BLOCKSEC

Security Audit
Report for SyncVault Contracts

Date: April 28, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	R25
Target	SyncVault Contracts

Version History

Version	Date	Description
1.0	April 28, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-04-28 09:12:12+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of SyncVault Contracts of R25.

The `SyncVault` contracts form an upgradeable tokenized asset vault system within R25. Following a synchronous redemption model, `SyncVault` issues time-locked shares upon deposit and processes redemptions either immediately within configured limits or through a queued flow for deferred fulfillment. Throughout this module, deposited assets are forwarded to a custodian address, withdrawals are funded by a dedicated payer account, and vault instances are deployed and tracked by a factory contract.

Specifically, for code in `Version 1`, the audit only focuses on the smart contracts in the following directories/files:

- src/r25-pc

For code in `Version 2`, the audit only focuses on the smart contracts in the following directories/files:

- src/r25-pc/SyncVault

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
solidity-contract	`Version 1`	`f11f8770a10d881c1ba0f0c58759a86c24b7cb15`
	`Version 2`	`33133de16b62f68e646bf38983a3b93509109f48`
	`Version 3`	`f3e56320fe7b86d9d332f17b8d1987cd2f3f4e76`

[1] https://github.com/RiftSwapLabs/solidity-contract

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness

- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **eight** potential security issues. Besides, we have **twelve** recommendations and **eight** notes.

- Low Risk: 8
- Recommendation: 12
- Note: 8

ID	Severity	Description	Category	Status
1	Low	Incorrect rounding direction	Security Issue	Fixed
2	Low	Circumvention of instant redemption limit	Security Issue	Confirmed
3	Low	Potential gas griefing attacks	Security Issue	Confirmed
4	Low	Lack of overrides for ERC4626 view functions	Security Issue	Fixed
5	Low	Lack of zero-value validation in deposit and withdrawal operations	Security Issue	Fixed
6	Low	Incorrect conversion when `totalManagedAssets` is zero	Security Issue	Fixed
7	Low	Incorrect implementation of ERC-4626 maximum limit functions	Security Issue	Partially Fixed
8	Low	Lack of handling for invalid requests	Security Issue	Confirmed
9	-	Add an upper-bound check in `setMaxFluctuationBps()`	Recommendation	Confirmed
10	-	Use `caller` as the parameter for function `_processWithdrawal()`	Recommendation	Confirmed
11	-	Correct the field name in the event `CreateVault`	Recommendation	Fixed
12	-	Fix storage gap size	Recommendation	Confirmed
13	-	Remove redundant code	Recommendation	Fixed
14	-	Align comments with implementation	Recommendation	Fixed
15	-	Add underlying consistency checks in `VaultMintable`	Recommendation	Fixed
16	-	Use EIP-7201 for slot derivation	Recommendation	Fixed
17	-	Add input validation in function `createVault()`	Recommendation	Confirmed
18	-	Optimize the sequence of checks	Recommendation	Fixed
19	-	Revise the incorrect parameters	Recommendation	Confirmed
20	-	Revise incorrect annotations	Recommendation	Confirmed
21	-	Token assumption	Note	-
22	-	Different effect time of personal quota	Note	-
23	-	Inconsistency among vault logic due to implementation updates	Note	-

24	-	Deviations from the EIP-4626 standard	Note	-
25	-	Centralized withdrawal mechanism	Note	-
26	-	Ensure the correctness and timeliness of Oracle reports	Note	-
27	-	Share token transfer restriction	Note	-
28	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Incorrect rounding direction

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `BaseVault`, the function `withdraw()` uses function `previewWithdrawWithFee()` to compute the fee-inclusive asset amount (i.e., `actualAssets`) before converting to shares. However, the function incorrectly rounds down when computing the fee-inclusive asset amount. This understates the resulting assets and shares amounts, causing the protocol to collect less fee revenue than intended on every withdrawal.

The same issue exists in the function `previewWithdraw()`.

```
454   function previewWithdraw(uint256 assets) public view override returns (uint256) {
455       uint256 actualAssets = assets.mulDiv(
456           FEE_PRECISION,
457           FEE_PRECISION - withdrawFeeBps,
458           Math.Rounding.Floor
459       );
460       return _convertToShares(actualAssets, Math.Rounding.Ceil);
461   }
462
463   /**
464    * @notice Previews the withdrawal outcome including both shares required and fees charged
465    * @dev Provides detailed breakdown of withdrawal transaction
466    * @param assets The amount of assets to withdraw
467    * @return shares The amount of shares that would be burned
468    * @return fee The amount of fees that would be charged
469    */
470   function previewWithdrawWithFee(uint256 assets) public view returns (uint256, uint256) {
471       uint256 actualAssets = assets.mulDiv(
472           FEE_PRECISION,
473           FEE_PRECISION - withdrawFeeBps,
474           Math.Rounding.Floor
475       );
476       uint256 shares = _convertToShares(actualAssets, Math.Rounding.Ceil);
477       return (shares, actualAssets - assets);
```

```
478     }
```

Listing 2.1: src/r25-pc/BaseVault.sol

Impact The function `withdraw()` may undercharge fees, causing cumulative protocol fee revenue loss over time.

Suggestion Revise the fee logic in functions `previewWithdraw()` and `previewWithdrawWithFee()`.

