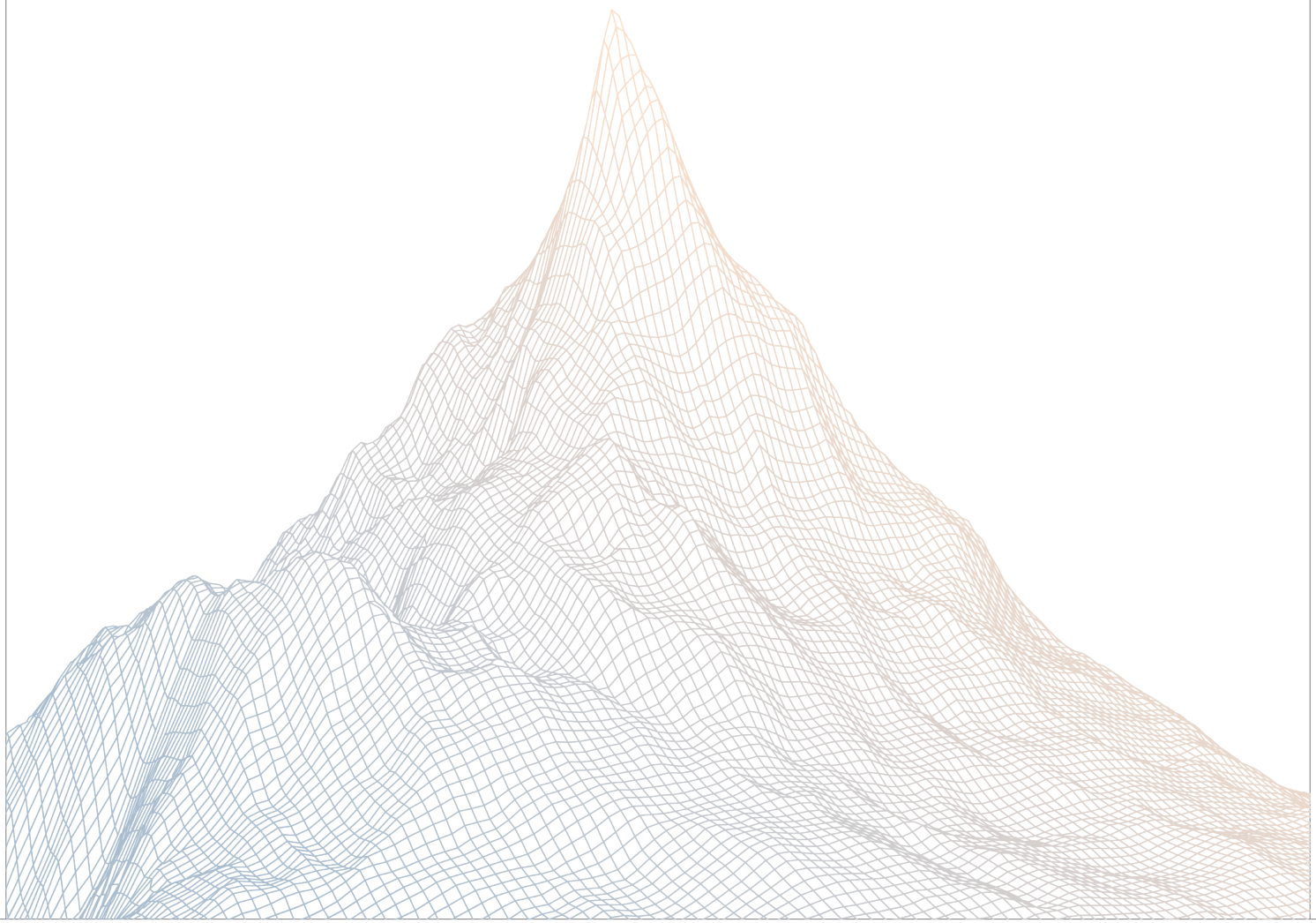
Zenith

Solv

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: September 12th to September 18th, 2025

AUDITED BY: Christian Vari
oakcobalt

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Solv

Solv Protocol is building the $1 trillion Bitcoin economy through a full-stack suite of financial services optimized for BTC holders. By enabling lending, liquid staking, yield generation, and fund management, Solv activates Bitcoin as a capital-efficient asset within a growing Bitcoin-native DeFi ecosystem. Supported by institutional partners and an on-chain BTC reserve, Solv bridges TradFi, CeFi, and DeFi, unlocking yield opportunities and expanding Bitcoin's role in programmable finance.

