ClarityAlliance

GRANITE (UPGRADE V2) SECURITY REVIEW

Conducted by:
KRISTIAN APOSTOLOV, ALIN BARBATEI (ABA)

JULY 2ND, 2025

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade 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

Granite
(Upgrade 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

Granite
(Upgrade v2)

3. Introduction

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

4. About Granite

Granite is a Bitcoin Liquidity Protocol that provides the first truly non-custodial, secure, and decentralized way to borrow against Bitcoin.

The protocol allows borrowers to take stablecoin loans using Bitcoin as collateral, without exposure to counterparty or rehypothecation risk. Liquidity providers can earn yield on stablecoins by providing liquidity to the pool, which is then lent to borrowers.

Loans in Granite are best thought of as lines of credit, without set terms or repayment schedules. As long as the borrower maintains an adequate loan-to-value ratio (LTV), keeping their account in good health, they are not subject to liquidation. If a borrower's LTV falls too low, a portion of their capital will be liquidated to bring their account back to solvency.

Granite enables BTC users to access DeFi without centralized custodians by leveraging Stacks' soon-to-be-launched Nakamoto upgrade and sBTC Bitcoin bridge.

CONTENTS

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

Granite
(Upgrade v2)

6. Security Assessment Summary

Scope

The following contracts were in the scope of the security review:

- `contracts/lp-incentives-v2.clar`
- `contracts/flash-loan-v1.clar`
- `contracts/modules/daily-caps-v1.clar`

Additionally, all updates to the Clarity smart contracts in the repository at the time of review were reviewed.

Initial Commit Reviewed:
20ff7dabb5448cc820ac544036522bcaa533cf3c

Intermediate Commit Reviewed:
2f3dc203a4de4359f69598f8d5e3b0d05845de3c

Final Commit After Remediations:
4f24f304dece7ea3f3560d0b3cf416dc2dbfc060

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

7. Executive Summary

Over the course of the security review, Kristian Apostolov, Alin Barbatei (ABA) engaged with - to review Granite. In this period of time a total of **28** issues were uncovered.

Protocol Summary

Protocol Name	Granite
Date	July 2nd, 2025

Findings Count

Severity	Amount
High	2
Medium	6
Low	5
QA	15
Total Findings	**28**

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

Summary of Findings

ID	Title	Severity	Status
[H-01]	FlashLoan Fee Is Not Accounted for in the State Contract	High	Resolved
[H-02]	Daily Caps Vulnerable to Abuse, Blocking All Capped Operations	High	Resolved
[M-01]	FlashLoan Fee Decimal Scaling Can Strip Fee Completely	Medium	Resolved
[M-02]	LP Incentives Scaling Can Strip Rewards	Medium	Resolved
[M-03]	Staking Contract Scaling Can Strip Withdraw Slashing	Medium	Resolved
[M-04]	Compromised Governance Can Instantly Drain Granite	Medium	Resolved
[M-05]	Proposals Don't Expire and Can't Be Canceled After Timelock Maturation	Medium	Resolved
[M-06]	Scaling Collateral Valuation to Market Decimals Introduces Precision Loss for Low Decimal Markets	Medium	Acknowledged
[L-01]	Full Protocol Pause Does Not Affect Flash Loans	Low	Acknowledged
[L-02]	Missing Bulk Claiming Rewards for Incentives Contract	Low	Acknowledged
[L-03]	Incentive Snapshot Amounts Are Not Correlated	Low	Acknowledged
[L-04]	Inconsistent Checks Between get-liquidation-data and Liquidating a Position	Low	Resolved
[L-05]	Minted Blocks Are Not a Reliable Time Measurement Unit	Low	Resolved
[QA-01]	FlashLoan Fee Amount Cannot Be Changed	QA	Resolved
[QA-02]	FlashLoan Allowed Contracts Cannot Be Dynamically Added On Mainnet	QA	Resolved
[QA-03]	LP Incentives Contract Snapshot Uploader Cannot Be Changed	QA	Resolved
[QA-04]	LP Incentives Contract Optimization	QA	Resolved
[QA-05]	Post Safety Module Wipe Considerations	QA	Resolved
[QA-06]	Improvements Suggested for the liquidator-v1 Contract	QA	Resolved
[QA-07]	Scaling Factor Ambiguities	QA	Resolved

CONTENTS

Summary of Findings

ID	Title	Severity	Status
[QA-08]	Withdrawal Caps Contract Can Be Slightly Improved	QA	Resolved
[QA-09]	Withdrawal Caps Are Not Validated to Remain Below 100%	QA	Resolved
[QA-10]	Detach Withdrawal Caps Scaling Factor From Constants to Avoid Future Ambiguity	QA	Resolved
[QA-11]	Governance Contract Can Be Slightly Improved	QA	Resolved
[QA-12]	Overlapping Error Code Ranges	QA	Resolved
[QA-13]	Remove Outdated Bad Debt Comment	QA	Resolved
[QA-14]	Remove Unused Let Variable Declarations	QA	Resolved
[QA-15]	Ambiguous Reversion on Repayment When Borrower Has No Debt	QA	Resolved

8. Findings

8.1. High Findings

[H-01] FlashLoan Fee Is Not Accounted for in the State Contract

Description

The `flash-loan-v1` contract is designed to allow users to execute flash loans of the market token from the Granite start contract. These loans are facilitated through the `flash-loan` function, which imposes a fixed 0.01% fee on the loaned amount.

However, while the fee amount is left in the `state-v1`, the internal contract accounting does not reflect this, rendering the fee effectively unused.

Since governance lacks a direct method to extract arbitrary token amounts from the state contract, and all Granite token operations rely on accounted amounts, the fee amount becomes lost or blocked.

