Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

TIE FINANCE

Spark Strategies

Veridise Inc.
November 21, 2024

► **Prepared For:**

TIE Finance
`https://tie-finance.gitbook.io/tie-finance-docs`

► **Prepared By:**

Tyler Diamond
Benjamin Sepanski

► **Contact Us:**

`contact@veridise.com`

► **Version History:**

Nov. 21, 2024	V2
Nov. 07, 2024	V1
Nov. 06, 2024	Initial Draft

Contents

1 Executive Summary

From Oct. 28, 2024 to Nov. 04, 2024, TIE Finance engaged Veridise to conduct a security assessment of the smart contracts implementing their new Spark Strategies. The security assessment covered small updates to several of TIE Finance's investment strategies, the integration of Spark into the existing strategies, and a new vault which allows deposits from different tokens. Veridise conducted the assessment over 2 person-weeks, with 2 security analysts reviewing the project over 1 week on commit `e0726433`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. Veridise has audited the Tie-Finance-ETH_LeverageOnAave* three times previously. The general structure of TIE Finance vaults work as follows. Each `Vault` is an ERC-4626 vault. An owner-configured `Controller` contract funnels funds from the `Vault` to a sub-strategy. This sub-strategy invests the funds, possibly acting as a controller to reinvest funds in other sub-strategies. Most of the sub-strategies define an investment portfolio using AAVE or a fork thereof.

This security assessment covered changes and extensions of two strategies. The first strategy, `ETHStrategy`, is designed to use AAVE (or a fork of AAVE) as a loan provider. The strategy uses the loan to margin Lido stETH against WETH at an owner-controlled loan-to-collateral ratio. The second strategy, `lendingStrategy`, deposits a separate token (intended to be either WBTC or a supported stablecoin) as collateral into AAVE, and then borrows WETH for investment into an underlying `ETHStrategy`.

The substantive changes in each strategy were (1) a slight reworking of the math (see related issue V-TIE-VUL-004) and (2) modifying the vaults to be more extensible by parameterizing the input token and deposit procedure. The strategies were then each extended with the new `farmSpark` contract, allowing collection and reinvestment of rewards† received from investing the strategies' funds into Spark (an AAVE V3 fork). Variants of the `lendingStrategy` using both stablecoins (subsequently converted to sDAI) and WBTC were created.

Additionally, the TIE Finance developers added various utilities including a multi-token-vault, which swaps tokens to/from the underlying vault deposit asset when depositing/withdrawing, and various utilities for interacting with Chainlink and Curve.

Code Assessment. The TIE Finance developers provided the source code of the Spark Strategies contracts for the code review. The source code appears to be mostly original code written by the developers. The Veridise analysts consulted with prior Veridise reviewers of the protocol, notes from prior audits, and the prior audit reports before beginning the review.

The initial project contained little to no internal documentation. To facilitate the Veridise security analysts' understanding of the code, the Spark Strategies developers provided high-level

* `https://github.com/Tie-Finance/Tie-Finance-ETH_LeverageOnAave/tree/e0726433`
† `https://docs.spark.fi/user-guides/farming-rewards/claiming-rewards`

documentation in the form of a gitbook‡. The analysts attempted to understand the intended behavior of the code from the source code and asked questions where they were unclear on the intent of the developers.

The source code contained a test suite, which the Veridise security analysts noted only checked that functions executed without error, and did not test any other properties of the functions, their return values, or the contract state. Further, these tests have very low coverage of the protocol. Average line coverage is only 20%, with over 75% of functions entirely untested (see Appendix A.1 for more details). No linting or CI/CD appears to be in place in the repository (see V-TIE-VUL-018). Further, despite the complexity of the protocol, there is no deployment script, and all tests are performed in a mocked environment. Note that this protocol interacts with a large number of third-party protocols, including AAVE, Balancer, Chainlink, Curve, Lido, Uniswap, and others.

Summary of Issues Detected. The security assessment uncovered 26 issues, 1 of which is assessed to be of high or critical severity by the Veridise analysts. Specifically, V-TIE-VUL-001 allows users to bypass the pausing mechanism and continue to deposit while the protocol is paused. The Veridise analysts also identified 3 medium-severity issues, including over-charging fees (V-TIE-VUL-002), several issues identified in a previous audit (V-TIE-VUL-003), and incorrect rebalancing when reducing the loan-to-collateral ratio (V-TIE-VUL-004). Additionally, the analysts raised 6 low-severity issues, 14 warnings, and 2 informational findings. These include centralization risks (V-TIE-VUL-005), incorrect historical data in oracles (V-TIE-VUL-006), usability issues which may provoke future errors (V-TIE-VUL-007), lack of slippage protection (V-TIE-VUL-008), as well as several maintainabality concerns and recommendations for improved engineering practices such as V-TIE-VUL-018, V-TIE-VUL-020, and V-TIE-VUL-024. The security analysts notified TIE Finance about the 0 unresolved issues, but have not yet received a response regarding acknowledgments or fixes.

Recommendations. After conducting the assessment of the protocol, the security analysts had several suggestions to improve the Spark Strategies project.

The first and most important of the recommendations is to test the protocol. The new features, and the protocol as a whole, are essentially untested. Every public method should be tested in CI within a forked or more realistically mocked environment. Further, the outputs of functions and states of the contract should be checked. For example, tests should be written to validate that the loan-to-collateral ratio is at the expected value (to prevent errors like V-TIE-VUL-004), that the amount of shares/assets received when depositing/withdrawing matches the expected amounts, and that collected fees match expectations. Further tests should flex negative scenarios such as ensuring access control is in place, validating that actions cannot be performed when paused (e.g. to prevent errors like V-TIE-VUL-001), and checking that the vault behaves as expected in various anticipated scenarios.

Testing is critical to the security of a protocol. Thorough testing should be performed and integrated into the development procedure and CI before this project is deployed or used in mainnet.

‡ `https://tie-finance.gitbook.io/tie-finance-docs/products/vaults`

The second most important recommendation is to write a deployment script. Deployments should be reproducible with both testnest and mainnet configurations. This is essential for projects which interact with a large number of third-party protocols as any misconfiguration can have potentially disastrous results. At the very least, it is more difficult to understand the intended behavior of the code, money may be wasted on a failed deployment, and it may be difficult to validate the deployed code matches the audited source code. At worst, a misconfigured deployment may lead to locked or stolen funds. For example, if manipulable oracles are used with the vault, flash loan attacks could be used to sandwich depositors and steal funds.

Thirdly, the Veridise analysts recommend commenting the codebase more thoroughly. Contracts and core functions should have nat-spec comments explaining their intended usage. Additionally, deprecated contracts should be clearly marked as not for production and deprecated code that has been commented out should be entirely removed.

Finally, the Veridise analysts strongly recommend following the recommendations in V-TIE-VUL-018 to better conform with Solidity best practices.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Spark Strategies	`e0726433`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Oct. 28–Nov. 04, 2024	Manual & Tools	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	1
Medium-Severity Issues	3	3	2
Low-Severity Issues	6	5	2
Warning-Severity Issues	14	13	7
Informational-Severity Issues	2	2	1
TOTAL	26	24	13

Table 2.4: Category Breakdown.

