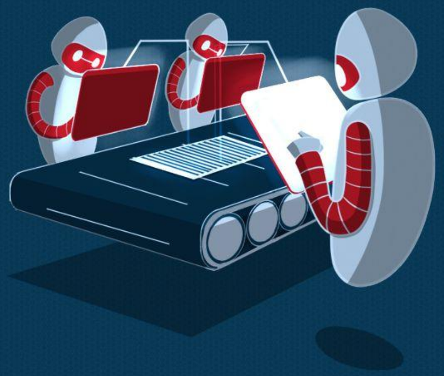
BlockApex

SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



Powered by XORD

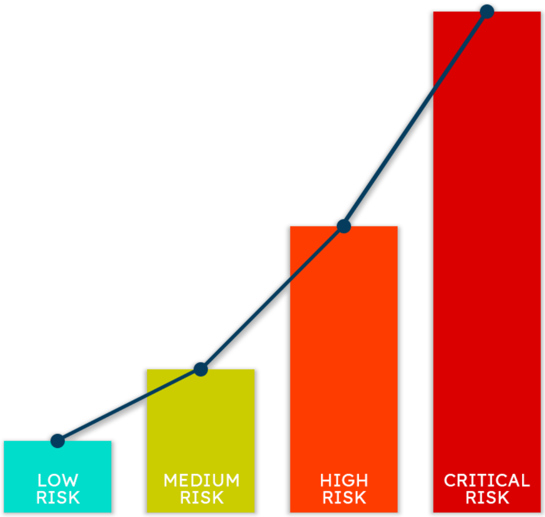
PREFACE

Objectives

The purpose of this document is to highlight the identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and intellectual property of the client. It also contains information about potential risks and the processes involved in mitigating/exploiting the risks mentioned below.

The usage of information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly with the intention to aid our growing blockchain community; under the discretion of the client.

Key understandings



CRITICAL RISK xxxx	Critical vulnerabilities are too easy to exploit and can lead to damages/loss in assets or manipulations.
HIGH RISK xxx	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution.
MEDIUM RISK xx	Medium-level vulnerabilities are equally imperative to fix but they tend to have minimal impact on asset loss or data manipulations.
LOW RISK x	Lowest-level vulnerabilities, informational errors, violating code styles/practices usually can't affect smart contract execution; hence they can be ignored.

TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by VoirStudio (Client) for the purpose of conducting a Smart Contract Audit/Code Review. This document presents the findings of our analysis which started from 26th Jan 2022.

Name
Unipilot-V2
Auditor
Moazzam Arif | Kaif Ahmed | Muhammad Jarir Uddin
Platform
Ethereum/Solidity
Type of review
Manual Code Review | Automated Code Review
Methods
Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
Git repository
https://github.com/VoirStudio/unipilot-v2/tree/revamp-structure
White paper/ Documentation
https://unipilot.gitbook.io/unipilot/
Document log
Initial Audit: 14th Feb 2022 (complete)
Quality Control: 14th - 22nd March 2022
Final Audit: 26th March 2022 (Complete)

Scope

The git-repository shared was checked for common code violations along with vulnerability-specific probing to detect **major issues/vulnerabilities**. Some specific checks are as follows:

Code review		**Functional review**
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	ERC20 API violation	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset’s integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

Unipilot is an automated liquidity manager designed to maximize ”in-range” intervals for capital through an optimized rebalancing mechanism of liquidity pools. Unipilot V2 also detects the volatile behavior of the pools and pulls liquidity until the pool gets stable to save the pool from impairment loss.

System Architecture

The protocol is built to support multiple dexes (decentralized exchanges) for liquidity management. Currently it supports only Uniswap v3’s liquidity. In future, the protocol will support other decentralized exchanges like Sushiswap (Trident). The architecture is designed to keep in mind the future releases.

The protocol has **5** main smart contracts and their dependent libraries.

UnipilotActiveFactory.sol

The smart contract is the entry point in the protocol. It allows users to create a vault if it's not present on protocol. Nevertheless Active vaults can only be created by governance.

UnipilotPassiveFactory.sol

The smart contract is the entry point in the protocol. It allows users to create a vault if it's not present on protocol. However passive vaults can be created by anyone.

UnipilotActiveVault.sol

Vault contract allows users to deposit, withdraw, readjustLiquidity and collect fees on liquidity. It mints an LPs to its users representing their individual shares. It also has a pullLiquidity function if liquidity is needed to be pulled.

UnipilotPassiveVault.sol

PassiveVault contract allows users to deposit, withdraw, readjustLiquidity and collect fees on liquidity. It mints an LPs to its users representing their individual shares.

UnipilotStrategy.sol

The smart contract to fetch and process ticks' data from Uniswap. It also decides the bandwidth of the ticks to supply liquidity.

Methodology & Scope

The codebase was audited in an iterative process. Fixes were applied on the way and updated contracts were examined for more bugs. We used a combination of static analysis tool (slither) and Automated testing tool (Foundry) which indicated some of the critical bugs in the code. We also did manual reviews of the code to find logical bugs, code optimizations, solidity design patterns, code style and the bugs/ issues detected by automated tools.

Privileged Roles

In a production environment, the unipilot protocol sets the address for a governance that exercises a privileged position over the factory and vault contracts in the system. The governor has the power to initiate a transfer of the governor role to a new address.

The governance address is capable of executing a set of actions including:

- Controlling the various whitelists for vaults in the factory contract
- Toggle a vault as whitelisted
- Set unipilot details to update addresses of Strategy and Index Fund contracts along with the Index Fund Fee Percentage
- Set an operator address

The operator address has following activities it can be used for:

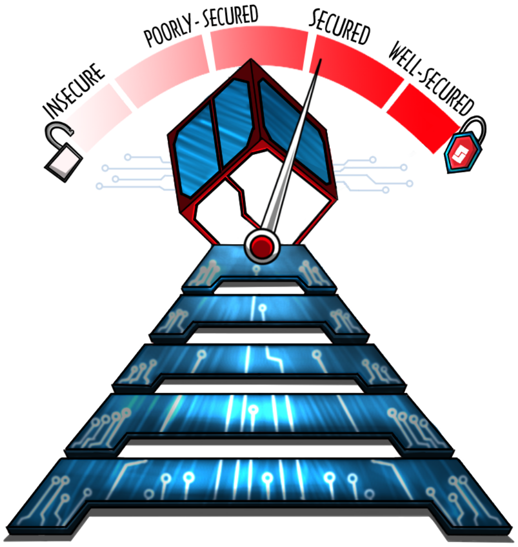
- Pull liquidity for a vault contract from its position on a Uniswap V3 Pool
- Call Readjust Liquidity to burn from and mint all liquidity back to a less volatile position on the Uniswap V3 Pool.

AUDIT REPORT

Executive Summary

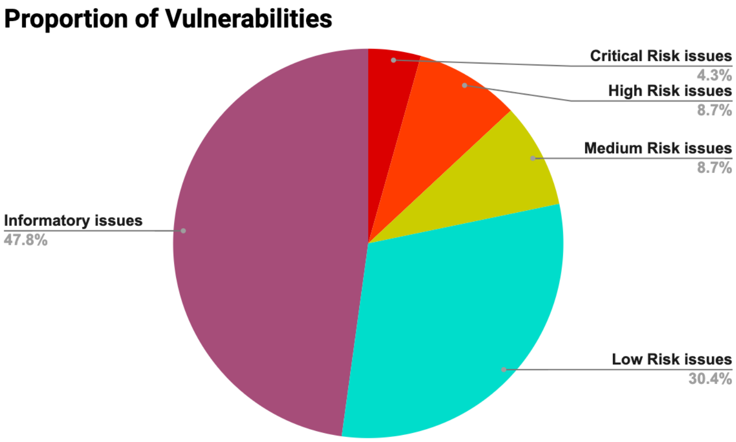
The analysis indicates that some of the functionalities in the contracts audited are **working properly**.

Our team performed a technique called "Filtered Audit", where the contract was separately audited by two individuals. After their thorough and rigorous process of manual testing, an automated review was carried out using Mythril, MythX, Surya and Slither. All the flags raised were manually reviewed and re-tested.



Our team found:

# of issues	Severity of the risk
1	Critical Risk issue(s)
2	High Risk issue(s)
2	Medium Risk issue(s)
7	Low Risk issue(s)
11	Informatory issue(s)



Findings

#	Findings	Risk	Status
1.	Deposit ether with zero value	Critical	Fixed
2.	*init()* should have only once check	High	Acknowledged
3.	Pulled liquidity rationale	High	Pending
4.	Unoptimized user liquidity is held in a vault.	Medium	Acknowledged
5.	Deposit should have non reentrant checks placed in both vaults	Medium	Fixed
6.	Zero Address checks not placed in contract *constructors()*	Low	Acknowledged
7.	Safecast in UnipilotPassiveVault.sol line 232 & 233 for `swapPercentage`	Low	Acknowledged
8.	Storage Layout is unoptimized.	Low	Acknowledged
9.	Check max cap for indexFundPercentage and swapPercentage	Low	Fixed
10.	*Deposit()* function optimization	Low	Fixed
11.	toggleWhitelistAccount() redundant.	Low	Pending
12.	*SortWeth()* optimization	Low	Acknowledged
13.	No natspec documentation in UnipilotActiveVault.sol	Informatory	Acknowledged

14.	Mark *token0* and *token1* as immutable in *UnipilotActivevault.sol* and *UnipilotPassiveVault.sol*	Informatory	Acknowledged
15.	*onlyGovernance() modifier in passive vault contract*	Informatory	Pending
16.	*_WETH* address should be hardcoded before production wherever necessary	Informatory	Acknowledged
17.	Factory function *createVault()* optimization version.	Informatory	Acknowledged
18.	Assignment of Params in order to receive the function signature.	Informatory	Acknowledged
19.	Order of functions in solidity style guides.	Informatory	Acknowledged
20.	uint256 can be cheaper than uint8	Informatory	Acknowledged
21.	*pullLiquidity()* is vulnerable in its current execution	Informatory	Pending
22.	Spelling mistakes in function signatures.	Informatory	Acknowledged
23.	Mark private functions as internal	Informatory	Acknowledged

Critical risk issues

1. Deposit ether with zero value.

Description:
If a deposit of tokens with ether as one is made, all the while the contract has pulled liquidity into the vault, the user making a deposit with 0 value can use the vault's ether to execute a successful transaction.

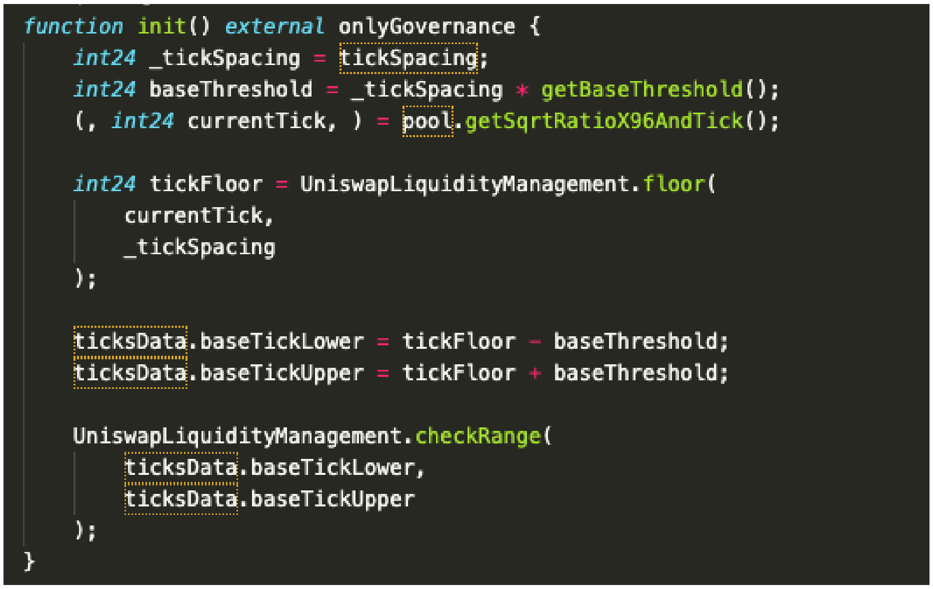
Remedy:
Introduce a *RefundETH()* function that ensures a proper transfer of value either be in ethers or an ERC compatible version (WETH)

Status:
Fixed

High risk issues

1. init() should have only once check

Description:
Calling *init()* any time after the first time it has been called, can lead to permanent loss of position at uniswap V3.



```
function init() external onlyGovernance {
    int24 _tickSpacing = tickSpacing;
    int24 baseThreshold = _tickSpacing * getBaseThreshold();
    (, int24 currentTick, ) = pool.getSqrtRatioX96AndTick();

    int24 tickFloor = UniswapLiquidityManagement.floor(
        currentTick,
        _tickSpacing
    );

    ticksData.baseTickLower = tickFloor - baseThreshold;
    ticksData.baseTickUpper = tickFloor + baseThreshold;

    UniswapLiquidityManagement.checkRange(
        ticksData.baseTickLower,
        ticksData.baseTickUpper
    );
}
```

Remedy:
There should be an onlyOnce modifier or a variable handling (as locks) that ensures init is never called again.

Status:

Acknowledged

2. Pulled liquidity rationale

Description:
Considering the scenario where the vault has pulled Liquidity with the intention of depositing it back to Uniswap V3 when the pool is relatively less volatile; the smart contract code assumes a rational behavior to override *checkDeviation* modifier by manually modifying ticks through strategy using *onlyGovernance* functions. But the code does not guarantee any logic to push liquidity back to v3 in a safe manner

Remedy:
Debatable.

Status:

Pending

Medium risk issues

1. Unoptimized user liquidity is held in a vault.

Description:
In Active vaults if a pool is created on 1-X ratio on Uniswap V3 and a user makes a deposit with 1-1 ratio through the vault, the vault is found to hold the remaining amount of the token initially at the price of X, giving users a complete share with proportion of the deposited amount while the remaining amount of user sits inactive within the vault.

Remedy:
The vault must ensure that the amounts provided by a user are equal to the amounts of tokens actually deposited on a Uniswap pool.

Status:

Acknowledged

2. Deposit should have non reentrant checks placed in both vaults

Description:
The *deposit()* function in both vaults contract, that is, the *UnipilotActiveVault* and the *UnipilotPassiveVault* does not contain a non-reentrant modifier which is a standard practice to prevent any adversarial intent related to the reentrancy exploits.

Remedy:
A good industry practice requires that the *deposit()* function executes with a non-reentrant modifier; this modifier should be placed to ensure security as the deposit marks external calls through linked libraries to deposit values to the Uniswap V3.

Status:

Fixed as per BlockApex recommendation.

Low risk issue

1. Zero Address checks not placed in contract constructors()

Description:
Constructor() does not contain checks for accepting params of address type whether an address is zero or not.

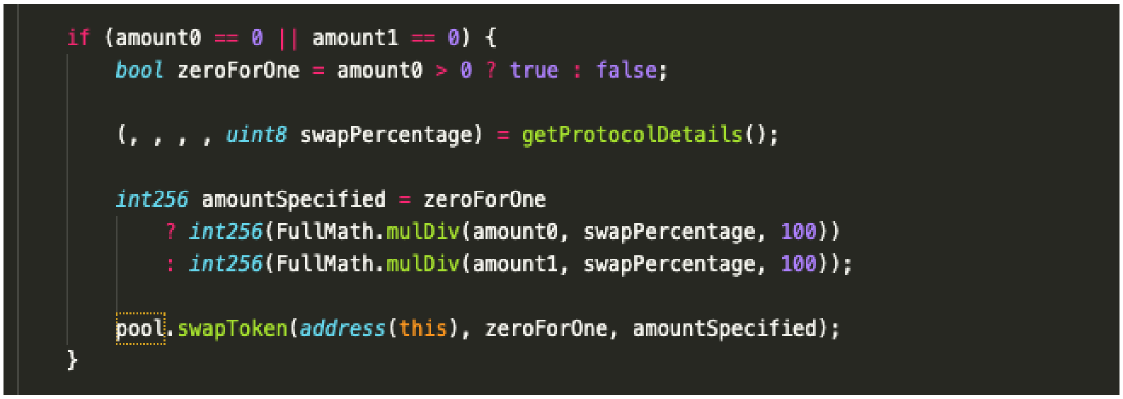
Remedy:
Since the constructor accepts an address from an argument, there should be a zero address check to ensure the functionality. These checks should be placed in almost every contract: Unipilot Factory , Unipilot Strategy , Unipilot Migration etc.

Status:

Acknowledged

2. Safecast in UnipilotPassiveVault.sol line 241 & 242 for `swapPercentage`

Description:
In UnipilotPassiveVault.sol the *readjustLiquidity()* reads the *swapPercentage* variable in Line 238 of the contract to calculate the amountSpecified variable in Lines 241-242, this Math is unsafe as the calculation is executed with different types for each param.



```
if (amount0 == 0 || amount1 == 0) {
    bool zeroForOne = amount0 > 0 ? true : false;

    (, , , , uint8 swapPercentage) = getProtocolDetails();

    int256 amountSpecified = zeroForOne
        ? int256(FullMath.mulDiv(amount0, swapPercentage, 100))
        : int256(FullMath.mulDiv(amount1, swapPercentage, 100));

    pool.swapToken(address(this), zeroForOne, amountSpecified);
}
```

Remedy:

Use safecasting for all type variables on lines 232-233 to ensure a seamless execution of the desired arithmetics.

Status:

Acknowledged

3. Storage Layout is unoptimized.

Description:

Variable tight packing is strongly recommended for both vaults and factories in state variable declaration as the contracts are composed in order that is gas-consuming.

Remedy:

A solidity design pattern 'Tight variable Packing' ensures that the smart contract is optimized to execute efficiently within the EVM environment.

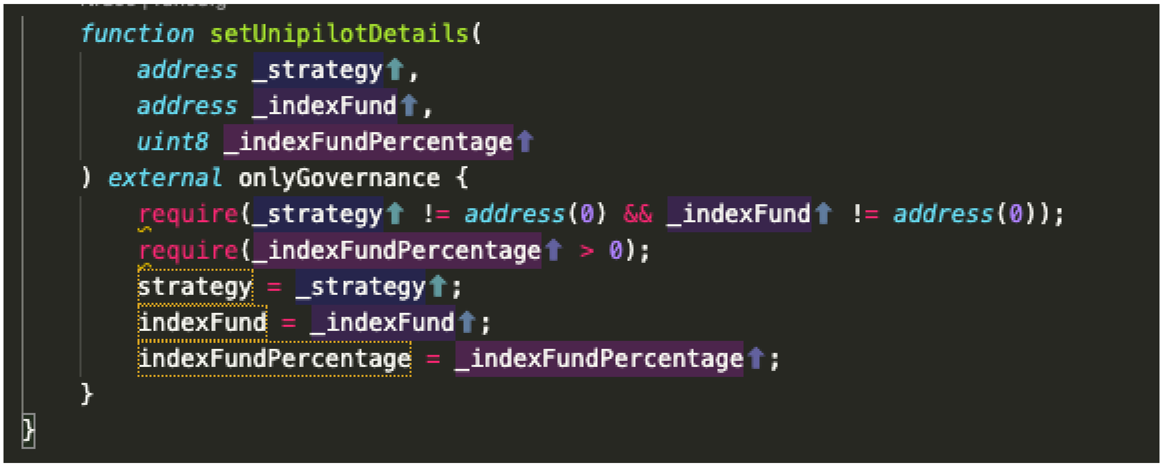
Status:

Acknowledged

4. Check max cap for indexFundPercentage and swapPercentage.

Description:

In *setUnipilotDetails()* the param indexFundPercentage is checked to receive a lowest value greater than zero.



```
    function setUnipilotDetails(
        address _strategy,
        address _indexFund,
        uint8 _indexFundPercentage
    ) external onlyGovernance {
        require(_strategy != address(0) && _indexFund != address(0));
        require(_indexFundPercentage > 0);
        strategy = _strategy;
        indexFund = _indexFund;
        indexFundPercentage = _indexFundPercentage;
    }
}
```

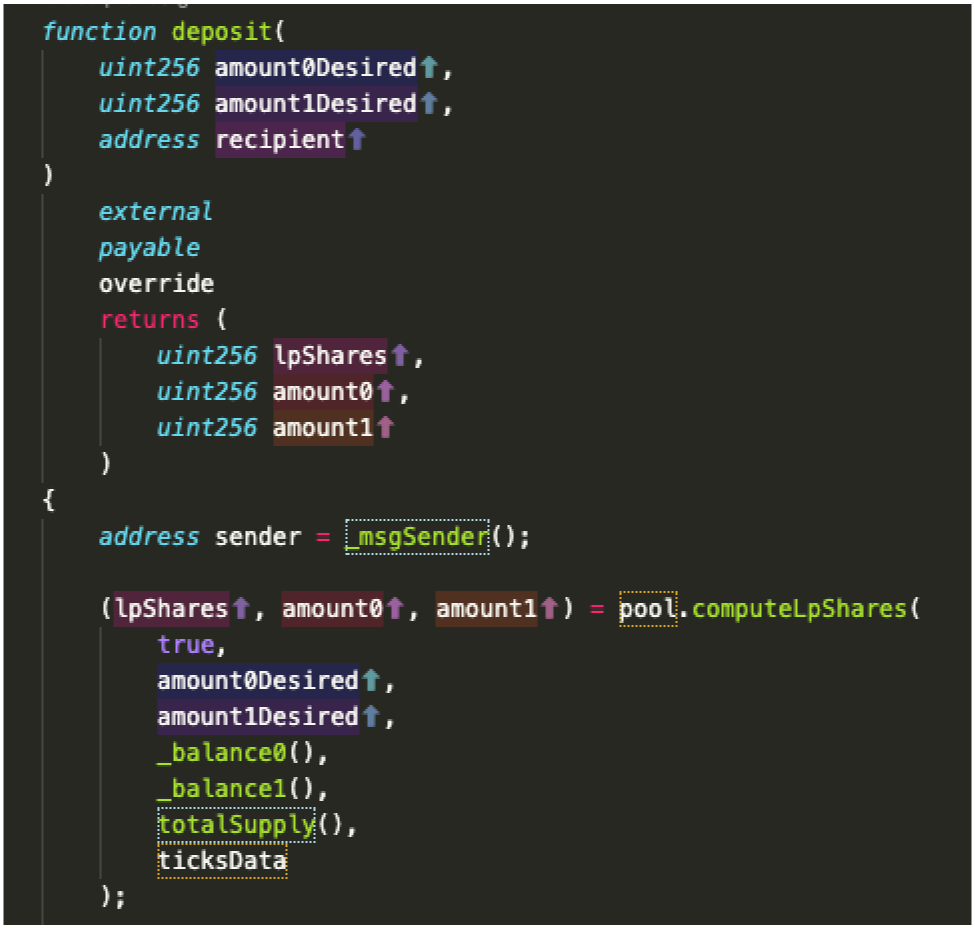
Remedy:
Ensure a check placed to bound the maximum value for the *indexFundPercentage*

Status:

Fixed

5. Deposit() function optimization.

Description:
In *UnipilotActiveVault.sol* and *UnipilotPassiveVault.sol*, Users can call *deposit()* with zero amounts of both tokens and the function executes until the end.



```
function deposit(
    uint256 amount0Desired,
    uint256 amount1Desired,
    address recipient
)
    external
    payable
    override
    returns (
        uint256 lpShares,
        uint256 amount0,
        uint256 amount1
    )
{
    address sender = _msgSender();

    (lpShares, amount0, amount1) = pool.computeLpShares(
        true,
        amount0Desired,
        amount1Desired,
        _balance0(),
        _balance1(),
        totalSupply(),
        ticksData
    );
```

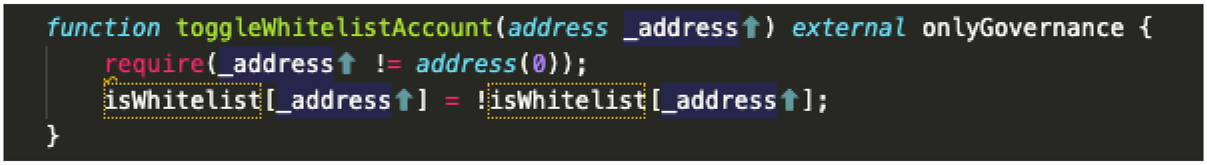
Remedy:
Function should check for zero value for both input args in the *deposit()* function in vaults contract.

Status:

Fixed

6. toggleWhitelistAccount() redundant.

Description:
toggleWhitelistAccount() can toggle the gov off in a redundant call of the same function to whitelist itself back.



```
function toggleWhitelistAccount(address _address) external onlyGovernance {
    require(_address != address(0));
    isWhitelist[_address] = !isWhitelist[_address];
}
```

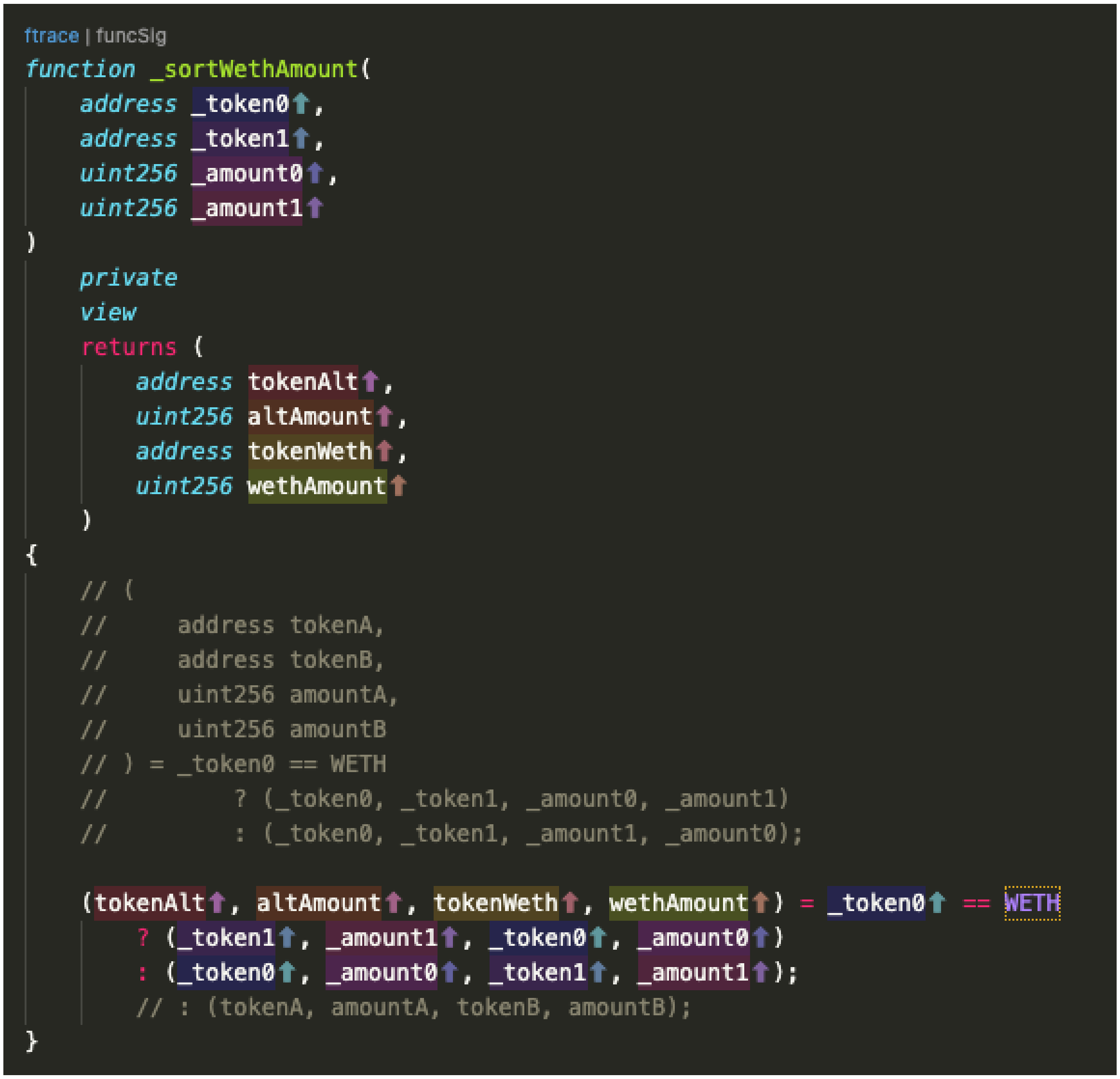
Remedy:

Ensure the address is checked to not allow governance to be toggled for whitelist.

Status:

Pending

7. _SortWeth() optimization



```
ftrace | funcSig
function _sortWethAmount(
    address _token0,
    address _token1,
    uint256 _amount0,
    uint256 _amount1
)
    private
    view
    returns (
        address tokenAlt,
        uint256 altAmount,
        address tokenWeth,
        uint256 wethAmount
    )
{
    // (
    //     address tokenA,
    //     address tokenB,
    //     uint256 amountA,
    //     uint256 amountB
    // ) = _token0 == WETH
    //         ? (_token0, _token1, _amount0, _amount1)
    //         : (_token0, _token1, _amount1, _amount0);

    (tokenAlt, altAmount, tokenWeth, wethAmount) = _token0 == WETH
        ? (_token1, _amount1, _token0, _amount0)
        : (_token0, _amount0, _token1, _amount1);
        // : (tokenA, amountA, tokenB, amountB);
}
```

Description:
This function's logic can be concise. The remedy, tested against the required logic, is mentioned as a code snippet in the screenshot above.

Status:

Acknowledged

Informatory issues

1. No NatSpec documentation

Description:
NatSpec documentation is an essential part of smart contract readability; it is therefore advised that all contracts and following files contain proper explanatory commenting;

- UnipilotActiveVault.sol
- UnipilotPassiveVault.sol
- UnipilotMigrator.sol

Status:
Acknowledged

2. Mark token0 and token1 as immutable in UnipilotActivevault.sol and UnipilotPassiveVault.sol

Description:
State variables containing the address of tokens should be marked as immutable as the constructor locks the values for each after deployment.

Status:
Acknowledged

3. onlyGovernance() modifier in passive vault contract

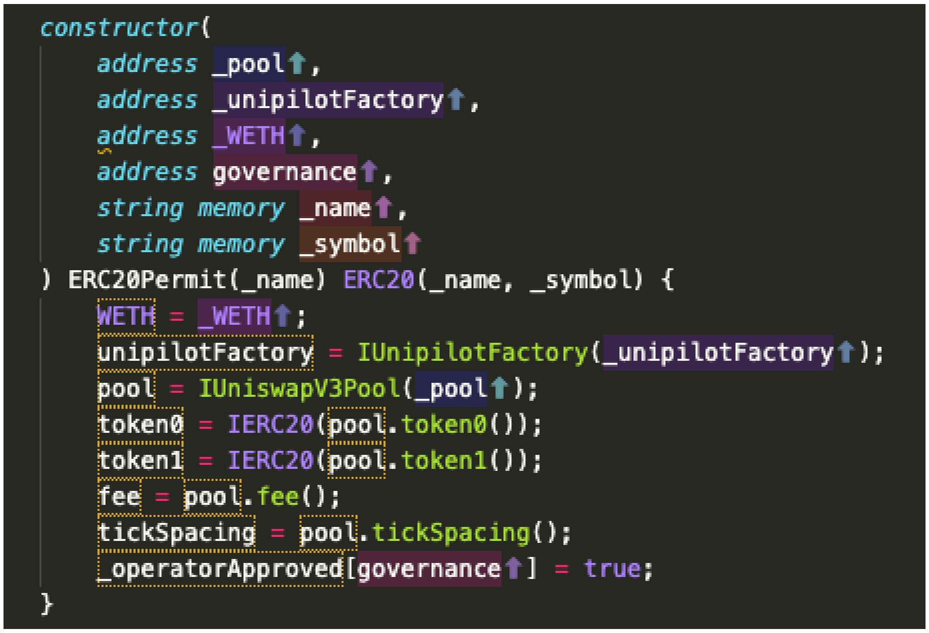
Description:
The onlyGovernance modifier in the Passive Vault contract remains unused within the contract.

Status:
Pending

4. _WETH address should be hardcoded before production wherever necessary

Description:

Address of the *WETH* token contract is passed as a constructor param in both Factories which can be optimized by hardcoding the actual address of *_WETH* in the final deployment of the production environment.



```
constructor(
    address _pool,
    address _unipilotFactory,
    address _WETH,
    address governance,
    string memory _name,
    string memory _symbol
) ERC20Permit(_name) ERC20(_name, _symbol) {
    WETH = _WETH;
    unipilotFactory = IUnipilotFactory(_unipilotFactory);
    pool = IUniswapV3Pool(_pool);
    token0 = IERC20(pool.token0());
    token1 = IERC20(pool.token1());
    fee = pool.fee();
    tickSpacing = pool.tickSpacing();
    _operatorApproved[governance] = true;
}
```

Status:

Acknowledged

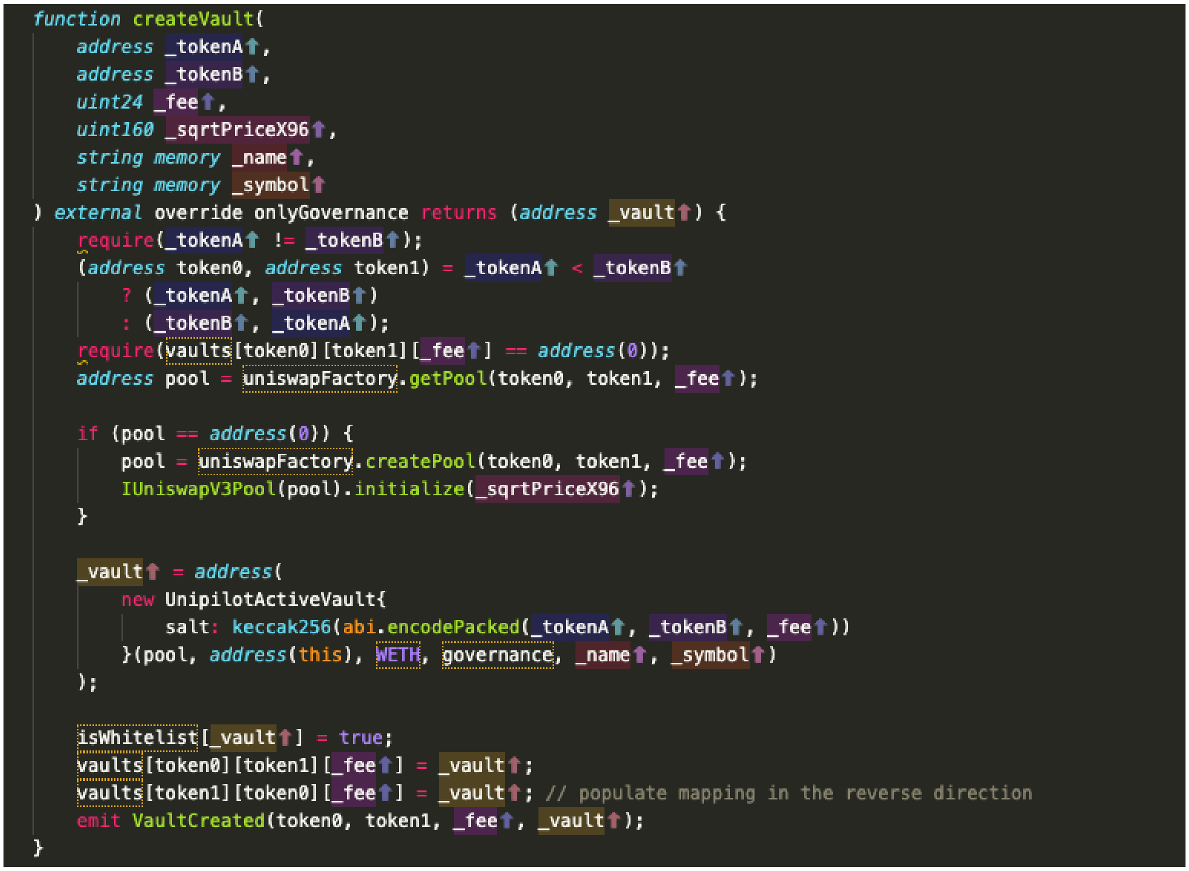
5. Factory function createVault() optimized version

Description:

createVault() is found to be optimized if it executes in the following recommended pattern:

- First check pool on v3 - if returns true then check vault exists - if returns true then return from function
- If pool returns false - create & initialize pool then create vault

Current Implementation:



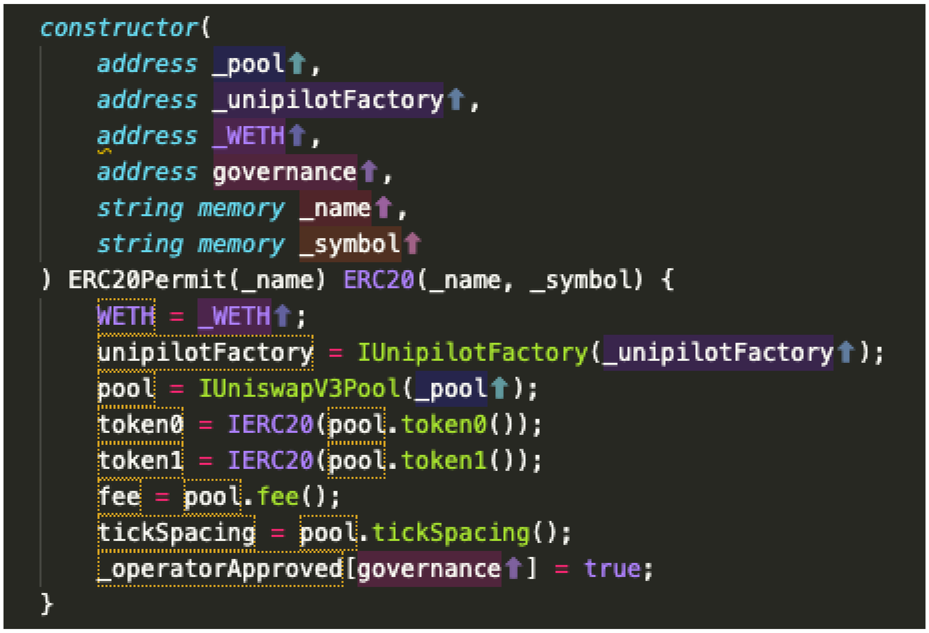
```
function createVault(
    address _tokenA,
    address _tokenB,
    uint24 _fee,
    uint160 _sqrtPriceX96,
    string memory _name,
    string memory _symbol
) external override onlyGovernance returns (address _vault) {
    require(_tokenA != _tokenB);
    (address token0, address token1) = _tokenA < _tokenB
        ? (_tokenA, _tokenB)
        : (_tokenB, _tokenA);
    require(vaults[token0][token1][_fee] == address(0));
    address pool = uniswapFactory.getPool(token0, token1, _fee);

    if (pool == address(0)) {
        pool = uniswapFactory.createPool(token0, token1, _fee);
        IUniswapV3Pool(pool).initialize(_sqrtPriceX96);
    }

    _vault = address(
        new UnipilotActiveVault{
            salt: keccak256(abi.encodePacked(_tokenA, _tokenB, _fee))
        }(pool, address(this), WETH, governance, _name, _symbol)
    );

    isWhitelist[_vault] = true;
    vaults[token0][token1][_fee] = _vault;
    vaults[token1][token0][_fee] = _vault; // populate mapping in the reverse direction
    emit VaultCreated(token0, token1, _fee, _vault);
}
```

Status:

Acknowledged

6. Assignment of Params in order to receive the function signature.

Description:
In all four contracts of vault and factory the constructor receives arguments in order which is out-of-sync to the one being assigned, reducing the code readability. Ensure param values and actual assignments are in sync for better code readability.



```
constructor(
    address _pool,
    address _unipilotFactory,
    address _WETH,
    address governance,
    string memory _name,
    string memory _symbol
) ERC20Permit(_name) ERC20(_name, _symbol) {
    WETH = _WETH;
    unipilotFactory = IUnipilotFactory(_unipilotFactory);
    pool = IUniswapV3Pool(_pool);
    token0 = IERC20(pool.token0());
    token1 = IERC20(pool.token1());
    fee = pool.fee();
    tickSpacing = pool.tickSpacing();
    _operatorApproved[governance] = true;
}
```

Status:
Acknowledged

7. Order of functions as in solidity Style Guide

Description:
Receive() and *Fallback()* should be moved on top, below constructor; following the solidity design patterns

Status:
Fixed

8. uint256 can be cheaper than uint8

Description:
Uint8 is proved to be more costly than *uint256* variables in a number of scenarios, where a better and optimized variable packing for *uint8* variables is recommended or replaced with *uint256/ uint64/ uint24* type vars.

Status:
Fixed

9. pullLiquidity() is vulnerable in its current execution

Description:
The pullLiquidity(address _recipient) method is vulnerable to some extent, holding potential for mal-intent or permanent loss of value. Checking for the address argument as not another whitelisted vault can ensure no accidental and permanent loss of tokens happen.

Status:
Pending

10. Spelling mistakes in function signatures

Description:
In the UnipilotMigrator.sol file,
migrateUnipilotLiquididty() and *_refundRemainingLiquidiy()* are spelled wrong, causing readability issues as well as creating the wrong function signature.

Status:
Fixed

11. Mark private functions as internal

Description:
In the UnipilotMigrator.sol file,
_sortWethAmount() and *_addLiquidityUnipilot()* are private, which are gas costly.

Status:

Acknowledged

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices till date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token, its sale or any other aspect of the project.

Crypto assets/tokens are results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third-party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks.

This audit cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.