



Zest v2 Audit Report

Version 1.0

Lead Auditors

[neumo](#)

[100proof](#)

4 December 2025

Contents

1	About Greybeard Security	2
2	Disclaimer	2
3	Risk Classification	2
4	Protocol Summary	2
5	Audit Scope	3
6	Executive Summary	3
7	Critical Findings	6
8	High Findings	9
9	Medium Findings	12
10	Low Findings	21
11	Informational Findings	26
12	Appendix	28

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 Zest V2 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. Zest and any users of the V2 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

Zest Protocol is a decentralized lending and borrowing platform built on the Stacks blockchain, designed to expand Bitcoin-aligned credit markets through permissionless lending, overcollateralized borrowing, and yield-bearing vaults. It enables users to supply assets such as STX, sBTC, stSTX, and USDC to earn yield, while borrowers can access liquidity against their collateral under clearly defined risk parameters.

The protocol uses a utilization-based interest-rate model: rates adjust dynamically depending on borrowing demand, target utilization, and market activity. This mechanism helps keep markets solvent, rewards lenders during periods of high utilization, and discourages excessive leverage when pools tighten.

For borrowers, Zest provides predictable credit terms, flexible collateral options, and access to liquidity without selling long-term holdings. For suppliers, it offers diversified yield opportunities backed by transparent risk parameters and a focus on Bitcoin-aligned assets.

5 Audit Scope

We conducted a security review of the upcoming V2 contracts of Zest.

The contracts in scope for this review are:

File	blank	comment	code
./contracts/market/market.clar	163	161	889
./contracts/vault/vault-*.clar	88	39	503
./contracts/market/market-vault.clar	77	53	329
./contracts/registry/egroup.clar	45	31	290
./contracts/registry/assets.clar	41	28	202
./contracts/dao/dao-multisig.clar	37	19	153
./contracts/registry/reserve-calculator.clar	32	25	101
./contracts/vault/traits.clar	11	9	55
./contracts/dao/dao-executor.clar	10	7	32
./contracts/dao/dao-treasury.clar	5	5	15
./contracts/dao/traits.clar	2	0	12
Total	511	377	2581

Only the items listed in the Scope section above are in scope, and the following items are explicitly out of scope:

- Full test coverage for the entire repository or unrelated subsystems
- Frontend, backend, and off-chain infrastructure

6 Executive Summary

The Greybeard Security team performed an audit over 15 days, on the [Zest v2](#) code provided by [Zest](#). A total of 26 issues were found.

This report presents the findings from our audit of Zest Protocol V2 — the next iteration of Zest's lending and borrowing system built on Stacks.

Zest V2 refines the protocol's core mechanisms for interest accrual, borrowing, liquidations, governance, and vault accounting. The upgrade focuses on correctness, consistency, and safety across the protocol's markets, while maintaining the architecture introduced in the previous version: utilization-based interest rates, egroup-based risk segmentation, and modular vaults for supported assets.

Our review evaluated the accuracy of debt accounting, liquidation logic, accrual functions, oracle usage, governance initialization, and cross-module interactions. The audit covered the full smart-contract codebase provided to us, including vaults, liquidation logic, interest-rate machinery, DAO contracts, and helper libraries.

Several issues were identified across all severity levels, including:

- **Critical:** incorrect debt repayment calculation in liquidations, and socialize-debt failing to decrease total-borrowed — both affecting core solvency and liquidation behavior.
- **High:** issues in health-check logic, debt preview under paused accruals, excessive repayment in both repay and liquidate, multisig initialization allowing duplicate signers, and required monotonicity guarantees for egroup LTV hierarchies.
- **Medium:** missing health checks in key flows, DoS scenarios in the accrue function, incorrect minting logic for treasury shares, liquidation factors exceeding 100%, a faulty pack-u16 implementation, and incorrect behavior under extremely frequent accrue calls.
- **Low:** oracle timestamp edge cases, incorrect caps and rounding in system-borrow, initializer misbehavior in dao-executor, proposal-expiry inconsistencies, wrong asserts in the new liquidation path, and missing treasury shares for flashloan fees.

• **Informational:** improvements to bit-handling, draft naming suggestions, and an incorrect ubalance implementation in vault-sbtc.

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

The Zest team has been cooperative throughout the audit, and has begun addressing the reported issues. The severity and breadth of findings highlight the importance of thorough testing and careful iteration before mainnet deployment. We recommend another audit be conducted.

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

Summary

Project Name	Zest v2
Repository	zest-core
Commit	f4987a8b177e...
Audit Timeline	20 Oct - 07 Nov 2025
Methods	Manual Review

Issues Found

Critical Risk	2
High Risk	4
Medium Risk	10
Low Risk	7
Informational	3
Gas Optimizations	0
Total Issues	26

Summary of Findings

[C-1] When liquidating, debt repayment is calculated wrongly	Resolved
[C-2] vault-*.socialize-debt does not decrease total-borrowed preventing any further repay or liquidation	Resolved
[H-1] Existing debt not used in health check when borrowing disabled on asset	Resolved
[H-2] Debt preview is wrongly calculated when accruals are paused	Resolved
[H-3] Function repay allows the user to repay excessively	Resolved
[H-4] Function liquidate allows the user to repay excessively	Resolved
[M-1] Lack of health check after repayment	Resolved
[M-2] When removing collateral there should be an additional health check at the end	Resolved
[M-3] Health check before borrowing should be done with current mask	Resolved
[M-4] DoS in accrue function	Resolved
[M-5] Not enough .dao-treasury shares minted in accrue function	Resolved

[M-6] Duplicate signers in dao-multisig initialization	Resolved
[M-7] Liquidation factor percentage should never surpass 100%	Resolved
[M-8] Faulty pack-u16 implementation	Resolved
[M-9] egroupp LTVs must always be monotonically decreasing when one egroupp is a superset of another	Resolved
[M-10] Extremely frequent calling of accrue could lead to z-tokens accruing no value and borrow index growing faster than it should	Resolved
[L-1] Oracle publish date could be in the future	Resolved
[L-2] Wrong cap in system-borrow function	Resolved
[L-3] Rounding direction in system-borrow should match borrow	Resolved
[L-4] dao-executor can be initialized multiple times	Resolved
[L-5] Proposal expiry time inconsistency	Resolved
[L-6] Wrong asserts in new liquidate function	Resolved
[L-7] No protocol fees received on flashloan	Resolved
[I-1] Replace bit-handling functions in pack with ones using Clarity bit-wise operations	Resolved
[I-2] Change field names for better readability	Acknowledged
[I-3] Wrong implementation of ubalance function in vault-sbtc	Resolved

7 Critical Findings

[C-1] When liquidating, debt repayment is calculated wrongly

Description

When you call `liquidate` on a position, you pass in a collateral and a debt token. So you will receive some of the collateral in exchange for the amount of debt you repay. You can pass a liquidation amount greater than the debt of that token the borrower has. The function caps the debt (well, it doesn't currently due to a separate issue) to the debt the borrower has of that particular debt token.

The amount of collateral to seize is wrongly calculated using the total debt the user has of all the debt tokens they borrowed, instead of using only the debt token the liquidator is actually repaying. So an attacker passing in a liquidation amount high enough will have the collateral received calculated with the whole debt but will just repay one of the debt tokens, potentially for a good profit.

Example: The borrower has deposited \$10,000 worth of sBTC and has borrowed \$100 worth of USDC and \$6,000 worth of USDh. After some time, the sBTC price drops and the position becomes fully liquidatable. The attacker calls `liquidate`, passing sBTC as collateral and USDC as debt, and an amount of \$6,100. The function then calculates the expected collateral amount, which is the full sBTC amount, but when it comes to the repayment, the system only charges \$100, because that's the actual USDC debt. The collateral expected amount must be calculated with the amount capped to the debt of the token, not the whole notional debt of the borrower.

Impact

An attacker can receive collateral in exchange for a tiny amount of debt repaid, so this issue has a Critical impact.

Recommended Mitigation

First calculate the debt the user will repay, taking into consideration the liquidation factor, the total amount of debt the borrower has of the repayment token, etc., and then calculate the amount of collateral the liquidator is eligible to receive.

Proof of Concept

See file `tests/greybeard/neumo.test.ts` in the Appendix.

Comments

Zest

Fixed in commit [35ad1fc8db4b930c94ca14a54a4a9b91be394b4d](#)

[C-2] `vault-*.socialize-debt` does not decrease `total-borrowed` preventing any further repay or liquidation

Description

The `vault-*.socialize-debt` function only ever decreases the `principal-scaled` data variable but leaves `total-borrowed` untouched.

This can lead to a situation where `(debt)` will return a value less than `(var-get total-borrowed)`

This affects later calls to `system-repay`.

```
(define-public (system-repay (amt uint))
  (let (
    ...
    @1> (d (debt))
    @2> (total-borrowed-amount (var-get total-borrowed))
    @3> (actual-amount (if (> amt d) d amt))
        (pratio (calc-principal-ratio-reduction actual-amount p d))
        (actual-reduction (if (> pratio p) p pratio))
        (p-new (- p actual-reduction))
    @4> (principal-repaid (mul-div-down actual-amount total-borrowed-amount d))
    @5> (interest-paid (- actual-amount principal-repaid))
    ...
```

Let:

- D be d
- B be total-borrowed-amount
- A be actual-amount
- R be principal-repaid
- $D < B$

Line @4> calculates R as

$$R = \left\lfloor \frac{AB}{D} \right\rfloor$$

but because $D < B$ we have

$$\frac{B}{D} > 1$$

This will often mean that $R > A$ by the equation above. Thus, we get an underflow on line @5> because $A - R < 0$.

A thorough derivation of the precise conditions for the underflow is as follows:

$$\begin{aligned} A - R &< 0 \\ A - \left\lfloor \frac{AB}{D} \right\rfloor &< 0 \\ \left\lfloor \frac{AB}{D} \right\rfloor &> A \\ \frac{AB}{D} &\geq A + 1 \\ AB &\geq D(A + 1) \\ A(B - D) &\geq D \\ A &\geq \frac{D}{B - D} \end{aligned}$$

If we take $A = 1$ as the minimum repay amount this means that an underflow will *always* happen when:

$$D \leq \frac{B}{2}$$

Since repay amounts have eight fixed-point decimals $A = 1$ is actually an extreme case. Looking at a realistic example where $D = 100,000,000$ and $B = 150,000,000$ we find that underflow happens any time $A \geq 2$! Only repayment of dust is possible, which effectively cripples the functionality.

Impact

Both `repay` and `liquidate` are effectively blocked.

The impact is severe since LPs can only redeem if there are enough underlying assets available. Not only do LPs not get interest payments from borrowers, some LPs are effectively blocked from ever getting their deposits back as a "race to the bottom" of LP withdrawal is likely once this issue becomes apparent.

Recommended Mitigation

Ensure that `total-borrowed` is correctly decreased.

Proof of Concept

See file `tests/greybeard/socialize-debt-blocks-repay.test.ts` in Appendix

Comments

Zest

Good catch. Fixed here: [b1b60c93a81636144f79060aa3908158b87dfb73](#).

8 High Findings

[H-1] Existing debt not used in health check when borrowing disabled on asset

Description

If a borrower borrowed asset A (but doesn't have it as collateral), but then it was disabled via `assets.disable` then when calling `liquidate` we have:

- `let ... (context (ctx borrower))`
- But `assets` field is set to `(assets mask)`
- `(assets mask)` returns a list of elements of form `{ id: id, collateral: c, debt: d, price: uint }`

```
(define-private (assets (mask-user uint))
  (let ((mask-enabled (enabled-mask))
        (safe-mask (user-safe-mask mask-user mask-enabled))
        (iter (mask-to-list-collateral safe-mask))
        (alist (status-multi iter)))
    ;; ...
    (map merge-price alist plist)))
```

- `user-safe-mask` will return a mask in which A's bit is set to 0

```
(define-private (mask-to-list-collateral (mask uint))
  (mask-to-list-internal mask u0 ITER-UINT-64))

(define-private (mask-to-list-internal (mask uint) (offset uint) (iter-list (list 64 uint)))
  (let ((init { mask: mask, offset: offset, result: (list) })
        (out (fold mask-to-list-iter iter-list init)))
    (get result out)))

(define-private (mask-to-list-iter (p uint) (acc {mask: uint, offset: uint, result: (list 64 uint)}))
  (let ((mask (get mask acc))
        (offset (get offset acc))
        (has? (asserts! (has-bit mask p) acc))
        (result (get result acc))
        (value (if (is-eq offset u0) p (- p offset)))
        (new (as-max-len? (append result value) u64)))
    (merge acc { result: (unwrap-panic new) })))
```

Impact

The disabled debt token is not included in the health check meaning the position is more healthy than it should be.

Recommended Mitigation

Proof of Concept

See file `test/greybeard/liquidation.test.ts` in the Appendix

Comments

Zest

We oversee all positions updated every block, and it's our job to monitor them constantly, so this would require poor decision making on our part. Even when off-boarding an asset, we would start by decreasing caps and not just disable an asset immediately. Even in an emergency we would pause borrowing or decrease caps to zero and could still liquidate. However fixing this would allow us to just immediately disable and liquidate old debt so that is indeed useful.

