ChainLight

VIBE

Vibe V/VXD/Gateway Security Assessment

: Vibe Smart Contracts Security Assessment Final Report

January 21, 2026

Revision 1.0

ChainLight@Theori

Table of Contents

Executive Summary

Theori's ChainLight team conducted a security assessment of Vibe smart contracts beginning December 8, 2025. Although the initial audit was completed within three days, an extended period was dedicated to the remediation and verification process to ensure all identified vulnerabilities were addressed.

The ecosystem utilizes two primary tokens: V, the native token of Vibe, and VXD, a reward token issued based on the value of V. While minting of VXD is typically managed by the Vibe team, the process is permissionless. Notably, the VXD mechanism may expose VXD redeemers to financial discrepancies or losses due to price fluctuations of the V token after the time of redemption request.

The primary objective of this audit was to determine if an external attacker could compromise the V/VXD tokens or exploit the VXD Gateway logic. While the stability of the Time-Weighted Average Price (TWAP) and the security of owner-controlled operations were not the primary focus, the team evaluated operational risks within the engagement's time constraints.

The assessment identified 16 issues, including a High-severity vulnerability where manipulation of the Redeem Queue could prevent other users from converting VXD back to V tokens. In addition to technical vulnerabilities, this report includes several design-level recommendations to improve the long-term stability of the protocol.

Note on VXD Design: VXD serves as a revenue-sharing mechanism for creators using the platforms integrated V and VXD. When revenue is generated (e.g., via credit card payments), a portion is allocated to creators in VXD after platform fees. When `$100` worth of VXD is distributed, while the market value is not strictly guaranteed at `$100`, the system is designed to maintain a minimum floor value of approximately `$50`.

Audit Overview

Scope

Name	Vibe V/VXD/Gateway Security Assessment
Target / Version	• Git Repository (`vibe-vxd/vxd`): ◦ commit `8e59af30aa7c78e59402492ea2a4cf109678c667` • Note: While the commit hashes have changed following the migration to a public repository, we have checked that the source code remains identical to the previous version.
Application Type	Smart Contracts
Lang. / Platforms	Smart contracts [Solidity]

Code Revision

- Patch PRs:
  - PR `#4` (`7d077eadf6079bea24c499367c19d194d2c3b5f1`)
  - PR `#5` (`1cdc7b7bb48350eeb107cbbb6de9ad8a98359ba3`)
  - PR `#6` (`0f523e268adeddd0ea0865051d1e3747a4809c1b`)

Severity Categories

Severity	Description
Critical	The attack cost is low (not requiring much time or effort to succeed in the actual attack), and the vulnerability causes a high-impact issue. (e.g., Effect on service availability, Attacker taking financial gain)
High	An attacker can succeed in an attack which clearly causes problems in the service's operation. Even when the attack cost is high, the severity of the issue is considered "high" if the impact of the attack is remarkably high.
Medium	An attacker may perform an unintended action in the service, and the action may impact service operation. However, there are some restrictions for the actual attack to succeed.
Low	An attacker can perform an unintended action in the service, but the action does not cause significant impact or the success rate of the attack is remarkably low.
Informational	Any informational findings that do not directly impact the user or the protocol.
Note	Neutral information about the target that is not directly related to the project's safety and security.

Status Categories

Status	Description
Reported	ChainLight reported the issue to the client.
WIP	The client is working on the patch.
Patched	The client fully resolved the issue by patching the root cause.
Mitigated	The client resolved the issue by reducing the risk to an acceptable level by introducing mitigations.
Acknowledged	The client acknowledged the potential risk, but they will resolve it later.
Won't Fix	The client acknowledged the potential risk, but they decided to accept the risk.

Finding Breakdown by Severity

Category	Count	Findings
Critical	**0**	• N/A
High	**3**	• `VIBE-001` • `VIBE-012` • `VIBE-013`
Medium	**2**	• `VIBE-002` • `VIBE-003`
Low	**4**	• `VIBE-004` • `VIBE-005` • `VIBE-006` • `VIBE-014`
Informational	**4**	• `VIBE-007` • `VIBE-008` • `VIBE-015` • `VIBE-016`
Note	**0**	• N/A