2.2 Scope

The engagement involved a review of the following targets:

Target	SolvBTC-Stellar-Contract
Repository	https://github.com/57blocks/SolvBTC-Stellar-Contract
Commit Hash	`369fdd37f4bc59d3459542058829469efae160de`

Files	`fungible-token/**.rs`
	`oracle/**.rs`
	`vault/**.rs`

Target	Solv Mitigation Review
Repository	https://github.com/57blocks/SolvBTC-Stellar-Contract
Commit Hash	`12738e6f8665c62023a5c81a2a2f4770591ed0de`

Files	`fungible-token/**.rs`
	`oracle/**.rs`
	`vault/**.rs`

2.3 Audit Timeline

September 12, 2025	Audit start
September 18, 2025	Audit end
October 10, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	2
High Risk	2
Medium Risk	5
Low Risk	5
Informational	2
Total Issues	**16**

3

Findings Summary

ID	Description	Status
C-1	withdraw_request uses incorrect burn method	Resolved
C-2	Withdrawal arithmetic overflow causes redemption Denial of Service	Resolved
H-1	create_withdraw_message doesn't confirm to EIP712, and may break signature validations	Resolved
H-2	User can bypass blacklisting through withdraw from vault	Resolved
M-1	calculate_domain_separator has incorrect type hash	Resolved
M-2	Blacklisted user can burn other token holders funds through allowance	Resolved
M-3	Absence of NAV freshness checks	Resolved
M-4	Overflow in deposit minting formula enables Denial of Service	Resolved
M-5	Instance storage TTLs are not extended, risks of data expiration and potential DOS	Acknowledged
L-1	Mutable withdraw parameters could prevent the correct functioning of the contract	Resolved
L-2	Unvalidated verifier keys	Resolved
L-3	Missing validation of deposit_fee_ratio in constructor	Resolved
L-4	NAV manager can bypass the rate increase limit	Resolved
L-5	Missing FX conversion in multi-currency deposits causes systematic mispricing	Acknowledged
I-1	Inconsistent event data layout in set_vault_by_admin	Resolved
I-2	Unused Datakey UsedRequestHash and WithdrawRequestStatus	Resolved

4

Findings

4.1 Critical Risk

A total of 2 critical risk findings were identified.

[C-1] withdraw_request uses incorrect burn method

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- vault.rs#L327

Description:

When a user calls withdraw_request, their share(fungible_token.rs) needs to be burned. The withdraw_request function uses the incorrect burn method that requires caller authorization, causing all withdrawal requests to fail and permanently locking user funds.

```
//vault/src/vault.rs

    fn withdraw_request(env: Env, from: Address, shares: i128,
    request_hash: Bytes) {
        from.require_auth(); // Verify caller identity
...
        // Burn user's shares directly via token burn
        token_client.burn(&from, &shares);
```

The fungible token's burn implementation calls `Base::burn(env, &from, amount)` which enforces that from must be the caller.

```
//fungible-token/src/fungible_token.rs

    fn burn(env: &Env, from: Address, amount: i128) {
        // SEP-41 standard: user can burn their own tokens
        Base::burn(env, &from, amount);
    }
```

In the vault's context, when `token_client.burn(&from, &shares)` is executed, the actual

caller is the vault contract itself (env.current_contract_address()), not the user. This fails token_client authentication, causing tx revert.

Note: current unit integration tests have `env.mock_all_auths()` enabled, which bypasses authorization checks.

Impact: High - This causes DOS of withdrawal functionality and also locks deposited user funds.