Overall my thinking is I agree on this issue though just not on the severity because the existing contracts allow us to still liquidate debt while off-boarding an asset before eventually disabling.

Fixing this will allow us to disable assets with debt while still allowing liquidations which adds great flexibility to our operations and makes emergency responses better.

Zest

Fixed in commit [26a4a619a037ee58bbaa57f71870b0e305a373e2](#)

[H-2] Debt preview is wrongly calculated when accruals are paused

Description

debt-preview uses the next-index to calculate the debt as if the index had already been updated. However, if accrual is paused, debt-preview is still calculated as if it were not, meaning it includes all the debt that would have accrued since the last last-update timestamp. When accruals are paused, last-update remains unchanged until accruals are resumed. Therefore, debt-preview should account for this and avoid returning an inflated value.

As it stands, this leads to incorrect results in total-assets-preview, which in turn causes get-total-assets, convert-to-assets-preview, and convert-to-shares-preview to return inaccurate values. These inaccuracies affect the behavior and outputs of the deposit and redeem functions.

Impact

The severity is assessed as High due to the significant impact of inaccurate return values from convert-to-assets-preview and convert-to-shares-preview. While the likelihood of accruals being paused is moderate, the consequences of these functions returning incorrect values during that state are substantial.

Recommended Mitigation

Make sure that next-index returns the current index if accruals are paused.

Proof of Concept

See file tests/greybeard/neumo.test.ts in the Appendix.

Comments

Zest

Fixed in commit [041b5245943429ee2e02d878bd44517c3bc2220f](#)

[H-3] Function repay allows the user to repay excessively

Description

In function repay, the call to vault-system-repay is made passing amount, but it should be capped if sdelta > scurr; otherwise, the user would be sending too much.

```
...
  (sdelta (mul-div-down amount PRECISION bi))
  (scurr (contract-call? .market-vault debt-scaled oid aid))
  (dec (if (> sdelta scurr) scurr sdelta)))

;; --- auth ---
(asserts!
  (or
    (is-eq account tx-sender)
    (is-eq account contract-caller)
  )
  ERR-AUTH)

;; --- preconditions ---
(asserts! (> amount u0) ERR-AMOUNT-ZERO)
(asserts! (> dec u0) ERR-INSUFFICIENT-SCALED-DEBT)

;; --- repay ---
(try! (vault-system-repay aid amount))
...
```

In the case where the scaled debt amount of the position (scurr) is less than the scaled debt amount passed in (sdelta), the amount repaid should not be amount, but instead should be scurr multiplied by the index.

Impact

We assessed the impact of this issue as High, due to its high likelihood of occurring. It could happen when a user inadvertently sends an excessive repayment amount.

Recommended Mitigation

If the amount sent to repay is greater than the debt of the position, cap it to avoid repaying too much.

Proof of Concept

See file `tests/greybeard/neumo.test.ts` in the Appendix.

Comments

Zest

Fixed in commit [a6a2265d0d51184d0d511127ac96b068c0046c3d](#)

[H-4] Function liquidate allows the user to repay excessively

Description

Just like in the repay function, the liquidation function scales the debt amount in `scaled-to-remove` with the comment `cap` at current debt (prevent over-repayment). However, the call to `vault-system-repay`, which is the function that actually moves funds, is made with `debt-final`, which is the unscaled and uncapped amount.

```
...
;; cap at current debt (prevent over-repayment)
(scaled-to-remove (if (> scaled-debt curr-scaled) curr-scaled scaled-debt)))

;; --- assertions ---

;; 0. liquidation must not be paused (check first for fail-fast)
(asserts! (not liq-paused) ERR-LIQUIDATION-PAUSED)
;; 1. position must be liquidatable (at or above partial threshold)
(asserts! (>= current-ltv ltv-liq-partial) ERR-HEALTHY)
;; 2. input must be non-zero
(asserts! (> debt-amount u0) ERR-AMOUNT-ZERO)
;; 3. must result in non-zero liquidation amounts
(asserts! (> debt-final u0) ERR-ZERO-LIQUIDATION-AMOUNTS)
(asserts! (> coll-actual u0) ERR-ZERO-LIQUIDATION-AMOUNTS)

;; --- execute liquidation ---
(try! (vault-system-repay debt-aid debt-final))
...
```

The amount to repay should not be `scaled-to-remove` multiplied by the index and not `debt-final`.

Impact

Recommended Mitigation

Fix the call to `vault-system-repay` so that the correct amount — the unscaled value of `scaled-to-remove` — is passed in.

Comments

Zest

Fixed in commit [a6a2265d0d51184d0d511127ac96b068c0046c3d](#)

9 Medium Findings

[M-1] Lack of health check after repayment

Description

After a user fully repays the debt of one of the assets they have borrowed, their egroup may change. While it is unlikely that a repayment would cause an account to become unhealthy, the current code allows for this possibility depending on how the various egroups are configured.

Impact

The possibility that a repayment could make an account unhealthy is unlikely and would only occur due to a misconfiguration of the system. However, the potential impact of such a scenario is High. Therefore, we assess the severity of this issue as Medium.

Recommended Mitigation

Check the health of the position right after the repayment.

Proof of Concept

See file `tests/greybeard/neumo.test.ts` in the Appendix.

Comments

Zest

Fixed in commit [341b03e5ef73503e160360851695f2ab0ed4cc14](#)

Greybeard Security

After the call to `debt-remove-scaled`, the mask is updated in the user in case all the debt is repaid, so there's no need to calculate the future mask, just retrieving the current mask of the user would suffice.

Also, would it make sense to create a private function `is-healthy` that accepts a user id and an optional mask and returns if the user is healthy according to the mask or to their actual mask if none is passed? I think the health is checked in several places in the code, and would make it much more readable.

Zest

Fixed in commit [2853c9f39d087c5711c53d342c0b9544f3189528](#)

[M-2] When removing collateral there should be an additional health check at the end

Description

The `collateral-remove` operation could result in a different mask than the one the user started with. This is because if all collateral of a given asset is removed, the corresponding bit is cleared from the mask. As a result, the effective egroup may change, leading to different LTV requirements.

Therefore, as is already done in the `borrow` function, it is necessary to recalculate the final egroup and perform a health check with the future mask immediately after the call to `.market-vault collateral-remove`.

Impact

The likelihood of performing the health check using the previous mask instead of the updated one—and the check passing with the previous mask but failing with the new mask—is Low. However, the potential impact of such a scenario is High. Therefore, we assess the severity of this issue as Medium.

Recommended Mitigation

Check the health of the position right after removing the collateral using the future mask.

Proof of Concept

See file `tests/greybeard/neumo.test.ts` in the Appendix.

Comments

Zest

Fixed in commit [bdb4c5381c30c1ab5101ce69a43ba590be8bd42f](#)

Greybeard Security

I would simplify it much more. Creating a `is-healthy` function (as suggested in [#5](#)) and calling it at the end of the `collateral-remove` would be enough. Thoughts?

Zest

Added here: [2853c9f39d087c5711c53d342c0b9544f3189528](#)

[M-3] Health check before borrowing should be done with current mask

Description

The way an account's health is checked before a borrow is incorrect. It is currently done using the future mask (the mask after borrowing the asset), but it should instead use the current mask.

```
...  
;; Calculate FUTURE mask (after adding this debt)  
;; For debt: bit position = aid + 64 (DEBT-OFFSET)  
(debt-bit-pos (+ aid u64))  
(future-mask (let ((div ((div (pow u2 debt-bit-pos))  
                             (shiftr (/ mask div))  
                             (bit (mod shiftr u2))  
                             (base (if (is-eq bit u0) div u0)))  
                (+ mask base))))  
  
;; Use egroup for FUTURE mask (what will govern position after borrow)  
(group (egroup future-mask))  
(ltvb (buff-to-uint-be (get LTV-BORROW group)))  
  
;; --- ltv ---  
(n (notional { position: pos, assets: alist }))  
(cc (get collateral n))  
(cd (get debt n))  
(hprev? (healthy? cc cd ltvb))  
;; Calculate borrow amount in USD using asset info we already have  
(rv (let ((oracle-data (get oracle a))  
          (price (unwrap-panic (price-resolve oracle-data aid))
```

```

        (decimals (get decimals a)))
      (normalize (* amount price) decimals true)))
    (pc (+ cd rv))
    (hpost? (healthy? cc pc ltvb)))
  ...

```

The future mask will differ from the current mask in cases where the asset has never been borrowed by the user.

Imagine an edge case where the account is currently eligible for liquidation, but borrowing a specific debt token would result in a more favorable LTV-BORROW value and make the account appear healthy. In such a case, because the health pre-check is done using the future mask, the account could appear healthy when it is not. By borrowing even a minuscule amount of the new debt token, the account could remain “healthy” and avoid liquidation.

This would depend, of course, on the configuration of the different egroups, but it is theoretically possible.

Impact

The combination of low likelihood and high impact leads us to assess this issue as having Medium severity.

Recommended Mitigation

Use the current mask to check the health of the position before the borrow, and the future mask to evaluate the position after the borrow.

Comments

Zest

Fixed in commit [a5e44d0b6ee0e8cbee420c0d2e0805fde6a44d1a](#)

Greybeard Security

Fixes look good to resolve the issue, but I would suggest again to implement an is-healthy function.

Zest

Fixed in commit [2853c9f39d087c5711c53d342c0b9544f3189528](#)

[M-4] DoS in accrue function

Description

Calls to the accrue function in the vault-* contracts will always revert during any period in which reserve-inc has a value of exactly 1. We assume here a vault state where the proportion of assets to shares is greater than 1 and less than 2 — meaning there are always more assets than shares, but the number of assets never reaches double the number of shares.

In such a scenario, when reserve-inc equals 1, the contract attempts to mint zero shares to the dao-treasury, which causes a revert.

```

... (if (> reserve-inc u0)
      (let ((treasury-lp (convert-to-shares-preview reserve-inc)))
        (try! (ft-mint? zft treasury-lp .dao-treasury)))
      false)
...

```

If we review in detail how the values re calculated:

```

(reserve-inc (mul-div-down debt-delta rf BPS)))

```

Assuming a reserve factor (rf) of 10% (1000), reserve-inc will be exactly 1 when $\text{debt-delta} * \text{rf} / \text{BPS} = \text{debt-delta} * 1000 / 10000 \Rightarrow 10 \leq \text{debt-delta} < 20$.

```

(old-debt (/ (* p idx) PRECISION))
(new-debt (/ (* p next) PRECISION))
(debt-delta (if (> new-debt old-debt) (- new-debt old-debt) u0))

```

For debt-delta to be greater than or equal to 10 and less than 20, the following must hold: $10 \leq (\text{new-debt} - \text{old-debt}) < 20$.

```
(new-debt - old-debt) = (principal-scaled * next / PRECISION) - (principal-scaled * idx / PRECISION)
= (principal-scaled * next - principal-scaled * idx) / PRECISION = principal-scaled * (next - idx) / PRECISION
=> PRECISION <= (principal-scaled * (next - idx)) < (2*PRECISION)
```

If we assume the interest rate does not change between the two debts (utilization stays the same):

```
calc-multiplier-delta = PRECISION + (interest-rate * time-delta * PRECISION / SECONDS-PER-YEAR-BPS)
next-index = (var-get index) * calc-multiplier-delta / PRECISION
```

So our assumption here is that calls to accrue will revert if principal-scaled is smaller than PRECISION. The smaller the value of principal-scaled, the lower the utilization, which means that a larger time-delta can cause the function to revert.

Impact

We will label this issue as Medium because the likelihood of it occurring is low (the borrowed amount in the vault must be small), but the impact is high: since accrue is widely used, the resulting denial of service would affect nearly the entire contract.

Recommended Mitigation

Possible fix:

```
(new-debt (/ (* p next) PRECISION))
(debt-delta (if (> new-debt old-debt) (- new-debt old-debt) u0))
- (reserve-inc (mul-div-down debt-delta rf BPS))
+ (reserve-inc (mul-div-down debt-delta rf BPS))
+ (treasury-lp (if (> reserve-inc u0) (convert-to-shares-preview reserve-inc) u0)))
(if (not (is-eq idx next))
    (var-set index next)
    false)
(if (not (is-eq lidx nliq))
    (var-set lindex nliq)
    false)
- (if (> reserve-inc u0)
-   (let ((treasury-lp (convert-to-shares-preview reserve-inc)))
-     (try! (ft-mint? zft treasury-lp .dao-treasury))))
+ (if (> treasury-lp u0)
+   (try! (ft-mint? zft treasury-lp .dao-treasury))
+   false)
(if (or (not (is-eq idx next)) (not (is-eq lidx nliq)))
    (var-set last-update stacks-block-time))
```

Proof of Concept

See file tests/greybeard/neumo.test.ts in the Appendix.

