Hermetica hBTC Audit Report

Version 1.0

Lead Auditors

neumo

100proof

16 December 2025

Contents

1 About Greybeard Security

We are Greybeard Security, a team of independent security researchers focused on high-impact, high-integrity smart contract audits and exploit research. Our work emphasizes depth, precision, and transparency, with a track record of identifying critical vulnerabilities across major DeFi protocols.

Greybeard Security is:

100proof

- X
- Blog
- Immunefi Profile

neumo

- X
- Blog
- Immunefi Profile

2 Disclaimer

This audit is a limited security assessment of Hermetica's hBTC smart contracts based on the code and documentation provided to the audit team. While every effort was made to identify vulnerabilities, bugs, and potential risks, no audit can guarantee the complete absence of issues.

This report does not constitute legal, financial, or investment advice. It should not be considered a warranty or certification of safety, nor a recommendation to use or interact with the protocol.

Smart contracts are inherently risky and may be subject to unexpected behaviors, including those introduced by third-party dependencies, network conditions, or economic exploits. Hermetica and any users of the contracts are solely responsible for the continued testing, monitoring, and safe deployment of the protocol.

The audit team disclaims all liability for any loss or damage resulting from the use of, or reliance on, the findings presented in this report.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

The hBTC protocol is a Bitcoin-denominated yield vault and liquid staking token built on top of the Stacks DeFi ecosystem. Users deposit sBTC into a vault smart contract and receive hBTC as a liquid staking token (LST) representing their pro-rata share of the vault. The deposited sBTC is then automatically deployed into Stacks DeFi protocols such as Granite and Zest v2 to generate sBTC-denominated yield.

Operationally, the vault treats sBTC as collateral in integrated money markets and uses it to borrow non-BTC assets, such as Hermetica's own USDh, which are then deployed into yield strategies within the Hermetica ecosystem. One representative strategy is to borrow USDh against sBTC on a money market, stake USDh for sUSDh on Hermetica, and periodically calculate the staking rewards to increase the share price of hBTC. hBTC is redeemable at any time. In effect, hBTC consolidates these steps into a single BTC-in/BTC-out product where the hBTC token tracks a claim on the underlying sBTC plus accrued yield.

The protocol is designed to be non-custodial and permissionless: assets in the vault can only be moved by the protocol's smart contracts, which are controlled by a multisig with a timelock, and users can withdraw at any time without KYC/AML gating. All trading and rebalancing activity is executed on-chain and can be independently verified. The strategy is subject to

predefined limits on leverage and delta exposure, and rebalancing is automated rather than discretionary, with the stated goal of maintaining institutional-grade risk controls.

5 Audit Scope

We conducted a security review of the upcoming Hermetica hBTC contracts

The contracts in scope for this review are:

File	blank	comment	code
contracts/hbtc/protocol/state-v1.clar	117	45	703
contracts/hbtc/protocol/hq-v1.clar	50	24	269
contracts/hbtc/protocol/vault-v1.clar	30	36	211
contracts/hbtc/protocol/interfaces/hermetica-interface-v1.clar	14	16	156
contracts/hbtc/protocol/controller-v1.clar	22	29	154
contracts/hbtc/protocol/interfaces/zest-interface-v1.clar	35	43	143
contracts/hbtc/protocol/trading-v1.clar	34	51	134
contracts/hbtc/protocol/blacklist-v1.clar	27	23	124
contracts/hbtc/protocol/interfaces/granite-interface-v1.clar	13	17	98
contracts/hbtc/tokens/hbtc-token.clar	19	15	58
contracts/hbtc/traits/granite-borrower-trait-v1.clar	8	31	37
contracts/hbtc/traits/vault-trait-v1.clar	3	5	19
contracts/hbtc/protocol/fee-collector-v1.clar	5	6	16
contracts/hbtc/traits/zest-vault-trait-v1.clar	4	6	16
contracts/hbtc/protocol/reserve-fund-v1.clar	5	7	14
contracts/hbtc/protocol/reserve-v1.clar	4	7	14
contracts/hbtc/traits/zest-market-trait-v1.clar	2	3	9
Total	392	364	2175

6 Executive Summary

The Greybeard Security team performed an audit over 10 days, on the Hermetica hBTC code provided by Hermetica. A total of 17 issues were found.

This report presents the findings from our audit of Hermetica's hBTC Bitcoin yield protocol.

Hermetica hBTC allows users to deposit and stake the Stacks sBTC token automatically aggregating yields from multiple DeFi protocols such as Granite and Zest v2.

Our review evaluated all the functionality of the hBTC protocol including deposits and redemptions, blacklisting functionality, reward calculations, handling of losses by the reserve fund, system roles and their authority to perform various operations, integrations with external protocols (Zest/Granite).

Overall the codebase was of very high quality with an extensive test suite. No Critical or High severity issues were found. The key findings were:

• **Medium:** blacklisting was not handled at the level of the hBTC token and could be bypassed by transfers, no way for users to cancel redeem claims possibly leading to loss and funding time

• **Low:** rounding issues in share calculations, activation delay can be bypassed, `process-claim` should disallow zero amounts

• **Informational:** improvements in implementation of reward calculations, minimum redeem sizes, caution about sBTCs `get-balance` function, caution that Zest's `repay` function can repay less than requested, and removal of unnecessary Clarity `as-contract` calls for safety.

All findings have been documented clearly with proof-of-concept examples and recommended mitigations.

The Hermetica team were very involved in the audit process implementing changes rapidly and providing immediate feedback and direction.

As with any protocol, ongoing testing, monitoring and rigorous adherence to best-practice security processes will remain essential after deployment.

Summary

Project Name	Hermetica hBTC
Repository	hermetica-contracts
Commit	7ceb2b6adf7b...
Audit Timeline	10 Nov - 21 Nov 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	6
Informational	8
Gas Optimizations	0
Total Issues	17

Summary of Findings

[M-1] Blacklisting should be handled at the level of the hBTC token	Resolved
[M-2] Lack of mechanism to cancel claim requests	Resolved
[M-3] Blacklist can be bypassed in a certain scenario	Resolved
[L-1] Blacklist should also be checked in `redeem-internal`	Resolved
[L-2] The `trading-v1` contracts do not check `usdh-token-trait`	Resolved
[L-3] Activation delay can be bypassed	Resolved
[L-4] `process-claim` should not allow zero `assets` claims	Resolved
[L-5] Rounding direction of `convert-to-shares` is in favour of user not protocol	Resolved
[L-6] Precision loss allows for a share price increase after funding a claim	Resolved
[I-1] `controller-v1.handle-profit` doesn't use or require `is-positive` parameter	Resolved
[I-2] `handle-loss-covered/exceeds` can be simplified	Resolved
[I-3] `zest-close` can repay less than expected	Resolved

[I-4] `sbtc`'s `get-balance` does not return available balance	Acknowledged
[I-5] `zest-deposit` and `zest-redeem` refactor	Resolved
[I-6] Remove unnecessary calls to `as-contract`	Resolved
[I-7] Consider minimum redeem sizes to avoid potential attacks based on small amounts	Resolved
[I-8] Inequivalence between `fund-claim` and `fund-claim-many`	Resolved

7 Medium Findings

[M-1] Blacklisting should be handled at the level of the hBTC token

Description

The blacklist is only enforced when calling `deposit` and `request-redeem` in the Vault contract. However, this provides limited protection if the `hBTC` contract itself does not also prevent blacklisted principals from receiving, sending, minting, or burning tokens.