Recommendation

Several options are available, depending on the protocol's intent. In the first two scenarios, the fee must first be transferred to the `flash-loan-v1` contract. After the final `state-v1::transfer-from`, an additional `transfer-to` should be added to transfer the fee to the flash loan contract. From there, it can either:

1. Be sent to the governance contract, allowing the protocol to decide whether to add it to the reserve or withdraw it for team expenses.
2. Be donated to the state contract through a combination of `add-assets` and `remove-assets` calls, which will leave one asset unit stranded.
3. Be directly transferred to a different fee recipient (this requires a governance action to set the flash loan fee or a separate owner on the flash loan contract).

If option (2) is implemented, the `add-assets` call would allocate the entire fee as assets but only mint one unit of LP shares. This single share unit would be entitled to liquidity but is practically insignificant. If desired, a subsequent `remove-assets` call with one share and one asset amount (necessary to avoid reversion) would result in no LP shares being minted but would leave one asset unit blocked in the flash loan contract until the next loan is repaid, at which point it can be utilized.

CONTENTS

The entire overhead for option (2) is necessary because `increase-total-assets` cannot be called on the mainnet.

To remove the blocked funds, governance can deploy a separate contract solely for extracting the fee from the state contract, setting it as approved. This arbitrary contract can directly extract token funds by invoking the underlying `state-v1::transfer-to` function.

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[H-02] Daily Caps Vulnerable to Abuse, Blocking All Capped Operations

Description

The latest version of Granite has introduced a time-dependent cap (initially set daily) on several operations: removing liquidity, removing a borrower's collateral, and borrowing.

This mechanism presents a potential issue where a malicious actor can repeatedly perform symmetrical actions (e.g., deposit + withdraw, or borrow + repay) to exhaust the caps, thereby blocking any further similar operations within the protocol until the cap is replenished.

The three caps introduced allow an attacker to saturate them at no cost, aside from on-chain execution fees:

1. In `liquidity-provider-v1`, an attacker can call `deposit` with an arbitrary amount (ensuring it remains within the allowed total protocol asset limit) and then call `withdraw` to remove it.
2. In `borrower-v1`, an attacker can call `borrow` with an arbitrary amount and then, within the same transaction, call `repay` with the entire amount.
3. In `borrower-v1`, an attacker can call `add-collateral` with an arbitrary amount and then, similarly, within the same transaction, call `remove-collateral`, ensuring they are not in a liquidatable position.

Granite imposes a fee only on generated open interest. Therefore, if borrowing-repaying and adding-removing collateral are executed in a loop from a smart contract, the attacker incurs no fee. Additionally, liquidity providers are not charged any fee, enabling this attack on the LP removal cap.

An attacker might execute this attack to damage the protocol's reputation or as part of a larger hack to prevent LP providers from withdrawing their tokens promptly. This situation would persist until the caps are adjusted through a governance action, which is currently subject to a timelock.

Recommendation

To address or mitigate this issue when using a global cap, two common approaches are recommended:

1. Implement a fee on any operation subject to a time-dependent cap.
2. Introduce a waiting queue or delay between the initiation and execution of an operation.

Imposing a static fee on operations such as borrowing, removing collateral, or adding LP is detrimental to user onboarding and misaligned with Granite's operational model. Introducing a waiting queue may be feasible for operations like removing LP, where such a queue can be justified.

CONTENTS

For example, a queue is implemented when removing LP tokens from the staking contract.

Pending collateral removal or borrowing cannot be implemented without significant overhead and changes to the protocol design.

After extensive discussions and brainstorming sessions with the team, we concluded that the best solution is to account for all inflows (non-capped). When the available outflow amount exceeds the initially desired maximum, a decay logic is applied. This decay logic swiftly reduces the outflow to the initially intended maximum amount.

This approach completely nullifies any flash loan-dependent attack variations and, depending on the decay window length, forces the attacker to keep tokens within the protocol, like any regular user, for the desired interval. This further removes any incentive and increases the attacker's loss, as they are compelled to act as a normal participant in the protocol.

This solution aligns with Granite's architectural design while providing the necessary protection to limit outflows and prevent the aforementioned DOS attack.

ClarityAlliance
Security Review

Granite
(Upgrade v2)

8.2. Medium Findings

[M-01] FlashLoan Fee Decimal Scaling Can Strip Fee Completely

Description

The `flash-loan-v1` contract is designed to enable users to flash-loan the market token from the Granite start contract. Loans are processed through the `flash-loan` function, which imposes a hardcoded 0.01% fee on the loaned amount.

An issue arises with the fee scaling process, which can result in the fee being completely eliminated for markets with tokens having fewer than 3 decimals.

The problem occurs due to the way the fee is scaled:

```
(scaled-fee
  (contract-call? .math-v1 to-fixed fee scaling-decimals market-decimals))
(flash-loan-fee (contract-call? .math-v1 divide-round-up
  (* amount scaled-fee) scaling-factor))
```

The `scaled-fee` variable initially scales the `fee` amount to match the market decimals. The fee amount is a fixed value of `10_000`, representing 0.01% of `100_000_000`. If the scaling, which utilizes `math-v1:: to-fixed`, reduces the fee excessively, it rounds down to 0. The final fee amount, `flash-loan-fee`, is determined by applying the `scaled-fee` percentage to the loan amount.

Consider the following scenario:

- Fee: `10_000/100_000_000` (0.01%)
- Token decimals: 2
- Flash loan amount: 100,000 full tokens, equating to `10,000,000`
- Expected fee: ceil (10,000,000 * 10,000/100,000,000) ⇒ `1,000` units
- However, because the fee percentage (treated as 8 decimal scaled) is first adjusted to market tokens (2 decimals)
- The intermediary scaled fee is `10_000 / pow(8 - 2)` ⇒ `10_000 / 1,000,000`, which rounds down to 0, resulting in no fee deduction.

