ClarityAlliance

ZEST PROTOCOL v2 SECURITY REVIEW

Conducted by:
KRISTIAN APOSTOLOV, ALIN BARBATEI (ABA), SILVEROLOGIST

OCTOBER 23RD, 2025

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

1. About Clarity Alliance

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

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

Learn more about Clarity Alliance at claritalliance.org.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

2. Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Clarity Alliance to perform a security assessment.

This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Clarity Alliance’s position is that each company and individual are responsible for their own due diligence and continuous security. Clarity Alliance’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree
to analyze.

The assessment services provided by Clarity Alliance are subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis.

Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third parties. Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Clarity Alliance does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

3. Introduction

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

4. About Zest Protocol

Zest Protocol is the DeFi protocol built for Bitcoin. Fully on-chain and open-source, it is building the future of Bitcoin finance.

We've launched Zest Protocol Borrow, enabling users to unlock liquidity by borrowing against their assets.

Live on Stacks—the leading Bitcoin Layer 2—Zest is now the top DeFi protocol on the network. Through the Stacks Market, users can deposit idle assets such as STX, sBTC, stSTX, USDC, and others to earn yield, accumulate points, and access overcollateralized loans..

Zest exists to make Bitcoin productive—every sat of it. The goal is to build a vibrant borrowing and lending ecosystem around Bitcoin as an asset.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

5. Risk Classification

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

5.1 Impact

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

5.2 Likelihood

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

5.3 Action required for severity levels

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

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

6. Security Assessment Summary

Scope

The following contracts, located in the zest-core repository, were in the scope of the security review:

- `dao/dao-multisig.clar`
- `dao/dao-executor.clar`
- `dao/dao-treasury.clar`
- `dao/traits.clar`
- `market/market.clar`
- `market/market-vault.clar`
- `registry/egroup.clar`
- `registry/assets.clar`
- `registry/reserve-calculator.clar`
- `vault/vault-stx.clar`
- `vault/vault-sbtc.clar`
- `vault/vault-ststx.clar`
- `vault/vault-usdc.clar`
- `vault/vault-usdh.clar`
- `vault/traits.clar`

Initial Commit Reviewed:
3c1f2ebe178081d118cd9005eddb02b97f6aaf95

Intermediate Commits Reviewed:
3924c43522c7523771fa340d5782966a9d59c52e

b1839d94a7a7f7f66f540bf3c198314b31802c99

bfbb7862c8def6a073af163da3b3425be0279381

Final Commit After Remediations:
f4987a8b177e3075ac658fabb87b1b249944a74d

Given the number and severity of findings identified, Clarity Alliance strongly recommend that the current snapshot undergo a follow-up audit and further security enhancements to ensure that any potential remaining underlying issues are addressed.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

7. Executive Summary

Over the course of the security review, Kristian Apostolov, Alin Barbatei (ABA), Silverologist engaged with - to review Zest Protocol. In this period of time a total of **47** issues were uncovered.

Protocol Summary

Protocol Name	Zest Protocol
Date	October 23rd, 2025

Findings Count

Severity	Amount
Critical	2
High	11
Medium	17
Low	7
QA	10
Total Findings	**47**

CONTENTS

Summary of Findings

ID	Title	Severity	Status
[C-01]	FlashLoan Vulnerability: Potential for Artificial Vault Share Inflation and Vault Draining	Critical	Resolved
[C-02]	Repaid Interest Is Not Accounted For	Critical	Resolved
[H-01]	Incorrect Rounding Direction When Calculating Vault Shares or Assets	High	Resolved
[H-02]	FlashLoan Balance Check Issue	High	Resolved
[H-03]	FlashLoan Fees Are Lost	High	Resolved
[H-04]	Share Price Manipulation Allows Token Theft	High	Resolved
[H-05]	Missing Interest Accrual When Removing Collateral	High	Resolved
[H-06]	Vault Accrual Through The Market Contract Issue	High	Resolved
[H-07]	Borrowers Can Prevent Bad Debt Socialization	High	Resolved
[H-08]	Liquidated Debt Incorrectly Socialized as Bad Debt	High	Resolved
[H-09]	Disabled Collaterals Can Be Used To Extract Liquidity	High	Resolved
[H-10]	Limiting and Bypassable FlashLoan Fee Payment Check	High	Resolved
[H-11]	Rounding Inconsistencies	High	Resolved
[M-01]	Vaults Are Not SIP-10 Compliant	Medium	Resolved
[M-02]	Inaccurate Conversion Results Due to Missing Interest Accrual	Medium	Resolved
[M-03]	Vaults Are Tied to the Router Contract	Medium	Resolved
[M-04]	Disabled Collateral Cannot Be Withdrawn	Medium	Resolved
[M-05]	Protocol Pausing Issues	Medium	Resolved
[M-06]	Governance Proposals Never Expire	Medium	Resolved
[M-07]	DAO Can Set Invalid Multisig Configuration and Disrupt Governance	Medium	Resolved

CONTENTS

Summary of Findings

ID	Title	Severity	Status
[M-08]	Vault Availability Implementation Issue	Medium	Resolved
[M-09]	Liquidation Fails When Borrowers Have No Debt and No Collateral	Medium	Resolved
[M-10]	Fee Reserve Percentage Change Is Applied Retroactively on Pending Debt	Medium	Resolved
[M-11]	Utilization Configuration Changes Are Applied Retroactively on Pending Debt	Medium	Resolved
[M-12]	Accrue Must Be Called Before Pausing to Prevent Loss of Pending Debt	Medium	Resolved
[M-13]	Rounding in Bad Debt Socialization Causes Debt Mismatches	Medium	Resolved
[M-14]	Precision Loss Leads to Underestimated Interest Rates	Medium	Resolved
[M-15]	Pyth Price Confidence Interval Is Not Validated	Medium	Resolved
[M-16]	Oracle Freshness Check Issues	Medium	Resolved
[M-17]	Liquidation Penalty Can Exceed Maximum Allowed	Medium	Resolved
[L-01]	Non-Standard Vault Withdraw Entry Point Behavior	Low	Resolved
[L-02]	Incorrect Return Value on Vault Operations	Low	Resolved
[L-03]	Severe Lack of Emitted Events	Low	Acknowledged
[L-04]	assets::get-nr-enabled Function is Unusable	Low	Resolved
[L-05]	Limiting and Bypassable FlashLoan Fee Payment Check	Low	Acknowledged
[L-06]	Market ZToken Data Insertion Overhead	Low	Resolved
[L-07]	Limited Precision in Liquidation Exponent Calculation	Low	Acknowledged

ClarityAlliance
Security Review

Zest Protocol
v2

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

Summary of Findings

ID	Title	Severity	Status
[QA-01]	Limiting Vault Borrow-Repay Recipient Interface	QA	Acknowledged
[QA-02]	Limiting FlashLoan Design	QA	Resolved
[QA-03]	Eliminate Redundant begin Blocks in let Blocks	QA	Resolved
[QA-04]	Improve Switch-Like Statements	QA	Acknowledged
[QA-05]	Severe Violation of Single Responsibility Principle	QA	Resolved
[QA-06]	Inline All Pack and Mathematical Operations	QA	Resolved
[QA-07]	Debt Cap Versus Borrow Cap Considerations	QA	Acknowledged
[QA-08]	Typographical Error	QA	Resolved
[QA-09]	Optimization of Accrue for Liquidity and Index Reads	QA	Resolved
[QA-10]	Cleanup Codebase Functions	QA	Resolved

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

8. Findings

8.1. Critical Findings

[C-01] FlashLoan Vulnerability: Potential for Artificial Vault Share Inflation and Vault Draining

Description

The vault's `flashloan` functionality operates by sending funds to a callback contract and then checking the ending contract balance, which must include the original balance plus a fee.

