Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

TIE FINANCE

ETH Leverage Vault

Veridise Inc.

December 3, 2023

► **Prepared For:**

TIE Finance
https://www.tie-finance.io/

► **Prepared By:**

Jon Stephens
Alberto Gonzalez

► **Contact Us:** contact@veridise.com

► **Version History:**

Nov. 7, 2023 V1

Contents

1 Executive Summary

From Nov. 1, 2023 to Nov. 6, 2023, TIE Finance engaged Veridise to review the security of their ETH Leverage Vault. The review covered the Solidity code associated with the vault contract and the investment logic. Veridise conducted the assessment over 8 person-days, with 2 engineers reviewing code over 4 days on commit `d2d7f10`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The ETH Leverage Vault developers provided the source code of the project for review. To facilitate the Veridise auditors' understanding of the code, the developers provided high-level documentation of the project and an Ethereum address where they deployed the current version of their code for testing purposes. The source code contained some in-line documentation to describe the intended behavior of the protocol.

The Veridise auditors made use of the test deployment provided by the developers to augment their understanding of the source code. While this deployment was very helpful in demonstrating some of the logic as well as the intended initialization of the contract, the Veridise auditors did note that not all major user-flows were tested here. At the time of the audit, for example, the only transactions sent to this address were deposits and no funds were withdrawn.

Summary of issues detected. The audit uncovered 23 issues, 3 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, V-TLA-VUL-001 identified a potential Denial of Service attack that could prevent funds from being deposited or withdrawn, V-TLA-VUL-002 identified inconsistent minting logic that gave users too many shares, and V-TLA-VUL-003 identified the potential for users to be credited with funds they did not deposit. The Veridise auditors also identified several medium-severity issues, including V-TLA-VUL-007 which identifies a read-only reentrancy that allows share values to be incorrect, V-TLA-VUL-004 which identifies potential issues when transferring tokens and V-TLA-VUL-008 which could charge users too many fees. In addition, the Veridise auditors identified several lower-severity issues.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the ETH Leverage Vault. As mentioned previously, the project was deployed on mainnet with the developers and tested by interacting with that contract. However, it seemed that they only tested the deposit logic of this contract and not the withdraw logic. We would advise that the developers test all major user-flows and also consider making use of an off-chain testing framework like hardhat, truffle or foundry. By doing so, they could model scenarios that might occur in the future such as high AAVE interest rates.

In addition, the Veridise auditors would recommend that the developers closely track the flow of funds throughout the protocol. Currently, several locations make use of a contract's entire ETH or token balance, which could have unintended consequences as discussed in some of the

issues in this report. To do so, the developers should make use of payable functions for ETH and approvals along with transferFrom for ERC20 tokens so that they can track explicitly how many funds are available and expected.

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
ETH Leverage Vault	`d2d7f10`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Nov. 1 - Nov. 6, 2023	Manual & Tools	2	8 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	1	1
High-Severity Issues	2	2
Medium-Severity Issues	7	7
Low-Severity Issues	6	6
Warning-Severity Issues	7	7
Informational-Severity Issues	0	0
TOTAL	23	23

Table 2.4: Category Breakdown.

Name	Number
Logic Error	6
Dead Code	6
Data Validation	3
Maintainability	2
Liquidation Risk	1
Reentrancy	1
Phishing	1
Centralization	1
Locked Funds	1
Usability Issue	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of ETH Leverage Vault's smart contracts. In our audit, we sought to answer the following questions:

- Can funds be locked in a contract?
- Are users appropriately compensated when depositing funds?
- Can users steal funds from the protocol?
- Will shares be appropriately converted to assets on a withdraw?
- Are appropriate protections in place to prevent a liquidation on AAVE?
- Will the protocol report the correct share value?
- Are users appropriately protected from slippage?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this audit is limited to the `contracts` folder of the source code provided by the ETH Leverage Vault developers, which contains the vault, investment strategy and contracts to interact with Balancer, Curve and AAVE.