Likelihood: High - This vulnerability will occur on every single withdraw_request call.

Recommendations:

Consider using `burn_from` instead of `burn`.

Solv Finance: Resolved in `5d0584fb18958188c22fb93960321d4e68200a99`

Zenith: Verified. Added `withdraw_request_with_allowance` which solves account contract/regular contract edge cases.

[C-2] Withdrawal arithmetic overflow causes redemption Denial of Service

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- vault/src/vault.rs#L1120-L1132

Description:

The `calculate_withdraw_amount` function calculates the numerator as `shares * nav * 10^withdraw_decimals` using plain `i128` arithmetic. Since the workspace configured for `overflow-checks = true` and `panic = "abort"` in release builds, any overflow causes the transaction to panic and revert. This makes certain withdrawals impossible under common parameter settings.

With the repository's typical configuration (`shares_decimals = 18`, `withdraw_decimals = 8`, `nav_decimals = 8`, and `nav ≈ 1e8` for `NAV = 1.0`), overflow occurs when shares exceed roughly `1.7e22` equivalent to about `17,014` WBTC worth of minted shares. As NAV increases, this threshold decreases linearly (e.g., at `NAV = 10.0`, the threshold is `~1,701` WBTC).

If the withdrawal currency is configured with `18` decimals, the overflow threshold collapses to approximately `1.7e12` shares, well below the `~1e18` shares minted from a single `1` WBTC deposit. In this configuration, nearly any withdrawal, and even withdrawal previews (`withdraw_request`), would panic and revert, effectively blocking redemptions. Separately, `10^x` itself can overflow for decimal values above `38`, introducing another source of reverts.

Recommendations:

We recommend bounding decimal values at configuration time to prevent unsafe exponentiation (e.g., restricting decimals to ≤ `18` individually and ≤ `38` in aggregate) as well as rearranging the formula.

Solv Finance: Resolved in `87f45abd9198663955e896e3e736a5de9a3c7cea`

Zenith: Verified

4.2 High Risk

A total of 2 high risk findings were identified.

[H-1] create_withdraw_message doesn't confirm to EIP712, and may break signature validations

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- vault.rs#L829-L843

Description:

The `create_withdraw_message` function does not conform to EIP712 standards, causing signature validation issues and breaking compatibility with the protocol client.

The `create_withdraw_message` function has multiple EIP712 compliance issues that can break the signature verification process:

1. Missing TypeHash (EIP712 Requirement):

The `withdraw` type hash is missing here.

```
fn create_withdraw_message(
    env: &Env,
    _user: &Address,
    target_amount: i128,
    target_token: &Address,
    nav: i128,
    request_hash: &Bytes,
) -> Bytes {
    let mut encoded = Bytes::new(env);
//@audit Missing withdraw type hash
```

2. Dynamic Types Not Hashed:

Both `action_bytes` and `request_hash` are dynamic bytes and should be hashed.

```
        // Add action (fixed as "withdraw")
        let action_bytes = Bytes::from_slice(env, b"withdraw");
       //@audit dynamic bytes/strings not hashed before appended
        encoded.append(&action_bytes);
...
        encoded.append(request_hash); //@audit Should be hashed for dynamic
    types
```

3. Integer Padding Issues:

`i128` value is not padded to 32 bytes.

```
// Add target amount (shares)
encoded.append(&Bytes::from_array(env, &target_amount.to_be_bytes()));
```

4. Inconsistent `Address` type encoding:

In `create_withdraw_message`, `Address` is hashed directly in its `.to_xdr` form which includes overheads making appended bytes exceeding 32 bytes long.

```
//create_withdraw_message

        // Add user address identifier
        encoded.append(&_user.to_xdr(env)); //@audit  > 32 bytes

        // Add target currency
        encoded.append(&target_token.to_xdr(env));//@audit  > 32 bytes
```