2.1.2 Circumvention of instant redemption limit

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract `BaseVault`, functions `getUserDailyWithdrawQuota()` and `_updateUserDailyQuota()` enforce a per-address daily instant withdrawal quota. However, this design is improper because the quota is tracked only by the address in the mapping `userDailyWithdrawQuota`. As a result, a single holder can split unlocked shares across multiple addresses to consume multiple address-level quotas within the same day, leading to more instant liquidity being withdrawn than intended and faster depletion of liquidity-provider capacity.

```
1090    /**
1091     * @notice Retrieves the current daily instant withdrawal quota for a specific user
1092     * @dev Returns both remaining quota and the day timestamp for tracking purposes
1093     * @param user The address of the user to query
1094     * @return remainingQuota The amount of assets available for instant withdrawal today
1095     * @return dayTimestamp The normalized timestamp representing the current quota day
1096     */
1097    function getUserDailyWithdrawQuota(address user) external view returns (uint256, uint64) {
1098        UserDailyWithdrawQuotaInfo memory quotaInfo = userDailyWithdrawQuota[user];
1099        uint64 currentDay = _normalize(block.timestamp);
1100
1101        if (quotaInfo.dayTimestamp != currentDay) {
1102            return (walletDailyInstantWithdrawQuota, currentDay);
1103        }
1104
1105        return (quotaInfo.remainingQuota, quotaInfo.dayTimestamp);
1106    }
```

Listing 2.2: src/r25-pc/BaseVault.sol

Impact This flaw allows a single holder to withdraw more instant liquidity than intended by distributing shares across multiple addresses, leading to faster depletion of liquidity-provider capacity and reduced exit availability for later users.

Suggestion Adjust the instant redemption limit logic to prevent circumvention through multiple addresses.

Feedback from the project This behavior is by design.

2.1.3 Potential gas griefing attacks

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The contract `BaseVault` stores each user's time-locked share positions as an ordered linked list and linearly traverses these batches across multiple execution paths. As the number of batches grows, the gas cost of these operations increases. However, there is no upper bound on the number of timelock batches. This allows an attacker to repeatedly transfer shares on N distinct days to inject nodes into a benign user's list. This griefing attack gradually increases the victim's list size and makes subsequent `deposit()`, `redeem()`, `withdraw()`, and related query paths increasingly gas-expensive.

```
1071    function getAvailableShares(address owner, uint64 currentTime) public view returns (uint256
            availableShares) {
1072        uint64 unlockTime = _nextTimelock[owner][GUARD];
1073        while (unlockTime != GUARD && unlockTime <= currentTime) {
1074            availableShares += _sharesLocks[owner][unlockTime];
1075            unlockTime = _nextTimelock[owner][unlockTime];
1076        }
1077    }
```

Listing 2.3: src/r25-pc/BaseVault.sol

Impact The attacker could intentionally increase the gas costs of a benign user's transfer, redemption, withdrawal, and query paths over time.

Suggestion Revise the logic accordingly.

Feedback from the project The project considered this risk acceptable, as a user is expected to hold only one share record per day. Exploiting this issue would require the attacker to transfer shares to the victim continuously over a long period.

2.1.4 Lack of overrides for ERC4626 view functions

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `BaseVault` is exposed through `IVault`, which inherits `IERC4626`. Its comments also describe it as ERC4626-compliant (line 27). However, functions `maxWithdraw()`, `maxRedeem()`, and `maxMint()` inherit the OpenZeppelin ERC4626 defaults, which only consider the user's share balance and asset conversion. They do not account for protocol-specific constraints, such as timelock unlock status and the vault's supply and fee rules. As a result, these view functions can return values that do not match the vault's actual execution constraints.

```
24/**
25 * @title BaseVault
26 * @author RiftWriter Company Limited
27 * @notice ERC4626-compliant time-locked vault with flexible withdrawal mechanisms
```

```
28 * @dev This is an ERC4626-compatible vault that implements a time-locked share mechanism.
```

Listing 2.4: src/r25-pc/BaseVault.sol

```
121    /// @notice Returns the maximum amount of the underlying asset that can be withdrawn from the
           owner balance in the
122    /// Vault, through a withdrawal call.
123    /// @dev
124    /// - MUST return a limited value if owner is subject to some withdrawal limit or timelock.
125    /// - MUST NOT revert.
126    function maxWithdraw(address owner) external view returns (uint256 maxAssets);
```

Listing 2.5: lib/forge-std_v1.9.6/src/interfaces/IERC4626.sol

```
156    /// @notice Returns the maximum amount of Vault shares that can be redeemed from the owner
           balance in the Vault,
157    /// through a redeem call.
158    /// @dev
159    /// - MUST return a limited value if owner is subject to some withdrawal limit or timelock.
160    /// - MUST return balanceOf(owner) if owner is not subject to any withdrawal limit or timelock
           .
161    /// - MUST NOT revert.
162    function maxRedeem(address owner) external view returns (uint256 maxShares);
```

Listing 2.6: lib/forge-std_v1.9.6/src/interfaces/IERC4626.sol

Impact Users may rely on functions `maxWithdraw()`, `maxRedeem()`, and `maxMint()` when initiating operations, and incorrect return values can cause those operations to revert.

Suggestion Override functions `maxWithdraw()`, `maxMint()`, and `maxRedeem()` to reflect accurate values.

