.go` has positional argument orders that do not match the proto field definitions for `CreateVault` and `SetShareDenomMetadata` commands. While AutoCLI uses `ProtoField` names to map arguments correctly, the order mismatch creates user confusion and documentation inconsistency.

Vulnerability Detail

Issue 1: CreateVault - Field Order Mismatch

Proto field order: `admin` (1), `share_denom` (2), `underlying_asset` (3) CLI positional order: `admin`, `underlying_asset`, `share_denom`

The CLI places `underlying_asset` (proto field 3) before `share_denom` (proto field 2).

Issue 2: SetShareDenomMetadata - Field Order Mismatch

Proto field order: `metadata` (1), `admin` (2), `vault_address` (3) CLI positional order: `admin`, `vault_address`, `metadata`

The CLI completely reverses the proto field order, placing `metadata` last when it's defined first in the proto.

Code Snippet

```
// vault/module.go:168-172 - CreateVault
{
    RpcMethod: "CreateVault",
    Use:       "create [admin] [underlying_asset] [share_denom]",  //  Wrong order
    PositionalArgs: []*autocliv1.PositionalArgDescriptor{
        {ProtoField: "admin"},              // Field 1
        {ProtoField: "underlying_asset"},   // Field 3  (should be field 2)
        {ProtoField: "share_denom"},        // Field 2  (should be field 3)
    },
}

// vault/module.go:191-195 - SetShareDenomMetadata
{
    RpcMethod: "SetShareDenomMetadata",
    Use:       "set-share-denom-metadata [admin] [vault_address] [metadata]",  //
     ↪  Wrong order
    PositionalArgs: []*autocliv1.PositionalArgDescriptor{
        {ProtoField: "admin"},              // Field 2 (should be field 1)
        {ProtoField: "vault_address"},      // Field 3 (should be field 2)
        {ProtoField: "metadata"},           // Field 1  (should be field 3)
    },
}
```

Tool Used

Manual Review

Recommendation

Match proto field order for consistency.

Discussion

nullpointer0x00

Good point! I have been meaning to better understand the autocli. I was so used to years of just writing them by hand. Thanks for this input. Will fix next release.

Issue L-12: Zero assets with non-zero shares allows share inflation [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/65

Summary

The `CalculateSharesProRataFraction` function uses a special first-deposit formula when `totalAssets` is zero, but does not verify that `totalShares` is also zero. If `totalAssets` is zero while `totalShares` is non-zero, the function mints shares using the first-deposit formula that ignores existing shares, causing massive share supply inflation and complete dilution of existing shareholders.

Vulnerability Detail

The function `CalculateSharesProRataFraction` in `vault/utils/shares.go` contains a branch that handles the first deposit case:

```
if totalAssets.IsZero() {
    shares := amountNumerator.Mul(ShareScalar).Quo(amountDenominator)
    return sdk.NewCoin(shareDenom, shares), nil
}
```

This branch assumes that zero assets means it is the first deposit. However, it does not check whether `totalShares` is also zero. The function proceeds to mint shares using the formula `amount * ShareScalar / denominator`, which completely ignores any existing shares.

The state where `totalAssets = 0` but `totalShares > 0` can occur due to floor division in the redemption calculation. When `CalculateRedeemProRataFraction` computes payout amounts, it uses integer floor division:

```
out := num.Quo(den)  // Floor division
```

Impact

When a deposit occurs with zero assets but non-zero shares, the share supply inflates by orders of magnitude.

Code Snippet

CalculateSharesProRataFraction (`vault/utils/shares.go`):

```
if totalAssets.IsZero() {
    // Assumes first deposit without checking totalShares
    shares := amountNumerator.Mul(ShareScalar).Quo(amountDenominator)
    return sdk.NewCoin(shareDenom, shares), nil
}
// Normal pro-rata calculation
ta := totalAssets.Add(VirtualAssets)
ts := totalShares.Add(VirtualShares)
den := amountDenominator.Mul(ta)
shares := amountNumerator.Mul(ts).Quo(den)
return sdk.NewCoin(shareDenom, shares), nil
```

Tool Used

Manual Review

Recommendation

Add validation to ensure both assets and shares are zero before using the first-deposit formula.

Discussion

nullpointer0x00

Will add this fix next release.

defsec

Fixed with https://github.com/ProvLabs/vault/pull/149/files.