Scaling the fee percentage (`scaled-fee`) is both unnecessary and introduces the aforementioned issue for tokens with low decimals. It is redundant because the fee is a percentage of 108, which remains consistent regardless of market decimals.

Recommendation

Eliminate the `scaled-fee` calculation entirely (including `scaling-factor`, `market-decimals`, and `scaling-decimals`) and directly compute the `flash-loan-fee` using the `fee` percentage and a 100% fee equivalent.

CONTENTS

Example implementation:

```
;; CONSTANTS
 (define-constant SUCCESS (ok true))
-(define-constant scaling-factor (pow u10
- (contract-call? .constants-v1 get-market-token-decimals)))
-(define-constant market-decimals
- (contract-call? .constants-v1 get-market-token-decimals))
-(define-constant scaling-decimals u8)
+(define-constant max-fee u100000000)
 ;; Fee of 0.01% for processing flash loan scaled to 10^8
 (define-constant fee u10000)

@@ -42,8 +40,7 @@

 (define-public (flash-loan (amount uint) (callback <callback-trait>) (data
   (optional (buff 20480))))
   (let (
-      (scaled-fee
- (contract-call? .math-v1 to-fixed fee scaling-decimals market-decimals))
-      (flash-loan-fee (contract-call? .math-v1 divide-round-up
- (* amount scaled-fee) scaling-factor))
+      (flash-loan-fee (contract-call? .math-v1 divide-round-up
+ (* amount fee) max-fee))
       (amount-with-fee (+ amount flash-loan-fee))
       (caller contract-caller)
       (callback-contract (contract-of callback))
```

[M-02] LP Incentives Scaling Can Strip Rewards

Description

The `lp-incentives-v2` contract employs a scaling constant determined by the market token's decimals:

```
(define-constant scaling-factor (pow u10
  (contract-call? .constants-v1 get-market-token-decimals)))
```

This scaling factor is applied in two scenarios. The first is when calculating the percentage of an epoch that has elapsed relative to the total epoch duration:

```
(ok (/ (* (- snapshot-time prev-snapshot-time) scaling-factor)
  (- epoch-end-time epoch-start-time)))
```

The second application is in determining the percentage of LP tokens held by a user, which is then used to calculate the rewards they are entitled to:

```
(percent-of-lp-shares (/ (* lp-shares scaling-factor) snapshot-lp-shares))
(snapshot-rewards (/ (* percent-of-epoch percent-of-lp-shares total-rewards)
  (* scaling-factor scaling-factor)))
```

In both cases, if the scaling constant is too low, it can lead to issues.

Consider a market token with 2 decimals and the following snapshot duration scenario:

- Epoch duration: 1 month `2592000` seconds
- Scaling factor: `100` (for a 2-decimal market token)
- The snapshot epoch percentage is calculated as:
  - `‹elapsed_time> * scaling-factor / ‹epoch_duration›` ⇒ `‹elapsed_time> * 100 / 2592000` ⇒ `‹elapsed_time> / 25920`.
  - Therefore, if `elapsed_time` is less than 25920 seconds (approximately 7 hours and 12 minutes), the epoch percentage rounds down to 0, resulting in no rewards.

Continuing with the example, focusing on the LP shares percentage calculation:

- Scaling factor: `100` (for a 2-decimal market token)
- `snapshot-lp-shares`, total LP deposited in the period: 5000 LPs (due to a large investor or favorable market conditions)
- The percentage of rewards a user receives is calculated as:
  - `<lp-shares> * scaling-factor / snapshot-lp-shares` ⇒ `<lp-shares > * 100 / 5000` ⇒ `‹lp-shares> / 50`
  - Thus, if anyone holds less than 50 full LP share tokens, their rewards would round down to 0.

While there are normal cases where rounding user rewards down to 0 is unavoidable, since the total LP amount is beyond team control, the scaling factor should be designed to minimize such occurrences.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

Recommendation

For any percentage-related scaling, use a larger, fixed scaling value, such as `u100000000`, instead of a dynamically changing one.

[M-03] Staking Contract Scaling Can Strip Withdraw Slashing

Description

When bad debt is socialized during a liquidation, the unbacked debt is first deducted from the staked LP token holders.

This slashing is executed in the `staking-v1::slash-total-staked-lp-tokens` function, impacting both regular stakers and amounts pending withdrawal.

The slashed amount is proportionally distributed between the pending withdrawal and active staking tokens.

```
(withdrawal-lp-token-rate (/
  (* withdrawal-lp-tokens scaling-factor) total-staked-lp-tokens))
(withdrawal-lp-tokens-to-slash (/
  (* lp-tokens withdrawal-lp-token-rate) scaling-factor))
(active-staked-lp-tokens-to-slash (- lp-tokens withdrawal-lp-tokens-to-slash))
```

The percentage of withdrawn LP tokens (`withdrawal-lp-token-rate`) is calculated using a scaling variable (`scaling-factor`).

A concern with the scaling factor is its direct proportionality to the market token decimals.

```
(define-constant scaling-factor (pow u10
  (contract-call? .constants-v1 get-market-token-decimals)))
```

Since the scaling factor is used to determine the ratio or percentage of withdrawn LP tokens relative to the total staked LP tokens, it remains unaffected by any decimal scaling and is ideally a larger value.

By tying it to the market decimals, consider scenarios with low decimal markets, such as 2 decimals in the following example:

- `scaling-factor` : 100
- `withdrawal-lp-tokens` : 10
- `total-staked-lp-tokens` : 1500
- The `withdrawal-lp-token-rate` is calculated as: `100 * 10 / 1500` , which rounds down to 0, meaning the pending withdrawal amounts do not incur any penalty.