Name	Number
Logic Error	6
Maintainability	6
Data Validation	4
Access Control	3
Frontrunning	2
Flashloan	2
Reentrancy	1
Usability Issue	1
Gas Optimization	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Spark Strategies's smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Does usage of any third-party protocols follow best practices and use APIs properly?
- Can the interactions between Spark and the underlying `ETHStrategy` lead to any accounting errors, locked funds, or other unintentional behaviors?
- Is the codebase vulnerable to any common Solidity issues like reentrancy, front-running, locked funds, slippage protection, incorrect access control, or large stakeholder attacks?
- Is the vault logic implemented correctly?
- Are standard attacks on vaults (such as inflation attacks) protected against?
- Do assets properly back shares as intended by the protocol?
- Can an attacker grief the protocol by forcing them into a lack of solvency?
- Does sending collateral to the AAVE pool negatively affect the strategy?
- Are fees accounted for correctly?
- Do strategies enforce proper invariants regarding solvency?
- Can non-permissioned users perform highly-permissioned actions?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract analysis tool Vanguard, as well as the open-source tool Semgrep. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* Security analysts leveraged fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, the desired behavior of the protocol was formulated as [V] specifications and then tested using Veridise's fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of this security assessment is limited to changes in the `contracts/` folder of the source code provided by the Spark Strategies developers, which contains the smart contract implementation of the Spark Strategies, since the last audited commit (9d8697cc).

During the security assessment, the Veridise security analysts referred to unchanged files (as well as the deprecated files `Exchange.sol` and `PolygonExchange.sol`), but assumed that they have been implemented correctly.

More specifically, the assessment was scoped to the following files, focusing only on changes to files which existed and had been previously audited on or before commit 9d8697cc, listed below. For the files not described in the project summary in Section 1, a brief description of the changes is listed.

- **contracts/core/**
  - `multiTokenVault.sol` (New)
- **contracts/rebalance/**
  - `aaveRebalance.sol` (Cosmetic changes only)
  - `IAaveStrategy.sol` (Cosmetic changes only)
- **contracts/subStrategies/**
  - `lendingStrategySpark.sol` (New)
  - `farmClaim.sol` (New)
  - `farmSpark.sol` (New)
  - `SavingDaiStrategy.sol` (New)
  - `ETHStrategySpark.sol` (New)
  - **aavePool/**
    - `aavePoolV3.sol` (Cosmetic changes only)
  - **exchange/**
    - `CurveRouterExchange.sol` (New)
    - `CurveRouterExchangeETH.sol` (New)
    - `curveExchangeETH.sol` (New)
    - `curveExchange.sol` (Modified)
  - **interfaces/**
    - `ICurveRouter.sol` (New)
    - `IRewardsController.sol` (New)
    - `AggregatorV3Interface.sol` (Modified)
    - `IExchange.sol` (Modified)
  - **oracle/**
    - `STGAggregator.sol` (New)
    - `WSTETHAggregator.sol` (New)
    - `STGAggregator` (Modified)
  - **silo/**
    - `farmStrategy.sol` (Return early instead of revert and parameterize base/deposit assets)
    - `SiloStrategy.sol` (Parameterize reward token)
    - `stargateStrategy.sol` (Cosmetic changes only)
    - **interfaces/**
      - `ISiloIncentivesController.sol` (Modified)
  - `ETHStrategy.sol` (Modified)
  - `lendingStrategy.sol` (Modified)
  - `saveApprove.sol` (Modified)

Methodology. Veridise security analysts reviewed the reports of previous audits for Spark Strategies, inspected the provided tests, and read the Spark Strategies documentation. They then began a review of the code assisted by both static analyzers and automated testing. During the security assessment, the Veridise security analysts regularly asked the Spark Strategies developers questions about the code in a shared group.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-TIE-VUL-001	Token deposits can occur while paused	High	Fixed
V-TIE-VUL-002	Fees should be collected before compounding	Medium	Fixed
V-TIE-VUL-003	Issues not fixed from previous audit	Medium	Acknowledged
V-TIE-VUL-004	reduceMLR() incorrect with flashloan fees	Medium	Fixed
V-TIE-VUL-005	Centralization Risk	Low	Acknowledged
V-TIE-VUL-006	wstETH oracle returns inaccurate . . .	Low	Fixed
V-TIE-VUL-007	CurveRouterExchangeETH cannot swap ETH	Low	Intended Behavior
V-TIE-VUL-008	Withdrawing has inadequate slippage . . .	Low	Fixed
V-TIE-VUL-009	Precision loss in arithmetic	Low	Acknowledged
V-TIE-VUL-010	Compounding can be frontrun	Low	Acknowledged
V-TIE-VUL-011	Unused program constructs	Warning	Fixed
V-TIE-VUL-012	Two-step ownership transfer is preferable	Warning	Acknowledged
V-TIE-VUL-013	getSTGPrice is vulnerable to flash loans	Warning	Acknowledged
V-TIE-VUL-014	Missing address zero-checks	Warning	Fixed
V-TIE-VUL-015	Duplicate code	Warning	Acknowledged
V-TIE-VUL-016	Deprecated code	Warning	Fixed
V-TIE-VUL-017	Cross-contract reentrancies for tokens with . . .	Warning	Fixed
V-TIE-VUL-018	Missing Solidity conventions/best practices	Warning	Acknowledged
V-TIE-VUL-019	Hard-coded constants	Warning	Fixed
V-TIE-VUL-020	Use of transfer() to send ETH	Warning	Fixed
V-TIE-VUL-021	Vaults ignore withdrawn asset amount	Warning	Acknowledged
V-TIE-VUL-022	reduceMLR() or raiseMLR() may be . . .	Warning	Intended Behavior
V-TIE-VUL-023	AAVE withdraw return ignored	Warning	Fixed
V-TIE-VUL-024	Chainlink oracle may be stale	Warning	Acknowledged
V-TIE-VUL-025	Gas Optimization	Info	Acknowledged
V-TIE-VUL-026	Typos and incorrect comments	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-TIE-VUL-001: Token deposits can occur while paused

Severity	High	**Commit**	e072643
Type	Access Control	**Status**	Fixed
File(s)	`contracts/core/multiTokenVault.sol`		
Location(s)	depositToken()		
Confirmed Fix At	d8ad2fa		

All vault deposit/withdrawal functions except for `depositToken()` have two modifiers: `nonReentrant` (in case of using tokens with callbacks in the future) and `unPaused` (to ensure the protocol is not paused).

The `depositToken()` function, defined in the `multiTokenVault` to allow deposits from assets other than the vault's deposit asset, is missing both modifiers.

```
function depositToken(address token,uint256 amount,uint256 minShares,address receiver
    ) external returns (uint256 shares)
```

Snippet 4.1: Definition of `depositToken()`

Impact Users may deposit while the vault is paused. If the vault is paused due to an emergency, this could cause more funds to be put at risk. If there is an exploit triggered by deposit functionality, it will be impossible to stop the exploit.

Further, reentrant tokens cannot be supported by the protocol.