Without enforcement at the token level, a blacklisted address could still interact with `hBTC` and send their tokens to a "clean" address in order to perform a successful `request-redeem` call.

Impact

This issue is Medium, as it makes the entire blacklist functionality useless.

Recommended Mitigation

Implement blacklist checks directly in the `hBTC` token contract for all state-changing operations involving token transfers, mints, and burns.

Comments

Hermetica

Fixed in PR 114.

Greybeard Security

The hbtc transfer function checks for the full blacklist, not the soft one, whereas both deposits and redemptions in the vault check for the soft blacklist. The fix still allows that a soft-blacklisted address (that is not full-blacklisted) transfers their tokens to a clean address to be able to deposit/redeem.

Is it possible that the hbtc contract disallows transfers when soft-blacklisted? Otherwise the check for the soft-whitelist in deposits/redemptions is not effective.

Hermetica

This is the system we have in place for USDh as well. The soft blacklist just prevents unauthorized addresses from staking/unstaking, think of this more like a compliance feature and less like a security feature. Any malicious actor would be full blacklisted which doesn't allow them to transfer or stake (implicitly).

Greybeard Security

Ok, if that's the case the scenario I'm talking about will never happen, so all good.

[M-2] Lack of mechanism to cancel claim requests

Description

There is no mechanism to cancel claim requests. This poses a risk to users because claims are funded every 24 hours, and the cooldown period can last up to 30 days. If the funding process fails, claims could remain unfunded with no way for users to recover their shares. In such cases, if the price of `hBTC` begins to decline, users may miss the opportunity to sell their shares on secondary markets and could end up incurring losses.

The funding of claims could fail for various reasons, such as a malfunction of the keeper bots or a temporary pause in the Zest protocol. In such scenarios, users would likely want the option to cancel their pending claims.

A `cancel-claim` function could be implemented to delete an existing claim and return the corresponding `hBTC` to the user. Since claim funding is an asynchronous process, care should be taken to ensure that a cancelled claim will not make the call to `fund-claim-many` revert.

Impact

We assess the impact of this issue as Medium, as it can lead to losses for users of the protocol in certain low-likelihood scenarios.

Recommended Mitigation

Implement a `cancel-claim` function that deletes a non-funded claim and refunds the user.

Comments

Hermetica

Fixed in PR 113.

Greybeard Security

The implementation looks correct, but I have the concern that a malicious user could frontrun calls to `fund-claim-many` and make the calls fail.

Hermetica

We refactored `fund-claim-many` in order to avoid a failed tx if there is a claim id in the list that doesn't exist anymore. Fixed in PR 113.

[M-3] Blacklist can be bypassed in a certain scenario

Description

Blacklisting an address as `full` implies a soft blacklisting. We can see this in the `blacklist-processor` function, which is responsible for storing whether an address is blacklisted or not:

```
(define-private (blacklist-processor (entry { address: principal, full: bool }))
  (if (get full entry)
    (begin
      (print { action: "add-blacklist", user: contract-caller, data: { entry: entry } })
      (map-set blacklist { address: (get address entry) } { soft: true, full: true })
    )
    (begin
      (print { action: "add-blacklist", user: contract-caller, data: { entry: entry } })
      (map-set blacklist { address: (get address entry) } { soft: true, full: false })
    )
  )
)
```

If the `full` value is `true`, the address is stored with both `full` and `soft` set to `true`. If it's `false`, only `soft` is set to `true`.

This implies that the two blacklists are not independent, and the `full` blacklist (i.e., the list of fully blacklisted addresses) is a subgroup of the `soft` blacklist. Therefore, if an address is `full` blacklisted, a check for its soft blacklist status should also return `true`.

This check can be bypassed for addresses that are fully blacklisted when the soft blacklist is not active.

```
(define-read-only (check-is-not-soft (address principal))
  (ok (if (get-soft-blacklist-active)
    (asserts! (not (get-soft-blacklist address)) ERR_SOFT_BLACKLISTED)
    true
  ))
)
```

When the soft blacklist is not active, the call to `check-is-not-soft` will return true, even if the address is fully blacklisted and the full blacklist is active.

The call to `check-is-not-soft` is made in the Vault contract, specifically in the `deposit` and `request-redeem` functions. In this case, a fully blacklisted address could still deposit or request redemptions from the protocol if the soft blacklist is not active.

Impact

The likelihood is Moderate, and the impact is High, so we assessed this issue's severity as Medium.

Recommended Mitigation

When the soft blacklist is not active, the `check-is-not-soft` function should call `check-is-not-full` instead of returning true.

Proof of Concept

See test `check-is-not-soft: success when blacklisted full` in file `tests/hbtc/neumo.test.ts` in the Appendix.

Comments

Hermetica

Fixed in PR 102.

---

8 Low Findings

[L-1] Blacklist should also be checked in `redeem-internal`

Description

In the `vault` contract, the blacklist is checked when calling `deposit` and `request-redeem`. `request-redeem` creates a claim but does not execute the actual redemption of sBTC. The user must send the `hBTC` that will be burned in exchange for sBTC.

However, the redeem action may occur some time after the claim is created, and during that period the account could become blacklisted. If that happens, the call to `redeem-internal` should revert.

An example scenario where this issue could arise is a hacker who has just carried out an exploit and wants to redeem their `hBTC`. They could request a claim before being blacklisted, and even if they are blacklisted afterward, they would still be able to redeem successfully.

Impact

We assess the severity as Low, as there is a Low likelihood of occurrence but a High impact.

Recommended Mitigation

Calls to `redeem-internal` should revert if the redeemer is blacklisted.

Proof of Concept

See test `check-is-not-soft: blacklist not effective after redeem request` in file `tests/hbtc/neumo.test.ts` in the Appendix.

Comments

Hermetica

Fixed in PR 100.

[L-2] The `trading-v1` contracts do not check `usdh-token-trait`

Description

Functions such as `zest-open` do not check that the `usdh-trait` passed is actually the trait for the USDh token. A trader might pass the sBTC token instead. This would lead to quite a different outcome than expected.

For example, assume the `reserve` has the following balances:

- 10,000 USDh
- 5,000 sUSDh
- 2 sBTC

A trader calls `zest-open` but erroneously passes in the sBTC token for the `usdh-trait` parameter and asks for an amount of 10 USDh to be borrowed and staked.

- 10 USDh is `u1000000000` (8 decimals) which is equivalent to an amount of 10 sBTC (8 decimals)
- in `zest-borrow` the call to `state.check-trading-auth` will pass since sBTC is a valid `asset-trait`
- 10 sBTC will be borrowed and transferred to `reserve`
- the call to `hermetica-interface-v1.hermetica-stake` will then
  - transfer 10 USDh from `reserve` to `hermetica-interface-v1`
  - stake that 10 USDh
  - transfer the 10 sUSDh to `reserve`
- This leaves us with these final balances:

  - 9,990 USDh
  - 5,010 sUSDh
  - 12 sBTC
- The final balances should have been:
  - 10,000 USDh
  - 5,010 sUSDh
  - 12 sBTC

Impact

The impact is minimal. Since functions in `trading-v1` are simply helper functions, any action can be undone by direct calls to `zest-interface-v1` and `hermetica-interface-v1`.

Also, the balance sheet is unaffected even though the outcome was unexpected since the borrowed sBTC ends up in the reserve, perfectly offsetting the increase in liabilities.

Recommended Mitigation

Consider removing the `usdh-token-trait` parameter and using the hard-coded values as is done in `hermetica-interface-v1`.