For a 2-decimal scaling factor, the ratio of pending to total of 100 rounds down to zero, which, although rare, can occur. This logic can be extended to fewer decimals and more decimals, with a decrease in precision loss as the number of decimals increases.

Recommendation

Use a constant scaling factor that provides sufficient granularity. Generally, in the Stacks ecosystem, `10^8` is used for such cases.

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[M-04] Compromised Governance Can Instantly Drain Granite

Description

The newly introduced timelock mechanism is designed to provide market participants with a guaranteed period to process any significant changes to the market's state. This is particularly crucial in scenarios such as a malicious governance compromise or the introduction of a faulty proposal.

Currently, only a limited number of actions are protected by a timelock. For those actions that are not, a compromised governance could immediately deplete the market's resources.

Consider the following scenario:

- A majority of governance members are hacked, resulting in the contract falling under malicious control.
- The malicious governance can instantly halt all methods of withdrawing tokens from Granite using an `ACTION_SET_MARKET_PAUSE_FLAG` action or by individually pausing each operation.
- The malicious governance can then introduce a harmful contract to the list of allowed contracts via `ACTION_SET_ALLOWED_CONTRACT`.
- Neither of these two actions is protected by a timelock.
- The malicious contract could then directly siphon funds from the market through a `state-v1:: transfer-to` call, effectively draining the contract.

Recommendation

In the worst-case scenario, to ensure market participants have at least one timelock period to withdraw from the market, all actions affecting outbound token flow should be subject to a timelock.

There are additional actions that also require timelocking, detailed as follows. For each, the reason for its inclusion is explained:

- `ACTION_SET_WITHDRAW_ASSET_FLAG` : Necessary because it can directly prevent users from withdrawing their LP.
- `ACTION_SET_REMOVE_COLLATERAL_FLAG` : Can directly prevent users from withdrawing collateral, even after repaying a loan.
- `ACTION_SET_REPAY_FLAG` : In the event of a hack, borrowers would need to repay the loan first to retrieve their original collateral.
- `ACTION_SET_MARKET_PAUSE_FLAG` : Users should not be immediately blocked in this scenario.
- `ACTION_SET_ALLOWED_CONTRACT` : This action can be used to instantly drain the market by approving a malicious contract.
- `ACTION_REMOVE_ALLOWED_CONTRACT` : This can block users from withdrawing their funds by removing all peripheral contracts, such as `liquidity-provider-v1` leaving lenders without any means to withdraw liquidity.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

- `ACTION_SET_STAKING_FLAG` : Stakers may be prevented from withdrawing their LPs.

The protocol team must still be able to pause the market in case of a different emergency (e.g., a hack), which is managed by the `guardian` role. This role can instantly pause the market via `governance-vi::guardian-pause-market` call. Therefore, adding a timelock to outbound token flows in governance does not restrict the team's ability to respond to emergencies or restart the protocol promptly.

It is important to note that adding a timelock `ACTION_SET_STAKING_FLAG` will prevent instant feature unblocking since staking lacks separate actions for enabling and disabling, or for inflow and outflow granularity (staking/unstaking). Therefore, an instant staking pause should also be implemented via the `guardian-pause-market` call (`contract-call? .state-v1 set-staking-flag false`) for emergency use.

CONTENTS

[M-05] Proposals Don't Expire and Can't Be Canceled After Timelock Maturation

Description

The current governance proposal mechanism allows:

- Proposals to expire if they have not reached quorum within a designated time.
- Proposals to be canceled if they have not reached quorum and all votes have been cast.

With the introduction of the new timelock mechanism, the previous execution point is no longer the start of the timelock period.

However, the cancellation and expiration logic was not updated to account for proposals that have reached quorum. After the 24-hour timelock wait period, the team may decide they do not wish to implement them. In such cases, proposals remain in a state where they can be executed by any member of governance, now or in the future.

This situation can lead to currently unwanted proposals being applied later, potentially causing significant market impact.

Recommendation

In the `execute-if-approve-threshold-met` function of the `governance-v1` contract, when the `threshold` is reached for the first time and the `governance-proposal` map is updated with the timelock maturation deadline, also include a timelock execution deadline in the map. This can initially be a hardcoded offset value, such as 24 hours after timelock maturation. Then, modify the `execute` function to check this deadline and revert if it has expired.

The `close` function can also be modified to include expired, matured proposals.

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[M-06] Scaling Collateral Valuation to Market Decimals Introduces Precision Loss for Low Decimal Markets

Description

The Granite protocol operates with several tokens, each having specific decimal considerations:

- Each protocol deployment has only one market token, with its decimals noted as `MARKET-TOKEN-DECIMALS`.
- Each collateral has its own decimals, which are saved and retrieved for price conversion.
- All prices are denoted with a precision of `10^8` and scaled using the `PRICE-SCALING-FACTOR`.

Within the protocol logic, whenever the collateral value in USD is required, the amount is adjusted to the market decimal precision. This adjustment is used in scenarios such as health check calculations or determining the returned collateral amount after a liquidation.

This process can lead to precision loss if the market decimals are too low.

To illustrate the truncation, we compare a `liquidation` call executed with initially identical amounts but scaled differently. Two proofs of concept (POCs) were conducted: one with a standard 8-decimal market token and another with a 6-decimal market token. In both cases, the collateral has 10 decimals.

For an 8-decimal market token, the `collateral-amount` calculated in `calc-collateral-to-giv` is shown to be `3897185`. This amount is in "value," meaning it is scaled to market decimals. The next operation converts it back to the original collateral precision, resulting in `389718500`. Notably, since there is a 2-decimal difference between the market and collateral, the intermediary amount is simply multiplied by 100.