This design flaw allows an attacker to execute a flashloan of the entire vault balance, redeposit it, pay a minimal fee on the loan, and consequently acquire approximately 50% of the vault shares.

With these 50% shares, the attacker can withdraw half of the vault's underlying liquidity. Therefore, if the flashloan fee applied to the entire vault balance is less than 50% of the vault's liquidity, the attack becomes profitable.

Note: This issue was identified by the developer during the testing phase.

Recommendation

Implement a mutex-type lock to prevent any deposits or withdrawals into the vault during the execution of a flashloan.

CONTENTS

[C-02] Repaid Interest Is Not Accounted For

Description

When a loan is repaid, the vaults are accessed through the market contract and router using the `vault::system-repay` function. This function calculates the actual reduction in the borrowed principal amount, which is less than the total repayment because it includes interest.

The actual principal deduction is represented by `pratio`:

```
(pratio (principal-ratio-reduction- actual-amt p d))
```

However, the paid interest is neither separated nor added to the `assets`. As a result, the paid interest is effectively lost within the vault.

Recommendation

Calculate the equivalent repaid debt amount for the user and identify the difference between the reduction amount and the debt amount as interest.

8.2. High Findings

[H-01] Incorrect Rounding Direction When Calculating Vault Shares or Assets

Description

When depositing into the new Zest vaults, the conversion of assets to shares during deposits and vice versa utilizes a `mul-div` function:

```
(define-private (convert-to-shares- (amt uint))
;; ... code ...
        (mul-div amt ts ta))))

(define-private (convert-to-assets- (amt uint))
;; ... code ...
        (mul-div amt ta ts))))
```

The `mul-div` function is a wrapper for a similarly named function from `math` the contract, implemented as follows:

```
(define-read-only (mul (x uint) (y uint)) (/ (+ (* x y)
  (/ PRECISION u2)) PRECISION))
(define-read-only (div (x uint) (y uint)) (/ (+ (* x PRECISION) (/ y u2)) y))
(define-read-only (mul-div (x uint) (y uint) (z uint)) (div (mul x y) z))
```

The `math::mul-div` function is composed of the `mul` function followed by the `div` function. Both `mul` and `div` employ a rounding method that rounds to the nearest whole unit, rather than consistently rounding up or down.

For any vault-related operations, rounding should always be in favor of the protocol to prevent value leakage. For instance, if the current rounding logic rounds up during `withdraw` operations, it could result in extracting more assets than intended.

This issue could even lead to the last withdrawer being unable to receive funds, as the rounding might attempt to withdraw more funds than are available in the contract at that time.

Recommendation

Implement a `mul-div-down` equivalent in the `vault-*` contracts, such as `(/(* x y) z)`, and inline it within each vault to minimize significant contract overhead calls.

CONTENTS

[H-02] FlashLoan Balance Check Issue

Description

The vault contracts permit the execution of flashloans using the underlying amounts. Users receive a specified amount and are required to repay it to the vault contract, along with a fee.

```
(define-public (flashloan (amt uint) (fc ‹flash-callback›) (memo (optional
  (buff 34))))
  (let (
      ;; ...
        (fee      (/ (* amt FEE-FLASH) BPS))
        (ubal     (ubalance))
        (delta    (+ ubal fee))
        (next     (+ ubal delta)))
      ;; ...
      (asserts! (is-eq (ubalance) next) ERR-LENDING-POSTCONDITIONS)
```

However, the final balance check, intended to verify that the new balance has increased by the fee, is incorrect. The `next` variable is calculated as twice the vault balance plus the fee. Consequently, the current condition effectively checks that after a flashloan, the balance has doubled plus fees.

This renders the functionality unusable, as users would incur a loss if they attempted to use it.

Recommendation

Modify the calculation of `next` to be `(+ ubal fee)` .

CONTENTS

[H-03] FlashLoan Fees Are Lost

Description

Once a flash loan is taken and the fees are repaid, they remain within the vault.

However, these fees are not included in the vault's `assets` total, which means they are effectively trapped in the contract and do not contribute to liquidity provider rewards.

Recommendation

Incorporate the fees into the contract's `assets` variable.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[H-04] Share Price Manipulation Allows Token Theft

Description

The calculation of LP shares for deposits into an empty vault allows the first depositor to potentially manipulate the LP token's share price. This manipulation enables attackers to extract value from subsequent depositors through a known variation of the first depositor attack.

When adding liquidity to a vault, the following formula is used to determine the number of shares for the LP:

- If the vault is empty: `LP = amount-of-assets-deposited`
- Otherwise: `LP = amount-of-assets-deposited * total-supply / total-assets`

By repeatedly making cleverly chosen small deposits and withdrawals, an attacker can inflate the shares-to-asset ratio, making 1 unit of share significantly more valuable. This allows for the complete depletion of a future depositor's assets. This strategy also requires some Stacks blocks to pass, allowing interest to accrue on the small deposits, which facilitates the attack.

Recommendation

To completely avoid this issue, a minimum LP must be locked on the first deposit into empty pools, ensuring no problematic rounding issues occur.

This can be implemented at contract deployment by calling `deposit` with a minimum amount and having the LP burned. The burning should be directed to the `NULL` address, not to the pool, to prevent retrieval in case of future custom pools.

Example implementation (to be set at the end of the `vault` contracts):

```
(define-constant NULL-ADDRESS (unwrap-panic (principal-construct?
  (if is-in-mainnet 0x16 0x1a) 0x0000000000000000000000000000000000000000)))
(deposit u1000 u0 NULL-ADDRESS)
```

Apply the same fix to all vaults.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[H-05] Missing Interest Accrual When Removing Collateral

Description

When a user withdraws collateral through `market::collateral-remove`, the resulting position is correctly validated to ensure a healthy Loan-to-Value (LTV) ratio.

This health check depends on the total debt value, which is calculated using the vault index at the time of withdrawal. If the vault index is outdated, the calculated debt value will be lower than the actual debt owed.

However, the withdrawal process does not trigger interest accrual to update the vault index before calculating the total debt value. Consequently, the total debt value used in the LTV check only includes interest accrued up to the last accrual event, not up to the current block.

Recommendation

Before invoking `notional` within `market::collateral-remove`, ensure interest is accrued on the relevant vaults to keep the vault index current.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[H-06] Vault Accrual Through The Market Contract Issue

Description

Before any user operation involving debt (such as removing collateral, borrowing, or liquidating), the `market` contract calls `vault-router:accrue`; targeting a specific vault involved in the operation.

However, this approach is insufficient because a user may have multiple borrowed assets, all of which need to be accrued before retrieving the debt index. This index is necessary to calculate the notional debt value. As a result, there is a risk of missing debt on user-borrowed assets.