Comments

Hermetica

Fixed in PR 110.

[L-3] Activation delay can be bypassed

Description

In the `hq-v1` contract, the function `check-activation-delay` is used in various places to enforce that `activation-delay` seconds have passed since:

- A new owner has been requested in `hq-v1` contract
- A new admin has been requested in `hq-v1` contract
- A new protocol address has been requested in `hq-v1` contract
- An new asset has been requested in `state-v1` contract
- An new contract has been requested in `state-v1` contract

This amount of time can be set by the owner at any time, with the only restriction that it must be greater than or equal to the minimum value of one day. The problem is that all previously created requests have their effective activation time changed whenever `activation-delay` is modified. This creates a security risk: for example, the owner could call `request-new-protocol` when `activation-delay` is set to several days, wait one day, then reduce `activation-delay` to one day and call `activate-protocol`, causing the activation to succeed earlier than originally intended.

The proper way to handle this is to record the activation time directly in the request entry, as is done in other parts of the code. This ensures that any update to `activation-delay` does not affect the time at which an existing request can be fulfilled.

Impact

The impact is Low, as it highlights a centralization risk in which a malicious owner could leverage their power to make the protocol behave in an unintended way.

Recommended Mitigation

When creating a request that uses an activation delay, record the activation timestamp directly in the request object.

For instance, in the `request-new-protocol` function:

```
(define-public (request-new-protocol (address principal))
  (let (
-    (new-entry { active: false, ts: (some (get-current-ts)) })
+    (new-entry { active: false, ts: (some (+ (get-current-ts) (get-activation-delay))) })
  )
    (try! (check-is-owner contract-caller))
    (try! (check-is-standard address))
    (print { action: "request-new-protocol", user: contract-caller, data: { address: address, old: (get-protocol address), new: new-entry } })
    (ok (asserts! (map-insert protocol { address: address } new-entry) ERR_DUPLICATE))
  )
)
```

Also modify function `check-activation-delay` to remove the dependency on the configured `activation-delay`:

```
(define-read-only (check-activation-delay (ts uint))
-  (ok (asserts! (>= (get-current-ts) (+ ts (get-activation-delay))) ERR_ACTIVATION))
+  (ok (asserts! (>= (get-current-ts) ts) ERR_ACTIVATION))
)
```

Proof of Concept

See test `hq: activate protocol before intended` in file `tests/hbtc/neumo.test.ts` in the Appendix.

Comments

Hermetica

Fixed in PR 107.

[L-4] `process-claim` should not allow zero `assets` claims

Description

Function `process-claim` calculates the assets like this:

```
(define-private (process-claim
  (claim-id uint)
  (share-price uint)
  (is-manager bool))
  (let (
    (claim (try! (get-claim claim-id)))
    (shares (get shares claim))
    (is-cooled-down (>= (get-current-ts) (get ts claim)))
    (assets (/ (* shares share-price) share-base))
...
```

In the edge case where the share price is below `share-base`, a claim request for 1 wei of shares would result in zero assets. Such a claim cannot be redeemed and would remain pending indefinitely.

The share price can fall below `share-base` after a liquidation event or shortly after deployment, when the initial price is equal to `share-base` and a `log-reward` call attempts to record a reward smaller than the `mgmt-fee`.

Impact

The impact is Low, because the likelihood of the share price falling below `share-base` is low.

Recommended Mitigation

Assert in `process-claim` that the calculated assets are greater than zero.

Proof of Concept

See file `tests/hbtc/neumo.test.ts` in the Appendix. Test `process-claim allows zero assets when share price below share base` shows how `process-claim` allows the number of assets in the claim to be zero.

Comments

Hermetica

Fixed in PR 103.

[L-5] Rounding direction of `convert-to-shares` is in favour of user not protocol

Description

An important property of deposits is that the rounding direction of the shares calculated for a deposit *must favour the protocol not the user*.

Public function `vault-v1.deposit` indirectly call the following:

```
(define-read-only (convert-to-shares (assets uint))
  (/ (* assets share-base) (contract-call? .state get-share-price))
)
```

The relevant snippet of `get-share-price` is:

```
(/ (* net-assets share-base) total-supply)
```

Since `get-share-price` returns a value that is rounded down this means that `convert-to-shares` will actually return a value that is too high. This is because a smaller denominator leads to a larger value upon division.

Impact

This small difference in rounding direction has led to serious attacks in the past, most notably lending protocols. However, in this case the impact is minimal. Nevertheless it is always a good idea to have "defence in depth" and ensure that such calculations cannot be exploited in the future if there are any upgrades.

Recommended Mitigation

The most accurate way to calculate the shares is to avoid the intermediate share price value and directly calculate it. We recommend moving `convert-to-shares` into the `state-v1` contract and implementing it directly as:

```
(define-read-only (convert-to-shares (assets-in uint))
  (let (
    (net-assets (get-net-assets))
    (total-supply (unwrap-panic (contract-call? .hbtc-token get-total-supply)))
  )
    (if (> total-supply u0)
      (/ (* assets-in total-supply) net-assets)
      assets-in ;; 1:1 for first deposit
    )
  )
)
```

(The renaming of `assets` to `assets-in` was necessary to avoid a collision with the `assets` data variable in `state-v1`.

Then call this function from `vault-v1` as

```
(define-read-only (preview-deposit (assets uint))
  (contract-call? .state convert-to-shares assets)
)
```

Comments

Hermetica

Fixed in PR 115.

[L-6] Precision loss allows for a share price increase after funding a claim

Description

When funding a claim, the resulting assets that the user will receive are calculated like this:

```
(define-private (process-claim
  (claim-id uint)
  (share-price uint)
  (is-manager bool))
  (let (
    (claim (try! (get-claim claim-id)))
    (shares (get shares claim))
    (is-cooled-down (>= (get-current-ts) (get ts claim)))
    (assets (/ (* shares share-price) share-base))
    (fee (/ (* assets (get fee-bps claim)) bps-base))
  )
...
```

Where `share-price` is obtained from `get-share-price`, which is:

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

In a scenario with very low liquidity in the vault and a share price that favors rounding errors — for example, a share price close to but below a 2:1 assets-to-shares ratio — a claim of 1 satoshi of shares could result in 1 asset being received. A large number of such claims would remove the same number of shares and assets from the vault, potentially impacting the share price significantly.

Impact

The impact is Low, because the likelihood of the vault entering such a specific state is very low, and we did not identify any practical way for an attacker to profit from this price change. However, we strongly encourage Hermetica to address this issue. Similar rounding-related inconsistencies have been exploited in high-profile incidents — such as the recent Balancer hack — and maintaining a defense-in-depth approach is always the safer path.

Recommended Mitigation

Enforce a minimum amount when requesting a redemption.

Proof of Concept

See file `tests/hbtc/neumo.test.ts` in the Appendix. The test `claims of 1 wei: many fund-claim` shows how, in a situation where the assets-to-shares ratio is 1.88:1 and liquidity in the vault is very low (18,800 satoshis), a malicious user could execute 1,000 claims of 1 satoshi and cause a 5.5% increase in the share price.

Comments

Hermetica

We implemented the fix with the recommended minimum redeem amount in finding **I-7**. Furthermore, remediation for finding **I-8** adds another layer of protection allowing only the manager to execute `fund-claim-many`.

---

9 Informational Findings

[I-1] `controller-v1.handle-profit` doesn't use or require `is-positive` parameter

Description

Function `handle-profit` is only called from `log-reward` if `is-profit` is `true`. Since this is defined as `(and is-positive (>= reward total-fees))` this means that `is-positive` must also be true.

