ClarityAlliance

Hermetica Vaults Security Review

Conducted by:
Kristian Apostolov, Alin Barbatei (ABA),
Silverologist

January 27th, 2026

Contents

[QA-21] Use Zest Market Bundle Operations

1. About Clarity Alliance

Clarity Alliance is a team of expert whitehat hackers specialising in securing protocols on Stacks.

They have disclosed vulnerabilities that have saved millions in live TVL and conducted thorough reviews for some of the largest projects across the Stacks ecosystem.

Learn more about Clarity Alliance at clarityalliance.org.

2. Disclaimer

This report is not, and should not be considered, an endorsement or disapproval of any project, team, product, or asset, nor does it reflect on their economics, value, business model, or legal compliance. It does not provide any warranty regarding the absolute bug-free nature or functionality of the analyzed technology.

Nothing in this report should be used to make investment or participation decisions. It does not constitute investment advice. Instead, it reflects an extensive assessment process intended to help clients improve code quality and reduce the inherent risks associated with cryptographic tokens and blockchain systems.

Blockchain technology presents ongoing and significant risk, and each company or individual remains responsible for their own due diligence and security posture. Clarity Alliance aims to reduce attack vectors and technological uncertainty but does not guarantee the security or performance of any system we review.

All assessment services are subject to dependencies and active development. Your access to and use of any services, reports, or materials is at your sole risk on an as-is and as-available basis.

Cryptographic tokens are emergent technologies with high technical uncertainty, and assessment results may include false positives, false negatives, or other unpredictable outcomes. Smart contracts may depend on multiple external parties, remain vulnerable to internal or external exploitation, and may carry elevated risks if owner privileges remain active. Accordingly, Clarity Alliance does not guarantee the explicit security of any audited smart contract, regardless of the reported verdict.

3. Introduction

A time-boxed security review of the Hermetica hBTC protocol, where Clarity Alliance reviewed the scope and provided insights on improving the protocol.

4. About Hermetica Vaults

What are Hermetica Vaults?

Hermetica Vaults are Stacks (Clarity) yield vault smart contracts that implement ERC-4626-style share mint/redeem, publish periodic (daily) NAV updates into a liquid staking token (LST), and execute yield strategies via integrations with external on-chain protocols.

hBTC is the reference implementation of the Hermetica Vault framework: a BTC-denominated "BTC-on-BTC" vault that deploys deposited BTC across integrated DeFi venues to generate yield while tracking assets and liabilities on-chain.

Deposited BTC is programmatically allocated to strategies that produce BTC-denominated returns. The core strategy borrows stablecoin liquidity against BTC collateral (e.g., borrowing USDh on Zest), deploys the borrowed assets into yield-bearing positions (e.g., staking USDh via Hermetica), and periodically realizes strategy P&L, converting net proceeds back into BTC for accrual to vault share value.

The core design principles are:

- Self-custodial: Assets move only via protocol smart contracts; privileged changes are gated by governance and enforced timelocks.
- Permissionless redemption: Any share holder can redeem according to contract rules, including withdrawal back to native Bitcoin.
- Full transparency: Positions, flows, and accounting are on-chain and independently verifiable.
- Automated risk controls: Pre-defined leverage, delta, and interest spread controls are enforced programmatically.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

Scope

The scope of this security assessment covered the Hermetica hBTC contracts in the Hermetica repository:

- `contracts/hbtc/protocol/controller-v1.clar`
- `contracts/hbtc/protocol/fee-collector-v1.clar`
- `contracts/hbtc/protocol/hq-hbtc-v1.clar`
- `contracts/hbtc/protocol/reserve-fund-v1.clar`
- `contracts/hbtc/protocol/reserve-v1.clar`
- `contracts/hbtc/protocol/state-v1.clar`
- `contracts/hbtc/protocol/trading-v1.clar`
- `contracts/hbtc/protocol/vault-v1.clar`
- `contracts/hbtc/protocol/interfaces/hermetica-interface-v1.clar`
- `contracts/hbtc/protocol/interfaces/zest-interface-v1.clar`
- `contracts/hbtc/tokens/hbtc-token.clar`

Initial Commit Reviewed

`43fff97fdecd8c3a2c2fc5b6070967fad4aac28f`

Final Commit After Remediations

`135b453e1b9c4bc4784640f99cac49f9532d4bff`

Subsequent Commits and Pull Requests Reviewed

- `97832bd64273a08b0a6267804b35133b6205d89e`
- `63e463aad9ed27896453f2158e69fea6dca6ef1e`
- PR#93
- PR#91: commit `b2e7ad8`
- PR#128: commit `e7bc47b`
- PR#130
- PR#138: commit `bf26a83`
- PR#139: commit `163e9ca`
- PR#140: commit `5929b0f`
- PR#143
- PR#144
- PR#152
- PR#153
- PR#171

Only the subsequent changes introduced *(after the final commit)* in the commits and pull requests listed above were reviewed.

7. Executive Summary

Over the course of the security review, Kristian Apostolov, Alin Barbatei (ABA), Silverologist engaged with - to review Hermetica Vaults. In this period of time a total of **52** issues were uncovered.

Protocol Summary

Protocol Name	Hermetica hBTC
Date	January 27th, 2026

Findings Count

Severity	Amount
High	4
Medium	10
Low	17
QA	21
Total Findings	**52**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Incorrect Performance Fee Calculation Leads To Unaccounted Rewards	High	Resolved
[H-02]	Reward Profit Handling Is Incorrect	High	Resolved
[H-03]	Excess Loss Handling Is Incorrect	High	Resolved
[H-04]	Covered Loss Handling Is Incorrect	High	Resolved
[M-01]	Vault Withdrawal Share Rounding Favors Users Over the Protocol	Medium	Resolved
[M-02]	Malicious Trader Can Drain Removable Zest Collateral	Medium	Resolved
[M-03]	Premature Share Price Snapshot Leads to Unbacked Pending Claims	Medium	Resolved
[M-04]	Vault Cannot Be Emptied After Share Price Divergence	Medium	Resolved
[M-05]	Updated Management And Performance Fees Are Applied Retroactively	Medium	Resolved
[M-06]	hBTC Token Is Not SIP-10 Compliant	Medium	Resolved
[M-07]	hBTC Public Share Burning Enables a First-Deposit DoS Attack Variant	Medium	Resolved

[M-08]	Maximum and Minimum Cap Limitations May Endanger the Protocol	Medium	Resolved
[M-09]	Tokens With tx-sender-Based Authorization Are Not Fully Supported	Medium	Resolved
[M-10]	STX Integration Issues Due to Clarity 4 Security Constraints	Medium	Resolved
[L-01]	Incomplete Blacklist Flow	Low	Acknowledged
[L-02]	Missing Reset Mechanism for Custom Parameters	Low	Resolved
[L-03]	Management Fee Overstated by Using Total Assets	Low	Resolved
[L-04]	Missing Explicit Validation for is-express Parameters	Low	Resolved
[L-05]	Vault withdraw-many Functionality Is Brittle	Low	Resolved
[L-06]	hBTC Token Name Should Not Be Changeable	Low	Partially Resolved
[L-07]	Reward Distribution Should Not Be Allowed After the Vault Has Been Emptied	Low	Resolved
[L-08]	Positive Slippage Remains in the Hermetica Interface Contract After a Mint	Low	Resolved
[L-09]	Race Condition During Fund Transfers Due to Rewarder Role Ambiguity	Low	Resolved
[L-10]	Admin Should Not Be Able To Change Price Staleness Threshold	Low	Resolved

[L-11]	Exit Fee Transfers Should Not Be Bound To Vault Status	Low	Resolved
[L-12]	Missing Transfer Authorization Check in Reserve Fund Transfer	Low	Resolved
[L-13]	Max Slippage Can Be Set for Arbitrary Assets	Low	Resolved
[L-14]	Security Enhancements in Case of Ownership Compromise	Low	Resolved
[L-15]	Unnecessary Token Allowance on Zest Operations With No Outflows	Low	Resolved
[L-16]	Repaying Zest Debt May Leave Dust in the Interface	Low	Resolved
[L-17]	Loose Token Allowances on Zest Market Interface Calls	Low	Resolved
[QA-01]	Unnecessary External Call for is-standard Verification	QA	Acknowledged
[QA-02]	Current Owner Can Claim Ownership Multiple Times	QA	Resolved
[QA-03]	Current Owner Can Request Next Ownership to Himself	QA	Resolved
[QA-04]	Incorrect Reuse of ERR_NOT_OWNER Code When Claiming Ownership	QA	Resolved
[QA-05]	Incorrect Event Action Name for request-new-admin Function	QA	Resolved
[QA-06]	Optimize Double Timestamp Retrieval	QA	Resolved
[QA-07]	Hardcode Constants Instead of Computing at Runtime	QA	Resolved

[QA-08]	Zest Interface Contract Can Be Slightly Improved	QA	Resolved
[QA-09]	Excessive Price Feed Updates in Trading Interface	QA	Resolved
[QA-10]	Trading Interface: Ambiguous Function Naming Convention	QA	Resolved
[QA-11]	Trading Interface Can Be Optimized	QA	Resolved
[QA-12]	Vault Deposit Cap Consideration	QA	Resolved
[QA-13]	Management Fee Max Amount Implementation – Documentation Discrepancy	QA	Resolved
[QA-14]	Vault Contract Can Be Slightly Improved	QA	Resolved
[QA-15]	First Depositor Inflation Attack Considerations	QA	Resolved
[QA-16]	Add Vault Action Preview Functions	QA	Resolved
[QA-17]	Codebase Print Statements Improvements	QA	Resolved
[QA-18]	Miscellaneous Codebase Improvements	QA	Resolved
[QA-19]	Hermetica Interface Mint Asset Transfer Restriction Can Be Improved	QA	Resolved
[QA-20]	Use the Recipient Feature of the Zest Market Interface	QA	Resolved
[QA-21]	Use Zest Market Bundle Operations	QA	Resolved

8. Findings

8.1. High Findings

[H-01] Incorrect Performance Fee Calculation Leads To Unaccounted Rewards

Location:

- controller-v1.clar#L36

Description In the current design, rewards are manually synchronized by the `rewarder` role calling `controller::log-reward`. Per the protocol documentation, this operation is intended to occur once per day.

Reward logging follows one of three execution paths:

- `handle-profit`, when a net profit is achieved
- `handle-loss-covered`, when a loss occurs but can be covered by the reserve fund
- `handle-loss-exceeded`, when a loss occurs and the reserve fund is insufficient

Whether the `handle-profit` branch is selected depends on the `is-profit` value:

```
(perf-fee (if is-positive (/ (* (get perf-fee fees) reward) bps-base) u0))
(mgmt-fee (/ (* (get mgmt-fee fees) total-assets) bps-base pct-base))
(total-fees (+ perf-fee mgmt-fee))
(is-profit (and is-positive (>= reward total-fees)))
```

However, `is-profit` can be incorrectly evaluated as `false` because the performance fee is calculated on the full `reward` amount rather than on the profit portion (i.e., the amount remaining after the management fee). This can cause execution to incorrectly enter one of the `handle-loss-*` branches.

Consider the following scenario:

- Logged reward: `+100` units
- Performance fee: 10% → `10` units
- Management fee: `91` units
- Total fees: `10 + 91 = 101` → `is-profit = false`

Here, the reward (`100`) exceeds the management fee (`91`) but is less than the combined total fees (`101`), so execution reaches the `handle-loss-*` branches.

As a result, the remaining `9` units are left unaccounted for, since the loss-handling branches neither apply fees nor distribute rewards to users.

Recommendation Apply the performance fee only to the portion of the reward remaining after the management fee is deducted (i.e., profit), rather than to the full reward amount. This change:

- Ensures the correct `handle-*` branch is selected when `reward < total-fees` and `reward > mgmt-fee`
- Prevents users from paying a performance fee on the management-fee portion
- Ensures consistent behavior with loss-handling cases, where performance fees are not applied
- Aligns with the documentation, which states that a “performance fee on profits” is applied

Example implementation:

```
-   (perf-fee (if is-positive (/ (* (get perf-fee fees) reward) bps-base) u0))
    (mgmt-fee (/ (* (get mgmt-fee fees) total-assets) bps-base pct-base))
+   (reward-after-mgmt-fee (if (>= reward mgmt-fee) (- reward mgmt-fee) u0))
+   (perf-fee (if is-positive (/ (*
+ (get perf-fee fees) reward-after-mgmt-fee) bps-base) u0))
    (total-fees (+ perf-fee mgmt-fee))
    (is-profit (and is-positive (>= reward total-fees)))
```

[H-02] Reward Profit Handling Is Incorrect

Location:

- controller-v1.clar#L133

Description In the current design, rewards are manually synchronized by the rewarder role via `controller::log-reward`. Per the protocol documentation, this operation is intended to be performed once per day.