Findings

Summary

#	ID	Title	Severity	Status
1	VIBE-001	Denial of Service (DoS) in VXDContract (Gateway) via Redemption Queue Manipulation Leading to Block Gas Limit Exhaustion	High	Patched
2	VIBE-002	Gas Exhaustion and Denial of Service (DoS) in VXDContract (Gateway) via Accumulated Failed Redemptions	Medium	Patched
3	VIBE-003	Collateral Value Manipulation in VXDContract via Donation Attack Leading to Financial Loss	Medium	Patched
4	VIBE-004	Excessive VXD Token Burning in `_processRedeemSingle` due to Insufficient Validation	Low	Patched
5	VIBE-005	Discrepancy in VGovernance Withdrawal Timing due to Dynamic `unlockPeriod` Updates	Low	Won't Fix
6	VIBE-006	Inconsistency Between Upgradeable Intent and Static Implementation in TokenV and TokenVXD	Low	Patched
7	VIBE-007	Minor Suggestions - Code Optimization and Maintenance Improvements	Informational	Patched
8	VIBE-008	Structural Design Flaws and Architectural Recommendations for VGovernance and VXDContract	Informational	Acknowledged
9	VIBE-012	Initial Supply Inflation via Redundant Multichain Minting in TokenVOFT	High	Patched

#	ID	Title	Severity	Status
10	`VIBE-013`	Redundant Inflation Minting in Multichain TokenVOFT	**High**	**Patched**
11	`VIBE-014`	Logic Inconsistency and Redundant Exception Handling in `emergencyCancelRedeems`	**Low**	**Patched**
12	`VIBE-015`	Index Update Failure for Failed Redemption Requests in `processRedeem`	**Informational**	**Patched**
13	`VIBE-016`	Redundant State Validation in `processRedeem`	**Informational**	**Patched**

#1 `VIBE-001` Denial of Service (DoS) in VXDContract (Gateway) via Redemption Queue Manipulation Leading to Block Gas Limit Exhaustion

ID	Summary	Severity
`VIBE-001`	The VXDContract uses a single global queue that allows attackers to create a large volume of canceled requests, inflating the gas cost of processing redemptions. This results in a Denial of Service (DoS) that prevents legitimate users from withdrawing their funds.	**High**

Description

The vulnerability comes from managing all redemption requests within a single shared queue. An attacker can exploit this by repeatedly calling `addRedeemQueue` to generate thousands of entries (e.g., 24,000) and then invoking `cancelRedeem` before the `unlockTime`.

When a user calls `processRedeem`, the contract must iterate through these entries to check the `request.isCanceled` status. Assuming a cost of approximately 2,500 gas per iteration, processing 24,000 canceled requests requires roughly 60 million gas. Because this exceeds the Block Gas Limit of most blockchain networks, the transaction will consistently fail, halting the redemption process for all users.

Impact

High

An attacker can permanently block the `processRedeem` function by intentionally inflating gas requirements. This results in the total loss of access to user assets, as legitimate withdrawal requests can no longer be processed.

Recommendation

Redesign the contract logic to replace the single global queue with individual, per-user redemption queues. This ensures that the gas cost for one user's transactions remains independent of others' actions, preventing cross-user DoS attacks.

Remediation

Patched

The logic was updated to use a `loopCount` variable instead of relying on `processedCount`. By comparing `loopCount` against a defined `maxCount`, the contract can now limit the number of iterations per transaction, preventing gas limit exhaustion.

Commit: `93cf11a25fb184008928194c945c2e5ad665d193`

#2 `VIBE-002` Gas Exhaustion and Denial of Service (DoS) in VXDContract (Gateway) via Accumulated Failed Redemptions

ID	Summary	Severity
`VIBE-002`	Failed redemption requests that are not manually canceled remain in the global queue, forcing the system to re-process them during every subsequent execution. This accumulation leads to escalating gas costs and can eventually cause a Denial of Service (DoS) by exceeding the block gas limit.	Medium

Description

In the VXDContract, if a redemption request fails (e.g., due to a user being on a USDC blacklist), the `isFailed` state is set to true. However, the `latestProcessedRedeemIndex` does not advance past these failed entries. Unless a user explicitly calls `cancelRedeem` to set `isCanceled` to true, the `processRedeem` function must iterate through every accumulated failed request during each call. This redundant processing results in a linear increase in gas consumption as failed requests remain in the queue.

Impact

Medium

The vulnerability leads to significant gas waste for the protocol and its users. If the number of failed requests grows sufficiently large, the gas required to iterate through them will exceed the block gas limit, rendering the `processRedeem` function unusable and permanently blocking all subsequent asset withdrawals.