Methodology. The Veridise auditors inspected the high-level documentation provided by the developers and inspected the on-chain test deployment of the protocol located at the following address: `0x3D2c816018BA19b436EE3c2AEf11214fC9Dbb38B`. They then began a manual audit of the code assisted by static analyzers.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

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

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

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

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-TLA-VUL-001	Vault deposits through curve swaps can get DoSed	Critical	Fixed
V-TLA-VUL-002	Incorrect computation of shares to mint	High	Fixed
V-TLA-VUL-003	User may be Credited with Left-Over Funds	High	Fixed
V-TLA-VUL-004	TransferHelper can hide transfer problems	Medium	Fixed
V-TLA-VUL-005	Use Payable and TransferFrom rather than Transf. .	Medium	Fixed
V-TLA-VUL-006	Funds Risk AAVE Liquidation	Medium	Intended Behavior
V-TLA-VUL-007	Withdraw Read Only Reentrancy	Medium	Fixed
V-TLA-VUL-008	Collect the fee before changing the fee rate	Medium	Fixed
V-TLA-VUL-009	Missing slippage protection for users	Medium	Fixed
V-TLA-VUL-010	Missing slippage protection in the withdraw fun. .	Medium	Fixed
V-TLA-VUL-011	Potential Invalid use of tx.origin	Low	Fixed
V-TLA-VUL-012	Use Token Decimals Instead of Hardcoding	Low	Fixed
V-TLA-VUL-013	Inconsistent Deposit Logic	Low	Invalid
V-TLA-VUL-014	Validate Function Arguments	Low	Fixed
V-TLA-VUL-015	Validate Withdraw Receiver	Low	Fixed
V-TLA-VUL-016	Centralization Risk	Low	Acknowledged
V-TLA-VUL-017	Unused Whitelist Contract	Warning	Fixed
V-TLA-VUL-018	Unused Contract Variable	Warning	Fixed
V-TLA-VUL-019	Unused Internal Function	Warning	Fixed
V-TLA-VUL-020	Use WETH deposit instead of fallback	Warning	Fixed
V-TLA-VUL-021	Unused contract variables in ETHStrategy	Warning	Fixed
V-TLA-VUL-022	Unused Inherits	Warning	Fixed
V-TLA-VUL-023	Incorrect price returned due to lack of decimal. . .	Warning	Fixed

4.1 Detailed Description of Issues

4.1.1 V-TLA-VUL-001: Vault deposits through curve swaps can get DoSed

Severity	Critical	**Commit**	d2d7f10
Type	Logic Error	**Status**	Fixed
File(s)	`Exchange.sol and ExchangePolygon.sol`		
Location(s)	swapStETH()		
Confirmed Fix At			

The `swapSteth` function contains logic to swap `ether` for `stETH` in a CurvePool, the logic looks like:

```
else {
      require(curveOut>=minAmount,"ETH_STETH_SLIPPAGE");
      ICurve(curvePool).exchange{value: address(this).balance}(
          0,
          1,
          amount,
          minAmount
     );
}
```

Snippet 4.1: Logic to swap `ether` for `stETH` in the `swapSteth` function from the `Exchange` contract.

As we can see, the logic sends to the pool all its ether balance, while passing `amount` as an actual parameter for the `dx` argument. The issue is that is possible for `address(this).balance` to become greater than `amount`. This can happen, for example by a malicious user sending 1 wei to the exchange contract.

The Curve Pool exchange logic has the following lines:

```
if _coin == 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE:
        assert msg.value == dx
```

Snippet 4.2: Code snippet from the `exchange` function in the Curve eth-stETH pool contract.

Reference: `https://etherscan.io/address/0xc5424b857f758e906013f3555dad202e4bdb4567#code`

LoC: 430

Impact By sending 1 wei to the exchange contract, an attacker can brick the swap functionality of the contract.

The same issue is present in the `ExchangePolygon` contract.

Reference: `https://polygonscan.com/address/0x5bca7ddf1bcccb2ee8e46c56bfc9d3cdc77262bc#code`

Recommendation Instead of sending `address(this).balance` ether, send `amount`.

4.1.2 V-TLA-VUL-002: Incorrect computation of shares to mint

Severity	High	**Commit**	d2d7f10
Type	Logic Error	**Status**	Fixed
File(s)	`Vault.sol`		
Location(s)	deposit()		
Confirmed Fix At			

The `shares` to mint are computed as follows:

```
shares = totalSupply() == 0 || totalDeposit == 0
            ? assets.mulDiv(
                10 ** decimals(),
                10 ** asset.decimals(),
                Math.Rounding.Down
            )
            : newDeposit.mulDiv(
                totalSupply(),
                totalDeposit,
                Math.Rounding.Down
            );
```

Snippet 4.3: Computation of `shares` in the `deposit` function from the `Vault` contract.

We are interested on the else branch. Here, the `shares` are computed using the current `totalSupply()` and `totalDeposit`. The value of the `totalDeposit` variable is obtained as follows:

```
// Total Assets amount until now
uint256 totalDeposit = IController(controller).totalAssets();

// Calls Deposit function on controller
uint256 newDeposit = IController(controller).deposit(assets);
```

Snippet 4.4: Code snippet from the `deposit` function in the `Vault` contract.