When `log-reward` is called and the reward includes a profit, the `handle-profit` function is executed. The system state is updated as follows:

- Pending fees = performance fee + management fee
- Pending reserve fund = reserve rate applied to the post-fee reward
- New reward = reward - pending fees - pending reserve fund

Within `state::update-state`, this results in:

- `pending-fees` increasing by the newly calculated pending fees
- `pending-rf` increasing by the newly calculated pending reserve fund
- `total-assets` increasing by the new reward amount

A related observation is that net assets (after claims and fees) are used to determine the share price. The `state::get-net-assets` function computes net assets as:

```
(define-read-only (get-net-assets)
  (- (get-total-assets) (get-pending-claims) (get-pending-fees)
    (get-pending-rf))
)
```

An issue arises in the current profit-handling flow because the amount added to `total-assets` should be the **gross** reward, not the reward net of fees and the reserve fund. This is because `get-net-assets` already subtracts `pending-fees` and `pending-rf` when calculating net assets.

As a result, fees and the reserve fund are effectively subtracted twice, causing net assets to be understated. This leads to an incorrect share price and can result in overpaid fees.

Recommendation In `controller::handle-profit`, when preparing the data passed to state updates, use the **gross** reward amount (before deducting fees and the reserve fund) for the `reward` field. Concretely, add the full `reward` amount to the state (not the `reward-net` amount) to maintain consistency with the net asset calculation.

[H-03] Excess Loss Handling Is Incorrect

Location:

- controller-v1.clar#L211

Description In the current implementation, rewards are manually synchronized by the `rewarder` role via `controller::log-reward`, which—per the protocol documentation—is expected to run once per day.

When rewards are processed, if the total loss (management fee plus the reward amount when the reward is negative) exceeds the available local funds (pending reserve fund plus the reward amount when the reward is positive), the protocol draws funds from the Reserve Fund (RF) contract to cover the shortfall.

If the RF contract cannot fully cover the deficit, the remaining amount becomes a net loss to total assets, effectively socializing the uncovered loss.

This scenario is handled by `controller::handle-loss-exceeds`. In this function, the loss is explicitly subtracted from the current reward state (i.e., from total assets):

```
(let (
  (loss (- req-rf total-rf))
)
;; ... code ...
(if (> total-rf pending-rf)
    (try! (contract-call? .reserve-fund-v1 transfer sbtc-token
      (- total-rf pending-rf) reserve none))
    true
)
(ok (try! (contract-call? .state-v1 update-state
(list
    { type: "pending-fees", amount: mgmt-fee, is-add: true }
    { type: "pending-rf", amount: pending-rf, is-add: false })
(some { reward: loss, is-add: false })
none)))
```

However, the current implementation does not account for losses correctly and can overstate (inflate) losses. The flow below clarifies the issue.

From a state-update perspective, `pending-fees` must be increased by the management fee, which is done correctly:

```
{ type: "pending-fees", amount: mgmt-fee, is-add: true }
```

Similarly, because `handle-loss-exceeds` is only reached when the pending reserve fund is insufficient to cover the loss, the entire `pending-rf` should be deducted. This is also done correctly:

```
{ type: "pending-rf", amount: pending-rf, is-add: false })
```

The problem lies in how the remaining loss is applied to total assets:

```
(some { reward: loss, is-add: false })
```

The management fee is already accounted for as a loss by being added to `pending-fees`. Therefore, it must not be deducted again from total assets via a negative reward update. As implemented, `loss` is derived from `req-rf` (which includes the management fee component) and is then applied as an additional negative reward, resulting in the management fee being double-counted.

Additionally, for correct accounting, any amount transferred from the RF contract (if any) must be reflected positively in the reward accounting. Denote this as `transfer-amount`.

At this point, two sub-cases exist:

1. **Reward is positive (`is-positive=true`)**

Correct accounting should be:

```
(some { reward: (+ reward transfer-amount), is-add: true })
```

Here, the transferred amount (which may be 0) is added to total assets alongside the reward. This covers cases where the management fee is large enough to create a net loss even when the reward is positive.

Example scenario:

- reward: 5
- management fee: 10
- pending reserve fund: 2
- reserve fund balance: 2

By adding the management fee to `pending-fees`, the system already records a `-10` impact. The true additional loss should be `-1`, because `req-rf=5` (management fee minus reward) and the available RF coverage is `2 + 2`, leaving a deficit of `1`.

The current implementation incorrectly records a `-11` loss instead of `-1`. Under the corrected approach, the management fee is recorded as `-10`, and `9` is added to assets as reward (`5` reward + `4` total available RF), resulting in the correct net loss.

2. **Reward is negative (`is-positive=false`)**

In this case, accounting must compare the magnitude of the negative reward against the RF transfer amount. If the negative reward exceeds the transferred amount, there is a net loss (in addition to the already-recorded management fee). If the transferred amount exceeds the negative reward, there is a net gain to total assets.

A correct approach is:

```
(let (
  (transfer-amount (- total-rf pending-rf))
  ;; only if reward is negative (is-positive = false)
  (account-data (if (>= reward transfer-amount)
      { reward: (- reward transfer-amount), is-add: false}
      { reward: (- transfer-amount reward), is-add: true}
  ))
)
;; ... code ...
(some { reward: (get reward account-data), is-add: (get is-add account-data) })
```

Recommendation

Update the implementation to correctly account for losses in both positive- and negative-reward scenarios, while ensuring the management fee is only accounted for once (via `pending-fees`). The fix should also incorporate the RF transfer amount into the reward accounting.

Example implementation fix (note that print events should also be updated accordingly):

```
(let (
    (transfer-amount (- total-rf pending-rf))
    (account-data (if is-positive
      { reward: (+ reward transfer-amount), is-add: true}
      (if (>= reward transfer-amount)
        { reward: (- reward transfer-amount), is-add: false}
        { reward: (- transfer-amount reward), is-add: true}
      )
    ))
  )
        ;; ... code ...
  (some { reward: (get reward account-data), is-add:
    (get is-add account-data) })
```

[H-04] Covered Loss Handling Is Incorrect

Location:

- controller-v1.clar#L174

Description In the current implementation, rewards are manually synchronized by the `rewarder` role via `controller::log-reward`, which—per the protocol documentation—is expected to run once per day.

When rewards are processed, if the total loss (management fee plus the reward amount when the reward is negative) exceeds the available local funds (pending reserve fund plus the reward amount when the reward is positive), the protocol draws funds from the Reserve Fund (RF) contract to cover the shortfall.

If the RF contract can cover the deficit, the underlying net assets should be preserved and user share-price loss should be avoided. This scenario is handled by `controller::handle-loss-covered`.

```
(let (
    (transfer-amount (if (> req-rf pending-rf) (- req-rf pending-rf) u0))
    (rf-decrease (if (> transfer-amount u0) pending-rf req-rf))
  )
    ;; ... code ...
    (if (> transfer-amount u0)
      (try!
        (contract-call? .reserve-fund-v1 transfer sbtc-token transfer-amount reserve none))
      true
    )
    (try! (contract-call? .state-v1 update-state
      (list
        { type: "pending-rf", amount: rf-decrease, is-add: false }
        { type: "pending-fees", amount: mgmt-fee, is-add: true })
      (some { reward: u0, is-add: false })
      none))
```

However, the current implementation does not correctly account for losses at the state level. The flow below illustrates the issue.

From a state-update perspective, `pending-fees` must be increased by the management fee, which is done correctly:

```
{ type: "pending-fees", amount: mgmt-fee, is-add: true })
```

Next, `pending-rf` must be decreased by the amount used to cover the deficit. If `pending-rf` is insufficient, an additional transfer is pulled from the RF contract (which is guaranteed to succeed in this code path). The `rf-decrease` value is computed correctly and deducted from the pending amounts:

```
{ type: "pending-rf", amount: rf-decrease, is-add: false }
```

The incorrect accounting occurs in the final step, where no reward/loss is applied to the underlying total assets:

```
(some { reward: u0, is-add: false })
```

At this point, the logic should add or remove funds (depending on the reward direction and the RF transfer amount) to keep the accounting balanced and preserve net assets. Instead, it always applies a zero reward, which can incorrectly reduce net assets and therefore the share price.

Consider the following example:

```
pending-fees = 20
pending-reserve = 20
pending-claim = 10
net-assets = 50

total-assets = pending-fees + pending-reserve + pending-claim + net-assets = 100

reserve-fund-balance = 50
```

Assume a reward of `-20` (`is-positive = false`) and a management fee of `-25` (the `-` here denotes a loss).

The resulting `req-rf` is `45`. Since `pending-reserve` is only `20`, an additional `25` tokens are transferred from `reserve-fund-balance` to cover the shortfall.

The intended outcome is that net assets remain unchanged (i.e., net assets before equals net assets after), while all components are accounted for correctly. Net assets are computed as: `net-assets = total-assets - pending-fees - pending-reserve - pending-claim`

Applying the current implementation:

1. `pending-fees` increases by `25` ⇒ net assets decrease by `25` ⇒ `net-assets = net-assets_original - 25`
2. `pending-reserve` decreases by `20` ⇒ net assets increase by `20` ⇒ `net-assets = net-assets_original - 25 + 20`

Because the reward/loss is not applied in the final step, the net result is still a `-5` change in net assets, which incorrectly reduces the share price.

A similar issue can occur when rewards are positive but the management fee is larger.

Recommendation To fix this, the state update must also account for the reward direction (gain vs. loss) and the RF transfer amount. In other words, the final `reward` update should reflect both the reward value and any amount transferred from the RF contract.

Example implementation (note that print events should be updated accordingly):

```
(let (
    (account-data (if is-positive
      { reward: (+ reward transferred-amount), is-add: true}
      (if (>= reward transferred-amount)
        { reward: (- reward transferred-amount), is-add: false}
        { reward: (- transferred-amount reward), is-add: true}
      )
    ))
  )
;; ... code ...
  (some { reward: (get reward account-data), is-add:
    (get is-add account-data) })
```

By incorporating both the reward and the transfer amount, the accounting remains balanced. Using the example above, the transfer amount is `45 - 20 = 25`. This offsets the remaining `-5` discrepancy: the resulting `account-data` becomes `25 - 20 = 5` with `is-add: true`, which correctly preserves net assets and prevents an unintended share-price decrease.

8.2. Medium Findings

[M-01] Vault Withdrawal Share Rounding Favors Users Over the Protocol

Location:

- vault-v1.clar#L134

Description When users initiate a withdrawal via `vault-v1::init-withdraw`, they specify the amount of assets to withdraw. The contract then calculates the number of shares to burn using the following formula, which rounds down:

```
(shares (/ (* assets share-base) (get share-price state)))
```

Because the division rounds down, the computed value can slightly underestimate the number of shares that should be burned. This allows users to withdraw marginally more assets than their burned shares would otherwise entitle them to. Over time, this discrepancy may compound and negatively impact the protocol.

Recommendation Rounding should generally favor the protocol. In this case, round up the number of shares to burn in `init-withdraw`.

Example implementation:

```
(define-public (init-withdraw (assets uint) (is-express bool))
  (let (
    (state
      (contract-call? .state-v1 get-withdraw-state contract-caller is-express))
-   (shares (/ (* assets share-base) (get share-price state)))
+   (share-price (get share-price state))
+   (shares (/ (+ (* assets share-base) (- share-price u1)) share-price))
  )
```

[M-02] Malicious Trader Can Drain Removable Zest Collateral

Location:

- zest-interface-v1.clar#L152-L237

Description Deposits into Zest v2 are performed via `zest-interface-v1::zest-deposit`, which is callable by the approved trader role. This function deposits into a specified Zest v2 vault via the `vault-trait` argument, and the resulting LP tokens (represented by `z-token-trait`) are then transferred.

By design, the trader role should only be able to call trading-related functions (interfaces and the trading contract) and should not be able to change which assets are approved for trading.

However, the current Zest deposit validations allow a malicious trader to remove all *removable* collateral from Zest (while keeping the position healthy) and then transfer it to an arbitrary address—effectively stealing all removable collateral.

The issue lies in how vault validations are handled in the current `zest-interface-v1` contract interface.

Zest vaults are, by definition, both vaults and tokens themselves (a subset of SIP-10). Zest vaults currently follow the interface below (note: Zest v2 is still under development and may change; however, the dual nature of a vault—deposit/redeem functionality plus SIP-10 compliance for LP transfers—will remain):

```
(define-trait tokenized-vault
  (
    ;; --- sip-10 ---
    (get-name         () (response (string-ascii 32) uint))
    (get-symbol       () (response (string-ascii 32) uint))
    (get-token-uri    () (response (optional (string-utf8 256)) uint))
    (get-decimals     () (response uint uint))
    (get-total-supply () (response uint uint))
    (get-balance      (principal) (response uint uint))

    (transfer         (uint principal principal (optional (buff 34)))
      (response bool uint))

    ;; --- reads ---
    (get-assets        ()     (response uint uint))
    (convert-to-shares (uint) (response uint uint))
    (convert-to-assets (uint) (response uint uint))

    ;; --- mutate ---
    (deposit  (uint uint principal) (response uint uint))
    (redeem (uint uint principal) (response uint uint))
  ))
```

In `zest-interface-v1`, the `zest-vault-trait-v1` trait is used instead, which only requires `deposit` and `redeem`:

```
(define-trait zest-vault-trait
  (
    (deposit (uint uint principal) (response uint uint))
    (redeem (uint uint principal) (response uint uint))
  )
)
```

This trait is used when treating the vault as a funds-moving contract. Additionally, `zest-interface-v1` uses a separate trait to represent the underlying vault LP token, `z-token-trait`, which is a SIP-10 trait.