In this scenario, the liquidator's repay amount is `69592592592`.

For a 6-decimal market token, the same `collateral-amount` is truncated by 2 more positions, resulting in `38971`. Consequently, the repay amount is also truncated by 2 positions, becoming `695925925` (compared to `69592592592`).

However, the collateral to be given to the liquidator, which remains unchanged between the two tests, is `389710000`.

Between the 8-decimal and 6-decimal Granite markets, due to intermediary truncation to market decimals, a liquidator receives fewer tokens in the latter case. In the example provided, the difference is `389718500 - 389710000 = 8500` units.

CONTENTS

This loss would compound over time with each liquidation, leading to liquidator losses.

These precision losses also slightly affect whether a user is perceived as healthy. The losses are also present in the borrower contract.

The greater the difference between the market token decimals and collateral decimals, the more significant the precision loss.

Recommendation

Throughout the entire codebase, use a distinct value precision for decimals when comparing values (e.g., debt vs. collateral for LTV /health, returned amount). Convert both the market token and collaterals to this precision, rather than the current method of adjusting collateral to market decimals.

If the precision loss is deemed acceptable, ensure it is thoroughly documented. However, if working with market tokens below 6 decimals, addressing this issue is essential.

8.3. Low Findings

[L-01] Full Protocol Pause Does Not Affect Flash Loans

Description

Granite has implemented a granular pause for specific operations/features, as well as a general market pause, through the `state-v1::pause-market` function.

Currently, none of the existing flags can prevent the flash loan protocol from operating, as there is no gating mechanism in place.

The only way to halt flash loans is to remove the `flash-loan-v1` contract from the list of approved contracts (equivalent to a pause) and then add it back (equivalent to an unpause).

Recommendation

A separate flag specifically for flash loans is necessary. This would require a different state contract for auxiliary functions. The current flash loan contract would need to check this contract to determine if the functionality is paused.

This change would also necessitate an action in `governance-v1`, which would include both a standalone version for pausing only the flash loan functionality and actions coupled with `ACTION_SET_MARKET_PAUSE_FLAG` and `ACTION_SET_MARKET_UNPAUSE_FLAG`.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[L-02] Missing Bulk Claiming Rewards for Incentives Contract

Description

The `lp-incentives-v2` contract is designed to incentivize users to hold Granite LPs. Currently, a privileged principal, known as the `snapshot uploader` , is required to identify LP holders using the `upload-snapshot` function. Subsequently, either the `snapshot-upload` or any other user can distribute rewards to these holders through a `claim-rewards` call.

As it stands, the incentives epoch cannot conclude until all rewards have been distributed. Any rewards that remain unallocated due to a lack of holders will be reclaimed through the `transfer-remaining-lp-tokens` call.

While the `upload-snapshot` function allows for setting 50 holders at a time, there is no corresponding bulk option for claiming rewards for multiple users.

Users will eventually want to claim their rewards, but they are not necessarily in a hurry to do so, especially if their rewards are not substantial. This means that any leftover rewards for the team to reclaim will have to wait until these positions are cleared.

Recommendation

Implement a bulk `claim-rewards` function to facilitate mass claiming in situations where users are not in a rush to claim, allowing the epoch to end more efficiently.

[L-03] Incentive Snapshot Amounts Are Not Correlated

Description

In the `lp-incentives-v2` contract, the snapshot uploader uses the `upload-snapshot` function with the necessary inputs to determine the corresponding holder-reward amounts.

This function requires, among other inputs, `details.total-lp-shares`, which represents the total number of LP share tokens held cumulatively during this part of the epoch. It also takes a list of users and the amount of shares they held during this period.

Using these values, the implementation calculates the percentage of rewards allocated to each holder based on the proportion of LP shares they held relative to the total LP shares for that period.
However, there is no validation to ensure that the total amount of user tokens, when summed, equals the provided `total-lp-shares`.

Without such validation, incorrect entries may inadvertently be added, leading to issues such as:

- Holders' reward percentages may be incorrect if an excessive amount is given, or insufficient at claim if a lower amount is provided.
- In extreme cases, more than 100% could be allocated to a single user, as no holder share percent validation is performed.

Another related issue arises during periods of the epoch when no users hold any LP tokens. For these periods, `upload-snapshot` allows the `batch` list to remain empty, but it still requires the `total-lp-shares` to be greater than 0.

This check should not be enforced when there are no holders.

Recommendation

Remove the `(asserts! (> (get total-lp-shares details) u0) ERR-ZERO-LP-SHARES)` check and modify `fold-upload-snapshot` to calculate the total LP shares of all users from the batch list (or 0 if there are none). Then, verify that the calculated sum matches the provided `details.total-lp-shares` amount.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[L-04] Inconsistent Checks Between get-liquidation-data and Liquidating a Position

Description

The `liquidator-v1::get-liquidation-data` function is utilized by third-party integrators to determine how a liquidation would proceed.

In the standard user liquidation process, liquidators are permitted to pass a repay amount of 0 only if the collateral price is 0. This is enforced by the `ensure-non-zero-repay-amount` call within the `execute-liquidation` function.

However, the `get-liquidation-data` function does not implement this check, allowing callers to pass values that would be disallowed in an actual liquidation call, leading to a potential revert.

Recommendation

In the `get-liquidation-data` function, retrieve the `collateral-price` from the `liquidation-info` and invoke the `ensure-non-zero-repay-amount` function to ensure consistent behavior with an actual liquidation.

