Code Assessment

of the Yelay Lite Smart Contracts

Mar 24, 2025

Produced for

yelay

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Yelay team,

Thank you for trusting us to help Yelay with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Yelay Lite according to Scope to support you in forming an opinion on their security risks.

Yelay implements a dedicated vault system that directs all yield into a yield extractor. Users will be rewarded outside of the protocol from the respective clients. The vault is for approved projects only.

The code is well structured and implements an upgrade architecture similar to the diamond proxy upgrade pattern. The most critical subjects covered in our audit are functional correctness and arithmetic correctness. The most severe issues is an incorrectly calculated redeem (Incomplete fund transfer when withdrawing) and a double counted balance when swapping (double-counting in swap). All issues were addressed and resolved if necessary. We advised to increase the test suite as the issues could have been caught by e.g., testing redeems with strategies that partially fulfill the request.

The team was always very responsive and was clarifying all questions quickly and professionally. In summary, we find that the current codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Yelay Lite repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	10 Jan 2025	4ecadb6b796e54799a646ee5d7b2f0b3746a1d9c	Initial Version
2	4 Feb 2025	4bf9fe5b00555351968772c9aa3473fd3a972ec4	Fix version
3	14 Feb 2025	b7da381440ee9d455005cdf8525b3f97bd547178	Fix version
4	17 Mar 2025	7d19b8bb5fe757109e5a30fa91c032b97a0aebfd	Fix version

For the solidity smart contracts, the compiler version 0.8.28 was chosen.

2.1.1 Excluded from scope

All third-party contracts, mock and test contracts. Especially, we excluded integrated exchanges, tokens and strategies. These integrations must be carefully tested and assessed to work with the system as intended.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

The Yelay light vault is, rather than a generic vault, a specialized vault for one use case. Clients need to be whitelisted, and projects need to be agreed on with Yelay before the vault can be used. Users can deposit to these projects and will receive project specific shares. However, all funds (from all projects) are managed together in the vault and invested together into whitelisted strategies. When these strategies earn a profit, these profits will not go to the investors but into a yield extractor. The investors are compensated outside of this protocol.

For the main system components to work correct and allow user deposits, first, a client and a project id needs to be created and activated. Secondly, there should be strategies activated. Else, the funds will sit idle in the vault. To send the yield to the yield extractor, a yield extractor needs to be defined. Additionally, the relevant functions should not be paused.

2.2.1 Client and Project Management

Clients need to enroll and be created by the contract owner. A client will receive a specific range of project ids. Each project id must be activated before it can be used. When a client with an activated project id exists, anyone can deposit funds into the Yelay lite vault providing a valid project id. The ownership (client data) can be transferred to another address at any time by the current account that has the client data assigned.

A user has the ability to either deposit assets for shares, migrate a position to another project id or redeem his shares for assets.

To deposit a client defines the asset amount to deposit, the project id and the share receiver address. The project id must be activated to deposit. In case the vault asset hasn't been updated and the fee has not been charged on the profit for more than a pre-defined update threshold (`lastTotalAssetsUpdateInterval`), this is done before the asset contribution is converted to shares and transferred into the vault. The user will receive shares proportional to the total assets and the assets will be deposited into the strategies via the adapter contracts. The shares are dedicated shares lined to the project id. A deposit queue determines the order the contract tries to deposit the funds. There are only full, no partial deposits into the strategies supported. In case it fails to deposit into a strategy, the contract will keep the asset tokens and account for the holdings in `underlyingBalance`.

To migrate a position, the user specifies an amount and the project id from and to which the shares shall be migrated. As all assets are managed by the same vault, the project id linked to the shares is only for tracking reasons but has no further impact on share or asset calculations. Hence, the migration will simply change the project id by burning the share tokens allocated to the old project and mint new ones with the updated project id. Additionally, the total asset amounts are updated and the fees charged on potential profits before.