The implementation fails to account for the fact that the vault itself is the SIP-10 token. In other words, the contract implementing `z-token-trait` must be the same contract as the one implementing `vault-trait`. This mismatch exists in both `zest-deposit` and `zest-redeem`, though the mismatch alone is not the most severe issue.

More importantly, `zest-interface-v1::zest-deposit` does not validate the `z-token-trait` argument at all. As a result, a trader can pass an arbitrary SIP-10-compliant contract and cause `zest-interface-v1` to call it with `as-contract` privileges:

```
(try! (as-contract
  (contract-call? z-token-trait transfer received this-contract .reserve-v1 none)))
```

This causes the called contract to observe `tx-sender` and `contract-caller` as `zest-interface-v1`. From this position, a malicious contract can drain assets by abusing the authority of `zest-interface-v1`.

In theory, `zest-interface-v1` should not hold SIP-10 tokens, since LP tokens and collateral tokens are expected to be transferred to and from the `reserve` contract.

However, within the malicious SIP-10 contract, an attacker can call `zest-market::collateral-remove` using `zest-interface-v1` as the account/recipient argument. Because `zest-interface-v1` is the `tx-sender` within the malicious SIP-10 contract, the Zest market contract permits the call and removes collateral (up to the maximum removable amount while keeping the position healthy) to the `zest-interface-v1` contract.

The malicious contract can then transfer that collateral to an arbitrary address by calling the collateral token's `SIP-10::transfer`. This transfer succeeds because SIP-10 requires `tx-sender` or `contract-caller` to match the sender, and in this context `tx-sender` remains `zest-interface-v1`, enabling the collateral drain.

Example attack:

- Trader role is compromised.
- The trader calls `zest-interface-v1::zest-deposit` with an arbitrary vault and supplies a malicious SIP-10 token as `z-token-trait`.
- `zest-deposit` calls `SIP-10::transfer` on the malicious contract.
- Inside the malicious SIP-10 contract (`tx-sender` and `contract-caller` are `zest-interface-v1`):
  - `zest-market::collateral-remove` is called (for all collaterals), pulling all removable collateral into `zest-interface-v1`.
  - `collateral::SIP-10::transfer` is called to move funds from `zest-interface-v1` (permitted since `tx-sender` is `zest-interface-v1`) to an attacker-controlled address.

Recommendation

Update `vault-trait` to use the `tokenized-vault` interface above, and reuse `vault-trait` for transferring zTokens to and from the reserve. Remove the separate `z-token-trait` entirely. Apply these changes in both `zest-deposit` and `zest-redeem`.

[M-03] Premature Share Price Snapshot Leads to Unbacked Pending Claims

Location:

- vault-v1.clar#L222-L245
- vault-v1.clar#L128-L164

Description
In the current implementation, users deposit assets into the vault and receive hBTC shares. When they later initiate a redemption or withdrawal by burning these shares, the amount of assets they receive is determined using the vault's current share price.

However, the vault's share price is not perfectly synchronized with its real asset value. Rewards and PnL adjustments are applied manually once per day (per protocol documentation). Unless a withdrawal occurs immediately after a reward distribution/update, the share price used for the calculation may be stale.

When a withdrawal is initiated, a claim is created using this potentially outdated price. The system must later fund the claim by transferring assets from the reserve to the vault.

Consider the following scenario:

- Alice deposits 100 units of assets into the vault; she is the only depositor.
- The trader role opens a position using Alice's funds.
- The position incurs a PnL of -1, but the daily reward update has not yet occurred.
- Alice initiates a withdrawal for all her shares, which are valued using the last recorded share price.
- Her claim is created for 100 assets, even though the vault's actual net assets are 99.
- At this point, the claim cannot be funded because the system's true holdings are insufficient to meet the promised payout.

To resolve this, the protocol team must manually transfer assets (outside of the project) to the reserve and then fund the claim. Additionally, because pending claims are not affected by subsequent PnL, negative PnL cannot be passed through to users with pending claims.

Recommendation
Restructure the withdrawal process so that the share price used is the one at the time the claim is funded, rather than at the time the withdrawal is initiated.

A full breakdown of the proposed changes:

- The vault `init-redeem` logic should transfer the hBTC shares to the vault itself.
- The `claims` map should include a new `shares` field to store the number of shares to be withdrawn and burned at funding time.
- Remove the `init-withdraw` function, since the recorded asset amount is not guaranteed, making the function unreliable and potentially confusing.
- When a claim is funded, burn the shares at that moment and store the resulting asset amount in the `amount` field of the `claims` entry, to be used normally in `withdraw`.
- The vault does not need to call the `"pending-claims"` operation, since all claims become "pending shares" and are tracked in the vault. If needed, the `pending-claims` field can be retained and repurposed to represent total shares pending burn. In that case, `net-assets` should no longer incorporate `pending-claims`.
- No changes are needed for the `trading-v1` contract.
- If `pending-claims` is not repurposed to track shares, the related logic can be removed from the `state-v1` contract.

This prevents scenarios like the one described, because users receive the exact amount available at the time of funding. As a result, users cannot force the protocol team to cover shortfalls caused by stale pricing.

Note that this change means users will realize any profit or loss that occurs between redeem initiation and claim funding. This modifies the current user-facing behavior, which implies that once a withdraw/redeem is initiated, the output amount is guaranteed to be the saved amount (albeit only after sufficient funds are added and the claim is funded).

With the new logic, the funding time affects the amount received. During the cooldown period, only the `manager` role can call `fund-claim`; after the cooldown, anyone can call it. In both cases, the caller can influence *when* a claim is funded and therefore (to some extent) the amount the user receives (e.g., before or after a reward update). This is not necessarily an issue in practice: the team can fund claims during cooldown via the manager role, and post-cooldown funding becomes market-driven since anyone can call it. This also prevents users from timing exits by waiting to accumulate rewards and then exiting instantly.

Ultimately, the trade-off is between (a) requiring the protocol to cover negative PnL that can arise from unbacked pending claims while guaranteeing a snapshotted withdrawal amount, and (b) ensuring all withdrawals are fully backed at funding time, but without guaranteeing the deliverable asset amount at initiation.

[M-04] Vault Cannot Be Emptied After Share Price Divergence

Location:

- state-v1.clar#L380

Description The vault's share price starts at a 1:1 ratio with the underlying assets and changes over time as rewards and fees accrue.

When a user initiates a withdrawal, the vault state is updated via `state::update-state`, which calculates the share price as follows:

```
(define-read-only (get-share-price)
  (let (
    (net-assets (get-net-assets))
    (total-supply (unwrap-panic
      (contract-call? .hbtc-token-v1 get-total-supply)))
  )
    (if (> total-supply u0)
      (/ (* net-assets share-base) total-supply)
      share-base  ;; 1:1 for the first deposit
    )
  )
)
```

After the state update, the contract enforces a maximum deviation check by comparing the pre-update and post-update share prices:

```
(define-read-only (check-max-deviation (old-price uint) (new-price uint))
  (let (
    (threshold (get-max-deviation))
    (abs-diff (if (> new-price old-price)
                 (- new-price old-price)
                 (- old-price new-price)))
    (deviation (if (> old-price u0)
                 (/ (* abs-diff bps-base) old-price)
                 u0))  ;; Handle edge case of first deposit
  )
    (
      print{action:"check-max-deviation",
      data:{old:old-price,
      new:new-price,
      deviation:deviation,
      max-deviation:threshold}}
    )
    (ok (asserts! (<= deviation threshold) ERR_DEVIATION))
  )
)
```

If a withdrawal fully empties the vault, the share price resets (because `total-supply` becomes `0` after burning shares). In that case, the max deviation check can fail and prevent the transaction from completing.

Consider the following scenario:

- Current share price: 1 asset = 4 shares
- Alice is the only remaining depositor with 100 shares
- Alice attempts to withdraw all her shares

The deviation check evaluates as follows:

```
old-price = 0.25 * share-base
new-price = share-base (since total supply is now 0 after share burn)
abs-diff = 0.75 * share-base
(abs-diff * bps-base) / old-price = 300%
```

Because the share price resets when the vault is emptied, the deviation exceeds any reasonable threshold, causing the final withdrawal to revert.

Recommendation

Bypass the max deviation check when the vault is being emptied (i.e., on the last withdraw/redeem). One approach is to replace the `(if (> old-price u0)` guard with a check on `total-supply`, where `total-supply` is the hBTC total supply at that time.

Example implementation:

```
(threshold (get-max-deviation))
    (abs-diff (if (> new-price old-price)
                  (- new-price old-price)
                  (- old-price new-price)))
+   (total-supply (unwrap-panic
+ (contract-call? .hbtc-token-v1 get-total-supply)))
-   (deviation (if (> old-price u0)
+   (deviation (if (> total-supply u0)
                  (/ (* abs-diff bps-base) old-price)
-                  u0))  ;; Handle edge case of first deposit
+                  u0))  ;; Handle edge case of last withdraw/redeem
```

Note: the `(if (> old-price u0)` check is not effective for the "first depositor" case implied by the comment `;; Handle edge case of first deposit`, since the first `old-price` will be `share-base` for the first deposit. However, after the first deposit, the share price does not change due to how it is calculated, which results in an implicit deviation bypass.

[M-05] Updated Management And Performance Fees Are Applied Retroactively

Location:

- state-v1.clar#L539-L551
- controller-v1.clar#L36-L37

Description In the current design, rewards are manually synchronized by the `rewarder` role calling `controller::log-reward`. According to the protocol documentation, this operation is intended to occur once per day. At that time, management fees are applied to protocol assets, and a performance fee may be applied if profit was generated during the previous 24 hours.

The management and performance fee percentages are configured and stored in the `state-v1.fees` map. This map can be updated by the trusted admin via `state-v1::set-fees`.

However, changing fees mid-cycle can create issues depending on where the protocol is within the daily reward cycle. For example:

- If fees are changed one hour before a `controller::log-reward` call, the new fees will be applied to the entire prior 24-hour period, even though the new fee was in effect for only one hour (1/24 of the period).

Depending on whether the fee is increased or decreased, the protocol will either overcharge or undercharge for the pending rewards and management effort—effectively applying the updated fees retroactively to the previous 24 hours.

Recommendation

In many protocols with reward or interest accrual systems, fee updates are coupled with an accrual/synchronization step (i.e., the accrual function is called as part of the fee-change logic). In this case, that would mean calling `controller-v1::log-reward` from `state-v1::set-fees`, which is not possible under the current system architecture.

Additionally, tightly coupling `controller::log-reward` and `state-v1::set-fees` (in either direction) can introduce other issues, particularly when only a subset of fees is being updated.

As a workaround, modify `set-fees` to detect whether the management fee or performance fee is actually changing (i.e., the call intends to modify these values). If so, only allow the update within a short time window after the most recent `log-reward` call, determined by comparing the current time to `last-log-ts`.

This approach allows fee changes shortly after rewards are distributed (e.g., within one hour), preserving operational flexibility while preventing meaningful retroactive fee application.

[M-06] hBTC Token Is Not SIP-10 Compliant

Location:

- hbtc-token.clar#L47
- hbtc-token.clar#L51

Description

The SIP-10 standard specifies that the `transfer` function should return error codes that follow the same pattern as the built-in `ft-transfer?` and `stx-transfer?` functions:

> When returning an error in this function, the error codes should follow the same patterns as the built-in ft-transfer? and stx-transfer? functions.

error code	reason
u1	`sender` does not have enough balance
u2	`sender` and `recipient` are the same principal
u3	`amount` is non-positive
u4	`sender` is not the same as `tx-sender`

However, the hBTC token contract uses a non-standard error code for the case where `sender` is not `tx-sender` or `contract-caller`:

```
(define-constant ERR_NOT_AUTHORIZED (err u100001))

(asserts! (or (is-eq sender tx-sender)
  (is-eq sender contract-caller)) ERR_NOT_AUTHORIZED)
```

Additionally, SIP-10 requires that the `memo` field is emitted only when it is not `none`:

> the implementer has to make sure that the memo is emitted by adding a print statement if the ft-transfer? is successful and the memo is not none.

The current implementation prints the memo unconditionally:

```
(match (ft-transfer? hBTC amount sender recipient)
  response (begin
    (print memo)
```

As implemented, `hBTC` is not SIP-10 compliant, which may cause integration issues with third-party tooling and services.

Recommendation

- Replace `ERR_NOT_AUTHORIZED` with `u4`.
- Update the `transfer` function to print the memo only when it is not `none`.