Recommendation

Restructure the contract to manage redemption requests in individual, per-user queues instead of a single global queue. This ensures that failed or stalled requests from one user do not affect the gas costs or transaction success of others.

Remediation

Patched

Commit: `93cf11a25fb184008928194c945c2e5ad665d193`

#3 `VIBE-003` Collateral Value Manipulation in VXDContract via Donation Attack Leading to Financial Loss

ID	Summary	Severity
`VIBE-003`	The VXDContract determines token redemption amounts based on real-time asset balances, which can be artificially manipulated by an attacker donating tokens directly to the contract. This "Donation Attack" distorts internal exchange rates, potentially resulting in financial losses for users or the depletion of protocol collateral.	Medium

Description

The vulnerability resides in the `calculateRedeemVAmount` function, which calculates the amount of V tokens to be redeemed by selecting the minimum value between a `proportionalV` and a `twapBased` calculation. The calculation for `proportionalV` relies on the contract's current asset balances (`contractUSDCBalance` and `contractVBalance`) obtained directly via the `balanceOf` function.

An attacker can exploit this by sending assets directly to the contract address without using the protocol's functions:

- USDC Donation: Increasing `contractUSDCBalance` decreases the resulting `proportionalV`. Consequently, users who redeem their VXD receive fewer V tokens than they are entitled to.
- V Token Donation: Increasing `contractVBalance` increases the `proportionalV`. This allows users to receive more V tokens than the protocol's economic model intended, leading to an unauthorized drain of collateral.

Impact

Medium

The ability to manipulate internal exchange rates through external asset injection compromises the protocol's integrity. Depending on the direction of the manipulation, it leads to either direct financial loss for users or the potential insolvency of the protocol's collateral pool.

Recommendation

Avoid using the `balanceOf` function to determine exchange rates. Instead, implement internal accounting by maintaining dedicated global variables to track asset balances. These variables should only be updated through authorized state-changing functions, such as `mint` and `processRedeem`, to ensure that direct token transfers (donations) do not affect internal calculations.

Remediation

Patched

Commit: `351921aabe574d80038d2dfdd9e7841105c983f5`

#4 VIBE-004 Excessive VXD Token Burning in _processRedeemSingle due to Insufficient Validation

ID	Summary	Severity
VIBE-004	A logic error in the _processRedeemSingle function fails to prevent processed redemption amounts from exceeding the requested amounts. This results in the unintended burning of VXD tokens and financial loss to the protocol.	Low

Description

The _processRedeemSingle function manages the execution of redemption requests. However, it lacks strict validation to ensure that the cumulative request.processedAmount does not surpass the original request.amount. Under certain execution conditions, the contract may burn more tokenVXD than the user initially requested or the system intended. This discrepancy leads to inaccurate internal accounting and a permanent loss of protocol assets through unnecessary token destruction.

Impact

Low

This issue results in the excessive burning of VXD tokens, leading to financial loss and the corruption of protocol-level token metrics. While it does not facilitate direct theft of assets, it violates the intended economic logic of the redemption process.

Recommendation

Tighten the logic used to determine the completion of a redemption request. It is recommended to update the fullyProcessed condition from (request.processedAmount >= request.amount) to a strict equality check: fullyProcessed = (request.processedAmount == request.amount). This ensures that a request is only marked as complete when the processed total exactly matches the requested amount, preventing over-processing.

Remediation

Patched

Commit: 5717b6e91811810b0682165009ed4d1cbc5247da

#5 `VIBE-005` Discrepancy in VGovernance Withdrawal Timing due to Dynamic `unlockPeriod` Updates

ID	Summary	Severity
`VIBE-005`	In the VGovernance contract, changes to the global `unlockPeriod` after a user has initiated an unlock request lead to a mismatch between the expected withdrawal time and the actual time allowed by the system. This inconsistency arises because the contract evaluates the waiting period at the time of withdrawal rather than locking it during the initial request.	Low

Description

The vulnerability centers on how the VGovernance contract calculates the waiting time required between calling `unlock` and `withdraw`. Currently, the logic does not store a fixed withdrawal timestamp when a user triggers `unlock`. Instead, it references the global `unlockPeriod` state variable at the moment the user attempts to `withdraw`.