When a user wants to redeem their shares for assets, they need to specify the amount of shares to redeem, the project id and a receiver. First, the total asset amount is updated and potential fees on profit charged. The share amount is converted to assets and the assets withdrawn from the strategies. The withdrawal process will loop over the strategies through a pre-defined withdrawal queue and try to pull the assets from the strategies. The user should receive the amount that could be pulled from the strategies or if nothing could be pulled from the strategies the full amount from the vault (currently there is a severe issue described in Incomplete fund transfer when withdrawing). In the end the user's shares are burned.

2.2.2 Strategy Management

Before a strategy can be activated and used by QUEUES_OPERATOR it needs to be created by the STRATEGY_AUTHORITY. If no strategy is activated, all deposit will idle in the vault. The ManagementFacet offers the functionality for the STRATEGY_AUTHORITY role to add, remove strategies and approve tokens to strategies if no paused. For the QUEUES_OPERATOR role it offers the ability to activate or deactivate a strategy. For each strategy activated or deactivated the deposit queue and withdraw queue order needs to be defined. The queues will determine in which order funds are added or withdrawn from the vault. The queue order can also be updated any time by the QUEUES_OPERATOR.

2.2.3 Fund management

Invested users can at any time (if not paused) chose to deposit more funds, migrate their positions to other project ids or redeem their position. Furthermore, the fund operator can deposit and withdraw funds sitting idle in the vault to strategies or migrate a position to another strategy. The operator is also able to claim rewards and trigger the fee calculation and total asset update. All actions can be paused individually.

2.2.4 Storage Management

Yelay decided to use a custom storage system to maintain upgradability and avoid reaching contract size limits by using a modular approach. Instead of using the pre-defined storage layout each storage slot address is encoded to avoid collisions and defined in one of library contracts. The main function is the getter for the storage pointer that keeps the corresponding information. Some of the library functions also include other convenient getters like the `onlyOwner` function in `LibOwner`. The only contracts that do not define a state with getters are the Error, Events and Roles library. As the names suggest, they define the events, errors and roles used in the system.

2.2.5 Helper Contracts

The `VaultWrapper` and `Swapper` contract are stand-alone helper contracts. They provide convenience functions for users and, in case of the `Swapper`, also the functionality for the `FUNDS_OPERATOR` to swap the rewards in the `FundsFacet`. User can use the wrapper to first swap their tokens to the underlying tokens and then deposit these tokens into the vault. The `VaultWrapper` offers to wrap and deposit ETH and to swap and deposit tokens. In case the user is using exchanges to swap before depositing the tokens, the `Swapper` contract provides the corresponding functionality.

As both contracts are stand-alone contracts, they are simpler and extend OpenZeppelin's `OwnableUpgradeable` pattern for the access permission and the `UUPSUpgradeable` pattern for upgrades. Hence, both contracts have a dedicated owner role that is allowed to change the implementation, transfer and renounce the ownership. Additionally, the owner can set the allowed exchanges for the `Swapper` contract.

2.2.6 Access Control

The `AccessFacet` contains the access control logic for the main contract. The owner can grant and revoke roles. The role logic extends OpenZeppelin's `AccessControlEnumerableUpgradeable` contract and uses `LibRoles` for the role definitions. The role management relies on OpenZeppelin except the `onlyOwner` check is implemented differently via the `LibOwner` contract as it uses the custom storage management (see Storage Management). The pauser role additionally has the ability to pause certain selectors. The function `_checkNotPaused` in the `LibPausableContract` will check the pause flag for each selector such that it can be used in the modifier `notPaused` by the `PausableCheck` contract. All functions in the vault except for the `setPaused` and owner functions can be paused separately. The following functions can be called externally by the indicated roles:

For registered clients:

- `transferClientOwnership`
- `activateProject`

For the `FUNDS_OPERATOR`:

- `setLastTotalAssetsUpdateInterval`
- `managedDeposit`
- `managedWithdraw`
- `reallocate`
- `swapRewards`
- `accrueFee`
- `claimStrategyRewards`

For the `QUEUES_OPERATOR`:

- `updateDepositQueue`
- `updateWithdrawQueue`

- `activateStrategy`
- `deactivateStrategy`

For the `STRATEGY_AUTHORITY`:

- `addStrategy`
- `removeStrategy`
- `approveStrategy`

For the `PAUSER` and `UNPAUSER`:

- `setPaused`

For the owner of the respective contract:

- `grantRole`
- `revokeRole`
- `createClient`
- `setSelectorToFacets`
- `transferOwnership`

For all accounts (users):

- `deposit`
- `redeem`
- `migratePosition`
- `wrapEthAndDeposit`
- `swapAndDeposit`

The two helper contracts have a dedicated owner as they use OpenZeppelin's UUPS proxy architecture. The owner has the power to call:

- `Swapper.updateExchangeAllowlist`
- `Swapper.transferOwnership`
- `Swapper.renounceOwnership`
- `Swapper.upgradeToAndCall`
- `VaultWrapper.transferOwnership`
- `VaultWrapper.renounceOwnership`
- `VaultWrapper.upgradeToAndCall`

2.2.7 Trust Model

The following assumptions are highlighted regarding the system and roles:

- Registered clients are trusted to activate their project as intended. This can be assumed as it is in their own interest.

- Fund operators are fully trusted to behave correctly. They could cause losses for the system by e.g., swapping funds in an unfavorable way or change allocations (reallocate, deposit or withdraw) in an unfavorable way from strategies. Less severe actions is to not claim outstanding rewards or updated the fee and total asset balance when needed.

- The Queue operator is only trusted to activate and deactivate and set the intended order for deposits and withdraws.

- The Strategy authority is fully trusted. This role is critical as it could add malicious strategies and ultimately steal funds.

- The Pauser is trusted to not DOS the system and pause the correct selectors. The Unpauser is trusted to unpause accordingly.

- The Yield extractor is trusted not to steal the yield and not to perform inflation attacks if the vault is empty.

- The contract owners are fully trusted. They have ultimate power and multiple ways to bankrupt the system, e.g., by using other roles like the strategy authority, changing contract implementations or selector to facet routes.

- General users are not trusted. However, general users might cause dust accumulations and incorrect accounting as mentioned in Fund accounting might deviate from real value by (costly) irrational interactions.

Furthermore, we assume:

- The swapper never holds any ETH or tokens as they could be taken by users.

- The vault wrapper should never hold tokens or ETH as they could be taken by users.

- The tokens used in the system have only one entry point. Else, the reward swapping can disrupt the accounting. Furthermore, all tokens are intensively tested and assessed to correctly work with the system before being added.

- All strategies are thoroughly tested and the functionality of the third-party protocols (which are out-of-scope) are tested and assessed to work as intended before being added.

- All exchanges that are allowed to be used are tested and assessed to work correctly with the system.

3   Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security : Related to vulnerabilities that could be exploited by malicious actors
- Correctness : Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical -Severity Findings	0
High -Severity Findings	0
Medium -Severity Findings	0
Low -Severity Findings	1

- Division by Zero in Deposit Acknowledged

5.1 Division by Zero in Deposit

Correctness Low Version 1 Acknowledged

The `_convertToShares()` function will revert if `newTotalSupply` is non-zero and `newTotalAssets` is zero.

This can only occur in an edge case where a loss in one of the strategies wipes out all the assets, and the internal balance is zero. If that is the case, `deposit()` will fail, meaning that the contract is bricked unless every single outstanding share is redeemed in exchange for zero assets such that totalSupply is zero again.

Acknowledged:

Yelay is aware and acknowledges the issue.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

`Critical`-Severity Findings	1

- Possible Re-entrancy When Depositing `Code Corrected`

`High`-Severity Findings	2

- Griefing of Client Data `Code Corrected`
- Incomplete Fund Transfer When Withdrawing `Code Corrected`

`Medium`-Severity Findings	2

- Double-Counting in Swap `Code Corrected`
- abi.decode Can Fail `Code Corrected`

`Low`-Severity Findings	1

- Fund Accounting Might Deviate From Real Value `Code Corrected`

Informational Findings	4

- Duplicated Code `Code Corrected`
- Event `Code Corrected`
- Unnecessary Approval `Code Corrected`
- Unused Import `Code Corrected`