Example fix:

```
maybe-market-asset-price
       maybe-total-liquid-ltv
       maybe-collateral-value
-      maybe-collateral-price
-  )))) (ok {liquidation-info: (get liquidation-info liquidation-info)})))
+      maybe-collateral-price)))
+    (collateral-price (get collateral-price liquidation-info))
+  )
+  (try!
+ (ensure-non-zero-repay-amount liquidator-repay-amount collateral-price))
+  (ok {liquidation-info: (get liquidation-info liquidation-info)})))
```

CONTENTS

[L-05] Minted Blocks Are Not a Reliable Time Measurement Unit

Description

In the newly implemented governance timelock mechanism, the timelock period is set to a fixed number of 17,280 Stacks blocks.

```
;; Timelock period before executing an approved proposal
;; approximately 24 hours
(define-constant TIME_LOCKED_PERIOD 17280)
```

Theoretically, a Stacks block is minted approximately every 5 seconds. However, real-time data indicates significant variability. For instance, using the Hiro API for time averages at Stacks Block #1,624,256 shows an average of 4.22 seconds per block over the last 24 hours.

```
{
    "last_1h": 3.99,
    "last_24h": 4.22,
    "last_7d": 5.96,
    "last_30d": 5.76
}
```

At 4.22 seconds per block, the timelock would last approximately 20 hours and 15 minutes. Considering the average time over the last 7 days, with a duration of 5.96 seconds, the timelock could extend to 28 hours and 36 minutes.

In practice, the timelock period could fluctuate between 20 and 28 hours.

Recommendation

It is generally not advisable to use the number of mined blocks to estimate time due to its inherent uncertainty. Therefore, one proposed solution is to use the previous Stacks block time. If the team still prefers to use minted blocks as a time measurement, the comment should be updated to indicate that the duration can vary by a few hours, acknowledging this as an accepted consideration.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

8.4. QA Findings

[QA-01] FlashLoan Fee Amount Cannot Be Changed

Description

The FlashLoan functionality currently imposes a fixed fee of 0.01% on the loaned amount. This fee is hardcoded and cannot be adjusted.

With the fee being unchangeable, third-party protocols offering similar services can set their fees lower than Granite's, making Granite less attractive from an economic standpoint and potentially reducing the influx of fees.

Recommendation

Modify the `flash-loan-v1` to allow for the fee to be adjustable. This change will require a corresponding update in `governance-v1`.

[QA-02] FlashLoan Allowed Contracts Cannot Be Dynamically Added On Mainnet

Description

The `flash-loan-v1` contract is designed to enable users to perform flash loans of the market token from the Granite start contract.

The `flash-loan` function restricts usage to approved contract callbacks only:

```
(asserts! (default-to false
  (map-get? allowed-contracts callback-contract)) ERR_CONTRACT_NOT_ALLOWED)
```

However, the `set-allowed-contract` function cannot be executed on the mainnet:

```
(asserts! (not is-in-mainnet) ERR_RESTRICTED_TO_TESTNET)
```

As a result, no allowed callback can be set after the deployment on the mainnet.

Recommendation

If Granite intends to dynamically add allowed contracts, it should modify the `set-allowed-contract` function to either operate on the testnet or be callable by the governance contract. In the `governance-v1` contract, a new action should be added to set the allowed callback contract for the flash loan.

Note: The team plans to introduce direct setters for allowed contracts in the production version (e.g. `(map-set allowed-contracts Liquidator true)`) and to remove the restriction in the future.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[QA-03] LP Incentives Contract Snapshot Uploader Cannot Be Changed

Description

The `lp-incentives-v2` contract has a privileged principal known as the `snapshot-uploader`, which is responsible for managing all snapshot-related actions within the contract.

Currently, this address cannot be modified. Although the contract's role is limited to a brief period of use, it might be beneficial to have the ability to transfer this role to a different principal if necessary.

Recommendation

Implement a function to transfer the snapshot uploader role. If this is not feasible, consider setting the uploader as a constant rather than a variable.

[QA-04] LP Incentives Contract Optimization

Description

In the `lp-incentives-v2` , several private functions utilize a redundant `begin` block to assess a statement and then return `(ok true)` if no failure occurs.

Example for `ensure-epoch-initialized` :

```
(define-private (ensure-epoch-initialized)
  (begin
    (asserts! (get epoch-initiated
      (var-get epoch-details)) ERR-EPOCH-NOT-INITIALIZED)
    SUCCESS
))
```

This function can be rewritten to eliminate the need for a `begin` block, thereby reducing execution fees.

```
(define-private (ensure-epoch-initialized)
  (ok (asserts! (get epoch-initiated
    (var-get epoch-details)) ERR-EPOCH-NOT-INITIALIZED))
)
```

Recommendation

Apply the aforementioned pattern to the `ensure-snapshot-uploader` , `ensure-epoch-uninitialized` , `ensure-epoch-initialized` , `ensure-epoch-closed` , and `ensure-epoch-not-closed` functions.

[QA-05] Post Safety Module Wipe Considerations

Description

When bad debt is socialized, the first market participants to have their funds slashed are the stakers from the LP Staking module, `staking-v1`.

With the new change, if the entire position is wiped, which can occur during a black swan event, the staking contract enters a wiped out state (indicated by `staking-wiped-out` being set), and the following features are permanently disabled:

- Staking via `stake`
- Initiating unstaking via `initiate-unstake`
- Finalizing unstaking via `finalize-unstake`

However, despite these disabled features, other actions remain permissible:

- Users can still transfer the staking contract shares, which may expose them to MEV and arbitrage risks, as external sources might experience a slight delay in adjusting the share price to 0 (since it is no longer backed).
- If governance mistakenly calls `reconcile-lp-token-balance` after a direct LP token transfer, the staking contract behaves as if staking occurred and begins to accrue interest, which can never be withdrawn. This would act as a buffer for socializing bad debt while depriving depositors of interest.