An example implementation (adapted from [`sBTC`]) is shown below:

```
(define-constant ERR_NOT_AUTHORIZED (err u4))

(define-public (transfer (amount uint) (sender principal)
  (recipient principal) (memo (optional (buff 34))))
        (begin
                (asserts! (or (is-eq tx-sender sender)
    (is-eq contract-caller sender)) ERR_NOT_AUTHORIZED)
                (try! (ft-transfer? hBTC amount sender recipient))
                (match memo to-print (print to-print) 0x)
                (
    print{action:"transfer",
    data:{sender:sender,
    recipient:recipient,
    amount:amount,
    block-height:burn-block-height}}
  )
    (ok true)
        )
)
```

Note: For logging consistency, the printed event's `sender` field should be set to `sender` (not `tx-sender`).

[M-07] hBTC Public Share Burning Enables a First-Deposit DoS Attack Variant

Location:

- hbtc-token.clar#L98-L101

Description The system's primary state variables are managed within the `state` contract.

Whenever the state is updated via `state::update-state`, the current share price is recorded and later compared against the new share price in `state::check-max-deviation`. If the relative difference exceeds the `max-deviation` threshold set by the contract owner, the update reverts.

The deviation check is implemented as follows:

```
(threshold (get-max-deviation))
(abs-diff (if (> new-price old-price)
          (- new-price old-price)
          (- old-price new-price)))
(deviation (if (> old-price u0)
          (/ (* abs-diff bps-base) old-price)
          u0))  ;; Handle edge case of first deposit
```

The share price itself is computed as:

```
(define-read-only (get-share-price)
  (let (
    (net-assets (get-net-assets))
    (total-supply (unwrap-panic
      (contract-call? .hbtc-token-v1 get-total-supply)))
  )
    (if (> total-supply u0)
      (/ (* net-assets share-base) total-supply)
      share-base  ;; 1:1 for first deposit
    )
  )
)
```

At the same time, users can freely burn their shares through the `hbtc::burn` function:

```
(define-public (burn (amount uint))
  (ft-burn? hBTC amount tx-sender)
)
```

Consider the following scenario:

- The vault starts empty.
- Bob deposits `x` assets at a 1:1 share price.
- Bob immediately burns all `x` shares.

At this point, the vault holds `x` net assets but has `0` total supply, so the share price defaults to `1:1`.

When a new user attempts to deposit, the minted shares will equal the deposited assets. However, the new share price used for the max-deviation check will be computed as:

```
newPrice = (x + y) * 1e8 / y = 1e8 + x * 1e8 / y
```

Therefore, `old-price = 1e8` and `new-price = 1e8 + x * 1e8 / y`. If the absolute difference `x * 1e8 / y` exceeds the system's `max-deviation` limit (default 0.05%, max 0.2%), the deposit transaction fails.

As a result, the initial depositor can intentionally burn their shares to realistically block all subsequent deposit amounts.

Recommendation

Remove the public `hbtc::burn` function. This eliminates the vector described above and also reduces the risk of uncovered accounting discrepancies.

If a public burn function is ever needed due to market demand, deploy a new approved wrapper contract that calls `hbtc::burn-for-protocol`.

Protocols commonly suffer from donation-style attacks caused by shares/assets manipulation. To mitigate these issues, user actions should be limited to strictly necessary operations. If a donation-like action is required (e.g., depositing and then burning shares), the protocol can instead decide to perform it internally at a later time.

[M-08] Maximum and Minimum Cap Limitations May Endanger the Protocol

Location:

- state-v1.clar#L605-L630
- state-v1.clar#L641-L648
- state-v1.clar#L731-L742
- state-v1.clar#L765-L775

Description
The current hBTC protocol logic enforces fixed maximum and minimum limits on certain state configuration values. These limits cannot be bypassed by any privileged role, including the owner.

The maximum limits are as follows:

Limit	Value	Description
Max Reward	`20 bps` (0.20%)	Maximum asset reward/loss per `log-reward` call
Max Deviation	`20 bps` (0.20%)	Maximum share price deviation per update
Max Slippage	`500 bps` (5.00%)	Maximum slippage for asset trades
Max Management Fee	`54 % of bps` (0.0054%)	Maximum daily management fee (2% annualized)
Max Performance Fee	`2000 bps` (20.00%)	Maximum performance fee on profits
Max Exit Fee	`100 bps` (1.00%)	Maximum exit fee on withdrawals

Max Reserve Rate	`5000 bps` (50.00%)	Maximum reserve fund allocation rate
Max Express Fee	`200 bps` (2.00%)	Maximum express withdrawal fee
Max Cooldown	`2,592,000 seconds` (30 days)	Maximum withdrawal cooldown period
Max Block Delay	`60 blocks` (~5 minutes)	Maximum price staleness check

The minimum limits are as follows:

Limit	Value	Description
Min Update Window	`3,600 seconds` (1 hour)	Minimum time between reward updates

Note: These values represent the maximum/minimum bounds that the configuration can be set to, not the default values.

Some of these limits may unintentionally introduce operational constraints and should not be subject to hard caps. Specifically:

1. **Max Reward: Maximum asset reward/loss per `log-reward` call**
   This parameter limits how much reward or loss can be applied in a single call, constraining the rewarder role. Only the owner can set this value via `state-v1::set-max-reward`, but it cannot be increased above 0.2%.
   In extreme or unforeseen situations (e.g., black swan events or prolonged market volatility), a higher value may be required. Under such conditions, the 0.2% cap could severely limit protocol PnL updates and cause the protocol to drift out of sync with external systems, potentially leading to user-facing issues such as missed rewards or incorrectly applied losses.

2. **Max Deviation: Maximum share price deviation per update**
   This parameter also acts as a safeguard on the rewarder role. Rather than limiting reward/loss relative to total assets, it restricts the

maximum allowed change in share price per update. In certain extreme scenarios, the protocol team may need to temporarily raise this limit beyond the current 0.2% maximum.
While this constraint helps mitigate the impact of a compromised rewarder, it can also block legitimate updates when rewards or losses are large relative to the vault's net assets. For example:

- A large position liquidation could generate fees that cause a significant share price drop, potentially exceeding the deviation threshold.
- If a whale withdraws shortly before a reward distribution, net assets may drop sharply while the reward (based on active trading assets) remains unchanged, resulting in a disproportionately high reward-to-asset ratio that fails the deviation check and reverts.
  Although the deviation check provides meaningful protection, the hard cap creates a critical operational limitation. Since only the owner can modify it via `state-v1::set-max-deviation`, the hard maximum should be removed to preserve flexibility during exceptional events.

3. **Max Slippage: Maximum slippage for asset trades**
   Slippage limits should generally remain adjustable to accommodate changing market conditions. In this protocol, the team defines the slippage incurred when minting USDh; however, if external conditions require slippage above 5% (the current maximum), trading would be effectively halted until a new trading contract is deployed that does not enforce this slippage check.
   Additionally, for any non-Hermetica swap interfaces added in the future, a 5% maximum slippage cap can introduce significant risk. Hardcoded slippage has previously caused issues in other systems (e.g., 1900 ETH blocked and the ULTI.ORG protocol failure).
   Since only the owner can set this value (via `set-max-slippage` or `request-new-asset`), the cap primarily introduces limitations without meaningfully improving security.

4. **Max Block Delay: Maximum price staleness check**
   This value is currently not used in the ecosystem. Additionally, using

block counts instead of time is not recommended. The description `60 blocks (~5 minutes)` is also inaccurate given current Stacks block time variability, which can be as low as ~3 minutes and 50 seconds. The cap should be removed, and the parameter should be converted to a time-based value (e.g., seconds) rather than a block-based value.

5. **Min Update Window: Minimum time between reward updates**
During highly volatile market conditions, more frequent updates may be necessary. A 1-hour minimum update window prevents real-time reaction and may be unjustified in uncertain conditions—especially given that only the owner can set this value via `set-update-window`. More broadly, since a compromised owner can already cause catastrophic damage (e.g., adding malicious components, removing admins, granting roles, and draining reserves), restricting trusted owner actions via hard caps—when those caps can also harm protocol operations—is not advisable.

Recommendation
Remove the hard maximum caps on the configurations listed above. Additionally, replace the block-based delay with a timestamp/time-based parameter.

[M-09] Tokens With `tx-sender`-Based Authorization Are Not Fully Supported

Location:

- hermetica-interface-v1.clar#L124
- hermetica-interface-v1.clar#L161
- hermetica-interface-v1.clar#L178
- zest-interface-v1.clar#L61
- zest-interface-v1.clar#L92
- zest-interface-v1.clar#L186

Description

The current protocol implementation has limitations when interacting with tokens that require `tx-sender`-based authorization. At least two relevant token types fall into this category:

- stSTX
- STX wrappers

These tokens require `tx-sender` authorization for transfers and do not work when the caller is `contract-caller`. As a result, transfer calls will revert unless executed under an `as-contract?` context. For example, the `stSTX` token contract enforces the following check: `(asserts! (is-eq tx-sender sender) (err ERR_NOT_AUTHORIZED))`

Similarly, STX wrappers typically rely on `stx-transfer?`, which requires that the `sender` is the current `tx-sender`.

This limitation is relevant in the following cases:

- In `hermetica-interface-v1`:
  - `hermetica-mint`: when transferring leftover minting assets to the reserve
  - `hermetica-redeem`: when transferring the `redeeming-asset` to the reserve
  - `sweep`: when transferring assets to the reserve

- In `zest-interface-v1`:
  - `zest-collateral-remove`: when transferring assets to the reserve
  - `zest-borrow`: when transferring assets to the reserve
  - `sweep`: when transferring assets to the reserve

Note that `stSTX` is particularly relevant because it is supported by Zest and may therefore be used as an integration target for the current interface.

Recommendation
For `zest-collateral-remove` and `zest-borrow`, it is sufficient to use the Zest market contract's `recipient` feature, avoiding the need to handle `tx-sender`-restricted transfers directly (this is also covered in a separate finding).

For the remaining cases, if tokens such as `stSTX` or STX wrappers are ever used as collateral in the Zest interface, then `zest-interface-v1:sweep` should include an `as-contract?` clause to enable sweeping these assets.

In `hermetica-interface-v1`, all three instances listed above should add an `as-contract?` clause (with the appropriate allowance/authorization) to support `tx-sender`-restricted assets (e.g., STX wrappers).

[M-10] STX Integration Issues Due to Clarity 4 Security Constraints

Location:

- hermetica-interface-v1.clar
- zest-interface-v1.clar
- fee-collector-v1.clar
- reserve-fund-v1.clar
- reserve-v1.clar

Description
Clarity 4 introduces breaking changes to `as-contract` behavior:

- `as-contract` was removed.
- `as-contract?` was introduced and applies fully restricted token allowances by default to all passed tokens.

Throughout the codebase, there are multiple instances where token allowances are either overly permissive (e.g., using `with-all-assets-unsafe`, such as in the Zest interface) or narrowly scoped to fungible tokens only (e.g., in `reserve-v1` via `((with-ft (contract-of asset) "*" amount))`).

While the current implementation targets sBTC, USDh, and sUSDh, project documentation indicates that STX may be supported in the future.

A key constraint arises with certain STX "wrapper" tokens that transfer native STX and do not have a backing fungible token. In such cases, transfers require `(with-stx amount)` rather than `with-ft`. Therefore, the code must differentiate between STX transfers and standard FT transfers.

The intent of an STX wrapper is to allow STX to be handled like other fungible tokens without requiring special-case logic. This was feasible prior to Clarity 4. However, with the `with-stx` distinction, a simple wrapper that only calls `stx-transfer?` becomes constrained because `stx-transfer?` requires the sender to be the current `tx-sender`.

Some parts of the codebase currently use `with-all-assets-unsafe`, which permits both STX and FT transfers but provides no meaningful security controls. This applies to the Zest interface (as noted in **Loose Token Allowances On Zest Market Interface Calls**) and the Hermetica interface (as noted in **Hermetica Interface Mint Asset Transfer Restriction Can Be Improved**).

In contrast, the token-holding contracts—`reserve`, `reserve-fund`, and `fee-collector`—cannot support STX outflows because `((with-ft (contract-of asset) "*" amount))` restricts outflows to fungible tokens only.

Recommendation

Supporting STX while preserving Clarity 4 security improvements can be achieved in several ways.

When integrating with Zest v2, a dedicated STX wrapper contract is required. That wrapper can also be used to detect whether a withdrawal requires STX allowances. The following pattern can be applied at each transfer site that may involve STX:

```
(let ((asset-contract (contract-of asset)))
  (if (is-eq asset-contract ZEST-STX-WRAPPER-CONTRACT)
    (as-contract? ((with-stx amount))
        (try! (contract-call? asset transfer amount tx-sender account none)))
    (as-contract? ((with-ft asset-contract "*" amount))
        (try! (contract-call? asset transfer amount tx-sender account none))))))
```

This approach adds overhead but preserves existing entry points and logic, provided it is consistently applied wherever STX may be transferred.