6.1 Possible Re-entrancy When Depositing

`Security` `Critical` `Version 1` `Code Corrected`

CS-YLY-LITE-016

In an independent second audit a missed re-entrancy in the `FundsFacet.deposit` function was uncovered. Through the ERC-1155 `_mint` function and the callback, an attacker could re-enter the deposit function. When re-entering, the `totalSupply` was already updated but not the total assets. Thus, the share allocated to the attacker became overproportionately high. Each iteration could increase the ratio in favor of the attacker. Consequently, an attacker was able to mint more shares than intended.

Code corrected:

The `_mint` function was moved after the asset update to prevent that the attacker can re-enter with an incorrect shares to asset ratio.

6.2 Griefing of Client Data

`Security` `High` `Version 1` `Code Corrected`

`transferClientOwnership` overwrites a location in the `ownerToClientData` mapping without checking if it is in use. This means that a malicious client can pass the address of another client and override their data.

Code corrected:

Yelay prevented the override by checking if the new address already has a client struct associated.

6.3 Incomplete Fund Transfer When Withdrawing

Correctness **High** **Version 1** **Code Corrected**

When withdrawing funds, the intended behavior is, to first try to withdraw the desired amount form the strategies in the queue. If funds are missing, the remainder is tried to be covered by the idle funds sitting in the vault. The implementation calculates the remaining funds to be covered by the vault by:

```
toReturn = assets == _assets ? assets : assets - _assets;
sF.underlyingAsset.safeTransfer(receiver, toReturn);
_burn(msg.sender, projectId, shares);
```

`assets` is the required amount and `_assets` is the remaining desired amount that could not be withdrawn from the strategies. `shares` is the amount of shares corresponding to `X` assets. There are now three possible scenarios:

1. We could cover the desired amount `X` by withdrawing from the strategies and do not need any additional funds from the vault. Hence, `assets = X` and `_assets = 0`. `toReturn` would be `X` and the user would receive the whole desired amount in the next transfer. The corresponding shares would be burned.

2. We could recover nothing from the desired amount by withdrawing from the strategies. Thus, `assets = X` and `_assets = assets`. `toReturn` would be `assets`. We would try to cover the whole amount with funds from the vault.

3. It was only possible to recover a fraction of the desired funds and we want to cover for the remaining funds with assets from the vault. This scenario was a dedicated provided specification. Let's assume `p` is the percentage we were able to recover from the desired balance with the strategies. In this case `assets = X` and `_assets = X * (1-P)`. `toReturn` would in this case be `X*P`.

In numbers the last example might be that the user wants to withdraw `100`. The strategies can only cover for `10` (`_assets = 90`). `toReturn` is in this case `10`. The transfer will in the end only transfer `10` tokens to the user and burn all shares corresponding to `100`. In this case, we also subtract `90` from `underlyingBalance`, which is inconsistent.

Code corrected:

In **Version 2**, the vault tries to cover the shortfall using its internal balance. If it cannot, the transaction reverts. As long as more than `WITHDRAW_MARGIN` wei are covered by the vault, the `underlyingBalance` is correctly updated.

6.4 Double-Counting in Swap

`Correctness` `Medium` `Version 1` `Code Corrected`

The `Swapper.swap()` function counts the amount of tokens it obtained by checking the swapper's balance. However, it adds together the successive balances in `tokenOutAmount`, meaning it would double-count tokens from the first swap if there is a second one, leading to a revert in `safeTransfer()`.

Code corrected:

The `tokenOutAmount` is now set to the token balance of the contract and not accumulated anymore.

6.5 `abi.decode` Can Fail

`Correctness` `Medium` `Version 1` `Code Corrected`

`redeem()` tries to decode the returndata from `IStrategyBase.withdraw()` even if the function reverted. If the function reverted, returndata would contain a revert reason of arbitrary length, which may not cleanly decode to an uint. This will cause the entire redemption to fail or other unintended behavior.