Recommendation Add the `unPaused` and `nonReentrant` modifiers to the method.

Developer Response The developers have added the two requested modifiers to the function.

4.1.2 V-TIE-VUL-002: Fees should be collected before compounding

Severity	Medium	**Commit**	e072643
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/subStrategies/ETHStrategySpark.sol`		
Location(s)	compound()		
Confirmed Fix At	d8ad2fa		

The `lastTotal` variable in `ETHStrategy` tracks the last known value in the aave pool, calculated simply as collateral − debt. `_calculateFee()` uses this to determine if interest has been accrued by the strategy, in which case a fee is charged on the earned interest. The `collectFee()` modifier mints the earned fee to the `feePool`, and is called for actions such as depositing, withdrawing and changing the fee.

The `farmClaim` contract exposes the `compound()` method which performs the auto-compounding subroutine of claiming rewards, swapping them to the strategy's `depositAsset`, and re-depositing into the strategy.

The `ETHStrategySpark` contract inherits from both `ETHStrategy` and `farmClaim` (via `farmSpark`). When the `compound()` function is called, `_totalAssets()` will increase from the compounding. However, `collectFee()` is not called prior to this, and `lastTotal` is not updated afterwards. Therefore, the next action that does call `collectFee()` will incorrectly charge a fee on the rewards claimed and converted in `compound()`.

Impact Users will experience higher fees then they should.

Recommendation Override the external compound to use the `collectFee()` modifier. Simply updating `lastTotal` at the end of `_compound` will not fix the issue.

Developer Response The developers defined an internal `_beforeCompound()` function that is called at the beginning of `_compound()`. The `ETHStrategySpark` and `lendingStrategySpark` then override this function to call the `collectFee` modifier.

4.1.3 V-TIE-VUL-003: Issues not fixed from previous audit

Severity	Medium	**Commit**	e072643
Type	Logic Error	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Several issues from the previous audit that Veridise engaged in have not been fixed. This includes two medium severity issues. These issue IDs include:

1. Unresolved issues from V1 of the "Silo Strategy" report, dated August 15, 2024:
   a) V-TSS-VUL-002: Received ETH in contracts becomes stuck.
      i. The `CurveExchange` still has a `receive()` function. Note that `CurveExchangeETH` will need a `receive` function.
   b) V-TSS-VUL-004: Unsuccessful swap prevents reward distribution.
   c) V-TSS-VUL-005: Centralization Risk.
   d) V-TSS-VUL-006: Chainlink oracle may be stale.
   e) V-TSS-VUL-007: Reward tokens address is not enforced.
      i. This issue is partially fixed as `SiloStrategy` does enforce the value of the first array entry. `FarmStrategy` does not touch the `rewardTokens` except for setting the array value, so it should be removed from the contract.
   f) V-TSS-VUL-008: Two-step ownership transfer is preferable.
   g) V-TSS-VUL-010: Non-standard ERC20s are unsupported.

Impact The impacts of the various issues are detailed in the previous report.

Recommendation Implement the fixes as originally recommended in the previous report.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.4 V-TIE-VUL-004: reduceMLR() incorrect with flashloan fees

Severity	Medium	Commit	e072643
Type	Logic Error	Status	Fixed
File(s)	contracts/subStrategies/ETHStrategy.sol		
Location(s)	reduceMLR()		
Confirmed Fix At	d8ad2fa		

`reduceMLR()` performs the following actions to reduce the loan-to-collateral ratio to the desired value of `_mlr`.

1. Flashloan amount in base asset.
2. Repay amount of debt.
3. Withdraw exactly `Curve_get_dx(amount*(1+f))` collateral from aave (in deposit asset).
4. Swap via curve to receive `amount*(1+f)` in the base asset.
5. Pay back flash loan.

The flashloan `amount` is computed as follows

```
uint256 fee = IFlashloanReceiver(receiver).getFee();
uint256 feeParam = fee + magnifier;
uint256 amount =(debt-debtNew)/(feeParam-_mlr);
uint256 outValue = getOracleOut(address(depositAsset), address(baseAsset), amount);
uint256 aaveValue = IAavePool(IaavePool).convertAmount(address(depositAsset), address
    (baseAsset),amount);
uint256 calMlr = _mlr*aaveValue/outValue;
amount =(debt-debtNew)/(feeParam-calMlr);
```

Snippet 4.2: Snippet from `reduceMLR()`

This is incorrect. Instead of `(debt - debtNew) / (feeParam - calMlr)`, the `amount` should be `(debt - debtNew) / (magifier - (magnifier + fee) * calMlr)`.

This can be seen by the following example:

```
// Example:
// Assume for simplicity that 1 stETH = 1 WETH
// mlr of 50%: 1 WETH of debt and 2 stETH of collateral
// mlr -> 25% (fee f)
//
// (1) flashloan amount
// (2) repay amount of debt (new debt is 1 WETH - amount)
// (3) withdraw amount*(1+f) of stETH (new collateral is 2 stETH - amount - f* amount
    )
// (4) swap this for amount*(1+f) of WETH (assuming 1-1)
// (5) pay back flashloan
//
// 0.25 = (1 - amount) / (2 - amount - f*amount)
// 1/4 * (2 - amount * (1+f)) = (1 - amount)
// 1/2 - amount * (1+f)/4 = 1 - amount
// amount * (1 - (1+f)/4) = 1/2
// amount = 1 / (2 * (1 - (1+f)/4) )

```

```
// requirement: amount >= 0
//
// 0 <= 1 / (2 * (1 - (1+f)/4))
// \iff
// 0 <= 2 * (1 - (1+f)/4)
// \iff
// 0 <= 1 - (1+f)/4
// \iff
// (1+f)/4 <= 1
```

This matches the correct formula, in which the `amount` is undefined when `(1+f)/4 >= 1`. However, the incorrect formula in the implementation still claims to provide a solution for arbitrarily large fees.

Impact The incorrect MLR will be used.

Recommendation Use the correct formula.

Developer Response The developers have changed the formula to match the recommended fix.

4.1.5 V-TIE-VUL-005: Centralization Risk

Severity	Low	**Commit**	e072643
Type	Access Control	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Similar to many projects, TIE's Leverage On Aave protocol declares an administrator role which is given special permissions. In particular, these administrators are given the following abilities:

1. The owner of `farmSpark` can change the rewards controller and supported assets.
2. The owner of a `multiVault` may
   a) set the exchange used for swapping tokens.
   b) set the list of whitelisted tokens.
3. The owner of an `ETHStrategy` may
   a) set a fee rate up to 99.99%.
   b) repeatedly raise and reduce the MLR.
   c) set an exchange to a malicious contract to steal funds up to the slippage parameter.
   d) Set the MLR to a value exposing the vault to high risk of liquidation.

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, as mentioned above, a malicious owner could intentionally extract funds from the protocol by waiting for many users to deposit, and then setting the fee rate to 99.99%.