But in `calculate_domain_separator`, `Address` is hashed as sha256 of `to_xdr` form, which appends exactly 32 bytes.

```
//calculate_domain_separator
        // 5. verifyingContract field's hash
        let verifying_contract = env.current_contract_address();
        // Directly use byte representation of contract address
        let contract_xdr = verifying_contract.to_xdr(env);
        let contract_hash = env.crypto().sha256(&contract_xdr);
        domain_encoded.append(&contract_hash.into()); //@audit = 32 bytes
```

Impact: High - Medium

on-chain `create_withdraw_message` is currently inconsistent with client implementation observed `createWithdrawMessage` from js-client/src/signature-utils.mjs. Besides the incompliance with EIP712 mentioned above, `create_withdraw_message` also has a different order of concat from createWithdrawMessage. The dynamic field treatment(e.g., `requestHash`) also differs from the client.

This means the current implementation will cause failed signature validation in production based on observed client code, leading to withdrawal DOS and locked user funds.

Likelihood: High - The issue arises with every `withdraw` or withdraw message creation on-chain.

Recommendations:

1. Ensure that `create_withdraw_message` hashing order and treatment of dynamic types are consistent with the client.
2. Handle `Address` hash consistently.
3. Update documentation: Either remove EIP712 language or clarify the intended level of compliance.

Solv Finance: Resolved in `48e4a1c472428fbbd4810b143b1a0fa3b1cc6177`

Zenith: Verified. Standardized typed data hashing.

[H-2] User can bypass blacklisting through withdraw from vault

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- vault.rs#L327
- vault.rs#278

Description:

Blacklisted users can bypass blacklist restrictions and withdraw their funds by burning share tokens in the vault's withdrawal process, as the token contract's burn functions lack blacklist validation.

```
    fn withdraw_request(env: Env, from: Address, shares: i128,
    request_hash: Bytes) {
|>        from.require_auth(); // Verify caller identity
...
        if user_balance < shares {
            panic_with_error!(env, VaultError::InsufficientBalance);
        }

        // Burn user's shares directly via token burn
|>        token_client.burn(&from, &shares);
        // Set withdraw request status to PENDING
        env.storage()
            .persistent()
            .set(&request_key, &WithdrawStatus::Pending);
```

Since `withdraw_request`, `burn` or `burn_from` doesn't validate whether `from` is blacklisted, attack path below is possible:

1. A user max approves the vault for the share token;
2. A user becomes blacklisted and shouldn't be able to access their funds;
3. Blacklisted user calls `vault.withdraw_request(shares, request_hash)`, which calls `burn`.(note: In a separate finding, `burn_from` is suggested to be used here instead of `burn`)

4. In `burn` or `burn_from`, user address `from` is not checked for blacklisting.
5. `withdraw_request` tx succeeds and off-chain bot automatically processes the request.
6. The user calls `vault.withdraw()` to receive underlying assets.

Security Guarantee Broken: The user can bypass blacklisting by essentially trading through the vault to access locked funds.

Impact: High

- User can bypass blacklist restriction to access funds;
- Since funds come from the treasurer, the protocol loses funds to blacklisted users

Likelihood: Medium

- Any blacklisted user can perform the attack with the vault allowance preset.
- The attack is possible when `burn_blacklisted_tokens_by_admin` is not called. (e.g. It could be a situation where a blacklisted user might be eligible to appeal to be removed from the blacklist status pending further governance decision)

Recommendations:

Consider adding blacklist validation of the user in `withdraw_request`. e.g.:

```
if token_client.is_blacklisted(&from) {
    panic_with_error!(env, VaultError::Unauthorized);
}
```

Solv Finance: Resolved in `01211c8c3b27e3bc5379dbe356320ed2ece2a8b0`

Zenith: Verified. Added blacklist check in `burn`.

4.3 Medium Risk

A total of 5 medium risk findings were identified.

[M-1] calculate_domain_separator has incorrect type hash

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- vault.rs#L878

Description:

The `calculate_domain_separator` function uses an incorrect EIP712 type hash, potentially causing signature verification inconsistencies.