Alternatively, new entry points can be introduced that explicitly target the STX wrapper (e.g., `zest-deposit-stx`). However, token-accumulating contracts (e.g., `reserve`) would still need a mechanism to distinguish STX from standard FT allowances.

Because STX integration may be required, the `reserve` contract (and any other contract that may hold STX) must be updated regardless. Contracts that may custody STX should either implement the conditional logic above or adopt a less restrictive compromise that allows both FT and STX outflows:

```
((with-ft (contract-of asset) "*" amount) (with-stx amount))
```

This enables STX support while still providing some limitation on outflows. It is preferable to `with-all-assets-unsafe`, though only marginally.

In summary, there are two main paths:

1. If STX support is planned, the current token-holding contracts do not support STX outflows and require changes (notably `fee-collector-v1`, `reserve-fund-v1`, and `reserve-v1`).
2. If the Zest or Hermetica interfaces must support STX, they should either retain `with-all-assets-unsafe` (not recommended) or be updated using the conditional approach above. Additionally, the recommendations in **Tokens With tx-sender Type Authorization Are Not Fully Supported** must be implemented to fully enable STX integration.

8.3. Low Findings

[L-01] Incomplete Blacklist Flow

Location:

- blacklist-v1.clar#L89-L94
- blacklist-v1.clar#L96-L101
- vault-v1.clar#L137-L138
- vault-v1.clar#L185-L205

Description
The blacklist mechanism in `sUSDh` acts as an access-control guard against bad actors and sanctioned entities. Depending on whether an address is softly or fully blacklisted, it is either denied access to staking-related calls or has its balance fully frozen.

In contrast, the hBTC blacklist mechanism is only partially enforced. The withdrawal/redeem flow performs blacklist checks during the `init-*` functions, but does not re-check blacklist status during the settlement step.

Additionally, hBTC transfers do not fully enforce blacklist restrictions for the sender and/or recipient.

Recommendation
Fully implement and consistently enforce the blacklist functionality across all relevant flows (including settlement and transfers), aligning the behavior with `sUSDh`.

[L-02] Missing Reset Mechanism for Custom Parameters

Location:

- state-v1.clar#L558
- state-v1.clar#L585

Description

In the `state-v1` contract, principals use the default exit fee and cooldown settings unless custom values are configured.

The fee setter can assign custom exit fees to specific principals, and the admin can assign custom cooldowns. These values are stored in the `custom-exit-fee` and `custom-cooldown` maps.

However, the contract does not provide a way to revert a principal's custom parameters back to the defaults. Even if a custom value is manually set to match the current default, it will not track future default updates. As a result, the principal may continue using an outdated value if the default changes later.

Recommendation

Add `reset-custom-exit-fee` and `reset-custom-cooldown` functions that remove entries from the `custom-exit-fee` and `custom-cooldown` maps, allowing principals to fall back to the default parameters.

[L-03] Management Fee Overstated by Using Total Assets

Location:

- controller-v1.clar#L37

Description In the current design, rewards are manually synchronized by the `rewarder` role calling `controller::log-reward`. Per the protocol documentation, this operation is intended to be performed once per day.

During this process, the management fee is calculated as a percentage of `total-assets`:

```
(mgmt-fee (/ (* (get mgmt-fee fees) total-assets) bps-base pct-base))
```

This is also reflected in the project documentation:

> Management fee:

- Timing: daily
- Target: % of total-assets

However, `total-assets` includes pending claims, pending fees, and the pending reserve fund. Charging the management fee on these components is excessive and/or incorrect, as it overstates the management fee.

As a result, users who remain in the protocol implicitly receive lower net rewards due to assets associated with users who are exiting the protocol.

Recommendation Replace `total-assets` with net assets (retrieved via `state::get-net-assets`) when calculating the management fee. For a more efficient implementation, the net assets value can be added to the `state::get-reward-state` getter.

[L-04] Missing Explicit Validation for `is-express` Parameters

Location:

- state-v1.clar#L539
- state-v1.clar#L562
- state-v1.clar#L571

Description
When initiating a redemption or withdrawal, users must provide a value for the `is-express` parameter.

Per the protocol documentation, `is-express` allows users to opt into a shorter cooldown period in exchange for a higher exit fee. However, this relationship is not enforced on-chain. For example, an admin could configure a shorter cooldown with a lower exit fee, which would contradict the intended economic design.

This inconsistency may result in unexpected or unfair withdrawal conditions for users.

Recommendation
Add validation logic in the state contract to enforce a consistent relationship between cooldown periods and exit fees. Specifically, update the `state` contract to ensure:

- When setting the regular exit fee, it is less than (or equal to, if intentionally allowing parity) the express exit fee.
- When setting the regular cooldown, it is greater than (or equal to) the express cooldown.
- When setting the express cooldown, it is less than (or equal to) the regular cooldown.

Note: Allowing equality may be appropriate if the intent is to temporarily remove the distinction between express and regular behavior.

[L-05] Vault `withdraw-many` Functionality Is Brittle

Location:

- vault-v1.clar#L166-L181

Description The current vault exit logic consists of multiple steps:

1. Depositors initiate a withdrawal by calling either `init-withdraw` or `init-redeem`, which creates a claim.
2. A privileged manager role then funds the specific claim.
3. Anyone can call `withdraw` for a specific claim, which correctly sends funds to the claim creator.

To optimize step (3), since anyone can withdraw on behalf of others, the `withdraw-many` function was introduced to facilitate up to 1,000 withdrawals in a single call.

However, the current implementation of `withdraw-many` is not well-suited for bulk withdrawals because it reverts the entire transaction if any underlying claim has already been withdrawn.

Consider the following scenario:

- A large trade occurs and 200 claims are funded.
- A keeper submits a `withdraw-many` transaction including those 200 claims.
- At the same time, normal users (who also see the funded claims) submit withdrawals independently.
- If any of those claims are withdrawn before the keeper's `withdraw-many` executes, the entire `withdraw-many` transaction reverts. This forces the keeper to retry with smaller batches over a longer period, since organic withdrawals may continue to occur.

Recommendation Modify `withdraw-many` to use a `map`-style approach instead of a `fold`, returning a list of responses from each `withdraw` call.

This would allow the function to process all claims and report per-claim failures without reverting the entire execution when a single withdrawal fails.

[L-06] hBTC Token Name Should Not Be Changeable

Location:

- hbtc-token.clar#L14-L15
- hbtc-token.clar#L71-L76

Description
The `hBTC` token contract currently allows the underlying fungible token name to be modified.

A token's name and symbol—together with the contract address—are commonly used by external integrators and price aggregators for display purposes. As a result, making these values changeable can lead to significant user confusion if they are updated after integration.

While SIP-10 does not explicitly prohibit changing these fields, it is generally expected that a fungible token's name and symbol remain immutable after launch.

In the current `hbtc-token` contract, the token name can be changed via the `set-token-name` function.

Recommendation
Remove the `set-token-name` function from the `hbtc-token` contract. Additionally, replace the `token-name` and `token-symbol` data variables with constants, since these values are not intended to change.

[L-07] Reward Distribution Should Not Be Allowed After the Vault Has Been Emptied

Location:

- controller-v1.clar#L29-L68
- state-v1.clar#L500

Description
In the current implementation, rewards are manually synchronized by the `rewarder` role via `controller::log-reward`, which—per the protocol documentation—is expected to be called once per day.

Because rewards are distributed daily, edge cases can arise and should be accounted for.

One such case occurs when the vault is emptied mid-period, yet there are still rewards accrued for that period. In this scenario, `log-reward` would technically attribute rewards to users, but since all users have already exited the system, distributing rewards has no practical meaning.

Additionally, when the vault is empty, calling `log-reward` (e.g., with `reward = 0` and `is-positive = true` to update `last-log-ts`) may revert in some cases.

This revert occurs because, even with zero assets in the vault, `state::update-state` still proceeds with values set to `0`. During event emission, the following computation is performed:

```
return: (/ (* (get reward data) bps-base pct-base) init-total-assets),
```

When `init-total-assets` is `0`, this results in a `DivisionByZero` error. Consequently, the reward distribution process fails when the vault is empty (i.e., no users hold hBTC and no assets remain in the vault).

A related edge case arises when the vault has zero net assets but some pending (uncommitted) assets. In this situation, `net-assets` is `0` while `total-assets` is positive, which avoids the division-by-zero.

However, if a positive reward is distributed under these conditions, the vault's net assets increase despite there being no shares to represent ownership of the reward. This could allow the next depositor to capture unowned profit, or potentially trigger a max deviation failure depending on the reward and deposit amounts.

Overall, the nuances and corner cases introduced by allowing `log-reward` to be called when the vault is empty outweigh any benefit of permitting it.

Recommendation
Since the system should not realistically require reward distribution when `total-supply` is `0`, explicitly forbid calling `log-reward` in this situation.

Specifically, in `log-reward`, assert that the hBTC token total supply is non-zero; otherwise, revert.

[L-08] Positive Slippage Remains in the Hermetica Interface Contract After a Mint

Location:

- hermetica-interface-v1.clar#L96-L124

Description In the `hermetica-interface-v1` interface contract, the `hermetica-mint` function allows a trader to convert approved assets into `USDh`.

From a slippage perspective, the function accepts a `slippage-tolerance-input` parameter. This value is capped at the maximum allowed slippage (5%) and then forwarded to the underlying `minting-auto-trait::mint` call, where the slippage check is ultimately enforced.

The auto-minting contract pulls the required `amount-asset` tokens from the `hermetica-interface-v1` caller and then transfers the resulting `USDh` amount (after slippage is applied) back to the caller.

However, the interface contract incorrectly assumes that the `amount-asset` retrieved from the reserve contract will be fully consumed by the `minting-auto-trait::mint` call.

If the amount is underestimated off-chain, the mint fails during the minting contract transfer step (see minting contract transfer):

```
(try! (contract-call? minting-asset transfer
  (- amount-asset-required fee-amount) contract-caller (unwrap-panic (get custody-address stat
```

If the amount is overestimated, the excess asset tokens remain in the `hermetica-interface-v1` contract. These funds remain stranded until the trader performs a `minting-auto-trait::sweep` call to recover them.

Recommendation

After the `minting-auto-trait::mint` call, query the interface contract's balance of the `minting-asset-trait` token. If the balance is non-zero, transfer the remaining tokens back to the reserve.

Note: This approach also effectively acts as a `sweep` for any assets previously left in the contract. If this behavior is not desired, perform a pre-/post-balance check to calculate the exact amount consumed and return only the newly leftover tokens.

[L-09] Race Condition During Fund Transfers Due to Rewarder Role Ambiguity

Location:

- controller-v1.clar#L79

Description In the current design, the rewarder role is authorized to both (1) log PNL/rewards and (2) move pending fees and reserve funds. This dual capability raises two concerns:

1. It is not documented. The current documentation only states: `Rewarder: Rewarders can log-rewards in .controller`.
2. Allowing the rewarder to fund transfers introduces a potential race condition with the manager.

The availability of funds in the reserve contract depends on trading activity managed by the trader role, which generates yield using the vault's capital. When a user initiates a withdrawal, the trader releases the required liquidity from trading positions to satisfy the withdrawal.

At that point, the manager can fund the withdrawal immediately (even if the cooldown period has not yet expired), or any user can fund it once the cooldown ends.

Consider the following scenario:

- All vault assets are currently locked in trading positions.
- Pending fees = 50, pending RF = 50.
- Alice initiates a withdrawal of 100 assets.
- The trader releases 100 assets.
- Simultaneously, the manager attempts to fund Alice's claim while the rewarder calls `controller::fund-transfers`.

Because there is no coordination between the manager and rewarder roles, a race condition can occur in which both roles attempt to act on the same funds concurrently.

Recommendation

Restrict access to the `fund-transfers` function by removing the rewarder role and granting access exclusively to the manager role. This allows the manager to properly coordinate the destination of released funds as needed.

Additionally, clearly document which roles are permitted to call `controller::fund-transfers`.

[L-10] Admin Should Not Be Able To Change Price Staleness Threshold

Location:

- state-v1.clar#L643

Description The current role documentation states that the admin role is:

> Admin: The admin is the second highest governance role. Admins can set permissions for day-to-day operations of the vault in the state contract without having critical governance access. A compromise of the admin cannot lead to loss of funds. The admin is a cold single-sig without a timelock.

However, an admin can indirectly cause loss of funds to third parties by being able to configure the protocol's staleness check via `state-v1::set-block-delay`.

While the current codebase snapshot does not use the block-delay logic, any future implementation that relies on it could be impacted. Specifically, setting an excessively high staleness threshold may allow the protocol to accept stale prices, which can lead to indirect loss of funds.

Recommendation Update `state-v1::set-block-delay` so it is only callable by the system owner.

Additionally, change the configuration to be time-based rather than block-based, since block time is not a reliable measure of elapsed time and can vary significantly (see average block times).