Code corrected:

The abi decoding is only done in the branch executed if the `withdraw()` call was successful.

6.6 Fund Accounting Might Deviate From Real Value

`Correctness` `Low` `Version 1` `Code Corrected`

In `fundsFacet.redeem()` the `underlyingBalance` is only updated if the assets exceed `WITHDRAW_MARGIN`. In case the vault has some underlying assets, a user could specify the share amount to redeem such that it will convert to `WITHDRAW_MARGIN` (or less). This will immediately break the loop and not withdraw from the strategies. Finally, it will use the vault's balance and transfer the amount to the user and burn the corresponding shares. Currently, `WITHDRAW_MARGIN` is set to `10` and the impact is marginal even if it accumulates over time. Yet, the accounting will be slightly off.

Code corrected:

In `Version 3`, the `underlyingBalance` is correctly updated in all cases, fixing the issue. In addition, redeem requests of less than `WITHDRAW_MARGIN` in assets will revert.

6.7 Duplicated Code

`Informational` `Version 1` `Code Corrected`

The `LibClients` contract defines the function `_isProjectActive`. A copy of this function is also implemented in the `ClientsFaucet` contract named `projectIdActive`.

Code corrected:

The function in `ClientsFaucet` now uses the function from `LibClients`.

6.8 Event

`Informational` `Version 1` `Code Corrected`

Most relevant setters or state changes emit an event. However, `FundsFacet.setLastTotalAssetsUpdateInterval` does not. Yelay might evaluate if this is needed.

Code corrected:

An event was added to the function.

6.9 Unnecessary Approval

`Informational` `Version 1` `Code Corrected`

In `AaveV3Strategy.onAdd`, it is not necessary to explicitly approve the Aave pool in order to redeem ATokens according to this information.

Code corrected:

The superfluous approval has been removed.

6.10 Unused Import

`Informational` `Version 1` `Code Corrected`

In `LibClients`, `ClientsFacet.sol` is imported. The imported definition is unused, and the import constitutes a circular import.

Code corrected:

The unused import has been removed.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Event Ordering

`Informational` `Version 1`

In case of re-entrancy, the events might not be emitted in the same order as the original call. Re-entrancy might be possible through the ERC-1155 standard, callbacks in exchanges or other integrated third party contracts.

7.2 Gas Costs

`Informational` `Version 1`

The current code structure uses mostly storage pointer. Hence, most variable modifications or reads are state operations. Gas could be saved in cases if there are reoccurring reads or writes to the same storage pointer by caching the relevant values in memory. However, attention needs to be paid to not load whole objects into memory if not needed.

7.3 Sanity Checks

`Informational` `Version 1`

- `createClient` and `transferClientOwnership` do not check the `clientOwner` for address zero.
- `StrategyData` does not validate `adapter` for address zero.

7.4 Swap Token Spender Must Be the Same as Entry Point

`Informational` `Version 1`

The `Swapper` always gives the approval to the same contract that it then calls to execute the swap. Some protocols such as Paraswap V5 (but not V6) require a different contract to be approved than the one that receives the call. The change log is documented here and including the corresponding code.

7.5 Withdraw Priority

`Informational` `Version 1`

Usually, withdrawing from strategy has a certain risk that fees or other losses might occur. In case the vault has funds sitting idle not in strategies, it might be beneficial to use these funds first for a withdraw. Currently, only in case when the strategies could not cover the desired amount, the funds sitting idle are used to eventually cover the remaining amount of a withdraw. However, it might be that certain business requirements make the implemented priority necessary.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Max Approval to Exchanges

Note **Version 1**

Currently, the system grants max approval to the integrated exchanges. In theory this might be problematic if one of the exchanges becomes malicious. Even if it is not whitelisted anymore, it still has access to the max allowance.

8.2 User Might Receive a Few WEI Less Than Expected

Note **Version 1**

When redeeming, a user might receive up to `WITHDRAW_MARGIN-1` wei less even if the fund could cover for the full request. This might happen if the difference between the funds withdrawn and the funds requested is smaller than `WITHDRAW_MARGIN`.