EIP712 domain separator currently uses the withdraw type instead of EIP712 domain separator type:

```
    // Calculate EIP712 DomainSeparator
    fn calculate_domain_separator(env: &Env) -> Bytes {
...
        // 1. EIP712Domain's TypeHash
        // keccak256("EIP712Domain(string name,string version,uint256
    chainId,address verifyingContract,bytes32 salt)")
        let type_hash = env.crypto().sha256(&Bytes::from_slice(env,
            b"Withdraw(uint256 chainId,string action,address user,address
    withdrawToken,uint256 shares,uint256 nav,bytes32 requestHash)"));
        domain_encoded.append(&type_hash.into());
```

According to the EIP712 standard, the domain separator should use the EIP712Domain type hash: `domainSeparator = hashStruct(eip712Domain)`("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract,bytes32 salt)")

Incorrect use of EIP712 domain type hash may lead to incompatibility with client. Currently conflicted version of client implementations are observed. For example:

```
//js-client/src/signature-utils.mjs

// Uses EIP712Domain type hash
const typeHash = sha256(Buffer.from("EIP712Domain(string name,string
    version,uint256 chainId,address verifyingContract,bytes32 salt)"));
```

Impact: Medium-Low

Currently, the SDK-client seems to also use the incorrect EIP712 domain separator version (consistent with vault.rs 's incorrect typehash). But a conflicted EIP712 domain type hash implementation also exists in the codebase.

The impact appears limited at the moment because only the protocol-controlled `withdrawVerifier` signs messages, not users. And users don't interact with EIP712 directly.

Likelihood: High - Incorrect type hash is used in every withdraw transaction.

Recommendations:

Consider consistently using the correct EIP712 domain separator typehash when creating message digests.

Solv Finance: Resolved in `48e4a1c472428fbbd4810b143b1a0fa3b1cc6177`

Zenith: Verified. Removed EIP-712 references.

[M-2] Blacklisted user can burn other token holders funds through allowance

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- fungible_token.rs#L177

Description:

The `burn_from` function in the fungible token contract lacks blacklist checks, allowing blacklisted users to bypass security restrictions and consume their allowances by burning other users' tokens.

Current behaviors in transfer/transferFrom restrict a blacklisted user from spending allowance, sending, or receiving tokens. For example, `transferFrom` prevents a blacklisted user from spending others' token allowance.

```
#[when_not_paused]
fn transfer_from(env: &Env, spender: Address, from: Address, to: Address,
    amount: i128) {
    // Check blacklist (including spender)
    Self::require_not_blacklisted(env, &spender);
    Self::require_not_blacklisted(env, &from);
    Self::require_not_blacklisted(env, &to);
    Base::transfer_from(env, &spender, &from, &to, amount);
}
```

However, does not perform blacklist validation on the spender parameter:

```
#[when_not_paused]
fn burn_from(env: &Env, spender: Address, from: Address, amount: i128) {
    // SEP-41 standard: burn using allowance mechanism
    Base::burn_from(env, &spender, &from, amount);
}
```

This is inconsistent with other token functions that properly implement blacklist checks:

- `transfer()` checks both `from` and `to` addresses
- `transfer_from()` checks `spender`, `from`, and `to` addresses
- `approve()` checks both `owner` and `spender` addresses (Note: This means users cannot approve zero to revoke existing allowance from a blacklisted user)
- `mint_from()` checks the `to` address

Attack path:

1. User A grants allowance to User B
2. User B gets blacklisted (e.g., for suspicious activity); User A cannot approve 0 to protect their funds.
3. User B calls `burn_from` to burn User A's tokens using the existing allowance
4. User A's tokens are destroyed, and the allowance is consumed

Impacts: Medium - Blacklisted users can bypass security controls and consume existing allowances, burn tokens from others' accounts.

Likelihood: Medium - Any blacklisted users that have existing allowance can burn funds form the target users.