[L-11] Exit Fee Transfers Should Not Be Bound To Vault Status

Location:

- fee-collector-v1.clar#L23

Description The `fee-collector-v1` contract accumulates exit fees paid by vault users. The permissionless `withdraw` function is then used to transfer the accumulated fees to the configured fee address.

`withdraw` performs an authorization check by calling `state-v1::check-transfer-auth`:

```
(define-read-only (check-transfer-auth (asset principal))
  (begin
    (try! (check-is-vault-active))
    (asserts! (get-transfer-active) ERR_TRANSFER_DISABLED)
    (check-is-asset asset)
  )
)
```

`check-transfer-auth` verifies that: (1) the asset is approved, (2) transfers are enabled, and (3) the vault is currently active.

However, the vault-active requirement is not relevant in this context. The exit fee has already been collected and should be transferable regardless of the vault's operational status. As a result, tying fee withdrawals to the vault being active can unnecessarily block fee transfers.

Recommendation Replicate the relevant checks from `state-v1::check-transfer-auth` while omitting the `check-is-vault-active` call.

Implementation-wise, add a helper in `state-v1`, such as `check-are-transfers-active`, that reverts with `ERR_TRANSFER_DISABLED` when transfers are disabled. Then, in `fee-collector-v1`, call `check-are-transfers-active` followed by `check-is-asset` before executing the transfer.

[L-12] Missing Transfer Authorization Check in Reserve Fund Transfer

Location:

- reserve-fund-v1.clar#L19-L28

Description The `reserve-fund-v1` contract accumulates funds intended to cover PnL losses.

Via the `transfer` function, approved protocol contracts can transfer any asset passed to the function. However, the function does not verify that the provided asset is approved, nor does it check the vault's active state or whether transfers are currently enabled.

Adding these checks would provide an additional security layer for the protocol.

Recommendation In `reserve-fund-v1::transfer`, call `state-v1::check-transfer-auth` for the provided asset contract.

[L-13] Max Slippage Can Be Set for Arbitrary Assets

Location:

- state-v1.clar#L765-L775

Description

New assets are introduced into the codebase (deactivated by default) via a `request-new-asset` call and are later activated via an `activate-asset` call, which can only be executed by the protocol owner. Both steps are required, and the call order is mandatory.

In the `state-v1` contract, the owner can also update the slippage of an existing asset by calling `set-max-slippage`. However, there is no validation that the provided asset address corresponds to an asset that has been previously requested/registered in the system. As a result, `set-max-slippage` can be called with an arbitrary principal, creating or updating an entry with invalid default values.

This occurs because `set-max-slippage` does not verify that the retrieved entry `(entry (get-asset address))` represents a real asset. Notably, `get-asset` does not revert when the asset is missing; instead, it returns a default struct with falsy/empty values:

```
(define-read-only (get-asset (address principal))
  (default-to

        { active: false, ts: none, price-feed-id: 0x, token-base: u0, max-slippage: u0, is-st
    (map-get? assets { address: address })
  )
)
```

Because `get-asset` defaults to these values, setting a max slippage for any arbitrary token will succeed and can introduce an erroneous map entry such as:

```
{
   active: false, ts: none, price-feed-id: 0x, token-base: u0, max-slippage: <valid-slippage-a
```

Recommendation

In `state-v1::set-max-slippage`, add a check that the asset entry exists (e.g., ensure `ts` is not `none`) and revert otherwise, similar to the validation performed in `activate-asset`:

```
(ts (unwrap! (get ts entry) ERR_NO_ENTRY))
```

[L-14] Security Enhancements in Case of Ownership Compromise

Location:

- hq-hbtc-v1.clar
- state-v1.clar

Description
In the event of an ownership key compromise, the protocol already includes several security measures to mitigate risk.

Additional improvements could further strengthen the protocol by giving users more time and opportunity to exit the system during a crisis.

Recommendation
Introduce a two-step mechanism (with a delay and a cancellation option) for all owner-gated operations, especially those related to role management.

For example:

- Adding an admin is already a two-step procedure. In the event of a key compromise, a malicious owner cannot immediately add new admins, but can still remove existing ones.
- Implement a two-step removal process as well, including a delay between initiating and finalizing the removal, and allow cancellation of both pending additions and removals.

Apply the same two-step add/remove mechanism to the guardian and keeper roles. Currently, adding these roles does not follow a two-step process; therefore, in extreme cases, a compromised owner could immediately add malicious keepers or guardians.

With role changes protected by delays and cancellation, an ownership compromise would be less impactful because competing actions (trusted vs. malicious) could be canceled, preventing roles from changing hands.

Finally, apply a similar “pending change + delay + cancel” pattern to owner-changeable configuration parameters. This would ensure that all configuration changes require a two-step process and can be canceled in a single call, helping keep the system stable while users exit.

[L-15] Unnecessary Token Allowance on Zest Operations With No Outflows

Location:

- zest-interface-v1.clar#L60
- zest-interface-v1.clar#L89

Description

In the `zest-interface-v1` contract, several Zest v2 protocol functions are invoked within an `as-contract` context even though they do not require transferring any tokens from the caller. Instead, these operations result in tokens flowing *to* the caller.

These cases include:

- Removing collateral: the caller receives tokens.
- Borrowing: the caller receives tokens.

In `zest-collateral-remove`:

```
(let ((remaining (try! (as-contract? ((with-all-assets-unsafe)) (try!
  (contract-call? market collateral-remove asset amount (some current-contract) none))))))
```

In `zest-borrow`:

```
(try! (as-contract? ((with-all-assets-unsafe)) (try!
  (contract-call? market borrow asset amount (some current-contract) none))))
```

In both cases, `as-contract?` is used with a `with-all-assets-unsafe` allowance, granting unnecessary and unrestricted token access despite there being no token outflow from the caller.

Recommendation

Update the `as-contract?` allowance to an empty allowance `()` in both `zest-borrow` and `zest-collateral-remove`. Since these functions involve inflows from the called interface rather than outflows, no token allowance is required.

[L-16] Repaying Zest Debt May Leave Dust in the Interface

Location:

- zest-interface-v1.clar#L118

Description When a repayment on a Zest loan is initiated via the `zest-interface-v1::zest-repay` function, an `amount` of tokens is first transferred from the reserve contract, and the repayment is then executed using that amount.

In theory, repayment can be performed on behalf of another user. In that case, the `market::repay` function may cap the repayment amount and repay only up to the outstanding debt, rather than reverting. This is a necessary feature to avoid denial-of-service scenarios for external integrators (such as Hermetica), where an attacker could pre-repay a minimal amount (e.g., 1 micro unit) to cause subsequent repayments to revert.

However, the current implementation of the `zest-repay` function does not account for the possibility that a portion of the transferred funds may remain unused. As a result, leftover tokens ("dust") may accumulate in the interface contract, potentially requiring a sweep at a later time.

Recommendation If the `repaid-amount` is less than the asset `amount` transferred from the `reserve`, return the leftover funds to the reserve.

[L-17] Loose Token Allowances on Zest Market Interface Calls

Location:

- zest-interface-v1.clar#L36
- zest-interface-v1.clar#L118
- zest-interface-v1.clar#L145
- zest-interface-v1.clar#L168

Description

In the `zest-interface-v1` contract, several calls are made to the Zest `market` (and `vault`) contracts that expect tokens to be withdrawn/consumed during execution.

In `zest-collateral-add`:

```
(let ((total (try! (as-contract? ((with-all-assets-unsafe)) (try!
  (contract-call? market collateral-add asset amount none))))))
```

In `zest-repay`:

```
(let ((repaid-amount (try! (as-contract? ((with-all-assets-unsafe)) (try!
  (contract-call? market repay asset amount (some current-contract)))))))
```

In `zest-deposit`:

```
(received (try! (as-contract? ((with-all-assets-unsafe)) (try!
  (contract-call? vault deposit amount min-shares reserve)))))
```

In `zest-redeem`:

```
(received (try! (as-contract? ((with-all-assets-unsafe)) (try!
  (contract-call? vault redeem shares min-amount reserve)))))
```

In each case, the required amounts are first transferred from the `reserve` contract and are then expected to be consumed by the subsequent call.

However, using `with-all-assets-unsafe` within the `as-contract?` asset restriction permits arbitrary asset removals, even though the exact (maximum) amounts to be withdrawn are known for these operations.

Recommendation
For each of the four functions (`zest-collateral-add`, `zest-repay`, `zest-deposit`, and `zest-redeem`), replace `with-all-assets-unsafe` with an exact `with-ft` allowance restriction to improve protocol safety.

Note that this change introduces a semantic difference. Without the change, the transferred amount could be as low as `u1` (the minimum to avoid a revert), and the semantics would be:

- an amount of assets to be taken from the reserve before executing the operation, allowing any pre-existing token balance in `zest-interface-v1` to be used by the operation (potentially avoiding the need for a `sweep` call), although this scenario is unlikely.

If the proposed change is applied, the semantics become:

- an amount of assets to be taken from the reserve before executing the operation **and** the maximum amount of tokens that the specific operation is allowed to consume. As a result, calls that rely on consuming leftover assets already held by `zest-interface-v1` would no longer be possible, and intended `sweep` calls would be required to manage such balances.

Finally, special care is required for STX wrapper contracts: they cannot use `with-ft` because, at the implementation level, they transfer raw STX, which requires a `with-stx` allowance.

8.4. QA Findings

[QA-01] Unnecessary External Call for `is-standard` Verification

Location:

- state-v1.clar#L533

Description The `state-v1::set-fee-address` function accepts a principal to be used as the fee recipient. It currently calls `hq-hbtc-v1::check-is-standard` to confirm that the provided principal is a standard principal.

This external contract call is unnecessarily costly, as `check-is-standard` only performs a simple `is-standard` validation that can be executed locally within the same contract.

Recommendation Remove the external `check-is-standard` call and replace it with an inline `is-standard` check.

Example implementation:

```
+(define-constant ERR_NOT_STANDARD (err u102017))

(define-public (set-fee-address (address principal))
  (begin
    (try! (contract-call? .hq-hbtc-v1 check-is-owner contract-caller))
-   (try! (contract-call? .hq-hbtc-v1 check-is-standard address))
+   (standard-ok (asserts! (is-standard address) ERR_NOT_STANDARD))
    (print { action: "set-fee-address", user: contract-caller, data: { old:
      (get-fee-address), new: address } })
    (ok (var-set fee-address address))
  )
)
```

[QA-02] Current Owner Can Claim Ownership Multiple Times

Location:

- hq-hbtc-v1.clar#L248-L258

Description The protocol owner is changed using a standard two-step ownership transfer process: the current owner first requests a new owner via `hq-hbtc-v1::request-new-owner`, and the proposed owner then finalizes the transfer via `hq-hbtc-v1::claim-owner`.

Unlike a typical two-step transfer, this implementation also enforces an activation period that must elapse after the request before the new owner can claim ownership.

However, `claim-owner` does not verify that ownership has not already been claimed. As a result, the current owner can call `claim-owner` multiple times, producing inconsistent or misleading event logs that may confuse off-chain monitoring systems.

Recommendation In `hq-hbtc-v1::claim-owner`, add a check to ensure the current owner is not already the next owner (i.e., prevent claiming if the transfer has already been completed).

[QA-03] Current Owner Can Request Next Ownership to Himself

Location:

- hq-hbtc-v1.clar#L237-L246

Description Changing the protocol owner follows a standard two-step ownership transfer process: the current owner first requests a new owner via `hq-hbtc-v1::request-new-owner`, and the proposed owner then finalizes the transfer by calling `hq-hbtc-v1::claim-owner`.

The current implementation does not validate that the requested new owner principal is different from the current owner. As a result, the current owner can request ownership to be transferred to himself, effectively "resetting" the ownership request.

This scenario is most likely to occur due to operator error by a trusted party and, at worst, could confuse third-party monitoring systems by producing ambiguous ownership-transfer events.

Recommendation Add a check in `hq-hbtc-v1::request-new-owner` to ensure the proposed next owner `address` is not equal to the current owner address.

[QA-04] Incorrect Reuse of `ERR_NOT_OWNER` Code When Claiming Ownership

Location:

- hq-hbtc-v1.clar#L253

Description

When the next-in-line owner calls `hq-hbtc-v1::claim-owner` to claim ownership, the function checks that the caller matches the recorded `next-address`; otherwise, it reverts:

```
(asserts! (is-eq next-address contract-caller) ERR_NOT_OWNER)
```

However, the revert uses the `ERR_NOT_OWNER` error code, which elsewhere in the codebase indicates that an action was not performed by the **current** owner.

In this context, that error code is misleading: the failure occurs because the caller is **not the designated next owner**, not because the caller is not the current owner.

Recommendation

Introduce a dedicated `ERR_NOT_NEXT_OWNER` error code constant and use it in `hq-hbtc-v1::claim-owner`.

[QA-05] Incorrect Event Action Name for `request-new-admin` Function

Location:

- hq-hbtc-v1.clar#L266

Description Relevant functions in the codebase emit events via the `print` function following a consistent pattern, where the `action` field of the emitted tuple matches the name of the function emitting it.