Recommendation

For each borrowed asset, accrue interest before performing any market operations. Due to limitations in Stacks and Clarity, implementing this may be challenging. Therefore, the following optimizations are recommended:

- Modify the `vault::accrue` function to return the updated index and liquidity index.
- In the `vault-router` contract, create a map of `asset-id` → `{last-updated-block:uint, index:uint, liquidity-index:uint}` to be populated by an accrue function.
- Develop a generic accrue function that iterates through all existing vaults (or a specified list of vault IDs) and calls `accrue` only if the cached amounts for the vault are not up to date.
- Implement a specific getter for retrieving these amounts.

By adopting this approach, only the first call in a block will need to perform the calculation, and the result will be stored in the router. Although there is a cache in the vaults themselves, not calling them ensures a -3 deduction on read count.

To further enhance the system, the cache can be implemented in the `market` contract itself, thereby improving read efficiency and reducing execution costs.

CONTENTS

Recommendation

Adjust `reserve-calculator::calc-utilization` to use the correct formula: `debt * BPS / (debt + available liquidity)`.

Example implementation:

```
(define-read-only (calc-utilization (available-liquidity uint) (debt uint))
   (if (is-eq assets u0)
       u0
       (contract-call? math mul-div debt BPS (+ debt available-liquidity))))
```

Additionally, modify the `vault:: utilization` function to call the updated function appropriately. The available liquidity can be obtained using the `vault:: available` function.

Example implementation:

```
(define-private (utilization)
  (let  ((a (available))
         (d (debt) ))
    (contract-call?  .reserve-calculator
                     calc-utilization
                     a
                     d)))
```

Note: The current implementation of `available` is incorrect, but this is addressed in another issue. Both issues need to be resolved to fully address this finding if the `available` function is reused.

CONTENTS

[H-07] Borrowers Can Prevent Bad Debt Socialization

Description

After a liquidation is executed, the protocol only socializes bad debt if the borrower's remaining collateral across all asset types is exactly zero and if they have exactly one type of collateral asset.

```
;; check if this liquidation removed ALL collateral
(let ((no-collateral-left
        (and
         ;; Only had one collateral type
         (is-eq (len (get collateral pos)) u1)
         ;; And we're taking all of it
         (is-eq user-coll-balance coll-actual))))
```

Additionally, liquidations are processed one collateral type at a time, which means that to reach this point, several other user liquidations are needed to remove the user's collateral to this extent.

This design creates an opportunity for borrowers to front-run liquidation transactions. A borrower can deposit a minimal amount of another collateral type (different from the one being liquidated) just before a liquidation that would otherwise trigger bad debt socialization.

By doing so, the borrower ensures that they never have zero collateral overall, thereby preventing bad debt socialization indefinitely.

Recommendation

Introduce a `liquidate-multi` function that enables liquidators to liquidate multiple positions in one call. Using this function, a liquidator can liquidate all collateral types associated with a borrower in a single transaction.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[H-08] Liquidated Debt Incorrectly Socialized as Bad Debt

Description

After a liquidation, if the borrower's remaining collateral is zero, the protocol initiates the process of socializing bad debt.

During this process, the contract iterates through each of the borrower's debt positions and calls `socialize-debt-asset`, using `get debt pos` as the debt data to be socialized.

However, the `pos` variable reference used here contains outdated data, reflecting the borrower's debt before the liquidation occurred.

This results in the liquidated debt being mistakenly treated and socialized as bad debt.

Recommendation

Ensure that debt socialization is performed on an updated debt list.

To implement this more efficiently, use the return value of `market-vault::debt-remove-scaled`, which represents the updated debt amount. Move the last two market calls from the `liquidate` function into the same `let` block with the `no-collateral-left` variable. In the `no-collateral-left` branch, directly remove the `debt-aid` entry from the `(get debt pos)` position using a filter, and only add it back with the updated amount if the updated amount is greater than 0 (indicating there is still debt).

[H-09] Disabled Collaterals Can Be Used To Extract Liquidity

Description

During liquidation, the liquidator selects a type of collateral to receive in exchange for repaying the borrower's debt. The contract retrieves the relevant collateral data from the `alist`, which contains only enabled collateral tokens:

```
(coll-asset-info (unwrap-panic (find-asset coll-aid alist)))
```

If a liquidation targets a disabled collateral token, the call to `find-asset` will return `none`, causing `unwrap-panic` to revert. This makes it impossible to request disabled collateral tokens during liquidation.

This behavior can be exploited to drain liquidity from the protocol. Consider the following attack scenario:

- Collateral token `C1` is about to be disabled.
- Bob front-runs the disabling by taking out loans across all vaults, using `C1` as collateral.
- After `C1` is disabled, Bob back-runs the transaction by depositing a minimal amount of another collateral type, `C2`, and then liquidates himself using the `C2` token.
- This liquidation triggers bad debt socialization for Bob's entire loan, as disabled collateral tokens are not accounted for.
- Once his debt is socialized, Bob can withdraw all of his `C1` collateral, effectively stealing the full liquidity previously borrowed.

Through this sequence, the attacker can drain the vaults' available liquidity while offloading the debt to the protocol via socialization.

Recommendation

Ensure that withdrawals of disabled collateral tokens are allowed only when the position remains healthy after the withdrawal. This means that in `market::remove-collateral`, two post-removal Loan-to-Value (LTV) ratios should be calculated: one for active collaterals and one for all collaterals.

If the active collateral LTV is healthy, the withdrawal can proceed successfully. Otherwise, the withdrawal can only proceed if the removed collateral is disabled and the full LTV is healthy.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[H-10] Limiting and Bypassable FlashLoan Fee Payment Check

Description

To utilize the flashloan vault functionality, a fee must be paid. The `vault::flashloan` function ensures that the fee is added to the vault's underlying balance by checking for a balance increase, referencing the balance before the callback is executed:

```
(let (
        ;;... code ...
        (ubal    (ubalance)))
      ;;... code ...
        (try! (utransfer amt callback false))
        (try! (contract-call? fc callback amt fee memo))
        (asserts! (is-eq (ubalance) (+ ubal fee)) ERR-LENDING-POSTCONDITIONS)
```

This approach is problematic because it assumes that the balance increase is solely due to fee repayment. This assumption is incorrect, as a caller, during a flashloan, can liquidate positions or repay debt on their own positions. Both actions would increase the underlying vault balance by simply repaying the loan.

This scenario allows for fee bypassing if the caller already intended to liquidate and /or repay a position. It results in a complete fee bypass if no more than the fee is repaid; otherwise, the `ERR-LENDING-POSTCONDITIONS` revert is triggered in `vault::flashloan`.

Additionally, the strict equality assertion in the ending balance check:

```
(asserts! (is-eq (ubalance) (+ ubal fee)) ERR-LENDING-POSTCONDITIONS)
```

can trigger the revert scenario during regular usage.

For example, if a liquidation bot uses a flashloan to repay debt on vaults, it will block itself due to the loan repayment, limiting its actions.

Recommendation

Modify the fee verification mechanism from a balance check to a per-transfer basis. To implement this, send the flashloaned amount to the caller (or designated principal) instead of the callback contract. At the end, transfer the original amount with the fees from the caller back to the vault.

This workaround is effective because funds can be pulled from any third-party address if the caller is that third party, meaning the funds can only be taken from either the `contract-caller` or `tx-sender`, provided the underlying token supports both authorization modes.