If an owner modifies `unlockPeriod` while a user is in the waiting phase, the user's required wait time changes dynamically. This causes the expected withdrawal time recorded in the initial event logs to become inaccurate, resulting in a discrepancy between the user's expectations and the contract's enforcement.

Impact

Low

This issue leads to operational inconsistency and user confusion regarding fund availability. While it does not represent a direct threat to the safety of assets, the lack of predictability in withdrawal times degrades the transparency and reliability of the protocol's governance and locking mechanisms.

Recommendation

Update the `UnlockInfo` struct to include a dedicated `withdrawTime` field. When the `unlock` function is invoked, the contract should calculate and permanently store the specific timestamp after which the withdrawal is permitted (e.g., `block.timestamp + unlockPeriod`). The `withdraw` function should then be modified to validate against this stored value using `require(block.timestamp >`

`info.withdrawTime)`, ensuring that changes to global settings do not retroactively affect pending requests.

Remediation

Won't Fix

The Vibe team has stated that the current behavior, applying the global configuration active at the time of withdrawal, is an intended design choice. Rather than altering the logic, the team has added detailed code comments to inform developers and users that the waiting period is subject to change based on global governance settings.

(Comment) Commit: `c681802a1266f596c0fab37aec314a56a3755191`

#6 VIBE-006 Inconsistency Between Upgradeable Intent and Static Implementation in TokenV and TokenVXD

ID	Summary	Severity
VIBE-006	While contract comments indicate plans for future upgrades and refactoring, the implementation lacks an upgradeable architecture. This discrepancy prevents seamless logic updates, requiring inefficient redeployment and data migration for future improvements.	Low

Description

The source code for `TokenV` and `TokenVXD` contains documentation suggesting the contracts are designed for future architectural changes and refactoring. However, the actual implementation does not utilize a proxy-based upgrade mechanism (e.g., Transparent or UUPS Proxy patterns). Consequently, the contract logic is immutable once deployed, which contradicts the stated intent in the code comments and limits the protocol's flexibility.

Impact

Low

The misalignment between documentation and implementation hinders maintenance. If bugs are discovered or feature expansions are required, the protocol cannot be upgraded in place; instead, developers must redeploy the VXDContract and perform complex state migrations, increasing operational overhead and risk.

Recommendation

Implement a standardized Proxy pattern for the `TokenV` and `TokenVXD` contracts to align the implementation with the intended maintenance goals. This will allow for logic updates while preserving the contract state and address.

Remediation

Patched

The client addressed the requirement by implementing the Layer Zero's Omnichain Fungible Token (OFT) standard. During this transition and the associated refactoring, the following related issues were identified and resolved (ref: PR #4):

- VIBE-012 (High): Resolved duplicate issuance of initial supply during multichain deployment.
- VIBE-013 (High): Resolved duplicate inflation minting during multichain deployment.
- VIBE-014: Corrected improper exception handling in the `emergencyCancelRedeems` function.
- VIBE-015 & VIBE-016: Improved `processRedeem` by ensuring index updates for failed requests and removing redundant checks.

Additionally, `setAccountedVBalance` and `setAccountedUSDCBalance` functions were introduced. While these provide the owner with significant control, the client implemented safeguards to prevent abnormal accounting by validating deployment timestamps and acknowledged the operational risks, intending to manage these powers through off-chain administrative security protocols. (Ref: https://github.com/vibe-vxd/VXDContract/blob/3dd7afe683f03a8046900a74e63fed794659d387/contracts/VXDGateway.sol#L1052-L1060)

#7 VIBE-007 Minor Suggestions - Code Optimization and Maintenance Improvements

ID	Summary	Severity
VIBE-007	Five areas for improvement were identified regarding code quality, gas efficiency, and operational transparency. These include removing redundant inheritance, synchronizing logic with documentation, adding event logs, resolving naming conflicts, and implementing rigorous constructor validations.	Informational