`totalDeposit` is obtained before processing the user's deposit, that means that the corresponding assets for the fees were deducted from `totalAssets`.

Returning to the `shares` computation, we can see the inconsistency that `totalSupply()` takes into consideration the shares minted to the treasury when collecting the fees, whereas `totalDeposit` considers this share not minted yet.

Impact Since `totalSupply()` is greater than it should be in relation to `totalDeposit`, then the logic will mint more `shares` to the user than it should. Lets see a numeric example:

- `TotalShares = 99`
- `TotalAssets = 99 + 1(reward)`
- `fee = 50%`

When a user calls `deposit`, depositing 10 of assets:

- `newDeposit = 100 - 1 * 0.5 = 99.5`

The protocol will collect the fee of 0.5 a mint a share for it. The amount of shares minted are:

- `shareFee = (0.5 * 99) / (100 - 0.5) = 0.4974 shares`

When the user shares are computed, they are with the following values:

- `userShares = (10 assets * 99.4974 shares) / 99.5 assets = 10 shares`

The user deposited 10 assets and received 10 shares which is wrong, the 10 shares are worth:

- `totalAssets = 110 assets`
- `totalShares = 109.4974 shares`
- `assetsWorth = 10 shares * 110 assets / 109.4974 shares = 10.047 assets`

The 10 shares are worth 10.047 assets, which is 4.7% more of what they should be worth (10 assets).

Recommendation Compute the `shares` to mint using the `totalSupply` before minting the share fee for the treasury.

4.1.3 V-TLA-VUL-003: User may be Credited with Left-Over Funds

Severity	High	**Commit**	d2d7f10
Type	Logic Error	**Status**	Fixed
File(s)	`ETHStrategy.sol`		
Location(s)	loanFallback, _deposit, withdraw, raiseLTV, reduceLTV		
Confirmed Fix At			

Instead of calculating or returning the expected number of funds, the protocol commonly uses its entire token balance, as shown in the example below. This approach can lead to funds being misappropriated, as the contract's starting balance is not taken into account when calculating a user's credit. Consequently, if the contract has a non-zero balance at the start, it will be credited to the next user who utilizes the protocol. Such a non-zero starting balance could occur if a previous transaction did not make use of all available funds or if funds are accidentally sent to the contract.

```
function withdraw(
    uint256 _amount
) external override onlyController collectFee returns (uint256) {
    ...

    uint256 toSend = address(this).balance;
    TransferHelper.safeTransferETH(controller, toSend);

    return toSend;
}
```

Snippet 4.5: Location in `withdraw` where the contract's entire balance is transferred

Additionally, several contracts declare a `receive` function that allows any user to transfer native tokens to the contract, allowing users to be credited for others' mistakes.

Impact If funds are left-over in the `ETHStrategy` contract, they could allow a user to be improperly credited for those funds. Additionally, due to both this pattern and the one described in V-TLA-VUL-005, it can be difficult to follow the flow of funds through the protocol.

Recommendation Explicitly track the expected amount of funds that a user should receive rather than relying on token balances. In addition, since it is known where most contracts should receive funds, restrict the addresses that may invoke `receive` (or if `payable` functions are used as suggested by V-TLA-VUL-005 remove `receive` altogether)

4.1.4 V-TLA-VUL-004: TransferHelper can hide transfer problems

Severity	Medium	**Commit**	d2d7f10
Type	Logic Error	**Status**	Fixed
File(s)	`TransferHelper.sol`		
Location(s)	safeApprove, safeTransfer, safeTransferToken, safeTransferFrom, safeTransferETH		
Confirmed Fix At			

The AAVE strategy uses the TransferHelper library to transfer ERC20 and ETH tokens between addresses similar to OpenZeppelin's SafeERC20 library. While it is intended to safely send tokens between users, several features that may hide problems when sending funds.

The first potential issue, is that if `token` is the null address, `safeTransfer` will instead send native tokens. Since the protocol commonly interacts with a mixture of native and ERC20 tokens, however, a configuration error could result in the incorrect token being sent to a destination.

```
function safeTransfer(
    address token,
    address to,
    uint256 value
) internal {
    if (address(token) == address(0)) {
        safeTransferETH(to, value);
    } else {
        safeTransferToken(address(token), to, value);
    }
}
```

Snippet 4.6: The `safeTransfer` function which sends native tokens if the `token` is the null address.