Comments

Zest

Fixed in commit [48deacf2f24a6ccdb7600f2337ff8239c13ccce6](#)

[M-5] Not enough .dao-treasury shares minted in accrue function

Description

The relevant lines from accrue are:

```
(define-public (accrue)
  ...
  (let ((next (next-index))
        (nliq (next-liquidity-index))
        (old-debt (/ (* p idx) PRECISION))
        (new-debt (/ (* p next) PRECISION))
        (debt-delta (if (> new-debt old-debt) (- new-debt old-debt) u0))
        (reserve-inc (mul-div-down debt-delta rf BPS)))
    ...
    (if (> reserve-inc u0)
        (let ((treasury-lp (convert-to-shares-preview reserve-inc)))
          (try! (ft-mint? zft treasury-lp .dao-treasury)))
        false))
  ...)
```

Let:

- reserve-inc be v .
- (total-assets-preview) be T' . The prime is added because it calculates the (debt-preview) on the (next-index)
- (total-supply) be S
- treasury-lp be s

We now want to work out how many shares s should be minted so that, after the mint, they have value v . We want to calculate:

$$v = \frac{s \cdot T'}{S'} \text{ where } S' = S + s$$

$$\begin{aligned} v &= \frac{s \cdot T'}{S'} \\ v &= \frac{s \cdot T'}{S + s} \\ (S + s) \cdot v &= s \cdot T' \\ S \cdot v + s \cdot v &= s \cdot T' \\ S \cdot v &= s \cdot T' - s \cdot v \\ S \cdot v &= s \cdot T' - v \\ \frac{(S \cdot v)}{T' - v} &= s \\ s &= \frac{v \cdot S}{T' - v} \end{aligned}$$

However, convert-to-shares-preview:

- calculates the total assets by adding assets to (debt-preview)
- (debt-preview) calculates cumulative debt using (next-index)

Thus convert-to-shares-preview calculates

$$s = \frac{v \cdot S}{T'}$$

Thus, the denominator is too big and the treasury does not get enough shares minted.

Impact

Since v will usually be small with respect to T' this will mean that the discrepancy is usually small.

Recommended Mitigation

```
- (let ((treasury-lp (convert-to-shares-preview reserve-inc)))
+ (let ((treasury-lp (mul-div-down reserve-inc (total-supply) (- (total-assets-preview) reserve-inc))))
  (try! (ft-mint? zft treasury-lp .dao-treasury)))
```

Comments

Zest

Fixed in commit [48abb747cfaa47e5f8c0b8f40e32b61985598132](#)

[M-6] Duplicate signers in dao-multisig initialization

Description

In contract dao-multisig, calling the init function with duplicate values in the signers list could create an inconsistency between the stored signer list and the signer-count variable.

```
(define-public (init (signer-list (list 20 principal)) (new-threshold uint))
  (begin
    (asserts!
      (and
        (is-eq DEPLOYER tx-sender)
        (> new-threshold u0)
        (<= new-threshold (len signer-list))
        (is-eq (var-get threshold) u0))
      ERR-SANITY-SIGNER)

    (map set-signer signer-list)
    (var-set threshold new-threshold)
    (var-set signer-count (len signer-list))
    (ok true)))

(define-private (set-signer (signer principal))
  (map-set signers signer true))
```

The signer map is populated via a call to set-signer for each element in the list. If there are duplicates, the corresponding map entry is simply overwritten. Meanwhile, signer-count is calculated as the length of the list.

As a result, the signer count could be higher than the actual number of unique signers registered in the contract. This could allow, for example, setting a threshold higher than the number of valid signers, potentially breaking the multisig's ability to reach quorum.

Impact

This issue could render the contract unusable, and with it the governance of the Zest protocol, if the threshold is set to the length of the signers list. In such a case, the actual signer set may never be able to meet the threshold requirements due to duplicate entries being ignored in the signer map.

The likelihood of this occurring is Low, but the potential impact is Critical. Therefore, we assess the severity of this issue as High.

Recommended Mitigation

```
(define-private (set-signer (signer principal))
-  (map-set signers signer true))
+  (begin
+    (unwrap-panic (set-signer-internal signer true))
+    (ok true)))
+
+(define-private (set-signer-internal (signer principal) (val bool))
+  (begin
+    (asserts! (map-insert signers signer val) ERR-SANITY-SIGNER)
+    (ok true)))
```

Proof of Concept

See file `tests/greybeard/neumo.test.ts` in the Appendix.

Comments

Zest

Fixed in commit [a640ee97873241ac31c9b98b49621de67592d0c1](#)

[M-7] Liquidation factor percentage should never surpass 100%

Description

`calc-liq-factor` is calculated as follows:

```
(define-read-only (calc-liq-factor (ltv-curr uint) (ltv-liq-partial uint) (ltv-liq-full uint))
  (div-bps-down (- ltv-curr ltv-liq-partial) (- ltv-liq-full ltv-liq-partial))) ;; Round down: conservative liquidation factor
```

If the value of `ltv-curr` is greater than `ltv-liq-full`, the function will return a value greater than BPS.

The function is called in the `liquidate` function as follows:

```
(liq-pct-linear (contract-call? .reserve-calculator
  calc-liq-factor current-ltv ltv-liq-partial ltv-liq-full))
```

where `current-ltv` is the LTV of the position being liquidated, and `ltv-liq-partial` and `ltv-liq-full` are configured at the egroup level.

This means that if the LTV goes beyond the value of `ltv-liq-full`, the function will return a value greater than 100%, which implies, assuming `exp >= BPS`:

- a value of `liq-pct-scaled` greater than 100%
- `max-debt-usd` greater than the total debt of the position
- `debt-actual-usd` equal to `max-debt-usd` if a sufficiently large amount is passed to the liquidation
- `coll-usd-expected` greater than it should be

Although the amounts of collateral seized and debt repaid are later capped (even more so after applying some of the fixes to other reported issues), the relationship between debt and collateral could be corrupted because of this issue, allowing the liquidator to receive too much collateral.

Impact

Medium, because the likelihood of a position not being liquidated until it crosses the full liquidation threshold is Low, even though the impact if it ever happened would be High.

Recommended Mitigation

With all the issues found in the liquidation function, we recommend rewriting it top down and simplifying it as much as possible to prevent edge cases from arising in the future.

Proof of Concept

See file `tests/greybeard/neumo.test.ts` in the Appendix.

Comments

Zest

Fixed in commit [0420ac583f64143257ec6a90721df91192ce0d78](#)

[M-8] Faulty pack-u16 implementation

Description

Function `pack-u16`, along with its fold function `iter-pack-u16`, has a couple of errors that can result in an incorrect calculation of the interest rate.

```
(define-public (pack-u16 (fields (list 8 uint)) (upper (optional uint)))
  (let ((clamped-upper (default-to MAX-U16 upper)))
    (init { word: u0, fields: fields, valid: true, max: clamped-upper })
    (out (fold iter-pack-u16 ITER-UINT-8 init))
    (valid (get valid out))
    (valid? (asserts! valid ERR-INVALID-U16))
    (word (get word out))
    (ok word)))

(define-private (iter-pack-u16 (i uint) (acc {word: uint, fields: (list 8 uint), valid: bool, max: uint}))
  (let ((fields (get fields acc))
        (max (get max acc))
        (pos (unwrap-panic (element-at fields i)))
        (valid (get valid acc))
        (valid? (asserts! (or (not valid) (<= pos max))
                          (merge acc { valid: false }))))
    (word (get word acc))
    (mul (* i BIT-U16))
    (offset (pow u2 mul))
    (shiffl (* pos offset))
    (nword (+ word shiffl))
    { fields: fields, word: nword, valid: true, max: max })))
```

The first error is the wrong clamping:

```
clamped-upper (default-to MAX-U16 upper)
```

This is not limiting the value of `upper` to `MAX-U16`; it is only assigning `MAX-U16` if the `upper` value passed is none. This means the function can potentially accept values of `upper` greater than `MAX-U16` without raising an error. This is problematic because each value in the list is stored in a 16-bit slot within the word, so no value in the list should exceed `MAX-U16`.

At the moment, this does not pose any risk because the function is called in only two places in the codebase: one passing `BPS` and the other passing `none`. Therefore, `clamped-upper` is never greater than `MAX-U16`. However, future changes may allow a user-supplied value for this limit, which could introduce a serious issue. If values in the list are allowed to exceed the maximum that fits in a slot, the subsequent slots could be corrupted.

The second error lies in how the validity of the slot values is assessed:

```
(valid? (asserts! (or (not valid) (<= pos max))
                  (merge acc { valid: false })))
```

If the value of a list element is greater than the upper max value, the `asserts` fail and the fold returns early with the `acc` object having `valid` set to `false`.

However, the asserts are skipped when `valid` is already `false`. In that case, execution continues, and as we can see, the last line returns `acc` with `valid` set to `true`, regardless of its previous value.

This means that the only case where the call to `pack-u16` will revert with `ERR-INVALID-U16` is when the *last* list element is invalid. If there is an invalid element in any other position in the list, the execution will still succeed; the only effect is that the slots corresponding to invalid values will be zeroed (because the failing asserts prevent those word updates).

In this scenario, the caller — `dao-executor` via `set-points-util` or `set-points-rate` — may not notice that the values were stored incorrectly. This could lead to an incorrect interest-rate calculation or even a denial of service due to underflows in `linear-interpolate`.

Impact

The severity is assessed as Medium, because the potential impact is High, even though the likelihood of it occurring — since it would require an admin mistake — is Low.

Recommended Mitigation

For the first error, ensure that the value of `clamped-upper` is set to `MAX-U16` if `upper` is none or if it is greater than `MAX-U16`.

For the second error, ensure that the `valid` value in the folded loop is never set back to `true` once it has been set to `false` in any iteration.

Proof of Concept

See file `tests/greybeard/neumo.test.ts` in the Appendix.

Comments

Zest

I'm not sure about severity just because it requires a DAO gated path to be called and it requires an input mistake from us, however it is valid and have fixed it here: [b8f80c89b32dcba774a32c3086970b243ce2f2f0](#)

[M-9] egroup LTVs must always be monotonically decreasing when one egroup is a superset of another

Description

Misconfigurations of egroups are easy to do. This is evidenced by fact that a misconfiguration exists in the test code already.

One invariant that must hold hold for egroups is that *LTV values of an egroup that is a super set of another egroup must be lower than its LTVs*.

In the test code we have `proposal-create-multiple-egroups.clar` with

```
;; Egroup 3: ststx collateral + USDC debt
;; MASK = 2^1 + 2^2 + 2^67 = 4 + 147573952589676412928 = 147573952589676412932
(try! (contract-call? .egroup insert {
  MASK: u147573952589676412932,
  LIQ-CURVE-EXP: u20000,
  LIQ-PENALTY-MIN: u500,
  LIQ-PENALTY-MAX: u1000,
  LTV-BORROW: u5000,                ;; 50% - Lower quality collateral
  LTV-LIQ-PARTIAL: u7000,
  LTV-LIQ-FULL: u8500
}))
```

and

```
;; Egroup 4: sBTC + ststx collateral + USDC debt
;; MASK = 2^1 + 2^2 + 2^67 = 2 + 4 + 147573952589676412928 = 147573952589676412934
(try! (contract-call? .egroup insert {
  MASK: u147573952589676412934,
  LIQ-CURVE-EXP: u20000,
  LIQ-PENALTY-MIN: u500,
  LIQ-PENALTY-MAX: u1000,
  LTV-BORROW: u6000,                ;; 60% - Mixed collateral quality
  LTV-LIQ-PARTIAL: u7500,
  LTV-LIQ-FULL: u8500
}))
```

)))

This means that a user who has become insolvent while using stSTX collateral and borrowing USDC could just add a dust amount of sBTC and potentially become solvent. This is because LTV-LIQ-PARTIAL for Egroup 4 is $u7500 > u7000$ (for Egroup 3)

Impact

User's can add dust amounts of collateral to make a liquidatable position solvent.

Recommended Mitigation

Add a check for the invariant that if an egroup is a superset of another it must have lower LTVs.

Proof of Concept

See file `tests/greybeard/egroups.test.ts` in the Appendix.

Comments

Zest

Fixed in commit [92108d1fd073f38646cf1d58b66ce05c6e4b8c1a](#)

[M-10] Extremely frequent calling of accrue could lead to z-tokens accruing no value and borrow index growing faster than it should

Description

A call to accrue will

- always increase *borrow index* by at least 1 due to rounding up.
- round down the *liquidity index* to zero if the time between calls to accrue is small enough.

An attacker could leverage this fact to extract extra interest payments from borrowers.

For example, for interest rates of 3% ($u300$) if you call accrue every 5 seconds you will round up to multiplier of 1.00000001 every time.

This means interest will compound much faster. Over the course of one year this comes to roughly 6.3% APY because $1.00000001^{365 * 86400 / 5} = 1.06307200$.

Given that Stacks contract calls can sometimes cost as little as 5000 uSTX this means you could get a great return on your deposits for very little.

$5000 \text{ uSTX} * 365 * 86400 / 5 = 31,536 \text{ STX}$ over the course of the year.

Since you are getting 3.3% more this means that you break even at

$$\begin{aligned}x \cdot 0.033 &= 31,636 \\x &= 955,636 \text{ STX}\end{aligned}$$

This is obviously a very large break-even point but shows that it would be possible, in principal, to extract more interest from borrows than it would cost to perform this attack.

Liquidity Index

Also the liquidity index multiplier would stay at 1.000000 (since it is rounded down, as opposed to index being rounded up)

This affects the value of a zToken as collateral.

