Blaize.Security

October 24th, 2023 / V. 1.0

FORTIFI

SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

SCORE **9.6**/10

The scope of the project encompasses the FortiFi smart contract repository:
contracts\fee-calculators\FortiFiFeeCalculator.sol
contracts\fee-managers\FortiFiFeeManager.sol
contracts\strategies\FortiFiDPFortress.sol
contracts\strategies\FortiFiDPStrategy.sol
contracts\strategies\FortiFiFortress.sol
contracts\strategies\FortiFiStrategy.sol
contracts\strategies\FortiFiVectorFortress.sol
contracts\strategies\FortiFiVectorStrategy.sol
contracts\vaults\FortiFiMASSVault.sol
contracts\vaults\FortiFiSAMSVault.sol

Repository: https://github.com/0xFortiFi/FortiFi-Vaults

Branch: main

Initial commit:

- 59f05234105dd30f3fb726e1e380aaf6a072b72d

Final commit:

- 3c6a0709427caa12ce6427965be34b20cbdcbb2c

TECHNICAL SUMMARY

At Blaize.Security, we conducted a security audit of the **FortiFi** smart contracts between **October 3rd and 24th, 2023**. We aimed to identify and describe any security issues in the platform's smart contracts. This report presents our findings and recommendations.

Testable code

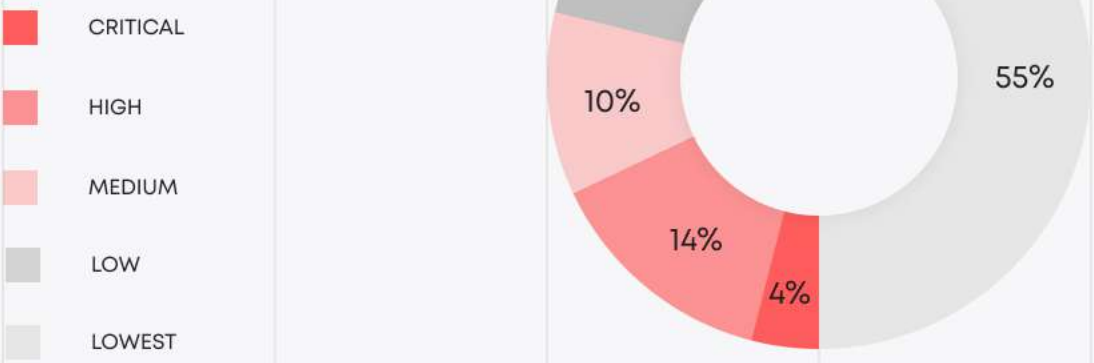
Auditors approved code as testable within the industry standard.

The audit scope encompasses all tests, scripts, documentation, and requirements the **FortiFi** team provides. The coverage is determined based on the Hardhat framework's tests and scripts, additional testing strategies, and testable code from manual and exploratory rounds.

However, toto ensure the security of the contract, the **Blaize.Security** team recommends the following post-audit steps:

1. launch **active protection** over the deployed contracts to have a system of early detection and alerts for malicious activity. We recommend the AI-powered threat prevention platform **VigiLens**, by the **CyVers** team.
2. launch a **bug bounty program** to encourage further active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. A total of 29 problems were found. 27 issues were fixed or verified by the FortiFi team. Two issues were not resolved.

	FOUND	FIXED/VERIFIED
Critical	1	1
High	4	4
Medium	3	3
Low	5	5
Lowest	16	14

SEVERITY DEFINITION

Critical

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

High

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

Medium

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

Low

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

Lowest

The system does not contain any issues critical to the secure work of the system, yet is relevant for best practices

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

Blaize.Security auditors start the audit by developing an **auditing strategy** - an individual plan where the team plans methods, techniques, approaches for the audited components. The strategy includes the following set of activities::

Manual audit stage

- Manual line-by-line code by at least 2 security auditors with crosschecks and validation from the security lead;
- Protocol decomposition and components analysis with building an interaction scheme, depicting internal flows between the components and sequence diagrams;
- Business logic inspection for potential loopholes, deadlocks, backdoors;
- Math operations and calculations analysis, formula modeling;
- Access control review, roles structure, analysis of user and admin capabilities and behavior;
- Review of dependencies, 3rd parties, and integrations;
- Review with automated tools and static analysis;
- Vulnerabilities analysis against several checklists, including internal Blaize.Security checklist;
- Storage usage review;
- Gas (or tx weight or cross-contract calls or another analog) optimization;
- Code quality, documentation, and consistency review.

For advanced components:

- Cryptographical elements and keys storage/usage audit (if applicable);
- Review against OWASP recommendations (if applicable);
- Blockchain interacting components and transactions flow (if applicable);
- Review against CCSSA (C4) checklist and recommendations (if applicable);

Testing stage:

- Development of edge cases based on manual stage results for false positives validation;
- Integration tests for checking connections with 3rd parties;
- Manual exploratory tests over the locally deployed protocol;
- Checking the existing set of tests and performing additional unit testing;
- Fuzzy and mutation tests (by request or necessity);
- End-to-end testing of complex systems;

In case of any issues found during audit activities, the team provides detailed recommendations for all findings.

EXECUTIVE SUMMARY

The Blaize Security team received a set of smart contracts from the FortiFi team. These contracts constitute the FortiFi Vaults Ecosystem, which is designed to deposit users' funds into a variety of underlying strategies. This system provides several layers of asset isolation within the FortiFi framework.

This audit aimed to confirm the correct functionality of these contracts and ensure they adhered to a known security level. Our auditors meticulously scrutinized each line of code, cross-referencing it against a checklist of potential vulnerabilities. They verified the business logic of the contracts, ensuring adherence to best practices, particularly in relation to gas expenditure. Any identified issues were swiftly communicated to the FortiFi team for resolution or verification. The setup and deployment scripts of the contracts were also subjected to rigorous auditing.

During the manual audit, we identified a total of 29 issues. This included one critical-severity issue, four high-severity issues, three medium-severity issues, five low-severity issues, and sixteen informational-severity issues. The critical severity issue was related to a potential front-running attack as swaps in the MASS vault as price was not properly validated. This issue was promptly addressed and resolved by the FortiFi team.

During the second iteration of the audit, we identified one high-severity issue related to the use of a deprecated Chainlink API call. Another high-severity issue was due to a potential mismatch between the strategy's token decimals and the Oracle's decimals. The remaining high-severity issues were associated with access control problems. All four high-severity issues were successfully resolved by the FortiFi team throughout four audit iterations.

The medium-severity findings were attributed to the absence of safety checks in transfers, the owner's ability to add duplicate strategies, and unvalidated prices returned by the Oracle. All three issues have been successfully resolved.

The low-severity findings were related to issues with adjusting contract settings, floating pragma, and the inability to set a custom route for swaps in the MASS Vault. Most of the informational issues were due to unoptimized code and business logic problems. These findings were discussed with the FortiFi team over several rounds of communication, and nearly all of them have been resolved, with the exception of two informational issues.

During the testing phase of our audit process, we reviewed the native tests prepared by the FortiFi team and developed a set of custom test scenarios. The development team provided only a handful of tests for the MASS Vaults. As a result, Blaize Security supplemented this with our own unit tests and additional scenarios to thoroughly cover the complex functionality of the FortiFi Vaults Ecosystem contracts. The comprehensive set of unit tests can be found in the Code Coverage and Test Result sections.

The issues regarding the MASS Vault complexity and strategies utilization were addressed to the FortiFi team. The FortiFi Vaults represent a complex smart contract system with multiple varying factors. The strategy management was discussed with the FortiFi team, and several suggestions for optimizing the MASS Vault were communicated.

	RATING
Security	9.8
Logic optimization	9.2
Code quality	9.4
Test coverage**	9.7
Total	9.6

**The FortiFi Vaults Ecosystem had a minimum viable native unit-test coverage. To achieve sufficient coverage and validate the business logic, the Blaize Security team wrote almost all the tests conducted during the audit.

PROTOCOL OVERVIEW

The FortiFi protocol is a complex system designed to optimize returns on deposited assets. Its core is built around **FortiFiSAMSVault**, which allows users to deposit their assets. Once deposited, these assets are spread across multiple yield-generating strategies. Vault, for almost all strategies, do not directly interacts with yield platforms, it delegates this task to various strategy contracts. Each strategy knows how to maximize returns on a specific platform or set of platforms. As the DeFi landscape changes, the owner of the vault can adapt by adding, removing, or modifying these strategies. Likewise, SAMS Vaults utilize **FortiFiFeeCalculator** and **FortiFiFeeManager** contracts to calculate and collect performance fees.

MASS Vaults allow for the deposit of a single token, which is then divided, swapped into other assets if necessary, and then deposited into sub-strategies, including **FortiFiStrategy** strategies and SAMS Vaults. Functioning similarly to ETF tokens, the underlying assets of MASS Vaults yield returns. MASS Vaults utilize FortiFiFeeCalculator and FortiFiFeeManager contracts to calculate and collect performance fees.

FortiFiFortress contracts are utilized to isolate users' receipt tokens, a necessary step for strategies that feature unique deposit or withdrawal functions. These Fortress contracts are deployed by FortiFiStrategy contracts and can only be accessed by the contract that deploys them. A fortress is established for each vault receipt token (1155) used for depositing into the FortiFiStrategy.

The FortiFiFeeCalculator contracts, used by both SAMS and MASS Vaults, are designed to calculate performance fees based on a user's NFT holdings as specified upon deployment. The FortiFiFeeManager contracts, also utilized by SAMS and MASS Vaults, serve to distribute performance fees among one or more addresses.

Roles & Responsibilities

FortiFiFeeCalculator.sol

1. Owner:

The owner has the authority to set new values for NFT contracts, threshold amounts, and threshold basis points using the setFees function. It is the owner's responsibility to verify that the provided NFT contract addresses are valid and that the token amounts and threshold basis points arrays are consistent and meet the specified criteria. Additionally, the owner can determine whether or not the NFT holdings should be combined using the setCombine function.