The second potential issue is that the library assumes that `token` refers to a contract. As shown below, the `safeTransferToken` (along with `safeApprove` and `safeTransferFrom`) use a low-level call to invoke the desired function. In cases where `token` refers to an EOA rather than a contract, though, low-level calls return with `success = true` and no calldata. This result will be accepted by the following require, and so the caller will continue as if was successful.

```
function safeTransferToken(
    address token,
    address to,
    uint256 value
) internal {
    // bytes4(keccak256(bytes('transfer(address,uint256)')));
    (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0xa9059cbb,
     to, value));
    require(success && (data.length == 0 || abi.decode(data, (bool))), "
    TransferHelper: TRANSFER_FAILED");
}
```

Snippet 4.7: The definition of the `safeTransferToken` function.

Impact As discussed above, in some cases this can result in the wrong token being sent to a destination and in other cases it can cause the protocol to believe tokens were transferred even though they weren't.

Recommendation Ideally use OpenZeppelin's Address and SafeERC20 libraries which already have the appropriate validation. Otherwise, since in all cases the protocol knows if it should transfer ETH or ERC20 tokens, remove the null address case in `safeTransfer` so that ETH and ERC20 transfers are explicit. Additionally require that `token` is a contract (i.e. has a non-zero code size) in `safeTransferToken`, `safeTransferFrom` and `safeApprove`.

4.1.5 V-TLA-VUL-005: Use Payable and TransferFrom rather than Transfer then Call

Severity	Medium	**Commit**	d2d7f10
Type	Maintainability	**Status**	Fixed
File(s)	`Vault.sol`, `Controller.sol`, `ETHStrategy.sol`, `Exchange.sol`, `ExchangePolygon.sol`		
Location(s)	N/A		
Confirmed Fix At			

Throughout the protocol, the developers commonly transfer funds to a contract's address, then invoke a function that will process those funds. Using this pattern, shown below, rather than making use of payable functions and `ERC20.transferFrom` is more error prone as one either must perform additional validation to ensure the correct number of funds is present or must make use of the contract's entire token balance, both of which is done by the protocol.

```
function _deposit(uint256 _amount) internal returns (uint256 depositAmt) {
        // Transfer asset to substrategy
    TransferHelper.safeTransferETH(subStrategy,_amount);

        // Calls deposit function on SubStrategy
    depositAmt = ISubStrategy(subStrategy).deposit(_amount);
}
```

Snippet 4.8: Definition of `_deposit` which uses the transfer then call strategy

Impact This pattern can cause users to be credited too few or too many funds (similar to V-TLA-VUL-003).

Recommendation Make user of payable and `transferFrom` rather than transfer-then-call to ensure funds are properly tracked.

4.1.6 V-TLA-VUL-006: Funds Risk AAVE Liquidation

Severity	Medium	**Commit**	d2d7f10
Type	Liquidation Risk	**Status**	Intended Behavior
File(s)	`ETHStrategy.sol`		
Location(s)	N/A		
Confirmed Fix At			

When users invest their funds with the AAVE investment strategy, the strategy will supplement the funds with a variable rate ETH loan from AAVE, swap all the ETH for stETH (or wstETH) and finally provide the stETH to AAVE as collateral. This will therefore generate yields of `(stETH Yield) + (AAVE Supply Interest) - (WETH Loan Interest)` as long as `(WETH Loan Interest)` is less than `(stETH Yield) + (AAVE Supply Interest)`. Since the interest rate increases with the utilization, however, the WETH Loan the strategy could lose money and the value of the WETH loan could exceed the value of the collateral, leading to liquidation. Currently the only thing done to protect against this eventuality is allow the owner to rebalance the loan back to the target non-zero LTV since funds will remain in AAVE between user deposits and withdraws. In cases where the interest rate is high, however, this will stall the liquidation but in these cases it would be useful to reduce the loan value to zero.

Impact If the AAVE loan is liquidated, the stETH collateral will be lost. Since all funds borrowed from AAVE are re-invested back into AAVE as stETH, this would effectively cause all funds to be lost.

Recommendation Consider including a mechanism to reduce the AAVE loan to zero so that high borrow interest rates can be avoided. Also, the current mechanism requires that the admin monitor the borrow interest rates. Consider allowing users to do so as well if the interest rate exceeds some threshold or if the health factor drops below some threshold.

Developer Response Our plan to avoid liquidation on AAVE is as follows:

1. Monitor the vault's liquidation risk to avoid borrowing too many funds. Additionally, we will keep a safety buffer to reduce the liquidation risk.
2. If there is an extended period of high borrowing interest rates that exceeds the interest received from stETH, we will adjust the MLR to repay a portion of the loan. In cases where the rate is extremely high, we will adjust this value so that our loan is very small.