Impact

Recommended Mitigation

Increase the PRECISION constant so that the rounding error does not lead to excessive growth of `index` and no growth of `lindex`. Aave V3 uses `1e27` but this may be excessive.

Comments

Zest

Fixed in commit [e0393f5b7dd1916c821675cf4f517c8c316a6662](#).

10 Low Findings

[L-1] Oracle publish date could be in the future

Description

Neither Pyth nor Dia rely on `stacks-block-time` for timestamping, so the freshness check in `oracle-timestamp-fresh` could theoretically revert due to `(delta (- stacks-block-time ts))` if the oracle-provided timestamp is in the future. This would cause a temporary denial of service (DoS) until the next block is mined.

```
(define-private (oracle-timestamp-fresh (ts uint) (prev uint) (type (buff 1)) (ident (buff 32)))
  (let ((delta (if (> ts stacks-block-time)
                   u0
                   (- stacks-block-time ts))))
    (max-staleness (get-feed-max-staleness type ident)))
  (and
   (<= delta max-staleness)
   (>= ts prev))))
```

Impact

The impact is considered Low due to the low likelihood of this scenario occurring, and because the denial of service would likely be resolved in the next block.

Recommended Mitigation

Set `delta` to zero if the `ts` timestamp is greater than `stacks-block-time`.

Comments

Zest

Fixed in commit [37bdfb543be4c8550755a3f7315deba3cac2a95f](#)

[L-2] Wrong cap in system-borrow function

Description

When borrowing, the amount passed to the `vault-* borrow` function is capped as follows:

```
(asserts! (<= (+ d amt) a) ERR-INSUFFICIENT-VAULT-LIQUIDITY)
```

where `d` is the current debt, and `a`:

```
(a (var-get assets))
```

is a data variable that is:

- increased on deposit
- reduced on redeem
- increased by interested-paid on system-repay
- increased by fee on flashloan

Notably, `assets` does not account for borrows at all. It is not a measure of the vault's actual token balance of the underlying asset.

The cap should be applied using the actual borrowable assets available in the vault.

Impact

The impact is Low, since it is not possible to borrow more than the available balance in the contract anyway.

Recommended Mitigation

The correct cap would be `assets - total-borrowed`, which is the value returned by the `available` function.

Comments

Zest

Fixed in commit [3a7463f1b9b184d2f9ab518d36afe5173821ff87](#)

[L-3] Rounding direction in system-borrow should match borrow

Description

In `vault-sbtc.system-borrow` the `principal-scaled` amount is increased by `scaled-amt` which is rounded-down.

```
(define-public (system-borrow (amt uint))
  (let (
    ...
    (p (var-get principal-scaled))
    ...
    (scaled-amt (/ (* amt PRECISION) idx))
    (p-new (+ p scaled-amt)))
    ...
    (var-set principal-scaled p-new)
```

However, in `market.borrow` a call is made to `market-vault.debt-add-scaled` using `sdelta` which uses the same calculation but rounded up.

```
(define-public (borrow (ft <ft-trait>) (amount uint) (account principal))
  ...
  (let ((bi (get index (unwrap-panic (get-cached-indexes aid))))
        (sdelta (mul-div-up amount PRECISION bi)))
    (try! (contract-call? .market-vault
      debt-add-scaled
      account
      sdelta
      aid)))
```

For a borrow of exactly `u1` this will mean that `(is-eq principal-scaled u0)` inside the vault.

On a subsequent call to `market.repay`

- `vault-*.system-repay` will be called
- `d` will be calculated as `u0` via a call to `(debt)` which calls `reserve-calculator.calc-cumulative-debt` passing in `(is-eq p u0)`.
- `pratio` will be calculated via a call to `(calc-principal-ratio-reduction actual-amount p d)` where both `p` and `d` are `u0`.
- this calls `reserve-calculator.calc-principal-ratio-reduction` which will evaluate it's body with `(mul-div-down amount u0 u0)` and cause a division by zero runtime error.

Impact

The only real impact is DoS of repay on a vault where `(is-eq principal-scaled u0)`. However, this will almost never occur since `principal-scaled` will almost never be `u0`.

The only way to have a non-zero total debt but `(is-eq principal-scaled u0)` is to continuously borrow `u1` which will almost never happen in practice.

Recommended Mitigation

Change the following line in `system-borrow`

```
- (scaled-amt (/ (* amt PRECISION) idx))  
+ (scaled-amt (mul-div-up amt PRECISION idx))
```

Comments

Zest

Fixed in commit [464aa21fea81dd422fb4f38731b9356b2a88a376](#)

[L-4] dao-executor can be initialized multiple times

Description

The `dao-executor` contract is supposed to be callable only by the `dao-multisig` contract after the deployer has initialized it via a call to `init`.

Nothing prevents the deployer from calling `init` again at any time and changing the `impl` contract, allowing them to bypass the multisig threshold.

This raises two potential concerns:

- 1) The deployer effectively acts as a super admin who can do anything.
- 2) If the deployer's keys are compromised, all the vaults and the treasury are at risk.

Impact

The severity is assessed as Low because, although the impact is High, the likelihood of it occurring is very Low.

Recommended Mitigation

Assert that the `impl` variable is none at the beginning of the `init` function.

Comments

Zest

Fixed in commit [c1bd6533dac9b24e3e92a74ad26645834d561337](#)

[L-5] Proposal expiry time inconsistency

Description

Function `set-default-expiry-duration` only checks that the duration parameter is greater than zero.

However, the `execute` function requires that the current block time has not yet reached `created-at + expiry` *and* that it is after `created-at + TIMELOCK`.

`TIMELOCK` cannot be modified and is set to 86400 seconds. The default expiry time, however, can be modified via a call to `set-default-expiry-duration` to a value smaller than `TIMELOCK`, creating a situation where normal proposals can never be executed:

```
(mature-at (+ created-at TIMELOCK))
...
(asserts! (< stacks-block-time expires-at) ERR-PROPOSAL-EXPIRED)
...
(>= stacks-block-time mature-at)
```

We refer to normal proposals here, because urgent proposals can always be executed by bypassing the maturity check.

Impact

The impact is Low because it can only occur due to insufficient input validation and an admin mistake.

Recommended Mitigation

Function `set-default-expiry-duration` should check that duration is `>= TIMELOCK`.

Comments

Zest

Fixed in commit [99a94922fccda3bc8431639a8bb09da27d9f9a9f](#)

[L-6] Wrong asserts in new liquidate function

Description

This issue is written with respect to commit [d1bd2f0df47704dfbbaa0b064dc7fdad7dc40ce1](#) which implements the new liquidation function.

```
(asserts! (> debt-final u0) ERR-ZERO-LIQUIDATION-AMOUNTS)
(asserts! (> coll-actual u0) ERR-ZERO-LIQUIDATION-AMOUNTS)
```

should be

```
(asserts! (> debt-to-repay u0) ERR-ZERO-LIQUIDATION-AMOUNTS)
(asserts! (> coll-final) ERR-ZERO-LIQUIDATION-AMOUNTS)
```

Impact

Very minor since it will almost never be the case that liquidation can occur and either of these values will be zero.

Recommended Mitigation

```
- (asserts! (> debt-final u0) ERR-ZERO-LIQUIDATION-AMOUNTS)
+ (asserts! (> debt-to-repay u0) ERR-ZERO-LIQUIDATION-AMOUNTS)
- (asserts! (> coll-actual u0) ERR-ZERO-LIQUIDATION-AMOUNTS)
+ (asserts! (> coll-final) ERR-ZERO-LIQUIDATION-AMOUNTS)
```

Comments

Zest

Fixed in commit [92108d1fd073f38646cf1d58b66ce05c6e4b8c1a](#)

[L-7] No protocol fees received on flashloan

Description

The vault accrue functions all mint shares for the treasury using the fee-reserve data variable to determine the rate. However, there is no minting of shares for the treasury in the `flashloan` function.

Impact

The impact is loss of fee revenue for the Zest protocol.

Recommended Mitigation

Add minting of treasure shares in the `flashloan` function.

Comments

Zest

We implemented a different solution where a direct fee transfer is made to the treasury.

11 Informational Findings

[I-1] Replace bit-handling functions in pack with ones using Clarity bit-wise operations

Description

The current bit-handling functions are almost certainly slower than ones written with equivalent bit-wise operations.

```
(define-read-only (has-bit (v uint) (p uint))
  (let ((div (pow u2 p))
        (shiftr (/ v div))
        (bit (mod shiftr u2)))
    (is-eq bit u1)))

(define-read-only (set-bit (v uint) (p uint))
  (let ((div (pow u2 p))
        (shiftr (/ v div))
        (bit (mod shiftr u2))
        (base (if (is-eq bit u0)
                   div
                   u0)))
    (+ v base)))

(define-read-only (clear-bit (v uint) (p uint))
  (let ((div (pow u2 p))
        (shiftr (/ v div))
        (bit (mod shiftr u2))
        (base (if (is-eq bit u1)
                   div
                   u0)))
    (- v base)))
```

Recommended Mitigation

```
(define-read-only (has-bit (v uint) (p uint))
  (> (bit-and v (pow u2 p)) u0))

(define-read-only (set-bit (v uint) (p uint))
  (bit-or v (pow u2 p)))

(define-read-only (clear-bit (v uint) (p uint))
  (bit-and v (bit-not (pow u2 p))))
```

Given how small these functions are they should probably just be inlined.

Comments

Zest

Greybeard Security

[I-2] Change field names for better readability

Description

Many data structures have fields that could have more readable names.

Many functions have parameters that could have better names.

Recommended Mitigation

Standardise names everywhere

Standardise parameters of functions:

- `asset-id` always stands for an asset id. i.e. $0 \leq \text{asset-id} < 64$.
- `id` always stands for account ID

Inside let-bindings:

- `aid` for asset-ids
- `id` for account ID

`pause-states`

It's easy to get confused what each field means. e.g. does `(is-eq (get redeem (var-get pause-states)) true)` mean redemption is enabled or paused?

```
(define-data-var pause-states
  {
-   deposit: bool,
+   is-deposit-paused: bool,
-   redeem: bool,
+   is-redeem-paused: bool,
-   borrow: bool,
+   is-borrow-paused: bool,
-   repay: bool,
+   is-repay-paused: bool,
-   accrue: bool,
+   is-accrue-paused: bool,
-   flashloan: bool
+   is-flashloan-paused: bool
```

`mask-update private function`

```
- (define-private (mask-update (base uint) (pos uint) (is-collateral bool) (is-insert bool))
+ (define-private (mask-update (base-mask uint) (asset-id uint) (is-collateral bool) (is-insert bool))
```

`vault-sbtc.total-assets-preview`

`interest -> unpaid-interest`

```
(define-private (total-assets-preview)
  (let ((a (var-get assets))
        (cd (debt-preview))
        (tb (var-get total-borrowed)))
-    (interest (if (> cd tb) (- cd tb) u0)))
-    (+ a interest)))
+    (unpaid-interest (if (> cd tb) (- cd tb) u0)))
+    (+ a unpaid-interest)))
```

`vault-sbtc.accrue`

- `next -> next-idx`
- `nliq -> next-lidx`

[I-3] Wrong implementation of ubalance function in vault-sbtc

Description

In contract vault-sbtc, the function ubalance is defined as:

```
(define-private (ubalance)
  (unwrap-panic (contract-call? .sbtc get-balance SELF)))
```

It should be:

```
(define-private (ubalance)
  (unwrap-panic (contract-call? .sbtc get-balance-available SELF)))
```

Because get-balance returns the sum of unlocked and locked sbtc balance. Locked balance cannot be transferred.

Impact

This has no real impact so we assessed it as an Informational.

It's just a formal issue, because in practice the contract should never have a locked balance, as there's no way for it to call initiate-withdrawal-request in the sbtc-withdrawal contract.