FortiFiFeeManager.sol

1. Owner:

The owner has the privilege to set new fee split configurations by specifying new receivers and their corresponding split percentages. Additionally, the owner can recover any ERC20 tokens that may have become stuck in the contract.

FortiFiFortress.sol

1. Owner:
   The owner is the FortiFiStrategy contract that creates this Fortress.
   The owner has the exclusive privilege to deposit and withdraw from the vault using the deposit and withdraw functions.
   The owner can recover stuck tokens using the recoverERC20 function.
   The owner can refresh the approval of the deposit token to the underlying strategy using the refreshApproval function.

FortiFiStrategy.sol

1. Owner:
   The owner has the privilege to set valid vaults using the setVault function.
   The owner can recover stuck tokens using the recoverERC20 function.
   The owner can recover stuck tokens from a Fortress using the recoverFromFortress function.

2. FortiFi Vaults:
- FortiFi Vaults are allowed to deposit to and withdraw from the FortiFiStrategy. They interact with the depositToFortress and withdrawFromFortress functions.
- Vaults are specified and validated by the isFortiFiVault mapping.

3. FortiFi Fortress:
- The FortiFiFortress contracts are created by the FortiFiStrategy to interact with the underlying strategy for a specific vault receipt token. This allows user deposits to be isolated.
- The FortiFiFortress contracts have the privilege to deposit and withdraw from the underlying strategy.

FortiFiMASSVault.sol

1. Owner:
- The owner has the privilege to set the minimum deposit amount, slippage, fee manager, fee calculator, and strategies.
- The owner can pause and unpause the contract.

2. Depositors:
- Depositors can deposit a single asset, which is then swapped into various assets and deposited into multiple yield-bearing strategies.
- Depositors can add to their deposit without needing to burn/ withdraw first.
- Depositors can withdraw their assets and receive the underlying strategy receipt tokens.
- Depositors can rebalance their receipt token's underlying assets.

3. Strategies:
- Strategies are the underlying yield-bearing contracts where the vault deposits funds to earn returns. The vault interacts with these strategies to deposit and withdraw funds.

4. Fee Manager and Fee Calculator:
- The fee manager collects fees from the vault.
- The fee calculator calculates the fees due for a withdrawal.

FortiFiSAMSVault.sol

1. Owner:
- Has the ability to set the minimum deposit amount (setMinDeposit).
- Can set the fee manager (setFeeManager).
- Can set the fee calculator (setFeeCalculator).
- Can specify which addresses are exempt from fees (setNoFeesFor).
- Can pause or unpause the contract (flipPaused).
- Can recover stuck ERC20 tokens (recoverERC20).
- Can refresh approvals for strategies (refreshApprovals).
- Can set the strategies used by the vault (setStrategies).

2. User:
- Can deposit assets into the vault (deposit).
- Can add more assets to an existing deposit (add).
- Can withdraw their assets and any profits (withdraw).
- Can rebalance their assets across strategies (rebalance).

List of valuable assets

FortiFiFeeCalculator.sol

1. NFT Contracts (nftContracts): An array of addresses representing the NFT contracts. These contracts are used to determine the user's NFT holdings when calculating fees.

FortiFiFeeManager.sol

1. Receivers: These are the addresses that receive a portion of the fees collected. They play a crucial role in determining who benefits from the protocol-generated fees.

FortiFiFortress.sol

1. Deposit Token (_dToken): Represents the tokens that users deposit into the vault. The balance of this token can vary based on deposits and withdrawals.
2. Wrapped Native Token (_wNative): This represents the wrapped version of the native token (e.g., WETH for Ethereum). The contract might hold some balance of this token, especially following interactions with the underlying strategy.
3. Strategy Receipt Tokens: After a deposit, the vault receives these tokens from the underlying strategy. They represent the vault's share in the strategy. The balance of these tokens in the vault indicates the amount of assets the vault has in the strategy.
4. Users: These are individuals who interact with the contract to deposit or withdraw funds. They are beneficiaries of the vault's operations and can receive refunds of leftover tokens or native currency.

FortiFiStrategy.sol

1. Deposit Token (_dToken): Represents the token that users deposit into the strategy. The balance of this token can fluctuate based on deposits and withdrawals.
2. Wrapped Native Token (_wNative): This represents the wrapped version of the native token (e.g., WETH for Ethereum). The contract might hold some balance of this token, especially following interactions with the underlying strategy.

3. Strategy Receipt Tokens: These tokens are minted by the FortiFiStrategy contract when users make a deposit, representing the user's share in the strategy.
4. Users: These are individuals who interact with the contract to deposit or withdraw funds, and are the beneficiaries of the strategy's operations.
5. Fortresses (vaultToTokenToFortress): These are individual FortiFiFortress contracts created for each user's deposit. They are mapped to a specific vault and token ID. Each fortress represents a user's isolated interaction with the underlying strategy.

FortiFiMASSVault.sol

1. Deposit Token: Represents the token that users deposit into the vault. The balance of this token can fluctuate based on deposits and withdrawals.
2. Wrapped Native Token: This represents the wrapped version of the native token (e.g., WETH for Ethereum). The contract might hold some balance of this token, especially after interactions with the strategies.
3. Receipt Tokens: ERC1155 tokens minted by the FortiFiMASSVault contract when users deposit. They represent the user's share in the vault.
4. Strategies: These are underlying yield-bearing contracts where the vault deposits funds. The vault can have multiple strategies, each with its own deposit token, router, and allocation percentage.

FortiFiSAMSVault.sol

1. Deposit Token (depositToken): Primary asset that users deposit into the vault.
2. Wrapped Native Token (wrappedNative): Represents a wrapped version of the native blockchain asset (e.g., WETH for Ethereum).
3. ERC1155 Receipt Tokens: These non-fungible tokens are minted for users when they deposit assets, representing both the user's deposit and any associated profits.

Settings

FortiFiFeeCalculator.sol

1. Token Amounts (tokenAmounts): An array of uint8 values representing the number of NFTs a user holds. These values are used in conjunction with thresholdBps to determine the fee percentage based on the user's NFT holdings.
2. Threshold Basis Points (thresholdBps): An array of uint16 values representing the fee percentages. The fee percentage applied to a user is determined based on their NFT holdings and the corresponding value in this array.
3. Combine NFT Holdings (combineNftHoldings): A boolean value that determines whether the user's NFT holdings across all contracts in the nftContracts array should be combined when calculating fees.
4. BPS Constant: A constant value representing the basis points used for fee calculations. This is a fixed value and is set to 10,000.

FortiFiFortress.sol

1. Underlying Strategy (_strat): Immutable setting that points to the address of the underlying strategy where the vault deposits funds to earn returns.
2. Minimum balance for fee: The contract ensures that it holds at least 1000 wei of the deposit token before disbursing fees. This is to prevent failures when splitting the amount amongst multiple receivers.
3. Approval for deposit token: Contract maintains an approval for the deposit token to the underlying strategy. This approval can be refreshed using the refreshApproval function.

FortiFiStrategy.sol

1. Underlying Strategy (_strat): Immutable setting that points to the address of the underlying strategy where the FortiFiStrategy deposits funds to earn returns.
2. FortiFi Vaults (isFortiFiVault): Determines which vaults are allowed to interact with the FortiFiStrategy.

3. Mapping of Vaults to Fortresses (vaultToTokenToFortress): Links a specific vault and token ID to a FortiFiFortress contract. It allows the strategy to know which fortress to interact with for a given vault and token ID.

FortiFiMASSVault.sol

1. Strategies: The vault can have multiple strategies. Each strategy has its own deposit token, router, and allocation percentage. The owner can set these strategies.
2. Minimum Deposit: Minimum amount a user must deposit. The owner can set this value.
3. Slippage: Slippage percentage used in swap functions. The owner can set this value.
4. Fee Manager and Fee Calculator

- The fee manager collects fees from the vault.
- The fee calculator calculates the fees due for a withdrawal.

5. Paused State: Contract can be paused or unpaused. While paused, users cannot deposit or add funds. Withdrawals are not restricted, which is paramount for users' funds ownership. The owner can flip the paused state.

FortiFiSAMSVault.sol

1. Name (name): A descriptive name for the vault.
2. Symbol (symbol): A short symbol representing the vault.
3. Minimum Deposit (minDeposit): The minimum amount a user can deposit into the vault.
4. Basis Points (BPS): A constant representing 10,000 basis points (or 100%).
5. Pause State (paused): Indicates whether the contract is paused. When paused, most of user interactions are restricted, but withdrawals are not restricted, which is paramount for users' funds ownership.
6. Fee Calculator (feeCalc): An external contract that calculates fees.
7. Fee Manager (feeMgr): An external contract that manages and distributes fees.
8. Strategies (strategies): An array of strategies that the vault uses to earn yield on deposited assets.

Deployment script

FortiFiFeeCalculator.sol

There are two deployment scripts where FortiFiFeeCalculator instance is created for two different assets BTCb and USDC. Contracts are deployed via standard deploy() method of hardhat without any post-deployment setup.

FortiFiFeeManager.sol

Contract is deployed via the standard deploy() method of hardhat without any post-deployment setup. The arguments are receivers array and splitBps array, which are single-element arrays as of deployment setup.