4.1.7 V-TLA-VUL-007: Withdraw Read Only Reentrancy

Severity	Medium	**Commit**	d2d7f10
Type	Reentrancy	**Status**	Fixed
File(s)	`Vault.sol`		
Location(s)	withdraw		
Confirmed Fix At			

Upon a withdraw, the protocol will remove a user's funds from AAVE, transfer them to the user and then burn the user's tokens. A low level call is used to transfer these funds to the user, though, which will allow them to perform arbitrary actions before their tokens are burnt. While the `ethVault` contract does have reentrancy guards on available state-modifying entry-points, users may still re-enter through view functions.

```
function _withdraw(
    uint256 assets,
    uint256 shares,
    address receiver
) internal {
    require(shares != 0, "SHARES_TOO_LOW");
    // Calls Withdraw function on controller
    (uint256 withdrawn, uint256 fee) = IController(controller).withdraw(
        assets,
        receiver
    );
    require(withdrawn > 0, "INVALID_WITHDRAWN_SHARES");

    // Burn shares amount
    _burn(msg.sender, shares);

    ...
}
```

Snippet 4.9: Definition of the `_withdraw` function which transfers control to `receiver` before burning

Impact If a user were to call `assetsPerShare`, `convertToShares` or `convertToAssets`, the reported value would not accurately reflect the state of the protocol because the assets were reduced while the shares have not been. Therefore, if a pricing oracle for the secondary market used these values, it would allow someone to buy shares at a discount.

Recommendation Burn the shares before withdrawing the funds so that the transfer occurs at the end of the withdraw request.

4.1.8 V-TLA-VUL-008: Collect the fee before changing the fee rate

Severity	Medium	**Commit**	d2d7f10
Type	Logic Error	**Status**	Fixed
File(s)	`ETHStrategy.sol`		
Location(s)	setFeeRate()		
Confirmed Fix At			

The `ETHStrategy` contract collects fees periodically via the modifier `collectFee`, which internally computes the fee to take for the treasury using the function `_calculateFee`.

The fee is computed as follows:

```
1 uint256 stFee = (currentAssets-lastTotal) * feeRate / magnifier;
```

Snippet 4.10: Computation of the fee in the `_calculateFee` function from the `ETHStrategy` contract.

Impact In the current code, the logic on the `setFeeRate` function will change the value of `feeRate` without collecting the pending fees. This will cause the contract to either get more or less fees of what it should.

Recommendation Collect the pending fees before changing the `feeRate` variable.

4.1.9 V-TLA-VUL-009: Missing slippage protection for users

Severity	Medium	**Commit**	d2d7f10
Type	Data Validation	**Status**	Fixed
File(s)	`Vault.sol`		
Location(s)	deposit() withdraw() redeem()		
Confirmed Fix At			

The `deposit`, `withdraw` and `redeem` functions in the `Vault` contract lack some high-level slippage protection for users. Slippage protection allows users to specify the minimum amount of `shares` to receive when depositing `assets`. Or to specify the minimum amount of `assets` to receive when redeeming some `shares`.

Impact Users transactions might get confirmed in the network in a different contract state that might not be desired for them. For example, `shares` may have become more expensive when depositing, or `shares` may have become cheaper when withdrawing.

Recommendation Allow users to specify the minimum amount of `shares` they expect to get minted or the minimum amount of `assets` they expect to receive when burning some shares.

4.1.10 V-TLA-VUL-010: Missing slippage protection in the withdraw function of the ETHStrategy contract

Severity	Medium	**Commit**	d2d7f10
Type	Data Validation	**Status**	Fixed
File(s)	`ETHStrategy.sol`		
Location(s)	withdraw		
Confirmed Fix At			

The `ETHStrategy` contract has two state variables `depositSlippage` and `withdrawSlippage`. The `depositSlippage` variable is used in the `_deposit` function as follows:

```
uint256 minOutput = (_amount * (magnifier - depositSlippage)) / magnifier;
require(deposited >= minOutput, "DEPOSIT_SLIPPAGE_TOO_BIG");
```

Snippet 4.11: Usage of `depositSlippage` for the computation of `minOutput` in the `_deposit` function from the `ETHStrategy` contract.

It is used to compute the minimum amount of assets that must be deposited, otherwise the code reverts.

The issue is that the `withdrawSlippage` variable is never used in the `withdraw` function or in another part of the code.

Impact The `withdraw` function in the `ETHStrategy` contract lacks slippage protection which might lead to loss of funds.

