BlockApex

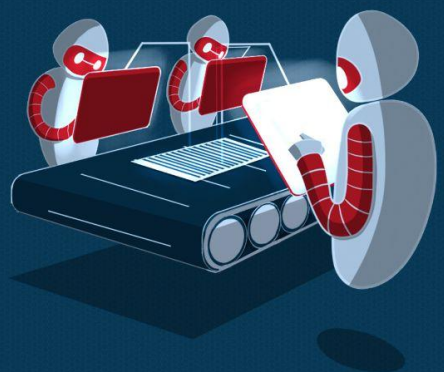
SMART CONTRACT SECURITY ANALYSIS REPORT

```
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

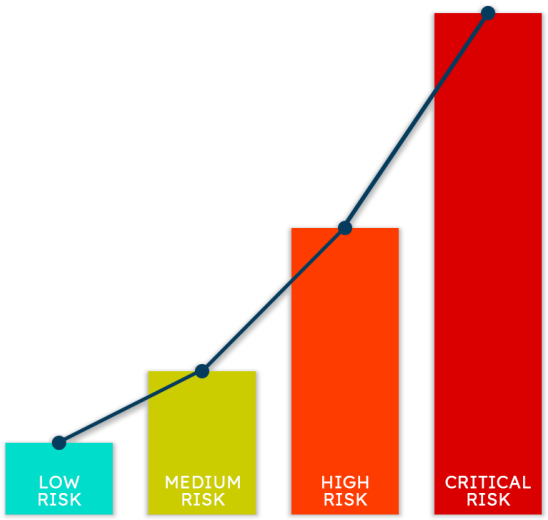
PREFACE

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 ××××	Critical vulnerabilities are too easy to exploit and can lead to damages/loss in assets or manipulations.
HIGH RISK ×××	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution.
MEDIUM RISK ××	Medium-level vulnerabilities are equally imperative to fix but they tend to have minimal impact on asset loss or data manipulations.
LOW RISK ×	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 took place from _19th August 2021_.

Name
Unipilot
Auditor
Moazzam Arif | Kaif Ahmed
Platform
Ethereum/Solidity
Type of review
Tokenomics, Liquidity Management, Index Fund
Methods
Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
Git repository
https://github.com/VoirStudio/unipilot-protocol-contract-v2/tree/update-structure
White paper/ Documentation
https://unipilot.medium.com/
Document log
Initial audit: 19th August 2021
Final audit: 11th October 2021

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 auto-liquidity management protocol built on top of Uniswap v3. It simplifies the liquidity management by rebasing the liquidity of the pool, when prices get out of range.

To readjust the liquidity of the pool, ***Captains*** (independent nodes) are compensated for the gas fee plus some bonus in $PILOT.

It also has a governing token i.e., $PILOT. When a protocol earns a fee from Uniswap v3, users have the option to claim a fee in equivalent $PILOT (price is fetched from $FEE_TOKEN/$WETH -> $PILOT/$WETH price oracles).

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). So the architecture is designed to keep in mind the future releases.

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

Unipilot.sol: The smart contact is the entry point in the protocol. It allows users to deposit, withdraw and collect fees on liquidity. It mints an NFT to it's users representing their individual shares.

V3Oracle.sol: It is a wrapper around Uniswap oracles. It also has helper functions to calculate TWAP (time-weighted average price) and other prices relevant data.

UniswapLiquidityManager.sol: The heart of the protocol that allows users to add, withdraw, collectFee and readjust the liquidity on Uniswap v3. This smart contract interacts with Uniswap v3Pool and v3Factory to add and remove liquidity. It maintains 2 positions on Uniswap i.e., base position and range position (left over tokens are added as range orders).

UniStrategy.sol: The smart contract to fetch and process ticks' data from Uniswap. It also decides the bandwidth of the ticks to supply liquidity (on the basis of governance).

ULMState.sol: This smart contract fetches the updated prices and tick ranges from Uniswap's v3 pools.

Steps to Add Liquidity:

1. User will give approval of both tokens to **Unipilot.sol**
2. User will call the deposit method of **Unipilot.sol**
3. **Unipilot.sol's deposit** method will call the deposit method of **UniswapLiquidityManager.sol** and transfer both the tokens to **UniswapLiquidityManager.sol**
4. **UniswapLiquditymanager.sol** will add the liquidity on Uniswap. This will:
    a. Mint an NFT denoting the user's shares in the liquidity provided
    b. Alter the existing liquidity by increasing the user's share

Methodology & Scope

Audit log

In the first week, we developed a deeper understanding of the protocol and its workings. We started by reviewing the five main contracts against common solidity flaws and did a static analysis using ***Slither***. In the second week, we wrote unit-test cases to ensure that the functions are performing their intended behaviour. In the 3rd week, we began with the line-by-line manual code review.

Note:

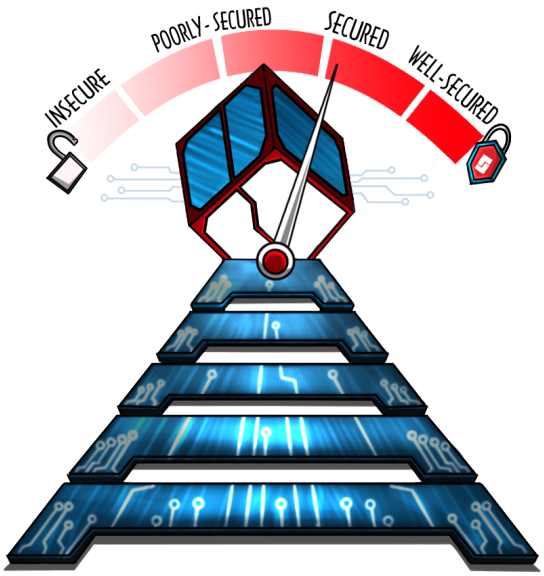
We haven't fuzzed the end-to-end smart contracts behaviour in this auditing round. So this is the version of the report with the issues that were found in all the steps we performed before the above mentioned date. We are now engaged again by Unipilot to further test and fuzz properties. A detailed analysis will be shared later in an updated version of the report which will be published later.

AUDIT REPORT

Executive Summary

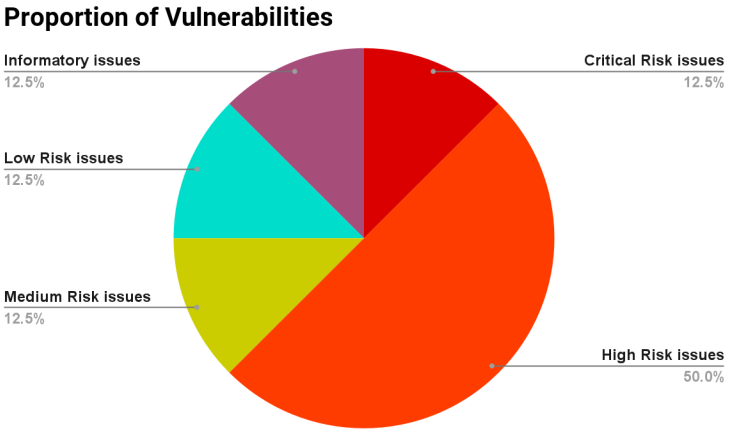
The analysis indicates that the contracts audited are **secure**.

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 Echidna 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)
4	High Risk issue(s)
1	Medium Risk issue(s)
1	Low Risk issue(s)
1	Informatory issue(s)



Findings

Critical-risk issues

1. **Attacker can steal the liquidity reward fees earned by other users**

   File: UniswapLiqudityManger.sol
   Description:

```
   function distributeFeesAndLiquidity(DistributeFeesParams memory
params)
       public
       returns (
      ...
      )
```

   Exploit Scenario:
   Any malicious user can call the ***distributeFeesAndLiquidity,*** providing valid params (i.e., NFT id) and add his address as recipient to claim fees.
   Remedy:
   Change the visibility or add proper access controls to restrict user permissions.

   Status:
   Fixed by changing the visibility to ***private***.

High-risk issues

1. **A function in PeripheryPayments.sol allows anyone to withdraw contract balance of ether/tokens**

   Description:

```
      function unwrapWETH9(uint256 amountMinimum, address recipient)
public {
...
}
```

   Remedy:
   Change the visibility of functions.

   Status:
   Fixed.

2. **Users can update their shares without actually adding the liquidity in the pool**

   File: UniswapLiquidityManager.sol
   Description:
   Users can call the ***deposit*** function directly from the Liquidity Manager contract and change their shares whereas the shares need to be calculated in **Unipilot.sol**

   This function should only be callable from **Unipilot.sol**

```
function deposit(
    address token0,
    address token1,
    address sender,
    uint256 amount0Desired,
    uint256 amount1Desired,
    uint256 shares,
    bytes memory data
) external payable override returns (uint256 mintedTokenId) {}
```

   Remedy:
   Add proper access controls using modifiers.

   Status:
   Fixed, using ***onlyUnipilot*** modifier.

3. **Users can earn unfair fees**

Description:
There is no check if the tokenId can be used for pool addresses. Users can use the same tokenId in different pools to manipulate the ***feeshare***. Consider the following exploit scenario, when a user can use a token minted for pool1, in pool2.

Attack Scenario:

1. A user deposits in the pool where ***feeGrowthGlobal0*** and ***feeGrowthGlobal1*** is low and mints the NFT (tokenId)
2. The user now deposits it in the pool where ***feeGrowthGlobal0*** and ***feeGrowthGlobal1*** is high using the previous tokenId
3. Now the user has more shares of fees earned

Consider the following code snippet:

```
userPosition.tokensOwed0 += FullMath.mulDiv(

                poolPosition.feeGrowthGlobal0(position of pool 2) -
userPosition.feeGrowth0 (position in pool 1),

                userPosition.liquidity,

                1e18

            );
```

Remedy:
Use the poolId from positions struct instead of getting from params.

Status:
Fixed, by using the ***addressesToNFTId*** mapping which stores ***tokenId*** and ***poolAddress*** mapping.

4. Reentrancy in withdrawls

Description:
Reentrancy is possible with ERC777 tokens. Look at the following code snippet. ***_distributeFeesInTokens*** is called before updating the ***userPosition***.

```
 if (
                params.pilotToken &&
                ethToken0Pair != address(0) &&
                ethToken1Pair != address(0)
            ) {
                (indexAmount0, indexAmount1) =
_distributeFeesInPilot(
                    params.recipient,
                    a.token0,
                    a.token1,
                    userPosition.tokensOwed0,
                    userPosition.tokensOwed1
                );
            } else {
                (indexAmount0, indexAmount1) =
_distributeFeesInTokens(
                    params.wethToken,
                    params.recipient,
                    a.token0,
                    a.token1,
```

```
                    userPosition.tokensOwed0,

                    userPosition.tokensOwed1

                );

            }

        }

        transferLiquidity(

        ...

        );

        (userAmount0, userAmount1) = (userPosition.tokensOwed0,
userPosition.tokensOwed1);

        position.fees0 =
position.fees0.sub(userPosition.tokensOwed0);

        position.fees1 =
position.fees1.sub(userPosition.tokensOwed1);

        userPosition.tokensOwed0 = 0;

        userPosition.tokensOwed1 = 0;

        userPosition.liquidity =
userPosition.liquidity.sub(params.liquidity);
```

Exploit Scenario:
A malicious user will create an ERC777/WETH pair. When he calls the withdraw function, in ***tokenReceived*** callback of smart contract (ERC777 calls the tokensReceived callback), the user can re-enter.
Remedy:
Add the ***nonReentrant*** modifier from OpenZeppelin.

Status:
Fixed, by adding the ***nonReentrant*** modifier.

Medium-risk issues

1. **poolPosition.feeGrowthGlobal0/1 are out of sync with v3Pool**

   Description:
   When a user deposits into Unipilot, user ***positions.feeFrowthGlobal0/1*** is set equal to the ***poolPosition.feeGrowthGlobal0/1***. This is used to ensure that if a user adds liquidity at a certain point when the protocol has earned X amount of fee reward. This X amount should be subtracted at the time when the user collects his fee reward.

```
            Position storage userPosition = positions[tokenId];

            userPosition.tokensOwed0 += FullMath.mulDiv(

                poolPosition.feeGrowthGlobal0 -
userPosition.feeGrowth0,

                userPosition.liquidity,

                1e18

            );
```

   Exploit Scenario:

   1. User1 adds liquidity via Unipilot
   2. Swap occurs, and the fee earned by Unipilot should be entitled to user1
   3. User2 adds liquidity
   4. User2 will earn an instant fee reward, as the ***poolPosition.feeGrowthGlobal0*** is not updated

```
            positions[mintedTokenId] = Position({
                nonce: 0,
                pool: pool,
                liquidity: shares,
                feeGrowth0: poolPosition.feeGrowthGlobal0, // not
synced
                feeGrowth1: poolPosition.feeGrowthGlobal1,
                tokensOwed0: 0,
                tokensOwed1: 0
            });
```

Remedy:
Call the burn method (by burning 0 liquidity) of v3pool before updating the ***feeGrowthGlobal0*** variables.

Status:
Fixed, by calling ***_collectPositionFee*** which in turn calls the burn function.

Low-risk issues

1. **Users can call *readjust* simultaneously if the price is in range causing the pool to sit idle for the remaining time of the day**

   Description:
   The ***readjust*** function can be called at most 2 times a day (depending on governance). Malicious users can exhaust the readjustment call frequency sending the pool to sit idle when the price is out of range, undermining the very idea of the protocol.

   Remedy:
   Add a check if readjustment is needed or not.

   Status:
   Not fixed yet (but acknowledged).

Informatory issues and Optimization

1. **Add readjustment frequency to *UniStrategy.sol* instead of hardcoding it as 2 times/day**

   Description:
   Depending upon the volatility of the pair, readjustments might be needed more than twice each day.

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.