Recommended Mitigation

```
(define-private (ubalance)
  - (unwrap-panic (contract-call? .sbtc get-balance SELF)))
  + (unwrap-panic (contract-call? .sbtc get-balance-available SELF)))
```

Comments

Zest

Fixed in commit [afccf5f190d7dcef10a09d320d7c51da7747af3b](#)

12 Appendix

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

tests/greybeard/mocks/proposal-test.clar

```
(impl-trait .dao-traits.proposal-script)

(define-constant DEPLOYER tx-sender)
(define-constant ERR-AUTH (err u100001))

(define-constant TEST_SET_FEED_MAX_STALENESS u0)
(define-constant TEST_PREPARE_VAULTS u1)
(define-constant TEST_DISABLE_SBTC_COLLATERAL u2)
(define-constant TEST_DISABLE_SBTC_BORROWING u3)
(define-constant TEST_INSERT_EGROUPS u4)
(define-constant TEST_PAUSE_STATES u5)
(define-constant TEST_SET_POINTS u6)
(define-constant TEST_SET_FEE_RESERVE u7)
(define-constant TEST_SET_THRESHOLD u8)
(define-constant TEST_INIT_VAULTS_WITH_INTEREST_RATES u9)
(define-constant TEST_SET_STX_FEED_ID u10)

(define-constant STX-FEED-ID 0xec7a775f46379b5e943c3526b1c8d54cd49749176b0b98e02dde68d1bd335c17)

(define-data-var function-id uint u0) ;; for setting which test you want to run

(define-data-var pause-states
{
  deposit: bool,
  redeem: bool,
  borrow: bool,
  repay: bool,
  accrue: bool,
  flashloan: bool
}
{
  deposit: false,
  redeem: false,
  borrow: false,
  repay: false,
  accrue: false,
  flashloan: false
})

(define-public (set-function-id (fid uint))
(begin
  (asserts! (is-eq contract-caller DEPLOYER) ERR-AUTH)
  (ok (var-set function-id fid))))

(define-public (set-pause-states (states {deposit: bool, redeem: bool, borrow: bool, repay: bool, accrue: bool, flashloan: bool}))
(begin
  (var-set pause-states states)
  (ok true)))

;; Add one
(define-public (execute)
(let ((fid (var-get function-id)))
  (if (is-eq fid TEST_SET_FEED_MAX_STALENESS) (test-set-feed-max-staleness)
    (if (is-eq fid TEST_PREPARE_VAULTS) (test-prepare-vaults)
      (if (is-eq fid TEST_DISABLE_SBTC_COLLATERAL) (test-disable-sbtc-collateral)
        (if (is-eq fid TEST_DISABLE_SBTC_BORROWING) (test-disable-sbtc-borrowing)
          (if (is-eq fid TEST_INSERT_EGROUPS) (test-insert-egroups)
            (if (is-eq fid TEST_PAUSE_STATES) (test-pause-states)
              (if (is-eq fid TEST_SET_POINTS) (test-set-points)
                (if (is-eq fid TEST_SET_FEE_RESERVE) (test-set-fee-reserve)
                  (if (is-eq fid TEST_SET_THRESHOLD) (test-set-threshold)
                    (if (is-eq fid TEST_INIT_VAULTS_WITH_INTEREST_RATES) (test-init-vaults-with-interest-rates)
                      (if (is-eq fid TEST_SET_STX_FEED_ID) (test-set-stx-feed-id)
                        (ok true))))))))))))))

(define-private (test-set-feed-max-staleness)
(begin
  (print { sender: tx-sender })
  (try! (contract-call? .market set-feed-max-staleness 0x17 0xcafe u42))
  (ok true)))

(define-private (test-prepare-vaults)
(begin
  (print { sender: tx-sender })
  (try! (contract-call? .market-vault set-impl .market))
  (try! (contract-call? .vault-sbtc set-cap-supply u1000000000000000000))
  (try! (contract-call? .vault-sbtc initialize))
```

[illegible]

```

(begin
  (try! (contract-call? .vault-usdc set-fee-reserve u1000))
  (ok true)))

(define-private (test-set-threshold)
  (begin
    (try! (contract-call? .dao-multisig set-threshold u4))
    (ok true)))

(define-constant CAP u1000000000000)

(define-private (test-init-vaults-with-interest-rates)
  (begin
    ;; Set vault-sbtc caps before initialization
    (try! (contract-call? .vault-sbtc set-cap-supply CAP))
    (try! (contract-call? .vault-sbtc set-cap-debt CAP))

    (try! (contract-call? .vault-sbtc set-points-util (list u0 u1425 u2850 u4275 u5700 u7125 u8550 u10000)))
    (try! (contract-call? .vault-sbtc set-points-rate (list u300 u400 u500 u600 u700 u800 u900 u1000)))

    ;; Set vault-usdh caps before initialization
    (try! (contract-call? .vault-usdh set-cap-supply CAP))
    (try! (contract-call? .vault-usdh set-cap-debt CAP))
    (try! (contract-call? .vault-usdh set-points-util (list u0 u1425 u2850 u4275 u5700 u7125 u8550 u10000)))
    (try! (contract-call? .vault-usdh set-points-rate (list u300 u400 u500 u600 u700 u800 u900 u1000)))

    ;; Set vault-usdc caps before initialization
    (try! (contract-call? .vault-usdc set-cap-supply CAP))
    (try! (contract-call? .vault-usdc set-cap-debt CAP))
    (try! (contract-call? .vault-usdc set-points-util (list u0 u1425 u2850 u4275 u5700 u7125 u8550 u10000)))
    (try! (contract-call? .vault-usdc set-points-rate (list u300 u400 u500 u600 u700 u800 u900 u1000)))

    ;; Set vault-ststx caps before initialization
    (try! (contract-call? .vault-ststx set-cap-supply CAP))
    (try! (contract-call? .vault-ststx set-cap-debt CAP))
    (try! (contract-call? .vault-ststx set-points-util (list u0 u1425 u2850 u4275 u5700 u7125 u8550 u10000)))
    (try! (contract-call? .vault-ststx set-points-rate (list u300 u400 u500 u600 u700 u800 u900 u1000)))

    ;; Initialize vault-sbtc (mints minimum liquidity)
    (try! (contract-call? .vault-sbtc initialize))

    ;; Initialize vault-usdh (mints minimum liquidity)
    (try! (contract-call? .vault-usdh initialize))

    ;; Initialize vault-usdc (mints minimum liquidity)
    (try! (contract-call? .vault-usdc initialize))

    ;; Initialize vault-ststx (mints minimum liquidity)
    (try! (contract-call? .vault-ststx initialize))

    ;; Authorize market contract in vault-sbtc
    (try! (contract-call? .vault-sbtc set-authorized-contract .market true))

    ;; Authorize market contract in vault-usdh
    (try! (contract-call? .vault-usdh set-authorized-contract .market true))

    ;; Authorize market contract in vault-usdc
    (try! (contract-call? .vault-usdc set-authorized-contract .market true))

    ;; Authorize market contract in vault-ststx
    (try! (contract-call? .vault-ststx set-authorized-contract .market true))
    (ok true)))

(define-private (test-set-stx-feed-id)
  (begin
    (try! (contract-call? .assets update .ststx { callcode: none, ident: STX-FEED-ID, type: 0x00}))
    (ok true)))

```

tests/greybeard/egroups.test.ts

```

import {
  deployer,
  alice,
  bob,
  charlie,
  executeDaoProposal,
  market,
  marketVault,
  vaultUsdc,
  vaultUsdh,
  egroup,
  contracts
} from '../unit/initialization/helpers';

```

```

import {
  init_pyth,
  set_initial_price,
  set_price,
  scalePriceForPyth,
  PythFeedIds
} from '../unit/helpers/pyth-helpers';

import { sbtc, usdc, usdh, ststx, buffToUint, initializeProtocolGreybeard, executeDaoProposalTest, proposalTest } from './helpers'; // Extra
↳ Greybeard helpers

import { beforeEach, describe, it, expect } from 'vitest';
import { rov, txOk, txErr, tx, } from '@clarigen/test';

const STX_FEED_ID = Buffer.from('ec7a775f46379b5e943c3526b1c8d54cd49749176b0b98e02dde68d1bd335c17', 'hex');

const ALL_ONES = 340282366920938463463374607431768211455n;

const checkEgroup = (account: string, expectedMask: bigint) => {
  const mask = rov(marketVault.position({ account, enabledMask: ALL_ONES })), account).mask;
  expect(mask).to.be.equal(expectedMask);
  const result = rov(egroup.resolve({ mask }));
  console.log(result);
  return result;
};

const tokIds = market.constants;

const preambleForStSTXCollatWithUSDCDebt = async () => {
  console.log("----- 1. Alice adds collateral -----");
  txOk(market.collateralAdd({ amount: 100_000_000000n, ft: ststx.identifier, account: alice }), alice);

  checkEgroup(alice, 2n**tokIds.STSTX);

  console.log("----- 2. Alice borrows at max LTV of 50% -----");
  txOk(market.borrow({ ft: usdc.identifier, amount: 50_000_000000n, account: alice }), alice);

  console.log(rov(marketVault.position({ account: alice, enabledMask: ALL_ONES })));

  checkEgroup(alice, 2n**tokIds.STSTX + 2n**(64n + tokIds.USDC));

  /*
   * l / (c * price) = 50%  l / (c * p) = 70%  l = 70%(c*p), p = l / 70% * c
   */

  console.log("----- 3. Lower price of STX to 50,000/71,000 (so LTV is 71%) -----");
  await set_price(STX_FEED_ID, 7042_2535n, -8, deployer);
};

describe('Greybeard Security', () => {
  beforeEach(async () => {
    initializeProtocolGreybeard();
    init_pyth(deployer);
    executeDaoProposal(contracts.proposalSetPriceStaleness);
    executeDaoProposal(contracts.proposalCreateMultipleEgroups);
    executeDaoProposalTest(proposalTest.constants.TEST_SET_STX_FEED_ID)

    // BTC at $100,000
    await set_price(
      PythFeedIds.BTC,
      scalePriceForPyth(100_000, 8),
      -8,
      deployer
    );

    // STX at $1
    await set_price(
      STX_FEED_ID,
      scalePriceForPyth(1, 8),
      -8,
      deployer
    );

    // USDC at $1
    await set_initial_price(
      PythFeedIds.USDC,
      scalePriceForPyth(1, -8),
      deployer
    );

    // Mint USDC to Charlie
    txOk(usdc.mint({
      amount: 1_000_000_000000n,
      account: charlie
    }));
  });
});

```

```

   )), deployer);

    // Mint sbtc to Alice
    txOk(sbtc.mint({
      amount: 10_0000000n,
      recipient: alice
    }), deployer);

    // Mint ststx to Alice
    txOk(contracts.ststx.mint({
      amount: 100_000_000000n,
      account: alice
    }), deployer);

    txOk(usdc.mint({ amount: 100_000_000000n, account: bob}), deployer);
    txOk(usdh.mint({ amount: 100_000_000000n, account: bob}), deployer);

    // Bob LPS some USDC
    txOk(vaultUsdc.deposit({
      amt: 100_000_000000n,
      minOut: 0n,
      recipient: bob
    }), bob);

    // Bob LPS some USDH
    txOk(vaultUsdh.deposit({
      amt: 100_000_000000n,
      minOut: 0n,
      recipient: bob
    }), bob);

  });

  it ("shows that Charlie can liquidate when sBTC not added as collateral", async () => {
    console.log("----- 5. Charlie can liquidate since LTV is not 71% -----");
    await preambleForStSTXCollatWithUSDCDebt();
    txOk(market.liquidate({
      borrower: alice,
      collateralFt: ststx.identifier,
      debtAmount: 50_000_000000n,
      debtFt: usdc.identifier}),
      charlie);
  });

  it ("strict superset should not have higher LTV", async () => {
    await preambleForStSTXCollatWithUSDCDebt();
    console.log("----- 4. Alice adds dust amount of sBTC to make LTV-PARTIAL higher (75%) -----");

    txOk(market.collateralAdd({ amount: 1n, ft: sbtc.identifier, account: alice }), alice);
    checkEgroup(alice, 2n**tokIds.STSTX + 2n**tokIds.SBTC + 2n**(64n + tokIds.USDC));

    console.log("----- 5. Charlie cannot liquidate -----");
    expect(() => tx(market.liquidate({
      borrower: alice,
      collateralFt: ststx.identifier,
      debtAmount: 1_000_000000n,
      debtFt: usdc.identifier}),
      charlie)).toThrow(new RegExp("ArithmeticUnderflow"));
  });
});

```

tests/greybeard/helpers.ts

```

import { project, accounts } from '../clarigen-types';
import { projectFactory } from '@clarigen/core';
import { deployer, executeDaoProposal, initializeDAO, registerAssets, initializeMarketVault,
  sbtcToken, daoExecutor, usdhToken, vaultSbtc, vaultUsdc, vaultUsdh, vaultStstx, market
} from '../unit/initialization/helpers';
import { txOk, rov } from '@clarigen/test';

export const contracts = projectFactory(project, "simnet");

// export more users

export const daniel = accounts.wallet_4.address;
export const eve = accounts.wallet_5.address;

// Export contracts as you need to test them
export const proposalTest = contracts.proposalTest;
export const sbtc = contracts.sbtc;
export const ststx = contracts.ststx;
export const usdc = contracts.usdc;
export const usdh = contracts.usdh;

```

```

export function buffToUint(buff: Uint8Array): bigint {
  let n = 0n;
  for (let i in buff) {
    n = BigInt(buff[i]) + (n * 256n);
  }
  return n;
}

export function executeDaoProposalTest(id: bigint) {
  txOk(proposalTest.setFunctionId(id), deployer);
  executeDaoProposal(proposalTest);
}

export function initializeVaultsGreybeard() {
  // Mint tokens to dao-executor for vault initialization (minimum liquidity = 1000)
  const MINIMUM_LIQUIDITY = 1000n;
  txOk(sbtcToken.mint(MINIMUM_LIQUIDITY, daoExecutor.identifier), deployer);
  txOk(usdhToken.mint(MINIMUM_LIQUIDITY, daoExecutor.identifier), deployer);
  txOk(contracts.usdc.mint(MINIMUM_LIQUIDITY, daoExecutor.identifier), deployer);
  txOk(contracts.ststx.mint(MINIMUM_LIQUIDITY, daoExecutor.identifier), deployer);

  executeDaoProposalTest(proposalTest.constants.TEST_INIT_VAULTS_WITH_INTEREST_RATES);

  // Verify authorization - execute read-only calls
  const sbtcAuthorized = rov(vaultSbtc.isAuthorizedContract(market.identifier));
  const usdhAuthorized = rov(vaultUsdh.isAuthorizedContract(market.identifier));
  const usdcAuthorized = rov(vaultUsdc.isAuthorizedContract(market.identifier));
  const ststxAuthorized = rov(vaultStstx.isAuthorizedContract(market.identifier));

  return {
    sbtcAuthorized,
    usdhAuthorized,
    usdcAuthorized,
    ststxAuthorized
  };
}

export function initializeProtocolGreybeard() {
  const daoStatus = initializeDAO();

  // 2. Register assets
  const assetIds = registerAssets();

  // 3. Initialize vaults
  const vaultStatus = initializeVaultsGreybeard();

  // 4. Initialize market-vault
  const marketVaultStatus = initializeMarketVault();

  return {
    dao: daoStatus,
    assets: assetIds,
    vaults: vaultStatus,
    marketVault: marketVaultStatus
  };
}