Recommendation Apply slippage protection on the `withdraw` function of the `ETHStrategy` contract.

4.1.11 V-TLA-VUL-011: Potential Invalid use of tx.origin

Severity	Low	**Commit**	d2d7f10
Type	Phishing	**Status**	Fixed
File(s)	`Whitelist.sol`		
Location(s)	isWhitelisted		
Confirmed Fix At			

The protocol declares a `Whitelist` contract which allows users to query if an address is whitelisted using the `isWhitelisted` function shown below. This function uses `tx.origin` to determine if an address is an EOA by comparing it to the passed in address.

```
function isWhitelisted(address _addr) public view returns (bool) {
    // if addr is EOA return true
    if(tx.origin == _addr){
        return true;
    }
    return whitelist[_addr];
}
```

Snippet 4.12: Definition of the `isWhitelisted` function.

Impact Depending on how this function is used (since `Whitelist` currently isn't used by the protocol), the use of `tx.origin` could allow phishing attacks.

Recommendation Only use `isWhitelisted` with `msg.sender` or remove the `Whitelist` contract since it is unused.

4.1.12 V-TLA-VUL-012: Use Token Decimals Instead of Hardcoding

Severity	Low	**Commit**	d2d7f10
Type	Maintainability	**Status**	Fixed
File(s)	`aavePoolV2.sol`, `aavePoolV3.sol`		
Location(s)	convertEthTo, convertToEth, getCollateralTo, getDebtTo, getCollateralAndDebtTo		
Confirmed Fix At			

The `aavePoolV2` and `aavePoolV3` contracts provide functionality to convert the strategy's collateral and debt values to other currencies. When doing so, the desired token's address is provided along with the desired number of decimals as shown below. If the `_decimals` argument does not match the token's actual decimals value, the conversion will be incorrect.

```
function getCollateralTo(address _user,address _token,uint256 _decimals) public view
    returns (uint256) {
    (uint256 c, , , , , ) = IAave(aave).getUserAccountData(_user);
    uint256 price = IAaveOracle(aaveOracle).getAssetPrice(_token);
    return c*_decimals/price;
}
```

Snippet 4.13: The definition of the `getCollateralTo` function which takes in `_token`'s decimals as `_decimals`

Impact If the specified `_decimals` value is incorrect, then the function could report an incorrect value. As these functions are used to calculate how many funds to pay users, this could therefore result in users being under- or over-paid.

Recommendation Since `_token` is known, use the `token.decimals` function rather than hard-coding the decimals value.

4.1.13 V-TLA-VUL-013: Inconsistent Deposit Logic

Severity	Low	**Commit**	d2d7f10
Type	Logic Error	**Status**	Invalid
File(s)	`Vault.sol`		
Location(s)	deposit		
Confirmed Fix At			

When a user deposits funds, the vault will calculate the associated number of shares that the user is owed. When calculating the first deposit (or rather when the total supply or number of deposits is 0), the share calculation uses `assets` whereas in all other cases it uses `newDeposit`. Notably, when `totalDeposit == 0`, then `newDeposit == assets - fees`.

```
function deposit(address receiver)
    public payable virtual override nonReentrant unPaused
    returns (uint256 shares)
{
    ...
        uint256 newDeposit = IController(controller).deposit(assets);

    require(newDeposit > 0, "INVALID_DEPOSIT_SHARES");

    // Calculate share amount to be mint
    shares = totalSupply() == 0 || totalDeposit == 0
        ? assets.mulDiv(
            10 ** decimals(),
            10 ** asset.decimals(),
            Math.Rounding.Down
        )
        : newDeposit.mulDiv(
            totalSupply(),
            totalDeposit,
            Math.Rounding.Down
        );
    ...
}
```

Snippet 4.14: Location in the `deposit` function which calculates user shares

Impact When `totalSupply == 0` or `totalDeposit == 0`, the depositor will not be charged for fees.

Recommendation Change `assets` in the first branch of the ternary operator to `newDeposit`.