Thus the `is-positive` parameter can be removed from `handle-profit`. In any case it's not even used as the function contains this `print` statement where it has constant value `true`.

```
(print {
...
data: { case: (if (is-eq reward-after-fees u0) "zero" "profit"), reward: { gross: reward, net: reward-net, rf: reward-rf, is-positive: true,
↪  is-add: true }, fees: { perf: perf-fee, mgmt: mgmt-fee }, rf: { old: total-rf, new: (+ total-rf reward-rf), required: reward-rf } }
```

Comments

Hermetica

Fixed in PR 99.

[I-2] `handle-loss-covered/exceeds` can be simplified

Description

An important invariant of the system is that *share price should remain constant when* `handle-loss-covered` *is called*.

However, verifying this invariant was quite complex because of the amount of branch logic due to `is-positive` and other if-expressions involving `reward` and `transfer-amount`.

The verification of the invariant also relies on the correct `req-rf` being passed in. This should be calculated locally as it only depends on `is-positive` and `mgmt-fee` which are already parameters.

The logic in this section is particularly complex:

```
    (reward-delta (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 })))
  )
```

This can be simplified. For the share price to remain the same the `state-v1.get-net-assets` must return the same before and after the call to `handle-loss-covered`.

Thus the `total-assets` data variable needs to change by the same amount as the sum of `pending-fees` and `pending-rf`. The code below shows us these are changed by (positive) `mgmt-fee` and (negative) `rf-decrease` respectively.

```
    (try! (contract-call? .state update-state
      (list
        { type: "pending-rf", amount: rf-decrease, is-add: false }
        { type: "pending-fees", amount: mgmt-fee, is-add: true })
      (some { reward: (get reward reward-delta), is-add: (get is-add reward-delta) })
      none))
```

Call the amount that `total-assets` will change by `asset-delta`. Using function `state-v1.get-net-assets` as our template we can now state the invariant as:

```
asset-delta - mgmt-fee + rf-decrease == 0
```

Using this invariant we can simplify the `reward-deltas` let-binding to:

```
    (delta (if (> mgmt-fee rf-decrease)
             { asset-delta: (- mgmt-fee rf-decrease), is-add: true}
             { asset-delta: (- rf-decrease mgmt-fee), is-add: false}))
    )
```

(Note: we have renamed the fields to better aid understanding of this function)

It is also possible to simply the calculation of `rf-decrease`. Since `transfer-amount` is the amount by which `req-rf` exceeds the `pending-rf` (possibly not at all i.e. zero), then the `rf-decrease` is equal to the difference between `req-rf` and `transfer-amount`.

```
-     (rf-decrease (if (> transfer-amount u0) pending-rf req-rf))
+     (rf-decrease (- req-rf transfer-amount))
```

Function `handle-loss-exceeds` can similarly be simplified. See below.

Recommended Mitigation

Apply the following diff to the entire file `controller-v1.clar`. It:

- moves the `req-rf` let-binding in `log-reward` since it is not used in the `is-profit == true` case
- removes the `perf-fee == u0` argument passed to `handle-loss-covered/exceeds`
- refactors `handle-loss-covered/exceeds` based on reasoning above

```
@@ -33,11 +33,6 @@
     (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)))
-    (req-rf (if is-profit
-      u0
-      (if is-positive
-        (- mgmt-fee reward)
-        (+ mgmt-fee reward))))
     (total-rf (+ (get-sbtc-balance rf) pending-rf))
   )
     (try! (contract-call? .hq-hbtc check-is-protocol-active))
@@ -51,13 +46,17 @@
       (try! (handle-profit reward is-positive total-rf pending-rf perf-fee mgmt-fee reserve-rate))

       ;; Handle loss scenarios
-      (if (<= req-rf total-rf)
+      (let (
+        (req-rf (if is-positive (- mgmt-fee reward) (+ mgmt-fee reward)))
+      )
+        (if (<= req-rf total-rf)
           ;; Reserve-fund can cover the loss -> token price does not change
-          (try! (handle-loss-covered reward is-positive total-rf pending-rf req-rf u0 mgmt-fee))
+          (try! (handle-loss-covered reward is-positive total-rf pending-rf req-rf mgmt-fee))

           ;; Reserve-fund cannot cover the loss -> token price decreases
-          (try! (handle-loss-exceeds reward is-positive total-rf pending-rf req-rf u0 mgmt-fee))
+          (try! (handle-loss-exceeds reward is-positive total-rf pending-rf req-rf mgmt-fee))
         )
+      )
     )
     (ok true)
   )
@@ -127,21 +126,18 @@
 (define-private (handle-loss-covered
   (reward uint) (is-positive bool)
   (total-rf uint) (pending-rf uint)
-  (req-rf uint)
-  (perf-fee uint) (mgmt-fee uint))
+  (req-rf uint) (mgmt-fee uint))
   (let (
     (transfer-amount (if (> req-rf pending-rf) (- req-rf pending-rf) u0))
-    (rf-decrease (if (> transfer-amount u0) pending-rf req-rf))
-    (reward-delta (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 })))
-  )
+    (rf-decrease (- req-rf transfer-amount))
+    (delta (if (> mgmt-fee rf-decrease)
+              { asset-delta: (- mgmt-fee rf-decrease), is-add: true}
+              { asset-delta: (- rf-decrease mgmt-fee), is-add: false}))
+    )
     (print {
       action: "log-reward",
       user: contract-caller,
-      data: { case: "loss-covered", reward: { gross: reward, net: (get reward reward-delta), rf: transfer-amount, is-positive: is-positive,
↪  is-add: (get is-add reward-delta) }, fees: { perf: u0, mgmt: mgmt-fee }, rf: { old: total-rf, new: (- total-rf req-rf), required: req-rf }
↪  }
+      data: { case: "loss-covered", reward: { gross: reward, net: (get asset-delta delta), rf: transfer-amount, is-positive: is-positive,
↪  is-add: (get is-add delta) }, fees: { perf: u0, mgmt: mgmt-fee }, rf: { old: total-rf, new: (- total-rf req-rf), required: req-rf } }
     })
```

```
    ;; Physical transfer if needed
@@ -155,7 +151,7 @@
       (list
         { type: "pending-rf", amount: rf-decrease, is-add: false }
         { type: "pending-fees", amount: mgmt-fee, is-add: true })
-      (some { reward: (get reward reward-delta), is-add: (get is-add reward-delta) })
+      (some { reward: (get asset-delta delta), is-add: (get is-add delta) })
       none))
     (ok true)
   )
@@ -164,25 +160,19 @@
 ;; @desc - Handle trading loss scenario where losses exceed reserve fund capacity
 (define-private (handle-loss-exceeds
   (reward uint) (is-positive bool)
-  (total-rf uint) (pending-rf uint) (req-rf uint)
-  (perf-fee uint) (mgmt-fee uint))
+  (total-rf uint) (pending-rf uint)
+  (req-rf uint) (mgmt-fee uint))
   (let (
-    (transfer-amount (- total-rf pending-rf))
-    ;; mgmt-fee accounted in pending-fees; reward-delta accounts reward vs RF transfer only
-    (reward-delta (if is-positive
-      ;; Positive reward < mgmt-fee (req-rf > total-rf): add reward + transfer-amount
-      { reward: (+ reward transfer-amount), is-add: true }
-      ;; is-positive = false: reward is absolute loss magnitude
-      (if (>= reward transfer-amount)
-        ;; Large loss: remaining loss = reward - transfer-amount (net decrease)
-        { reward: (- reward transfer-amount), is-add: false }
-        ;; Small loss: RF covers it, net positive = transfer-amount - reward (mgmt-fee pushes req-rf > total-rf)
-        { reward: (- transfer-amount reward), is-add: true })))
-  )
+    (transfer-amount (- total-rf pending-rf))         ;; always equal to sBTC balance of the reserve-fund
+    (transfer-with-fees (+ mgmt-fee transfer-amount))
+    (delta (if (> transfer-with-fees req-rf)
+              { asset-delta: (- transfer-with-fees req-rf), is-add: true}
+              { asset-delta: (- req-rf transfer-with-fees), is-add: false}))
+    )
     (print {
       action: "log-reward",
       user: contract-caller,
-      data: { case: "loss-exceeds", reward: { gross: reward, net: (get reward reward-delta), rf: transfer-amount, is-positive: is-positive,
↪  is-add: (get is-add reward-delta) }, fees: { perf: u0, mgmt: mgmt-fee }, rf: { old: total-rf, new: u0, required: req-rf } }
+      data: { case: "loss-exceeds", reward: { gross: reward, net: (get asset-delta delta), rf: transfer-amount, is-positive: is-positive,
↪  is-add: (get is-add delta) }, fees: { perf: u0, mgmt: mgmt-fee }, rf: { old: total-rf, new: u0, required: req-rf } }
     })

     (if (> transfer-amount u0)
@@ -195,7 +185,7 @@
       (list
         { type: "pending-fees", amount: mgmt-fee, is-add: true }
         { type: "pending-rf", amount: pending-rf, is-add: false })
-      (some { reward: (get reward reward-delta), is-add: (get is-add reward-delta) })
+      (some { reward: (get asset-delta delta), is-add: (get is-add delta) })
       none)))
   )
 )
```