```

tests/greybeard/liquidation.test.ts

```

import {
  deployer,
  alice,
  bob,
  charlie,
  executeDaoProposal,
  market,
  initializeProtocol,
  vaultSbtc,
  egroup,
  contracts
} from '../unit/initialization/helpers';

import {
  init_pyth,
  set_initial_price,
  set_price,
  scalePriceForPyth,
  PythFeedIds
} from '../unit/helpers/pyth-helpers';

import { proposalTest, sbtc, usdc, buffToUint, executeDaoProposalTest } from './helpers'; // Extra Greybeard helpers

```

```

import { beforeEach, describe, it, expect } from 'vitest';
import { rov, txOk, tx } from '@clarigen/test';

describe('Greybeard Security', () => {

  const SBTC_PRICE = 60_000n;
  const USDC_COLLAT = 100_000_000000n;

  const preamble = async () => {

    const ltvResult = rov(egroup.lookup({ id: egroup.constants.DEFAULTMASKID }), alice);
    const ltvBorrow = buffToUint(ltvResult.LTVBORROW);
    console.log("ltvBorrow", ltvBorrow);

    // Mint sbtc to Alice
    txOk(sbtc.mint({
      amount: 10_0000_0000n,
      recipient: alice
    }), deployer);

    // Mint usdc to Bob
    txOk(usdc.mint({
      amount: 1_000_000_000000n,
      account: bob
    }), deployer);

    // Mint sbtc to Charlie
    txOk(sbtc.mint({
      amount: 1_0000_0000n,
      recipient: charlie
    }), deployer);

    // Alice, the LP, deposits
    txOk(vaultSbtc.deposit({
      amt: 10_0000_0000n,
      minOut: 0n,
      recipient: alice
    }), alice);

    // console.log("vault balance", rovOk(sbtc.getBalance(vaultSbtc.identifer)));

    txOk(market.collateralAdd({
      ft: usdc.identifier,
      amount: USDC_COLLAT,
      account: bob
    }), bob);

    // Borrow the maximum
    const borrowAmount = (USDC_COLLAT * ltvBorrow / 10_000n) * 10n ** (8n - 6n) / SBTC_PRICE;
    console.log("borrowAmount", borrowAmount);

    txOk(market.borrow({
      ft: sbtc.identifier,
      amount: borrowAmount,
      account: bob
    }), bob);

    // Now change the price of sBTC

    await set_price(PythFeedIds.BTC, scalePriceForPyth(91_000, 8), -8, deployer);

    // Now disable sBTC for borrowing
    executeDaoProposalTest(proposalTest.constants.TEST_DISABLE_SBTC_BORROWING);

    return borrowAmount;
  }

  beforeEach(async () => {
    initializeProtocol();
    init_pyth(deployer);
    executeDaoProposal(contracts.proposalSetPriceStaleness);

    // 3. Set initial prices via Pyth
    // BTC at $60,000
    await set_price(
      PythFeedIds.BTC,
      scalePriceForPyth(60_000, 8),
      -8,
      deployer
    );

    // USDC at $1
    await set_initial_price(
      PythFeedIds.USDC,
      scalePriceForPyth(1, -8),

```

```

    deployer
  );

});

it('health checks fail to take into account existing debt when asset disabled - preventing liquidation', async () => {

  const borrowAmount = await preamble();

  expect(() => tx(market.liquidate({
    borrower:    bob,
    collateralFt: usdc.identifier,
    debtFt:      sbtc.identifier,
    debtAmount:  borrowAmount
  })), charlie)).toThrow(new RegExp("ArithmeticUnderflow"));

});

it('health checks fail to take into account existing debt when asset disabled - withdrawal of collateral allowed', async () => {
  await preamble();

  txOk(market.collateralRemove({
    ft:    usdc.identifier,
    amount: USDC_COLLAT,
    account: bob
  })), bob);

});

});

```

tests/greybeard/neumo.test.ts

```

import { describe, it, expect, beforeEach } from 'vitest';
import { rov, rovOk, txOk } from '@clarigen/test';

import {
  deployer,
  alice,
  bob,
  charlie,
  initializeProtocol,
  executeDaoProposal,
  contracts,
} from '../unit/initialization/helpers';

import {
  proposalTest,
  executeDaoProposalTest,
  sbtc,
  buffToUint,
} from '../helpers';

import {
  init_pyth,
  set_initial_price,
  PythFeedIds,
  set_price,
  scalePriceForPyth,
} from '../unit/helpers/pyth-helpers';
import { assert } from 'console';

describe('PoCs', () => {
  beforeEach(async () => {
    initializeProtocol();
    init_pyth(deployer);

    // CRITICAL: Execute staleness proposal BEFORE setting prices
    console.log('\n=== Executing Staleness Proposal ===');
    executeDaoProposal(contracts.proposalSetPriceStaleness);
    const threshold = rovOk(contracts.market.getDefaultMaxStaleness());
    console.log('Staleness threshold after proposal:', threshold);

    // Set prices - MUST await these async calls!
    await set_initial_price(PythFeedIds.BTC, scalePriceForPyth(60000, -8), deployer);
    await set_initial_price(PythFeedIds.USDC, scalePriceForPyth(1, -8), deployer);

    // Verify prices were stored
    console.log('\n=== Verifying Prices ===');
    const btcPrice = rov(contracts.pythStorageV4.getPrice({ priceIdentifier: PythFeedIds.BTC }));
    const usdcPrice = rov(contracts.pythStorageV4.getPrice({ priceIdentifier: PythFeedIds.USDC }));

```

```

    console.log('BTC price stored:', btcPrice);
    console.log('USDC price stored:', usdcPrice);
  });

it('disabling a collateral can turn a healthy position into a liquidatable one', () => {
  // Step 1: Setup collateral
  console.log('\nStep 1: Setting up collateral');
  const sbtcAmount = 100000000n; // 1 BTC
  txOk(contracts.sbtc.mint(sbtcAmount, alice), deployer);

  try {
    const addCollateralResult = txOk(
      contracts.market.collateralAdd(contracts.sbtc.identifier, sbtcAmount, alice),
      alice
    );
    console.log('collateral-add succeeded');
  } catch (e) {
    console.log('ERROR in collateral-add:', e);
  }

  // Step 2: Setup vault liquidity
  console.log('\nStep 2: Setting up vault liquidity');
  txOk(contracts.usdc.mint(100000000000n, bob), deployer);
  txOk(contracts.vaultUsdc.deposit(100000000000n, 0n, bob), bob);

  // Step 3: Try to borrow
  console.log('\nStep 3: Attempting to borrow');
  // With DEFAULT 40% LTV, max borrow = $24,000
  const borrowAmount = 24000000000n; // $24,000
  try {
    const borrowResult = txOk(
      contracts.market.borrow(contracts.usdc.identifier, borrowAmount, alice),
      alice
    );
    console.log('borrow succeeded with $24,000 (40% LTV)!');
  } catch (e) {
    console.log('ERROR in borrow:', e);
  }

  // Step 4: Disable collateral
  console.log('\nStep 4: Disable collateral');
  executeDaoProposalTest(proposalTest.constants.TEST_DISABLE_SBTC_COLLATERAL);

  // Step 5: Try to liquidate
  console.log('\nStep 5: Attempting to liquidate');
  const liquidateResult = txOk(
    contracts.market.liquidate(alice, contracts.sbtc.identifier, contracts.usdc.identifier, borrowAmount),
    alice
  );
});

it('repayment can turn a healthy position into a liquidatable one', () => {
  // Step 1: Insert egroups
  console.log('\nStep 1: Inserting egroup data');
  executeDaoProposalTest(proposalTest.constants.TEST_INSERT_EGROUPS);

  // Step 2: Setup collateral
  console.log('\nStep 2: Setting up collateral');
  const sbtcAmount = 200000000n; // 2 BTC
  txOk(contracts.sbtc.mint(sbtcAmount, alice), deployer);

  try {
    const addCollateralResult = txOk(
      contracts.market.collateralAdd(contracts.sbtc.identifier, sbtcAmount, alice),
      alice
    );
    console.log('collateral-add succeeded');
  } catch (e) {
    console.log('ERROR in collateral-add:', e);
  }

  // Step 3: Setup vault liquidity
  console.log('\nStep 3: Setting up vault liquidity');
  txOk(contracts.usdc.mint(100000000000n, bob), deployer);
  txOk(contracts.vaultUsdc.deposit(100000000000n, 0n, bob), bob);
  txOk(contracts.sbtc.mint(200000000n, bob), deployer);
  txOk(contracts.vaultSbtc.deposit(200000000n, 0n, bob), bob);

  // Step 4: Try to borrow usdc
  console.log('\nStep 4: Attempting to borrow USDC');
  const borrowAmount = 18000000000n; // $18,000
  try {
    const borrowResult = txOk(
      contracts.market.borrow(contracts.usdc.identifier, borrowAmount, alice),
      alice
    );
  }

```

```

    );
    console.log('borrow succeeded with $24,000 (40% LTV)!');
  } catch (e) {
    console.log('ERROR in borrow:', e);
  }
  let bitmap = rov(contracts.assets.getBitmap());
  let position = rov(contracts.marketVault.position(alice, bitmap));
  let egroup = rov(contracts.egroup.resolve(position.mask));
  console.log(position);
  console.log(egroup);

  // Step 5: Try to borrow sbtc
  console.log('\nStep 5: Attempting to borrow SBTC');
  // With this mask 65% LTV, max borrow left = 1 BTC
  const sbtcBorrowAmount = 130000000n; // 1.3 BTC
  try {
    const borrowResult = txOk(
      contracts.market.borrow(contracts.sbtc.identifier, sbtcBorrowAmount, alice),
      alice
    );
    console.log('borrow succeeded with 1.3 BTC (65% LTV)!');
  } catch (e) {
    console.log('ERROR in borrow:', e);
  }
  bitmap = rov(contracts.assets.getBitmap());
  position = rov(contracts.marketVault.position(alice, bitmap));
  egroup = rov(contracts.egroup.resolve(position.mask));
  console.log(position);
  console.log(egroup);

  // Step 6: Try to repay
  console.log('\nStep 6: Attempting to repay');
  try {
    const repayResult = txOk(
      contracts.market.repay(contracts.usdc.identifier, borrowAmount, alice),
      alice
    );
    console.log('repay succeeded with $18,000!');
  } catch (e) {
    console.log('ERROR in repay:', e);
  }

  bitmap = rov(contracts.assets.getBitmap());
  position = rov(contracts.marketVault.position(alice, bitmap));
  egroup = rov(contracts.egroup.resolve(position.mask));
  console.log(position);
  console.log(egroup);

  // Step 7: Try to liquidate (should not be possible after a repay)
  console.log('\nStep 7: Attempting to liquidate');
  const liquidateResult = txOk(
    contracts.market.liquidate(alice, contracts.sbtc.identifier, contracts.sbtc.identifier, sbtcBorrowAmount),
    alice
  );

  bitmap = rov(contracts.assets.getBitmap());
  position = rov(contracts.marketVault.position(alice, bitmap));
  egroup = rov(contracts.egroup.resolve(position.mask));
  console.log(position);
  console.log(egroup);
});

it('assets and shares preview is wrong when accrual is paused', () => {
  const ASSETS_AMOUNT: bigint = 1_0000_0000n;
  const SHARES_AMOUNT: bigint = 2_0000_0000n;

  // Step 1: Set points
  console.log('\nStep 1: Set points');
  executeDaoProposalTest(proposalTest.constants.TEST_SET_POINTS);

  // Step 2: Setup collateral
  console.log('\nStep 2: Setting up collateral');
  const sbtcAmount = 2000000000n; // 2 BTC
  txOk(contracts.sbtc.mint(sbtcAmount, alice), deployer);

  try {
    const addCollateralResult = txOk(
      contracts.market.collateralAdd(contracts.sbtc.identifier, sbtcAmount, alice),
      alice
    );
    console.log('collateral-add succeeded');
  } catch (e) {
    console.log('ERROR in collateral-add:', e);
  }

  // Step 3: Setup vault liquidity