4.1.14 V-TLA-VUL-014: Validate Function Arguments

Severity	Low	**Commit**	d2d7f10
Type	Dead Code	**Status**	Fixed
File(s)	`ExchangePolygon.sol`, `Exchange.sol`		
Location(s)	swapStETH, swapETH		
Confirmed Fix At			

The `ETHLeverExchangePolygon` contract attempts to maintain the same ABI as the `ETHLeverExchange` contract. To match functions in `EthLeverExchange`, the `ETHLeverExchangePolygon` contract takes `token` as an argument but never uses it as it assumes `token` is `stETH`. Similarly, `ETHLeverExchange` uses the `token` argument but assumes that it is either `stETH` or `wstETH`.

```
function swapETH(address token,uint256 amount,uint256 minAmount) external override
    onlyLeverSS {
    require(
        IERC20(stETH).balanceOf(address(this)) >= amount,
        "INSUFFICIENT_STETH"
    );

    // Approve STETH to curve
    IERC20(stETH).approve(curvePool, 0);
    IERC20(stETH).approve(curvePool, amount);
    ICurve(curvePool).exchange(0, 1, amount, minAmount,true);

    uint256 ethBal = address(this).balance;

    // Transfer STETH to LeveraSS
    TransferHelper.safeTransferETH(leverSS, ethBal);
}
```

Snippet 4.15: Definition of the `swapETH` function which doesn't use the `token` argument

Impact The function may not behave as expected if a users specifies a token other than `stETH`. Similarly in `EthLeverExchange`, these functions may not behave as intended if `token` is not `stETH` or `wstETH`.

Recommendation Validate that `token` is `stETH` in `ETHLeverExchangePolygon` and that `token` is `stETH` or `wstETH` in `ETHLeverExchange`.

4.1.15 V-TLA-VUL-015: Validate Withdraw Receiver

Severity	Low	**Commit**	d2d7f10
Type	Data Validation	**Status**	Fixed
File(s)	`Vault.sol`		
Location(s)	withdraw, redeem		
Confirmed Fix At			

When a user withdraws their funds from the AAVE investment strategy, they must specify a recipient that will receive those funds. There is no validation performed on this value which could allow users to accidentally lock their funds by sending them to the null address.

```
function withdraw(uint256 assets, address receiver)
    public virtual nonReentrant unPaused returns (uint256 shares)
{
    require(assets != 0, "ZERO_ASSETS");
    require(assets <= maxWithdraw, "EXCEED_ONE_TIME_MAX_WITHDRAW");
    // Calculate share amount to be burnt
    shares =
        (totalSupply() * assets) /
        IController(controller).totalAssets();

    require(shares > 0, "INVALID_WITHDRAW_SHARES");
    require(balanceOf(msg.sender) >= shares, "EXCEED_TOTAL_DEPOSIT");

    _withdraw(assets, shares, receiver);
}
```

Snippet 4.16: Definition of the `withdraw` function

Impact Funds could be locked if the user specifies the null address.

Recommendation Either strictly disallow `receiver` to be `address(0)` or in this case change the receiver to be `msg.sender`.

4.1.16 V-TLA-VUL-016: Centralization Risk

Severity	Low	**Commit**	d2d7f10
Type	Centralization	**Status**	Acknowledged
File(s)	`Vault.sol`, `Controller.sol`, `ETHStrategy.sol`, `Whitelist.sol`		
Location(s)	N/A		
Confirmed Fix At			

The AAVE investment strategy specifies an owner address which is given special privileges in the `ethVault`, `ethController`, `Whitelist`, and `EthStrategy` contracts. In particular the owner can change contract addresses, value thresholds, charged fees, whitelisted addresses, collect fees, and manipulate AAVE loans. As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance or multi-sig contract as an EOA introduces a single point of failure.

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, a malicious owner could set contract addresses to ones owned by the malicious actor to steal funds.

Recommendation Utilize a decentralized governance or multi-sig contract as the owner.

Developer Response We will use a multi-sig for the time being and build a DAO for community use as the project becomes more mature.

4.1.17 V-TLA-VUL-017: Unused Whitelist Contract

Severity	Warning	**Commit**	d2d7f10
Type	Dead Code	**Status**	Fixed
File(s)	`Whitelist.sol`		
Location(s)	N/A		
Confirmed Fix At			

The protocol declares a `Whitelist` contract which allows an owner to add addresses to a whitelist and query whether an address is in a whitelist. None of the contracts in the protocol make use of the `Whitelist` contract, though.

Impact If it is intended for the protocol to only interact with whitelisted contracts, it currently will not do so because this contract is unused.

Recommendation Either remove the Whitelist contract or make use of it in the protocol.

4.1.18 V-TLA-VUL-018: Unused Contract Variable

Severity	Warning	**Commit**	d2d7f10
Type	Dead Code	**Status**	Fixed
File(s)	`Controller.sol`		
Location(s)	N/A		
Confirmed Fix At			

The `ethController` contract declares the contract variable `asset` as shown below. This variable is not set or used anywhere in the contract since the asset used by the controller is ETH.

```
contract ethController is IController, Ownable, ReentrancyGuard {
        ...

    // Asset for deposit
    ERC20 public asset;

        ...
}
```

Snippet 4.17: The variable declaration of the unused variable in `ethController`

Recommendation Remove the unused variable.

