BlockApex

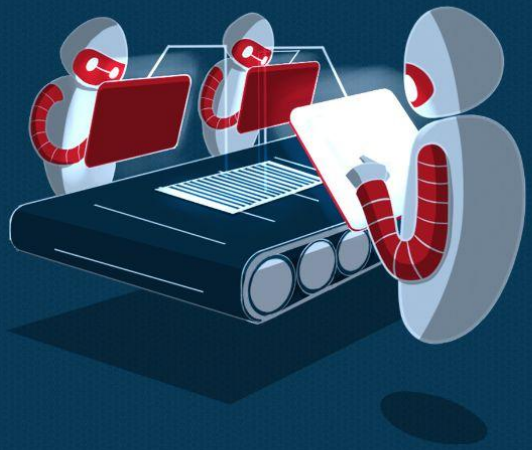
SMART CONTRACT SECURITY ANALYSIS REPORT

```
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 any 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 the 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; at the discretion of the client.

Key understandings

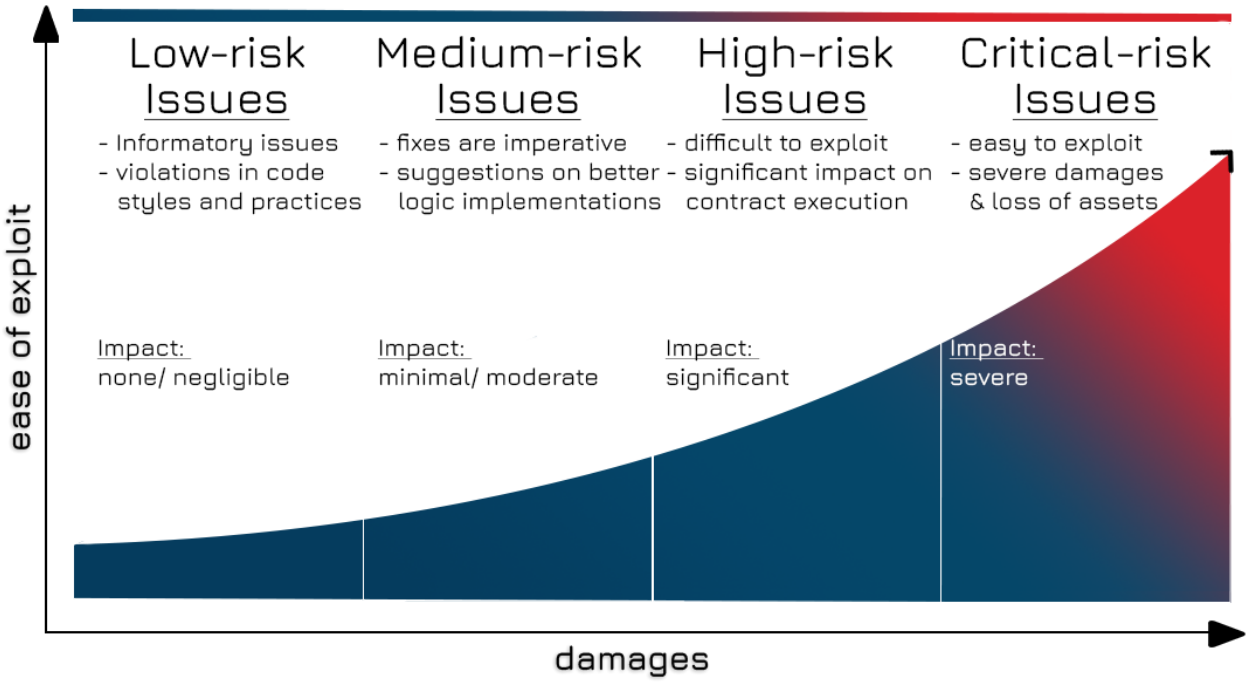


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by Unipilot (Client) for the purpose of conducting a Smart Contract Audit/ Code Review. This document presents the findings of our analysis which started on 4th Jan '2023

Name
Unipilot
Auditor
BlockApex
Platform
Arbitrum (L2) | Solidity
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review | Functional Testing | Computer-Aided Verification | Manual Review
Git repository | Commit Hash
Public Repo | 641dbff849fa2efdbc0a1cbad4155535eae066ab
White paper/ Documentation
https://unipilot.gitbook.io/unipilot/
Document log
Initial Audit Completed: 8th Jan 2023
Final Audit Completed: 11th Jan 2023

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	Fungible token violations	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 the future, the protocol will support other decentralized exchanges like Sushiswap (Trident). The architecture is designed to keep in mind future releases.

The protocol has 6 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 in the 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 in the 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 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 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.

UnipilotMigrator.sol

The smart contract aids to migrate users' liquidity from other Uniswap V3 Liquidity Optimizer Protocols to Unipilot V2 Protocol.

Methodology & Scope

The codebase was audited using a filtered audit technique. A pair of auditors scanned the codebase in an iterative process for one (1) week.

Starting with the recon phase, a basic understanding was developed and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews with the motive to find logical flaws in the codebase complemented with code optimizations, software, and security design patterns, code styles, best practices, and identifying false positives that were detected by automated analysis tools.

Privileged Roles

In a production environment, the unipilot protocol sets the address for governance that exercises a privileged position over the factory and vault contracts in the system. The governor can transfer the governance 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

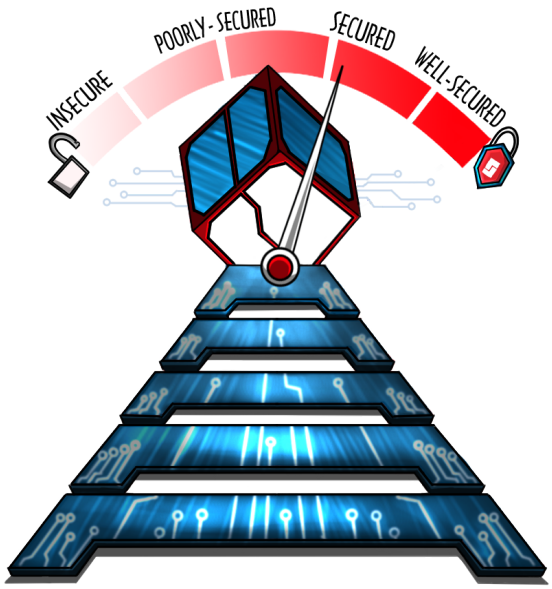
The operator address has following activities it can be used for

- 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

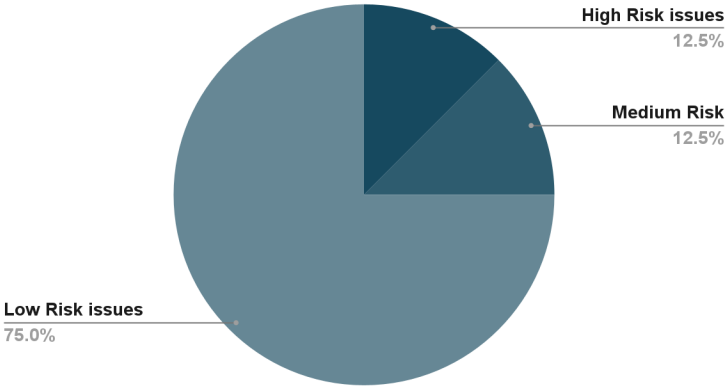
Our team performed a technique called "Filtered Audit", where the contract was separately audited by four individuals. After a thorough and rigorous process of manual testing, an automated review was carried out using Slither & Mythril for static analysis and Foundry for fuzzing invariants. All the flags raised were manually reviewed and re-tested to identify the false positives.



Our team found:

#Issues	Severity Level
0	Critical Risk issue(s)
1	High Risk issue(s)
1	Medium Risk issue(s)
6	Low Risk issue(s)
0	Informatory issue(s)

Proportion of Vulnerabilities



Key Findings

#	Findings	Risk	Status
1	Pulled liquidity rationale	High	Acknowledged
2	Unoptimized user liquidity is held in a vault	Medium	Acknowledged
3	Insufficient data validation	Low	Fixed
4	Missing Safecast	Low	Fixed
5	_sortWethAmount() optimization	Low	Acknowledged
6	Mark token0 and token1 as immutable	Low	Partially Fixed
7	createVault() optimization	Low	Fixed
8	Assignment of params in order	Low	Fixed

Detailed Overview

High-risk issues

ID	1
Title	Pulled liquidity rationale
Path	contracts/UnipilotActiveVault.sol, contracts/UnipilotPassiveVault.sol
Function Name	pullLiquidity

Description: In both vault contracts, the function pullLiquidity has potential use cases which bring it into the spotlight. 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

Recommendation: Remove the above-mentioned function due to no safe measure of a complete cycle first to pull and then push liquidity.

Status: Acknowledged by Client and Auditor.

Developer's Response: Governance will manually update deviation in case of price volatility.

Auditor's Response: Acknowledged with a disclaimer to ensure that the vulnerability is contained by only executing the pullLiquidity function in none other than an authorized manner.

Medium-risk issues

ID	2
Title	Unoptimized user liquidity is held in a vault
Path	contracts/UnipilotActiveVault.sol
Function Name	deposit

Description: In Active vaults, if a pool is created on a 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 the proportion of the deposited amount while the remaining amount of user sits inactive within the vault.

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

Status: Acknowledged by Client and Auditor.

Developer's Response:

"We make the user's share with 3 elements:

- Unipilot position liquidity amounts in terms of the token (0,1)
- Unipilot position uncompounded fees in terms of the token (0,1)
- Unused amounts in the vault

So during shares calculation if any amount is present in the vault it will make new share according to the ratio present in the vault thus accumulate in the vault again.

We are doing this because we have a method called pullLiquidity which burns Unipilot's position and collects amounts in the contract in case the market is volatile so the contract will HODL (position liquidity cannot be sent to external it can be only held in the contract).

So in this case, if a new user comes in during the HODL period our formula will create the shares with all the amounts which are in the contract because the Uniswap position is empty so this way we mint the correct amounts.
This way the contract can mint correct shares during the HODL period if we neglect the contract balance then the user's shares will be messed up.

Remedy: Contract balance will become 0 after execution of readjustLiquidity so new user's liquidity will not be held in the vault next time."

Auditor's Response: Acknowledged with the disclaimer that to contain this ill behavior the Unipilot's governance and the operator must ensure responsible usage of the functions pullLiquidity and readjustLiquidity, such that an uninterrupted flow of contract is guaranteed.

Low-risk issues

ID	3
Title	Insufficient data validation
Path	contracts/**.sol
Function Name	constructors

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

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

Status: Fixed.

Developer's Response: We can not add many require statements because of the contract size issues.

Auditor's Response: Acknowledged. No follow-up is required.

ID	4
Title	Missing safecast
Path	contracts/UnipilotPassiveVault.sol
Function Name	readjustLiquidity

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);
}
```