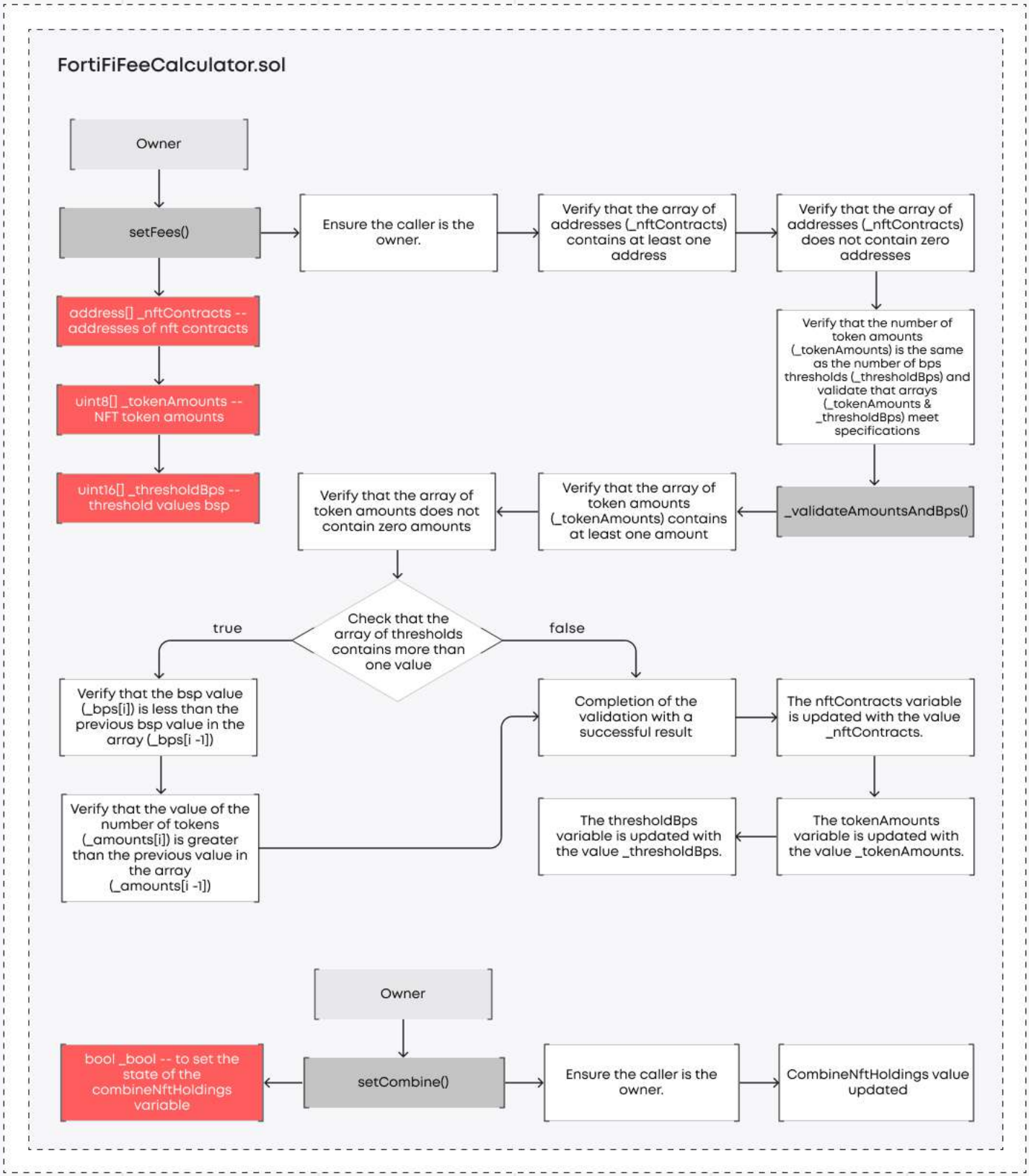
Strategies deployment

Strategies are deployed via the standard deploy() method of hardhat without any post-deployment setup. All strategies' deployments are unified. Arguments in deploy are strategy address, deposit token address and native currency address.

Potential threats

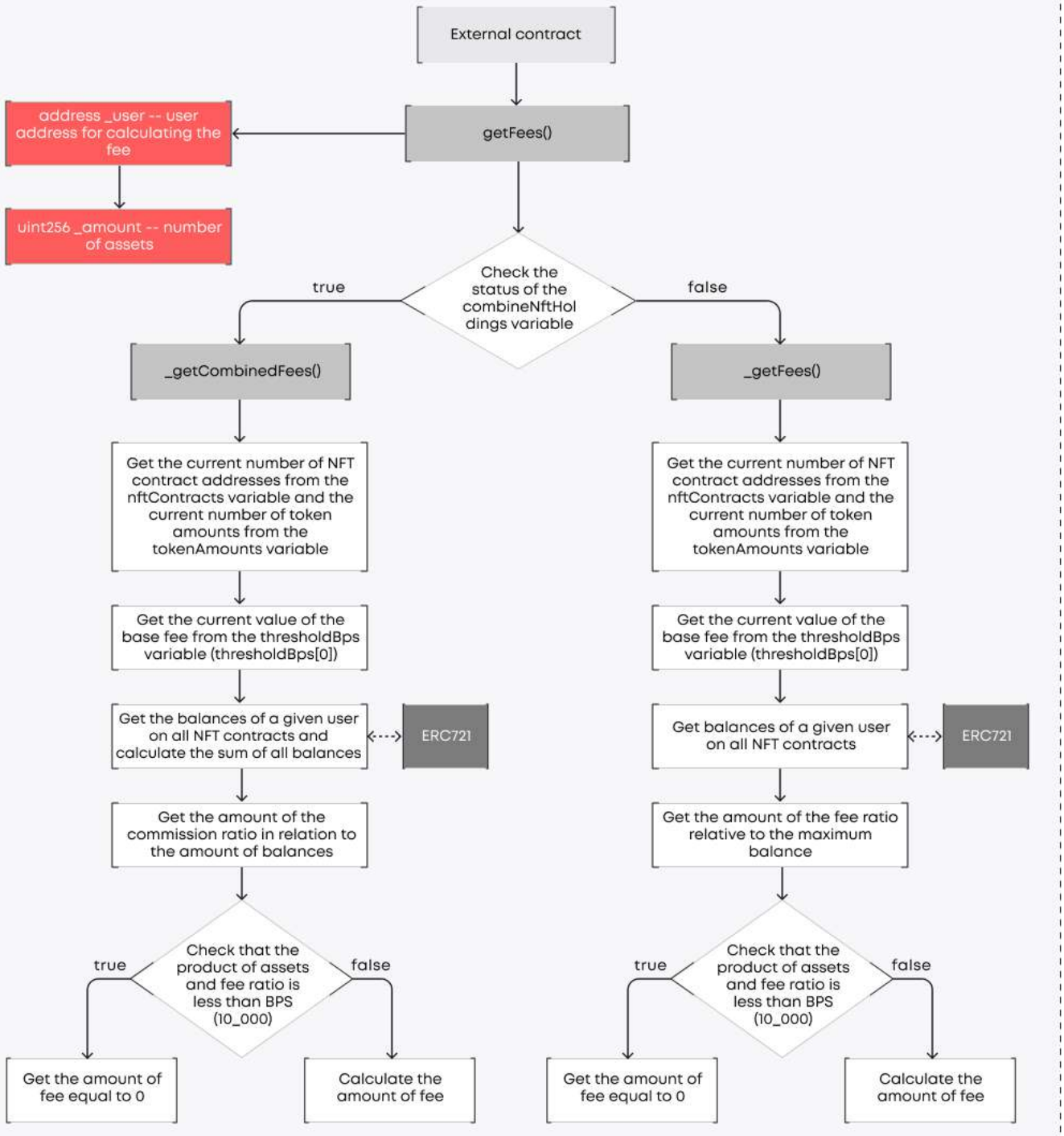
- **Inconsistent usage of underlying strategies:** The strategies' utilisation is handled with complex logic. Mostly, FortiFi Vault interacts with underlying protocols via multiple levels of funds isolation, which is important also for ensuring the logic of strategy is working in the expected and unified way. However, there are cases for strategies which are supposed to be used without these layers of isolation (for example, Yield Yak strategy). That might lead to unexpected behaviour. While auditors validated usage of Yield Yak strategy, if there will be need for adding another protocols for direct usage, it should be subjected to rigorous audit.