2.1.5 Lack of zero-value validation in deposit and withdrawal operations

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `BaseVault`, functions `redeem()` and `deposit()` lack zero-value verification. Specifically, the function `redeem()` does not verify whether `assetsAfterFee` is greater than zero. As a result, a user may burn a non-zero amount of shares yet receive zero assets after rounding and fees. Similarly, function `deposit()` does not check whether `previewDepositWithFee(assets)` returns non-zero shares, allowing the user to transfer assets without receiving any shares in return.

```
594    function redeem(
595        uint256 shares, address receiver, address owner
596    ) public override nonReentrant whenNotPaused returns (uint256) {
597        if (receiver != owner) {
598            revert InvalidReceiver(owner, receiver);
599        }
600        if (shares < minRedeemLimit) {
```

```
601             revert BelowMinRedeem(shares, minRedeemLimit);
602         }
603
604         uint256 sharesUnlocked = getAvailableShares(owner, uint64(block.timestamp));
605         if (shares > sharesUnlocked) {
606             revert InsufficientUnlockedShares(shares, sharesUnlocked);
607         }
608
609         (uint256 assetsAfterFee, uint256 fee) = previewRedeemWithFee(shares);
610
611         uint256 requestId = _customWithdraw(_msgSender(), receiver, owner, assetsAfterFee, shares);
612
613         totalManagedAssets -= assetsAfterFee + fee;
614         totalFeesCollected += fee;
615
616         emit CustomWithdraw(_msgSender(), receiver, owner, assetsAfterFee, shares, fee, requestId);
617         return assetsAfterFee;
618     }
```

Listing 2.7: src/r25-pc/BaseVault.sol

```
517     function deposit(uint256 assets, address receiver) public override nonReentrant whenNotPaused
            returns (uint256) {
518         if (receiver != _msgSender()) {
519             revert InvalidReceiver(_msgSender(), receiver);
520         }
521         if (assets < minDepositLimit) {
522             revert BelowMinDeposit(receiver, assets, minDepositLimit);
523         }
524
525         uint256 maxAssets = maxDeposit(receiver);
526         if (assets > maxAssets) {
527             revert ERC4626ExceededMaxDeposit(receiver, assets, maxAssets);
528         }
529
530         (uint256 shares, uint256 fee) = previewDepositWithFee(assets);
531         if (totalSupply() + shares > maxSupply) {
532             revert ExceededMaxSupply(receiver, shares, maxSupply);
533         }
534         _deposit(_msgSender(), receiver, assets, shares);
535
536         totalManagedAssets += assets - fee;
537         totalFeesCollected += fee;
538
539         emit CustomDeposit(_msgSender(), receiver, assets, shares, fee);
540         return shares;
541     }
```

Listing 2.8: src/r25-pc/BaseVault.sol

Impact Users can lose value by burning shares for zero assets or depositing assets for zero shares under certain rounding and fee conditions.

Suggestion Add non-zero value validation to these operations.

2.1.6 Incorrect conversion when `totalManagedAssets` is zero

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `BaseVault`, the functions `_convertToShares()` and `_convertToAssets()` provide conversion between assets and shares. When the vault is in the initialization phase (i.e., the variables `totalManagedAssets` or `totalSupply()` are zero), the conversion logic returns a one-for-one rate. However, the implementation is problematic, since it does not identify the correct initial state. For example, when the vault is shallow and the loss is excessive, function `handleOracleReport()` updates the lower NAV, causing `totalManagedAssets` to be calculated as 0. Subsequent conversions can deviate from the residual share state, leading to distorted pricing for subsequent asset flows.

```
1135    /**
1136     * @dev Internal conversion function (from assets to shares) with support for rounding
             direction.
1137     */
1138    function _convertToShares(uint256 assets, Math.Rounding rounding) internal view override
            returns (uint256) {
1139        uint256 totalSupply = this.totalSupply();
1140        if (totalSupply == 0 || totalManagedAssets == 0) {
1141            return assets;
1142        }
1143
1144        return assets.mulDiv(totalSupply, totalManagedAssets, rounding);
1145    }
```

Listing 2.9: src/r25-pc/BaseVault.sol

```
845         totalManagedAssets = totalSupply().mulDiv(report.todayNav, NAV_PRECISION, Math.Rounding.
                Floor);
846         lastAccountingTimestamp = report.accountingTimestamp;
847         lastReportId = report.reportId;
848         emit OracleReportHandled(report.accountingTimestamp, report.reportId, totalSupply(),
                totalManagedAssets, report.todayNav, totalFeesCollected);
849     }
```

Listing 2.10: src/r25-pc/BaseVault.sol

Impact This flaw can cause incorrect conversions in a zero-asset boundary state, leading to distorted pricing for subsequent asset flows.

Suggestion Restrict the one-for-one rate conversion to the true initialization case.

2.1.7 Incorrect implementation of ERC-4626 maximum limit functions

Severity Low

Status Partially Fixed in `Version 3`

Introduced by `Version 2`

Description In the contract `SyncVault`, the ERC-4626 maximum limit functions are implemented incorrectly. Specifically, functions `maxMint()` and `maxWithdraw()` do not account for fees, and the functions `maxDeposit()` and `maxMint()` do not reflect the `maxSupply` constraint. Additionally, none of the max functions return `0` when the vault is paused, as required by the ERC-4626 specification [1]. As a result, integrators may receive incorrect capacity signals, leading to failed transactions and inconsistent ERC-4626 behavior.

```
416   /**
417    * @notice Returns the maximum amount of shares that can be minted by a given address
418    * @dev Calculated from maxDeposit limit
419    * @return The maximum mint amount in share units
420    */
421   function maxMint(address) public view override returns (uint256) {
422       return _convertToShares(maxDepositLimit, Math.Rounding.Floor);
423   }
424
425   /**
426    * @notice Returns the maximum amount of assets that can be withdrawn by a given address
427    * @dev Returns the user's total balance converted to assets
428    * @param owner The address to query
429    * @return The maximum withdrawable asset amount
430    */
431   function maxWithdraw(address owner) public view override returns (uint256) {
432       uint256 unlockedBalance = getAvailableShares(owner, uint64(block.timestamp));
433       return _convertToAssets(unlockedBalance, Math.Rounding.Floor);
434   }
```