However, `hq-hbtc-v1::request-new-admin` emits an event with the action `request-admin-update`, which does not match the function name.

Recommendation To maintain codebase uniformity and adhere to the established event emission pattern, update the `print` statement so the `action` field is set to `request-new-admin`.

[QA-06] Optimize Double Timestamp Retrieval

Location:

- state-v1.clar#L445

Description The `update-last-log-ts` function updates `last-log-ts` with the latest timestamp retrieved via `get-current-ts` from the previous block information.

At present, the function calls `get-current-ts` twice—once to update the value and once to log it—resulting in unnecessary overhead.

Recommendation

Retrieve the timestamp once and reuse it for both the update and the log. Example implementation:

```
(define-private (update-last-log-ts)
  (let (
    (current (get-current-ts))
  )
    (print { action: "update-last-log-ts", data: { old:
      (get-last-log-ts), new: current } })
    (var-set last-log-ts current)
  )
)
```

[QA-07] Hardcode Constants Instead of Computing at Runtime

Location:

- controller-v1.clar#L13-L15
- state-v1.clar#L45-L47
- vault-v1.clar#L19-L20
- hermetica-interface-v1.clar#L21

Description
The codebase uses `(pow u10 uN)` to define exponent-based constants (e.g., `1eN`). While functionally correct, using `pow` introduces unnecessary runtime cost: it adds a fixed compute overhead during contract deployment (in addition to the cost of the constant itself), and it also increases runtime costs on subsequent contract loads.

This approach slightly increases block runtime usage at deployment time, as shown below:

```
>> ::get_costs (define-constant usdh-base (pow u10 u8))
+----------------------+----------+------------+------------+
|                      | Consumed | Limit      | Percentage |
|----------------------+----------+------------+------------|
| Runtime              | 442      | 5000000000 | 0.00 %     |
|----------------------+----------+------------+------------|
| Read count           | 0        | 15000      | 0.00 %     |
|----------------------+----------+------------+------------|
| Read length (bytes)  | 0        | 100000000  | 0.00 %     |
|----------------------+----------+------------+------------|
| Write count          | 0        | 15000      | 0.00 %     |
|----------------------+----------+------------+------------|
| Write length (bytes) | 0        | 15000000   | 0.00 %     |
+----------------------+----------+------------+------------+
none
>> ::get_costs (define-constant usdh-base u100000000)
+----------------------+----------+------------+------------+
|                      | Consumed | Limit      | Percentage |
|----------------------+----------+------------+------------|
|----------------------+----------+------------+------------|
| Runtime              | 256      | 5000000000 | 0.00 %     |
|----------------------+----------+------------+------------|
| Read count           | 0        | 15000      | 0.00 %     |
|----------------------+----------+------------+------------|
| Read length (bytes)  | 0        | 100000000  | 0.00 %     |
|----------------------+----------+------------+------------|
| Write count          | 0        | 15000      | 0.00 %     |
|----------------------+----------+------------+------------|
| Write length (bytes) | 0        | 15000000   | 0.00 %     |
+----------------------+----------+------------+------------+
none
```

Recommendation

Where possible, replace `(pow u10 uN)` with the equivalent hardcoded `uint` constant to avoid unnecessary runtime overhead.

```
- (define-constant usdh-base (pow u10 u8))
+ (define-constant usdh-base u100000000) // 1e8
```

[QA-08] Zest Interface Contract Can Be Slightly Improved

Location:

- zest-interface-v1.clar

Description
The `zest-interface-v1` contract serves as a wrapper around Zest v2 market and vault operations. While the contract is generally well written, a few minor improvements could enhance consistency and reduce unnecessary logic:

1. The `reserve` constant is defined to reference the `reserve-v1` contract principal: `(define-constant reserve .reserve-v1)`. However, the contract inconsistently alternates between using this constant and directly referencing the hardcoded contract principal.

   Examples where the hardcoded contract principal is used (and where the constant could be reused, noting that there are cases where it cannot be reused):

   - zest-interface-v1.clar:L77
   - zest-interface-v1.clar:L113
   - zest-interface-v1.clar:L189
   - zest-interface-v1.clar:L231

   The only two examples where the `reserve` constant is used:

   - zest-interface-v1.clar#L252-L253

   To improve uniformity, either use the constant everywhere it is applicable, or remove it and consistently use the direct `reserve-v1` contract principal.

2. Zest vault interaction functions—`zest-deposit`, `zest-redeem`—as well as collateral addition via `zest-collateral-add` do not require price feed

updates. As a result, the `price-feed-*` handling in these functions is redundant.

Removing this logic would simplify the external interface and slightly reduce execution fees.

3. Consider adding event fields to indicate whether a price update was provided.

   For functions where price feeds may be supplied, consider including in the printed event `data` whether each feed was provided. This would improve off-chain monitoring without materially increasing execution size.

   Example event fields:

   ```
   price-feed-1: (if (is-some price-feed-1) true false),
   price-feed-2: (if (is-some price-feed-2) true false)
   ```

Recommendation
Apply the changes above to improve consistency, simplify the interface, and enhance observability of the Zest interface contract.

[QA-09] Excessive Price Feed Updates in Trading Interface

Location:

- trading-v1.clar

Description The `trading-v1` contract is a trading wrapper that calls functions from the `hermetica-interface-v1` and `zest-interface-v1` contracts.

The `zest-interface-v1` contract allows updating Pyth prices for up to two feeds via optional parameters. Updating a Pyth price feed is costly even if fees have already been processed in the same block, because the feed data must still be parsed.

Currently, `trading-v1` does not check whether a prior call to the Zest interface has already updated the price feeds. As a result, it repeatedly passes price feed data and triggers redundant updates across chained Zest interactions.

Breakdown of price feed updates per `trading-v1` function:

- `zest-open`: 1 feed update (expected)
- `zest-close`: 1 feed update (expected)
- `zest-open-add`: 2 duplicate feed updates (once in `zest-collateral-add`, and again in `zest-open`)
- `zest-close-remove`: 2 duplicate feed updates (once in `zest-close`, and again in `zest-collateral-remove`)
- `zest-open-add-deposit`: 3 duplicate feed updates (in `zest-deposit`, `zest-collateral-add`, and `zest-open`)
- `zest-close-remove-redeem`: 3 duplicate feed updates (in `zest-close`, `zest-collateral-remove`, and `zest-redeem`)

Each redundant feed update significantly increases execution costs and should be avoided.

Recommendation For functions that currently trigger duplicate updates, pass `price-feed-1` and `price-feed-2` only to the first internal call where they are accepted. For subsequent internal calls in the same flow, pass `none` for these parameters to prevent redundant price feed parsing and updates.

[QA-10] Trading Interface: Ambiguous Function Naming Convention

Location:

- trading-v1.clar#L89-L107
- trading-v1.clar#L177-L200

Description

The `trading-v1` contract implements multiple functions that interact with both the Zest and Hermetica interfaces. These functions are generally named after the operations they perform, in the order those operations occur.

Examples of function names and the corresponding sequence of operations:

- `zest-open`: opens a position on Zest by borrowing `USDh` from the Zest market and staking it in Hermetica.
- `zest-close`: closes a position on Zest by unstaking `sUSDh` from Hermetica and repaying `USDh` to the Zest market.
- `zest-close-remove`: (1) closes a Zest position and (2) removes collateral from the Zest market.
- `zest-close-remove-redeem`: (1) closes a Zest position, (2) removes collateral from the Zest market, and (3) redeems Zest vault shares.

While the functions above follow a consistent naming convention that reflects the operation order, `zest-open-add` and `zest-open-add-deposit` do not:

- `zest-open-add`: (1) adds collateral to the Zest market and (2) opens a Zest position.
- `zest-open-add-deposit`: (1) deposits into the Zest vaults, (2) adds collateral to the Zest market, and (3) opens a Zest position.

Recommendation

Rename the functions (and their corresponding event `action` values) to reflect the actual order of operations:

- Rename `zest-open-add` to `zest-add-open`.
- Rename `zest-open-add-deposit` to `zest-deposit-add-open`.

[QA-11] Trading Interface Can Be Optimized

Location:

- trading-v1.clar#L29-L69

Description The `trading-v1` contract implements several functions that support both the Zest and Hermetica interfaces. These functions perform multiple logical operations and ultimately call either `zest-open` or `zest-close`.

Specifically:

- `zest-open-add` and `zest-open-add-deposit` call `zest-open`
- `zest-close-remove` and `zest-close-remove-redeem` call `zest-close`

All of these functions are public, including `zest-open` and `zest-close`, and therefore include authorization and input validation checks.

This call pattern introduces unnecessary execution overhead because checks performed in the calling public functions are repeated again inside `zest-open` and `zest-close` (e.g., `check-is-trader` and amount validations).

Recommendation Refactor `zest-open` into two functions:

- Introduce an internal function (e.g., `zest-open-internal`) that contains the core logic for calling `zest-interface-v0-2` and `hermetica-interface-v1`, but does not perform authorization/amount checks or emit print events.
- Keep `zest-open` as the public entry point, performing the required authorization and amount checks, then delegating execution to `zest-open-internal`. After the internal call, emit `zest-open`-specific print events.
- Update other local functions that currently call `zest-open` to call `zest-open-internal` instead, avoiding redundant checks.

Apply the same approach to `zest-close`. This removes duplicated validation overhead and ensures print events are emitted from the public `zest-open` and `zest-close` functions, which are currently missing them.

[QA-12] Vault Deposit Cap Consideration

Location:

- vault-v1.clar#L85

Description The current vault implementation enforces a maximum, pre-set deposit cap:

```
(asserts! (<= (+ (get total-assets state) assets)
  (get deposit-cap state)) ERR_DEPOSIT_CAP_EXCEEDED)
```

In this case, the cap is checked against `total-assets`, which includes pending claims (amount reserved for users' pending claims), pending fees (amount reserved for protocol fees), the pending reserve fund (amount to be transferred to the reserve fund), and net assets (the only stationary funds within the system).

As a result, applying the deposit cap to the entire `total-assets` value may be overly restrictive if the intent is to limit only user-owned protocol funds.

Recommendation Depending on the protocol's business logic, consider applying the cap to net assets plus pending claims (i.e., funds that are user-bound). Otherwise, explicitly acknowledge and accept the current behavior.

[QA-13] Management Fee Max Amount Implementation – Documentation Discrepancy

Location:

- state-v1.clar#L32

Description
The current documentation states that the management fee has a maximum allowed value of **54**, as shown below:

Setting	Current Value	Description	Max Value
Management Fee	`0 bps` (0.00%)	Daily management fee on total assets	54 % of bps (0.0054%)

This is described as corresponding to a **2% annualized** fee:

Limit	Value	Description
Max Management Fee	`54 % of bps` (0.0054%)	Maximum daily management fee (2% annualized)

However, annualizing a daily fee of **0.0054%** over a 365-day year results in an effective annual fee of approximately **1.971%**, which does not reach the documented 2%.

By comparison, using **55 % of bps** (0.0055%) as the maximum daily fee would yield an annualized fee of approximately **2.0075%** over 365 days.

As a result, the current maximum annualized fee of **~1.971%** (at 54) deviates from the documented **2%** target, which may be material at higher TVL.

Recommendation
Either:

- Update the protocol documentation to reflect a maximum annualized management fee of approximately **1.971%**, or
- Increase the in-code maximum to **55**, which corresponds to approximately **2.0075%** annualized and more closely matches the documented 2% target.

[QA-14] Vault Contract Can Be Slightly Improved

Location:

- vault-v1.clar#L106
- vault-v1.clar#L232

Description

The `vault-v1` contract manages all hBTC vault operations. While the contract is generally well written, it can be slightly improved in the following areas:

1. **Remove unused `assets-net` computation**

During user withdrawals or redemptions, the `create-claim` function updates the claims map and the global state.

At the beginning of this function, the variable `assets-net` is computed as `(assets-net (- assets fee))`. However, this value is not used anywhere in the subsequent claim-creation logic and can be removed to reduce unnecessary computation and improve clarity.

2. **Improve the `fund-claim` cooldown check**

The `fund-claim` function allows execution if it is either called by the `manager` role or the claim is outside the cooldown period:

```
(define-public (fund-claim (claim-id uint))
  (let (
    ;; ... code ...
    (is-cooled-down (>= (get-current-ts) (get ts claim)))
    (is-manager (get manager
      (contract-call? .hq-hbtc-v1 get-keeper contract-caller)))
  )
    ;; ... code ...
    (if is-manager true (
      ifis-managertrue

    )
```

This `if` expression can be rewritten as a single, more readable `asserts!` statement:

```diff
-  (if is-manager true
- (asserts! is-cooled-down ERR_NOT_COOLED_DOWN)) ;; if the caller is a manager, skip the coold
+  (asserts! (or is-manager is-cooled-down) ERR_NOT_COOLED_DOWN)
```

Recommendation

Apply the changes above to the vault contract.