Recommendation

Clearly document and inform stakers that, after a wipe, their staking share tokens will have no intrinsic value. Blocking transfers of Granite Staked LP Tokens is not recommended, as external integrators may encounter issues transferring out the tokens. Add a check in `staking-v1::reconcile-lp-token-balance` to prevent it from being called if the staking contract has been wiped out.

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[QA-06] Improvements Suggested for the `liquidator-v1` Contract

Description

The `liquidator-v1` contract can be enhanced by implementing a few changes that would either reduce execution fees or improve the contract's overall consistency and utility:

1. In the `get-liquidation-data` function, instead of returning a tuple with a single element named `liquidation-info`, which contains the tuple information from the `get-liquidation-info` function call, directly return the contents of the `get-liquidation-info.liquidation-info` tuple element.
2. In the `liquidate` function, the `repay-amount-without-discount` is only returned and not used. This value is never utilized in subsequent calls and is already available in the `repayment-info`, which is also returned by the same function. This results in the `repay-amount-without-discount` element being redundantly included twice. Remove it from the `liquidate` return tuple.
3. In the `ensure-non-zero-repay-amount` function, there is a typo in the comment. The word `dont` should be corrected to `don't`.

Recommendation

Implement the suggested changes.

[QA-07] Scaling Factor Ambiguities

Description

Throughout the codebase, a `SCALING-FACTOR` variable is used in two distinct ways.

The first form is as follows:

```
(define-constant SCALING-FACTOR
  (contract-call? .constants-v1 get-scaling-factor))
```

Here, `constants-v1::get-scaling-factor` returns `u100000000` (`10^8`). This form is utilized in the `math-v1`, `state-v1`, `borrower-v1`, and `liquidator-v1` contracts.

The second form is:

```
(define-constant scaling-factor (pow u10
  (contract-call? .constants-v1 get-market-token-decimals)))
```

This form is used in the `flash-loan-v1`, `lp-incentives-v2`, and `staking-v1` contracts.

In these contracts, the semantics of the `SCALING-FACTOR` can vary up to three times within the same contract, leading to confusion and increasing the likelihood of long-term issues.

We will elaborate on how `SCALING-FACTOR` is used in each case and suggest ways to improve comprehension.

To begin, the second form is incorrect as it is used as a percentage scaling that depends on market decimals. This has caused issues in each of the mentioned contracts, which have been addressed separately in this finding.

In examining all contracts using the first form, we identify three interpretations:

1. Price Decimal Scaling
2. Percentage Scaling
3. Position Health Ratio

Price Decimal Scaling

The Pyth oracle adapter has a hardcoded 8-decimal conversion value, which mandates that all price operations must be adjusted by the same multiple of a full unit.

Since the hardcoded decimal value is 8, the original `SCALING-FACTOR` poses no issue, but there are constraints to consider.

In `math-v1` , the `scaling-factor` must be a full unit with the same decimals as the price decimals since we are calculating market asset value.

In the `borrower-v1` contract, the `SCALING-FACTOR` has a dual meaning, with the decimal price scaling interpretation being used to determine collateral value.

In the `liquidator-v1` contract, the `SCALING-FACTOR` has a triple meaning, one of which is the price scale decimal interpretation when evaluating collateral.

Across these three contracts, the `SCALING-FACTOR` can be separated into a different variable, `PRICE-SCALING-FACTOR` , while keeping it synchronized with the `pyth-adapter-v1` price decimal value.

Percentage Scaling

`SCALING-FACTOR` is also used to represent percentages.

In the `state-v1` contract, `SCALING-FACTOR` is used as a percentage precision when calculating liquidation premium (with validations). Additionally, LTV values are passed off-chain with `SCALING-FACTOR` interpreted as `100%` . Protocol reserves percentages are also relative to the scaling factor.

In the `borrower-v1` contract, the second `SCALING-FACTOR` meaning is percentage representation. The LTV percentage valuation uses the factor as a 100% equivalent.

In the `liquidator-v1` contract, the second `SCALING-FACTOR` interpretation is also percentage representation. There are four clusters, L370-L374, L393- L394, L451, and L461, where this interpretation is used.

A critical constraint in all mentioned locations and across all mentioned contracts is that they must all share the same value.

Position Health Ratio

The `liquidator-v1` contract also uses `SCALING-FACTOR` as a ratio/percentage, but separately and only related to position health. While all other shared percentages between state, liquidator, and borrower must be identical, in `liquidator-v1` , the ratio for health check usage can have its own separate value and still be fully safe (e.g., `MINIMUM_HEALTH_RATIO`).

There are three locations where the scaling factor is used as a health ratio: L91, L322, and L424.

Having all the above interpretations and roles for the same value can cause issues with future development due to developer confusion.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

Recommendation

Create and use a different scaling factor for when using price decimal scaling or liquidation health factor scaling.

For example, you can create a new constant-v2 contract that takes all the previously existing values from the constant-v1 contract and adds a getter for the price decimals and price scaled factor (based on price decimals).

In the pyth adaptor contract get the price decimal value and use it when converting pyth feed prices.

Replace every instance in the code where scaling factor was used as a price scaling factor with the newly created constant.

In the liquidator contract, create a separate ratio for position health, e.g. `MINIMUM_HEALTH_RATIO` and use it with positions that are corresponding.

[QA-08] Withdrawal Caps Contract Can Be Slightly Improved

Description

In the `withdrawal-caps-v1` contract, there are several modifications that could reduce execution fees or enhance the contract's overall uniformity and utility:

1. In all the sync functions: `sync-lp-bucket` , `sync-debt-bucket` , and `sync-collateral-bucket` , the current value is retrieved both in the `current-bucket` variable and again when emitting the `old-*` value in the final `print` function calls. Reuse the `current-bucket` in all three instances instead of retrieving it again.
2. The contract title is `daily-caps-module` , although the contract name is `withdrawal-caps` , which is a remnant of the previous contract name.Remove the title itself to allow the first row to be the license.
3. At line 37, there is a typo in the word `collateal` . Change it to `collateral` .
4. The governance call check, which is duplicated in each setter, can be moved into its own function, e.g., `caller-is-governance` , and reused. This will reduce both runtime and read length execution costs.