FORTIFI SCHEME

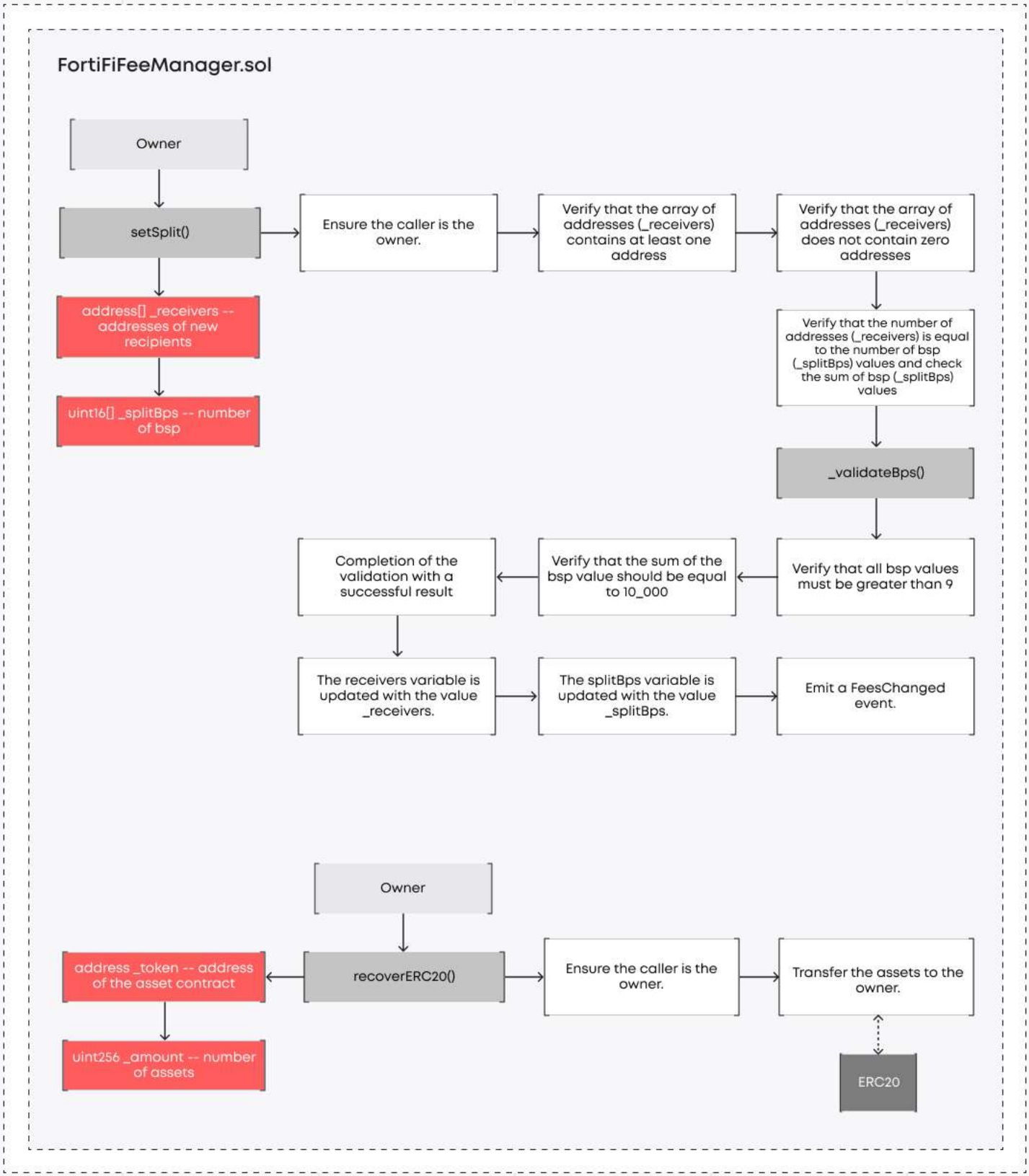


FORTIFI SCHEME

FortiFiFeeCalculator.sol

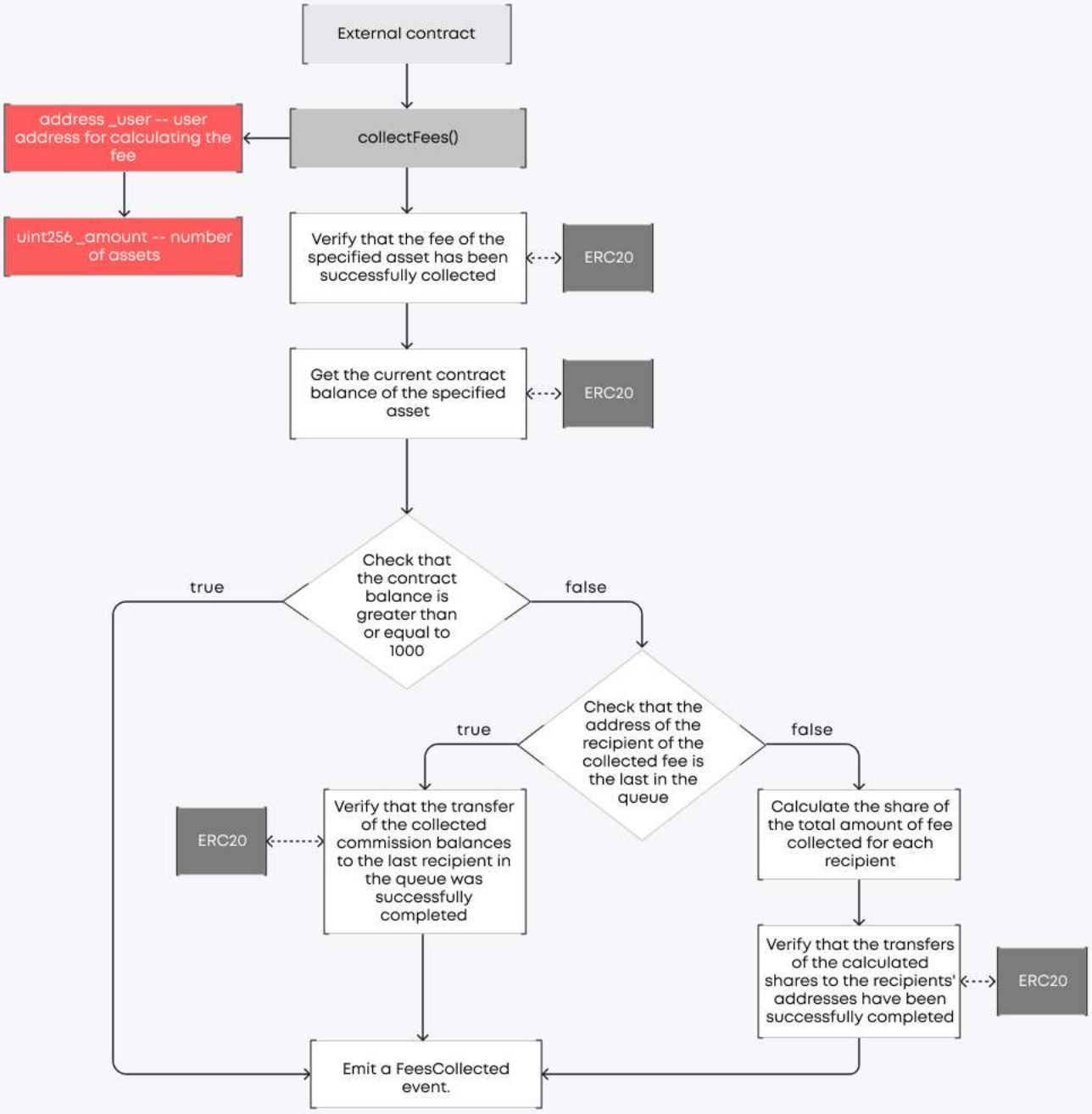


FORTIFI SCHEME



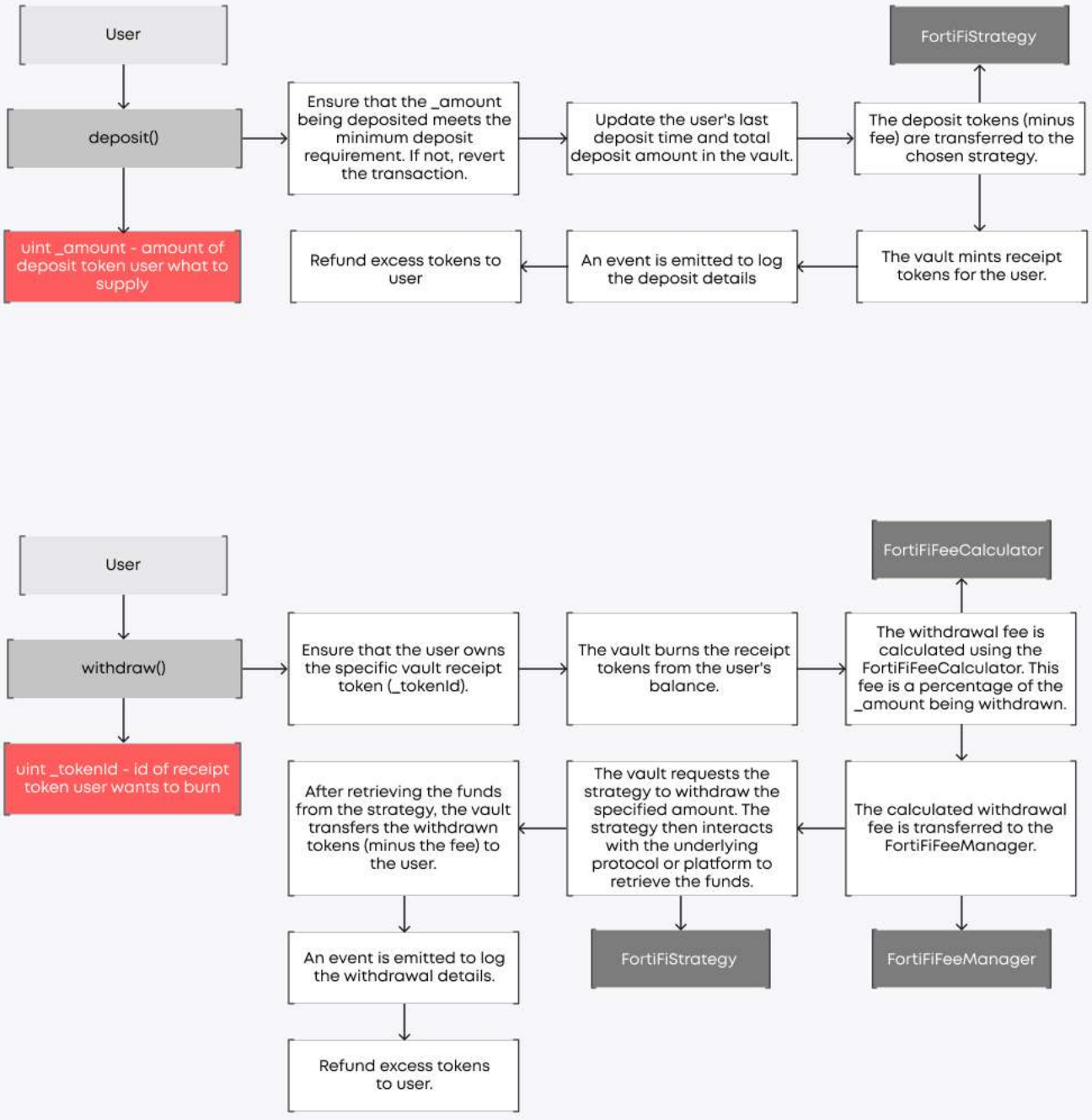
FORTIFI SCHEME

FortiFiFeeManager.sol



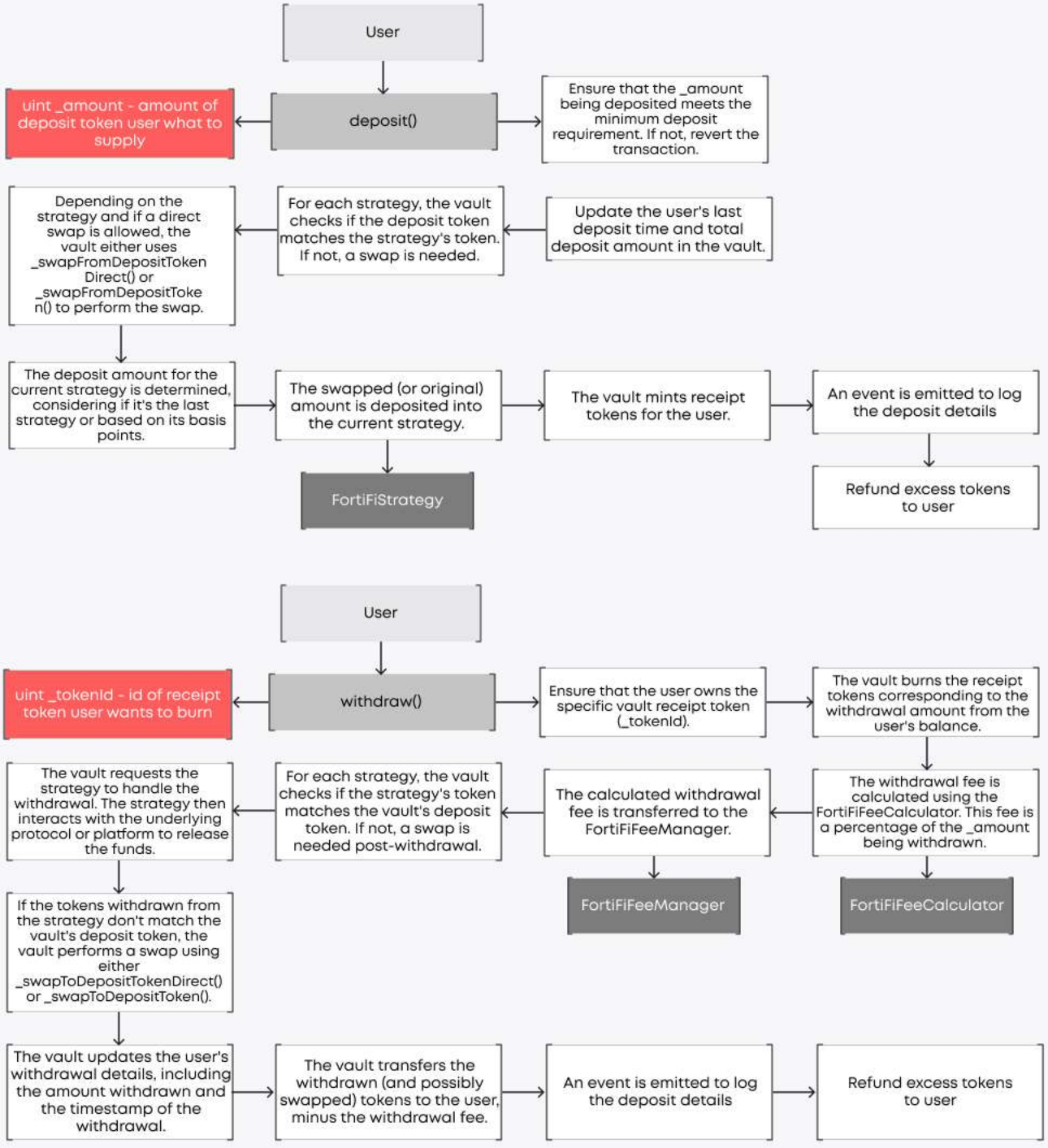
FORTIFI SCHEME

Flow of deposit and withdraw methods in SAMS Vault



FORTIFI SCHEME

Flow of deposit and withdraw methods in MASS Vault



FORTIFI SCHEME

FortiFi Vault general representation

Level 1: User Interaction

FortiFiSAMSVault:

- This is the primary contract users will interact with.
- Users can deposit their assets, which are then split and deposited into multiple yield-bearing strategies.
- Users receive an ERC1155 non-fungible receipt token as proof of their deposit.

Level 2: Strategy Management

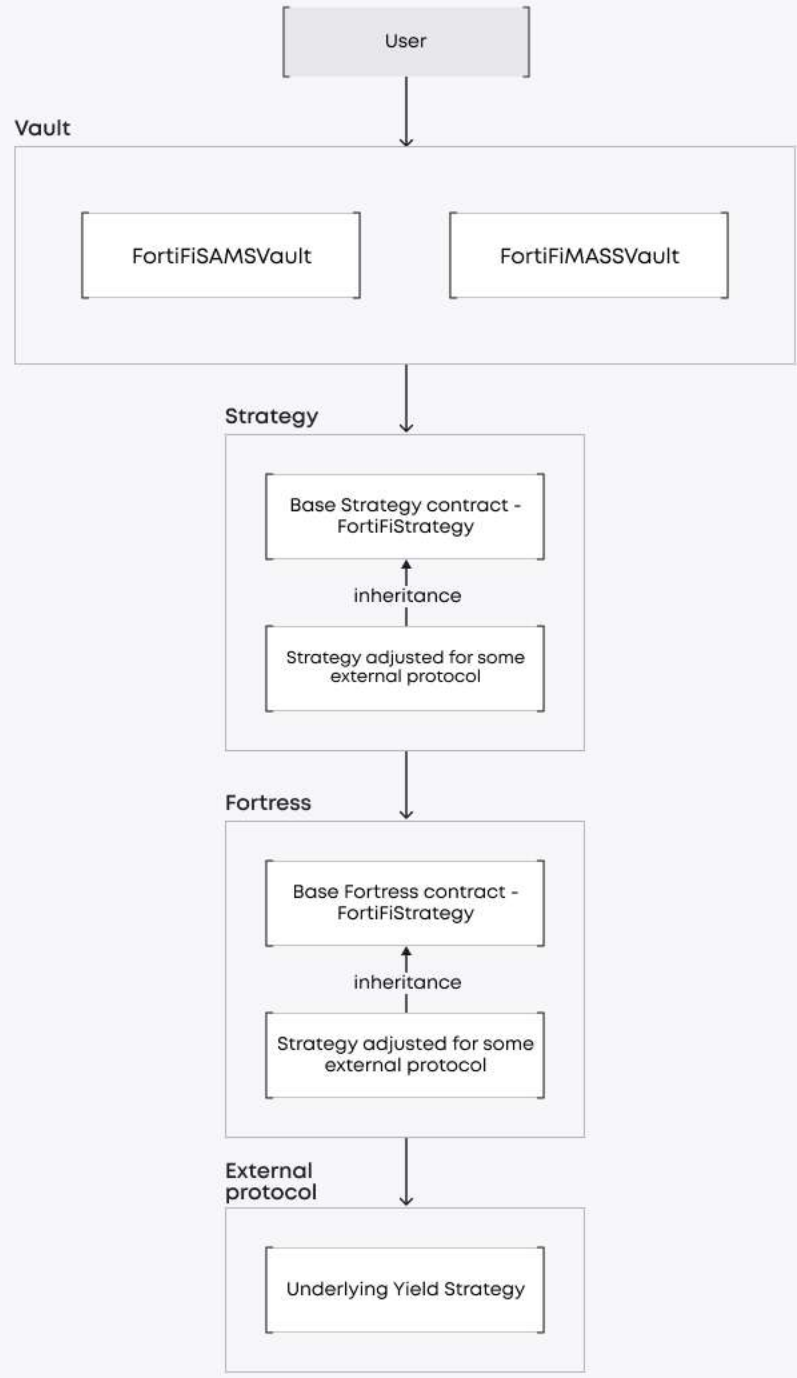
FortiFiStrategy:

- This contract is a base strategy that can be used when a yield strategy requires special logic.
- It's designed to be called only by FortiFi SAMS and MASS Vaults.
- The strategy involves depositing assets into a "Fortress" to interact with the underlying yield strategy.

Level 3: Asset Isolation

FortiFiFortress:

This contract acts as an isolated vault for individual user deposits.
It interacts directly with the underlying yield strategy.
The Fortress ensures that user deposits are isolated, allowing for balance-specific logic from underlying strategies.



COMPLETE ANALYSIS

CRITICAL-1	✔ Resolved

Frontrun attack possibility.

FortiFiMASSVault.sol, FortiFiSAMSVault.sol: _swap().