4.1.19 V-TLA-VUL-019: Unused Internal Function

Severity	Warning	**Commit**	d2d7f10
Type	Dead Code	**Status**	Fixed
File(s)	`Controller.sol`		
Location(s)	getBalance		
Confirmed Fix At			

The `ethController` contract declares the internal function `getBalance`, shown below, but never uses it.

```
function getBalance(
    address _asset,
    address _account
) internal view returns (uint256) {
    if (address(_asset) == address(0) || address(_asset) == weth)
        return address(_account).balance;
    else return IERC20(_asset).balanceOf(_account);
}
```

Snippet 4.18: The definition of `getBalance` which is never used

Recommendation Remove the unused function.

4.1.20 V-TLA-VUL-020: Use WETH deposit instead of fallback

Severity	Warning	**Commit**	d2d7f10
Type	Locked Funds	**Status**	Fixed
File(s)	`ETHStrategy.sol`		
Location(s)	loanFallback		
Confirmed Fix At			

Rather than directly using native tokens, AAVE supports ETH by using the wrapped Ethereum (WETH) token. Since the AAVE investment strategy makes use of ETH directly, rather than WETH, it occasionally wraps ETH, by directly transferring the ETH to WETH. This method of wrapping is more error-prone than using the `deposit` function though.

```
function loanFallback(
    uint256 loanAmt,
    uint256 feeAmt,
    bytes calldata userData
) external override onlyReceiver {
    ...

    if (curState == SrategyState.Deposit) {
        ...
    } else if (curState == SrategyState.Withdraw) {
        ...

        // Deposit WETH
        TransferHelper.safeTransferETH(weth, (loanAmt + feeAmt));

                ...
    }

        ...
}
```

Snippet 4.19: Location in `loanFallback` where the weth fallback is used rather than `deposit`

Impact If the contract is misconfigured, funds could accidentally be transferred to an undesired address.

Recommendation Use `weth.deposit` to wrap tokens.

4.1.21 V-TLA-VUL-021: Unused contract variables in ETHStrategy

Severity	Warning	Commit	d2d7f10
Type	Dead Code	Status	Fixed
File(s)	ETHStrategy.sol		
Location(s)	N/A		
Confirmed Fix At			

The `ETHStrategy` contract declares the contract variables `harvestGap` and `latestHarvest` as shown below. The variable `latestHarvest` is not set or used anywhere in the contract; the variable `harvestGap` is not used anywhere in the contract.

```
contract ETHStrategy is Ownable, ISubStrategy, IETHLeverage {
        ...

    // Harvest Gap
    uint256 public override harvestGap;

    // Latest Harvest
    uint256 public override latestHarvest;

        ...
}
```

Snippet 4.20: The variable declaration of the unused variables in `ETHStrategy`.

Recommendation Remove the unused variables.

4.1.22 V-TLA-VUL-022: Unused Inherits

Severity	Warning	**Commit**	d2d7f10
Type	Dead Code	**Status**	Fixed
File(s)	`Controller.sol`, `Exchange.sol`, `ExchangePolygon.sol`		
Location(s)	N/A		
Confirmed Fix At			

The `ethController`, `ETHLeverExchange` and `ETHLeverExchangePolygon` contracts all inherit from OpenZeppelin utility contracts but never use them. Specifically, `ethController` inherits from `ReentrancyGuard` while `ETHLeverExchange` and `ETHLeverExchangePolygon` both inherit from `Ownable` but none of the contracts use the defined functionality.

Recommendation Remove unused inherits.

4.1.23 V-TLA-VUL-023: Incorrect price returned due to lack of decimals representation

Severity	Warning	**Commit**	d2d7f10
Type	Usability Issue	**Status**	Fixed
File(s)	`Vault.sol`		
Location(s)	assetsPerShare()		
Confirmed Fix At			

The `assetPerShare` function of the Vault contract looks like:

```
function assetsPerShare() public view returns (uint256) {
    return IController(controller).totalAssets() / totalSupply();
}
```

Snippet 4.21: `assetsPerShare` function from the `Vault` contract.

It just a division of `totalAssets` and `totalSupply` of shares without decimal precision.

Impact This may cause third party integrations problems due to the loss of precision. For example:

`totalAssets = 150e18`

`totaSupply = 100e18`

The function will return an `assetsPerShare` of 1.

Recommendation Use a decimal precision escalation.