Issue L-13: Incomplete zero checks in `UnitPriceFraction` [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/66

Summary

The `UnitPriceFraction` function checks for zero volume in forward NAV and zero price in reverse NAV, but does not check for zero price in forward NAV or zero volume in reverse NAV. This creates inconsistent validation and allows invalid NAV data to be used in calculations, resulting in zero conversion values without error indication.

Vulnerability Detail

The function `UnitPriceFraction` in `vault/keeper/valuation_engine.go` handles two NAV directions with asymmetric validation:

Forward NAV Path :

```
if useForward {
    if fwd.Volume == 0 {
        return math.Int{}, math.Int{}, fmt.Errorf("nav volume is zero for %s/%s",
         ↪  srcDenom, underlyingAsset)
    }
    priceAmt := fwd.Price.Amount  // No check if zero
    volAmt := math.NewIntFromUint64(fwd.Volume)
    return priceAmt, volAmt, nil
}
```

Reverse NAV Path:

```
if rev.Price.Amount.IsZero() {
    return math.Int{}, math.Int{}, fmt.Errorf("nav price is zero for %s/%s",
     ↪  underlyingAsset, srcDenom)
}
priceAmt := math.NewIntFromUint64(rev.Volume)  // No check if zero
volAmt := rev.Price.Amount
return priceAmt, volAmt, nil
```

The function validates:

- Forward NAV: checks `fwd.Volume == 0` but not `fwd.Price.Amount.IsZero()`
- Reverse NAV: checks `rev.Price.Amount.IsZero()` but not `rev.Volume == 0`

When these unchecked zero values are returned, they are used in `ToUnderlyingAssetAmount`:

```
func (k Keeper) ToUnderlyingAssetAmount(ctx sdk.Context, vault types.VaultAccount,
 ↪  in sdk.Coin) (math.Int, error) {
    priceAmount, volume, err := k.UnitPriceFraction(ctx, in.Denom, vault)
    if err != nil {
        return math.Int{}, err
    }
    return in.Amount.Mul(priceAmount).Quo(volume), nil
}
```

Impact

1. **Forward NAV with Zero Price:** If `fwd.Price.Amount` is zero, the function returns `(0, volume)`. The calculation `amount * 0 / volume = 0` produces zero results for all conversions, indicating invalid NAV data without error.
2. **Reverse NAV with Zero Volume:** If `rev.Volume` is zero, the function returns `(0, priceAmount)`. The calculation `amount * 0 / priceAmount = 0` produces zero results for all conversions, indicating invalid NAV data without error.

Code Snippet

UnitPriceFraction (`vault/keeper/valuation_engine.go`):

```
if useForward {
    if fwd.Volume == 0 {
        return math.Int{}, math.Int{}, fmt.Errorf("nav volume is zero for %s/%s",
         ↪  srcDenom, underlyingAsset)
    }
    // Missing check: fwd.Price.Amount.IsZero()
    priceAmt := fwd.Price.Amount
    volAmt := math.NewIntFromUint64(fwd.Volume)
    return priceAmt, volAmt, nil
}

if rev.Price.Amount.IsZero() {
    return math.Int{}, math.Int{}, fmt.Errorf("nav price is zero for %s/%s",
     ↪  underlyingAsset, srcDenom)
}
// Missing check: rev.Volume == 0
priceAmt := math.NewIntFromUint64(rev.Volume)
volAmt := rev.Price.Amount
return priceAmt, volAmt, nil
```

ToUnderlyingAssetAmount (`vault/keeper/valuation_engine.go`):

```
func (k Keeper) ToUnderlyingAssetAmount(ctx sdk.Context, vault types.VaultAccount,
↪  in sdk.Coin) (math.Int, error) {
    priceAmount, volume, err := k.UnitPriceFraction(ctx, in.Denom, vault)
    if err != nil {
        return math.Int{}, err
    }
    return in.Amount.Mul(priceAmount).Quo(volume), nil
}
```

Tool Used

Manual Review

Recommendation

Add comprehensive zero checks to both NAV paths to ensure consistent validation.

Discussion

nullpointer0x00

I will add this validation in the future. Currently, the `marker` module does a validate on those values and neither can be zero when adding it to the store.