3. We further recommend that the `reward` field name get changed in `state-v1` to `asset-delta` to aid understanding.

Comments

Hermetica

Fixed in PR 109.

[I-3] `zest-close` can repay less than expected

Description

`zest-repay` can leave a leftover `usdh` amount in the reserve if the function attempts to repay more than the actual debt, because Zest's `repay` caps the repaid amount to the position's outstanding debt. As a result, `zest-close` will display a value for `usdh-amount` that does not necessarily correspond to the amount actually repaid. If off-chain components or third parties rely on this value being the true repaid amount, this could lead to incorrect behavior.

```
...
    ;; Repay to Zest market
    (try! (as-contract (contract-call? market-trait repay asset-trait amount this-contract)))

    (print { action: "zest-repay", user: contract-caller, data: { market: market-trait, asset: asset-trait, amount: amount } })
...
```

`zest-close-internal` calls Zest's `repay` and returns the `usdh` amount that was unstaked from Hermetica, although that is not necessarily the same amount that was actually repaid. As a result, `zest-close` prints the amount returned by `zest-close-internal`.

NOTE: We have recommended that Zest modify their `repay` function so that it returns the actual amount repaid instead of `(ok true)`.

Impact

Since the off-chain infrastructure does not rely on print logs for the balance sheets, we have decided to assess this issue as Informational.

Recommended Mitigation

We expect Zest to update their `repay` function. In that case, the amount returned and printed by `zest-repay` should be the amount actually repaid.

Comments

Hermetica

Fixed in PR 112.

[I-4] `sbtc`'s `get-balance` does not return available balance

Description

When calling `get-balance` on `sbtc`, the returned value includes both locked and unlocked balances. Whenever possible, the `get-balance-available` function should be used instead, as it returns only the unlocked (spendable) portion.

```
(define-read-only (get-balance (who principal))
    (ok (+ (ft-get-balance sbtc-token who) (ft-get-balance sbtc-token-locked who)))
)
...
(define-read-only (get-balance-available (who principal))
    (ok (ft-get-balance sbtc-token who))
)
```

In the current codebase, all balance checks are performed on either the `reserve` contract or the `reserve-fund` contract. This is not an issue in the current state of the code, because the only way to lock an `sbtc` balance is by calling the `initiate-withdrawal-request` function in the `sbtc-withdrawal` contract. Neither the `reserve` contract nor the `reserve-fund` contract makes this call, meaning they cannot hold any locked `sbtc` tokens.

This is something to keep in mind for future improvements to the protocol.

Impact

This finding is Informational, as it is merely a recommendation and not a security risk.

Recommended Mitigation

Use `get-balance-available` whenever possible when checking `sbtc` balances.

Comments

Hermetica

Acknowledged

[I-5] `zest-deposit` and `zest-redeem` refactor

Description

The `zest-deposit` function includes a call to the vault's `deposit` function, where the `recipient` of the shares can be specified. Instead of using `this-contract` as the recipient, the call could be made with the reserve contract as the recipient directly. This would eliminate the need for a separate transfer of shares to the reserve contract.

The same approach can be applied in the `zest-redeem` function, which calls `redeem` in the Zest market contract and also accepts a recipient principal.

Also, the `asset-trait` parameter can be removed from `zest-redeem`'s function signature, as it is no longer used. This function is only called from `zest-close-remove-redeem`, which can also be simplified by removing the `sbtc-token-trait` parameter, as it is only used in the call to `zest-redeem`.

Impact

The issue is informational, as it does not pose any kind of security risk. It simply means that a couple of extra external calls could be avoided.

Recommended Mitigation

Apply the following changes:

```
    ;; Deposit to Zest vault (z-tokens minted to this interface contract)
    (let (
-      (received (try! (as-contract (contract-call? vault-trait deposit amount min-shares this-contract))))
+      (received (try! (as-contract (contract-call? vault-trait deposit amount min-shares reserve))))
    )
-      ;; Transfer z-tokens (vault shares) to reserve
-      (try! (as-contract (contract-call? vault-trait transfer received this-contract reserve none)))
```

```
(define-public (zest-redeem
  (vault-trait <zest-vault>)
-  (asset-trait <ft>)
  (shares uint)
  (min-amount uint))
  (begin
    (try! (contract-call? .dev-hq-hbtc check-is-trader contract-caller))
-    (try! (contract-call? .dev-state check-trading-auth (contract-of vault-trait) none (some (contract-of asset-trait)) none))
+    (try! (contract-call? .dev-state check-trading-auth (contract-of vault-trait) none none none))
...
    (let (
      ;; Redeem from Zest vault (burns vault shares (z-tokens), receives underlying tokens)
-      (received (try! (as-contract (contract-call? vault-trait redeem shares min-amount this-contract))))
+      (received (try! (as-contract (contract-call? vault-trait redeem shares min-amount reserve))))
    )
-      ;; Transfer received tokens back to reserve
-      (try! (as-contract (contract-call? asset-trait transfer received this-contract reserve none)))
-      (print { action: "zest-redeem", user: contract-caller, data: { vault: vault-trait, asset: asset-trait, shares: shares, min-amount:
↪  min-amount, amount: received } })
+      (print { action: "zest-redeem", user: contract-caller, data: { vault: vault-trait, shares: shares, min-amount: min-amount, amount:
↪  received } })
```

```
(define-public (zest-close-remove-redeem
  (market-trait <zest-market>) (vault-trait <zest-vault>) (staking-trait <staking>) (staking-silo-trait <staking-silo>) (hbtc-vault-trait
  ↪  <hbtc-vault>)
-  (sbtc-token-trait <ft>) (usdh-token-trait <ft>)
```

```
+  (usdh-token-trait <ft>)
...
    ;; Step 3: Redeem sBTC from vault (burn z-tokens, get actual sBTC amount)
-    (try! (contract-call? .zest-interface zest-redeem vault-trait sbtc-token-trait collateral-amount min-sbtc-amount))
+    (try! (contract-call? .zest-interface zest-redeem vault-trait collateral-amount min-sbtc-amount))
...
```

Comments

Hermetica

Fixed in PR 106.