Listing 2.11: src/r25-pc/SyncVault/SyncVault.sol

```
412   function maxDeposit(address) public view override returns (uint256) {
413       return maxDepositLimit;
414   }
415
416   /**
417    * @notice Returns the maximum amount of shares that can be minted by a given address
418    * @dev Calculated from maxDeposit limit
419    * @return The maximum mint amount in share units
420    */
421   function maxMint(address) public view override returns (uint256) {
422       return _convertToShares(maxDepositLimit, Math.Rounding.Floor);
423   }
424
425   /**
426    * @notice Returns the maximum amount of assets that can be withdrawn by a given address
427    * @dev Returns the user's total balance converted to assets
428    * @param owner The address to query
429    * @return The maximum withdrawable asset amount
430    */
431   function maxWithdraw(address owner) public view override returns (uint256) {
432       uint256 unlockedBalance = getAvailableShares(owner, uint64(block.timestamp));
433       return _convertToAssets(unlockedBalance, Math.Rounding.Floor);
```

[1] https://eips.ethereum.org/EIPS/eip-4626

```
434    }
435
436    /**
437     * @notice Returns the maximum amount of shares that can be redeemed by a given address
438     * @dev Returns the user's total balance
439     * @param owner The address to query
440     * @return The maximum redeemable share amount
441     */
442    function maxRedeem(address owner) public view override returns (uint256) {
443        return getAvailableShares(owner, uint64(block.timestamp));
444    }
```

Listing 2.12: src/r25-pc/SyncVault/SyncVault.sol

Impact This may cause failed transactions due to incorrect capacity signals returned by these functions.

Suggestion Align these functions with the specification of EIP4626.

Feedback from the project The project clarified that they temporarily do not consider the pause situation and will upgrade the implementation in the future.

2.1.8 Lack of handling for invalid requests

Severity Low

Status Confirmed

Introduced by `Version 2`

Description In the contract `SyncVault`, the function `fulfillPendingWithdrawals()` processes pending withdrawal requests in FIFO order, transferring assets to the `recipient` through `IERC20(asset()).safeTransferFrom()`. If the underlying asset enforces blacklist restrictions, e.g., `USDC`, the transfer will revert when the recipient is blacklisted. While the function `skipWithdrawalRequest()` allows the `CONTROLLER` to skip such a request, it only moves the request to the end of the queue rather than permanently removing it. Meanwhile, the contract lacks other handling logic for such invalid requests. If the `recipient` remains blacklisted, the same revert will occur when the request is processed again. Over time, the queue may accumulate permanently unexecutable requests, causing repeated transaction failures.

```
726    function fulfillPendingWithdrawals(uint256 budget) external override nonReentrant onlyRole(
           BizRole.CONTROLLER) {
727        if (payer == address(0)) {
728            revert PayerNotSet();
729        }
730        if (IERC20(asset()).balanceOf(payer) < budget) {
731            revert PayerInsufficientBalance(payer, IERC20(asset()).balanceOf(payer), budget);
732        }
733        if (IERC20(asset()).allowance(payer, address(this)) < budget) {
734            revert PayerInsufficientAllowance(payer, IERC20(asset()).allowance(payer, address(this)
                ), budget);
735        }
736
737        uint256 count = 0;
```

```
738         uint256 preRequestIndex = getLastProcessedRequestId();
739         uint256 currentIndex = preRequestIndex + 1;
740         uint256 lastRequestId = getLastRequestId();
741         uint256 preRequestAmount = _getQueue()[preRequestIndex].cumulativeAmount;
742         for (; currentIndex <= lastRequestId && count < MAX_REQUEST_BATCH; currentIndex++) {
743             // process pending requests
744             WithdrawalRequest storage request = _getQueue()[currentIndex];
745             if (request.fulfilled) {
746                 // Update preRequestAmount and lastProcessedRequestId when skipping fulfilled
                        requests
747                 _setLastProcessedRequestId(currentIndex);
748                 preRequestAmount = request.cumulativeAmount;
749                 continue;
750             }
751
752             uint256 requestAmount = request.cumulativeAmount - preRequestAmount;
753             if (requestAmount > budget) {
754                 break;
755             }
756
757             budget -= requestAmount;
758             preRequestAmount = request.cumulativeAmount;
759
760             request.fulfilled = true;
761             _getRequestsByOwner()[request.owner].remove(currentIndex);
762             _setLastProcessedRequestId(currentIndex);
763             IERC20(asset()).safeTransferFrom(payer, request.recipient, requestAmount);
764             emit WithdrawalFulfilled(currentIndex, request.requestor, request.recipient, request.
                    owner, block.timestamp, requestAmount);
765             count++;
766         }
767     }
```

Listing 2.13: src/r25-pc/SyncVault/SyncVault.sol

```
779     function skipWithdrawalRequest(uint256 requestId) external override onlyRole(BizRole.
            CONTROLLER) {
780         if (requestId == 0 || requestId > getLastRequestId()) {
781             revert InvalidInput();
782         }
783
784         WithdrawalRequest storage request = _getQueue()[requestId];
785         if (request.fulfilled) {
786             revert InvalidInput();
787         }
788
789         // Store original request data
790         uint256 newRequestId;
791         address requestor = request.requestor;
792         address recipient = request.recipient;
793         address owner = request.owner;
794         uint256 originalAmount = request.cumulativeAmount - _getQueue()[requestId - 1].
                cumulativeAmount;
```

```
795
796         // Mark original request as fulfilled
797         request.fulfilled = true;
798
799         // Remove old requestId from user's request set
800         _getRequestsByOwner()[owner].remove(requestId);
801
802         // Create new request at the tail of the queue using _requestWithdrawal
803         newRequestId = _requestWithdrawal(originalAmount, requestor, recipient, owner);
804
805         // Emit events
806         emit RequestIdChanged(requestId, newRequestId);
807         emit WithdrawalFulfilled(requestId, requestor, recipient, owner, block.timestamp, 0);
808     }
```

Listing 2.14: src/r25-pc/SyncVault/SyncVault.sol

Impact This issue may leave permanently unexecutable requests in the queue and continuously increase operational overhead.