In the function _swap(), the user's funds are swapped into another token using the AMM protocol, with the minimum output amount calculated based on the estimated output. The function uses the estimated output from _router.getAmountsOut() to determine the minimum acceptable output by applying the slippage. However, this estimated output is not immune to market fluctuations and can become outdated by the time the actual swap occurs. A malicious actor could potentially frontrun the transaction by making a trade that shifts the price unfavorably for the original transaction. Consequently, when the original transaction is submitted, the actual output might be significantly less than the estimated output, even if it's still above the minimum acceptable output. This could result in the user receiving less than expected, even after accounting for the specified slippage.

Recommendation:

Use either on-chain TWAP oracles in order to validate the price retrieved from AMM **OR** off-chain oracles such as Chainlink **OR** provide the minimum output amount as a function parameter.

Post-audit:

The contract was set to fetch the price from an off-chain oracle.

HIGH-1	✔ Resolved

Owner is able to recover strategy tokens from Fortress.

FortiFiFortress.sol: recoverERC20().
The function is designed to recover any tokens that have been mistakenly sent directly to the contract. However, Fortress smart contracts are intended to store _strat, which adheres to the standard ERC-20 and can be transferred. As a result, the owner has the ability to withdraw tokens that represent users' deposits. This issue is classified as "High" primarily because it creates a dangerous backdoor that could be exploited if the owner's key is compromised, leading to a "single point of failure" design flaw. Additionally, it could negatively impact user trust. Since only the owner has the ability to execute this function, it could be exploited by attackers if the owner's private key is lost.

Recommendation:

Validate that provided _token to withdraw is not the strategy token.

