Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Davos Stable Asset

Veridise Inc.
November 30, 2022

► **Prepared For:**

Davos Finance
https://davos.xyz/

► **Prepared By:**

Jon Stephens
Xiangan He
Bryan Tan

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

► **Version History:**

Nov 15, 2022	V1
Nov 30, 2022	V2

Contents

1 Executive Summary

From October 21 to November 16, Davos engaged Veridise to review the security of their Stable Asset. The review covered the on-chain contracts that implement the protocol logic. Veridise conducted the assessment over 9 person-weeks, with 3 engineers reviewing code over 3 weeks from commit `46973a1` to commit `8bf1474` of the `sikka-smart-contracts/contracts` repository. The auditing strategy involved tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Summary of issues detected. The audit uncovered 18 issues, 2 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, bug V-DAV-VUL-001 causes the oracle to return potentially incorrect prices, while bug V-DAV-VUL-002 can lead to users receiving an incorrect amount of funds on performing a withdraw. The Veridise auditors also identified several moderate-severity issues, including redundant checks in Colander (V-DAV-VUL-006) and truncating computations (V-DAV-VUL-008). In addition to these concerns, auditors also identified a number of other concerns, including potentially stuck funds due to a lack of revert on contracts receiving ETH (V-DAV-VUL-012), instances of unsafe typecasting (V-DAV-VUL-014), as well as several code optimizations and maintainability suggestions (V-DAV-VUL-016, V-DAV-VUL-017, V-DAV-VUL-018).

Code assessment. The Davos Stable Asset is based on a fork of the Helio stable asset. From Helio, Davos inherits a modified version of MakerDAO, Sikka DAO and Ceros. Davos extends upon this by adding several contracts to generate further yields for their users. One such contract is Colander, which rewards users for staking Sikka used to losslessly purchase collateral from auctions. In addition Davos adds an ERC4626 vault that will allocate funds to several investment strategies. Currently, the only strategy Davos has written will invest funds in Ceros, allowing it to gain yields. Finally, Davos adds in a Swap pool that swaps MATIC and Ceros tokens. They also integrate the swap pool into several components of the protocol, including Ceros and the CerosYieldConverterStrategy.

Davos provided the source code for the Stable Asset contracts for review. A hardhat-based test-suite accompanied the source-code with tests written by the developers. These tests encompassed only the MasterVault and SwapPool. In addition, the client provided documentation describing the intended behavior for the contracts.

Code Stability. Over the period of the audit, new code was pushed to the repository 13 times, with the most recent commit occurring on Nov. 14. The primary purpose of these commits was to fix issues discovered during the course of the audit however a few new features were added. The Veridise auditors have therefore reviewed some portions of the code more than others.

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.

Project Dashboard 2

Table 2.1: Application Summary.

Name	Version	Type	Platform
Davos Stable Asset	`46973a1 - 8bf1474`	Solidity	Polygon

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Oct. 21 - Nov. 16, 2022	Manual & Tools	3	9 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	2	2
Medium-Severity Issues	7	7
Low-Severity Issues	6	6
Warning-Severity Issues	3	3
Informational-Severity Issues	0	0
TOTAL	18	18

Table 2.4: Category Breakdown.

Name	Number
Logic Error	4
Locked Funds	2
Maintainability	2
Data Validation	5
Usability	5

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the on-chain portion of the Stable Asset. In our audit, we sought to answer the following questions:

- Are users rewarded fairly for staking funds?
- Can users eventually withdraw funds deposited in MasterVault?
- Will Colander perform lossless purchases from auctions?
- Are funds properly distributed among strategies according to their allocation?
- Can a user steal funds from the MasterVault?
- Can new yield conversion strategies be added in the future?
- Is MasterVault robust against possible bugs in strategies?
- Does SwapPool fairly swap one token for another?
- Does CerosVault correctly pass yields onto the user?

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, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. This audit reviewed the on-chain behaviors of the Stable Asset, including user deposits and withdrawals, yield collection and distribution via. DeFi strategy primitives, as well as internal behaviors and liquidations. As such, Veridise auditors first reviewed the provided whitepaper and documentation to understand the desired behavior of the protocol as a whole. Then, the auditors inspected the provided tests to better understand the desired behavior of the provided contracts at a more granular level. Finally, auditors began a multi-week manual audit of the code assisted by both static analyzers and automated testing.

In terms of the audit, the key components include the following:

- The ERC4626 MasterVault and Waiting Pool
- MasterVault Yield Farming Strategies

- Davos Swap Pool
- Davos Colander Lossless Auction Purchasing Contract and Rewards
- Davos's inherited components from Helio

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.

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

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

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

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

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

Vulnerability Report 4

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-DAV-VUL-001	Truncation on result of division in PriceOracle	High	Fixed
V-DAV-VUL-002	Strategy can cause over-drafting	High	Fixed
V-DAV-VUL-003	Potential Loss of Funds from Vault	Medium	Intended Behavior
V-DAV-VUL-004	Incorrect Logic Adjusting Allocated Funds	Medium	Intended Behavior
V-DAV-VUL-005	Incorrect Amount of LP Tokens Minted	Medium	Intended Behavior
V-DAV-VUL-006	Noop on threshold checks for surge	Medium	Fixed
V-DAV-VUL-007	totalDebt may be reduced by incorrect amount	Medium	Fixed
V-DAV-VUL-008	Withdraw Amount Truncation	Medium	Fixed
V-DAV-VUL-009	Colander doesn't obey profit threshold	Medium	Fixed
V-DAV-VUL-010	State Vars Not Set in Initialize	Low	Intended Behavior
V-DAV-VUL-011	Lack of token burn access control	Low	Intended Behavior
V-DAV-VUL-012	Add Revert to Receive to Prevent Stuck Funds	Low	Fixed
V-DAV-VUL-013	Potential Reentrancy in SwapPool	Low	Fixed
V-DAV-VUL-014	Unsafe Typecasting in SwapPool	Low	Fixed
V-DAV-VUL-015	Ceros Strategy may withdraw fewer than requeste	Low	Fixed
V-DAV-VUL-016	No checks for 0x0	Warning	Fixed
V-DAV-VUL-017	Contracts should inherit from their interfaces	Warning	Fixed
V-DAV-VUL-018	Use SafeERC20 Functions	Warning	Fixed

4.1 Detailed Description of Bugs

4.1.1 V-DAV-VUL-001: Truncation on result of division in PriceOracle

Severity	High	**Commit**	466a036
Type	Truncation	**Status**	Fixed
Files	oracle/PriceOracle.sol		
Functions	peek()		

As a result of the following operation on the result of division, the price returned by `peek` of the PriceOracle (used in multiple parts of the protocol such as IkkaRewards, colander, etc.) can be incorrect. Note this issue was fixed in Helio.

```
function peek() public view returns (bytes32, bool) {
    ...
    uint256 price = oneTokenOut / amountOut * 10**18 ;
    return (bytes32(price), true);
  }
```

Snippet 4.1: Multiplication operation after division can cause truncation.

Impact Inaccurate oracle prices returned from calling peek.

Recommendation Fix the error by doing multiplication first.

4.1.2 V-DAV-VUL-002: Malicious or buggy strategy can cause over-drafting

Severity	High	**Commit**	7430197
Type	Overdrafting	**Status**	Fixed
Files	MasterVault/MasterVault.sol		
Functions	withdrawETH()		

When performing a withdraw, it might be the case that the MasterVault does not have sufficient funds to pay back the user. If this occurs, it will attempt to retrieve any additional funds necessary from a strategy. In doing so, it updates the amount of funds that should be sent to the user in the statement `shares = withdrawFromActiveStrategies(amount - wethBalance)` . Here the developers make the assumption that `shares == amount - wethBalance` or `shares == 0`.

```
shares = amount;
_burn(src, shares);
uint256 wethBalance = totalAssetInVault();
if(wethBalance < amount) {
    shares = withdrawFromActiveStrategies(amount - wethBalance);
    if(shares == 0) {
        // submit to waiting pool
        waitingPool.addToQueue(account, amount);
        if(wethBalance > 0) {
            IWETH(asset()).withdraw(wethBalance);
            shares = _assessSwapFee(amount);
            payable(address(waitingPool)).transfer(wethBalance);
        }
        emit Withdraw(src, src, src, amount, amount);
        return amount;
    }
    shares += _assessSwapFee(wethBalance);
} else {
```

Snippet 4.2: The affected logic in `withdrawETH()`

Impact If a strategy deviates from the assumption identified above so that `shares < amount - wethBalance` , the user will receive fewer funds than they are owned. Similarly, if `shares > amount - wethBalance` the user will receive more funds than they are owed.

Recommendation The developers clarified that they do this because `withdrawFromStrategies` may charge fees; however, we believe they should perform some validation to ensure the returned value is within an acceptable range.

4.1.3 V-DAV-VUL-003: Potential Loss of Funds from Vault

Severity	Medium	**Commit**	466a036
Type	Locked Funds	**Status**	Intended Behavior
Files	MasterVault.sol		
Functions	withdrawETH(address account, uint256 amount)		

In a situation where:

- wethBalance of the vault < withdrawal amount (with valid amount)
- shares == 0 from strategy withdrawal
- wethBalance > 0

A transfer call is made to the waitingPool address with value of the weth balance of the MasterVault. By default, all solidity addresses are initialized as 0x0.

```
function withdrawETH(address account, uint256 amount)
external
override
nonReentrant
whenNotPaused
onlyProvider
returns (uint256 shares) {
    ...
    if(wethBalance < amount) {
        shares = withdrawFromActiveStrategies(amount - wethBalance);
        if(shares == 0) {
            ...
            if(wethBalance > 0) {
                IWETH(asset()).withdraw(wethBalance);
                payable(address(waitingPool)).transfer(wethBalance);
            }
            emit Withdraw(src, src, src, amount, shares);
            return amount;
        }
        ...
}
```

Snippet 4.3: The affected logic in MasterVault.sol

Impact An unwrapping and sending of funds to payable(address(waitingPool)).transfer(wethBalance); results in loss of all funds from the treasury if waitingPool was never set.

Developer Response The deployment script will ensure the waiting pool is properly initialized.

4.1.4 V-DAV-VUL-004: Incorrect Logic Adjusting Allocated Funds

Severity	Medium	**Commit**	466a036
Type	Logic Error	**Status**	Intended Behavior
Files	MasterVault.sol		
Functions	allocate()		

Currently, the `allocate` function is responsible for providing funds to strategies based on their allocations percentage. Allocate should rebalance the strategy allocation portfolio by withdrawing and depositing from and to strategy pools such that the final allocated amount matches the percentage of funds set to be allocated to strategies. Currently, allocate only evaluates whether or not a strategy should be deposited to. However, it does not do the following:

- If allocation ratio is set to 0 for any strategy, withdraw all existing funds
- If allocation ratio is higher than funds allocated to total assets ratio, withdraw appropriate amount of funds

```
    if(strategyRatio < allocation) {
        uint256 depositAmount = ((totalAssets * allocation) / 1e6) - strategy.debt;
        if(totalAssetInVault() > depositAmount) {
            _depositToStrategy(strategies[i], depositAmount);
            // IBaseStrategy(strategies[i]).depositAll();
        }
        // } else {
        //     uint256 withdrawAmount = strategy.debt - (totalAssets * allocation) /
    1e6;
        //     if(withdrawAmount > 0) {
        //         _withdrawFromStrategy(strategies[i], withdrawAmount);
        //     }
        // }

        ...
```

Snippet 4.4: The deposit logic is there, but the withdrawal logic seems commented out.

Seeing that there are no other functions that call to `withdrawFromStrategy` besides from `retireStrat` or `migrateStrategy`, there currently is no way to be able to rebalance the portfolio on a strategy-by-strategy basis.

Impact Since old funds are stuck inside the strategies and not rebalanced, it is possible for the portfolio to have incorrect total debt asset accounting via. the ratio of strategies to total assets. This affects multiple functions inside MasterVault.

Recommendation Since this has been recognized as intended behavior, this dead code should be removed

Developer Response The withdraw behavior was removed to reduce the number of incurred fees. Instead, admins will be responsible for determining when to withdraw from a strategy.

4.1.5 V-DAV-VUL-005: Incorrect Amount of LP Tokens Minted

Severity	Medium	**Commit**	466a036
Type	Logic Error	**Status**	Intended Behavior
Files	SwapPool.sol		
Functions	`_addLiquidity`		

SwapPool's `_addLiquidity` function mints an amount of LP tokens to the user equivalent to the total of their native token and ceros token deposit. However, since the ratio of both the native and Ceros tokens are denominated in 1e18, and the mint here only mints 10**8, there may have been a mistake in the minting logic.

```
 if (nativeTokenAmount == 0 && cerosTokenAmount == 0) {
     require(amount0 > 1e18, "cannot add first time less than 1 token");
     nativeTokenAmount = amount0;
     cerosTokenAmount = amount1;

     lpToken.mint(msg.sender, (2 * amount0) / 10**8);
   }
```

Snippet 4.5: Should mint 2 * amount / 10**18

Impact LP Tokens would be minted incorrectly on calls to add liquidity to the swap pools, impacting users' ability to withdraw the correct amount since they would be burning less tokens than they're supposed to have.

Recommendation Fix the above by raising to 1e18.

Developer Response Since this value likely only affects the first individual to interact with the contract and since it has already been posted to the blockchain, the value cannot be changed.

4.1.6 V-DAV-VUL-006: Noop on threshold checks for surge

Severity	Medium	**Commit**	466a036
Type	Validation Error	**Status**	Fixed
Files	colander.sol		
Functions	surge(address `_collateral`, uint256 `_auction_id`)		

The surge function is responsible for checking on a maximum of auction prices: they should always be strictly lower than prices from the price feed and abacus, and at least lower than a threshold price.

Firstly, there is already a check assuring that abacusPrice is always strictly lower than feedPrice (else revert)

```
if (abacusPrice >= feedPrice) revert IStabilityPool.BufZone();
```

So the fact that auctionPrice is checked to be less than abacusPrice after being checked to be lower than feedPrice is redundant.

```
if (auctionPrice >= feedPrice) revert IStabilityPool.AbsurdPrice();
else if (auctionPrice >= abacusPrice) revert IStabilityPool.SinZone();
```

Snippet 4.6: The feedPrice check can be removed because `abacusPrice < feedPrice`.

Next, auctionPrice is checked to be less than or equal to threshold, and so is abacusPrice. But `threshold < feePrice`.

```
uint256 threshold = feedPrice - y;

if (auctionPrice > threshold) revert IStabilityPool.AbsurdThreshold();
 else if (abacusPrice > threshold) revert IStabilityPool.InactiveZone();
```

Snippet 4.7: We now have `auctionPrice < abacusPrice < threshold < feedPrice`

Therefore, given the above checks, the checks on auction price greater than threshold would not be necessary at the moment, since it's already being checked to be less than abacus price.

Additionally, Colander tests are currently skipped. The success case for `surge` actually fails, in addition to several other tests for surge reverting with only `AbsurdPrice()` when they were described to revert with other messages.

Impact Redundant checks create no-ops, and cases where certain lines of checks will never revert. Removing these increases code clarity and improves code maintainability while saving contract size and gas.

Note that in the `colander.js` tests, the function call expected to revert with `InactiveZone()`, `AbsurdThreshold()`, and `SinZone()` already reverts only with `AbsurdPrice`, which may contain logic errors that pertains to the oracle within itself.

Additionally, the failing test on the success case of surge is alarming for the actual protocol functionality. These tests should be fixed for guarantees that the protocol functions as intended, especially with its oracle interactions in light of issues having been discovered.

Recommendation

- Keep only `if (auctionPrice >= abacusPrice) revert` and `if (abacusPrice >= threshold ) revert`
- Or change `uint256 threshold = feedPrice - y;` to `uint256 threshold = feedPrice + y;` since it's supposed to indicate (gathering from comments in this case) the amount of profit they should receive; then the variables would be different and checks would be meaningful again (`auctionPrice < abacusPrice < feedPrice < threshold`); still; remove the auctionPrice checks against feedPrice and threshold as auctionPrice will always be less than these two since it's already checked to be less than `abacusPrice`

4.1.7 V-DAV-VUL-007: totalDebt may be reduced by incorrect amount

Severity	Medium	**Commit**	ee8ff78
Type	Logic Error	**Status**	Fixed
Files	MasterVault/MasterVault.sol		
Functions	`_withdrawFromStrategy()`		

The `_withdrawFromStrategy()` makes a call to `IBaseStrategy(strategy).withdraw(amount)` , which then returns the actual amount `value` that is withdrawn. However, both `totalDebt` and `strategyParams[strategy].debt` are decreased by `amount` instead of `value`, so they may be decreased by a greater amount than should be expected.

```
function _withdrawFromStrategy(address strategy, uint256 amount) private returns(
    uint256) {
    require(amount > 0, "invalid withdrawal amount");
    require(strategyParams[strategy].debt >= amount, "insufficient assets in strategy
    ");
    uint256 value = IBaseStrategy(strategy).withdraw(amount);
    if(value > 0)
        totalDebt -= amount;
        strategyParams[strategy].debt -= amount;
        emit WithdrawnFromStrategy(strategy, amount);
    }
    return value;
}
```

Snippet 4.8: Implementation of `_withdrawFromStrategy()`

Impact This can cause `totalDebt` to be out of sync with the actual strategy, which could cause funds to be locked in the strategy or unexpected reverts to occur.

Recommendation Subtract by `value` instead of `amount`. If `value` may be less than `amount` validate that the return value is within an acceptable threshold of the given amount.

4.1.8 V-DAV-VUL-008: Withdraw Amount Truncation

Severity	Medium	**Commit**	466a036
Type	Truncation	**Status**	Fixed
Files	MasterVault.sol		
Functions	payDebt		

As a result of the following operation on the result of division, the `withdrawAmount` calculated by payDebt can be incorrect.

```
    function payDebt() public {
    ...
    if (waitingPoolDebt > waitingPoolBal) {
        uint256 maxFee = swapPool.FEE_MAX();
        uint256 withdrawAmount =
            (
                ((waitingPoolDebt - waitingPoolBal) * 1e18) /
                (maxFee - swapPool.unstakeFee()) * maxFee
            ) / 1e18;
        uint256 withdrawn = withdrawFromActiveStrategies(withdrawAmount +
    1);
        if(withdrawn > 0) {
            IWETH(asset()).withdraw(withdrawn);
            payable(address(waitingPool)).transfer(withdrawn);
        }
    }
}
```

Recommendation Fix the error by doing multiplication first.

4.1.9 V-DAV-VUL-009: Colander doesn't obey profit threshold

Severity	Medium	**Commit**	ee8ff78
Type	Access Control	**Status**	Fixed
Files	Colander.sol		
Functions	surge()		

The colander contract allows admins to set a target amount the contract should profit from a particular sale. However, at the moment that profit range is not being adhered to since `threshold < feedPrice`, rendering the checks redundant.

```
function surge(address _collateral, uint256 _auction_id) external isLive
    auth nonReentrant {
    ...

      if (auctionPrice >= feedPrice) revert IStabilityPool.AbsurdPrice();
      else if (auctionPrice >= abacusPrice) revert IStabilityPool.SinZone()
    ;
      // require(auctionPrice < feedPrice, "Colander/absurd-price");
      // require(auctionPrice < abacusPrice, "Colander/sin-zone");

      uint256 y = (feedPrice * profitRange) / RAY;
      uint256 threshold = feedPrice - y;

      if (auctionPrice > threshold) revert IStabilityPool.AbsurdThreshold()
    ;
      else if (abacusPrice > threshold) revert IStabilityPool.InactiveZone
    ();

    ...
}
```

Impact This could cause the colander contract to actually lose money in cases where the `priceImpact` allows for the sale of an asset for less than the `abacusPrice` and `feedPrice`. Note, that this still will never allow user funds to be lost since that would cause `surplus = stablecoin .balanceOf(address(this)) - totalSupply;` to revert.

4.1.10 V-DAV-VUL-010: State Vars Not Set in Initialize

Severity	Low	**Commit**	466a036
Type	Data Validation	**Status**	Intended Behavior
Files	Multiple		
Functions	initialize()		

Many state variables are not assigned to a default value, meaning that they can be 0 or 0x0 where potentially undesirable.

Some examples of where this happens is listed below.

```
    function initialize(
        string memory name,
        string memory symbol,
        uint256 maxDepositFees,
        uint256 maxWithdrawalFees,
        IERC20MetadataUpgradeable asset,
        uint8 maxStrategies,
        address swapPoolAddr
    ) public initializer {
        require(maxDepositFees > 0 && maxDepositFees <= 1e6, "invalid maxDepositFee");
        require(maxWithdrawalFees > 0 && maxWithdrawalFees <= 1e6, "invalid maxWithdrawalFees");

        __Ownable_init();
        __Pausable_init();
        __ReentrancyGuard_init();
        __ERC20_init(name, symbol);
        __ERC4626_init(asset);
        manager[msg.sender] = true;
        maxDepositFee = maxDepositFees;
        maxWithdrawalFee = maxWithdrawalFees;
        MAX_STRATEGIES = maxStrategies;
        feeReceiver = payable(msg.sender);
        swapPool = ISwapPool(swapPoolAddr);
    }
```

Snippet 4.9: MasterVault doesn't check `maxStrategies` initialization input

```
    function initialize(string memory _name, string memory _symbol, address
    _stablecoin, address _interaction, address _spotter, address _dex, address
    _rewards, uint256 _flashDelay) public initializer {
        wards[msg.sender] = 1;
        live = 1;
        name = _name;
        symbol = _symbol;
        stablecoin = IERC20Upgradeable(_stablecoin);
        interaction = IDao(_interaction);
        spotter = SpotLike(_spotter);
        dex = DexV3Like(_dex);
        rewards = IColanderRewards(_rewards);
        flashDelay = _flashDelay;

        decimals = 18;

        __ReentrancyGuard_init_unchained();

        emit Initialize(msg.sender);
    }
```

Snippet 4.10: No validations on nonzero state values outside of initialize (such as `spread`)

Impact In some past examples of attacks (e.g. Nomad), having variables be set to 0x0 automatically proves to be a threat. The code becomes potentially difficult to maintain, as taking into account every state variable that gets initialized intentionally to 0x0 when writing safe code becomes hard with multiple contracts.

Recommendation Initialize all variables to an appropriate default value.

4.1.11 V-DAV-VUL-011: Lack of token burn access control

Severity	Low	**Commit**	466a036
Type	Access Control	**Status**	Intended Behavior
Files	Sikka.sol		
Functions	burn(address usr, uint wad)		

Currently anyone is allowed to burn their Sikka tokens.

```
function burn(address usr, uint wad) external {
    require(usr != address(0), "Sikka/burn-from-zero-address");
    require(balanceOf[usr] >= wad, "Sikka/insufficient-balance");
    if (usr != msg.sender && allowance[usr][msg.sender] != type(uint256).max) {
        require(allowance[usr][msg.sender] >= wad, "Sikka/insufficient-allowance");
        allowance[usr][msg.sender] -= wad;
    }
    balanceOf[usr] -= wad;
    totalSupply    -= wad;
    emit Transfer(usr, address(0), wad);
}
```

Snippet 4.11: Logic allowing anyone to burn tokens

Impact Since this reduces the supply of tokens, it could impact the price of the Sikka stable asset.

Developer Response Since burning tokens will cost the user funds, we think it is unlikely that users will be able to burn sufficient funds to impact the price of Sikka.

4.1.12 V-DAV-VUL-012: Add Revert to Receive to Prevent Stuck Funds

Severity	Low	**Commit**	466a036
Type	Locked Funds	**Status**	Fixed
Files	MasterVault.sol		
Functions	receive()		

Currently, receive does not have a revert, allowing the contract to arbitrarily receive funds sent from transactions without explicit calls to the deposit function. Since the whole mechanic of users being able to get their funds back via. withdrawal depends on a minting of share tokens on calls to `deposit`, not having a `revert()` in receive can potentially cause user funds that are sent to the contract to be stuck.

```
receive() external payable {}
```

Snippet 4.12: Empty Receive function allowing anyone to send funds to MasterVault

Impact Not having a `revert()` in receive can potentially cause user funds that are sent to the contract to be stuck.

Recommendation Revert unless funds are received from an expected entity

4.1.13 V-DAV-VUL-013: Potential Reentrancy in SwapPool

Severity	Low	**Commit**	466a036
Type	Reentrancy	**Status**	Fixed
Files	SwapPool.sol		
Functions	`_addLiquidity`		

The `_sendValue` function uses a low-level call to transfer funds to a user. Since such calls do not restrict the amount of gas the call consumes, however, it leaves the protocol vulnerable to a reentrancy attack. While it appears that most functions in SwapPool use a Reentrancy Guard, ReentrancyGuards can potentially be removed if instead a `.transfer` was used instead of `.call` (or `.send` to the same effect).

```
function _sendValue(address receiver, uint256 amount) virtual internal {
  // solhint-disable-next-line avoid-low-level-calls
  (bool success, ) = payable(receiver).call{ value: amount }("");
  require(success, "unable to send value, recipient may have reverted");
}
```

Impact While the reentrancy doesn't seem harmful in this case, any future development that uses functions that can eventually call `sendValue` are vulnerable to reentrancy attacks.

Recommendation Since none of the protocol's contracts perform any computation in their onReceive function, a transfer should be used rather than a call.

4.1.14 V-DAV-VUL-014: Unsafe Typecasting in SwapPool

Severity	Low	**Commit**	466a036
Type	Validation Error	**Status**	Fixed
Files	SwapPool.sol		
Functions	`_withdrawOwnerFee(),_swap()`		

SwapPool performs multiple typecasts from uint256 inputs to uint128, which is unsafe in various scenarios. Consider the following code when a user inputs `type(uint128).max + 1`

```
function _withdrawOwnerFee(
    uint256 amount0Raw,
    uint256 amount1Raw,
    bool useEth
  ) internal virtual {
    uint128 amount0;
    uint128 amount1;
    if (amount0Raw == type(uint256).max) {
      amount0 = ownerFeeCollected.nativeFee;
    } else {
      amount0 = uint128(amount0Raw);
    }
```

Snippet 4.13: The following function contains no reverts (and also doesn't emit an event) for owner withdrawing their stored fee for said input

In Solidity, when greater uints are typed into smaller uints, bits are truncated; in the case of the example above, `type(uint128).max + 1` would simply be 1 after truncation. This is obviously not the correct withdrawal logic and should be corrected.

For the sake of safety, `_swap` also has similar typecasts from uint256 to uint128. Though they are relatively unlikely to happen (stake fees must be close to 1, amountIn is greater than uint128), a similar thing will happen where an incorrect amount is casted into the variable.

```
ownerFeeCollected.nativeFee += uint128(ownerFeeAmt);
mmanagerFeeCollected.nativeFee += uint128(managerFeeAmt);
```

Snippet 4.14: Similarly, bits are truncated here.

Impact Incorrect amounts are calculated for withdrawal, which causes incorrect behavior for the user (owner or otherwise).

Recommendation One can change the typing on the input itself to match with the rest of the variables tracked in uint128, or simply increase the size of the state variables using uint128 (to uint256) to be uniform and increase code clarity.

4.1.15 V-DAV-VUL-015: Ceros Strategy may withdraw fewer than requested funds

Severity	Low	**Commit**	466a036
Type	Logic Error	**Status**	Fixed
Files	CerosYieldConverterStrategy.sol		
Functions	`_withdraw()`		

When funds are withdrawn from CerosStrategy, they are first withdrawn from the CerosRouter and then swapped to the desired currency using the SwapPool. However, the SwapPool charges fees for its use and therefore the amount returned may be less than the amount desired. Due to this, withdrawing from the CerosStrategy may withdraw fewer than the requested funds. In several places in MasterVault it is assumed that withdraw will either return 0 or the requested amount.

```
function _withdraw(uint256 amount) internal returns (uint256 value) {
    ...

    (uint256 amountOut, bool enoughLiquidity) = ISwapPool(_swapPool).getAmountOut(
    false, ((amount - wethBalance) * _certToken.ratio()) / 1e18, false); // (amount *
     ratio) / 1e18
    if (enoughLiquidity) {
        value = _ceRouter.withdrawWithSlippage(address(this), amount - wethBalance,
    amountOut);
        require(value >= amountOut, "invalid out amount");
        uint256 withdrawAmount = wethBalance + value;
        if (amount < withdrawAmount) {
            // transfer extra funds to feeRecipient
            underlying.transfer(feeRecipient, withdrawAmount - amount);
        } else {
            amount = withdrawAmount;
        }
        underlying.transfer(address(vault), amount);
        return amount;
    }
}
```

Snippet 4.15: The withdraw function in `CerosYieldConverterStrategy` which can reduce the requested amount

Impact By assuming the amount withdrawn is equal to the requested amount, it is possible to underpay users and inaccurately track funds.

4.1.16 V-DAV-VUL-016: No checks for 0x0

Severity	Warning	**Commit**	466a036
Type	Validation Error	**Status**	Fixed
Files	Multiple		
Functions	Multiple		

Thoughout the contracts repository, there are places where there are 0x0 checks while other places don't. Given that there have been past hacks (e.g. Nomad) as a result of a lack of data validation checks, we recommend for the following (and any other instances in the repo) to include more validation. The following highlighted code snippets are some common patterns of not having 0x0 checks.

```
    function changeVault(address vault) external onlyOwner {
        // update allowances
        _certToken.approve(address(_vault), 0);
        _vault = IVault(vault);
        _certToken.approve(address(_vault), type(uint256).max);
        emit ChangeVault(vault);
    }
    function changeDex(address dex) external onlyOwner {
        IERC20(_wMaticAddress).approve(address(_dex), 0);
        _certToken.approve(address(_dex), 0);
        _dex = ISwapRouter(dex);
        // update allowances
        IERC20(_wMaticAddress).approve(address(_dex), type(uint256).max);
        _certToken.approve(address(_dex), type(uint256).max);
        emit ChangeDex(dex);
    }
    function changeSwapPool(address swapPool) external onlyOwner {
        IERC20(_wMaticAddress).approve(address(_pool), 0);
        _certToken.approve(address(_pool), 0);
        _pool = ISwapPool(swapPool);
        IERC20(_wMaticAddress).approve(swapPool, type(uint256).max);
        _certToken.approve(swapPool, type(uint256).max);
        emit ChangeSwapPool(swapPool);
    }
    function changeProvider(address masterVault) external onlyOwner {
        _masterVault = IMasterVault(masterVault);
        emit ChangeProvider(masterVault);
    }
    function changePairFee(uint24 fee) external onlyOwner {
        _pairFee = fee;
        emit ChangePairFee(fee);
    }
    function changePriceGetter(address priceGetter) external onlyOwner {
        _priceGetter = IPriceGetter(priceGetter);
    }
```

Snippet 4.16: Most of the setters (especially the ones in CerosRouter) have no 0x0 checks.

```
function _addLiquidity(
  uint256 amount0,
  uint256 amount1,
  bool useEth
) internal virtual {
  uint256 ratio = cerosToken.ratio();
  uint256 value = (amount0 * ratio) / 1e18;
  if (amount1 < value) {
    amount0 = (amount1 * 1e18) / ratio;
  } else {
    amount1 = value;
  }
  if (useEth) {
    nativeToken.deposit{ value: amount0 }();
    uint256 diff = msg.value - amount0;
    if (diff != 0) {
      _sendValue(msg.sender, diff);
    }
  } else {
    nativeToken.transferFrom(msg.sender, address(this), amount0);
  }
  cerosToken.transferFrom(msg.sender, address(this), amount1);
```

Snippet 4.17: There are also cases where unchecked transfers are made

```
  function _depositToStrategy(address strategy, uint256 amount) private returns (
  bool success){
      require(amount > 0, "invalid deposit amount");
      IWETH weth = IWETH(asset());
      require(totalAssetInVault() >= amount, "insufficient balance");
      if (IBaseStrategy(strategy).canDeposit(amount)) {
          weth.transfer(strategy, amount);
          uint256 value = IBaseStrategy(strategy).deposit(amount);
          if(value > 0) {
              totalDebt += value;
              strategyParams[strategy].debt += value;
              emit DepositedToStrategy(strategy, amount);
              return true;
          }
      }
  }
```

Snippet 4.18: There are no checks for the existence (or correct input) of the strategy address in MasterVault either, including for 0x0 inputs

Impact Incorrect inputs significantly damage the protocol's ability to function properly.

Recommendation Add appropriate `address(0)` checks.

4.1.17 V-DAV-VUL-017: Contracts should inherit from their interfaces

Severity	Warning	**Commit**	466a036
Type	Maintainability	**Status**	Fixed
Files	Multiple		
Functions	N/A		

There are several contracts with interfaces that the contract does not inherit from. These interfaces include:

- ILP
- ISwapPool
- IWaitingPool
- PipLike (clip.sol)
- SpotterLike (clip.sol)
- DogLike (clip.sol)
- ClipperCallee (clip.sol)
- AbacusLike (clip.sol)
- ClipperLike (dog.sol)
- VatLike (dog.sol)
- DSTokenLike (join.sol)
- VatLike (join.sol)

Impact If there are differents between the interface and contract function definitions, functions in the contract may not be callable via the intervace.

Recommendation Always make contracts inherit from interfaces used to call them so that the compiler may catch any issues with the interface.

4.1.18 V-DAV-VUL-018: Use SafeERC20 Functions

Severity	Warning	**Commit**	0056e2a
Type	Maintainability	**Status**	Fixed
Files	Multiple		
Functions	N/A		

When using ERC20 functions the developers should consider using the SafeERC20 library. In the current codebase it is common to use ERC20 functions without checking the return value. Since some tokens return false on failure rather than reverting, funds can be lost if the tokens are not properly vetted.

```
    function _removeLiquidity(uint256 removedLp, bool useEth) virtual internal {
        ...

        if (useEth) {
            nativeToken.withdraw(amount0Removed);
            _sendValue(msg.sender, amount0Removed);
        } else {
            nativeToken.transfer(msg.sender, amount0Removed);
        }
        cerosToken.transfer(msg.sender, amount1Removed);

        ...
    }
```

Impact In the event that a token returned false on failure rather than reverting, a malicious user could steal funds from the protocol.

Recommendation Use the SafeERC20 library when interacting with an ERC20 token.