Suggestion Add a permanent invalidation function for queued withdrawal requests that cannot be executed.

Feedback from the project The project confirmed that the impact was acceptable.

2.2 Recommendation

2.2.1 Add an upper-bound check in `setMaxFluctuationBps()`

Status Confirmed

Introduced by `Version 1`

Description In contract `BaseVault`, functions `setMaxSupply()`, `setMaxDepositLimit()`, and `setMaxFluctuationBps()` lack boundary validation on crucial configurations.

Specifically, functions `setMaxSupply()` and `setMaxDepositLimit()` permit a value of zero, which would cause every subsequent `deposit()` and `mint()` invocation to revert immediately. Function `setMaxFluctuationBps()` has no lower or upper bound. Setting it to zero causes every oracle report to be rejected as the computed `fluctuationBps` will always exceed the limit, permanently blocking NAV updates. Setting it above `FEE_PRECISION` renders the fluctuation check trivially satisfiable, negating its intended protection against abrupt NAV manipulation.

```
927     function setMaxSupply(uint256 maxSupply_) external onlyRole(BizRole.CONTROLLER) {
928         uint256 preMaxSupply = maxSupply;
929         maxSupply = maxSupply_;
930         emit ConfigUpdated(preMaxSupply, maxSupply_, ConfigType.MAX_SUPPLY);
931     }
```

Listing 2.15: src/r25-pc/BaseVault.sol

```
940     function setMaxDepositLimit(uint256 maxDepositLimit_) external onlyRole(BizRole.CONTROLLER) {
941         uint256 preMaxDepositLimit = maxDepositLimit;
942         maxDepositLimit = maxDepositLimit_;
```

```
943         emit ConfigUpdated(preMaxDepositLimit, maxDepositLimit_, ConfigType.MAX_DEPOSIT_LIMIT);
944     }
```

Listing 2.16: src/r25-pc/BaseVault.sol

```
984     function setMaxFluctuationBps(uint64 maxFluctuationBps_) external onlyRole(BizRole.CONTROLLER)
            {
985         uint64 preMaxFluctuationBps = maxFluctuationBps;
986         maxFluctuationBps = maxFluctuationBps_;
987         emit ConfigUpdated(preMaxFluctuationBps, maxFluctuationBps_, ConfigType.MAX_FLUCTUATION_BPS
                );
988     }
```

Listing 2.17: src/r25-pc/BaseVault.sol

Suggestion Enforce clear boundary checks for the functions.

2.2.2 Use `caller` as the parameter for function `_processWithdrawal()`

Status Confirmed

Introduced by `Version 1`

Description The function `_processWithdrawal()` receives the input `caller` as the intended requestor. However, it does not forward it to `_requestWithdrawal()` and uses `msg.sender` as the requestor. It is recommended to forward the `caller` parameter and use it as the requestor to remain consistent with the function's own input context and internal logic

```
1217    function _processWithdrawal(
1218        uint256 assets,
1219        address receiver,
1220        address owner
1221    ) internal returns(uint256 requestId) {
1222        // Update user daily quota and get remaining quota
1223        uint256 userRemainingQuota = _updateUserDailyQuota(owner);
1224
1225        // Check payer account balance and allowance
1226        uint256 payerBalance = IERC20(asset()).balanceOf(payer);
1227        uint256 payerAllowance = IERC20(asset()).allowance(payer, address(this));
1228
1229        // Decision logic: instant vs queued withdrawal
1230        if (assets <= userRemainingQuota && assets <= payerBalance && assets <= payerAllowance) {
1231            // Instant redemption: all conditions met
1232            userDailyWithdrawQuota[owner].remainingQuota -= assets;
1233
1234            // Direct transfer from payer to receiver
1235            IERC20(asset()).safeTransferFrom(payer, receiver, assets);
1236            requestId = 0; // 0 indicates instant withdrawal
1237        } else {
1238            // Queued redemption: create withdrawal request
1239            requestId = _requestWithdrawal(assets, msg.sender, receiver, owner);
1240        }
```

Listing 2.18: src/r25-pc/BaseVault.sol

Suggestion Forward the `caller` parameter from `_customWithdraw()` into `_processWithdrawal()`, and pass `caller` to `_requestWithdrawal()` instead of `msg.sender`.

2.2.3 Correct the field name in the event `CreateVault`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the event `CreateVault`, the field name `assetSource` represents the underlying data source, not the asset. The current inconsistency may cause confusion for integrators and users.

```
28    event CreateVault(
29        address indexed owner,
30        address asset,
31        address assetSource,
32        address underlying,
33        bytes32 tokenName,
34        bytes32 tokenSymbol,
35        uint64 maturityPeriod,
36        bool mintable
37    );
```

Listing 2.19: src/r25-pc/VaultFactory.sol

```
76    function _deposit(
77        address caller,
78        address receiver,
79        uint256 assets,
80        uint256 shares
81    ) internal override {
82        super._deposit(caller, receiver, assets, shares);
83
84        // Mint underlying tokens from underlying asset minter to vault
85        underlyingAssetMinter.mint(address(this), shares * (10 ** decimalOffset));
86    }
```

Listing 2.20: src/r25-pc/VaultMintable.sol

Suggestion Rename the `assetSource` field in the event `CreateVault` to an accurate name.

