ClarityAlliance

ZEST PROTOCOL SECURITY REVIEW

Conducted by:
KRISTIAN APOSTOLOV

MAY 2ND, 2024

CONTENTS

1. About Clarity Alliance

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

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

Learn more about Clarity Alliance at clarityalliance.org.

CONTENTS

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

3. Introduction

A time-boxed security review of Zest Protocol, where Clarity Alliance reviewed the scope, whilst simultaneously building out a testing suite for the protocol.

4. About Zest Protocol

Zest Protocol is a decentralized lending platform on Stacks that enables users to trustlessly lend and borrow assets.

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.

CONTENTS

5.2 Likelihood

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

5.3 Action required for severity levels

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

6. Security Assessment Summary

Review Commit Hash:
523c546ea944a2bcca4e038123df1086c6b9f02a

CONTENTS

7. Executive Summary

Over the course of the security review, Kristian Apostolov engaged with Zest Protocol to review Zest Protocol. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Zest Protocol
Repository	https://github.com/Zest-Protocol/zest-contracts
Date	May 2nd, 2024
Protocol Type	Lending Protocol

Findings Count

Severity	Amount
High	5
Medium	3
Low	3
QA	1
Total Findings	**12**

CONTENTS

Summary of Findings

ID	Title	Severity	Status
[H-01]	get-reserve-available-liquidity underflows on accrued-to-treasury > ft-balance	High	Resolved
[H-02]	Not calling cumulate-balance before minting to collection address	High	Resolved
[H-03]	Conditional liquidation bonus miscomputation	High	Resolved
[H-04]	Interest rate calculations will break core functionality after Nakamoto hardfork	High	Acknowledged
[M-01]	get-stx-per-ststx can be manipulated	Medium	Resolved
[M-02]	Not calling cumulate-balance for repaid asset leads to lower leveraged returns	Medium	Acknowledged
[M-03]	Calling cumulate-balance after updating interest rates in supply causes causes lower asset returns	Medium	Resolved
[L-01]	Not checking last-block for staleness	Low	Resolved
[L-02]	flashloan not lending out accrued-to-treasury amount	Low	Resolved
[L-03]	Cannot liquidate yourself and receive receipt tokens	Low	Resolved
[QA-01]	Use get-price decimals with to-fixed instead of magic numbers	QA	Resolved

CONTENTS

8. Findings

8.1. High Findings

[H-01] `get-reserve-available-liquidity` underflows on `accrued-to-treasury > ft-balance`

Description

The `get-reserve-available-liquidity` function calculates the current liquid reserve of a given token minus the treasury allocation amount, which is the protocol's share of interest. The function uses the following formula:

`(- (try! (contract-call? asset get-balance (get-reserve-vault asset))) accrued-to-treasury)`

Since `accrued-to-treasury` is a receipt token amount transferred to the protocol's collection address via `mint-to-treasury`, the conidtion `accrued-to-treasury <= (.asset[x] get-balance reserve)` must hold to prevent underflow errors. However, during the `repay` process, `accrued-to-treasury` is incremented before the repayment amount is sent to the reserve, potentially violating this condition and causing a transaction to revert with an underflow error.

A PoC demonstrating how this state can be reached is available here.

Impact

Users who encounter this issue may be unable to repay their outstanding debt, leading to forced interest payments and potential liquidation.

Recommendation

Implement an overloaded `get-reserve-available-liquidity` function with a `liquidity-added` parameter, which should be included in the available liquid balance before it is sent to the reserve:

```
(- (+ (try! (contract-call? asset get-balance
  (get-reserve-vault asset))) liquidity-added) accrued-to-treasury)`
```

CONTENTS

[H-02] Not calling `cumulate-balance` before minting to collection address

Description

The `pool-borrow-v-2.supply` function calls `cumulate-balance` before updating the user's balance. This function accrues any interest the user may have earned since their last collateral supply and mints it as receipt tokens. The protocol collects fees as a percentage of the interest generated through borrowing, paid in receipt tokens and accumulated in `accrued-to-treasury` within the asset `reserve-state`. To claim these accrued tokens, the protocol must call `mint-to-treasury`, which mints the tokens to the collection address. The issue is that `mint-to-treasury` does not call `cumulate-balance` before minting the tokens, unlike the previous `zToken` implementation, which included this step in its `mint` function.

V1 reference:

```
lp-sbtc-v1.clar

(define-public (mint (amount uint) (recipient principal))
  (begin
     (try! (is-approved-contract contract-caller))
     (let (
          (ret (try!
         (cumulate-balance-internal recipient))) ;; @audit <--- cumulating
balance befo
     )
          (mint-internal amount recipient)
     )
  )
)
```

Impact

Collateral held by the collection address may accrue significantly higher interest rates because its index and timestamp are only updated during manual `supply` and `withdraw` operations. When a manual collateral mutation occurs, the interest rate for the time delta is applied to the entire amount as if it had been present for the entire period.