Recommendations:

Add blacklist validation to the `burn_from` function to ensure blacklisted users cannot consume allowances and burn others' funds:

```
#[when_not_paused]
fn burn_from(env: &Env, spender: Address, from: Address, amount: i128) {
    // Check blacklist for spender
    Self::require_not_blacklisted(env, &spender);
    Base::burn_from(env, &spender, &from, amount);
}
```

Solv Finance: Resolved in `2fc729db1c658e49c4d20f2889f8f35cdf24921c`

Zenith: Verified. Added blacklist check in `burn_from`.

[M-3] Absence of NAV freshness checks

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- vault/src/vault.rs#L232

Description:

The vault relies on the oracle's `get_nav()` value to price deposits and withdrawals but does not enforce any freshness constraint. The oracle itself only stores `Nav` and `NavDecimals`, without recording a timestamp or ledger sequence.

As a result, vault operations may be executed against outdated NAV values. Because NAV is constrained to increase monotonically, stale NAV values will almost always be lower than the true current value. This benefits depositors, who can mint excess shares at the artificially low NAV, while harming withdrawers, who receive less than they should until NAV is updated.

Recommendations:

We recommend extending the oracle to persist the ledger sequence or timestamp of the last NAV update and return it alongside the NAV value. The vault should then enforce a configurable maximum age for NAV data

Solv Finance: Resolved in `893e291423216a0291d79da151505855302d7438`

Zenith: Verified

[M-4] Overflow in deposit minting formula enables Denial of Service

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- vault/src/vault.rs#L1033-L1066

Description:

The deposit minting logic in `calculate_mint_amount` multiplies deposit_amount by `10^(shares_decimals)` and `10^(nav_decimals)` before performing division. This multiplication often exceeds the maximum range of `i128` (approximately `1.7e38`), causing an overflow and triggering a revert. While the use of `checked_mul` prevents silent corruption, it results in deposits failing outright, creating a usability denial of service.

The issue manifests under common decimal configurations. For example, with `shares_decimals = 18`, `nav_decimals = 18`, and `currency_decimals = 6`, even a single token deposit (`10^6` units) overflows: `10^6 × 10^18 × 10^18 = 10^42`, far beyond the `i128` limit.

A general constraint emerges: for a 1-token deposit to succeed, the sum of `shares_decimals` and `nav_decimals` must be less than or equal to `38` minus `currency_decimals`. Otherwise, even the smallest deposit unit causes immediate overflow.

Recommendations:

We recommend implementing strict validation to prevent overflow.

Solv Finance: Resolved in `87f45abd9198663955e896e3e736a5de9a3c7cea`

Zenith: Verified

[M-5] Instance storage TTLs are not extended, risks of data expiration and potential DOS

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- fungible_token.rs#L482
- oracle.rs#L161
- vault.rs#L573-L575

Description:

Instance storage TTL is not extended in any of the three contracts (oracle, fungible-token, vault), creating risk of data expiration and potential DOS.

The contracts extensively use instance storage for critical data but lack TTL management. In Stellar Soroban, all storage types (including instance storage) have TTLs that must be periodically extended to prevent data from becoming inaccessible.

Key data stored in instance storage: Oracle: NAV, NAV decimals, NAV manager, vault address Fungible Token: Minters mapping, blacklist data, manager addresses Vault: Oracle address, treasurer, verifier keys, fee ratios, currency mappings

Impact: Medium - DOS. Temporary unavailability during manual data recovery affects users and could disrupt critical operations.

Likelihood: Medium While manual RPC upkeep to extend contract instance TTL is possible, this adds significant operational overhead and risks. With multiple instances of vault or Oracle deployment, the complexity increases. Currently, no off-chain upkeep mechanisms are observed in the codebase, making manual intervention the only mitigation strategy.

Recommendations:

Consider adding `env.storage().instance().extend_ttl(LEDGER_THRESHOLD_INSTANCE, LEDGER_BUMP_INSTANCE)` calls at the beginning of contract functions to automatically extend instance storage TTL.

Solv Finance: Acknowledged. We've chosen to manage TTL off-chain via a dedicated

Keeper to improve user experience.

Zenith: Resolved with off-chain keeper approach. The CAP-66 safety net provides good redundancy.