2.2.4 Fix storage gap size

Status Confirmed

Introduced by `Version 1`

Description The contract `BaseVault` declares a storage gap to reserve space for future upgrades. However, since the contract declares some state variables. Therefore, the size of `__gap` should not be 50 considering OpenZeppelin's convention.

```
1467    /// forge-lint: disable-next-line(mixed-case-variable)
1468    uint256[50] private __gap;
```

```
1469}
```

Listing 2.21: src/r25-pc/BaseVault.sol

Suggestion Recalculate and adjust the `__gap` length to conform to the standard 50-slot layout.

2.2.5 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The error `BatchAlreadyExist` is defined in the contract `BaseVault`, but it is never used. It is recommended to remove them for better code readability.

```
254   /// @notice Batch with this unlock time already exists
255   error BatchAlreadyExist();
```

Listing 2.22: src/r25-pc/BaseVault.sol

Suggestion Remove the redundant code.

2.2.6 Align comments with implementation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Some comments in the current implementation do not accurately reflect the implemented behavior. It is recommended to revise them for better code accuracy and readability. Specifically, the following comments should be aligned with the implementation.

1. In the contract `WithdrawQueue`, the comment of the struct `WithdrawalRequestStatus` refers to a non-existent function `_getWithdrawalStatus()` (line 45). The corresponding helper is the function `getWithdrawalStatus()`.

```
45   /// @notice output format struct for `_getWithdrawalStatus()` method
```

Listing 2.23: src/r25-pc/WithdrawQueue.sol

```
127   /// @dev Returns the status of the withdrawal request with `_requestId` id
128   function _getStatus(
129       uint256 _requestId
130   ) internal view returns (WithdrawalRequestStatus memory status) {
131       if (_requestId == 0 || _requestId > getLastRequestId()) revert InvalidRequestId(_requestId)
              ;
132
133       WithdrawalRequest storage request = _getQueue()[_requestId];
134       WithdrawalRequest storage previousRequest = _getQueue()[_requestId - 1];
135
136       return WithdrawalRequestStatus({
137           amountToWithdraw: request.cumulativeAmount - previousRequest.cumulativeAmount,
138           requestor: request.requestor,
139           recipient: request.recipient,
```

```
140             owner: request.owner,
141             timestamp: request.timestamp,
142             isFulfilled: request.fulfilled
143         });
144     }
```

Listing 2.24: src/r25-pc/WithdrawQueue.sol

2. In the contract `WithdrawQueue`, the comment of the function `_requestWithdrawal()` says that a zero `_owner` falls back to `msg.sender` (line 149), but the implementation does not apply that behavior.

```
149     /// If `address(0)` is passed, `msg.sender` will be used as owner.
150     /// @return requestId the created withdrawal request id
151     function _requestWithdrawal(
152         uint256 _amountToWithdraw, address _caller, address _recipient, address _owner
153     ) internal returns (uint256 requestId) {
154         requestId = _enqueue(_amountToWithdraw, _caller, _recipient, _owner);
155
156         emit WithdrawalRequested(
157             requestId,
158             _caller,
159             _recipient,
160             _owner,
161             block.timestamp,
162             _amountToWithdraw
163         );
164     }
```

Listing 2.25: src/r25-pc/WithdrawQueue.sol

3. In the contract `WithdrawQueue`, the comment (line 169) of the function `_enqueue()` says that it emits the event `WithdrawalRequested`, but the function body does not emit that event. The event is emitted by the function `_requestWithdrawal()` instead.

```
169     /// Emits WithdrawalRequested event
170     function _enqueue(
171         uint256 _amountToWithdraw, address _caller, address _recipient, address _owner
172     ) internal returns (uint256 requestId) {
173         uint256 lastRequestId = getLastRequestId();
174         WithdrawalRequest storage lastRequest = _getQueue()[lastRequestId];
175
176         uint256 cumulativeAmount = lastRequest.cumulativeAmount + _amountToWithdraw;
177
178         requestId = lastRequestId + 1;
179
180         _setLastRequestId(requestId);
181
182         uint40 timestamp = uint40(block.timestamp);
183
184         _getQueue()[requestId] = WithdrawalRequest({
185             cumulativeAmount: cumulativeAmount,
186             requestor: _caller,
187             recipient: _recipient,
```

```
188             owner: _owner,
189             timestamp: timestamp,
190             fulfilled: false
191         });
192         _getRequestsByOwner()[_owner].add(requestId);
193     }
```

Listing 2.26: src/r25-pc/WithdrawQueue.sol

```
156         emit WithdrawalRequested(
157             requestId,
158             _caller,
159             _recipient,
160             _owner,
161             block.timestamp,
162             _amountToWithdraw
163         );
```

Listing 2.27: src/r25-pc/WithdrawQueue.sol

Suggestion Align the comments with the implementation.