[I-6] Remove unnecessary calls to `as-contract`

Description

It is generally recommended to avoid using `as-contract` unless strictly necessary, as it enables a class of attacks where a smart contract can execute code as if it were the `tx-sender`. This can lead to unintended privilege escalation or logic vulnerabilities if not handled carefully.

In this case, some calls in the codebase can be made without using `as-contract`, because the funds are transferred to the `contract-caller` principal, not the `tx-sender`. For instance, in the `granite-borrow` function, the borrowing logic does not require the caller to be the contract:

```
(try! (as-contract (contract-call? borrower-trait borrow none amount none)))
```

This call can be safely rewritten without as-contract:

```
(try! (contract-call? borrower-trait borrow none amount none))
```

The `borrow` function within the trait implementation sets the `user` as `contract-caller`:

```
(define-public (borrow (pyth-price-feed-data (optional (buff 8192))) (amount uint) (maybe-user (optional principal)))
  (begin
    (try! (contract-call? .withdrawal-caps-v1 check-withdrawal-debt-cap amount))
    (try! (contract-call? .pyth-adapter-v1 update-pyth pyth-price-feed-data))
    (try! (accrue-interest))
    (asserts! (>= (contract-call? .state-v1 get-borrowable-balance) amount) ERR-INSUFFICIENT-FREE-LIQUIDITY)
    (let
      (
        (user (match maybe-user user (begin (asserts! (is-eq user tx-sender) ERR-NOT-TX-SENDER) user) contract-caller))
...
      (try! (contract-call? .state-v1 update-borrow-state {
        user: user,
        user-debt-shares: total-user-debt-shares,
        user-collaterals: position-collaterals,
        user-borrowed-amount: (+ (get borrowed-amount position) amount),
        shares: new-debt-shares,
        amount: amount,
        total-borrowed-amount: (+ (get total-borrowed-amount borrow-params) amount)
      }))
...
```

Additionally, the borrow-related state update in contract `state-v1` uses the `user` to send the funds to:

```
(define-public (update-borrow-state (borrow-state {user: principal, user-debt-shares: uint, user-collaterals: (list 10 principal),
↪  user-borrowed-amount: uint, shares: uint, amount: uint, total-borrowed-amount: uint}))
...
    (try! (transfer-to .mock-usdc user amount))
    (var-set total-borrowed-amount (get total-borrowed-amount borrow-state))
    SUCCESS
))
```

The same pattern appears in `granite-remove-collateral` of the same contract.

and also

- In `hermetica-unstake`, function `unstake` can also be called without `as-contract`.
- In `hermetica-withdraw`, function `withdraw` can also be called without `as-contract`.
- In `hermetica-unstake-and-withdraw`, both functions `unstake` and `withdraw` can also be called without `as-contract`.

These functions all perform operations on behalf of the `contract-caller`, making the use of `as-contract` redundant.

Impact

This is an Informational finding that highlights a best practice in Clarity smart contracts development.

Recommended Mitigation

Remove all unnecessary `as-contract` calls in the codebase.

Comments

Hermetica

Fixed in PR 101.

[I-7] Consider minimum redeem sizes to avoid potential attacks based on small amounts

Description

A defensive strategy that is gaining traction is to enforce minimum deposit/withdraw/redeem sizes in protocol. The system currently has a minimum deposit amount but not one for redemptions.

Many subtle bugs occur due to the weaponisation of small positions (often of the minimum size of 1 satoshi). Since these kinds of deposits or redeems would never occur in practice it may make sense to prevent them altogether by implementing minimum sizes.

This leads to some additional complexity and the consideration of how to handle dust positions.

For instance, assume a minimum redeem size has been set. If a redeem would leave behind dust smaller than the minimum redeem it would need to be blocked. The user would need to redeem the full remaining shares in this case.

Credit goes to Dacian from Cyfrin Audits for recently suggesting this.

Comments

Hermetica

Fixed in PR 104.

[I-8] Inequivalence between `fund-claim` and `fund-claim-many`

Description

`fund-claim-many` calculates the share price at the beginning and then calls the fold function with all the claim IDs:

```
;; @desc - Optimized batch funding of claims
(define-public (fund-claim-many (claim-ids (list 1000 uint)))
  (let (
    (is-manager (get manager (contract-call? .hq-hbtc get-keeper contract-caller)))
    (share-price (contract-call? .state get-share-price))
    (initial-accum { total-shares: u0, total-assets: u0, share-price: share-price, is-manager: is-manager })
  )
    (asserts! (> (len claim-ids) u0) ERR_EMPTY_LIST)
    (match (fold fund-claim-iter claim-ids (ok initial-accum))
...
```

In a situation of very low liquidity, rounding errors in the calculation of `assets` inside `process-claim` could alter the share price in each iteration.

```
(define-private (process-claim
  (claim-id uint)
  (share-price uint)
  (is-manager bool))
  (let (
    (claim (try! (get-claim claim-id)))
    (shares (get shares claim))
    (is-cooled-down (>= (get-current-ts) (get ts claim)))
```

```
    (assets (/ (* shares share-price) share-base))
    (fee (/ (* assets (get fee-bps claim)) bps-base))
  )
...
```

But the `assets` will continue to be calculated using the initial share price.

This would create an inequivalence between calling `fund-claim-many` with a batch of claim IDs and calling `fund-claim` on each claim ID individually, because the resulting share price at the end of execution would differ between both approaches.

Impact

This is an Informational issue.

Recommended Mitigation

The correct solution to this issue would be to recalculate `share-price` on every iteration of `fund-claim-iter`. However, doing so would significantly increase computation costs and the number of external calls. Considering that this edge case requires both a very high share price and very low liquidity, and given that the protocol already includes a price deviation check preventing any meaningful manipulation of share price across up to 1000 claims processed in a single `fund-claim-many` call, we believe that enforcing a minimum claim amount is a sufficient and practical mitigation. This approach avoids unnecessary computational overhead while preventing zero-asset or near-zero-asset claims from introducing inconsistencies.

Comments

Hermetica

Fixed in PR 108.

---

10 Appendix

This appendix contains the source code listings we used as Proof of Concept for our findings.