Recommendation As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance or multi-sig contract as opposed to a single account, which introduces a single point of failure.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.6 V-TIE-VUL-006: wstETH oracle returns inaccurate getRoundData

Severity	Low	**Commit**	e072643
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/subStrategies/oracle/WSTETHAggregator.sol`		
Location(s)	getRoundData()		
Confirmed Fix At	d8ad2fa		

The `wstETHAggregator` implements Chainlink's `AggregatorV3Interface`, and utilizes Chainlink's `stETH` oracle to determine the price of `wstETH`. It exposes the `getRoundData()` function, which is used to access historical data. However, this function still uses the current ratio of `wstETH` and `stETH`, which is constantly changing. Therefore, inaccurate historical price data is returned.

Impact Users of `getRoundData` for historical price records will receive inaccurate data, with a direct relationship between how old a record is and its inaccuracy.

Recommendation Either document to callers that this is intentional, or find a historical source of this data.

Developer Response The developers do not support the `getRoundData()` function and now revert whenever it is called, therefore the oracle can only return the latest answer.

4.1.7 V-TIE-VUL-007: CurveRouterExchangeETH cannot swap ETH

Severity	Low	**Commit**	e072643
Type	Data Validation	**Status**	Intended Behavior
File(s)	`contracts/subStrategies/exchange/CurveRouterExchangeETH.sol`		
Location(s)	swap()		
Confirmed Fix At	N/A		

The `CurveRouterExchange` and `CurveRouterExchangeETH` internally use the zero address to represent ETH until interacting with Curve, in which it substitutes the zero addresses with curve's representation. Therefore, when one wants to `swap()` ETH to another token, they must set `tokenIn` to `address(0)`. However, there are two issues with this:

1. The `swap()` function is not payable, therefore there is no way to send ETH for the swap except for sending with a separate call before calling this function.
2. The first line of the function (shown below) will cause a reversion, as `address(0)` is not an ERC20 token.

```
IERC20(tokenIn).safeTransferFrom(msg.sender,address(this),amount);
```

Snippet 4.3: Snippet from `subStrategies/exchange/CurveRouterExchangeETH.sol:swap()`

Even if one were to send ETH before calling `swap()`, (2) ensures that the swap cannot happen.

Impact One cannot swap ETH with this contract.

Recommendation If ETH swaps are intended to be supported, handle the case of ETH separately from the ERC20 transfers. Otherwise, consider requiring the `tokenIn` to not be ETH.

Developer Response The developers have indicated that only WETH should be used and not ETH.

4.1.8 V-TIE-VUL-008: Withdrawing has inadequate slippage protection

Severity	Low	**Commit**	e072643
Type	Frontrunning	**Status**	Fixed
File(s)	`contracts/core/multiTokenVault.sol`		
Location(s)	withdrawToken()		
Confirmed Fix At	d8ad2fa		

The `multiTokenVault` allows users to deposit and withdraw differing tokens than the `asset` natively supported by the `Vault`. It does so by swapping between the deposited/withdrawn token and the `asset`. However, as seen in the snippet below, the `withdrawToken()` and `redeemToken()` methods do not provide slippage protection with the `minWithdraw` variable if a user calls these functions with `token` equal to the underlying `Vault asset`.

```
uint256 amount = _withdraw(assets, shares,0, address(this));
if (token != address(asset)){
    approve(address(asset),exchange);
    amount = IExchange(exchange).swap(address(asset),token,amount,minWithdraw);
}
```

Snippet 4.4: Snippet from `core/multiTokenVault.sol:withdrawToken()`

Impact Users of these functions may suffer from front-running attacks in which they do not receive a fair amount of `tokens` compared to what their shares are worth.

Recommendation Add slippage protection, such as

```
else {
    require(amount >= minWithdraw);
}
```

Developer Response The developers implemented the recommended fix.

4.1.9 V-TIE-VUL-009: Precision loss in arithmetic

Severity	Low	**Commit**	e072643
Type	Logic Error	**Status**	Acknowledged
File(s)	`contracts/subStrategies/farmClaim.sol`		
Location(s)	_compound()		
Confirmed Fix At	N/A		

The following locations contain arithmetic which may be imprecise when handling small values.

1. The below snippet computes the amount of fees to be minted to the fee-pool. The division by `calDecimals` before the multiplication `_total` may cause a loss in precision.

```
uint256 _fee = curDeposit*farmFee/calDecimals;
_rebFee = bDepositFee ? curDeposit*compoundFee/calDecimals : 0;
uint256 _totalBalance = totalDeposit+curDeposit-_fee-_rebFee;
if(_fee>0){
    uint256 mintAmount = _total*_fee/_totalBalance;
```

Snippet 4.5: Snippet from `_compound()`

2. The below snippet computes the minimum amount to be expected for the given slippage parameter. The multiplication by `(calDecimals - slippage)` should be moved above the if-else statement.

```
uint256 amountOut = amount*price0;
if(decimals0+decimals00 > decimals1+decimals11){
    amountOut = amountOut/(10**(decimals0+decimals00-decimals1-decimals11));
}else{
    amountOut = amountOut*(10**(decimals1+decimals11-decimals0-decimals00));
}
return amountOut*(calDecimals-slippage)/price1/calDecimals;
```

Snippet 4.6: Snippet from `getMinOut()`.

Impact Small amounts of rewards may result in fewer fees to the fee pool than expected.

Recommendation

1. Perform the computation of `_fee`, `_rebFee`, and `_totalBalance` in `calDecimals`-many decimals, then divide by `calDecimals` when storing the final computation into `mintAmount`. Similarly, perform the same computation of the amount minted to the `receiver` when `bDepositFee == true`.
2. Move the multiplication by `(calDecimals - slippage)` to before the if-else statement.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.10 V-TIE-VUL-010: Compounding can be frontrun

Severity	Low	**Commit**	e072643
Type	Frontrunning	**Status**	Acknowledged
File(s)	`contracts/subStrategies/farmClaim.sol`		
Location(s)	_compound()		
Confirmed Fix At	N/A		

The `compound()` function in `farmClaim` can be frontrun as there is no guarantee that the tokens that are deposited in the corresponding `Vault` contributed to the strategy earning the rewards. For example, one could take a flashloan, deposit many times with `maxDeposit` tokens, and then call the `compound()` function. After their shares have gained value from compounding, they can then withdraw from the `Vault` and net a profit.

Note that this is only profitable if the user's share appreciation plus the `compoundFee` is greater than the cost of the withdraw fee plus Curve swapping fees.

Impact Users may extract value out of the strategy from rewards that were generated without their capital.

Recommendation Ensure proper fee configuration so that this is not a profitable avenue, in addition to frequently calling `compound()` so the stale reward amounts are low.

Developer Response The developers have acknowledged the issue and communicated that they will frequently call `compound()` so that this attack is not profitable.

4.1.11 V-TIE-VUL-011: Unused program constructs

Severity	Warning	**Commit**	e072643
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	d8ad2fa		

Description The following program constructs are unused:

- `subStrategies/exchange/CurveRouterExchange.sol`:
  - `poolInfo.route`.

Impact These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

Developer Response The developers now use the `route` field inside of `getRouteInfo()`.