Description

- A - Redundant Inheritance (VGovernance): Since VGovernance acts as a non-transferable Soulbound Token (SBT), inheriting `ERC20Permit` is unnecessary. Enabling signature-based approvals is meaningless if the transfer functionality is disabled; removing this reduces deployment costs and code complexity.
- B - TWAP Logic Discrepancy: In `VXDContract._getTWAPPrice`, the internal comments and the actual code logic are inconsistent regarding the period calculation. Revising the comment from `(period + 60)` to `period`, or `secondsAgos[0] = period;` to `secondsAgos[0] = period + 60;`, or `int56(uint56(period - 60)))` to `int56(uint56(period)))`.
- C - Missing Event Logs: Functions that modify global state variables, specifically `setTWAPPeriods`, `setMintConfig`, and `setRedeemConfig` do not emit events. This limits the ability of off-chain tools to track administrative
- D - Naming Collision: In VGovernance, the `owner` argument in the nonces function conflicts with a function name in the parent contract, leading to potential developer confusion.
- E - Constructor Arguments Sanity Check: The VXDContract constructor lacks input validation. There are no checks to ensure that token decimals (18 for V, 6 for USDC/VXD) or pool addresses are initialized correctly.

Impact

Informational

These findings do not present immediate security vulnerabilities. However, they impact gas efficiency, protocol transparency, and the long-term maintainability of the smart contracts.

Recommendation

Apply the specific code adjustments detailed in the Description to align the project with Solidity best practices and improve operational clarity.

Remediation

Patched

- A: `ef047a8e09bd6257ff6914e465f9527d25516972`
- B: `55ed09bba8248b2f11ca4b47311ab8bfd274c9b9`
- C: `c33c00f5f4c1796a625e0efd253562ef57bbb913`
- D: `c5a05fbc2dceac1af67979229e4d555de05baeaa`
- E: `981b90c4ed6f14364b3c738539da3f7c83571447`, `1cdc7b7bb48350eeb107cbbb6de9ad8a98359ba3`

#8 `VIBE-008` Structural Design Flaws and Architectural Recommendations for VGovernance and VXDContract

ID	Summary	Severity
`VIBE-008`	The VGovernance and VXDContract designs exhibit six structural limitations concerning asset flexibility, risk distribution, and queue management. These flaws may discourage governance participation, create unfavorable economic conditions for users, and expose the protocol to stability risks during extreme market volatility.	Informational

Description

The following structural issues and potential improvements were identified:

1. Lack of Partial Unlock in VGovernance: The unlock function only permits full unlocking of staked assets. This "all-or-nothing" approach discourages creators from participating in governance due to a lack of liquidity management. Adding an amount parameter to allow partial unlocking would enhance flexibility.
2. TWAP Latency Risks: The Time-Weighted Average Price (TWAP) may lag behind rapid market shifts. If the V token price surges, the protocol may overpay V tokens during redemption; if it crashes, users receive insufficient value. Reducing the `twapPeriodForRedeem` during high volatility would better align the protocol with current market prices.
3. Asymmetric Risk/Reward Structure: Users entering the redemption queue face a period (between days 11 and 21) where they cannot cancel or withdraw. During this time, they bear the full risk of price depreciation (e.g., a 90% drop results in a 90% loss), yet their gains are capped at the original dollar value if the price increases, denying them any market upside.
4. Single Queue Constraints: A global queue forces users to pay gas to process the preceding requests of others. Additionally, a user intending to cancel after 21 days could be "force-processed" by another actor during the 11–21 day window, effectively removing their ability to opt out.
5. Daily Limit Dilemma: Low limits create bottlenecks for high-volume creators, while high limits increase the risk of a "Death Spiral." Similar to the Iron Finance incident, rapid price drops could trigger mass redemptions and dumping, potentially leading to an ecosystem collapse or a cheap governance takeover as users exit VGovernance.
6. UX Impact of Emergency Resets: While `emergencyCancelRedeems` ensures data integrity, it forces users near the end of their waiting period to restart the entire two-week process in a new contract.

Establishing a clear prior-notice policy is essential to mitigate user dissatisfaction during such interventions.

Impact

Informational

While these findings do not represent direct code vulnerabilities, the inflexible withdrawal policies and asymmetric risk distribution may lead to user churn. The single-queue architecture and Daily Limit dilemma specifically weaken the protocol's resilience against bank runs or malicious governance attacks.

Recommendation

Implement partial unlocking, dynamic TWAP periods, and per-user redemption queues to improve flexibility and gas efficiency. Furthermore, refine the Daily Limit parameters and establish transparent operational policies for emergency resets to balance protocol security with user experience.

References

- https://en.wikipedia.org/wiki/Iron_Finance#Tokens

Remediation

Acknowledged

The Vibe team has acknowledged these structural concerns and the associated risks but has elected to proceed with the current design without implementing code changes at this time.