`tests/hbtc/neumo.test.ts`

```
import { describe, expect, it, beforeEach } from 'vitest';
import { Cl, ClarityType, cvToValue } from '@stacks/transactions';
import {
  initHbtcProtocol,
  setupVaultTokenPrice,
  fundTestWalletsWithSBTC,
} from './helper/init';
import { errorCodes } from './settings/errors';
import {
  deployer,
  base,
  TESTS_TIMEOUT,
  wallet_1,
  getNextTestNumber,
} from './settings/constants';
import { CURRENT_TEST_CONTRACT } from './settings/contracts';

describe('Greybeard Tests', () => {
  beforeEach(() => {
    initHbtcProtocol();
  });

  describe('PoCs', () => {

    it(
      `(Test ${getNextTestNumber()}) claims of 1 wei: fund-claim-many`,
      async () => {
        await fundTestWalletsWithSBTC();

        setupVaultTokenPrice({
          tokenPrice: 100000000, // 1:1 ratio
        });

        // Verify share price is 10^8 initially
        let sharePrice = simnet.callReadOnlyFn(
          CURRENT_TEST_CONTRACT.state,
          'get-share-price', [], deployer
        );
        expect(sharePrice.result).toBeUint(100000000);

        const setCapResult = simnet.callPublicFn(
          CURRENT_TEST_CONTRACT.state,
          'set-deposit-cap',
          [Cl.uint(1000000000)],
          deployer
        );
        expect(setCapResult.result).toBeOk(Cl.bool(true));

        // Set deployer as manager
        const setManagerResult = simnet.callPublicFn(
          CURRENT_TEST_CONTRACT.hq,
          'set-keeper',
          [
            Cl.principal(deployer),
            Cl.bool(false),
            Cl.bool(true),
            Cl.bool(true),
            Cl.bool(false),
          ],
          deployer
        );
        expect(setManagerResult.result).toBeOk(Cl.bool(true));

        // Deposit to have tokens
        const depositResult = simnet.callPublicFn(
          CURRENT_TEST_CONTRACT.vault,
          'deposit',
          [Cl.uint(1000000000), Cl.none()], // 0.0001 sBTC
          deployer
        );
        // Deposit returns shares minted
        expect(depositResult.result).toHaveClarityType(ClarityType.ResponseOk);

        // Simulate increase in share price
        const logRewardResult = simnet.callPrivateFn(
          CURRENT_TEST_CONTRACT.state,
          'update-total-assets',
          [Cl.uint(880000000), Cl.bool(true)],
```

```
      deployer
    );
    expect(logRewardResult.result).toHaveClarityType(ClarityType.ResponseOk);

    // Verify share price is 1.7 * 10^8 at some point
    sharePrice = simnet.callReadOnlyFn(
      CURRENT_TEST_CONTRACT.state,
      'get-share-price', [], deployer
    );
    expect(sharePrice.result).toBeUint(188000000);

    // Create multiple withdrawal claims
    let arrayClaims = [];
    for (let i = 1; i <= 100; i++) {
      const withdraw = simnet.callPublicFn(CURRENT_TEST_CONTRACT.vault, 'request-redeem', [Cl.uint(1), Cl.bool(false)], deployer);
      expect(withdraw.result).toBeOk(Cl.uint(i));
      arrayClaims.push(Cl.uint(i));
    }
    //arrayClaims.push(Cl.uint(10000));

    // Wait for cooldown
    simnet.mineEmptyBlocks(500);

    // Fund all claims
    const fundResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.vault,
      'fund-claim-many',
      [Cl.list(arrayClaims)],
      deployer
    );
    expect(fundResult.result).toBeOk(Cl.bool(true));

    // Call withdraw-many to batch withdraw all claims
    const withdrawManyResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.vault,
      'redeem-many',
      [Cl.list(arrayClaims)],
      deployer
    );

    // Should succeed and return total withdrawn amount
    expect(withdrawManyResult.result.type).toBe('ok');

    // Verify share price is 170098137 at the end (slight increase)
    sharePrice = simnet.callReadOnlyFn(
      CURRENT_TEST_CONTRACT.state,
      'get-share-price', [], deployer
    );
    expect(sharePrice.result).toBeUint(188000008);
  },
  TESTS_TIMEOUT
);

it(
  `(Test ${getNextTestNumber()}) claims of 1 wei: many fund-claim`,
  async () => {
    await fundTestWalletsWithSBTC();

    setupVaultTokenPrice({
      tokenPrice: 100000000, // 1:1 ratio
    });

    // Verify share price is 10^8 initially
    let sharePrice = simnet.callReadOnlyFn(
      CURRENT_TEST_CONTRACT.state,
      'get-share-price', [], deployer
    );
    expect(sharePrice.result).toBeUint(100000000);

    const setCapResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.state,
      'set-deposit-cap',
      [Cl.uint(1000000000)],
      deployer
    );
    expect(setCapResult.result).toBeOk(Cl.bool(true));

    // Set deployer as manager
    const setManagerResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.hq,
      'set-keeper',
      [
        Cl.principal(deployer),
        Cl.bool(false),
        Cl.bool(true),
        Cl.bool(true),
        Cl.bool(false),
```

```
        ],
        deployer
      );
      expect(setManagerResult.result).toBeOk(Cl.bool(true));

      // Deposit to have tokens
      const depositResult = simnet.callPublicFn(
        CURRENT_TEST_CONTRACT.vault,
        'deposit',
        [Cl.uint(10000), Cl.none()], // 0.0001 sBTC
        deployer
      );
      // Deposit returns shares minted
      expect(depositResult.result).toHaveClarityType(ClarityType.ResponseOk);

      // Simulate increase in share price
      const logRewardResult = simnet.callPrivateFn(
        CURRENT_TEST_CONTRACT.state,
        'update-total-assets',
        [Cl.uint(8800), Cl.bool(true)],
        deployer
      );
      expect(logRewardResult.result).toHaveClarityType(ClarityType.ResponseOk);

      // Verify share price is 1.88 * 10^8 at some point
      sharePrice = simnet.callReadOnlyFn(
        CURRENT_TEST_CONTRACT.state,
        'get-share-price', [], deployer
      );
      expect(sharePrice.result).toBeUint(188000000);

      // Create multiple withdrawal claims
      for (let i = 1; i <= 1000; i++) {
        const withdraw = simnet.callPublicFn(CURRENT_TEST_CONTRACT.vault, 'request-redeem', [Cl.uint(1), Cl.bool(false)], deployer);
        expect(withdraw.result).toHaveClarityType(ClarityType.ResponseOk);
      }

      // Wait for cooldown
      simnet.mineEmptyBlocks(500);

      // Fund all claims
      for (let i = 1; i <= 1000; i++) {
        let fundResult = simnet.callPublicFn(CURRENT_TEST_CONTRACT.vault, 'fund-claim', [Cl.uint(i)], deployer);
        expect(fundResult.result).toHaveClarityType(ClarityType.ResponseOk);
      }

      // Verify share price is 197777777 at the end
      // This means a 5.5% increase in share price
      sharePrice = simnet.callReadOnlyFn(
        CURRENT_TEST_CONTRACT.state,
        'get-share-price', [], deployer
      );
      expect(sharePrice.result).toBeUint(197777777);
    },
    TESTS_TIMEOUT
  );

  it(
    `(Test ${getNextTestNumber()}) process-claim allows zero assets when share price below share base`,
    async () => {
      await fundTestWalletsWithSBTC();
      setupVaultTokenPrice({ tokenPrice: base.sBTC });

      simnet.callPublicFn(
        CURRENT_TEST_CONTRACT.state,
        'set-deposit-cap',
        [Cl.uint(100 * base.sBTC)],
        deployer
      );

      // Set non-zero exit fee
      const setFeesResult = simnet.callPublicFn(
        CURRENT_TEST_CONTRACT.state,
        'set-fees',
        [
          Cl.uint(0), // mgmt-fee
          Cl.uint(0), // perf-fee
          Cl.uint(100), // exit-fee = 100 bps (1%)
          Cl.uint(100), // expr-fee
        ],
        deployer
      );
      expect(setFeesResult.result).toBeOk(Cl.bool(true));

      // Set deployer as manager
      simnet.callPublicFn(