4.4 Low Risk

A total of 5 low risk findings were identified.

[L-1] Mutable withdraw parameters could prevent the correct functioning of the contract

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- vault/src/vault.rs#L630-L643

Description:

The withdrawal process is vulnerable to state changes between `withdraw_request` and `withdraw`. When a user submits `withdraw_request`, their shares are immediately burned and a `Pending` entry is stored under a request key derived from the user address, the current `withdraw_currency`, the user-supplied `request_hash`, the share amount, and the current NAV.

At withdrawal time, `withdraw()` recomputes this key using the current on-chain `withdraw_currency`. If an administrator changes the `withdraw_currency` after the request is created, the recomputed key will no longer match the stored entry. This causes `withdraw()` to fail with `InvalidRequestStatus` while the user's shares have already been burned and are unrecoverable, effectively freezing funds.

Separately, the contract does not snapshot the withdraw fee at request creation. Instead, it reads the fee at execution time, allowing an administrator to increase the fee after shares have been burned. This extracts additional value from users who have no recourse.

Recommendations:

We recommend snapshotting all withdrawal parameters, including `withdraw_currency` and the `withdraw fee`, at the time of `withdraw_request` and binding them to the stored request entry.

Solv Finance: Resolved in `ed7beb4052edeae26d90be89908539e12daaa3f9`

Zenith: Verified

[L-2] Unvalidated verifier keys

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- vault/src/vault.rs#L653-L671

Description:

The function `set_withdraw_verifier_by_admin` accepts a verifier public key as raw `Bytes` for any `signature_type` without validating its length or format.

Later, the withdraw function retrieves this value as `BytesN<32>` for Ed25519 or `BytesN<65>` for Secp256k1 and uses it directly.

If the stored key has the wrong size or encoding, the conversion into a fixed-size `BytesN` panics due to a type or length mismatch. The surrounding `unwrap_or_else` only covers the case where the key has not been set at all, leaving decode errors unhandled. This means that a simple misconfiguration can escalate into a runtime abort, blocking all withdrawals until the verifier key is corrected.

Additionally, there is no enforcement that Secp256k1 keys are stored in the correct uncompressed format with a 0x04 prefix. As a result, invalid or malformed keys may be persisted, and any attempt to use them will cause the contract to panic, resulting in a withdrawal denial of service.

Recommendations:

We recommend enforcing strict key validation at the time of configuration. The contract should reject verifier keys that do not match the expected length (32 bytes for Ed25519, 65 bytes for Secp256k1) or format (including the '0x04 prefix for uncompressed Secp256k1 keys).

Solv Finance: Resolved in `515efeba3b66ad5e4d07239ceb8fee3d4dd1ceb2`

Zenith: Verified

[L-3] Missing validation of `deposit_fee_ratio` in constructor

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- vault/src/vault.rs#L145-L162

Description:

The `__constructor` function validates `withdraw_fee_ratio` to ensure it falls within `[0, FEE_PRECISION]` but does not apply the same bound check to `deposit_fee_ratio` before storing it.

This omission allows unsafe values to be set at deployment. During deposits, the contract computes the fee as `(amount * deposit_fee_ratio) / FEE_PRECISION` and then subtracts it from the deposit amount.

If `deposit_fee_ratio` is negative, the calculation effectively credits users instead of charging them, inflating supply. If `deposit_fee_ratio` is greater than `FEE_PRECISION` (greater than 100%), the subtraction produces a negative `amount_after_fee`, leading to underflows or reverts.

Recommendations:

We recommend enforcing the same `[0, FEE_PRECISION]` validation on `deposit_fee_ratio` during contract deployment.

Solv Finance: Resolved in `cee5e907e0e7196931b0cefb1115d82139737a95`

Zenith: Verified

[L-4] NAV manager can bypass the rate increase limit

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- oracle/src/oracle.rs#L194-L207

Description:

The `set_nav_by_manager` enforces the NAV cap only per call by comparing a single-step increase against the vault's `withdraw_fee_rate` basis-point ratio relative to the current NAV.