This validate is called before any NAV is saved to the store.

```
// Validate returns error if NetAssetValue is not in a valid state
func (mnav *NetAssetValue) Validate() error {
    if err := mnav.Price.Validate(); err != nil {
        return err
    }

    if mnav.Price.Amount.GT(sdkmath.NewInt(0)) && mnav.Volume < 1 {
        return fmt.Errorf("marker net asset value volume must be positive value")
    }

    return nil
}
```

Therefore:

```
fwd, errF := k.MarkerKeeper.GetNetAssetValue(ctx, srcDenom, underlyingAsset)
    rev, errR := k.MarkerKeeper.GetNetAssetValue(ctx, underlyingAsset, srcDenom)
```

would never return zeros. However, it is better to have our own validations for possible future changes.

defsec

Fixed with https://github.com/ProvLabs/vault/pull/155.

Issue L-14: Lack of atomicity in withdrawal processing leads to potential double spending. [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/69

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `processSwapOutJobs` function processes withdrawals in a loop. Inside the loop, `processSingleWithdrawal` performs two critical state changes: sending assets to the user and burning the user's shares. These operations are not atomic. If the asset transfer succeeds but the share burning fails (returning a critical error), the vault is paused, but the asset transfer is not reverted. This leaves the system in an inconsistent state where assets have left the vault without the corresponding liability (shares) being extinguished.

Vulnerability Detail

In `processSingleWithdrawal`, the code executes:

`k.BankKeeper.SendCoins(..., assets)`: Transfers underlying assets to the user. `k.BankKeeper.SendCoins(..., shares)`: Transfers shares to the marker account. `k.MarkerKeeper.BurnCoin(...)`: Burns the shares. If step 3(or step 2) fails, a CriticalError is returned. In `processSwapOutJobs`, this error is caught. The code calls `autoPauseVault` and then executes continue. This allows the loop to proceed or finish, committing the `sdkCtx` state changes (the asset transfer) to the blockchain.

Impact

The vault is paused, but the funds are lost. If an admin later refunds the shares to fix the failed withdrawal, the user ends up with both the assets and the shares. Additionally, if vault is unpaused without manually burning shares that didn't complete, accounting(TVV calculations) will be flawed.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/6d949a344913469816bbcd75e1e05c557364bf4b/vault/keeper/payout.go#L137-L157

Tool Used

Manual Review

Recommendation

Use `sdkCtx.CacheContext()` to wrap the execution of `processSingleWithdrawal`. This ensures that if any part of the withdrawal fails, the entire transaction (including the asset transfer) is rolled back. Only commit the cached context if the function returns nil.

Discussion

nullpointer0x00

Great suggestion on this one.

We were aware that an autopause here would leave the state messy and require manual admin intervention to fix. However, your point about using CacheContext is a much better approach. It handles the rollback automatically so we don't have to worry about the state becoming inconsistent.

I am going to test the CacheContext implementation and add it to our milestone.

Issue L-15: `ValidateInterestRateLimits` bypasses validation when only one limit is set [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/74

Summary

The `ValidateInterestRateLimits` function in `vault/keeper/vault.go` returns early if either `minRateStr` or `maxRateStr` is empty, preventing validation of the non-empty parameter.

Vulnerability Detail

The function uses an OR condition (||) in the early return check:

```
func (k Keeper) ValidateInterestRateLimits(minRateStr, maxRateStr string) error {
    if minRateStr == "" || maxRateStr == "" {  // @audit Returns if EITHER is empty
        return nil
    }

    minRate, err := sdkmath.LegacyNewDecFromStr(minRateStr)
    if err != nil {
        return fmt.Errorf("invalid min interest rate: %w", err)
    }
    maxRate, err := sdkmath.LegacyNewDecFromStr(maxRateStr)
    if err != nil {
        return fmt.Errorf("invalid max interest rate: %w", err)
    }

    if minRate.GT(maxRate) {
        return fmt.Errorf("minimum interest rate %s cannot be greater than maximum
         ↪  interest rate %s", minRate, maxRate)
    }

    return nil
}
```

Problem Cases:

1. Only minimum rate set:

```
ValidateInterestRateLimits("invalid-decimal", "")
// Returns nil without validating "invalid-decimal"
```

2. Only maximum rate set:

```
ValidateInterestRateLimits("", "not-a-number")
// Returns nil without validating "not-a-number"
```

3. Both set with invalid values: `ValidateInterestRateLimits("abc", "xyz")`

Impact

Malformed decimal strings can be set as `MinInterestRate` or `MaxInterestRate`.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/6d949a344913469816bbcd75e1e05c557364bf4b/vault/keeper/vault.go#L320

Tool Used

Manual Review

Recommendation

Validate each rate string independently before checking the `min < max` relationship.