Recommendation

Ensure `cumulate-balance` is called before `mint` in `mint-to-treasury`.

[H-03] Conditional liquidation bonus miscomputation

Description

In the `liquidation-manager-v1-1` contract, the function `calculate-available-collateral-to-liquidate` determines the collateral a liquidator will receive and the debt required. It calculates the `original-collateral-purchasing-power` as the collateral amount that could be purchased with the provided debt and then computes `max-collateral-amount-from-debt` by adding a liquidation bonus.

```
original_collateral_purchasing_power = (
debt_to_liquidate * collateral_price / debt_currency_price
)
* (10^collateral_decimals / 10^principal_decimals)
```

```
max_collateral_amount_from_debt = original_collateral_purchasing_power * (
1 + liquidation_bonus
)
```

When `max-collateral-amount-from-debt` is less than or equal to `user-collateral-balance`, the calculations are correct. However, when `max-collateral-amount-from-debt` exceeds `user-collateral-balance`, both `debt-needed` and `liquidation-bonus-collateral` are miscalculated. The `debt-needed` should be the amount necessary to purchase the user-collateral balance adjusted for the liquidation bonus, but it is incorrectly calculated as the amount required to buy the `user-collateral-balance` reduced by the liquidation bonus percentage. Similarly, the `liquidation-bonus-collateral` is inaccurately computed.

Impact

These miscalculations can lead to incorrect liquidation outcomes, causing potential financial discrepancies in the protocol.

Recommendation

Correct the logic to ensure that in cases where `max-collateral-amount-from-debt` exceeds `user-collateral-balance`, the `debt-needed` and `liquidation-bonus-collateral` are accurately calculated based on the full liquidation bonus.

CONTENTS

[H-04] Interest rate calculations will break core functionality after Nakamoto hardfork

Description

Currently, several variables that track a user's position and health are dependent on `burn-block-height`. For instance, compounded interest is calculated based on the difference between `burn-block-height` and `last-updated-block-reserve`. This dependency poses a problem because it assumes a 1:1 linkage between the `burn-block-height` —the current height of the underlying Bitcoin chain—and Stacks block production. With the upcoming Nakamoto upgrade, Stacks block production will be decoupled from Bitcoin, potentially causing core functionality to break.

Impact

This issue could enable multi-block, interest-free loans within Stacks blocks that are ordered within the same tenure.

Recommendation

Replace the use of `burn-block-height` for functions intended to be linked to Stacks block production with `stacks-block-height` or `block-height`. Functions that do not rely on Stacks block production and require consistent time measurement should continue using burn-block-height, as Bitcoin block production remains consistently around 600 seconds.

CONTENTS

8.2. Medium Findings

[M-01] `get-stx-per-ststx` can be manipulated

Description

The `stSTX` oracle determines the current price through the following steps:

1. Querying the current `STX/USD` rate.
2. Retrieving the current `stSTX/STX` rate directly from the `reserve-1` contract.
3. Multiplying the STX price by the `stSTX/STX` rate to obtain the `stSTX/USD` price.

This methodology depends directly on the current ratio of STX and `stSTX` in `stacking-dao-core-v1`. The function `stacking-dao-core-v1.get-stx-per-ststx` invokes `reserve-1.get-total-stx`, which reads the STX balance of the contract directly. This direct balance approach allows STX donations to immediately alter the rate, thereby influencing the price of `stSTX` on Zest. Typically, such donations are absorbed through MEV and don't affect the token's price on oracles. However, a donation attack can instantly impact the `stSTX/STX` rate used on Zest, potentially leading to exploit scenarios.

Recommendation

Utilize `arkadiko-oracle-v2-3`, which already supports `stSTX`, to mitigate this issue.

CONTENTS

[M-02] Not calling `cumulate-balance` for repaid asset leads to lower leveraged returns

Description

Users can increase their collateral leverage by supplying an asset, borrowing it, and then supplying it again, repeating this process until they reach their desired leverage level. This strategy enhances collateral effectiveness but also raises the risk of liquidation. When a leveraged user wishes to deleverage, they must repeatedly repay and withdraw excess collateral to achieve their target leverage level. However, the `repay` function does not call `cumulate-balance` for the borrowed asset before repayment.

Impact

Failing to cumulate the borrowed asset before recalculating interest rates results in lower accrued interest for the leveraged balance, as interest rates are based on the `total loan size/total collateral provided` ratio. This makes leveraging less profitable and, in some cases, even unprofitable, depending on how the initial repayment affects the interest rate.