```

```

console.log('\nStep 3: Setting up vault liquidity');
txOk(contracts.usdc.mint(36_000_000000n, bob), deployer);
txOk(contracts.vaultUsdc.deposit(36_000_000000n, 0n, bob), bob);

// Step 4: Try to borrow usdc
console.log('\nStep 4: Attempting to borrow USDC');
const borrowAmount = 19_000_000000n;
try {
  const borrowResult = txOk(
    contracts.market.borrow(contracts.usdc.identifier, borrowAmount, alice),
    alice
  );
  console.log(borrowResult);
  console.log(`borrow succeeded with ${borrowAmount}`);
} catch (e) {
  console.log('ERROR in borrow:', e);
}
let bitmap = rov(contracts.assets.getBitmap());
let position = rov(contracts.marketVault.position(alice, bitmap));
let egroup = rov(contracts.egroup.resolve(position.mask));
console.log("position", position);
console.log("egroup", egroup);

// Step 5: Pause accruals
console.log('\nStep 5: Pause accruals');
txOk(proposalTest.setPauseStates({deposit: false, redeem: false, borrow: false, repay: false, accrue: true, flashloan: false}), deployer);
executeDaoProposalTest(proposalTest.constants.TEST_PAUSE_STATES);

// Step 6: Get assets and shares preview
console.log('\nStep 6: Get assets and shares preview');

let assetsPreviewPre = rov(contracts.vaultUsdc.convertToAssets(ASSETS_AMOUNT));
let sharesPreviewPre = rov(contracts.vaultUsdc.convertToShares(SHARES_AMOUNT));
console.log('assetsPreview: ' + assetsPreviewPre.value);
console.log('sharesPreview: ' + sharesPreviewPre.value);

// Step 7: Time goes by...
console.log('\nStep 7: Time goes by...');
simnet.mineEmptyBlocks(600);

// Step 8: Get assets and shares preview after some blocks
console.log('\nStep 8: Get assets and shares preview');
let assetsPreviewPost = rov(contracts.vaultUsdc.convertToAssets(ASSETS_AMOUNT));
let sharesPreviewPost = rov(contracts.vaultUsdc.convertToShares(SHARES_AMOUNT));
console.log('assetsPreview: ' + assetsPreviewPost.value);
console.log('sharesPreview: ' + sharesPreviewPost.value);

/* Bug */
expect(assetsPreviewPost.value).greaterThanBigint(assetsPreviewPre.value);
expect(sharesPreviewPost.value).lessThanBigint(sharesPreviewPre.value);

let interestRate = rov(contracts.vaultUsdc.getInterestRate());
console.log('interestRate: ' + interestRate.value);

let utilization = rov(contracts.vaultUsdc.getUtilization());
console.log('utilization: ' + utilization.value);
});

it('collateral remove can turn a healthy position into a liquidatable one', () => {

  // Step 1: Insert egroups
  console.log('\nStep 1: Inserting egroup data');
  executeDaoProposalTest(proposalTest.constants.TEST_INSERT_EGROUPS);

  // Step 2: Setup collateral
  console.log('\nStep 2: Setting up collateral');
  const sbtcAmount = 2000000000n; // 2 BTC
  txOk(contracts.sbtc.mint(sbtcAmount, alice), deployer);

  try {
    const addCollateralResult = txOk(
      contracts.market.collateralAdd(contracts.sbtc.identifier, sbtcAmount, alice),
      alice
    );
    console.log('collateral-add succeeded');
  } catch (e) {
    console.log('ERROR in collateral-add:', e);
  }
  const usdcAmount = 18000000000n; // $18,000
  txOk(contracts.usdc.mint(usdcAmount, alice), deployer);

  try {
    const addCollateralResult = txOk(
      contracts.market.collateralAdd(contracts.usdc.identifier, usdcAmount, alice),
      alice
    );
    console.log('collateral-add succeeded');
  }

```

```

} catch (e) {
  console.log('ERROR in collateral-add:', e);
}

// Step 3: Setup vault liquidity
console.log('\nStep 3: Setting up vault liquidity');
txOk(contracts.usdc.mint(100000000000n, bob), deployer);
txOk(contracts.vaultUsdc.deposit(100000000000n, 0n, bob), bob);
txOk(contracts.sbtc.mint(2000000000n, bob), deployer);
txOk(contracts.vaultSbtc.deposit(2000000000n, 0n, bob), bob);

let bitmap = rov(contracts.assets.getBitmap());
let position = rov(contracts.marketVault.position(alice, bitmap));
let egroup = rov(contracts.egroup.resolve(position.mask));
console.log(position);
console.log(egroup);

// Step 4: Try to borrow sbtc
console.log('\nStep 4: Attempting to borrow SBTC');
const sbtcBorrowAmount = 160000000n; // 1.6 BTC
try {
  const borrowResult = txOk(
    contracts.market.borrow(contracts.sbtc.identifier, sbtcBorrowAmount, alice),
    alice
  );
  console.log('borrow succeeded with 1.6 BTC (80% LTV)!');
} catch (e) {
  console.log('ERROR in borrow:', e);
}
bitmap = rov(contracts.assets.getBitmap());
position = rov(contracts.marketVault.position(alice, bitmap));
egroup = rov(contracts.egroup.resolve(position.mask));
console.log(position);
console.log(egroup);

// Step 5: Try to remove collateral
console.log('\nStep 5: Attempting to remove collateral');
try {
  const repayResult = txOk(
    contracts.market.collateralRemove(contracts.usdc.identifier, usdcAmount, alice),
    alice
  );
  console.log('collateral successfully removed!');
} catch (e) {
  console.log('ERROR in collateral:', e);
}

bitmap = rov(contracts.assets.getBitmap());
position = rov(contracts.marketVault.position(alice, bitmap));
egroup = rov(contracts.egroup.resolve(position.mask));
console.log(position);
console.log(egroup);

// Step 6: Try to liquidate (should not be possible after a repay)
console.log('\nStep 6: Attempting to liquidate');
const liquidateResult = txOk(
  contracts.market.liquidate(alice, contracts.sbtc.identifier, contracts.sbtc.identifier, sbtcBorrowAmount),
  alice
);

bitmap = rov(contracts.assets.getBitmap());
position = rov(contracts.marketVault.position(alice, bitmap));
egroup = rov(contracts.egroup.resolve(position.mask));
console.log(position);
console.log(egroup);
});

it('should not repay too much', () => {

  // Step 1: Setup collateral
  console.log('\nStep 1: Setting up collateral');
  const sbtcAmount = 2000000000n; // 2 BTC
  txOk(contracts.sbtc.mint(sbtcAmount, alice), deployer);
  txOk(contracts.sbtc.mint(sbtcAmount, bob), deployer);

  try {
    txOk(
      contracts.market.collateralAdd(contracts.sbtc.identifier, sbtcAmount, alice),
      alice
    );
    txOk(
      contracts.market.collateralAdd(contracts.sbtc.identifier, sbtcAmount, bob),
      bob
    );
    console.log('collateral-add succeeded');
  } catch (e) {
    console.log('ERROR in collateral-add:', e);
  }

```

```

}

// Step 2: Setup vault liquidity
console.log('\nStep 2: Setting up vault liquidity');
txOk(contracts.usdc.mint(100000000000n, bob), deployer);
txOk(contracts.vaultUsdc.deposit(100000000000n, 0n, bob), bob);

// Step 3: Try to borrow usdc
console.log('\nStep 3: Attempting to borrow USDC');
const borrowAmount = 18000000000n; // $18,000
try {
  const borrowResult = txOk(
    contracts.market.borrow(contracts.usdc.identifier, borrowAmount, alice),
    alice
  );
  console.log('borrow succeeded with $18,000!');
} catch (e) {
  console.log('ERROR in borrow:', e);
}
try {
  const borrowResult = txOk(
    contracts.market.borrow(contracts.usdc.identifier, borrowAmount, bob),
    bob
  );
  console.log('borrow succeeded with $18,000!');
} catch (e) {
  console.log('ERROR in borrow:', e);
}

// Step 4: Mint usdc to alice
console.log('\nStep 4: Minting usdc to alice');
txOk(contracts.usdc.mint(360000000000n, alice), deployer);

// Step 5: Try to repay
console.log('\nStep 5: Attempting to repay');
let balanceBefore = rov(contracts.usdc.getBalance(alice)).value;
try {
  const repayResult = txOk(
    contracts.market.repay(contracts.usdc.identifier, 2n * borrowAmount, alice),
    alice
  );
  console.log('repay succeeded with $18,000!');
} catch (e) {
  console.log('ERROR in repay:', e);
}
let balanceAfter = rov(contracts.usdc.getBalance(alice)).value;
balanceAfter = balanceAfter == null ? 0n : balanceAfter;

console.log(balanceBefore);
console.log(balanceAfter);

expect(balanceBefore).toEqual(balanceAfter + 2n * borrowAmount);

let bitmap = rov(contracts.assets.getBitmap());
let position = rov(contracts.marketVault.position(alice, bitmap));
console.log(position);
});

it('repayment amount is wrongly calculated in liquidations', async () => {

  // Step 1: Setup collateral
  console.log('\nStep 1: Setting up collateral');
  const sbtcAmount = 2000000000n; // 2 BTC
  txOk(contracts.sbtc.mint(sbtcAmount, alice), deployer);
  txOk(contracts.sbtc.mint(sbtcAmount, bob), deployer);

  try {
    txOk(
      contracts.market.collateralAdd(contracts.sbtc.identifier, sbtcAmount, alice),
      alice
    );
    txOk(
      contracts.market.collateralAdd(contracts.sbtc.identifier, sbtcAmount, bob),
      bob
    );
    console.log('collateral-add succeeded');
  } catch (e) {
    console.log('ERROR in collateral-add:', e);
  }

  // Step 2: Setup vault liquidity
  console.log('\nStep 2: Setting up vault liquidity');
  txOk(contracts.usdc.mint(1_000_000_000000n, bob), deployer);
  txOk(contracts.vaultUsdc.deposit(1_000_000_000000n, 0n, bob), bob);
  txOk(contracts.usdh.mint(1_000_000_000000n, bob), deployer);
  txOk(contracts.vaultUsdh.deposit(1_000_000_000000n, 0n, bob), bob);
  txOk(contracts.usdc.mint(20_000_000000n, charlie), deployer);

```

```

// Step 3: Try to borrow usdc
console.log('\nStep 3: Alice attempting to borrow USDC');
const usdcBorrowAmount = 100_000000n; // $100
try {
  txOk(
    contracts.market.borrow(contracts.usdc.identifier, usdcBorrowAmount, alice),
    alice
  );
  console.log('borrow succeeded with $100!');
} catch (e) {
  console.log('ERROR in borrow:', e);
}
let bitmap = rov(contracts.assets.getBitmap());
let position = rov(contracts.marketVault.position(alice, bitmap));
console.log(position);

// Step 4: Try to borrow USDh
console.log('\nStep 4: Alice attempting to borrow USDh');
const usdhBorrowAmount = 39_900_000000n;
try {
  const borrowResult = txOk(
    contracts.market.borrow(contracts.usdh.identifier, usdhBorrowAmount, alice),
    alice
  );
  console.log('borrow succeeded with $39,900!');
} catch (e) {
  console.log('ERROR in borrow:', e);
}

const totalUsdBorrowAmount = usdhBorrowAmount + usdcBorrowAmount;

bitmap = rov(contracts.assets.getBitmap());
position = rov(contracts.marketVault.position(alice, bitmap));
console.log(position);

// Step 5: Modify BTC price to trigger liquidation
console.log('\nStep 5: Modifying BTC price to trigger liquidation');
const newSbtcPrice = scalePriceForPyth(30_000, 8);

await set_price(PythFeedIds.BTC, newSbtcPrice, -8, deployer);

let balanceBeforeSbtc = rov(contracts.sbtc.getBalance(charlie)).value;
let balanceBeforeUsdc = rov(contracts.usdc.getBalance(charlie)).value;
let balanceBeforeUsdh = rov(contracts.usdh.getBalance(charlie)).value;
balanceBeforeUsdc = balanceBeforeUsdc == null ? 0n : balanceBeforeUsdc;
expect(balanceBeforeSbtc).to.equal(0n);
expect(balanceBeforeUsdc).to.equal(20_000_000000n);
expect(balanceBeforeUsdh).to.equal(0n);
console.log('balanceBeforeSbtc: ' + balanceBeforeSbtc);
console.log('balanceBeforeUsdc: ' + balanceBeforeUsdc);

// Step 6: Try to liquidate (we can check at the end how Charlie gets way more sBTC than they should for 100 USDC)
console.log("\nStep 6: Charlie attempting to liquidate Alice's USDC position (not USDH)");
const liquidateResult = txOk(
  contracts.market.liquidate(alice, contracts.sbtc.identifier, contracts.usdc.identifier, totalUsdBorrowAmount),
  charlie
);

bitmap = rov(contracts.assets.getBitmap());
position = rov(contracts.marketVault.position(alice, bitmap));
let egroup = rov(contracts.egroup.resolve(position.mask));
console.log("position", position);

/*
 * Calculate the value of `max-debt-usd` in `liquidate` function
 */
const currentLTV = totalUsdBorrowAmount * 10000n / (sbtcAmount * newSbtcPrice / (1_000000000n * 100n));
console.log("currentLTV", currentLTV);
const ltvLiqFull = buffToUint(egroup.LTVLIQFULL);
const ltvLiqPartial = buffToUint(egroup.LTVLIQPARTIAL);
const liqPenaltyMin = buffToUint(egroup.LIQPENALTYMIN);
const liqPenaltyMax = buffToUint(egroup.LIQPENALTYMAX);

// Mirrors calculation in `reserve-calculator.calc-liq-factor`
const liqPercent = (currentLTV - ltvLiqPartial) * 10000n / (ltvLiqFull - ltvLiqPartial);
console.log("liqPercent", liqPercent);
// Mirrors calculation in `reserve-calculator.calc-liq-factor-bound`
const liqPenalty = (liqPenaltyMin + (liqPercent * (liqPenaltyMax - liqPenaltyMin) / 10000n));
console.log("liqPenalty", liqPenalty);

const sbtcForTotalUsdBorrowAmount = totalUsdBorrowAmount * 1_000000000n * 100n / newSbtcPrice;
const denom = 10000n * 10000n;
const sbtcReceived = (liqPercent * (10000n + liqPenalty) * sbtcForTotalUsdBorrowAmount + denom - 1n) / denom;
console.log("sbtcReceived", sbtcReceived);