Not all tokens support both authorization modes. For instance, some newer tokens, such as sBTC, authorize either the caller or sender:

CONTENTS


```
(define-public (transfer (amount uint) (sender principal)
  (recipient principal) (memo (optional (buff 34))))
        (begin
                (asserts! (or (is-eq tx-sender sender)
    (is-eq contract-caller sender)) ERR_NOT_OWNER)
```

However, older tokens, such as stSTX, only authorize `tx-sender`.

```
(define-public (transfer (amount uint) (sender principal)
  (recipient principal) (memo (optional (buff 34))))
  (begin
    (asserts! (is-eq tx-sender sender) (err ERR_NOT_AUTHORIZED))
```

Notably, `stSTX` is the only token in the current vaults that does not support `contract-caller` transfer validation.

[H-11] Rounding Inconsistencies

Description

The `math` contract is extensively utilized throughout the codebase for its mathematical helper functions.

```
(define-read-only (mul (x uint) (y uint)) (/ (+ (* x y)
  (/ PRECISION u2)) PRECISION))
(define-read-only (div (x uint) (y uint)) (/ (+ (* x PRECISION) (/ y u2)) y))
(define-read-only (mul-div (x uint) (y uint) (z uint)) (div (mul x y) z))

(define-read-only (mul-bps (x uint) (y uint)) (/ (+ (* x y) (/ BPS u2)) BPS))
(define-read-only (div-bps (x uint) (y uint)) (/ (+ (* x BPS) (/ y u2)) y))
(define-read-only (mul-div-bps (x uint) (y uint) (z uint)) (div-bps
  (mul-bps x y) z))
```

These operations exhibit a concerning rounding behavior, consistently rounding to the nearest whole unit. This is achieved by adding half of the divisor in each case. Consequently, the result is rounded down if the original result is less than the semantic equivalent of `.5`, or rounded up if it is `.5` or greater.

This leads to both rounding up and rounding down within the same operation, depending on the inputs throughout the protocol. Such inconsistency in behavior can cause significant issues in various parts of the system, including dust accumulation, mismatched accounting, incorrect assumptions of rounding down, and other potential side effects.

Recommendation

Implement the following changes:

- Rename `mul` and `div` to `mul-precision` and `div-precision` to accurately div-precision reflect that they implicitly divide by precision.
- Create specific -down or -up versions for each function to ensure consistent rounding `-down` or `-up`.
- Inline all these functions within the contracts that utilize them, as the overhead of duplicating these one-liner functions is not justified by the read count block limitation.
- Clearly indicate the intent of each rounding operation in the codebase. Use the up or down version as necessary to ensure the protocol does not incur a loss.

While these recommendations are general, we have identified and separated several specific rounding issues into their own distinct versions.

8.3. Medium Findings

[M-01] Vaults Are Not SIP-10 Compliant

Description

The SIP-10 standard specifies that the `transfer` function should return error codes following a specific pattern:

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

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

However, the Zest vaults do not adhere to this standard. They combine all checks and return a non-standard error value.

```
(asserts!
  (and
    (is-eq tx-sender from)
    (is-eq contract-caller from)
    (> amt u0)
    (not (is-eq SELF to))
  )
  ERR-TOKENIZED-VAULT-PRECONDITIONS)
```

As a result, the vaults are not SIP-10 compliant, which may lead to issues with third-party integrations.

This issue affects all vault contracts: `vault-sbtc` , `vault-ststx` , `vault-stx` , `vault-usdc` , `vault-usdh` .

Recommendation

Separate the checks as indicated by the standard and return the appropriate error code for each case.

- `(is-eq tx-sender from)` and `(is-eq contract-caller from)` check → `u4`
- `(> amt 0)` → `u3`
- `(not (is-eq SELF to))` → custom, will be `u5`

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[M-02] Inaccurate Conversion Results Due to Missing Interest Accrual

Description

The functions `convert-to-assets` and `convert-to-shares` enable users and integrators to estimate the current exchange rate between assets and shares.

However, the calculation of `total-assets`, defined as `assets + interest - reserves`, relies on the current accrued interest. If the interest has not been updated before this computation, both the interest and reserves values become outdated.

Consequently, the `total-assets` figure is underestimated, leading to inaccurate conversion results:

- `convert-to-assets` returns a lower-than-expected value.
- `convert-to-shares` returns a higher-than-expected value.

Recommendation

Develop two versions of the conversion functions:

- `convert-to-assets-stale` and `convert-to-shares-stale`, which function exactly as they do now and are read-only.
- `convert-to-assets` and `convert-to-shares`, which first call to accrue interest, are public, and would require signing a transaction.

Another approach is to simulate an interest accrual (without saving the state) and use that previewed accrual to determine precisely how the conversions would be, even after an accrual. This also has the advantage of maintaining the two conversion functions as read-only.

[M-03] Vaults Are Tied to the Router Contract

Description

In the current system design, the `market` contract serves as the entry point for borrowing, lending, and liquidation. It is stateless and directly calls the `vault-router` as an intermediary to access the vaults, thereby avoiding the need to pass traits.

In the event of an update, the intended design is to modify the `market` and `vault-router` contracts, as they do not hold any state, while reusing the other existing contracts within the system.

However, the `vault-*` contracts have hardcoded the system authorization check, which allows access to the underlying liquidity, to the specific `market` and `vault-router` contracts:

```
(define-private (SYSTEM)
  (begin
    (asserts!
      (or
       (is-eq contract-caller market)
       (is-eq contract-caller .vault-router))
      ERR-AUTH)
    (ok true)))
```

This setup makes it impossible to reuse the vaults in the event of an upgrade involving a new asset. For instance, if a new asset is enabled and the `vault-router` does not support it, the `vault-router` needs to be changed. However, only it and the existing `market` contract are permitted to interact with the current vaults, necessitating changes to the `market` contract as well.

Recommendation

If the design intention is to avoid using traits, modify the `SYSTEM` check in all vaults to utilize pre-set, DAO-approved contract addresses (which can default to the existing `.market` and `vault-router`).

Additionally, since there is no direct call from the `market` contract to the vault contracts, only through the `vault-router`, consider whether the `(is- eq contract-caller .market)` condition should be removed.

CONTENTS

[M-04] Disabled Collateral Cannot Be Withdrawn

Description

In the `market-vault` contract, users can deposit collateral for any token that has been enabled. However, the DAO has the authority to disable any collateral token at any time using `assets::disable`.

If a collateral token that was previously approved is later disabled, users should still be able to withdraw their deposited collateral. In the current implementation, however, all withdrawal attempts for disabled collaterals will fail.

In the `market::collateral-remove` process, the notional values of all relevant assets are initially retrieved. Disabled collaterals are not included in this set.

Consequently, in `calc-delta`, when calculating the value of the asset to be removed based on the notional assets data, the operation will fail. This failure occurs when `unwrap-panic` is invoked on the result of `find-asset`, which returns `none`:

```
(let ((a     (unwrap-panic (find-asset aid alist)))
```

Recommendation

Since disabled collateral tokens are excluded from the LTV calculation, their real price-based values do not need to be computed. Instead, the value can be hardcoded to `0`.