#9 VIBE-012 Initial Supply Inflation via Redundant Multichain Minting in TokenVOFT

ID	Summary	Severity
VIBE-012	The TokenVOFT constructor contains logic that automatically mints the initial supply of 10 billion tokens to the owner. When deploying the contract across multiple chains to enable Omnichain Fungible Token (OFT) functionality, this results in the creation of 10 billion tokens on every chain, leading to a massive and unintended increase in the global total supply.	High

Description

The vulnerability exists in the implementation of the TokenVOFT constructor, which executes a _mint operation for 10 billion tokens upon deployment. Because the LayerZero OFT standard requires the same contract to be deployed across all participating blockchain networks, the constructor runs independently on each chain.

As a result, the aggregate total supply across the entire network becomes the intended supply multiplied by the number of deployed chains (e.g., 30 billion tokens if deployed on three chains). If these tokens are moved across chains via a bridge, the total supply on a single chain could exceed the protocol's intended cap, causing catastrophic failures in supply management and tokenomics.

Impact

High

This issue leads to an uncontrolled inflation of the total token supply, directly violating the protocol's economic design. It introduces a high risk of total supply corruption, which would undermine the token's value and the integrity of the multichain ecosystem once bridging is active.

Recommendation

Restrict the initial supply generation to a single designated "Hub Chain" using one of the following methods:

- Chain ID Validation: Define a constant HUB_CHAIN_ID within the contract and wrap the minting logic in a conditional check: `if (block.chainid == HUB_CHAIN_ID) { _mint(...); }`.

- Deployment Variation: Ensure the constructor logic that mints the initial supply is only present in the deployment script or contract version intended for the primary chain (Hub Chain), while secondary chain deployments are initialized with zero supply.

Remediation

Patched

Commit: `1bae5eb36c9506c17c28ee08bcb471e77ba32f8b`

#10 VIBE-013 Redundant Inflation Minting in Multichain TokenVOFT

ID	Summary	Severity
VIBE-013	The TokenV contract hardcodes an annual inflation amount that is triggered independently on every chain where it is deployed. In a multichain environment, this creates an unintended multiplier effect, increasing the global inflation rate by the number of active chains and severely diluting token value.	High

Description

The contract defines a constant ANNUAL_INFLATION of 20 million tokens, which the DAO can claim annually. Under the LayerZero OFT (Omnichain Fungible Token) standard, identical contract logic is deployed across multiple networks such as Ethereum, Arbitrum, and Optimism. Because each contract instance operates in isolation and lacks a shared state regarding inflation claims, every chain independently permits the minting of the 20 million token allocation. Consequently, the aggregate annual inflation for the ecosystem scales to 20,000,000 times N(where N is the number of chains), rather than the intended 20 million tokens total.

Impact

High

This flaw leads to uncontrolled token supply expansion that far exceeds the protocol's economic design. The resulting hyper-inflation would collapse the project's tokenomics, devalue existing holdings, and undermine the long-term viability of the ecosystem.

Recommendation

Refer to VIBE-012.

Remediation

Patched

Commit: c9708cc27671ea12ba261092b809d14d1f6f22af, 7d077eadf6079bea24c499367c19d194d2c3b5f1

#11 `VIBE-014` Logic Inconsistency and Redundant Exception Handling in `emergencyCancelRedeems`

ID	Summary	Severity
`VIBE-014`	The `emergencyCancelRedeems` function incorrectly skips failed redemption requests and includes redundant state checks. This creates a logical inconsistency with the standard cancellation process and limits the effectiveness of the emergency recovery mechanism.	Low

Description

The current implementation of `emergencyCancelRedeems` utilizes the following conditional check to skip certain requests: `if (request.processed || request.isCanceled || request.isFailed) continue;`

This implementation presents two primary issues:

1. Inconsistency with `cancelRedeem`: The standard `cancelRedeem` function permits the cancellation of requests even if they are in a failed state (`isFailed`). The emergency function should mirror this behavior to ensure that all non-finalized requests can be cleared during critical protocol events. By skipping `isFailed` requests, the emergency function fails to provide comprehensive state recovery.
2. Redundant Logic: The check for `request.processed` is structurally redundant within the context of this function, as it targets redemptions that have not yet reached a finalized "processed" state.