Post-audit. Withdrawing strategy tokens via recoverERC20 method is now forbidden.

HIGH-2	✔ Resolved

Arbitrary addresses can be set as vaults.

FortiFiStrategy.sol: setVault().
Arbitrary addresses can be designated as vaults via the setVault() function in FortiFiStrategy.sol. Addresses assigned the Vault role have the ability to deposit and withdraw users' funds. Therefore, only legitimate smart contracts with the necessary Vault functionality must be given this role. Currently, any address, including EOA, can be assigned this role. Consequently, once assigned the role, such accounts can gain direct access to the funds. This issue is classified as "High" because it creates a dangerous backdoor with significant impact if exploited. Although only the owner can execute this function, it could be exploited by attackers who have obtained their private key.

Recommendation:

Ensure that parameter `_strategy` implements the desired Vault interface.
Post-audit: A validation was added to ensure that the vault supports the ISAMS of IMASS interface.

HIGH-3	✔ Resolved

Deprecated Chainlink API call is used for validating price.

FortiFiMASSVault.sol: _swapFromDepositToken(), _swapFromDepositTokenDirect(), _swapToDepositToken(), _swapToDepositTokenDirect().
Starting from Chainlink version 0.6, the latestAnswer() function has been deprecated and replaced with the latestRoundData() function. The latestRoundData() function provides more detailed data about the most recent round, enabling API callers to validate the retrieved response more effectively. This expanded information about the price includes the answer itself, the round's ID, and the timestamp of the update, among other details. The use of Chainlink's latestAnswer() function, which has been deprecated for over two years, introduces a multi-faceted risk to the operations

of the smart contract. From a security standpoint, deprecated functions may contain known vulnerabilities that have been addressed in later versions, leaving contracts that still use them vulnerable to potential exploits.

Recommendation:

Use latestRoundData for retrieving the latest price, instead of deprecated latestAnswer().
To migrate from latestAnswer to latestRoundData, the contract should support AggregatorV3Interface and you should replace calls to latestAnswer with latestRoundData. This function returns the round ID, the data value, timestamps of when the round started and was last updated, and the round ID in which the answer was provided. While the primary interest is typically the data value, the additional metadata can be essential for advanced use cases and validations. It's also important to note that latestRoundData can revert if a round ID doesn't exist, necessitating error handling.
(uint80 roundID, int256 **answer**, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound) = priceFeed.**latestRoundData**(roundID) – here is the full response of latestRoundData API smart contract call with all the metadata it provides.
Detailed about using the latestRoundData method can be found in the Chainlink's documentation: https://docs.chain.link/data-feeds/api-reference#latestrounddata

Post-audit:

latestRoundData method is now used instead of deprecated latestAnswer call.

HIGH-4	✔ Resolved

Incorrect calculation of the swap amount.

FortiFiMASSVault.sol: _swapFromDepositToken(), _swapFromDepositTokenDirect(), _swapToDepositToken(), _swapToDepositTokenDirect().
The formula used to calculate the swap amount does not consider the decimal places of the price oracle. If the decimal precision of the oracle does not align with the decimal placement of the strategy, an incorrect quantity is calculated. Consequently, the expected minimum amount after the swap is also inaccurately calculated.

Recommendation:

It's crucial to consider the decimal precision of the price oracle, as it may differ from the decimal precision of the storage. Additionally, the decimal accuracy of the price list may also vary. To address this, you could enhance the functionality of FortiFiPriceOracle.sol by adding a decimals() method that will query the oracle. Alternatively, when deploying the FortiFiPriceOracle.sol contract, you could specify the required decimal precision in advance. For example, the formula for calculating the amount of swap has been changed: swapFrom:
_swapAmount = ((_amount * (10 ** _strat.decimals)) / _latestPrice) * 10 ** (_strat.decimals - DECIMALS); // current version
->
_swapAmount = ((_amount * (10 ** _strat.decimals)) / _latestPrice) * 10 ** (_decimalsPrice - DECIMALS); // modified
swapTo:
_swapAmount = (_amount * (_latestPrice / 10 ** _strat.decimals)) / 10 ** (_strat.decimals - DECIMALS); // current version
->
_swapAmount = (_amount * _latestPrice * 10 ** _strat.decimals)) / 10 ** (_strat.decimals + (_decimalsPrice - DECIMALS)); // modified

Post-audit: A validation was added to ensure that the decimal precision of the strategy's deposit token matches the decimal precision of the oracle's price returned.

MEDIUM-1	✔ Resolved

Transfers lack safety checks.

FortiFiMASSVault.sol: deposit(), add(), withdraw(), recoverERC20(), _refund();
FortiFiMASSVault.sol: deposit(), add(), withdraw(), recoverERC20(), _refund();
FortiFiDPFortress.sol: withdraw();
FortiFiDPStrategy: depositToFortress();
FortiFiFortress.sol: deposit(), withdraw(), recoverERC20(), _refund();
FortiFiStrategy: depositToFortress(), withdrawFromFortress(), recoverERC20();
FortiFiVectorStrategy: depositToFortress(), withdrawFromFortress(), recoverERC20();
FortiFiVectorFortress: withdrawVector().
Transfers of ERC-20 tokens are conducted with regular .transfer()/.transferFrom() methods and the result of calls is checked for returning “true” in the require-statement. While this approach can work for tokens that adhere to this return behavior, it's not unified across all ERC-20 tokens. Some tokens might revert on failure, while others might return false. Also, some of the tokens may not return true on the success of transfer which will cause the function, which uses “require” to validate the transfer, to revert. This inconsistency can lead to potential issues when interacting with different token implementations. SafeERC20 library is providing important safety checks as it wraps transfers around the standard ERC-20 functions, adding an extra layer of safety.

Recommendation:

It is recommended to replace .transfer()/transferFrom() with safeTransfer()/safeTransferFrom() functions.

Post-audit: The function SafeERC20 is now used across the contracts.

MEDIUM-2	✔ Resolved

Owner can add duplicating strategies.

FortiFiMASSVault.sol: setStrategies(), FortiFiSAMSVault.sol: setStrategies().
The setStrategies() function in the contract permits the owner to specify the strategies that the vault will employ. However, this function lacks essential validation to prevent the owner from setting duplicate strategies. The presence of duplicate strategies can create confusion for both the contract owner and users. It may be unclear which strategy is actually in use or if both are operating concurrently, potentially leading to mismanagement or misinterpretation of the contract's state and operations. If funds are allocated to duplicate strategies, it could result in an uneven distribution of assets rather than diversifying the assets across different strategies to maximize yield or minimize risk.

Recommendation:

Implement validation to prevent the owner from setting duplicate strategies.

Post-audit: Strategies are now validated for not duplicating.

MEDIUM-3	✔ Resolved

Chainlink price feed's response is not validated.

The FortiFiPriceOracle contract, which is designed to fetch price data using the Chainlink oracle, has potential vulnerabilities in its getPrice function. The function does not validate the possibility of negative prices. Since the answer from the Chainlink oracle is of type int256 and can be negative, directly casting to uint256 without checks could lead to unexpected results. Additionally, the function does not check for stale prices. If the Chainlink oracle stops updating, the contract might rely on outdated information, leading to miscalculations or unintended behaviors.

Recommendation:

To address these concerns, it's recommended to:
1) First of all, you should retrieve the fourth parameter from the price feed's response which is responsible for the timestamp of the last price update. By checking this metadata, it is possible to validate the price for staleness.

```
(, int256 answer, , uint256 timeStamp
 , ) =fundingTokenChainLinkFeeds[fundingToken].latestRoundData();
```

2) Ensure the answer is non-negative before casting. For example:

```
if (answer <= 0) {
   revert NonPositivePrice();
}
```

3) Retrieve and validate the updatedAt timestamp from the Chainlink oracle to ensure data freshness. If freshness condition is not passed, contract can be paused, as pausable mechanism is already implemented.

```
if (updatedAt < block.timestamp - 60 * 60 /* 1 hour */) {
   flipPaused();
}
```

Post-audit: Important health checks were added to validate the fetched price from oracle.

LOW-1	✔ Resolved

Fixed Solidity version should be used.

Currently, all contracts use Solidity version ^0.8.18, a floating version that does not align with best practices. As a result, the compilation may include unstable versions of compilers with major bugs introduced during intermediate releases. Contracts should be deployed with the same compiler version and options with which they have been most thoroughly tested. Locking the pragma version helps ensure that the contract is not accidentally deployed using a different version. Therefore, it is recommended to use the latest stable version, which is currently 0.8.21.

Recommendation:

Use the last stable Solidity version.

Post-audit: Solidity 0.8.21 is now used across all the contracts.

LOW-2	✔ Resolved

Minimum deposit value might be set to zero in the SAMS vault.

FortiFiSAMSVault.sol: setMinDeposit().
In the SAMS Vault's constructor, the minimum deposit initial value is validated to be greater than the BPS variable, which has a constant value of 10000. Nevertheless, in the special setter for minimum deposit, the new value is not validated at all. New minDeposit amount might be set to a value which is less than BPS, or even zero. This could lead to unexpected behavior if users deposit a small number of tokens.

Recommendation:

Add the validation in the setter for minimum deposit value to be greater than BPS.

Post-audit. Custom error was added to validate the minDeposit is not less than BPS.

LOW-3	✔ Resolved

Slippage can be set to any amount in the MASS vault.

FortiFiMASSVault.sol: setSlippage().
This flexibility could potentially lead to front-running and arbitrage. If the slippage is set too high, the contract becomes vulnerable to front-running attacks. In such scenarios, malicious actors can anticipate the contract's trades and manipulate the price in their favor. This manipulation could cause the contract to purchase assets at an inflated price or sell them at a deflated price. While the ability to set slippage to any value provides flexibility for management, as different protocols may require varying slippage adjustments, it is still considered good practice to establish a suitable range for possible slippage changes.

Recommendation:

Add the validation for slippage to be in the certain range.

Post-audit: Slippage is now validated to be in the range [1%;5%].