Recommendation

Implement the suggested changes.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[QA-09] Withdrawal Caps Are Not Validated to Remain Below 100%

Description

When governance sets the withdrawal caps, they provide a new factor value to the `withdrawal-caps-v1` contract, where it is directly stored.

Each factor represents a percentage of the available amounts to be withdrawn (total liquidity/borrowable liquidity or total collateral).

As a percentage, it should not be allowed to exceed 100% of the amount.

Implementing this check would enhance code robustness and consistency, as exceeding 100% would effectively behave as if 100% is chosen, since users cannot withdraw more than what is available.

Recommendation

In each of the cap setter functions of the `withdrawal-caps-v1` contract—`set-lp-cap`, `set-debt-cap`, and `set-collateral-cap`—ensure that the `new-cap` parameter does not exceed the `SCALING-FACTOR`.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[QA-10] Detach Withdrawal Caps Scaling Factor From Constants to Avoid Future Ambiguity

Description

In the `withdrawal-caps-v1` contract, the `SCALING-FACTOR` is currently set to the global scaling factor from the constants.

```
(define-constant SCALING-FACTOR
  (contract-call? .constants-v2 get-scaling-factor))
```

In previous versions of the codebase, this approach has led to confusion. Using the same scaling factor for multiple components is only constrained by its application (for example, the state and liquidation contracts must share the same scaling factor). However, the withdrawal caps module can independently have a different value.

Recommendation

To prevent future confusion, declare the `SCALING-FACTOR` constant directly as `10^8` within the `withdrawal-caps-v1` contract.

[QA-11] Governance Contract Can Be Slightly Improved

Description

In the `governance-v1` contract, there are a few modifications that could enhance fee execution and improve the overall uniformity and utility of the contract:

1. Simplification of `approve` and `deny` functions

When approving or denying a proposal through the `approve` / `deny` functions, the current proposal from the `governance-proposal` map is only updated to increase the approve or deny count, respectively. All other proposal parameters remain unchanged.

Currently, the entire `governance-proposal` map is updated element by element in both cases.

```
(map-set governance-proposal proposal-id {
  action: (get action proposal),
  approve-count: (+ (get approve-count proposal) u1),
  deny-count: (get deny-count proposal),
  expires-at: (get expires-at proposal),
  closed: (get closed proposal),
  executed: (get executed proposal),
  execute-at: (get execute-at proposal)
})
```

This approach is inefficient and can be streamlined by utilizing the `merge` Clarity system function, as shown below:

```
(map-set governance-proposal proposal-id (merge proposal { approve-count: (+
  (get approve-count proposal) u1) }))
```

2. Typographical error in `execute-if-approve-threshold-met`

In the `execute-if-approve-threshold-met` function, there is a typographical error in the comment `;; proposal will excuted after time-lock`. The word `excuted` should be corrected to `executed`.

Recommendation

Implement the suggested changes to enhance code readability and improve uniformity.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[QA-12] Overlapping Error Code Ranges

Description

In the codebase, each contract should have a distinct error code range to easily identify the originating contract of an error.

Currently, the `staking-reward-v1` and `withdrawal-caps-v1` contracts have overlapping error code ranges at `u90000`.

Such overlaps can lead to confusion when debugging failed transactions.

Recommendation

Adjust the error codes for either the `staking-reward-v1` or `withdrawal-caps-v1` contract to `u120000`, which is the next available range.

[QA-13] Remove Outdated Bad Debt Comment

Description

In the `liquidator-v1::is-bad-debt` function, there is a comment suggesting that only a debt liquidator can liquidate bad debt:

```
;;
   if so, ensure if the liquidator is a bad debt liquidator else do not allow liquidation
```

This feature has not been implemented, making the comment outdated and misleading.

Recommendation

Remove the comment on line L484 from the `liquidator-v1` contract.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[QA-14] Remove Unused Let Variable Declarations

Description

The `get-liquidate-params` function in the `liquidator-v1` contract contains several unused `let` variable declarations that can be removed to reduce execution costs.

Specifically, the variables `current-debt-adjusted`, `total-liquid-ltv` `liquidation-discount`, `collateral-liquid-ltv`, and `collateral-decimals` are not utilized.

Recommendation

Remove the `let` declarations for the five mentioned variables from the `get-liquidate-params::liquidator-v1`.

CONTENTS

ClarityAlliance
Security Review

Granite
(Upgrade v2)

[QA-15] Ambiguous Reversion on Repayment When Borrower Has No Debt

Description

When repaying a loan through `borrower-v1::repay`, if a caller attempts to repay on behalf of a previous user of the protocol who currently has no outstanding debt, the transaction correctly reverts. However, it does so ambiguously due to a division by zero error.

This division by zero occurs during the calculation of the interest portion, as the `current-debt` variable is zero.

```
(interest-portion
  (contract-call? .math-v1 calculate-interest-portions current-debt borrowed-amount repay-amou
```

Any external integrator who mistakenly calls the repay function for a user without debt will find it challenging to identify the cause of the error.

Recommendation

In the `borrower-v1::repay` function, add a check for existing debt to exit early or revert with `ERR-NO-DEBT` if no debt is present.

```
(current-debt (get current-debt repay-info))
+   (debt-check (asserts! (> current-debt u0) ERR-NO-DEBT))
    (interest-portion
      (contract-call? .math-v1 calculate-interest-portions current-debt borrowed-amount repay-
```