        CURRENT_TEST_CONTRACT.hq,
```

```
        'set-keeper',
        [
          Cl.principal(deployer),
          Cl.bool(false),
          Cl.bool(false),
          Cl.bool(true),
          Cl.bool(false),
        ],
        deployer
      );

      // Deposit
      simnet.callPublicFn(
        CURRENT_TEST_CONTRACT.vault,
        'deposit',
        [Cl.uint(100000000), Cl.none()],
        deployer
      );

      // Create withdrawal claim (will have 1% fee)
      const withdrawResult = simnet.callPublicFn(
        CURRENT_TEST_CONTRACT.vault,
        'request-redeem',
        [Cl.uint(1), Cl.bool(false)],
        deployer
      );
      expect(withdrawResult.result).toBeOk(Cl.uint(1));

      // Wait for cooldown
      simnet.mineEmptyBlocks(500);

      // Instead of calling fund-claim we call process-claim directly
      // with a share price below share base
      const fundResult = simnet.callPrivateFn(
        CURRENT_TEST_CONTRACT.vault,
        'process-claim',
        [
          Cl.uint(1),
          Cl.uint(98000000), // share price = 0.98:1
          Cl.bool(false),
        ],
        deployer
      );
      expect(fundResult.result).toHaveClarityType(ClarityType.ResponseOk);

      // Check the claim values
      const claim = simnet.callReadOnlyFn(
        CURRENT_TEST_CONTRACT.vault,
        'get-claim',
        [Cl.uint(1)],
        deployer
      );
      // We confirm assets is zero and is-funded is true
      expect(claim.result.value.value.assets).toStrictEqual(Cl.uint(0));
      expect(claim.result.value.value['is-funded']).toStrictEqual(Cl.bool(true));
    },
    TESTS_TIMEOUT
  );

  it(`(Test ${getNextTestNumber()}) check-is-not-soft: success when blacklisted full`, () => {

    // Set deployer as blacklister for testing
    const setResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.blacklist,
      'set-blacklister',
      [Cl.standardPrincipal(deployer), Cl.bool(true)],
      deployer
    );
    expect(setResult.result).toBeOk(Cl.bool(true));

    // Add to blacklist full
    const entries = [
      Cl.tuple({
        address: Cl.standardPrincipal(wallet_1),
        full: Cl.bool(true),
      }),
    ];

    const resultBlacklist = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.blacklist,
      'add-blacklist',
      [Cl.list(entries)],
      deployer
    );

    expect(resultBlacklist.result).toBeOk(Cl.list([Cl.bool(true)]));
```

```
    // Disable soft blacklist
    const disableResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.blacklist,
      'set-soft-blacklist-active',
      [Cl.bool(false)],
      deployer
    );
    expect(disableResult.result).toBeOk(Cl.bool(true));

    // Enable full blacklist
    const enableResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.blacklist,
      'set-full-blacklist-active',
      [Cl.bool(true)],
      deployer
    );
    expect(enableResult.result).toBeOk(Cl.bool(true));

    // Check soft
    const result = simnet.callReadOnlyFn(
      CURRENT_TEST_CONTRACT.blacklist,
      'check-is-not-soft',
      [Cl.standardPrincipal(wallet_1)],
      deployer
    );

    expect(result.result).toBeOk(Cl.bool(true));
  });

  it(`(Test ${getNextTestNumber()}) check-is-not-soft: blacklist not effective after redeem request`,
    async () => {

    await fundTestWalletsWithSBTC();

    // Setup vault and make initial deposit
    setupVaultTokenPrice({
      tokenPrice: 100000000,
    });

    // Set deployer as blacklister for testing
    const setResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.blacklist,
      'set-blacklister',
      [Cl.standardPrincipal(deployer), Cl.bool(true)],
      deployer
    );
    expect(setResult.result).toBeOk(Cl.bool(true));

    // Step 1: User calls vault.deposit() with valid amount
    const depositAmount = 70000000;
    const finalDepositTest = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.vault,
      'deposit',
      [Cl.uint(depositAmount), Cl.none()],
      wallet_1
    );
    expect(finalDepositTest.result.type).toBe('ok'); // Returns shares amount

    // Request redeem
    const user1WithdrawResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.vault,
      'request-redeem',
      [
        Cl.uint(depositAmount),
        Cl.bool(false), // expr: false for regular withdrawal
      ],
      wallet_1
    );

    // request-redeem now returns the claim ID directly
    expect(user1WithdrawResult.result.type).toBe('ok');
    const claimId = Number(cvToValue(user1WithdrawResult.result).value);

    // Add to blacklist full
    const entries = [
      Cl.tuple({
        address: Cl.standardPrincipal(wallet_1),
        full: Cl.bool(true),
      }),
    ];

    const resultBlacklist = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.blacklist,
      'add-blacklist',
      [Cl.list(entries)],
      deployer
    );
```

```
    expect(resultBlacklist.result).toBeOk(Cl.list([Cl.bool(true)]));

    // Fund and process claim
    const fundClaimResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.vault,
      'fund-claim-many',
      [Cl.list([Cl.uint(claimId)])],
      deployer
    );
    expect(fundClaimResult.result).toBeOk(Cl.bool(true));

    // Block advancement
    simnet.mineEmptyBlocks(1440);

    // Redeem succeeds despite being blacklisted
    const user1ClaimResult = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.vault,
      'redeem',
      [Cl.uint(claimId)],
      wallet_1
    );
    // withdraw returns the amount withdrawn
    expect(user1ClaimResult.result.type).toBe('ok');
  });

  it(`(Test ${getNextTestNumber()}) hq: activate protocol before intended`, () => {

    // Set activation delay to 4 days
    let newDelay = 345600; // 4 days in seconds (above minimum)
    let resultSetActivationDelay = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.hq,
      'set-activation-delay',
      [Cl.uint(newDelay)],
      deployer // owner can set activation delay
    );
    expect(resultSetActivationDelay.result).toBeOk(Cl.bool(true));

    // Setup wallet_1 as protocol
    const setupProtocol = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.hq,
      'request-new-protocol',
      [Cl.principal(wallet_1)],
      deployer
    );
    expect(setupProtocol.result).toBeOk(Cl.bool(true));

    // Advance time and try to activate (fails because it's too soon)
    simnet.mineEmptyBlocks(300);
    let activateProtocol = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.hq,
      'activate-protocol',
      [Cl.principal(wallet_1)],
      deployer
    );
    expect(activateProtocol.result).toBeErr(Cl.uint(errorCodes.v0_1.hq.ERR_ACTIVATION));

    // Set activation delay to 1 day
    newDelay = 86400; // 1 day in seconds
    resultSetActivationDelay = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.hq,
      'set-activation-delay',
      [Cl.uint(newDelay)],
      deployer // owner can set activation delay
    );
    expect(resultSetActivationDelay.result).toBeOk(Cl.bool(true));

    // Try to activate again (this time it works)
    activateProtocol = simnet.callPublicFn(
      CURRENT_TEST_CONTRACT.hq,
      'activate-protocol',
      [Cl.principal(wallet_1)],
      deployer
    );
    expect(activateProtocol.result).toBeOk(Cl.bool(true));

    // Now check both protocols
    const { result } = simnet.callReadOnlyFn(
      CURRENT_TEST_CONTRACT.hq,
      'check-is-protocol',
      [Cl.principal(wallet_1)],
      deployer
    );

    // Should return ok when both are valid protocols
    expect(result).toBeOk(Cl.bool(true));
  });
```

```
  });
});
```