2.2.7 Add underlying consistency checks in `VaultMintable`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `VaultMintable`, the variable `underlying` is fixed during initialization, while the variable `underlyingAssetMinter` can still be updated later. The deposit path mints through the current `underlyingAssetMinter`, but the withdrawal path always burns the fixed `underlying`. However, the contract does not validate that both paths remain tied to the same underlying token. This can leave the mint path and burn path inconsistent after a minter update.

```
59      function setUnderlyingAssetMinterer(address underlyingAssetMinter_) public onlyRole(BizRole.
            CONTROLLER) {
60          if (underlyingAssetMinter_ == address(0)) {
61              revert InvalidInput();
62          }
63          address preUnderlyingAssetMinter = address(underlyingAssetMinter);
64          underlyingAssetMinter = IMinter(underlyingAssetMinter_);
65          emit UnderlyingAssetMinterUpdated(preUnderlyingAssetMinter, underlyingAssetMinter_);
66      }
```

Listing 2.28: src/r25-pc/VaultMintable.sol

```
76      function _deposit(
77          address caller,
78          address receiver,
79          uint256 assets,
80          uint256 shares
81      ) internal override {
```

```
82        super._deposit(caller, receiver, assets, shares);
83
84        // Mint underlying tokens from underlying asset minter to vault
85        underlyingAssetMinter.mint(address(this), shares * (10 ** decimalOffset));
86    }
```

Listing 2.29: src/r25-pc/VaultMintable.sol

Suggestion Add underlying consistency checks in contract `VaultMintable`.

2.2.8 Use EIP-7201 for slot derivation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `WithdrawQueue`, the storage locations for `QUEUE_POSITION`, `LAST_REQUEST_ID_POSITION`, `LAST_PROCESSED_REQUEST_ID_POSITION`, and `REQUEST_BY_OWNER_POSITION` are expected to minimize collision risk in upgradeable contexts. However, they are derived directly from `keccak256("...")` without the `keccak256(keccak256(namespace id) - 1) & ~0xff` offset recommended by EIP-7201, which weakens the convention for avoiding storage slot clashes.

```
18    bytes32 internal constant QUEUE_POSITION = keccak256("r25.WithdrawQueue.queue");
19
20    bytes32 internal constant LAST_REQUEST_ID_POSITION = keccak256("r25.WithdrawQueue.
          lastRequestId");
21
22    bytes32 internal constant LAST_PROCESSED_REQUEST_ID_POSITION = keccak256("r25.WithdrawQueue.
          lastProcessedRequestId");
23
24    // key: owner address, value: set(request id)
25    bytes32 internal constant REQUEST_BY_OWNER_POSITION = keccak256("r25.WithdrawQueue.
          requestsByOwner");
```

Listing 2.30: src/r25-pc/WithdrawQueue.sol

Impact The storage layout is more susceptible to accidental collisions across upgrades or inherited modules.

Suggestion Derive each slot as `keccak256(keccak256(namespace id) - 1) & ~0xff` to align with EIP-7201.

2.2.9 Add input validation in function `createVault()`

Status Confirmed

Introduced by `Version 1`

Description In the contract `VaultFactory`, function `createVault()` is expected to enforce meaningful identity parameters for registry keys and ERC-20 metadata. However, it forwards `tokenName` and `tokenSymbol` without checks and derives the registry key from `keccak256(abi.encodePacked(organizat tokenName))` even when `organization` is zero and the strings are empty. This lack of validation allows a vault with empty metadata and a zero-organization namespace to be deployed.

```
88      bytes32 tokenNameHashed = keccak256(abi.encodePacked(
89          initVaultParam.organization,
90          initVaultParam.tokenName
91      ));
92      /// forge-lint: disable-next-line(asm-keccak256)
93      bytes32 tokenSymbolHashed = keccak256(abi.encodePacked(
94          initVaultParam.organization,
95          initVaultParam.tokenSymbol
96      ));
```

Listing 2.31: src/r25-pc/VaultFactory.sol

Suggestion Reject empty `tokenName`, empty `tokenSymbol`, and `organization == bytes32(0)` before deployment.

2.2.10 Optimize the sequence of checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `BaseVault`, the function `mint()` accepts `shares` as an input parameter. Therefore, the check on it (line 570) can be moved before the calculation of required assets (line 560) to reduce unnecessary operations when the max supply is reached.

```
555   function mint(uint256 shares, address receiver) public override nonReentrant whenNotPaused
          returns (uint256) {
556       if (receiver != _msgSender()) {
557           revert InvalidReceiver(_msgSender(), receiver);
558       }
559
560       (uint256 assets, uint256 fee) = previewMintWithFee(shares);
561       if (assets < minDepositLimit) {
562           revert BelowMinDeposit(receiver, assets, minDepositLimit);
563       }
564
565       uint256 maxAssets = maxDeposit(receiver);
566       if (assets > maxAssets) {
567           revert ERC4626ExceededMaxDeposit(receiver, assets, maxAssets);
568       }
569
570       if (totalSupply() + shares > maxSupply) {
571           revert ExceededMaxSupply(receiver, shares, maxSupply);
572       }
```

Listing 2.32: src/r25-pc/BaseVault.sol

Suggestion Check the shares before calculating the required asset amounts.

2.2.11 Revise the incorrect parameters

Status Confirmed

Introduced by Version 2

Description In contract SyncVault, the error ExceededMaxSupply defines its second parameter as assets, while the functions deposit() and mint() actually pass shares.

```
190   error ExceededMaxSupply(
191       address receiver,
192       uint256 assets,
193       uint256 maxSupply
194   );
```

Listing 2.33: src/r25-pc/SyncVault/SyncVault.sol

Suggestion Rename the error parameter from assets to shares.

2.2.12 Revise incorrect annotations

Status Confirmed

Introduced by Version 2

Description There are some annotations inconsistent with implementations, as follows:

1. In the contract SyncVault, the function maxRedeem() states that it returns the user's total balance, while it actually returns only unlocked shares.

2. In the contract SyncVault, the function handleOracleReport() states that it updates total managed assets, while it actually updates latestNav and related accounting state.