ClarityAlliance
Security Review

Zest Protocol
v2

[M-05] Protocol Pausing Issues

Description

Borrowing and lending protocols typically involve a wide range of operations and components. Throughout the lifecycle of a protocol, there may be instances where the system, or parts of it, need to be paused, such as during component upgrades (e.g., the `market` contract).

The current codebase lacks any protocol control flow operations, making it difficult to handle black swan events or unforeseen situations.

Recommendation

Implement individual pausing mechanisms for each component and operation:

- In each vault contract, add separate pause functionality for:
  - `deposit` : providing liquidity
  - `redeem` : removing liquidity
  - `system-borrow` : borrowing
  - `system-repay` : repaying
  - `accrue-` : accruing interest
  - `flashloan` : allowing flashloans
- In each `market-vault` contract, add separate pause functionality for:
  - `collateral-add`
  - `collateral-remove`
  - `debt-add-scaled`
  - `debt-remove-scaled`

It is important to consider several caveats regarding pausability:

1. If repaying debt is blocked (either `market-vault::debt-remove-scaled` or `vault::system-repay`), then liquidation must also be blocked. Allowing users to be liquidated when they cannot protect themselves is not a valid approach.
2. If the system is generally paused, interest should not accrue (some pauses need to be correlated here); it must be skipped.
3. After unpausing liquidations (if repays were also paused), there should be a grace period during which no user can be liquidated until the period has ended. The grace period should provide enough time for users to manage their positions but not be too long, as it allows any bad debt to accumulate interest during that period; a 24-hour maximum is the recommended upper limit.
4. If interest accruing is paused, the function itself must not revert but should continue as a pass-through. Otherwise, other operations (e.g., vault depositing) are implicitly blocked as well.

CONTENTS

[M-06] Governance Proposals Never Expire

Description

When a proposal is created in the `dao-multisig`, there is currently no mechanism to add an expiration date or deadline to it. Proposals only include the script to be executed, the approvers, whether it was executed, the creation time, and its urgency.

```
(define-map proposals
            uint
            {
              script     : principal,
              approvals  : (list 20 principal),
              executed   : bool,
              created-at : uint,
              urgent     : bool
            })
```

As a result, any outdated proposal from the past can still be executed in the future, even if years have passed. This situation allows any future governance holder, whether compromised or simply in disagreement with others, to take any past proposal and approve it, potentially making it valid and executable.

Additionally, since proposals cannot be rejected, there is no way to cancel a stale proposal.

Recommendation

Introduce a deadline for proposals so that the `execute` function will reject any proposal after a specific time, and the `approve` function will also reject the approval of an expired proposal.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[M-07] DAO Can Set Invalid Multisig Configuration and Disrupt Governance

Description

Description The current `dao-multisig` implementation permits future upgrade scripts to modify both the current signers and the required approval threshold. However, there are no sanity checks on these inputs, allowing configurations that could permanently obstruct governance.

Possible scenarios include:

1. An unlimited number of signers can be added via `add-signer`, even though a maximum of only 20 can approve a proposal.
2. All signers from the multisig can be removed via `remove-signer`, effectively blocking any further operations.
3. Removing a signer must always ensure that the remaining number of signers is sufficient to meet the approval count.
4. The approval threshold can be set via `set-threshold` to a number exceeding the current number of signers, thereby blocking the multisig.

Recommendation

Monitor the current number of signers in a data variable and implement the following checks:

- Do not allow `add-signer` to add more than 20 unique signers at any time; a check in `add-signer` must ensure that an existing member is not added again.
- Do not allow `remove-signer` to be called if there is only one signer at that moment; `remove-signer` should also verify that it does not attempt to remove an already removed or non-existent member.
- Do not allow `remove-signer` to be called if the new signer count is less than the current threshold value.
- Do not allow `set-threshold` to be called with a threshold exceeding the existing number of signers.

[M-08] Vault Availability Implementation Issue

Description

With the latest changes, the `vault::available` function, which is used to manage liquidity redemption, is implemented as follows:

```
(define-read-only (available)
  (let  ((a (assets))
         (d (debt) ))
    (- a d)))
```

There are two issues with this implementation:

1. It can overflow when the debt exceeds the deposited assets.

Although `available` is only used to determine the assets available for redemption, and the `vault:redeem` function would have reverted due to the failed precondition of `(>= av inkind)`, third-party integrators may encounter problems. For instance, they might want to check availability before attempting a redemption.

2. It does not accurately represent the amount available for redemption.

Assets represent the total liquidity deposited, and debt represents the amount owed. However, their difference does not accurately reflect the tokens available for redemption. Debt includes both the borrowed amount and accumulated interest.

Consider an example where:

- Assets are 50 million
- Borrowed assets are 35 million
- Pending interest is 10 million

The availability is calculated as 0, even though there are still 15 million liquid assets in the vault that can be redeemed.

Recommendation

Modify the `available` function to check for underflow and return 0 if no funds are available. Additionally, adjust it to determine the exact borrowed amount (debt minus interest), subtract that from the assets, and use this as the correct available amount.

CONTENTS

[M-09] Liquidation Fails When Borrowers Have No Debt and No Collateral

Description

After a liquidation is executed, the system checks if bad debt socialization is necessary.

This process is initiated when the borrower's remaining collateral is exactly zero. In such instances, `socialize-debt-asset` is called for each of the borrower's debt positions, which then calls `vault-socialize-debt`, and finally `vault::socialize-debt`.

Within this sequence, the `vault::socialize-debt` function ensures that only positive debt amounts can be socialized:

```
(asserts! (> amount u0) ERR-LENDING-PRECONDITIONS)
```

However, there is no prior safeguard to prevent execution when liquidation results in both zero collateral and zero debt.

Consequently, these liquidations will always revert when they attempt to socialize non-existent debt.

Recommendation

Before initiating the bad debt socialization process, ensure that the borrower still has an outstanding debt amount.

CONTENTS

[M-10] Fee Reserve Percentage Change Is Applied Retroactively on Pending Debt

Description

The DAO has the ability to modify the fee percentage of each debt increase that is allocated to the treasury reserve. This percentage can be adjusted using the `vault::set-fee-reserve` function.

A significant oversight with this function is that it does not accrue interest on the current amount before applying the new fee.

As a result, all accrued interest, which was not accounted for under the previous configurations from the last `accrue` call until the fee change, is incorrectly split using the new fee reserve. This effectively acts as a retroactive fee change, which should not occur.

Recommendation

Invoke `accrue-` before changing the fee reserve in the `vault::set-fee-reserve` function. Ensure this change is applied to all vaults.

CONTENTS

[M-11] Utilization Configuration Changes Are Applied Retroactively on Pending Debt

Description

The DAO has the ability to modify the vaults' fund utilization configuration settings, which determine the interest rates. These configurations include the kink points (adjusted via `vault::set-points-util`) and the rate changes at each kink point.

A significant oversight with these two functions is that they do not accrue interest based on the current configurations before making changes. This oversight results in a different interest rate being applied retroactively. From the last `accrue` call up to any invocation of the configuration-changing functions, the new settings are incorrectly applied.

Recommendation

Invoke `accrue-` before making any changes to the utilization configuration. Ensure that the change is applied to all vaults. Additionally, consider merging `set-points-util` and `set-points-rate` into a single function to ensure correct correspondence between points and rates.