4.1.12 V-TIE-VUL-012: Two-step ownership transfer is preferable

Severity	Warning	Commit	e072643
Type	Access Control	Status	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Many of the contracts in the protocol have ownership roles that have elevated permissions and perform important configurations. These contracts inherit from OpenZeppelin's `Ownable` contract, which is considered unsafe as ownership transfer is not confirmed before finalization.

The following contracts are effected: `ETHStrategy`, `farmClaim`, `Vault`.

Additionally, V-TSS-VUL-008 from the previous report details contracts that also are still effected. Lastly, other contracts that were not in-scope of the audit use `Ownable`.

Impact Specifying a wrong address for the ownership transfer will make the guarded administrative functions unavailable.

Recommendation It is recommended to use an `Ownable2Step` pattern with a two step ownership transfer, implemented with `Ownable2Step` contract from the OpenZeppelin library.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.13 V-TIE-VUL-013: getSTGPrice is vulnerable to flash loans

Severity	Warning	**Commit**	e072643
Type	Flashloan	**Status**	Acknowledged
File(s)	`contracts/subStrategies/oracle/STGAggregator.sol`		
Location(s)	getSTGPrice		
Confirmed Fix At	d8ad2fa		

The `getSTGPrice()` function calculates the price of STG simply by dividing the balance of USDC by the balance of STG in the Sushi pair, and multiplying by the Chainlink price of USDC. This is vulnerable to flashloans in which an attacker can manipulate the reserves of the sushi pair to manipulate the price returned by this oracle.

Impact The oracle can be easily attacked, and any contracts relying on this oracle for accurate price data may be vulnerable to attack. This issue is only a warning as the developers have indicated it will not be used in production, but would be labeled as a high issue otherwise.

Recommendation Depending on the chain, Chainlink offers `STG` price-feed that can be used. Outside of that a TWAP price oracle should be implemented. Care should be taken to also be resistant to multi-block MEV.

More generally, the `STGAggregator` is not ready for production use, and should be clearly marked as test-only. For example, historical price data is not implemented correctly.

Developer Response The developers have renamed the file to `STGAggregator_test.sol` to indicate this file is only used for testing.

4.1.14 V-TIE-VUL-014: Missing address zero-checks

Severity	Warning	**Commit**	e072643
Type	Data Validation	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	d8ad2fa		

Description The following functions take addresses as arguments, but do not validate that the addresses are non-zero:

- `CurveRouterExchange.sol`:
  - `constructor()`: `_curveRouter`.
- `CurveRouterExchangeETH.sol`:
  - `constructor()`: `_stETH`.
- `SavingDaiStrategy`.sol
  - `constructor()`: `_savingDai`.

Impact If zero is passed as the address, then various external contracts will not be set correctly.

Developer Response The developers now check for the zero address in the constructor of all 3 contracts.

4.1.15 V-TIE-VUL-015: Duplicate code

Severity	Warning	**Commit**	e072643
Type	Maintainability	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

The implementation of various behaviors in the protocol is observed to be duplicated, leading to maintainability difficulty and possible issues if the implementations were to diverge:

1. `CurveRouterExchange`:
   a) Several functions repeatedly use the expression `getSwapParams(getSwapPath(tokenIn,tokenOut))`. This pattern is generalized by `CurveRouterExchangeETH` using methods like `exchangeByPath()`.
2. `ETHStrategy.convertDepositToBase()`, `ETHStrategy.getOracleOut()` and `farmClaim.getMinOut()` all have near identical functionality.
3. `ETHStrategy.rewardsSwap()` manually checks for the `tokenIn` allowance, even though it could instead inherit from `saveApprove`.
4. `farmClaim.getMinOut()` and `farmStrategy.getMinOut()` have nearly identical implementations.

Impact It is more difficult to maintain the project, and components of the protocol may become out of sync due to requiring updating code in multiple places.

Recommendation Implement composability such that the duplicated functionality has a single source of implementation.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.16 V-TIE-VUL-016: Deprecated code

Severity	Warning	Commit	e072643
Type	Maintainability	Status	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	d8ad2fa		

Description The following program constructs are deprecated, or used for test-only purposes. These contracts should either be deleted, clearly marked as deprecated, or organized into different directories based on the version of the protocol.

1. `contracts/subStrategies/exchange/Exchange.sol`.
   a) In fact, the changes to this contract make it incompatible with the upgraded `EthStrategy`. This line in getCurve_dy reverts if `tokenOut != weth`, but `getFlashloanAmount()` calls `getCurve_dy` with `tokenOut` set to `address(depositAsset)`, which is usually `stETH` (and was even formerly named stETH).
2. `contracts/subStrategies/exchange/PolygonExchange.sol`.
   a) Similarly to `Exchange.sol`, usage of this contract may result in a completely unusable exchange.

Impact These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

Developer Response The developers marked the `Exchange.sol` and `ExchangePolygon.sol` files as deprecated.

4.1.17 V-TIE-VUL-017: Cross-contract reentrancies for tokens with hooks

Severity	Warning	**Commit**	e072643
Type	Reentrancy	**Status**	Fixed
File(s)	contracts/subStrategies/farmClaim.sol		
Location(s)	compound()		
Confirmed Fix At	d8ad2fa		

The `compound()` function may be called directly on any strategy which extends the `farmClaim` contract.

```
function compound(uint256 slippage,address receiver,bool bDepositFee) nonReentrant
    external {
```

Snippet 4.7: Signature of `farmClaim.compound()`.

All other flows in the protocol begin at the `Vault` contract, which has its own reentrancy protections. These two reentrancy guards will not affect each other, meaning that if an attacker gains control of the execution flow while calling `compound()`, they may call `withdraw()` or `deposit()` while the sub-strategy is in an intermediate state.

Impact The current configuration does not use tokens with receiver hooks, which is the only current means of performing this reentrancy. However, if future deployments do use tokens with hooks, this could become an attack vector for the protocol.

Recommendation Start the `compound()` process from the `Vault`, as done in other workflows. Use a `nonReentrant` guard there to prevent reentrancies.

Developer Response The `deposit()` and `withdraw()` functions in `ETHStrategy` and `lendingStrategy` now have `nonreentrant` modifiers. The `ETHStrategySpark` and `lendingStrategySpark` also put the `nonreentrant` modifier on the overriden `_beforeCompound()` function. Therefore the `deposit()` and `withdraw()` functions will share the same reentrancy protection as the `compound()` function.

4.1.18 V-TIE-VUL-018: Missing Solidity conventions/best practices

Severity	Warning	**Commit**	e072643
Type	Maintainability	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

The following breaks from coding conventions in Solidity make the repository less readable and more time-intensive to understand.

1. Natspec comments should be added to contracts and functions. This is **especially important** for undefined functions in abstract contracts. For example, the `farmClaim` contract defines seven functions which have no definition. Reasoning about their correct execution requires guesswork and cross-referencing multiple implementations across multiple different files.
2. All contract names should begin with a capital letter. Contracts (and their defining files) like `farmSpark`, `farmClaim`, `chainlinkOracle`, `lendingStrategy`, `lendingStrategySpark`, `operator`, `saveApprove`, `farmStrategy`, `stargateStrategy`, `curveExchange`, and `curveExchangeETH` should be renamed to `FarmSpark`, `FarmClaim`, `ChainlinkOracle`, `LendingStrategy`, etc.
3. Interfaces defined for interactions with third-party contracts do not contain any references to the contract being used. Additionally, the names do not always match the third-party interfaces. For example, `ICurveRouter` should be named `ICurveRouterNG`.
4. Some third-party interfaces are combined into a single interface. For example, the Chainlink aggregator V3 interface is combined with the V1 interface behind the single `AggregatorV3Interface` interface.
5. In `FarmClaim._compound()`, it would be more robust to calculate the fee based on the balance change before and after calling `claimRewards()`, as opposed to calculating it only based on the balance after said call.

Impact Over time, the repository becomes difficult to understand as a whole. This affects the future maintainability of the protocol as future developments may mistakenly introduce errors, or new developers may misunderstand the code.

Recommendation Follow the above Solidity conventions:

1. Add Natspec comments to all undefined functions in abstract contracts. Consider also adding Natspec comments to all contracts and critical functions (like `getFlashloanAmount()`, `_deposit()`, `_withdraw()`, or `_totalAssets()`).
2. Rename all contracts (and their defining files) to begin with a capital letter.
3. Provide a github permalink or docs link to each third-party interface being used. Rename the interfaces to match the names in the third-party documentation/source code.
4. Use a separate interface for each distinct third-party interface being used.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.19 V-TIE-VUL-019: Hard-coded constants

Severity	Warning	**Commit**	e072643
Type	Maintainability	**Status**	Fixed
File(s)	contracts/subStrategies/farmClaim.sol		
Location(s)	setFarmFee()		
Confirmed Fix At	d8ad2fa		

`setFarmFee()` allows the owner to set the farm and compound fees. Fees are currently represented in four decimals, as shown in the below snippet with `calDecimals = 10000`. A total maximum fee of `5000` (or 50%) is enforced on any changes to the farm and compound fee.

```
uint256 public constant calDecimals = 10000;
// ...
function setfarmFee(uint256 _farmFee,uint256 _compoundFee,uint256 slippage) public
    nonReentrant onlyOwner {
    _compound(slippage,getFeePool(),true);
    require(_farmFee+_compoundFee <5000, "INVALID_RATE");
```

Snippet 4.8: Snippet from `example()`

The interpretation of `5000` depends on the value stored in `calDecimals`. If `calDecimals` is increased or decreased, the value used to represent 50% will also need to change.

Impact If the contract configuration is changed before deployment or during an upgrade, developers may forget to also change the fee restriction.

Recommendation Compute the maximum fee as `calDecimals / 2` instead of hard-coding it to `5000`.

Developer Response The developers have implemented the fix as recommended.

4.1.20 V-TIE-VUL-020: Use of transfer() to send ETH

Severity	Warning	Commit	e072643
Type	Usability Issue	Status	Fixed
File(s)	`contracts/core/ethVault.sol`		
Location(s)	withdrawEth(), redeemEth()		
Confirmed Fix At	d8ad2fa		

`withdrawEth()` and `redeemEth()` both use `transfer()`, a method designed to transfer ETH without re-entrancies by restricting the gas to 2300. Using `transfer` instead of call is frequently recommended against, as many wallets require more gas than 2300 in their receive logic.

Impact Some users may be unable to interact with the protocol.

Recommendation Use `call` instead of `transfer`.

Developer Response The developers have created a `safeTransferETH()` function which uses the low-level call and replaced the instances of `transfer()` with said function.

Updated Veridise Response It would be best practice to introduce this shared function into a utility file. Nonetheless, the issue is resolved.

4.1.21 V-TIE-VUL-021: Vaults ignore withdrawn asset amount

Severity	Warning	**Commit**	e072643
Type	Logic Error	**Status**	Acknowledged
File(s)	`contracts/core/ethVault.sol`, `contracts/core/Vault.sol`		
Location(s)	redeemEth(), redeem()		
Confirmed Fix At	N/A		

The `redeemEth()` function returns the requested number of assets to withdraw instead of the actual `amount` withdrawn (shown in the below snippet) which is calculated post-fees.

```
function redeemEth(uint256 shares,uint256 minWithdraw,address payable receiver)
        external nonReentrant unPaused returns (uint256 assets)
    {
        // ...
        assets =
            (shares * IController(controller).totalAssets()) /
            totalSupply();
        // ...
        uint256 amount = _withdraw(assets, shares,minWithdraw, address(this));
```

Snippet 4.9: Snippet from `redeemEth()`

Impact Third-party protocols building on top of the result will not receive as much ETH as they expect based on the return value.

Recommendation Return `amount` instead of `assets` in both `ethVault.redeemEth()` and `Vault.redeem()`.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.22 V-TIE-VUL-022: reduceMLR() or raiseMLR() may be sandwiched

Severity	Warning	**Commit**	e072643
Type	Flashloan	**Status**	Intended Behavior
File(s)	`contracts/subStrategies/ETHStrategy.sol`		
Location(s)	reduceMLR(), raiseMLR()		
Confirmed Fix At	N/A		

`reduceMLR()`, `raiseMLR()`, and `getFlashloanAmount()` all perform similar computations to determine how large of a flash-loan is necessary to perform a deposit or change the loan-to-collateral ratio. For this, the relative valuation of the deposit and base assets in both Curve and AAVE are necessary. For example, the snippet from `getFlashloanAmount()` uses the below two variables:

```
uint256 outValue = IExchange(exchange).getCurve_dy(address(baseAsset), address(
    depositAsset), loanAmt+_amount);
uint256 aaveValue = IAavePool(IaavePool).convertAmount(address(baseAsset), address(
    depositAsset),loanAmt+_amount);
```

Snippet 4.10: Snippet from `getFlashloanAmount()`

Both are necessary for different purposes: the curve valuation is necessary to understand the value received when swapping, and the AAVE valuation is necessary to understand the value of the debt asset relative to the collateral asset. However, `reduceMLR()` and `raiseMLR()` use the `getOracleOut()` method, which relies on AAVE, instead of `getCurve_dy`.

```
uint256 outValue = getOracleOut(address(depositAsset), address(baseAsset), amount);
uint256 aaveValue = IAavePool(IaavePool).convertAmount(address(depositAsset), address
    (baseAsset),amount);
```

Snippet 4.11: Snippet from `reduceMLR()`.

If the Curve pool is in a manipulated state when the reduction/raise occurs, an incorrect value may be used leaving the strategy at a loan-to-collateral ratio different from the operator's desired target.

Impact A malicious block producer or MEV bundler may sandwich a call to `setMLR()` and leave the strategy in a state which is in near danger of liquidation.

Recommendation Use `IExchange.getCurve_dy` instead of `getOracleOut()`.

Developer Response Using `IExchange.getCurve_dy` will be sandwiched. Using ChainLink oracle will prevent sandwiching.

Updated Veridise Response We downgraded the issue to a Warning, since the developers are correct that, by passing in a slippage parameter in terms of bps instead of as minimum/maximum amounts, a non-manipulable third-party oracle must be used.

Since this oracle does not take into account the exact curve state or fees, it may still be beneficial to specify slippage protection using `minAmount/maxAmount` (computed by simulating the operations off-chain) instead of the current manner of passing a maximum bps tolerance. This may cost less gas, and will enable use of the more precise Curve oracle.

4.1.23 V-TIE-VUL-023: AAVE withdraw return ignored

Severity	Warning	**Commit**	e072643
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/subStrategies/ETHStrategy.sol`		
Location(s)	withdraw()		
Confirmed Fix At	d8ad2fa		

The `withdraw()` function withdraws from the underlying AAVE pool. The function returns the withdrawn amount, which should be used for the value passed into the swap.

```
IAave(aave).withdraw(address(depositAsset), withdrawAmount, address(this));
// Swap depositAsset to baseAsset
return IExchange(exchange).swap(address(depositAsset),address(baseAsset),
    withdrawAmount,0);
```

Snippet 4.12: Snippet from `withdraw()`

Impact If, for some unknown reason in the future, AAVE pools fail to allow withdrawals at a 1-1 rate, this function could break.

Recommendation Use the return amount from `withdraw()`.

Developer Response The developers now read the response from `IAave.withdraw()` and assign it to the `withdrawAmount` variable.

4.1.24 V-TIE-VUL-024: Chainlink oracle may be stale

Severity	Warning	**Commit**	e072643
Type	Data Validation	**Status**	Acknowledged
File(s)	contracts/substrategies/oracle/STGAggregator.sol		
Location(s)	getSTGPrice()		
Confirmed Fix At	d8ad2fa		

The `AggregatorV3Interface` of Chainlink returns a struct that contains an `updatedAt` parameter, which provides the last time that the oracle updated its data. However, the `getSTGPrice()` function does not consume this variable:

```
function getSTGPrice() internal view returns (int256){
        address usdc = sushiPair.token0();
        address stg = sushiPair.token1();
        (,int256 price,,,) = usdcOracle.latestRoundData();
        ...
}
```

Snippet 4.13: Snippet from `substrategies/oracle/STGAggregator.sol:getSTGPrice()`

Additionally, as mentioned in V-TIE-VUL-003, `chainLinkOracle` still does not check this parameter.

Impact If an Oracle becomes stale (no longer updated, or significantly out of date), then the oracle may return an inaccurate price. This can have undesirable effects on the consumers relying on this oracle. This issue is only a warning as the developers have indicated it will not be used in production, but would be labeled as a low issue otherwise.

Recommendation Check the `updatedAt` parameter to ensure it is within a reasonable time window relative to the current transaction.

Developer Response The developers have marked this file as test only, and therefore do not care about staleness.

4.1.25 V-TIE-VUL-025: Gas Optimization

Severity	Info	**Commit**	e072643
Type	Gas Optimization	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

In the following locations, the auditors identified missed opportunities for potential gas savings.

1. `core/multiTokenVault.sol`:
   a) Instead of iterating over the `groupToken` array, it would be more efficient to use a mapping.
2. `core/Vault.sol`:
   a) `_deposit()`: Consider using mulDiv(uint256,uint256,uint256) (which rounds down automatically) instead of passing `Math.Rounding.Down` into the more generic `mulDiv(uint256,uint256,uint256,Rounding)` implementation which must handle both rounding cases.
3. `subStrategies/oracle/WSTETHAggregator.sol`:
   a) The `stEthAggregator` and `wstETH` variables should be immutable.
4. `subStrategies/exchange/curveExchangeETH.sol`:
   a) The `stETH` and `wstETH` variables should be immutable.
   b) `getSwapPath()`: Consider moving the requirement that `routeInfo[index] <= 5` to `setSwapRoute()` to avoid checking the invariant during every swap.
5. `subStrategies/exchange/CurveRouterExchangeETH.sol`:
   a) `getSwapPathETH()`: Using nested if statements is more gas efficient than `if (curRoute[i] == address(0) && curRoute[i+1] == stETH)`.
   b) `getSwapPath()`: The syntax `(a, b) = (b, a)` will swap more efficiently.
   c) `getSwapPath()`: `require(a); require(b);` is more gas-efficient than `require(a && b)`.
6. `subStrategies/exchange/UniExchangeV3.sol`:
   a) `getSwapPath()`: The syntax `(a, b) = (b, a)` will swap more efficiently.
   b) `getSwapPath()`: `require(a); require(b);` is more gas-efficient than `require(a && b)`.
7. `subStrategies/farmClaim.sol`
   a) `_compound()`: Using nested if statements is more gas efficient than `if(!bDepositFee && _rebFee>0)`.
8. `subStrategies/farmSpark.sol`
   a) `claimRewards()`: Storing `rewardsList.length` in a variable outside the loop will avoid repeated storage reads.

9. `subStrategies/lendingStrategy.sol`:
    a) `harvested`: Explicitly initializing an uninitialized variable with its default value costs unnecessary gas during deployment.
10. General recommendations:
    a) Consider using custom errors rather than reverting with a string. These are more gas efficient, and the developer may view the error in detail using the NatSpec comments.
    b) Consider using `++i` instead of `i++`, as the former uses less gas than the latter.

Impact Gas may be wasted, costing users extra funds.

Recommendation Perform the optimizations.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.26 V-TIE-VUL-026: Typos and incorrect comments

Severity	Info	**Commit**	e072643
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	d8ad2fa		

Description In the following locations, the auditors identified minor typos:

1. `subStrategies/farmClaim.sol`:
   a) `depositToken()`: Consider renaming this function to avoid naming collisions with other contracts (like the multi-token vault).
2. `subStrategies/farmSpark.sol`:
   a) `setRawardsInfo()`: Should be r**e**wards instead of r**a**wards.

Impact These minor errors may lead to future developer confusion.

Developer Response The developers have renamed `depositToken()` to `depositFunds()` for `farmClaim` and the contracts that derive it. They also fixed the typo in `farmSpark`.

A Appendix

A.1 Code Coverage

Code coverage was obtained by rewriting the tests from `https://github.com/Tie-Finance/Tie-Finance-ETH_LeverageOnAave/tree/e0726433` into Foundry tests, then running the command `forge coverage --no-match-coverage "(script|contracts/mock|test)"`.

File	% Lines	% Statements	% Branches	% Funcs
contracts/core/Controller.sol	93.55% (29/31)	91.67% (33/36)	69.23% (9/13)	91.67% (11/12)
contracts/core/Vault.sol	61.40% (35/57)	63.89% (46/72)	29.55% (13/44)	50.00% (9/18)
contracts/core/ethVault.sol	100.00% (23/23)	100.00% (26/26)	50.00% (12/24)	100.00% (4/4)
contracts/core/multiTokenVault.sol	0.00% (0/44)	0.00% (0/49)	0.00% (0/38)	0.00% (0/7)
contracts/periphery/EthStrategyLens.sol	0.00% (0/28)	0.00% (0/50)	0.00% (0/3)	0.00% (0/3)
contracts/periphery/lendingStrategyLens.sol	0.00% (0/39)	0.00% (0/67)	0.00% (0/5)	0.00% (0/4)
contracts/rebalance/aaveRebalance.sol	0.00% (0/38)	0.00% (0/45)	0.00% (0/19)	0.00% (0/8)
contracts/subStrategies/ETHStrategy.sol	55.91% (123/220)	53.75% (172/320)	36.23% (25/69)	81.82% (27/33)
contracts/subStrategies/ETHStrategySpark.sol	0.00% (0/9)	0.00% (0/13)	0.00% (0/1)	12.50% (1/8)
contracts/subStrategies/SavingDaiStrategy.sol	0.00% (0/13)	0.00% (0/19)	0.00% (0/1)	0.00% (0/3)
contracts/subStrategies/Silo/SiloStrategy.sol	0.00% (0/45)	0.00% (0/66)	0.00% (0/7)	0.00% (0/13)
contracts/subStrategies/Silo/farmStrategy.sol	0.00% (0/91)	0.00% (0/126)	0.00% (0/45)	0.00% (0/19)
contracts/subStrategies/aavePool/aavePoolV2.sol	0.00% (0/44)	0.00% (0/73)	0.00% (0/6)	0.00% (0/11)
contracts/subStrategies/aavePool/aavePoolV3.sol	0.00% (0/43)	0.00% (0/71)	0.00% (0/8)	0.00% (0/11)
contracts/subStrategies/exchange/CurveRouterExchange.sol	0.00% (0/53)	0.00% (0/75)	0.00% (0/14)	0.00% (0/11)
contracts/subStrategies/exchange/CurveRouterExchangeETH.sol	0.00% (0/41)	0.00% (0/65)	0.00% (0/7)	0.00% (0/9)
contracts/subStrategies/exchange/Exchange.sol	0.00% (0/58)	0.00% (0/73)	0.00% (0/35)	0.00% (0/7)

contracts/subStrategies/exchange/ExchangePolygon.sol	0.00% (0/33)	0.00% (0/49)	0.00% (0/23)	0.00% (0/7)
contracts/subStrategies/exchange/UniExchange.sol	0.00% (0/26)	0.00% (0/26)	0.00% (0/15)	0.00% (0/5)
contracts/subStrategies/exchange/UniExchangeV3.sol	0.00% (0/50)	0.00% (0/68)	0.00% (0/15)	0.00% (0/9)
contracts/subStrategies/exchange/curveExchange.sol	0.00% (0/127)	0.00% (0/158)	0.00% (0/75)	0.00% (0/13)
contracts/subStrategies/exchange/curveExchangeETH.sol	0.00% (0/39)	0.00% (0/47)	0.00% (0/18)	0.00% (0/5)
contracts/subStrategies/farmClaim.sol	0.00% (0/43)	0.00% (0/72)	0.00% (0/14)	0.00% (0/5)
contracts/subStrategies/farmSpark.sol	0.00% (0/14)	0.00% (0/18)	0.00% (0/2)	0.00% (0/3)
contracts/subStrategies/lendingStrategy.sol	50.29% (88/175)	51.25% (123/240)	26.87% (18/67)	66.67% (16/24)
contracts/subStrategies/lendingStrategySpark.sol	0.00% (0/12)	0.00% (0/16)	0.00% (0/3)	11.11% (1/9)
contracts/subStrategies/loanReceivers/BalancerReceiver.sol	0.00% (0/20)	0.00% (0/22)	0.00% (0/6)	0.00% (0/6)
contracts/subStrategies/operator.sol	0.00% (0/4)	0.00% (0/4)	0.00% (0/4)	0.00% (0/2)
contracts/subStrategies/oracle/STGAggregator.sol	0.00% (0/18)	0.00% (0/30)	100.00% (0/0)	0.00% (0/11)
contracts/subStrategies/oracle/WSTETHAggregator.sol	0.00% (0/16)	0.00% (0/26)	100.00% (0/0)	0.00% (0/12)
contracts/subStrategies/oracle/chainLinkOracle.sol	0.00% (0/8)	0.00% (0/9)	0.00% (0/2)	0.00% (0/4)
contracts/subStrategies/saveApprove.sol	0.00% (0/3)	0.00% (0/4)	0.00% (0/3)	0.00% (0/1)
Total	20.34% (298/1465)	19.66% (400/2035)	13.14% (77/586)	23.23% (69/297)

Glossary

AAVE Aave is an Open Source Protocol to create Non-Custodial Liquidity Markets to earn interest on supplying and borrowing assets. To learn more, visit `https://aave.com`. 1, 2, 5, 41

AMM Automated Market Maker. 40, 41

Balancer An AMM suite extended with functionality for several other popular DeFi actions. See `https://balancer.fi` to learn more. 2

Chainlink An on-chain solution for off-chain data-sources, most commonly price feeds. See `https://chain.link` to learn more. 1, 2

Curve An DEX designed for highly efficient stablecoin trading. See `https://docs.curve.fi` to learn more. 1, 2

DEX Decentralized Exchange. 40

ERC Ethereum Request for Comment. 40

ERC-20 The famous Ethereum fungible token standard. See `https://eips.ethereum.org/EIPS/eip-20` to learn more. 41

ERC-4626 An Ethereum Request for Comment (ERC) describing a tokenized vault representing shares of an underlying ERC20 token. See `https://eips.ethereum.org/EIPS/eip-4626` for the full ERC. 1

Ethereum Request for Comment Peer-reviewed proposals describing application-level standards and conventions. Visit `https://eips.ethereum.org/erc` to learn more. 40

Foundry An Ethereum development environment which uses Solidity-native utilities to compile, test, and deploy EVM contracts. See `https://book.getfoundry.sh` to learn more. 38

Lido A Liquid Staking protocol. See `https://lido.fi` to learn more. 1, 2, 40

Lido stETH Token The rebasable Liquid Staking protocol issued by Lido. See `https://docs.lido.fi/guides/lido-tokens-integration-guide#steth` to learn more. 41

Liquid Staking A tokenized contract representing shares of a staked native currency. See `https://chain.link/education-hub/liquid-staking` to learn more. 40

Savings Dai Savings Dai is a tokenized representation of Dai deposited in the Dai Savings Rate (DSR) offered by Sky. See `https://docs.spark.fi/user-guides/earning-savings/sdai-overview` to learn more. 40

sDAI Savings Dai. 1

Semgrep Semgrep is an open-source, static analysis tool. See `https://semgrep.dev` to learn more. 5

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 1, 40

Solidity The standard high-level language used to develop smart contracts on the Ethereum blockchain. See `https://docs.soliditylang.org/en/v0.8.19/` to learn more. 3, 40

Spark A lending platform and AAVE fork built into the Sky ecosystem. See `https://devs.spark.fi` to learn more. 1, 5

stETH Lido stETH Token. 1

Uniswap One of the most famous deployed AMMs. See `https://uniswap.org` to learn more. 2

WBTC Wrapped Bitcoin. 1

WETH Wrapped Ethereum. 1

Wrapped Bitcoin A ERC-20 token issued at a 1:1 exchange rate with Bitcoin. 41

Wrapped Ethereum A ERC-20 token issued at a 1:1 exchange rate with Ethereum. 41