Recommendation

Consider adding a `cumulate-balance` call to `pool-borrow-v1-2.repay` before the `update-state-on-repay` function:

```
(try! (contract-call? lp cumulate-balance on-behalf-of))
;; *update-state-on-repay call*
```

[M-03] Calling `cumulate-balance` after updating interest rates in supply causes causes lower asset returns

Description

The `pool-borrow-v1-2.supply` function updates the current borrowing indexes and then calls `cumulate-balance` for the asset being supplied before minting receipt tokens to the user and transferring their collateral to the reserve. Interest rates are based on the asset's loan utilization ratio, which increases with higher utilization and decreases with lower utilization.

```
(try!
(contract-call? .pool-0-reserve update-state-on-deposit asset owner amount (is-eq cu
(try!
(contract-call? lp cumulate-balance owner)) ;; @audit <--- cumulating balance after
(try!
(contract-call? lp mint amount owner)) ;; @audit <--- minting receipt tokens
(try!
(contract-call? .pool-0-reserve transfer-to-reserve asset owner amount)) ;; @audit <
```

The issue arises because `cumulate-balance` is called after `update-state-on-deposit`, which includes interest rate recalculation.

Impact

The interest for the user's initially supplied asset amount will be calculated at the now-lowered rate, as if the utilization was lower before the `supply` call.

Recommendation

Restructure the order of the functions in `pool-borrow-v1-2.supply` to call `cumulate-balance` before `update-state-on-deposit`.

```
(try!
(contract-call? lp cumulate-balance owner)) ;; @audit <--- cumulating balance *befor
(try!
(contract-call? .pool-0-reserve update-state-on-deposit asset owner amount (is-eq cu
(try!
(contract-call? lp mint amount owner)) ;; @audit <--- minting receipt tokens
(try!
(contract-call? .pool-0-reserve transfer-to-reserve asset owner amount)) ;; @audit <
```

CONTENTS

8.3. Low Findings

[L-01] Not checking `last-block` for staleness

Description

Arkadiko Oracle's `get-price` function returns a tuple with three values: `(tuple (decimals x) (last-block y) (last-price z))` `last-block` represents the last `block-height` at which the price was updated.

The issue arises because the `last-price` is used across token-specific contracts in the protocol without any safeguards against price staleness.

Impact

Data mutations could be executed using stale and inaccurate prices.

Recommendation

Implement a check to ensure that `last-block` is within the current `block-height` plus an acceptable staleness period.

CONTENTS

[L-02] flashloan not lending out `accrued-to-treasury` amount

Description

The `flashloan` function allows users to take out a flash loan in an enabled asset, which they must repay along with a flashloan fee in the callback they receive. The fee is a small fraction of the flash-lent amount. However, the amount available for lending is determined by `get-reserve-available-liquidity`, which subtracts the fees accrued to the protocol's treasury (`accrued-to-treasury`). Although this amount is owned by the protocol, it is part of the available liquidity and should be eligible for flash borrowing.

Impact

The protocol is missing out on additional, albeit small, amounts of yield.

Recommendation

Consider modifying the following code:

```
(available-liquidity-before (try! (contract-call? .pool-0-reserve get-reserve-available-liquidity asset)))
```

to this implementation, which returns the liquid amount of tokens directly:

```
(available-liquidity-before (try! (contract-call? asset get-balance (get-reserve-vault asset)))
```

CONTENTS

[L-03] Cannot liquidate yourself and receive receipt tokens

Description

When a user calls `liquidation-call` on an unhealthy position, they can choose to receive either the underlying collateral or the receipt token representation. The issue arises with the `to-receive-atoken` option, where the receipt transfer function first performs a `SIP-10` `ft-transfer?` from the liquidated user to the liquidator. This is problematic because one of the requirements for `ft-transfer?` is that `recipient != sender` .

Impact

Due to this requirement, if a user attempts to self-liquidate, they will encounter an unexpected revert with error `u2`, limiting them to using `to-receive-atoken = false` when self-liquidating.

Recommendation

Consider bypassing the `SIP-10` `ft-transfer?` step when `recipient == sender` and only perform cumulate-balance:

```
(define-private (execute-transfer-internal
(amount uint)
(sender principal)
(recipient principal)
)
(let (
(from-ret (try! (cumulate-balance-internal sender)))
(to-ret (try! (cumulate-balance-internal recipient)))
)
(if (not
(is-eq sender recipient)) ;; @audit only call transfer if recipient != sender
(try! (transfer-internal amount sender recipient none))
true
)
;; ...
```

CONTENTS

8.4. QA Findings

[QA-01] Use `get-price decimals` with to-fixed instead of magic numbers

Description

Use decimals from the `get-price` tuple and `to-fixed` instead of the current implementation.