Discussion

nullpointer0x00

This one is a good find and an easy fix. Will add it to our next release.

defsec

Fixed with https://github.com/ProvLabs/vault/pull/141/files

Issue L-16: Query endpoints ignore error returns from `GetVault` [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/75

Summary

The `Vaults` query endpoint in `query_server.go` ignores the error return from `GetVault`, allowing invalid or corrupted vault data to be included in query responses.

Vulnerability Detail

```go
func (k queryServer) Vaults(goCtx context.Context, req *types.QueryVaultsRequest)
 ↪  (*types.QueryVaultsResponse, error) {
    // ...
    _, pageRes, err := query.CollectionFilteredPaginate(
        ctx,
        k.Keeper.Vaults,
        req.Pagination,
        func(key sdk.AccAddress, _ []byte) (include bool, err error) {
            vault, _ := k.GetVault(ctx, key)  // @audit Error ignored
            vaults = append(vaults, *vault)
            return true, nil
        },
        // ...
    )
}
```

The error from `GetVault` is explicitly ignored with _.

Impact

Partial vault data from failed validation might be exposed.

Code Snippet

```go
vault, _ := k.GetVault(ctx, key)  // Error ignored
vaults = append(vaults, *vault)   // Dereferences potentially nil vault
```

Tool Used

Manual Review

Recommendation

Consider adding error validation.

Discussion

nullpointer0x00

Adding to next release. Thanks!

Issue L-17: No user-initiated cancellation for pending withdrawals [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/81

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Users cannot cancel pending withdrawal requests leading to forced exposure during cooldown period.

Vulnerability Detail

The vault module provides no mechanism for users to cancel their pending withdrawal requests after calling `SwapOut`. Users are forced to wait for the cooldown period with their shares escrowed, exposing them to unwanted NAV fluctuations and vault state changes.

Current Flow:

1. User calls `SwapOut`.
2. Shares immediately escrowed from user to vault address.
3. Request queued with `payoutTime = now + WithdrawalDelaySeconds`.
4. User must wait for cooldown (typically days).
5. No way to cancel and get shares back.

Impact

User locked in during cooldown.

Code Snippet

```go
func (k *Keeper) SwapOut(ctx sdk.Context, vaultAddr, owner sdk.AccAddress, shares
↪  sdk.Coin, redeemDenom string) (uint64, error) {
    // ... validation ...

    // Escrow shares immediately
    if err := k.BankKeeper.SendCoins(ctx, owner, vault.GetAddress(),
    ↪  sdk.NewCoins(shares)); err != nil {
        return 0, fmt.Errorf("failed to escrow shares: %w", err)
    }
    payoutTime := ctx.BlockTime().Unix() + int64(vault.WithdrawalDelaySeconds)
    pendingReq := types.PendingSwapOut{
        Owner:        owner.String(),
        VaultAddress: vaultAddr.String(),
        RedeemDenom:  redeemDenom,
        Shares:       shares,
    }
    requestID, err := k.PendingSwapOutQueue.Enqueue(ctx, payoutTime, &pendingReq)
    // ...
    return requestID, nil
}
```

Tool Used

Manual Review

Recommendation

Consider adding user-initiated cancellation.

Discussion

nullpointer0x00

We have discussed this as a future feature and all the implications.

Issue L-18: Missing Cosmos SDK invariants [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/82

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The vault module does not implement `RegisterInvariants`, which is a standard Cosmos SDK pattern for modules to register invariant checks that can be run by the crisis module. This means there are no automated checks to detect state inconsistencies, accounting errors, or protocol violations that could occur due to bugs, attacks, or edge cases.

Vulnerability Detail

Cosmos SDK modules typically implement the `HasInvariants` interface and register invariant checks via `RegisterInvariants`. These invariants are periodically checked by the crisis module to detect state corruption or protocol violations. The vault module does not implement this pattern.