[QA-15] First Depositor Inflation Attack Considerations

Location:

- vault-v1.clar

Description Smart contract vaults have a well-known set of typical issues.

One such issue is that share calculations for deposits into an empty vault can allow the first depositor to inflate the share price. This can enable an attacker to extract value from subsequent depositors via a known variation of the first depositor attack, which can apply even if the vault accounts for direct deposits.

Through repeated, carefully chosen dust deposits and withdrawals, an attacker can exponentially inflate the shares-to-assets ratio such that 1 unit of shares becomes significantly more valuable. This can cause future deposits to round down substantially, allowing the attacker to capture the rounded-down amount.

Note that this attack requires the first depositor to wait until the vault's share price diverges from the initial 1:1 ratio.

The current vault design mitigates this issue by enforcing a maximum price divergence mechanism, making the attack unfeasible.

However, even with a divergence mechanism in place, corner cases related to rounding may still exist and can be fully eliminated.

Recommendation

Document the following procedure and execute it at deployment:

After deployment, have the team deposit an initial amount of assets and transfer dust (e.g., 1000 LP nano units) to an inaccessible address (e.g., `SP000000000000000000002Q6VF78`) that will never be burned.

An on-chain alternative is to lock a minimum LP amount on the first deposit to ensure no problematic rounding behavior. This can be implemented using an initialization-gated variable and transferring the locked amount directly to a burn address, e.g.: `(define-constant NULL-ADDRESS (unwrap-panic (principal-construct? (if is-in-mainnet 0x16 0x1a) 0x0000000000000000000000000000000000000000)))`.

The overhead of an on-chain implementation is not justified in this case, provided the price deviation threshold is kept low during vault deployment.

However, please document and execute the procedure described above after deployment.

[QA-16] Add Vault Action Preview Functions

Location:

- vault-v1.clar

Description The `vault-v1` contract implements a `deposit` function for inflows and two functions for outflows: `init-withdraw` and `init-redeem`.

Because vault rounding must differ depending on whether an asset-to-share conversion occurs during deposit versus during withdrawal initialization, third parties need a way to accurately preview these operations. Typically, four preview functions are used:

- `previewDeposit`: calls the existing `convert-to-shares` function with rounding **down**
- `previewMint`: similar to deposit, but allows users to deposit the amount required to receive a specific number of shares. In this case, `convert-to-assets` would be called with rounding **up**. This variation is not implemented in the current vault contract and can be ignored.
- `previewWithdraw`: calls `convert-to-shares` with rounding **up**. This should match the behavior of `init-withdraw` (after the rounding fix).
- `previewRedeem`: calls `convert-to-assets` with rounding **down**

Of these, `previewDeposit`, `previewWithdraw`, and `previewRedeem` are relevant and should be implemented in the vault to support third-party integrators.

Recommendation Modify the `convert-to-shares` function to accept a boolean parameter indicating whether to round up. Then implement the three preview functions—`previewDeposit`, `previewWithdraw`, and `previewRedeem`—each calling the appropriate conversion function with the correct rounding behavior.

These read-only functions can also be reused internally (e.g., calling `previewDeposit` from `deposit`) to simplify and centralize the logic.

[QA-17] Codebase Print Statements Improvements

Location:

- controller-v1.clar#L124
- controller-v1.clar#L158
- controller-v1.clar#L198
- state-v1.clar#L384
- trading-v1.clar#L250
- hermetica-interface-v1.clar#L91
- zest-interface-v1.clar#L191
- zest-interface-v1.clar#L233
- state-v1.clar#L500
- hbtc-token.clar#L52

Description

Across the codebase, several event `print` statements could be improved for consistency and completeness:

1. In `controller-v1`, include the `case` field within the `data` section to avoid deviating from the general print pattern used throughout the codebase.
2. In `state-v1::check-max-deviation`, the `user` field is missing, even though it is present in other print events. Notably, this is also the only read-only `check`-type function that emits an event. Consider whether this print is necessary at all.
3. In `trading-v1::zest-close-remove-redeem`, consider adding `min-sbtc-amount` to the print `data` section.
4. In `hermetica-interface-v1::hermetica-unstake-and-withdraw`, the print `data` section includes both `usdh-expected` and `usdh-received`, but they are identical. Consider keeping only one of these fields in the emitted event.
5. In `zest-interface-v1::zest-deposit`, consider adding `min-shares` to the print `data` section.
6. In `zest-interface-v1::zest-redeem`, consider adding `min-amount` to the print `data` section.

7. In `state-v1::update-state`, for the `commit-reward` print, update the `return` field to contain the reward `data` directly, and rename the field to something more appropriate.
8. In `hbtc-token::transfer`, consider using `stacks-block-height` instead of `burn-block-height` for the `block-height` entry in the print `data` section.

Recommendation
Apply the indicated logging improvements.

[QA-18] Miscellaneous Codebase Improvements

Location:

- controller-v1.clar#L118
- controller-v1.clar#L131
- state-v1.clar
- hq-hbtc-v1.clar#L309-L317
- reserve-fund-v1.clar#L19-L28

Description
Across the codebase, there are several opportunities for minor improvements that would increase execution efficiency, improve readability, and add additional (non-critical) input validation.

Recommendation

Apply the following changes:

1. In `controller-v1::handle-profit`, the expression `(+ perf-fee mgmt-fee)` is computed twice. Store the result in a `let` binding and reuse it.

2. In `state-v1`, the functions `update-total-assets`, `update-pending-claims`, `update-pending-fees`, and `update-pending-rf` all follow a `var-set` pattern. In their `print` statements, avoid re-calling the corresponding `get-*` function. Instead, compute the new value once in a `let` binding and reuse it for both `var-set` and `print`. This reduces read counts per operation.

3. In `state-v1::update-state`, cache the `get-share-price` result (e.g., near the end of the function) and reuse it in both the `check-max-deviation` call and the `print` statement. This avoids an extra `get-share-price` call.

4. Update `state-v1::check-trading-auth` to be `read-only`.

5. Add an `is-standard` check (to ensure the provided address belongs to the current network—mainnet vs. testnet) in:

   - `hq-hbtc-v1::request-new-protocol` for the `address` principal,
   - `state-v1::request-new-contract` for the `address` principal, and
   - `reserve-fund-v1::transfer` for the `recipient` principal.

6. The documentation states that the guardian role can modify the deposit cap:

   > Guardian: The guardian is a trusted and neutral third party with a public reputation. Guardians can freeze deposits and withdrawals, set deposit-cap, force unwinds.

   However, the guardian cannot currently modify the deposit cap. This capability also appears unnecessary, as the guardian can directly disable deposits. Update the documentation to reflect the implemented behavior.

[QA-19] Hermetica Interface Mint Asset Transfer Restriction Can Be Improved

Location:

- hermetica-interface-v1.clar#L118

Description In the `hermetica-interface-v1::hermetica-mint` function, when calling the auto-minting contract, a `with-all-assets-unsafe` clause is used. This is done because the amount of `minting-asset` tokens that must be consumed to mint the requested `amount-usdh` is not known in advance.

However, `hermetica-mint` first transfers a fixed amount of tokens from the reserve (`amount-assets`), which effectively serves as the maximum amount allocated for the mint.

Since `amount-assets` semantically represents the intended maximum amount to be spent, the `with-all-assets-unsafe` clause can be replaced with a more restrictive `with-ft` allowance for `minting-asset`. This ensures that no additional tokens can be consumed beyond the intended limit.

Recommendation Replace the `with-all-assets-unsafe` clause in `hermetica-interface-v1::hermetica-mint` with a `with-ft` allowance for `minting-asset` and `amount-assets`.

Additionally, because the `minting-auto` contract performs a Pyth price decode that charges a fee, a `with-stx` allowance is also required. To determine this allowance, one approach is to store the Pyth fee calculation parameters in the Hermetica interface contract and compute a `pyth-fee-amount-allowance` from them. Alternatively, add a dedicated parameter to `hermetica-mint` to supply the required STX allowance.

Example implementation:

```
(try! (as-contract? ((with-ft minting-asset "*" amounts-assets)
  (with-stx pyth-fee-amount-allowance)) (try! (contract-call? minting-auto mint minting-asset
```

Note that this change also introduces a semantic shift. Without the change, `amount-assets` could be set to `u1` (minimum to avoid reverting), and semantically it would represent:

- the amount of assets to be taken from the reserve before executing the mint. This allows any previously leftover token balance in the contract to be used for minting (reducing the need for a `sweep` call).

If the proposed change is applied, `amount-assets` would instead represent:

- the amount of assets to be taken from the reserve before executing the mint **and** the maximum amount of tokens allowed to be consumed by the `minting-auto::mint` call. As a result, after the change, `hermetica-mint` calls can no longer rely on leftover assets held by the `hermetica-interface-v1` contract, and explicit `sweep` calls would be required when that behavior is desired.

Finally, special care is required for STX wrapper contracts: they cannot be handled via a `with-ft` allowance because, at the implementation level, they transfer raw STX and therefore require a `with-stx` allowance.

[QA-20] Use the Recipient Feature of the Zest Market Interface

Location:

- zest-interface-v1.clar#L60-L61
- zest-interface-v1.clar#L88-L92

Description

The updated Zest market interface supports specifying a recipient address for funds when removing collateral and borrowing assets.

However, the `zest-collateral-remove` and `zest-borrow` functions in `zest-interface-v1` currently use the interface contract itself as the recipient, and then transfer the funds to the `reserve` contract afterward.

In `zest-collateral-remove`:

```
(let ((remaining (try! (as-contract? ((with-all-assets-unsafe)) (try!
  (contract-call? market collateral-remove asset amount (some current-contract) none)))))
  (try! (contract-call? asset transfer amount current-contract reserve none))
```

In `zest-borrow`:

```
;; Borrow from Zest market (debt recorded under this interface contract)
(try! (as-contract? ((with-all-assets-unsafe)) (try!
  (contract-call? market borrow asset amount (some current-contract) none))))

;; Transfer borrowed tokens to reserve
(try! (contract-call? asset transfer amount current-contract reserve none))
```

In both cases, `(some current-contract)` is passed as the recipient, causing funds to be received by the current contract and then moved to the reserve via a subsequent `SIP10:transfer`.

Recommendation

In `zest-collateral-remove` and `zest-borrow`, pass the `reserve` contract directly as the recipient instead of `(some current-contract)`, and remove the follow-up transfer call.

[QA-21] Use Zest Market Bundle Operations

Location:

- zest-interface-v1.clar
- trading-v1.clar
- zest-market-trait-v1.clar

Description

The Zest `market` contract exposes two helper functions that can simplify the logic in `trading-v1` by bundling common multi-step operations into single calls:

`market::supply-collateral-add`

```
;; -- Supply and collateral-add for topping up ztoken collateral
;; Deposits underlying token
  (STX, sBTC, USDC, etc.) to a vault, receives zTokens,
;; and adds those zTokens as collateral - all in one transaction.
(define-public (supply-collateral-add (ft <ft-trait>) (amount uint)
  (min-shares uint) (price-feeds (optional (list 3 (buff 8192)))))
```

and

`market::collateral-remove-redeem`

```
;;
   -- Collateral-remove and redeem for withdrawing underlying from ztoken collateral
(define-public (collateral-remove-redeem (ft <ft-trait>) (amount uint)
  (min-underlying uint) (receiver (optional principal)) (price-feeds (optional (list 3 (buff 8
```

Within the `trading-v1` contract, `zest-deposit-add-open` performs the following sequence:

```
;; Step 1: Deposit collateral to vault and get z-tokens
;; Step 1b: Add z-tokens as collateral to Zest market
;; Validate that borrow token is the canonical borrow token
;; Borrow asset from Zest v2 market
;; Stake the borrowed asset into Hermetica
```

Currently, Step 1 and Step 1b are implemented as two separate calls to the Zest interface:

```
;; Step 1: Deposit collateral to vault and get z-tokens
(z-tokens-received (try!
  (contract-call? .zest-interface zest-deposit vault collateral-token collateral-amount min-sh

;; Step 1b: Add z-tokens as collateral to Zest market
(try!
  (contract-call? .zest-interface zest-collateral-add market vault z-tokens-received price-fee
```

These two steps can be reduced to a single call by using `market::supply-collateral-add`.

Similarly, in the trading contract, `zest-close-remove-redeem` performs the reverse operations:

```
;; Unstake asset from Hermetica (instant withdrawal)
;; Repay loan to Zest v2 market
;; Step 2: Remove z-token collateral
;; Step 3: Redeem collateral from vault
  (burn z-tokens, get actual collateral amount)
```

Step 2 and Step 3 can also be combined into a single call by using `market::collateral-remove-redeem`.

Recommendation

In `zest-interface-v1`, implement wrapper functions for `market::supply-collateral-add` and `market::collateral-remove-redeem`, and then use these wrappers in `trading-v1::zest-deposit-add-open` and `trading-v1::zest-close-remove-redeem` to reduce execution costs.

Additionally, update the `zest-vault-trait-v1` trait to include the two new functions.