Recommendation: Use safe casting for all type variables on lines 232-233 to ensure seamless execution of the desired arithmetics.

Status: Fixed.

ID	5
Title	_sortWethAmount optimization
Path	contracts/UnipilotMigrator.sol
Function Name	_sortWethAmount

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

```
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);
}
```

Recommendation: Same as in the description above.

Status: Acknowledged by Client and Auditor.

ID	6
Title	Mark token0 and token1 as immutable
Path	contracts/UnipilotActivevault.sol, contracts/UnipilotPassiveVault.sol
Function Name	constructors

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

Recommendation: Same as in the description above.

Status: Partial Fixed.

Developer's response: PassiveVault uses immutables however active vaults don't due to contract size issues.

Auditor's Response: Acknowledged. No follow-up is required.

ID	7
Title	createVault() optimization
Path	contracts/UnipilotActiveFactory.sol, contracts/UnipilotPassiveFactory.sol
Function Name	createVault

Description: The function createVault() in both factory contracts can be optimized if it executes in the following recommended pattern.

- Check,
  - If the pool exists on UniswapV3,
    - Also, if the UnipilotVault exists,
      - Revert from the function
- Since,
  - The pool does not exist on UniswapV3;
    - Create & initialize pool then create vault.

Code:

Current Flow

```
require(_tokenA != _tokenB);
(address token0, address token1) = _tokenA < _tokenB
    ? (_tokenA, _tokenB)
    : (_tokenB, _tokenA);
require(vaults[token0][token1][_fee][_vaultStrategy] == address(0));
address pool = uniswapFactory.getPool(token0, token1, _fee);

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

Suggested Flow

```
require(_tokenA != _tokenB);
```

```
        (address token0, address token1) = _tokenA < _tokenB
            ? (_tokenA, _tokenB)
            : (_tokenB, _tokenA);

        address pool = uniswapFactory.getPool(token0, token1, _fee);

        if(pool != address(0)) {
            require(vaults[token0][token1][_fee][0] == address(0));
        }

        else {
            pool = uniswapFactory.createPool(token0, token1, _fee);
            IUniswapV3Pool(pool).initialize(_sqrtPriceX96);
        }
        // continue for passive OR active...
```

Recommendation: Same as in the description above.

Status: Fixed.

ID	8
Title	Assignment of params in order
Path	contracts/GuildHub.sol
Function(s)	claimBacKTokens | unLend

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;
}
```

Recommendation: Same as in the description above.

Status: Fixed.

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 the 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 another 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 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 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 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 the security of smart contracts.