Impact

Low

This issue represents a logic flaw that prevents the uniform handling of redemption requests during an emergency. While it does not directly facilitate the theft of assets, it hinders administrative recovery efforts and creates an inconsistent user experience by leaving failed requests in an unresolvable state during emergency interventions.

Recommendation

Simplify the exception handling logic within the `emergencyCancelRedeems` function. It is recommended to update the condition to only skip requests that have already been successfully canceled: `if`

`(request.isCanceled) continue;`

Remediation

Patched

Commit: `894ee36e2292217907961c0ef4e9269713707483`

#12 VIBE-015 Index Update Failure for Failed Redemption Requests in `processRedeem`

ID	Summary	Severity
VIBE-015	The `processRedeem` function fails to advance the global processing index when a redemption request enters a failed state. This leads to redundant evaluations of previously failed requests in subsequent function calls, causing unnecessary gas consumption.	Informational

Description

Within the `processRedeem` function, if a specific request fails, triggering `request.isFailed = true`, the logic does not update the `latestProcessedRedeemIndex`. For instance, if the final request in a processed batch fails, the transaction concludes without incrementing the index to reflect that this entry has already been addressed.

As a result, the next time `processRedeem` is invoked, the system starts from the outdated index and re-examines the same failed requests. This lack of index progression forces the contract to perform redundant operations and loop through known-failed entries repeatedly.

Impact

Informational

This is a minor logic flaw that leads to operational inefficiency. While it does not represent a direct security threat or risk to funds, it causes persistent gas waste for the protocol and its users by inflating the computation required for redemption processing.

Recommendation

Modify the `processRedeem` function to ensure that `latestProcessedRedeemIndex` is updated to `i + 1` whenever a request is marked as failed. This ensures the queue pointer always advances past handled entries, maintaining the efficiency of the redemption pipeline.

Remediation

Patched

Commit: `90cdedff384bd9bf577a73a1839ffa25519a1516`

The Vibe team implemented a conditional index update strategy to balance processing efficiency with recovery requirements. If a redemption fails specifically due to insufficient tokenUSDC or tokenV balances in the contract, the `latestProcessedRedeemIndex` is not increased. This ensures that the next execution of `processRedeem` restarts from the same request once liquidity is available.

However, for all other execution errors caught within the try-catch block, the contract now performs `latestProcessedRedeemIndex = i + 1`. This allows the system to skip problematic or malicious requests that would otherwise permanently stall the queue, while marking them as failed.

#13 VIBE-016 Redundant State Validation in processRedeem

ID	Summary	Severity
VIBE-016	The processRedeem function includes an unnecessary check for the processed status of requests. Since the system's pointer always starts after processed items, this condition is logically unreachable and results in minor gas waste.	Informational

Description

In the VXDContract logic, once a redemption request is finalized (`request.processed = true`), the `latestProcessedRedeemIndex` is immediately updated to the next index in the queue (`queueIndex + 1`). When the `processRedeem` function subsequently iterates through the queue to handle pending requests, it starts from this updated index. Consequently, it is structurally impossible for the loop to encounter a request where `request.processed` is already `true`. The existing conditional check for this state is redundant and performs unnecessary computations during execution.

Impact

Informational

This issue is a minor optimization flaw. While it does not pose a security risk or affect functional correctness, it causes unnecessary gas consumption by executing a validation check for a state that can never be reached within the function's loop.

Recommendation

Remove the redundant request.processed check from the loop's conditional logic to optimize gas usage. It is recommended to simplify the statement from `if (request.processed || request.isCanceled)` to `if (request.isCanceled)`.

Remediation

Patched

Commit: `0f523e268adeddd0ea0865051d1e3747a4809c1b`

Revision History

Version	Date	Description
1.0	January 21, 2026	Initial Report

Theori, Inc. (“We”) is acting solely for the client and is not responsible to any other party. Deliverables are valid for and should be used solely in connection with the purpose for which they were prepared as set out in our engagement agreement. You should not refer to or use our name or advice for any other purpose. The information (where appropriate) has not been verified. No representation or warranty is given as to accuracy, completeness or correctness of information in the Deliverables, any document, or any other information made available. Deliverables are for the internal use of the client and may not be used or relied upon by any person or entity other than the client. Deliverables are confidential and are not to be provided, without our authorization (preferably written), to entities or representatives of entities (including employees) that are not the client, including affiliates or representatives of affiliates of the client.