The check lacks any notion of elapsed time, ledger gating, or cumulative budget. Because `withdraw_fee_rate` is a static ratio and the oracle does not track a last-update sequence or consumed allowance, a NAV manager can bypass the check and submit many small sequential updates in the same ledger or transaction, each under the per-call threshold, to staircase the NAV to an arbitrary level.

Recommendations:

We recommend introducing a period-aware rate limiter.

Solv Finance: Resolved in `893e291423216a0291d79da151505855302d7438`

Zenith: Verified

[L-5] Missing FX conversion in multi-currency deposits causes systematic mispricing

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- vault/src/vault.rs#L1033-L1066

Description:

The vault accepts deposits in multiple admin-approved currencies but prices them using a single NAV value from the oracle. The minting logic in `calculate_mint_amount` assumes that the deposit currency is valued one-to-one with the NAV unit, adjusting only for decimal scaling via `10^currency_decimals` and `10^shares_decimals`. This ignores the actual economic value of the deposit currency relative to the NAV denomination.

As a result, deposits are systematically mispriced. If the allowed deposit token is cheaper than the NAV unit, depositors are over-credited with shares and can extract value by withdrawing in the designated `WithdrawCurrency`. Conversely, if the deposit currency is more valuable than the NAV unit, users are under-credited, suffering a loss relative to their contribution.

Since the deposit function only validates that the token is in the allowed set and retrieves its decimals, without applying per-currency pricing, the system conflates numerical scale with economic value.

Recommendations:

We recommend aligning deposits strictly to the withdrawal currency to eliminate valuation mismatches. NAV should be defined in the same unit, and deposits restricted accordingly. If multi-currency support is required, the oracle must provide per-currency FX rates.

Solv Finance: Acknowledged. From a business perspective, supporting multiple currencies ensures that the value of each currency remains consistent — the intended behavior is for all supported deposit currencies to be treated as equivalent to the NAV denomination (i.e., priced 1:1).

Zenith: This is considered resolved based on the confirmation that 1:1 pricing is business logic. The admin's control over allowed currencies provides appropriate oversight.

4.5 Informational

A total of 2 informational findings were identified.

[I-1] Inconsistent event data layout in `set_vault_by_admin`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- oracle/src/oracle.rs#L109-L132

Description:

The `set_vault_by_admin` function emits an event with the vault address placed in the topic tuple and the admin address in the event data.

By contrast, `set_nav_manager_by_admin` emits only the event symbol in the topic and places (admin, manager) in the data.

This inconsistency in how topics and data are structured makes off-chain indexing and filtering unreliable. Consumers cannot depend on a consistent event schema across admin "set_*" actions, meaning that filters written for one event (e.g., topic = symbol, data = (admin, target)) will fail for the other. Additionally, the field ordering semantics differ, further complicating downstream parsing.

Recommendations:

We recommend standardizing event emission for admin update actions.

Solv Finance: Resolved in `89ece6ca157fd208af29131c84506e860e14bf54`

Zenith: Verified.

[I-2] Unused Datakey UsedRequestHash and WithdrawRequestStatus

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- vault.rs#L77-L80

Description:

DataKey enum variants `UsedRequestHash` and `WithdrawRequestStatus` are defined but never referenced in the codebase, representing dead code that should be removed.

```
pub enum DataKey {
...
    /// Withdrawal request status
    WithdrawRequestStatus,
    /// Used request hash mapping
    UsedRequestHash(Bytes),
}
```

Currently, `request_key` is used in `withdraw_request` and `withdraw`:

```
// Set withdraw request status to PENDING
env.storage()
    .persistent()
    .set(&request_key, &WithdrawStatus::Pending);
```

This represents dead code that increases contract bytecode size unnecessarily.

Recommendations:

Consider removing unused datakey variants.

Solv Finance: Resolved in `d4aabed287b418b1eac5df776985e81c4d1803fb`

Zenith: Verified. Removed unused datakey variants.