The `AppModule` in `vault/module.go` does not implement `RegisterInvariants`:

```
// vault/module.go
type AppModule struct {
    AppModuleBasic
    keeper        *keeper.Keeper
    addressCodec address.Codec
    markerKeeper types.MarkerKeeper
    bankKeeper   types.BankKeeper
}

// @audit No RegisterInvariants method
// @audit Does not implement module.HasInvariants
```

Impact

1. **Share Supply Consistency:**
    - `vault.TotalShares` should match or exceed the marker's actual supply
    - Marker supply should not exceed `TotalShares` (bridge capacity check)
    - Escrowed shares in vault account should be accounted for
2. **Share Backing Invariant:**

  - Total Vault Value (TVV) should be sufficient to back all outstanding shares
  - Principal marker balances should match or exceed the value represented by shares
  - NAV per share should be positive and consistent

3. **Queue Consistency:**
   - All entries in `PendingSwapOutQueue` should reference valid vaults
   - Escrowed shares in vault account should match pending withdrawal requests
   - Queue timestamps should be valid and ordered
4. **Interest Rate Bounds:**
   - `CurrentInterestRate` should respect `MinInterestRate` and `MaxInterestRate` bounds
   - Interest rate strings should be valid decimal numbers
   - Rate changes should be within configured limits

Code Snippet

Tool Used

Manual Review

Recommendation

Consider implementing invariants on the system.

Discussion

nullpointer0x00

We will add in follow up release. Thanks for this one!

Issue L-19: MaxSwapOutBatchSize should be governance parameter [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/85

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `MaxSwapOutBatchSize` constant is hardcoded to 100 in the vault module's code. This prevents the protocol from adjusting the batch processing limit without a code upgrade, making it difficult to respond to network conditions, queue sizes, or discovered vulnerabilities. The value should be configurable via governance parameters to allow dynamic adjustment.

Vulnerability Detail

The `MaxSwapOutBatchSize` constant is defined as a hardcoded value in `vault/keeper/abci.go`:

```
const (
    // MaxSwapOutBatchSize is the maximum number of pending swap-out requests
    // to process in a single EndBlocker invocation. This prevents a large queue
    // from consuming excessive block time and memory. This is a temporary value
    // and we will need to do more analysis on a proper batch size.
    // See https://github.com/ProvLabs/vault/issues/75.
    MaxSwapOutBatchSize = 100
)
```

This constant is used in the `EndBlocker` to limit how many withdrawal requests are processed per block:

```
func (k *Keeper) EndBlocker(ctx context.Context) error {
    if err := k.processPendingSwapOuts(ctx, MaxSwapOutBatchSize); err != nil {
        return err
    }
    // ...
}
```

Impact

The hardcoded batch size limits the protocol's ability to adapt to changing conditions.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/main/vault/keeper/abci.go#L13

Tool Used

Manual Review

Recommendation

Implement `MaxSwapOutBatchSize` as a governance parameter.

Discussion

nullpointer0x00

This was something we were considering in the next iteration. Good call out.

Issue L-20: Systematic yield loss due to interest truncation on short intervals. [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/86

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The vault interest calculation logic truncates the accrued interest to the nearest integer at every reconciliation step. If reconciliation occurs frequently (e.g., every few seconds due to high user activity) or if the principal/rate combination results in less than 1 unit of interest per interval, the vault generates zero yield. While this can be exploited by an attacker performing a "Dusting Attack," the primary risk is that a naturally active vault will systematically underpay interest, effectively erasing the vault's obligations to shareholders.

Vulnerability Detail

The core issue lies in the `CalculateInterestEarned` function, which computes interest using high-precision decimals but returns the result using `interestAmountDec.TruncateInt()`. This truncation discards any fractional interest (dust) less than 1 unit.

When `reconcileVaultInterest` is triggered, which happens automatically during public operations like `SwapIn` and `SwapOut`, it calculates interest for the elapsed time since the last update and then resets the `PeriodStart` timestamp. If the elapsed time is short (e.g., ~6 seconds), the accrued interest is often less than 1 integer unit (e.g., 0.95 units). The truncation logic rounds this down to 0. The system then advances the `PeriodStart` timestamp, effectively marking that time period as "paid" despite 0 interest being distributed. If a vault experiences frequent interaction, this truncation happens repeatedly. The fractional interest that should have accumulated over time is instead discarded at every step, resulting in a total yield of 0 over long durations.