CONTENTS

[M-12] Accrue Must Be Called Before Pausing to Prevent Loss of Pending Debt

Description

When the DAO pauses the accrue function in the vaults, no accrual occurs during the paused period. Upon unpausing, the system skips the paused period by updating the last-update to the current time:

```
(was-paused (get accrue current))
    (now-unpaused (not (get accrue states))))
;; When unpausing accrue, jump Last-update to now to skip paused period
(if (and was-paused now-unpaused)
    (var-set last-update- stacks-block-time)
    false)
```

However, this logic overlooks the necessity of calling accrue before initiating the pause. Failing to do so results in the loss of all pending interest generated from the last accrue call up to the point of pausing.

Recommendation

Ensure that `accrue-` is called within `set-points-states` only if accrue is currently active and is about to be paused. Implement this fix across all vault.

CONTENTS

[M-13] Rounding in Bad Debt Socialization Causes Debt Mismatches

Description

After a liquidation is executed, if a position has no remaining collateral but still holds outstanding debt, the system attempts to socialize this residual debt.

This process is managed by calling `market::socialize-debt-asset` for each remaining debt asset, passing in the scaled debt amount. This function converts the scaled value back to an actual amount:

```
(actual-debt (/ (* scaled idx) PRECISION)))
```

It then calls `vault::socialize-debt` with this actual amount. Inside the vault, the amount is re-scaled:

```
(scaled-amount (/ (* amount PRECISION) idx)))
```

Since both conversions round down, the resulting socialized bad debt becomes slightly smaller than the actual remaining debt of the position. This effectively erases a small portion of the debt instead of redistributing it among the vault's borrowers.

Recommendation

Pass the original `scaled` debt amount directly to `vault-router:socialize-debt` to avoid unnecessary conversions, and modify `socialize-debt` to directly reduce the value, instead of applying the index again.

ClarityAlliance
Security Review

Zest Protocol
v2

[M-14] Precision Loss Leads to Underestimated Interest Rates

Description

The `math::mul-div` function is currently implemented as follows:

```
(define-read-only (mul (x uint) (y uint)) (/ (+ (* x y)
  (/ PRECISION u2)) PRECISION))
(define-read-only (div (x uint) (y uint)) (/ (+ (* x PRECISION) (/ y u2)) y))
(define-read-only (mul-div (x uint) (y uint) (z uint)) (div (mul x y) z))
```

The variables x, y, and z each use a precision of `1e8`, and the function's result is also expected to maintain this precision.

However, precision is applied separately in the `mul` and `div` steps. In a simplified form (when both rounding steps round down), the calculation becomes:

`x * y / PRECISION * PRECISION / z`

The terms `/ PRECISION * PRECISION` effectively discard the lower 8 digits of the `x * y` product, leading to unnecessary precision loss. This also makes the rounding direction of the product irrelevant.

As a result, derived values such as utilization, and consequently interest rates, will be underestimated.

Recommendation

Revise the `mul-div` implementation to compute simply `(x * y) / z` when rounding down and `(x * y + z - 1) / z` when rounding up, without relying on the individual `mul` and `div` functions. Then, extract the `mul-div`, `mul-div-down`, and `mul-div-up` functions into separate instances in the codebase where they are used, to avoid the contract calling overhead for just three lines of extra code.

CONTENTS

[M-15] Pyth Price Confidence Interval Is Not Validated

Description

Prices provided by the Pyth Network include a level of uncertainty, represented by a confidence interval.

Currently, the `market` contract's Pyth integration only validates the freshness of the price, not the confidence level. It is essential to validate the confidence level as well to ensure that the price returned by the network falls within an acceptable range for Granite.

For example, a price for `STX` might be `$3` with a confidence of `± $2`. In this scenario, the network is uncertain about the exact price, placing it within a `[$1, $5]` range.

Although such a situation would be highly unusual, it is still possible and could lead to financial losses for users if this price is used in collateral evaluation.

Recommendation

In the `market` contract, implement a minimum confidence threshold (price/confidence), adjustable by the DAO, to be checked when retrieving the price. Note that a confidence interval of 0 indicates no spread in price and should be considered a valid price. Ensure the confidence interval is checked in the `resolve-pyth` function.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[M-16] Oracle Freshness Check Issues

Description

Within the `market` contract, the oracle price freshness is verified using the `oracle-timestap-fresh` function, as shown below:

```
;; Oracle timestamp validation
(define-constant CARDINALITY u120)

(define-private (oracle-timestamp-fresh (ts uint) (prev uint))
  (let ((curr stacks-block-time)
        (delta (- curr ts)))
    (and
      (<= delta CARDINALITY)
      (>= ts prev))))
```

The function uses a hardcoded maximum staleness check of 120 seconds. This approach presents two long-term issues:

- The protocol cannot set a specific price freshness value for each oracle or asset type.
- The protocol cannot modify the duration, which may be necessary in extreme situations or during highly volatile periods.

Recommendation

Implement a DAO-gated setter to adjust the price freshness. Additionally, create a mapping for specific token-price feeds to verify their freshness, defaulting to a globally set value if not individually specified.

[M-17] Liquidation Penalty Can Exceed Maximum Allowed

Description

The `egroup` configuration specifies two parameters, `LIQ-PENALTY-MIN` and `LIQ-PENALTY-MAX`, which define the minimum and maximum penalty percentages applicable to liquidated debt.

The final penalty percentage is calculated using the following function:

```
(define-read-only (calc-liq-factor-bound (liq-factor uint) (bound-min uint)
  (bound-max uint))
  (+ bound-min (contract-call? .math mul-bps liq-factor
    (- bound-max bound-min))))
```

This implementation assumes that `liq-factor` will not exceed `BPS`. However, this assumption is flawed. If `liq-factor` is greater than `BPS`, the resulting penalty will exceed the maximum allowed value.

During liquidation, the `calc-liq-factor` function first calculates the percentage distance between the position's LTV and the partial liquidation LTV, relative to the full liquidation LTV. This results in `liq-pct-linear`. The value is then raised to the power of the configured `LIQ-CURVE-EXP` parameter, producing `liq-factor`.

For instance, if a position's LTV slightly surpasses the full liquidation LTV, `liq-pct-linear` will already be over 100%. When combined with a curve exponent of 100% or more (valid range is 50%-400%), the resulting liquidation factor can exceed 100%, causing the applied penalty to surpass the intended maximum.

Recommendation

Limit the calculated liquidation penalty to the maximum allowed value.

Example fix:

```
(define-read-only (calc-liq-factor-bound (liq-factor uint) (bound-min uint)
  (bound-max uint))
-  (+ bound-min (contract-call? .math mul-bps liq-factor
- (- bound-max bound-min))))
+  (let (
+    (liq-factor-capped (if (< liq-factor BPS) liq-factor BPS)))
+    (+ bound-min (contract-call? .math mul-bps liq-factor-capped
+ (- bound-max bound-min)))
+  )
+)
```

CONTENTS

8.4. Low Findings

[L-01] Non-Standard Vault Withdraw Entry Point Behavior

Description

In asset-share type smart contract vault systems, the typical entry points for adding assets to the vault and receiving shares are:

- `deposit` → add assets and receive a calculated amount of shares
- `mint` → specify a target amount of shares, and the assets to deposit are calculated

Similarly, when removing assets from a vault, the expected functions are:

- `withdraw` → remove a specific amount of assets, and the required amount of shares is calculated
- `redeem` → remove a specific amount of shares, and the received assets are calculated

The current vault implementation correctly handles the `deposit` function. However, the `withdraw` function operates like a `redeem` function, with the `amt` variable representing a share amount rather than an asset amount.