3. Some minor documentation issues remain. The field ReportData.submitTimestamp in the contract SyncVault is defined but never read. The license header in SyncVault contains the typo "govered".

```
438    * @dev Returns the user's total balance
```

Listing 2.34: src/r25-pc/SyncVault/SyncVault.sol

```
863    * @dev Updates total managed assets based on reported NAV, enforces sequential reporting
```

Listing 2.35: src/r25-pc/SyncVault/SyncVault.sol

```
296   struct ReportData {
297       // Off-chain report accounting timestamp
298       uint64 accountingTimestamp;
299       // Off-chain report submission timestamp
300       // depends on Oracle's reporting settlement completion
301       uint64 submitTimestamp;
```

Listing 2.36: src/r25-pc/SyncVault/SyncVault.sol

```
3// Use of this software is govered by the Business Source License included in the LICENSE.TXT file
      and at r25.xyz/busl11.
```

Listing 2.37: src/r25-pc/SyncVault/SyncVault.sol

Suggestion Revise the annotations above so that they match the actual implementation.

2.3 Note

2.3.1 Token assumption

Introduced by Version 2

Description This project only supports assets with a decimal place hardcoded to 6. Specifically, the contract SyncVault enforces a 6-decimal asset during initialization, and the share token decimals are also fixed to 6. Meanwhile, the project assumes that the underlying asset follows standard ERC20 semantics with exact nominal accounting. Therefore, non-standard tokens (e.g., Fee-on-Transfer tokens or rebasing tokens) are outside the supported scope. If such tokens are used, the related deposit, withdrawal, redemption, and accounting flows may deviate from the intended behavior and produce unexpected results.

2.3.2 Different effect time of personal quota

Introduced by Version 2

Description In contract SyncVault, function updatePersonalDailyWithdrawQuota() allows updating the daily instant withdrawal quota for users.

However, according to function getUserDailyWithdrawQuota(), the quota will take effect at different times among users. Specifically, users who have already consumed their quota today keep their current remaining quota until the next day's reset. Other users get a new quota immediately when function getUserDailyWithdrawQuota() or _updateUserDailyQuota() is invoked.

2.3.3 Inconsistency among vault logic due to implementation updates

Introduced by Version 2

Description In contract SyncVaultFactory, function setVaultSyncLogic() allows modifying the implementation logic of vault contracts. However, only vaults deployed after this change adopt the updated logic, leaving inconsistencies with previously deployed vaults. Therefore, the project should carefully perform upgrades to existing vaults via privileged roles if all vault instances should retain the same implementation logic.

2.3.4 Deviations from the EIP-4626 standard

Introduced by Version 2

Description The contract SyncVault deviates from the EIP-4626 specification as a deliberate design decision. It introduces a state variable isReceiverEqSender to control whether the receiver must equal the caller (or owner) for certain operations. When enabled, function deposit() and mint() enforce receiver to be _msgSender(), while function redeem() and withdraw() enforce receiver to be the owner. Neither pair supports a distinct receiver address when this constraint is active, which may conflict with the EIP-4626 standard. Integrators should account for these constraints accordingly.

2.3.5 Centralized withdrawal mechanism

Introduced by Version 2

Description In this project, the vault does not hold user assets. Upon deposit, user assets are forwarded from the vault to the `custodian` rather than retained in the vault. Successful withdrawals depend on the separate `payer` address maintaining sufficient liquidity and granting adequate allowance to the vault. This creates a single point of operational dependency, and the funds and approval configuration must therefore be managed with strict care at all times.

2.3.6 Ensure the correctness and timeliness of Oracle reports

Introduced by Version 2

Description In contract `SyncVault`, the `ORACLE_REPORTER` can invoke the function `handleOracleReport()` to submit updated NAV data. Sequential report validation is only enforced after both `lastAccountingTimestamp` and `lastReportId` are non-zero, and NAV deviation checks apply only after a non-zero NAV has been submitted. Additionally, since NAV updates are applied asynchronously on-chain, a front-running attack may arise where an attacker can observe a pending NAV-increasing report transaction in the mempool. The attacker could deposit before the report is executed to mint shares at the old NAV, and then profit immediately from the increased NAV after execution.

Therefore, the project must ensure that `ORACLE_REPORTER` submits reports in a timely manner to preserve correct NAV-based conversion, and configure fees and lock-up periods appropriately to minimize the impact of front-running attacks.

Feedback from the project The project stated that the impact of front-running is acceptable.

2.3.7 Share token transfer restriction

Introduced by Version 2

Description Contract `SyncVault` overrides the function `_update()` but does not apply the modifiers `whenNotPaused` and `onlyWhitelisted` to the function. This means that the transfer operation is not subject to emergency pause mechanisms and whitelist access controls.

```
1329    function _update(address from, address to, uint256 value) internal override {
```

Listing 2.38: src/r25-pc/SyncVault/SyncVault.sol

2.3.8 Potential centralization risks

Introduced by Version 2

Description In this project, several privileged roles (e.g., `DEFAULT_ADMIN_ROLE`, `CONTROLLER`, `ORACLE_REPORTER_ROLE`, and the upgrade operators in `TransparentUpgradeDeployer`) can conduct sensitive operations, which introduces potential centralization risks. For example, the `DEFAULT_ADMIN_ROLE` can grant any role to any address, including itself, giving it ultimate control over all role assignments. The `CONTROLLER` role can redirect deposited assets by changing

the `custodian`, redirect withdrawal liquidity by changing the `payer`, widen or disable NAV fluctuation checks via `maxFluctuationBps`, pause all vault operations, and selectively skip or fulfill queued withdrawal requests. The `ORACLE_REPORTER_ROLE` directly controls NAV updates that determine share valuation, and since `CONTROLLER` can widen the fluctuation threshold arbitrarily, the Oracle restraint is ultimately discretionary. Upgrade operators can replace all vault logic via the transparent proxy pattern. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS