Cyberscope

Audit Report

Contrax

February 2024

Repository https://github.com/Contrax-co/contrax-smart-contracts/tree/main

Commit 84a60005ec3e9fb50c8f030fa96f8dac38a158c7

Audited by © cyberscope

Table of Contents

Table of Contents **1**
Review **4**
Audit Updates 4
Source Files 4
Overview **6**
Audit Scope 6
Vault Contract 6
Controller Contract 7
Strategy Contracts 7
Zapper Contracts 7
Findings Breakdown **9**
Diagnostics **10**
VAM - Vault Address Manipulation 12
Description 12
Recommendation 15
CR - Code Repetition 16
Description 16
Recommendation 19
CCR - Contract Centralization Risk 20
Description 20
Recommendation 22
DPI - Decimals Precision Inconsistency 23
Description 23
Recommendation 24
EIS - Excessively Integer Size 26
Description 26
Recommendation 26
IDI - Immutable Declaration Improvement 27
Description 27
Recommendation 27
MEM - Misleading Error Messages 28
Description 28
Recommendation 28
MCC - Missing Constructor Checks 29
Description 29
Recommendation 30
MEE - Missing Events Emission 32
Description 32
Recommendation 32

MSV - Missing Strategy Validation 34
Description 34
Recommendation 34
MU - Modifiers Usage 35
Description 35
Recommendation 35
NTTR - Native Token Transfer Restriction 36
Description 36
Recommendation 37
PVI - Path Validation Inadequacy 38
Description 38
Recommendation 38
PBV - Percentage Boundaries Validation 40
Description 40
Recommendation 41
PLPI - Potential Liquidity Provision Inadequacy 42
Description 42
Recommendation 43
PTAI - Potential Transfer Amount Inconsistency 44
Description 44
Recommendation 45
RRS - Redundant Require Statement 46
Description 46
Recommendation 46
RSML - Redundant SafeMath Library 47
Description 47
Recommendation 47
RSW - Redundant Storage Writes 49
Description 49
Recommendation 51
OCTD - Transfers Contract's Tokens 52
Description 52
Recommendation 52
WTD - Withdraw Token Discrepancy 54
Description 54
Recommendation 55
L02 - State Variables could be Declared Constant 57
Description 57
Recommendation 57
L04 - Conformance to Solidity Naming Conventions 58
Description 58
Recommendation 59

L06 - Missing Events Access Control 60
Description 60
Recommendation 60
L07 - Missing Events Arithmetic 61
Description 61
Recommendation 61
L09 - Dead Code Elimination 62
Description 62
Recommendation 62
L11 - Unnecessary Boolean equality 64
Description 64
Recommendation 64
L13 - Divide before Multiply Operation 65
Description 65
Recommendation 65
L14 - Uninitialized Variables in Local Scope 66
Description 66
Recommendation 66
L16 - Validate Variable Setters 67
Description 67
Recommendation 67
L17 - Usage of Solidity Assembly 68
Description 68
Recommendation 68
Functions Analysis **69**
Summary **75**
Disclaimer **76**
About Cyberscope **77**

Review

Repository	https://github.com/Contrax-co/contrax-smart-contracts/tree/main
Commit	84a60005ec3e9fb50c8f030fa96f8dac38a158c7

Audit Updates

Initial Audit	12 Feb 2024 https://github.com/cyberscope-io/audits/blob/main/contrax/v1/audit.pdf
Corrected Phase 2	19 Feb 2024

Source Files

Filename	SHA256
vault.sol	feedaed505cb6e986e13300ec06c3dbd9df5e3e170e1c4776aae68d7334c5ae8
strategy-uni-base.sol	d10c0f5f8940f69318c8fe80363d271e6c915a7d94df8d40a5be4e9143017f1b
strategy-base-v3.sol	43f80ae7cc1f7c533ab8d80cc19d4226abe0b1f36693f818b1b4ef8ecb6c54ca
new-zapper.sol	a034d07b29f894944335b2d4caacadca4a0b4eebce92afbcc03b480df51f85f8
controller.sol	cf12f7b363ae9b26430d9cff1d9f854b4126a04643b90eebfccc7f1b0edb1604
zapper/zapper-base.sol	17d66bff11660fbfcc853b9a67dfef1aeaccdf666fdbb04d0e008f979343d2ef

zapper/vault-zapper.sol	ba4a27ffb3c37d4dba458697220da918da 209f0410d68806dd9f5f6862efb633

Overview

Audit Scope

The contract audit scope includes the following contracts:

- Vault, vault.sol
- Controller, controller.sol
- StrategyUniBase, strategy-uni-base.sol
- StrategyBaseV3, strategy-base-v3.sol
- VaultZapEthSushi,ZapperBase, new-zapper.sol

These contracts heavily rely on each iteration's interaction with the `converter` and `OneSplit` contract addresses. However, the `converter` and `OneSplit` contracts are out of the scope of the current audit. Any integration with these contracts should prompt the team to additionally consider auditing these external contracts to ensure comprehensive security and functionality verification across the system.

Vault Contract

The `Vault` contract serves as a foundational component in the decentralized finance (DeFi) ecosystem, offering users a secure and flexible platform for depositing and withdrawing ERC20 tokens. This contract enables users to transfer their tokens directly into the vault for safekeeping and allows for their retrieval at any time, ensuring a straightforward and efficient management of digital assets. The contract emphasizes user control and transparency, allowing for the dynamic adjustment of operational parameters such as minimum balance requirements to enhance gas efficiency and fund utilization. Furthermore, the Vault provides mechanisms for both comprehensive and incremental asset management, including functions for depositing all tokens held by a user or withdrawing them based on their proportional share of the total supply. This ensures that users can manage their investments efficiently, with the added benefit of governance features that safeguard the contract's integrity and adaptability.

Controller Contract

The `Controller` contract is essential for managing asset allocations to investment strategies in the DeFi sector, acting as an intermediary to optimize user returns. It enables the strategic deployment of funds through the earn function, directing assets to appropriate strategies for yield enhancement. This process ensures that assets are not idle but are actively generating returns. The contract is designed with a focus on security and efficiency, allowing only approved strategies to participate. It also includes mechanisms for strategy management, fee implementation, and emergency safeguards to protect assets. Through its operations, the Controller contract aims to achieve optimal asset growth and strategy diversification for its users.

Strategy Contracts

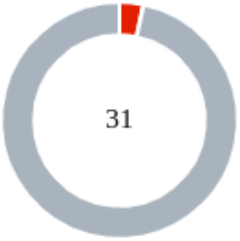
The Strategy Base V3 and Strategy UniBase contracts are integral components of the DeFi ecosystem, designed to enhance asset management and yield optimization for liquidity providers. These contracts are abstract foundations that future strategy implementations will inherit, focusing on automating the process of asset allocation across various DeFi protocols. They manage the deployment of ERC20 tokens into liquidity pools, leveraging platforms like Uniswap for trading and staking to earn rewards. These strategies are equipped with mechanisms for fee distribution, governance, and emergency safeguards, ensuring secure and efficient operation. By interfacing with controllers, these strategies enable dynamic asset reallocation, responding to market conditions to maximize returns. Additionally, they incorporate performance fees to reward strategy developers and governance participants, aligning incentives across the ecosystem. Overall, these contracts facilitate a decentralized approach to asset management, emphasizing transparency, security, and user autonomy in the DeFi space.

Zapper Contracts

The VaultZapEthSushi contract, streamlines the process of entering and exiting liquidity positions within the DeFi ecosystem, particularly focusing on Ethereum-based protocols. By automating the conversion and allocation of assets, these contracts facilitate efficient and optimized interactions with various DeFi strategies, including liquidity provision and yield farming. ZapperBase lays the foundational infrastructure for asset swapping and staking, enabling users to seamlessly zap in using ETH or ERC20 tokens into desired positions.

VaultZapEthSushi extends this functionality, specifically catering to the SushiSwap platform, to offer a targeted solution for users aiming to participate in SushiSwap's liquidity pools. This combination of contracts simplifies the DeFi investment process, reducing complexity and enhancing user experience by providing a unified interface for automated and strategic asset management.

Findings Breakdown



Critical	1
Medium	0
Minor / Informative	30

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	0	0	0	0
Minor / Informative	30	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	VAM	Vault Address Manipulation	Unresolved
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DPI	Decimals Precision Inconsistency	Unresolved
●	EIS	Excessively Integer Size	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MEM	Misleading Error Messages	Unresolved
●	MCC	Missing Constructor Checks	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MSV	Missing Strategy Validation	Unresolved
●	MU	Modifiers Usage	Unresolved
●	NTTR	Native Token Transfer Restriction	Unresolved
●	PVI	Path Validation Inadequacy	Unresolved
●	PBV	Percentage Boundaries Validation	Unresolved

●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	RRS	Redundant Require Statement	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	OCTD	Transfers Contract's Tokens	Unresolved
●	WTD	Withdraw Token Discrepancy	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L06	Missing Events Access Control	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved

VAM - Vault Address Manipulation

Criticality	Critical
Location	new-zapper.sol#L1557
Status	Unresolved

Description

The contract permits users to specify the vault address in the `_swapAndStake` function without conducting any form of validation or authenticity checks on the provided `vault_addr` address. This oversight introduces a significant security vulnerability, as attackers could potentially input a malicious vault address designed to mimic legitimate vault behavior while harboring harmful intentions. Such a scenario could lead to unauthorized actions, including but not limited to, the redirection of funds to attacker-controlled addresses, manipulation of token swaps, and extraction of assets under false pretenses. The reliance on user-provided vault addresses without verification exposes the contract to risks of exploitation, potentially compromising the integrity of transactions and the security of assets managed by the contract.

Suppose that a malicious actor takes advantage of the lack of validation for the `vault_addr` parameter in the `_swapAndStake` function of the smart contract. Here's a step-by-step breakdown of the exploit:

1. The attacker identifies a smart contract that utilizes the `_swapAndStake` function to swap tokens and stake them in a liquidity pool.
2. The attacker creates a custom `vault_addr` that points to a malicious contract designed to imitate the behavior of a legitimate vault. This contract is under the attacker’s control.
3. When `_swapAndStake` is invoked with the malicious `vault_addr`, the `_getVaultPair` function returns a pair (IVault vault, IUniswapV2Pair pair) that includes the attacker's custom contracts. The IUniswapV2Pair contract is specifically crafted to interact with a malicious token created by the attacker.
4. The custom IUniswapV2Pair contract’s getReserves function is manipulated to return artificially high reserve values, ensuring the liquidity pair reserve check (reserveA > minimumAmount && reserveB > minimumAmount) is passed.

5. The attacker ensures the liquidity pair includes the malicious token and the contract’s intended swap token (tokenIn), thereby passing the input token presence check.
6. The swap is executed with the malicious tokens and the contract’s tokens. Given the router address is fixed and legitimate, the swap occurs as expected, but the presence of the malicious token in the liquidity pool skews the swap in favor of the attacker.
7. The `addLiquidity` function is called, adding liquidity to the pool.
8. The liquidity tokens generated from adding liquidity to the pool with the malicious token are deposited into the contract's preset `vault` address, which is legitimate and cannot be altered by the attacker. This deposit is intended to stake the liquidity tokens on behalf of the contract.
9. The attacker exploits the fact that the liquidity pool now contains a malicious token paired with the contract’s token.
10. The `_returnAssets` function is called and returns any remaining balances of tokens held by the contract to the caller.

Final Outcome: The attacker successfully exploits the contract by introducing a malicious token into the swap and liquidity addition process. The exploitation leads to the attacker draining the contract’s assets, despite the swap and staking operations occurring through the contract's preset and legitimate `vault` address.

This revised scenario highlights the vulnerability introduced by accepting an arbitrary `vault_addr` without sufficient validation. It underscores the necessity of rigorous security measures, including validation of external contract addresses and scrutiny of tokens involved in swaps and liquidity pools, to prevent exploitation.

```
    function _swapAndStake(address vault_addr, uint256 tokenAmountOutMin,
address tokenIn) public override {
        (IVault vault, IUniswapV2Pair pair) = _getVaultPair(vault_addr);

        (uint256 reserveA, uint256 reserveB, ) = pair.getReserves();
        require(reserveA > minimumAmount && reserveB > minimumAmount,
"Liquidity pair reserves too low");

        bool isInputA = pair.token0() == tokenIn;
        require(isInputA || pair.token1() == tokenIn, "Input token not
present in liquidity pair");

        address[] memory path = new address[](2);
        path[0] = tokenIn;
        path[1] = isInputA ? pair.token1() : pair.token0();

        uint256 fullInvestment =
IERC20(tokenIn).balanceOf(address(this));
        uint256 swapAmountIn;
        if (isInputA) {
            swapAmountIn = _getSwapAmount(fullInvestment, reserveA,
reserveB);
        } else {
            swapAmountIn = _getSwapAmount(fullInvestment, reserveB,
reserveA);
        }

        _approveTokenIfNeeded(path[0], address(router));
        uint256[] memory swappedAmounts = UniswapRouterV2(router)
            .swapExactTokensForTokens(
                swapAmountIn,
                tokenAmountOutMin,
                path,
                address(this),
                block.timestamp
            );

        _approveTokenIfNeeded(path[1], address(router));
        (, , uint256 amountLiquidity) =
UniswapRouterV2(router).addLiquidity(
            path[0],
            path[1],
            fullInvestment.sub(swappedAmounts[0]),
            swappedAmounts[1],
            1,
            1,
            address(this),
            block.timestamp
        );
```

```
        _approveTokenIfNeeded(address(pair), address(vault));
        vault.deposit(amountLiquidity);

        //taking receipt token and sending back to user
        vault.safeTransfer(msg.sender, vault.balanceOf(address(this)));

        _returnAssets(path);
    }
```

Recommendation

It is recommended to implement stringent validation mechanisms for `vault_addr` addresses provided by users within the functions. This could involve maintaining a whitelist of approved vault addresses and checking against this list before proceeding with any operations involving user-specified vault addresses. Additionally, incorporating mechanisms to verify the authenticity and integrity of the vault contract, such as checking for adherence to known vault interface signatures or employing registry contracts that maintain verified vault addresses, could further safeguard against manipulation. These measures will significantly reduce the risk of malicious activities facilitated by counterfeit vault addresses, ensuring the contract operates as intended and protecting users' assets from potential threats.

CR - Code Repetition

Criticality	Minor / Informative
Location	strategy-base-v3.sol#L277,377,397
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

Specifically, the `_distributePerformanceFeesAndDeposit` and `_distributePerformanceFeesBasedAmountAndDeposit` functions share similar code segments. Adidionally the `_swapSushiswap` function could reuse the `_swapSushiswapWithPath` function.

```
    function _distributePerformanceFeesAndDeposit() internal {
        uint256 _want = IERC20(want).balanceOf(address(this));

        if (_want > 0) {
            // Treasury fees
            IERC20(want).safeTransfer(
                IController(controller).treasury(),

_want.mul(performanceTreasuryFee).div(performanceTreasuryMax)
            );

            // Performance fee
            IERC20(want).safeTransfer(
                IController(controller).devfund(),
                _want.mul(performanceDevFee).div(performanceDevMax)
            );

            deposit();
        }
    }

    function _distributePerformanceFeesBasedAmountAndDeposit(uint256
_amount) internal {
        ...
        if (_amount > 0) {
            // Treasury fees
            IERC20(want).safeTransfer(
                IController(controller).treasury(),

_amount.mul(performanceTreasuryFee).div(performanceTreasuryMax)
            );

            // Performance fee
            IERC20(want).safeTransfer(
                IController(controller).devfund(),
                _amount.mul(performanceDevFee).div(performanceDevMax)
            );

            deposit();
        }
    }
```

```
function _swapSushiswap(
        address _from,
        address _to,
        uint256 _amount
    ) internal {
        require(_to != address(0));

        address[] memory path;

        if (_from == weth || _to == weth) {
            path = new address[](2);
            path[0] = _from;
            path[1] = _to;
        } else {
            path = new address[](3);
            path[0] = _from;
            path[1] = weth;
            path[2] = _to;
        }

        IERC20(_from).safeApprove(sushiRouter, 0);
        IERC20(_from).safeApprove(sushiRouter, _amount);
        UniswapRouterV2(sushiRouter).swapExactTokensForTokens(
            _amount,
            0,
            path,
            address(this),
            block.timestamp.add(60)
        );
    }

    function _swapSushiswapWithPath(
        address[] memory path,
        uint256 _amount
    ) internal {
        require(path[1] != address(0));

        IERC20(path[0]).safeApprove(sushiRouter, 0);
        IERC20(path[0]).safeApprove(sushiRouter, _amount);
        UniswapRouterV2(sushiRouter).swapExactTokensForTokens(
            _amount,
            0,
            path,
            address(this),
            block.timestamp.add(60)
        );
    }
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	strategy-base-v3.sol#L106
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

The contract's design permits privileged addresses to invoke critical withdrawal functions, thereby enhancing their control over the contract's assets and operational decisions. Additionally, specific addresses are granted the authority to modify essential contract parameters and roles, further centralizing control within a limited group of participants.

```
  function whitelistHarvester(address _harvester) external {
        require(msg.sender == governance ||
             msg.sender == strategist || harvesters[msg.sender], "not
authorized");
        harvesters[_harvester] = true;
    }

    function revokeHarvester(address _harvester) external {
        require(msg.sender == governance ||
             msg.sender == strategist, "not authorized");
        harvesters[_harvester] = false;
    }

    function setFeeDistributor(address _feeDistributor) external {
        require(msg.sender == governance, "not authorized");
        feeDistributor = _feeDistributor;
    }

    function setWithdrawalDevFundFee(uint256 _withdrawalDevFundFee)
external {
        require(msg.sender == timelock, "!timelock");
        withdrawalDevFundFee = _withdrawalDevFundFee;
    }

    function setWithdrawalTreasuryFee(uint256 _withdrawalTreasuryFee)
external {
        require(msg.sender == timelock, "!timelock");
        withdrawalTreasuryFee = _withdrawalTreasuryFee;
    }

    function setPerformanceDevFee(uint256 _performanceDevFee) external
{
        require(msg.sender == timelock, "!timelock");
        performanceDevFee = _performanceDevFee;
    }

    function setPerformanceTreasuryFee(uint256 _performanceTreasuryFee)
        external
    {
        require(msg.sender == timelock, "!timelock");
        performanceTreasuryFee = _performanceTreasuryFee;
    }

    function setStrategist(address _strategist) external {
        require(msg.sender == governance, "!governance");
        strategist = _strategist;
    }

    function setGovernance(address _governance) external {
        require(msg.sender == governance, "!governance");
```

```
        governance = _governance;
    }

    function setTimelock(address _timelock) external {
        require(msg.sender == timelock, "!timelock");
        timelock = _timelock;
    }

    function setController(address _controller) external {
        require(msg.sender == timelock, "!timelock");
        controller = _controller;
    }

    // Controller only function for creating additional rewards from
dust
    function withdraw(IERC20 _asset) external returns (uint256 balance)
{
        require(msg.sender == controller, "!controller");
        ...
    }

    // Withdraw partial funds, normally used with a vault withdrawal
    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        ...
        IERC20(want).safeTransfer(_vault,
_amount.sub(_feeDev).sub(_feeTreasury));
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	vault.sol#L146 zapper-base.sol#L57 vault-zapper-1.sol#L419
Status	Unresolved

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

```
function zapInETH(address vault, uint256 tokenAmountOutMin, address
tokenIn) external payable{

require(msg.value >= minimumAmount, "Insignificant input amount");

WETH(weth).deposit{value: msg.value}();

// allows us to zapIn if eth isn't part of the original pair
if (tokenIn != weth){
    uint256 _amount = IERC20(weth).balanceOf(address(this));

    (, IUniswapV2Pair pair) = _getVaultPair(vault);

    (uint256 reserveA, uint256 reserveB, ) = pair.getReserves();
    require(reserveA > minimumAmount && reserveB > minimumAmount,
"Liquidity pair reserves too low");

    ...

    function getRatio() public view returns (uint256) {
        return balance().mul(1e18).div(totalSupply());
    }
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

All the decimals could be normalized to 18 since it represents the ERC20 token with the greatest digits.

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	vault.sol#L17
Status	Unresolved

Description

The contract is using a bigger unsigned integer data type that the maximum size that is required. By using an unsigned integer data type larger than necessary, the smart contract consumes more storage space and requires additional computational resources for calculations and operations involving these variables. This can result in higher transaction costs, longer execution times, and potential scalability bottlenecks.

```
uint256 public min = 9500;
```

Recommendation

To address the inefficiency associated with using an oversized unsigned integer data type, it is recommended to accurately determine the required size based on the range of values the variable needs to represent.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	strategy-uni-base.sol#L70 strategy-base-v3.sol#L72 strategies/strategy-uni-base.sol#L70 strategies/strategy-base-v3.sol#L72
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
want
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	strategy-uni-base.sol#L64,65,66,67,68,80,281,311 strategy-base-v3.sol#L66,67,68,69,70,82,282,312 strategies/strategy-uni-base.sol#L64,65,66,67,68,80,281,311 strategies/strategy-base-v3.sol#L66,67,68,69,70,82,282,312
Status	Unresolved

Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(_want != address(0))
require(_governance != address(0))
require(_strategist != address(0))
require(_controller != address(0))
require(_timelock != address(0))

require(
            harvesters[msg.sender] ||
                msg.sender == governance ||
                msg.sender == strategist
        )
require(_to != address(0))
require(path[1] != address(0))
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MCC - Missing Constructor Checks

Criticality	Minor / Informative
Location	vault.sol#L24 controller.sol#L44 zapper-base.sol#L27 vault-zapper-1.sol#L388
Status	Unresolved

Description

The contract is currently designed to initialize with external addresses for `token`, `governance`, `timelock`, `controller` and `router` parameters within its constructor. However, it lacks critical validation checks to ensure that these addresses are not the zero address (0x0). The absence of such validation exposes the contract to potential risks, as initializing crucial components like the token, governance, timelock, or controller with the zero address could lead to malfunctioning of the contract, hindering operations such as token transfers, governance decisions, and access control management. This oversight may result in a scenario where the contract is deployed in an unusable state or susceptible to security vulnerabilities, affecting the overall integrity and functionality of the contract.

```
    constructor(
        address _token,
        address _governance,
        address _timelock,
        address _controller
    )
        ERC20(
            string(abi.encodePacked("freezing ",
ERC20(_token).name())),
            string(abi.encodePacked("s", ERC20(_token).symbol()))
        )
    {
        _setupDecimals(ERC20(_token).decimals());
        token = IERC20(_token);
        governance = _governance;
        timelock = _timelock;
        controller = _controller;
    }

    constructor(
        address _governance,
        address _strategist,
        address _timelock,
        address _devfund,
        address _treasury
    ) {
        governance = _governance;
        strategist = _strategist;
        timelock = _timelock;
        devfund = _devfund;
        treasury = _treasury;
    }

    constructor(address _router) {
        ...
        router = _router;
    }
```

Recommendation

It is recommended to implement checks within the constructor to validate that the addresses for the `token`, `governance`, `timelock`, `controller` and `router` are not the zero address before setting them. This can be achieved by adding require statements for each parameter to ensure they are not equal to the zero address. Such a preventative measure will enhance the contract's security and robustness by

ensuring that all critical components are properly initialized, thereby preventing potential operational failures or security loopholes. This validation should be considered a standard practice in smart contract development to safeguard against common pitfalls associated with improper initialization.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	controller.sol#L80,95,130 new-zapper.sol#L1206
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
    function setOneSplit(address _onesplit) public {
        require(msg.sender == governance, "!governance");
        onesplit = _onesplit;
    }

    function setVault(address _token, address _vault) public {
        require(
            msg.sender == strategist || msg.sender == governance,
            "!strategist"
        );
        require(vaults[_token] == address(0), "vault");
        vaults[_token] = _vault;
    }

    function setStrategy(address _token, address _strategy) public {
        ...
        strategies[_token] = _strategy;
    }

    function addToWhitelist(address _vault) external onlyGovernance {
        whitelistedVaults[_vault] = true;
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MSV - Missing Strategy Validation

Criticality	Minor / Informative
Location	controller.sol#L143
Status	Unresolved

Description

he contract is designed to facilitate the redirection of funds to investment strategies via the `earn` function, which takes `_token` and `_amount` as parameters. This function locates the corresponding strategy for the given `_token` but fails to validate whether the retrieved `_strategy` address actually exists or is valid. Without this check, if a strategy for the specified `_token` does not exist within the strategies mapping, the function will proceed with operations on an undefined or zero address, leading to potential failure of the transaction or unintended behavior.

```
    function earn(address _token, uint256 _amount) public {
        address _strategy = strategies[_token];
        address _want = IStrategy(_strategy).want();
        if (_want != _token) {
            address converter = converters[_token][_want];
            IERC20(_token).safeTransfer(converter, _amount);
            _amount = Converter(converter).convert(_strategy);
            IERC20(_want).safeTransfer(_strategy, _amount);
        } else {
            IERC20(_token).safeTransfer(_strategy, _amount);
        }
        IStrategy(_strategy).deposit();
    }
```

Recommendation

It is recommended to incorporate a validation check immediately after retrieving the `_strategy` address from the strategies mapping to ensure it is not the zero address. This can be achieved with a `require` statement asserting that `_strategy` is a valid contract address, coupled with a clear error message if the condition is not met. This preventative measure ensures that only defined and active strategies are interacted with, minimizing the risk of errors or lost funds.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	vault.sol#L50,56,61,66 controller.sol#L59,64,69,74,80,85,96,104,109,119,131,163 strategy-uni-base.sol#L111,117,122 strategy-base-v3.sol#L107,119,124
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(msg.sender == governance, "!governance");
require(msg.sender == timelock, "!timelock");
require(
            msg.sender == strategist || msg.sender ==
governance,
            "!strategist"
        );
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

NTTR - Native Token Transfer Restriction

Criticality	Minor / Informative
Location	vault-zapper-1.sol#L397
Status	Unresolved

Description

The contract's `_returnAssets` function is designed to return assets to the msg.sender, including both ERC20 tokens and native tokens (e.g., ETH). However, the function's current implementation only attempts to transfer native tokens if there is a positive balance of an ERC20 token, due to its reliance on the balance check within the loop iterating over ERC20 tokens. This design oversight means that if the contract does not hold any ERC20 tokens (i.e., their balance is zero), but does hold a balance of native tokens intended for distribution, these native tokens will not be distributed. This limitation restricts the contract's ability to return native tokens independently of ERC20 token balances, potentially leading to scenarios where native tokens remain undistributed despite the intention to return them to the sender.

```
    function _returnAssets(address[] memory tokens) internal {
        uint256 balance;
        for (uint256 i; i < tokens.length; i++) {
            balance =
IERC20(tokens[i]).balanceOf(address(this));
            if (balance > 0) {
                if (tokens[i] == weth) {
                    WETH(weth).withdraw(balance);
                    (bool success, ) = msg.sender.call{value:
balance}(
                        new bytes(0)
                    );
                    require(success, "ETH transfer failed");
                } else {
                    IERC20(tokens[i]).safeTransfer(msg.sender,
balance);
                }
            }
        }
    }
```

Recommendation

It is recommended to decouple the distribution of native tokens from the ERC20 token balance check within the `_returnAssets` function. Specifically, the contract should include a separate logic branch to handle the transfer of native tokens, ensuring that such transfers are not conditional on the presence or balance of ERC20 tokens. This could involve adding a check for the contract's native token balance outside of the ERC20 token iteration loop and executing the native token transfer logic independently. Implementing this adjustment will ensure that native tokens can be returned to the msg.sender even in cases where the contract holds no ERC20 tokens, thereby enhancing the contract's asset distribution capabilities and aligning with the intended functionality.

PVI - Path Validation Inadequacy

Criticality	Minor / Informative
Location	strategy-base-v3.sol#L309
Status	Unresolved

Description

The contract is designed to facilitate token swaps via Sushiswap, utilizing a specified path for the swap operation. However, the current implementation only superficially checks the validity of the path by ensuring the second address in the path array is not a zero address. This approach overlooks two critical aspects which is ensuring that the path array contains more than one address to constitute a valid swap path and verifying that the path does not contain duplicate addresses, which could lead to unnecessary swaps or potential manipulation. The lack of comprehensive path validation could result in inefficient swaps or expose the contract to vulnerabilities associated with unexpected path configurations.

```
    function _swapSushiswapWithPath(
        address[] memory path,
        uint256 _amount
    ) internal {
        require(path[1] != address(0));

        IERC20(path[0]).safeApprove(sushiRouter, 0);
        IERC20(path[0]).safeApprove(sushiRouter, _amount);
        UniswapRouterV2(sushiRouter).swapExactTokensForTokens(
            _amount,
            0,
            path,
            address(this),
            block.timestamp.add(60)
        );
    }
```

Recommendation

It is recommended to enhance the path validation logic within the `_swapSushiswapWithPath` function to ensure robust and secure swap operations. Specifically, the contract should include checks to verify that the path array contains at

least two unique addresses, confirming a legitimate swap path from one token to another. Additionally, implementing a mechanism to detect and reject duplicate addresses within the path array will prevent redundant swaps and protect against potential exploits. These improvements will contribute to the efficiency and security of the swap functionality, ensuring that token swaps are conducted as intended and safeguarding against manipulative practices.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	strategy-uni-base.sol#L126,175 controller.sol#L125
Status	Unresolved

Description

The contract is using the variables `withdrawalDevFundFee` and `withdrawalTreasuryFee` for calculations. However, these variables are used in multiplication operations and if `withdrawalDevFundFee` or `withdrawalTreasuryFee` is set to a value greater than `100000`, it could lead to incorrect calculations, potentially causing unintended behavior or financial discrepancies within the contract's operations.

```
   function setWithdrawalDevFundFee(uint256
_withdrawalDevFundFee) external {
        require(msg.sender == timelock, "!timelock");
        withdrawalDevFundFee = _withdrawalDevFundFee;
    }

    function setWithdrawalTreasuryFee(uint256
_withdrawalTreasuryFee) external {
        require(msg.sender == timelock, "!timelock");
        withdrawalTreasuryFee = _withdrawalTreasuryFee;
    }

    function withdraw(uint256 _amount) external {
        ...

        uint256 _feeDev =
_amount.mul(withdrawalDevFundFee).div(
            withdrawalDevFundMax
        );
        ...
        uint256 _feeTreasury =
_amount.mul(withdrawalTreasuryFee).div(
            withdrawalTreasuryMax
        );
        ....
        }
```

```
    function setConvenienceFee(uint256 _convenienceFee) external
{
        require(msg.sender == timelock, "!timelock");
        convenienceFee = _convenienceFee;
    }
```

Recommendation

It is recommended to ensure that the values of `withdrawalDevFundFee` and `withdrawalTreasuryFee` cannot exceed `100000`. This can be achieved by adding checks whenever these variables are set.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	vault-zapper-1.sol#L677
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
        if(swapToken == weth || desiredToken == weth){
            address[] memory path = new address[](2);
            path[0] = swapToken;
            path[1] = desiredToken;

            _approveTokenIfNeeded(path[0], address(router));
            UniswapRouterV2(router).swapExactTokensForTokens(
                IERC20(swapToken).balanceOf(address(this)),
                desiredTokenOutMin,
                path,
                address(this),
                block.timestamp
            );

        }else {
            address[] memory path = new address[](3);
            path[0] = swapToken;
            path[1] = weth;
            path[2] = desiredToken;

            _approveTokenIfNeeded(path[0], address(router));
            UniswapRouterV2(router).swapExactTokensForTokens(
                IERC20(swapToken).balanceOf(address(this)),
                desiredTokenOutMin,
                path,
                address(this),
                block.timestamp
            );
        }
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	vault.sol#L76 controller.sol#L143
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
    function earn() public {
        uint256 _bal = available();
        token.safeTransfer(controller, _bal);
        IController(controller).earn(address(token), _bal);
    }

    function earn(address _token, uint256 _amount) public {
        address _strategy = strategies[_token];
        address _want = IStrategy(_strategy).want();
        if (_want != _token) {
            address converter = converters[_token][_want];
            IERC20(_token).safeTransfer(converter, _amount);
            _amount = Converter(converter).convert(_strategy);
            IERC20(_want).safeTransfer(_strategy, _amount);
        } else {
            IERC20(_token).safeTransfer(_strategy, _amount);
        }
        IStrategy(_strategy).deposit();
    }
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

`Actual Transferred Amount = Balance After Transfer - Balance Before Transfer`

RRS - Redundant Require Statement

Criticality	Minor / Informative
Location	new-zapper.sol#L33
Status	Unresolved

Description

The contract utilizes a `require` statement within the `add` function aiming to prevent overflow errors. This function is designed based on the SafeMath library's principles. In Solidity version 0.8.0 and later, arithmetic operations revert on overflow and underflow, making the overflow check within the function redundant. This redundancy could lead to extra gas costs and increased complexity without providing additional security.

```
function add(uint256 a, uint256 b) internal pure returns
(uint256) {
      uint256 c = a + b;
      require(c >= a, "SafeMath: addition overflow");
      return c;
}
```

Recommendation

It is recommended to remove the `require` statement from the add function since the contract is using a Solidity pragma version equal to or greater than 0.8.0. By doing so, the contract will leverage the built-in overflow and underflow checks provided by the Solidity language itself, simplifying the code and reducing gas consumption. This change will uphold the contract's integrity in handling arithmetic operations while optimizing for efficiency and cost-effectiveness.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	vault.sol strategy-uni-base.sol strategy-base-v3.sol new-zapper.sol controller.sol vaults/vault.sol strategies/strategy-uni-base.sol strategies/strategy-base-v3.sol strategies/sushi/sushi-zapper/new-zapper.sol controllers/sushi-controller.sol controllers/stargate-controller.sol controllers/plutus-controller.sol controllers/jones-controller.sol controllers/hop-controller.sol controllers/gmx-controller.sol controllers/fish-controller.sol controllers/dpx-controller.sol controllers/dodo-controller.sol controllers/controller.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	vault.sol#L55 controller.sol#L94 strategy-uni-base.sol#L104 new-zapper.sol#L1206
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
    function setGovernance(address _governance) public {
        require(msg.sender == governance, "!governance");
        governance = _governance;
    }

    function setTimelock(address _timelock) public {
        require(msg.sender == timelock, "!timelock");
        timelock = _timelock;
    }

    function setController(address _controller) public {
        require(msg.sender == timelock, "!timelock");
        controller = _controller;
    }
 function setVault(address _token, address _vault) public {
        require(
            msg.sender == strategist || msg.sender ==
governance,
            "!strategist"
        );
        require(vaults[_token] == address(0), "vault");
        vaults[_token] = _vault;
    }

    function approveVaultConverter(address _converter) public {
        require(msg.sender == governance, "!governance");
        approvedVaultConverters[_converter] = true;
    }

    function revokeVaultConverter(address _converter) public {
        require(msg.sender == governance, "!governance");
        approvedVaultConverters[_converter] = false;
    }

    function approveStrategy(address _token, address _strategy)
public {
        require(msg.sender == timelock, "!timelock");
        approvedStrategies[_token][_strategy] = true;
    }
    function whitelistHarvester(address _harvester) external {
        require(msg.sender == governance ||
             msg.sender == strategist ||
harvesters[msg.sender], "not authorized");
        harvesters[_harvester] = true;
    }

    function revokeHarvester(address _harvester) external {
        require(msg.sender == governance ||
             msg.sender == strategist, "not authorized");
        harvesters[_harvester] = false;
```

```
    }

    function setFeeDistributor(address _feeDistributor)
external {
        require(msg.sender == governance, "not authorized");
        feeDistributor = _feeDistributor;
    }

    function addToWhitelist(address _vault) external
onlyGovernance {
        whitelistedVaults[_vault] = true;
    }

    function removeFromWhitelist(address _vault) external
onlyGovernance {
        whitelistedVaults[_vault] = false;
    }
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	controller.sol#L172
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `inCaseTokensGetStuck` or `inCaseStrategyTokenGetStuck` functions.

```
   function inCaseTokensGetStuck(address _token, uint256
_amount) public {
        require(
            msg.sender == strategist || msg.sender ==
governance,
            "!governance"
        );
        IERC20(_token).safeTransfer(msg.sender, _amount);
    }

    function inCaseStrategyTokenGetStuck(address _strategy,
address _token)
        public
    {
        require(
            msg.sender == strategist || msg.sender ==
governance,
            "!governance"
        );
        IStrategy(_strategy).withdraw(_token);
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

WTD - Withdraw Token Discrepancy

Criticality	Minor / Informative
Location	strategy-base-v3.sol#L177 vault.sol#L121 controller.sol#L245
Status	Unresolved

Description

The contract is designed to facilitate withdrawals by transferring a specified `token` directly to the user. If the vault's balance of this token is insufficient, it triggers a withdrawal from the controller, which in turn calls the strategy's withdrawal function. The strategy responds by returning `want` tokens to the vault. This process inherently assumes that the `want` tokens are identical to the tokens the vault intends to transfer to the user. However, if the `want` token differs from the vault's token, this mechanism can lead to discrepancies and potential issues in fulfilling withdrawal requests accurately, risking the vault's reliability.

```
   function withdraw(uint256 _amount) external {
        ...

        address _vault =
IController(controller).vaults(address(want));
        require(_vault != address(0), "!vault"); // additional
protection so we don't burn the funds

        IERC20(want).safeTransfer(_vault,
_amount.sub(_feeDev).sub(_feeTreasury));
    }

    function withdraw(address _token, uint256 _amount) public {
        require(msg.sender == vaults[_token], "!vault");
        IStrategy(strategies[_token]).withdraw(_amount);
    }

    function withdraw(uint256 _shares) public {
       ...
        // Check balance
        uint256 b = token.balanceOf(address(this));
        if (b < r) {
            uint256 _withdraw = r.sub(b);
            IController(controller).withdraw(address(token),
_withdraw);
            uint256 _after = token.balanceOf(address(this));
            uint256 _diff = _after.sub(b);
            if (_diff < _withdraw) {
                r = b.add(_diff);
            }
        }

        token.safeTransfer(msg.sender, r);
        emit Withdraw(tx.origin, block.timestamp, r, _shares);
    }
```

Recommendation

It is recommended to implement a validation mechanism within the vault to ensure alignment between the `want` tokens returned by the strategy and the tokens intended for user withdrawals. If a discrepancy exists, the vault should have a conversion process in place to swap `want` tokens into the desired withdrawal tokens before executing the transfer to the user. This could involve integrating with a decentralized exchange or utilizing a dedicated conversion contract. Additionally, clear documentation and error handling

should be incorporated to manage situations where a direct conversion may not be possible or economical. This approach will enhance the contract's flexibility and reliability, ensuring that user withdrawals are processed efficiently and accurately.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	strategy-uni-base.sol#L27,53 strategy-base-v3.sol#L28,29
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
address public univ2Router2 =
0xE54Ca86531e17Ef3616d22Ca28b0D458b6C89106
address public uniswapRouterV2 =
0xcDAeC65495Fa5c0545c5a405224214e3594f30d8
address public uniswapRouterV3 =
0xE592427A0AEce92De3Edee1F18E0157C05861564
address public sushiRouter =
0x1b02dA8Cb0d097eB8D57A175b88c7D8b47997506
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	vault.sol#L49,55,60,65,89,121 strategy-uni-base.sol#L104,110,116,121,126,131,136,143,148,153,158,167,175,203,239 strategy-base-v3.sol#L106,112,118,123,128,133,138,145,150,155,160,169,177,205,241 new-zapper.sol#L929,931,962,1208,1213,1217,1238,1362,1398,1486,1557,1609,1628 controller.sol#L58,63,68,73,79,84,89,94,103,108,113,118,124,129,143,157,161,169,177,188,189,205,206,245
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
uint256 _min
address _governance
address _timelock
address _controller
uint256 _amount
uint256 _shares
address _harvester
address _feeDistributor
uint256 _withdrawalDevFundFee
uint256 _withdrawalTreasuryFee
uint256 _performanceDevFee
uint256 _performanceTreasuryFee
address _strategist
IERC20 _asset

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	strategy-uni-base.sol#L145,150 strategy-base-v3.sol#L147,152
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
strategist = _strategist
governance = _governance
```

Recommendation

To avoid this issue, it's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	vault.sol#L52 strategy-uni-base.sol#L133,140 strategy-base-v3.sol#L135,142
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
min = _min
performanceDevFee = _performanceDevFee
performanceTreasuryFee = _performanceTreasuryFee
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	strategy-uni-base.sol#L276,307,324,344 strategy-base-v3.sol#L277,308,330,355,376,396 new-zapper.sol#L331,357,382,392,642,663,701,761,766 controller.sol#L250
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function _swapUniswap(
        address _from,
        address _to,
        uint256 _amount
    ) internal {
        require(_to != address(0));
...
            _amount,
            0,
            path,
            address(this),
            block.timestamp.add(60)
        );
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	controller.sol#L134
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
require(approvedStrategies[_token][_strategy] == true,
"!approved")
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	new-zapper.sol#L1610,1621
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 halfInvestment = investmentA.div(2)
swapAmount = investmentA.sub(
            Babylonian.sqrt(
                (halfInvestment * halfInvestment * nominator) /
denominator
            )
        )
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	new-zapper.sol#L1222
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 i
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	vault.sol#L37,38,39,57,62,67 strategy-uni-base.sol#L118,145,150,155,160 strategy-base-v3.sol#L120,147,152,157,162 controller.sol#L51,52,53,54,55,60,65,70,81,86,91
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
governance = _governance
timelock = _timelock
controller = _controller
feeDistributor = _feeDistributor
strategist = _strategist
devfund = _devfund
treasury = _treasury
onesplit = _onesplit
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	strategy-uni-base.sol#L248 strategy-base-v3.sol#L250 new-zapper.sol#L311,410 controller.sol#L257
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
            let succeeded := delegatecall(
                sub(gas(), 5000),
                _target,
                add(_data, 0x20),
                mload(_data),
...

            switch iszero(succeeded)
                case 1 {
                    // throw if delegatecall failed
                    revert(add(response, 0x20), size)
                }
        }

...
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Vault	Implementation	ERC20		
		Public	✓	ERC20
	balance	Public		-
	setMin	External	✓	-
	setGovernance	Public	✓	-
	setTimelock	Public	✓	-
	setController	Public	✓	-
	available	Public		-
	earn	Public	✓	-
	depositAll	External	✓	-
	deposit	Public	✓	-
	withdrawAll	External	✓	-
	harvest	External	✓	-
	withdraw	Public	✓	-
	getRatio	Public		-
StrategyUniBase	Implementation			
		Public	✓	-

	balanceOfWant	Public		-
	balanceOfPool	Public		-
	balanceOf	Public		-
	getName	External		-
	whitelistHarvester	External	✓	-
	revokeHarvester	External	✓	-
	setFeeDistributor	External	✓	-
	setWithdrawalDevFundFee	External	✓	-
	setWithdrawalTreasuryFee	External	✓	-
	setPerformanceDevFee	External	✓	-
	setPerformanceTreasuryFee	External	✓	-
	setStrategist	External	✓	-
	setGovernance	External	✓	-
	setTimelock	External	✓	-
	setController	External	✓	-
	deposit	Public	✓	-
	withdraw	External	✓	-
	withdraw	External	✓	-
	withdrawForSwap	External	✓	-
	withdrawAll	External	✓	-
	_withdrawAll	Internal	✓	
	_withdrawSome	Internal	✓	
	harvest	Public	✓	-

	execute	Public	Payable	-
	_swapUniswap	Internal	✓	
	_swapUniswapWithPath	Internal	✓	
	_distributePerformanceFeesAndDeposit	Internal	✓	
	_distributePerformanceFeesBasedAmountAndDeposit	Internal	✓	
StrategyBaseV3	Implementation			
		Public	✓	-
	balanceOfWant	Public		-
	balanceOfPool	Public		-
	balanceOf	Public		-
	getName	External		-
	whitelistHarvester	External	✓	-
	revokeHarvester	External	✓	-
	setFeeDistributor	External	✓	-
	setWithdrawalDevFundFee	External	✓	-
	setWithdrawalTreasuryFee	External	✓	-
	setPerformanceDevFee	External	✓	-
	setPerformanceTreasuryFee	External	✓	-
	setStrategist	External	✓	-
	setGovernance	External	✓	-
	setTimelock	External	✓	-
	setController	External	✓	-

	deposit	Public	✓	-
	withdraw	External	✓	-
	withdraw	External	✓	-
	withdrawForSwap	External	✓	-
	withdrawAll	External	✓	-
	_withdrawAll	Internal	✓	
	_withdrawSome	Internal	✓	
	harvest	Public	✓	-
	execute	Public	Payable	-
	_swapSushiswap	Internal	✓	
	_swapSushiswapWithPath	Internal	✓	
	_swapUniswap	Internal	✓	
	_swapUniswapWithPath	Internal	✓	
	_distributePerformanceFeesAndDeposit	Internal	✓	
	_distributePerformanceFeesBasedAmountAndDeposit	Internal	✓	
Controller	Implementation			
		Public	✓	-
	setDevFund	Public	✓	-
	setTreasury	Public	✓	-
	setStrategist	Public	✓	-
	setSplit	Public	✓	-
	setOneSplit	Public	✓	-

	setGovernance	Public	✓	-
	setTimelock	Public	✓	-
	setVault	Public	✓	-
	approveVaultConverter	Public	✓	-
	revokeVaultConverter	Public	✓	-
	approveStrategy	Public	✓	-
	revokeStrategy	Public	✓	-
	setConvenienceFee	External	✓	-
	setStrategy	Public	✓	-
	earn	Public	✓	-
	balanceOf	External		-
	withdrawAll	Public	✓	-
	inCaseTokensGetStuck	Public	✓	-
	inCaseStrategyTokenGetStuck	Public	✓	-
	getExpectedReturn	Public		-
	yearn	Public	✓	-
	withdraw	Public	✓	-
	_execute	Internal	✓	
ZapperBase	Implementation			
		Public	✓	-
		External	Payable	-
	addToWhitelist	External	✓	onlyGovernance

	removeFromWhitelist	External	✓	onlyGovernance
	_getSwapAmount	Public		-
	_returnAssets	Internal	✓	
	_swapAndStake	Public	✓	-
	zapInETH	External	Payable	onlyWhitelistedVaults
	zapIn	External	✓	onlyWhitelistedVaults
	zapOutAndSwap	Public	✓	-
	_removeLiquidity	Internal	✓	
	_getVaultPair	Internal		
	_approveTokenIfNeeded	Internal	✓	
	zapOut	External	✓	onlyWhitelistedVaults
VaultZapEthSushi	Implementation	ZapperBase		
		Public	✓	ZapperBase
	zapOutAndSwap	Public	✓	onlyWhitelistedVaults
	zapOutAndSwapEth	Public	✓	onlyWhitelistedVaults
	_swapAndStake	Public	✓	-
	_getSwapAmount	Public		-
	estimateSwap	Public		-

Summary

The Contrax contract is designed to operate within the decentralized finance (DeFi) ecosystem, embodying principles of decentralization and financial innovation. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope’s prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope’s position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io