This behavior is confirmed by the `TestCumulativeTruncationLoss` test case given below (running it in interest_test.go). The test simulates interest calculation over 10,000 blocks (6 seconds each) for a principal of 100,000,000 units at 5% APY. When calculated iteratively to simulate per-block updates, the total interest returned is 0. However, when calculated in bulk to simulate a single update over the same total duration, the result is 9513 units of interest.

```go
func TestCumulativeTruncationLoss(t *testing.T) {
    denom := "uatom"
    // Principal: 100 tokens (100,000,000 units)
    principal := sdk.NewCoin(denom, sdkmath.NewInt(100_000_000))
    rate := "0.05"        // 5% APY
    blockTime := int64(6) // 6 seconds per block
    iterations := 10000   // Simulate 10,000 blocks (approx 16 hours)

    // 1. Calculate iteratively (simulating per-block reconciliation)
    // This mimics the vault state updating every block, truncating fractions each
    ↪  time.
    sumIterative := sdkmath.ZeroInt()
    for i := 0; i < iterations; i++ {
        interestAmt, err := interest.CalculateInterestEarned(principal, rate,
        ↪  blockTime)
        require.NoError(t, err)
        sumIterative = sumIterative.Add(interestAmt)
    }

    // 2. Calculate bulk (simulating a single reconciliation after the total time)
    // This represents the mathematical ideal if precision were preserved.
    totalTime := blockTime * int64(iterations)
    sumBulk, err := interest.CalculateInterestEarned(principal, rate, totalTime)
    require.NoError(t, err)

    t.Logf("Principal: %s", principal)
    t.Logf("Rate: %s", rate)
    t.Logf("Total Duration: %d seconds", totalTime)
    t.Logf("Iterative Sum (Actual Yield):   %s", sumIterative)
    t.Logf("Bulk Sum (Expected Yield):      %s", sumBulk)

    // Assertion: The bulk calculation should yield significantly more than the
    ↪  iterative one.
    // In this specific scenario, the iterative sum will be exactly 0 due to
    ↪  truncation.
    require.True(t, sumBulk.GT(sumIterative), "Iterative calculation lost yield due
    ↪  to truncation")
}
```

Impact

Shareholders lose 100% of the yield if the vault is highly active. The vault's internal accounting deviates significantly from the expected compounding curve, leading to a permanent loss of value for depositors.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/6d949a344913469816bbcd75e1e05c557364bf4b/vault/interest/interest.go#L36-L66

Tool Used

Manual Review

Recommendation

Implement a dust accumulator in the vault state to track fractional interest that would otherwise be lost during truncation. When calculating interest for a period, store the full high-precision decimal result. Separate the integer portion for the immediate payout and add the remaining fractional part to a persistent `InterestDust` field in the `VaultAccount`. In subsequent reconciliations, add this stored dust to the newly calculated interest before performing the truncation step. This ensures that small fractional accruals eventually sum up to whole units and are paid out rather than being discarded.

Discussion

nullpointer0x00

This is a good point. I will mark this as something to implement for next release.

nullpointer0x00

After discussing the with the team, the test case does confirmed that cumulative truncation of fractional interest can technically lead to a 100% loss of yield for highly active vaults, for our current vault use cases and token precision ($10^-6 YLDS = USD) show the actual monetary loss is too minimal (approx $0.00000095 per block) to warrant implementing a dust accumulator for this next release.

Issue L-21: `MustAccAddressFromBech32` usage on state data [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/91

Summary

The vault module uses `MustAccAddressFromBech32` to parse bech32 addresses from state data in critical execution paths, particularly in `processSingleWithdrawal` and `refundWithdrawal`.

Vulnerability Detail

The `MustAccAddressFromBech32` function is designed to be used only when the input is guaranteed to be valid (e.g., from validated user input or constants).

These functions are called in critical paths:

- `processSingleWithdrawal` is called from EndBlocker to process withdrawals.
- `refundWithdrawal` is called when withdrawals fail.
- `ExpeditePendingSwapOut` is called by admins to expedite withdrawals.

Impact

`MustAccAddressFromBech32` can panic whenever there is an error.