LOW-4	✔ Resolved

Iteration through the storage array.

FortiFiMASSVault.sol: _withdraw(), line 397.
FortiFiSAMSVault.sol: _withdraw(), line 302.
When funds are withdrawn, the process iterates through all of the user's positions. If the number of strategies is too large, the transaction may fail due to an out-of-gas error. This issue is considered low risk because the number of positions equals the number of strategies set by the protocol owner. However, given the complex logic behind withdrawals (especially if one of the strategies is a SAMS Vault), it's recommended to set a maximum limit for the number of strategies an owner can set.

Recommendation:

Verify that the planned number of strategies fits within the Avalanche block gas limit and add a limitation on the number of strategies that the owner can set in the setStrategies() function.

Post-audit: The number of strategies is now capped at 4.

LOW-5	✔ Resolved

Swap route is created strictly via native coin.

FortiFiMASSVault.sol, FortiFiSAMSVault.sol: _swap(), _swapOut().
In the swap functions, the swap route is composed using the native coin as an intermediary token. While this approach should generally work because most tokens have a pair with the native coin, it may not be universally applicable. For instance, two tokens to be swapped might share a common pair with sufficient liquidity, or one of the tokens might not have a liquidity pair with the native coin. Therefore, a more flexible approach to composing the swap route is recommended.

Recommendation:

Review the existing routing system and consider implementing a more flexible method for composing swap routes. For example, an optimal swap route between two tokens could be set by the owner in a separate method. **OR** consider implementing a sanitizing policy for used tokens, such as checking for the existence of necessary pairs.

Post-audit:

The ability to set a specific route for each strategy has been implemented.

LOWEST-1	✔ Resolved

Strategies bps might be changed only in a way of resetting strategies.

FortiFiMASSVault.sol: setStrategies();
FortiFiSAMSVault.sol: setStrategies().
If strategies need frequent adjustments in their bps, then having to reset the entire strategy to change the bps can be cumbersome and inefficient. Also, resetting strategies entirely to change bps can be more gas-intensive than just updating a single value. This can lead to higher costs for the contract owner or whoever is responsible for making these changes.

Recommendation:

Consider implementing separate mechanisms for adjusting strategies' bps.

Post-audit. Separate function for setting bps was added and now new bps for strategies might be changed without resetting the whole strategies information in storage.

LOWEST-2	✔ Resolved

Local variable initialization is redundant.
FortiFiMASSVault.sol: deposit(), add();
FortiFiSAMSVault.sol: deposit(), add().
In the deposit() and add() methods of the MASS Vault and SAMS Vault contracts, a local variable _depositToken is instantiated to represent the IERC20 interface of the depositToken address. However, this variable is only used once in the subsequent transferFrom call, rendering its declaration unnecessary. It would be more gas-efficient and streamline the code to directly use the state variable depositToken in the transferFrom call, thereby eliminating the need for the redundant instantiation of the local variable.

Recommendation:

Access deposit token directly without creating local variable.

Post-audit: Redundant initialization was removed.

LOWEST-3	✔ Resolved

Getters for private variables.
FortiFiFortress.sol, FortiFiStrategy.sol.
From a gas efficiency perspective, using public state variables would automatically generate getter functions without manually defining them. However, in this contract, the state variables are declared internal, which means they are not directly accessible outside the contract. To provide external access to these variables, explicit getter functions are defined.
The gas overhead of these manually defined getter functions is negligible compared to automatically generated ones from public state variables. However, from a code simplicity and readability standpoint, if there's no specific reason to keep these variables internal, making them public would automatically generate these getters and reduce the amount of manual code.

Recommendation:

Consider changing variables' visibility from internal to public in order to remove getters and reduce code complexity.

Post-audit: All the variables' visibility was changed to public.

LOWEST-4	✔ Resolved

Nat-spec does not match with business logic.

1) FortiFiMASSVault.sol: deposit();
FortiFiSAMSVault.sol: deposit().
In the NatSpec documentation of the deposit() function in both vaults, there is a statement that "the user must deposit at least the minDeposit" amount of deposit token. However, the amount is validated to be strictly greater than "minDeposit". The non-strict inequality is a good practice for validating input amount and current implementation violates it.
2) FortiFiStrategy.sol: deposit(), line 47.
In the comments it is stated that "If the user has not deposited previously, deploy Fortress" however, the address of msg.sender which vault is used in the lines of code following the comment.

Recommendation:

Change the NatSpec **OR** change require-statement to be non-strict inequality.

Post-audit: Non-strict inequality was implemented, and now NatSpec matches the code.

LOWEST-5	✔ Resolved

Lack of events.

- FortiFiMASSVault.sol: refreshApprovals(), setStrategies(), setMinDeposit(), setFeeManager(), setFeeCalculator(), setNoFeesFor(), flipPaused().
- FortiFiSAMSVault.sol: refreshApprovals(), setStrategies(), setMinDeposit(), setFeeManager(), setFeeCalculator(), setNoFeesFor(), flipPaused().
- FortiFiFeeCalculator.sol: setFees(), setCombine().
- FortiFiFeeManager.sol: recoverERC20().
- FortiFiFortress.sol: deposit(), withdraw(), refreshApproval(), recoverERC20().
- FortiFiDPFortress.sol: withdraw().
- FortiFiVectorFortress.sol: withdrawVector().
- FortiFiStrategy.sol: setVault(), recoverERC20(), recoverFromFortress().

Essential functions such as setters or functions which interact with funds or storage should emit events in order to keep track of historical changes of data.

Recommendation: Emit events in the corresponding functions.

Post-audit: Events were added.

LOWEST-6	✔ Acknowledged

Local variable is set to zero in the initialization.
FortiFiSAMSVault.sol: setStrategies(), _bps.
The _bps variable is initialized with zero value. In Solidity, when declared without initialization, it defaults to its zero value. For uint types, this default value is 0. From a gas perspective, there's no difference in contract deployment cost between the two approaches. Both will result in the same gas usage for the initialization of that variable. However, from a code readability standpoint, being explicit can sometimes help in conveying the developer's intention. If it wanted to emphasize that the initial value should be 0, then it might choose to initialize it explicitly. Otherwise, just declaring "_bps"; is more concise and achieves the same result.

Recommendation: Remove explicit variable's initialization.

Post-audit: The FortiFi team has decided to leave explicit initialization.

LOWEST-7	✔ Resolved

Variable for storing slippage has different data dimensions over contracts.
Maintaining consistency across contracts for the same kind of data is generally a good practice. This makes the codebase easier to understand and reduces the potential for errors.

Recommendation:

Standardize the data type for slippageBps across all contracts. Choose the data type that best fits the range of values slippageBps is expected to have, and consider the gas implications.

Post-audit: The slippageBps variable was normalized across all the contracts.

LOWEST-8	✔ Resolved

“Magic” numbers.
FortiFiMASSVault.sol: _swap().
The usage of values directly in code reduces its readability. Thus, it is recommended to use constants for the value 1800 instead of using the value directly for better code readability and pointing out that this value stands for handling the swap deadline.

Recommendation:

Use constants instead of values directly.

Post-audit: Constant SWAP_DEADLINE_BUFFER was added, and “magic” number was removed.

LOWEST-9	✔ Resolved

Custom errors should be used.
Multiple files.
Starting from the 0.8.4 version of Solidity, it is recommended to use custom errors instead of storing error message strings in storage and use “require” statements. Using custom errors is more efficient regarding gas spending and increases code readability.

Recommendation:

Use custom errors instead of require-statements.

Post-audit: Require-statements were replaced with custom errors.

LOWEST-10	✔ Resolved

Redundant transferring of ownership.
FortiFiFortress.sol: constructor, line 31.
The instances of FortiFiFortress smart contract are deployed by the _fortiFiStrat, which is the message’s sender of the deployment. Thus, _fortiFiStrat is set as the owner of the contract. Therefore, setting it as an owner in line 31 is redundant.

Recommendation:

Remove redundant transfer of ownership.

Post-audit: Redundant ownership’s transfer was removed.

LOWEST-11	✔ Resolved

Duplicating functions in smart contracts.
FortiFiFortress.sol, FortiFiDPFortress.sol: withdraw().
The contract FortiFiDPFortress inherits the functionality of FortiFiFortress. However, the overridden function withdraw() is identical to the one declared in the FortiFiFortress. Thus, overriding it is redundant.

Recommendation:

Remove redundant overridden functions.

Post-audit: Duplicating functions were removed.

LOWEST-12	✔ Resolved

Similar naming of variables.
FortiFiMASSVault.sol, FortiFiSAMSVault.sol: _swap(), _swapOut(), variables depositToken and _depositToken.
The variables have similar names, which decreases the code's readability and might lead to mistakes when updating the code (mainly if another developer updates it). This issue is marked as info since the current code uses the variables correctly.

Recommendation:

Rename the variables to make them more distinctive. For example, _depositToken can be named as _strategyDepositToken.

Post-audit: The variables' naming was changed, and the code readability was increased.

LOWEST-13	✔ Resolved

Users are unable to withdraw funds when Vault is paused.
FortiFiMASSVault.sol: withdraw(), FortiFiSAMSVault.sol: withdraw(). The FortiFiMASS and FortiFiSAMS vaults contract incorporates a pausing mechanism, controlled by the paused state variable. When the contract is in the paused state, several functions, including the critical withdraw function, become inaccessible to users. Auditors recognize the necessity of having the pause mechanism which prevents funds lost in case of a hack. However, such design choice can lead to situations where users are unable to access and withdraw their funds from external protocols, especially during times when they might need to act urgently.

Issue is marked as “Info”, as for now it looks like a design choice. But, the issue may be re-qualified later (based on comments of the team and results of testing stage) up to “High”-risk, as it is not directly leading to a loss of funds, but users might be blocked from accessing their funds, which is crucial in the decentralized Money Market. Although since only the owner can execute the function, it leads to a “single point of failure” design flow that might be exploited by attackers in the event of the owner's private key loss.