```

```

let balanceAfterSbtc = rov(contracts.sbtc.getBalance(charlie)).value;
let balanceAfterUsdc = rov(contracts.usdc.getBalance(charlie)).value;
let balanceAfterUsdh = rov(contracts.usdh.getBalance(charlie)).value;
balanceAfterUsdc = balanceAfterUsdc == null ? 0n : balanceAfterUsdc;
expect(balanceBeforeUsdc - balanceAfterUsdc).to.equal(100_000000n);
expect(balanceAfterSbtc).to.equal(sbtcReceived);
expect(balanceAfterUsdh).to.equal(0n);
console.log('balanceAfterSbtc: ' + balanceAfterSbtc);
console.log('balanceAfterUsdc: ' + balanceAfterUsdc);
});

it('DoS on accrue', () => {
  const ASSETS_AMOUNT: bigint = 1_0000_0000n;
  const SHARES_AMOUNT: bigint = 2_0000_0000n;

  // Step 1: Execute proposals
  console.log('\nStep 1.1: Set points');
  executeDaoProposalTest(proposalTest.constants.TEST_SET_POINTS);
  console.log('\nStep 1.2: Set fee reserve');
  executeDaoProposalTest(proposalTest.constants.TEST_SET_FEE_RESERVE);

  // Step 2: Setup collateral
  console.log('\nStep 2: Setting up collateral');
  const sbtcAmount = 2000000000n; // 2 BTC
  txOk(contracts.sbtc.mint(sbtcAmount, alice), deployer);

  try {
    const addCollateralResult = txOk(
      contracts.market.collateralAdd(contracts.sbtc.identifier, sbtcAmount, alice),
      alice
    );
    console.log('collateral-add succeeded');
  } catch (e) {
    console.log('ERROR in collateral-add:', e);
  }

  // Step 3: Setup vault liquidity
  console.log('\nStep 3: Setting up vault liquidity');
  txOk(contracts.usdc.mint(36_000_000000n, bob), deployer);
  txOk(contracts.vaultUsdc.deposit(36_000_000000n, 0n, bob), bob);

  // Step 4: Try to borrow usdc
  console.log('\nStep 4: Attempting to borrow USDC');
  const borrowAmount = 250_000000n;
  try {
    const borrowResult = txOk(
      contracts.market.borrow(contracts.usdc.identifier, borrowAmount, alice),
      alice
    );
    console.log(borrowResult);
    console.log(`borrow succeeded with ${borrowAmount}`);
  } catch (e) {
    console.log('ERROR in borrow:', e);
  }

  let bitmap = rov(contracts.assets.getBitmap());
  let position = rov(contracts.marketVault.position(alice, bitmap));
  console.log("position", position);

  // Step 5: Get relevant variables
  console.log('\nStep 5: Get relevant variables');

  let interestRate = rov(contracts.vaultUsdc.getInterestRate());
  console.log('interestRate: ' + interestRate.value);
  let utilization = rov(contracts.vaultUsdc.getUtilization());
  console.log('utilization: ' + utilization.value);
  let index = rov(contracts.vaultUsdc.getIndex());
  console.log('index: ' + index.value);
  let nextIndex = rov(contracts.vaultUsdc.getNextIndex());
  console.log('nextIndex: ' + nextIndex.value);
  let feeReserve = rov(contracts.vaultUsdc.getFeeReserve());
  console.log('feeReserve: ' + feeReserve.value);

  // Step 6: Accrue and check DoS
  console.log('\nStep 6: Accrue and check DoS');
  try {
    txOk(
      contracts.vaultUsdc.accrue(),
      alice
    );
    console.log(`accrual succeeded`);
  } catch (e) {
    console.log('ERROR in accrue:', e);
  }
});

```

```

it('check liquidation factor greater than 100% is possible', () => {
  let liqFactor = rov(contracts.reserveCalculator.calcLiqFactor(9000n, 6500n, 8000n));
  expect(liqFactor).toBeGreaterThan(10000);
});

it('check faulty pack implementation', () => {
  // To run this test change temporarily the visibility of valut-usdc.pack-u16 from `private` to `public`

  let pack = txOk(contracts.vaultUsdc.packU16([10n, 0n, 0n, 0n, 65535n, 0n, 0n, 0n], 10000n), alice);
  // Executing with one of the values of the list greater than the upper bound of 10000n. Should revert.
  // Returns result: { type: 'ok', value: { type: 'uint', value: 10n } }
  // That's because 65535n is more than the limit, but instead of reverting, the value is just skipped (as if it was zero)
  console.log(pack);

  pack = txOk(contracts.vaultUsdc.packU16([65536n, 10n, 0n, 0n, 0n, 0n, 0n, 0n], 70000n), alice);
  // Executing with an upper bound of 70000n, so greater than the max value that fits in a 16bit slot (65536n). Should clamp to 65536n.
  // Returns result: { type: 'ok', value: { type: 'uint', value: 720896n } }
  // That's because 65536n is 2*16, so it adds up with the next slot, that has value 10. The resulting word is 11*2^16 = 720896
  console.log(pack);

  try {
    pack = txOk(contracts.vaultUsdc.packU16([10n, 0n, 0n, 0n, 0n, 0n, 0n, 65535n], 10000n), alice);
    console.log('pack succeeded');
  } catch (e) {
    // Executing with one of the values of the list greater than the upper bound of 10000n. Should revert and it does.
    // Reverts because it's the last element list the one that goes over the limit, if 65535n was in a different position it would not revert.
    console.log('ERROR in pack:', e);
  }
});
});

describe('DAO PoCs', () => {
  beforeEach(async () => {
  });

  it('duplicate signers when initializing DAO', () => {
    const daoMultisig = contracts.daoMultisig;
    const daoExecutor = contracts.daoExecutor;

    // Step 1: Initialize multisig
    console.log('\nStep 1: Initialize multisig with deployer and alice as signers and bob duplicated, threshold 1');
    txOk(daoMultisig.init([deployer, alice, bob, bob], 1n), deployer);

    // Step 2: Initialize executor
    console.log('\nStep 2: Initialize executor with multisig as implementation');
    txOk(daoExecutor.init(daoMultisig.identifier), deployer);

    // Step 3: Verify setup
    console.log('\nStep 3: Verify setup');
    const threshold = rov(daoMultisig.getThreshold());
    const signerCount = rov(daoMultisig.getSignerCount());
    const isSignerDeployer = rov(daoMultisig.isSigner(deployer));
    const isSignerAlice = rov(daoMultisig.isSigner(alice));
    const isSignerBob = rov(daoMultisig.isSigner(bob));

    expect(threshold).toEqual(1n);
    expect(signerCount).toEqual(4n);
    expect(isSignerDeployer).toEqual(true);
    expect(isSignerAlice).toEqual(true);
    expect(isSignerBob).toEqual(true);

    // Step 4: Set threshold
    console.log('\nStep 4: Execute threshold change to 4');
    executeDaoProposalTest(proposalTest.constants.TEST_SET_THRESHOLD);

    // Step 5: Confirm threshold updated
    console.log('\nStep 5: Confirm threshold updated');
    const thresholdAfter = rov(daoMultisig.getThreshold());
    const signerCountAfter = rov(daoMultisig.getSignerCount());
    expect(thresholdAfter).toEqual(4n);
    expect(signerCountAfter).toEqual(4n);
  });
});

```

tests/greybeard/socialize-debt-block-repay.test.ts

```

import {
  deployer,

```

```

    alice,
    bob,
    charlie,
    executeDaoProposal,
    market,
    marketVault,
    assets,
    initializeProtocol,
    vaultSbtc,
    egroup,
    contracts
} from '../unit/initialization/helpers';

import {
    init_pyth,
    set_initial_price,
    set_price,
    scalePriceForPyth,
    PythFeedIds
} from '../unit/helpers/pyth-helpers';

import { sbtc, usdc, buffToUint, daniel } from "../helpers"; // Extra Greybeard helpers

import { beforeEach, describe, it, expect } from 'vitest';
import { rov, rovOk, txOk, tx, txErr } from '@clarigen/test';

describe('Greybeard Security', () => {

    const SBTC_PRICE = 60_000n;
    const USDC_COLLAT = 100_000_000000n;

    beforeEach(async () => {
        initializeProtocol();
        init_pyth(deployer);
        executeDaoProposal(contracts.proposalSetPriceStaleness);

        // 3. Set initial prices via Pyth
        // BTC at $60,000
        await set_price(
            PythFeedIds.BTC,
            scalePriceForPyth(60_000, 8),
            -8,
            deployer
        );

        // USDC at $1
        await set_initial_price(
            PythFeedIds.USDC,
            scalePriceForPyth(1, -8),
            deployer
        );
    });

    it('repayment is bricked after bad debt socialization and debt < total-borrowed', async () => {

        const ltvResult = rov(egroup.lookup({ id: egroup.constants.DEFAULTMASKID }), alice);
        const ltvBorrow = buffToUint(ltvResult.LTVBORROW);
        console.log("LtvBorrow", ltvBorrow);

        // Mint sbtc to Alice
        txOk(sbtc.mint({
            amount: 10_0000_0000n,
            recipient: alice
        }), deployer);

        // Mint usdc to Bob
        txOk(usdc.mint({
            amount: 1_000_000_000000n,
            account: bob
        }), deployer);

        // Mint sbtc to Charlie
        txOk(sbtc.mint({
            amount: 1_0000_0000n,
            recipient: charlie
        }), deployer);

        // Mint usdc to Daniel
        txOk(usdc.mint({
            amount: 1_000_000_000000n,
            account: daniel
        }), deployer);

        // Alice, the LP, deposits
        txOk(vaultSbtc.deposit({
            amt: 10_0000_0000n,

```

```

        minOut: 0n,
        recipient: alice
    )), alice);

    txOk(market.collateralAdd({
        ft: usdc.identifier,
        amount: USDC_COLLAT,
        account: bob
    )), bob);

    // Borrow the maximum
    const borrowAmount = (USDC_COLLAT * ltvBorrow / 10_000n) * 10n** (8n - 6n) / SBTC_PRICE;
    console.log("borrowAmount", borrowAmount);

    txOk(market.borrow({
        ft: sbtc.identifier,
        amount: borrowAmount,
        account: bob
    )), bob);

    const DANIEL_SBTC_BORROW_AMOUNT = 1000_0000n;

    // Daniel borrows some sBTC too
    txOk(market.collateralAdd({
        ft: usdc.identifier,
        amount: USDC_COLLAT,
        account: daniel
    )), daniel);

    txOk(market.borrow({
        ft: sbtc.identifier,
        amount: DANIEL_SBTC_BORROW_AMOUNT,
        account: daniel
    )), daniel);

    const debt0 = rovOk(vaultSbtc.getDebt());

    // Now change the price of sBTC
    // Make the price of sBTC ridiculously high so there is bad debt
    await set_price(PythFeedIds.BTC, scalePriceForPyth(200_000,8), -8, deployer);

    const enabledMask = rov(contracts.assets.getBitmap());
    console.log(rov(marketVault.position(bob, enabledMask)));

    // Charlie liquidates
    txOk(market.liquidate({
        borrower: bob,
        collateralFt: usdc.identifier,
        debtFt: sbtc.identifier,
        debtAmount: borrowAmount
    )), charlie);

    console.log("Bob's position", rov(marketVault.position(bob, enabledMask)));
    console.log("Daniels's position", rov(marketVault.position(daniel, enabledMask)));

    const debt1 = rovOk(vaultSbtc.getDebt());

    // Debt has decreased by Bob's borro amount
    expect(debt0 - debt1).to.be.equal(borrowAmount);

    // Can't read the total-borrow data-var since no public getter function exists.

    // Daniel can't even pay a single sBTC sat back
    expect(() => tx(market.repay({
        ft: sbtc.identifier,
        account: daniel,
        amount: 1n
    )), daniel)).toThrow(new RegExp("ArithmeticUnderflow.*vault-sbtc:system-repay"));
    });
    });

```