This ambiguity could lead to issues for existing DeFi third-party integrators, potentially causing integration problems.

Recommendation

Rename the `vault:withdraw` function to `redeem` across all vault instances. If a `withdraw` function is intended, modify the logic to accept asset amounts as input instead of shares, or create a separate function for this purpose.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[L-02] Incorrect Return Value on Vault Operations

Description

When a user deposits into the vault, they call `vault::deposit`, and when they withdraw funds, they call `vault::withdraw`.

In 3 out of the 6 vault contracts, these functions incorrectly return the updated total assets amount instead of the amount of assets withdrawn or the shares received.

Specifically, in the `vault-usdc`, `vault-usdh`, and `vault` contracts, the `withdraw` function returns the new total assets (`delta`) when it should return `inkind`. Similarly, the `deposit` function returns the new total assets instead of the `inkind` variable.

This inconsistency can cause issues when integrating with third parties, as some vaults return the correct amount while others do not.

Recommendation

In the `vault-usdc`, `vault-usdh`, and `vault` contracts, modify the `withdraw` and `deposit` functions to always return `inkind`.

CONTENTS

[L-03] Severe Lack of Emitted Events

Description

The entire codebase is completely devoid of any `print` statements.

This significantly restricts off-chain monitoring and integration capabilities.

Recommendation

Incorporate print events into all public functions and entry points within the codebase.

A standardized print/event structure can be implemented to facilitate off-chain processing. An example of such a structure is:

```
(print {
  action: "<function-name or action>",
  caller: <caller>,
  data: {
    <key1>: <value1>,
    <key2>: <value2>,
    ...
    <keyN>: <valueN>
  }
})
```

CONTENTS

[L-04] `assets::get-nr-enabled` Function is Unusable

Description

The `assets::get-nr-enabled` function returns the combined number of assets that are both borrowable and usable as collateral. This information is not useful for third parties because it does not indicate the total number of assets in the system.

Consider the following example:

- 10 assets
- 5 enabled as debt
- 10 enabled as collateral
- `assets::get-nr-enabled` returns 15

A third-party integrator cannot determine whether there are 15 assets, all enabled as either collateral or debt, or if there are 8 assets, with 7 enabled as both debt and collateral, and 1 enabled in only one category.

Recommendation

Either remove the function entirely, including all related `pack` logic, or modify it to return a tuple containing: the total asset count, the count of assets enabled as debt, and the count of assets enabled as collateral.

CONTENTS

[L-05] Small Loans May Be Unprofitable to Liquidate

Description

A common issue in borrowing and lending protocols is the potential unprofitability of liquidating small loans.

When the borrowed amount and the collateral deposited are too low, the discounted collateral a liquidator receives from a liquidation might not be sufficient to cover the execution cost of the liquidation operation itself.

In such cases, there is no incentive to liquidate loans that are becoming insolvent, leading the protocol to accumulate interest and, eventually, bad debt.

The issue of small positions has been widely discussed in public forums. Protocols generally adopt one of three approaches:

1. Implement a minimum borrow amount to ensure users have a sufficiently large amount of backing collateral.
2. Require borrowers to deposit a gas compensation guarantee to be used in case of liquidation. This approach is employed by Liquity V2.
3. Maintain the system as it is. Small liquid positions will accumulate over time, but in practice, these small positions have not been extensively proven to affect the markets. If necessary, governance itself would liquidate them at a loss. For example, both Euler v1 and Euler v2 follow this approach.

Recommendation

As mentioned in the description, potential solutions include implementing a minimum borrow amount or requiring borrowers to deposit a gas compensation. However, this would increase code complexity and can be added later if actually needed.

Therefore, the recommendation is to acknowledge the possibility of this issue and, if necessary, have governance liquidate small positions itself. This comes with the trade-off of paying fees to liquidate positions.

CONTENTS

[L-06] Market ZToken Data Insertion Overhead

Description

In the `market` contract, the function is-ztoken is used to determine if a token is a Zest vault token. This function relies on the ztoken-assets-ids map, which contains all the IDs hardcoded directly in the contract root.

```
(define-map ztoken-asset-ids uint bool)

;; Initialize with current ztoken asset IDs
;; NOTE: These IDs depend on the asset registration order in assets.clar
(map-insert ztoken-asset-ids u5 true) ;; vault-stx (zSTX)
(map-insert ztoken-asset-ids u6 true) ;; vault-sbtc (zsBTC)
(map-insert ztoken-asset-ids u7 true) ;; vault-ststx (zstSTX)
(map-insert ztoken-asset-ids u8 true) ;; vault-usdc (zUSDC)
(map-insert ztoken-asset-ids u9 true) ;; vault-usdh (zUSDH)

(define-private (is-ztoken (aid uint))
  (default-to false (map-get? ztoken-asset-ids aid)))
```

Since the ztokens are precisely known at deployment, there is no need to add them to a map, which incurs a read count penalty on every collateral accrual. This results in significant overhead that can be optimized.

Recommendation

Given that the project has accepted the need to redeploy the `market` contract for any new vault, it is unnecessary to add these values to a map that will remain unchanged within the current contract. Instead, a pure check can be implemented as follows:

```
;;
   NOTE: These ztoken asset IDs depend on the asset registration order in assets.clar
(define-constant zSTX   u5) ;; vault-stx
(define-constant zsBTC  u6) ;; vault-sbtc
(define-constant zstSTX u7) ;; vault-ststx
(define-constant zUSDC  u8) ;; vault-usdc
(define-constant zUSDH  u9) ;; vault-usdh
(define-constant ztokens (list zSTX zsBTC zstSTX zUSDC zUSDH))

(define-private (is-ztoken (aid uint))
  (is-some (index-of? ztokens aid)))
```

[L-07] Limited Precision in Liquidation Exponent Calculation

Description

The `LIQ-CURVE-EXPONENT` parameter is designed to control the steepness of the liquidation penalty curve with a precision of four decimal places. However, the precision is effectively lost due to the way the `reserve-calculator::calc-liq-factor-exp` function performs its calculations:

```
(define-read-only (calc-liq-factor-exp (factor uint) (exp uint))
  (if (is-eq exp BPS)
    factor
    (if (> exp BPS)
        (/ (pow factor (/ exp BPS)) (pow BPS (- (/ exp BPS) u1)))
        (sqrti (* factor BPS)))
    ))
```

In this logic, `exp` is always divided by `BPS` before being applied.

When configured in the egroup settings, this parameter is constrained to the interval `[MAX-FACTOR-MUL, MAX-FACTOR-DENOM] = [5000, 40000] = [50%, 400%]`. Consequently, only a few discrete exponent values `(5000, 10000, 20000, 30000, 40000)` yield distinct outcomes.

This effectively reduces the intended precision of `LIQ-CURVE-EXPONENT`, limiting the ability to fine-tune the penalty amounts based on the LTV level.

Recommendation

If maintaining the original precision is deemed valuable despite added complexity, the current formula should be modified to support higher precision. Alternatively, this issue should be acknowledged.

CONTENTS

8.5. QA Findings

[QA-01] Limiting Vault Borrow-Repay Recipient Interface

Description

The current market-vault borrowing mechanism interfaces operate by having the market contracts call either the `system-borrow` or `system-repay` functions of the underlying `vault` contract.

A limitation of the existing setup is that the recipient of the call is always considered to be the `tx-sender`. This imposes constraints on all integrating contracts, as the sender must remain as intended through any third-party integration pipeline.

This results in a more restrictive model and may increase the difficulty of future development.

Recommendation

Modify the vault's `system-borrow` and `system-repay` functions to accept an `account` parameter, which would serve as the `recipient` for a borrow and the benefactor for a `repay`. Additionally, update the `market` contract to utilize this interface.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[QA-02] Limiting FlashLoan Design

Description

The current flashloan functionality of the vault restricts users to passing only a 34-byte buffer type argument named `memo`.

This limitation could lead to integration challenges for third-party protocols, as they are unable to pass dynamic values through the flashloan functionality to the callback. Consequently, they are compelled to pre-save these values on the callback contracts.

Recommendation

To enhance the general applicability of flashloans, consider replacing the optional memo with an optional `calldata` or `data` parameter, allowing for at least 4k bytes.

CONTENTS

[QA-03] Eliminate Redundant `begin` Blocks in `let` Blocks

Description

Across the codebase, `begin` blocks are unnecessarily added after `let` declarations. This is redundant because `let` blocks naturally allow subsequent statements without needing a `begin` block. This pattern is prevalent in almost all contracts, and all instances of these `begin` blocks can be removed.

Recommendation

Remove the `begin` blocks throughout the codebase and integrate the inner logic directly after the `let` variable declarations.

ClarityAlliance
Security Review

Zest Protocol
v2

[QA-04] Improve Switch-Like Statements

Description

The codebase contains several instances of switch-like statements implemented using multiple `if-else` clauses.

From a formatting perspective, instead of using `if-else` with continuous rightward indentation:

```
(if (is-eq type TYPE-PYTH)
      (resolve-pyth ident)
      (if (is-eq type TYPE-DIA)
          (resolve-dia ident)
            ERR-TYPE)))
```

it is preferable to format them on separate lines, like this:

```
(define-private (resolve-price (type (buff 1)) (ident (buff 32)))
  (if (is-eq type TYPE-PYTH) (resolve-pyth ident)
  (if (is-eq type TYPE-DIA) (resolve-dia ident)
  ERR-TYPE)))
```

Alternatively, you can create a switch equivalent using inverted `assets!`. For example:

```
(define-private (resolve-price (type (buff 1)) (ident (buff 32)))
  (begin
    (asserts! (not (is-eq type TYPE-PYTH)) (resolve-pyth ident))
    (asserts! (not (is-eq type TYPE-DIA)) (resolve-dia ident))
    ERR-TYPE
  )
)
```

These improvements can be particularly beneficial for the `vault-router` contract.

Recommendation

Revise the `if-else` switch-like clauses to a more comprehensible form, as suggested above.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[QA-05] Severe Violation of Single Responsibility Principle

Description

Over the years, several standard programming patterns have been recognized for producing good, easy-to-maintain, and reliable code. One such fundamental pattern is the Single Responsibility Principle (SRP), which dictates that certain parts of the codebase should be responsible for only one task.

Throughout the codebase, there are multiple instances of severe SRP violations. These not only decrease code readability but also significantly increase the difficulty of maintenance and extension.

A few examples include:

- A single function is used for both adding and removing debt, where the caller specifies whether to add or subtract the debt: `market-vault::insert-debt-scaled`.
- A single function is used for both adding and removing user collateral, where the caller specifies whether to add or subtract the collateral amount: `market-value::insert-collateral`.
- In the market vault, a single function handles both sending and receiving asset tokens, where the caller passes a boolean to indicate the intention to receive tokens: `market-vault::transfer`.
- In the vault, a single function manages both sending and receiving underlying tokens, where the caller passes a boolean to indicate the intention to receive tokens: `vault::utransfer`.

Additionally, due to the design of Clarity smart contracts, supporting dual operational roles per function in the `market-vault` case increases almost all possible execution costs, further degrading the quality of the codebase.

Recommendation

Implement separate functions for each distinct functionality. If there is common functionality within the resulting split functions, refactor that code logic into common functions and utilize them accordingly.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[QA-06] Inline All Pack and Mathematical Operations

Description

In Clarity, invoking external contracts leads to a notable increase in read count.

There are several operations within the codebase that repeatedly call either the `pack` or `math` contracts, significantly increasing the block cost overhead.

Recommendation

Inline as many operations as possible from the `pack` and `math` contracts.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[QA-07] Debt Cap Versus Borrow Cap Considerations

Description

The current system logic permits each vault to maintain a specific debt amount at any given time. Debt encompasses both the borrowed principal and the accrued interest.

By linking the protocol contract's capping mechanism to the total debt rather than just the borrowed principal, situations may arise where only a portion of the underlying assets is utilized due to significant debt accrual.

Consider the scenario of a USDC vault with:

- 100 million in assets
- 50 million in borrowed principal
- a 75 million debt cap
- 30 million in interest on the 50 million borrowed

In this situation, although 50% of the assets remain unborrowed, no additional borrowing can occur because the total debt has reached 80 million, surpassing the debt cap.

Recommendation

Evaluate whether a debt cap or a borrow cap is more suitable from a business logic perspective. If the current implementation is intentional, acknowledge this issue; otherwise, consider changing the cap to a borrow cap instead of a debt cap.

CONTENTS

[QA-08] Typographical Error

Description

Within the `oracle` contract, there is a typographical error in the `STSTX-RAITO-DECIMALS` constant. The term `RAITO` should be corrected to `RATIO`.

These types of errors can diminish code consistency and slightly hinder comprehension.

Recommendation

Correct the identified typo.

CONTENTS

[QA-09] Optimization of Accrue for Liquidity and Index Reads

Description

The `vault::accrue` function is designed to return the index and liquidity index after updating them, if necessary. However, the current implementation is not optimal.

The public `accrue` function is structured as follows:

```
(define-public (accrue)
  (begin
    (accrue-)
    (ok { index: (index), lindex: (lindex) })))
```

In this setup, the `index` and `lindex` functions each increase the read count to return the updated indexes. However, the `accrue-` function already determines, updates, and saves these values when necessary, but it only returns `true`.

Recommendation

It is recommended to modify the `accrue-` function to return the updated interest and liquidity indexes directly, eliminating the need for the public `accrue` function to re-read them.

CONTENTS

ClarityAlliance
Security Review

Zest Protocol
v2

[QA-10] Cleanup Codebase Functions

Description

The codebase employs several unconventional naming conventions and implementation choices:

- Private function names and some variables have a trailing hyphen
- Instead of directly calling `var-get` to retrieve a local variable, a redundant wrapper is used that merely calls `var-get` .
- There are identical versions of functions that exist as both public and private.

These choices significantly impair code readability.

Recommendation

Eliminate the trailing hyphen in naming, remove the redundant getter wrappers, and eliminate any duplicated functions.