Code Snippet

processSingleWithdrawal (vault/keeper/payout.go):

```
func (k *Keeper) processSingleWithdrawal(ctx sdk.Context, id uint64, req
↪   types.PendingSwapOut, vault types.VaultAccount) error {
    vaultAddr := sdk.MustAccAddressFromBech32(req.VaultAddress)
    ownerAddr := sdk.MustAccAddressFromBech32(req.Owner)
}
```

refundWithdrawal (vault/keeper/payout.go):

```
func (k *Keeper) refundWithdrawal(ctx sdk.Context, id uint64, req
↪   types.PendingSwapOut, reason string) error {
    vaultAddr := sdk.MustAccAddressFromBech32(req.VaultAddress)
    ownerAddr := sdk.MustAccAddressFromBech32(req.Owner)
}
```

ExpeditePendingSwapOut (vault/keeper/msg_server.go:436):

```
vaultAddr := sdk.MustAccAddressFromBech32(swapOut.VaultAddress)
// @audit No error handling if address is invalid
```

Tool Used

Manual Review

Recommendation

Replace `MustAccAddressFromBech32` with `AccAddressFromBech32` and add proper error handling.

Discussion

nullpointer0x00

This is good input and easy fix. Currently, I believe all paths to these have already validated the address at this point, so they shouldn't panic. However, I will fix this next release.

defsec

Fixed with https://github.com/ProvLabs/vault/pull/152

Issue L-22: Lack of decimal normalization in cross-chain transfers [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/92

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The contract transfers raw token amounts between the user and the vault during cross-chain transfers without performing any decimal normalization. If the external chain uses a different decimal precision, such as 18 decimals USDC on BNB Chain versus 6 decimals on Provenance, relying on raw amounts will lead to significant value discrepancies unless handled entirely by off-chain relayers.

Vulnerability Detail

The `swap_in` function accepts a coin amount and forwards it directly to the vault, while `burn_swap_in_receipt` burns a specific raw amount of shares to trigger a cross-chain release. On chains where the decimal precision of USDC differs from Provenance Blockchain, such as those using 18 decimals versus the standard 6, relying on raw amounts causes value mismatches. For example, transferring 1 unit from an 18-decimal chain could be interpreted as 1 trillion units on a 6-decimal chain if the raw integer value is preserved without scaling.

Impact

This relies on the correctness of off-chain infrastructure. Failure to normalize decimals in the relayer or destination contract will result in massive inflation or loss of funds.

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/6d949a344913469816bbcd75e1e05c557364bf4b/vault/keeper/vault.go#L195-L223

Tool Used

Manual Review

Recommendation

Ensure that the off-chain infrastructure or the smart contracts on the counterparty chain strictly enforce decimal normalization to maintain value parity during cross-chain transfers.

Discussion

scirner22

Acknowledged. The off chain manager currently normalizes all coins to 18 decimals internally and converts them back to the per chain decimal representation.

As added protection I'll brainstorm some contract layer checks on top of that. For instance maybe the deposit and withdraw evm functions can take the decimal value and it can be checked against the erc20 decimal. That would prove the caller is explicitly stating the correct decimal precision, although doesn't strictly mean the value they are sending in is representative of the decimal value.

Issue L-23: `InitGenesis` calls `SetVaultLookup` twice for same vault [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/issues/93

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `InitGenesis` function calls `SetVaultLookup` twice for the same vault address: once when processing vaults from `GetAllAccounts` and again when processing vaults from `genState.Vaults`. While this doesn't cause an error, it's inefficient and could mask inconsistencies if the vault data differs between the two sources.

Vulnerability Detail

The `InitGenesis` function processes vaults in two separate loops. The first loop iterates through `GetAllAccounts` and calls `SetVaultLookup` for each vault account found. The second loop iterates through `genState.Vaults` and also calls `SetVaultLookup` for each vault.

If a vault exists in both `GetAllAccounts` (from x/auth state) and `genState.Vaults` (from genesis JSON), `SetVaultLookup` will be called twice for the same vault address. While this doesn't cause an error since it just overwrites the same empty byte array, it's inefficient.

The `SetVaultLookup` function doesn't validate that the vault being stored matches any existing vault at that address. If there's a mismatch between the vault in `GetAllAccounts` and the vault in `genState.Vaults` (e.g., different share denom, admin, or other fields), the second call would silently overwrite the lookup entry without detecting the inconsistency.

Impact

Duplicate `SetVaultLookup` calls .

Code Snippet

https://github.com/sherlock-audit/2025-11-provlabs-nu-blockchain-nov-20th/blob/main/vault/keeper/genesis.go#L27

Tool Used

Manual Review

Recommendation

Skip calling `SetVaultLookup` in the second loop if the vault already exists in the lookup.

Discussion

nullpointer0x00

Adding this to next milestone.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.