Recommendation:

Verify the need to pause the withdrawals **OR** provide a description of mitigation and rescue strategies **OR** Consider allowing users to withdraw their funds even when a contract is paused **OR** consider diversifying roles.

Post-audit: Withdrawals are now enabled when the contract is in paused mode.

LOWEST-14	✔ Resolved

DECIMALS constant variable is responsible only for storing USDC's decimals.

FortiFiMASSVault.sol: DECIMALS.

DECIMALS constant variable is used for storing USDC decimals, so the name for that variable should point out that fact. The name USDC_DECIMALS should be more appropriate.

Recommendation:

Change the name of the constant variable "DECIMALS" to indicate that it represents USDC decimals.

Post-audit: The name of the variable was changed to "USDC_DECIMALS".

LOWEST-15	✘ Acknoweledged

Yield Yak strategy might return not full profit.

FortiFiMASSVault.soll, FortiFiSAMSVault.soll: withdraw(), rebalance().

Considering how the rewards are distributed in the strategy Yield Yak, namely by calling the reinvest() method (https://docs.yieldyak.com/for-farmers/reinvest). There is a possibility of not getting the full profit from the Yield Yak strategy if the reinvest() call was made long ago relative to the current block number. The vault does not execute the reinvest() call before calling withdraw() on the strategy contract.

Recommendation:

It should be confirmed that the possibility of incomplete profit from the Yield Yak`strategy is quite normal and expected.

It is not a complete solution to supplement the vaults functionality by calling ŕeinvest()' before ẁithdraw()' or ŕebalance()' on the vault contract to take the full amount of profit, as there may be problems with the lack of total gas.

LOWEST-16	✕ Acknoweledged

Contract might be optimised in order to reduce its size.
FortiFiMASSVault.soll, FortiFiSAMSVault.soll.
The functionality of direct interaction (without a wrapper) of a vault with a strategy is compatible only with Yield Yak strategies (at the moment). It is unclear if this functionality will be suitable for any other protocols that may be included in the future. To ensure the contract fits the contracts' limit of Avalanche and there is space for future possible updates and features.

Recommendation:

To optimize the size of vault contracts, remove the functionality of direct interaction with the strategy. Since this functionality is compatible ONLY with one strategy Yield Yak (at the moment). Instead, it may be more efficient to interact with all strategies through wrappers only.

✔ Re-entrancy	Pass
✔ Access Management Hierarchy	Pass
✔ Arithmetic Over/Under Flows	Pass
✔ Delegatecall Unexpected Ether	Pass
✔ Default Public Visibility	Pass
✔ Hidden Malicious Code	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass
✔ External Contract Referencing	Pass
✔ Short Address/Parameter Attack	Pass
✔ Unchecked CALL Return Values	Pass
✔ Race Conditions/Front Running	Pass
✔ General Denial Of Service (DOS)	Pass
✔ Uninitialized Storage Pointers	Pass
✔ Floating Points and Precision	Pass
✔ Tx.Origin Authentication	Pass
✔ Signatures Replay	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

FortiFiFeeCalculator

initialize contract

- ✓ Should be proper address (1377ms)
- ✓ Should return the address of the contract owner
- ✓ Should return the correct NFT addresses
- ✓ Should return the correct token amounts
- ✓ Should return the correct threshold bps
- ✓ Should restore the correct state of combine NFT

getFees method

- ✓ Should calculate fee in relation to sum of NFT balances (85ms)
- ✓ Should calculate fee in relation to max of NFT balances (87ms)
- ✓ Should return 0 fee for small amounts with combine balance NFT (69ms)
- ✓ Should return 0 fee for small amounts without combine balance NFT (69ms)

setFees method

- ✓ Should return an access error for a non-owner (42ms)
- ✓ Should return an error for an empty address array
- ✓ Should return an error for zero addresses
- ✓ Should return an error for different numbers of values in data arrays
- ✓ Should return an error for an empty token amount array
- ✓ Should return an error for a non-zero amount of tokens in the first slot of the array
- ✓ Should return an error for an invalid token amount sequence
- ✓ Should return an error for an invalid threshold bps sequence
- ✓ Should be set new the data for the fee (111ms)

setCombine method

- ✓ Should return an access error for a non-owner
- ✓ Should be set to new value for combine NFT

FortiFiFeeManager

initialize contract

- ✓ Should be proper address (151ms)
- ✓ Should return the address of the contract owner
- ✓ Should return the correct receiver addresses
- ✓ Should return the correct split bps

setSplit method
- ✓ Should return an access error for a non-owner
- ✓ Should return an error for an empty receivers array
- ✓ Should return an error for zero address receiver
- ✓ Should return an error for different numbers of values in data arrays
- ✓ Should return an error for too small a split value bsp
- ✓ Should return an error for an invalid total split value bsp
- ✓ Should be set new the data for split bsp (54ms)

collectFees method
- ✓ Should collect the fee of the specified asset and the amount without sharing it by recipients (92ms)
- ✓ Should collect the fee of the specified asset and the amount with sharing it by recipients (102ms)

recoverERC20 method
- ✓ Should return an access error for a non-owner
- ✓ Should transfer the specified asset and amount to contract owner (38ms)

FortiFiFortress

initialize contract
- ✓ Should be proper address (158ms)
- ✓ Should return the address of the contract owner
- ✓ Should return the correct deposit token address
- ✓ Should return the correct wrapped native token address
- ✓ Should return the correct strategy contract address

deposit method
- ✓ Should return an access error for a non-owner
- ✓ Should return an error for zero amount
- ✓ Should add deposit for user (89ms)
- ✓ Should add deposit for user and refund left over tokens (92ms)

withdraw method
- ✓ Should return an access error for a non-owner
- ✓ Should return an withdraw error
- ✓ Should execute token withdrawal (86ms)

refreshApproval method
- ✓ Should execute token approve for strategy contract

recoverERC20 method
- ✓ Should return an access error for a non-owner

- ✓ Should return an access error for a invalid token address
- ✓ Should transfer the specified asset and amount to contract owner (49ms)

FortiFiMASSVault

initialize contract

- ✓ Should be proper address (593ms)
- ✓ Should return the address of the contract owner
- ✓ Should return the correct deposit token address
- ✓ Should return the correct wrapped native token address
- ✓ Should return the correct fee calculator address
- ✓ Should return the correct fee manager address
- ✓ Should return the correct strategy data

setMinDeposit method

- ✓ Should return an access error for a non-owner
- ✓ Should return an error for invalid value of min deposit
- ✓ Should be set new value for min deposit

setSlippage method

- ✓ Should return an access error for a non-owner
- ✓ Should return an error for invalid value of slippage
- ✓ Should be set new value for slippage

setFeeManager method

- ✓ Should return an access error for a non-owner
- ✓ Should return an error for zero address of new fee manager
- ✓ Should be set new address for fee manager (64ms)

setFeeCalculator method

- ✓ Should return an access error for a non-owner (51ms)
- ✓ Should return an error for zero address of new fee calculator
- ✓ Should be set new address for fee calculator (52ms)

setDirectSwapFor method

- ✓ Should return an access error for a non-owner
- ✓ Should be set new state of direct swap for token address

flipPaused method

- ✓ Should return an access error for a non-owner
- ✓ Should be set new state for paused

FortiFiStrategy

initialize contract

- ✓ Should be proper address (170ms)
- ✓ Should return the address of the contract owner

- ✓ Should return the correct deposit token address
- ✓ Should return the correct wrapped native token address
- ✓ Should return the correct strategy contract address

depositToFortress method

- ✓ Should return an error for zero amount
- ✓ Should return error if call is not from valid vault
- ✓ Should add deposit with creating new fortress contract (160ms)
- ✓ Should add deposit without creating new fortress contract (263ms)

setVault method

- ✓ Should return an access error for a non-owner
- ✓ Should return an error for invalid vault address
- ✓ Should set new vault contract address

recoverERC20 method

- ✓ Should return an access error for a non-owner
- ✓ Should transfer the specified asset and amount to contract owner (48ms)

recoverFromFortress method

- ✓ Should return an access error for a non-owner
- ✓ Should transfer the specified asset and amount to contract owner (151ms)

withdrawFromFortress method

- ✓ Should return an error for zero amount
- ✓ Should return error when invalid fortress address
- ✓ Should execute token withdrawal(253ms)

FortiFiVectorFortress

initialize contract

- ✓ Should be proper address (178ms)
- ✓ Should return the address of the contract owner
- ✓ Should return the correct strategy contract address

withdraw method

- ✓ Should return an access error for a non-owner
- ✓ Should return error

withdrawVector method

- ✓ Should return an access error for a non-owner (41ms)
- ✓ Should return an withdraw error
- ✓ SShould execute token withdrawal (121ms)

FortiFiVectorStrategy

initialize contract

- ✓ Should be proper address (154ms)

- ✓ Should return the address of the contract owner

depositToFortress method

- ✓ Should return an error for zero amount
- ✓ Should return error if call is not from valid vault
- ✓ Should add deposit with creating new fortress contract (158ms)
- ✓ Should add deposit without creating new fortress contract (246ms)

withdrawFromFortress method

- ✓ Should return an error for zero amount
- ✓ Should return error when invalid fortress address
- ✓ Should execute token withdrawal (286ms)

setSlippage method

- ✓ Should return an access error for a non-owner
- ✓ Should be set new slippage value

113 passing (8s)

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY FORTIFI TEAM

Basic MASS Vault Tests

- ✓ Check that ERC20 tokens are minted to addresses
- ✓ Check that NFT1 tokens are minted to addresses
- ✓ Check that ERC20 tokens can be deposited and withdrawn (312ms)
- ✓ Check that vault can handle profits and fees (268ms)
- ✓ Check that users can add to position (853ms)
- ✓ Check that users can rebalance positions (696ms)
- ✓ Check that invalid configurations revert (187ms)
- ✓ Check that invalid transactions revert (428ms)

8 passing (8s)

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.