BlockApex

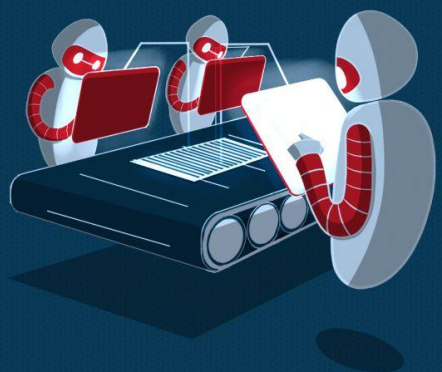
SMART CONTRACT SECURITY ANALYSIS REPORT

```
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

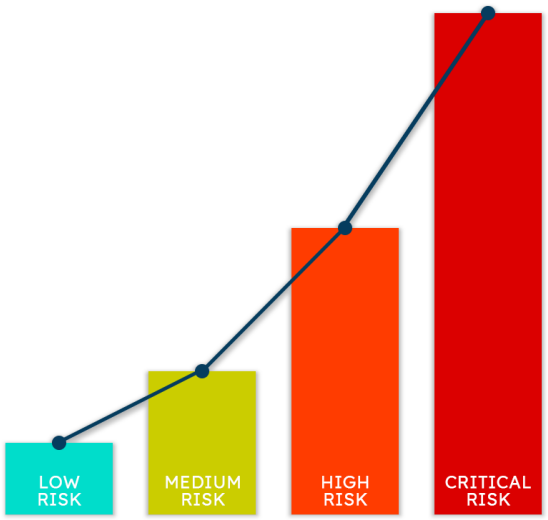
PREFACE

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 took place on _____.

Name
Unipilot
Auditor
Moazzam Arif | Kaif Ahmed | Muhammad Jarir uddinn

Platform
Ethereum/Solidity
Type of review
Maths and Economics
Methods
Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
Git repository
https://github.com/VoirStudio/unipilot-protocol-contract-v2/tree/update-getSharesAndAmounts
White paper/ Documentation
Document log

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

We were contacted again by Voirstudio to verify the Maths and economics of Unipilot. We fuzzed the smart contracts to test properties of the protocol and found some issues which are reported below.

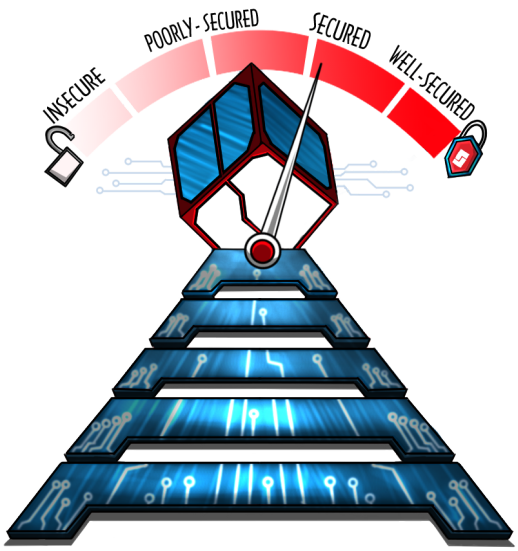
We also have tested and argued some of the points regarding the economics of the protocol. Also we discussed contingency plans around it.

Note: we worked closely with devs and the fixes were incrementally applied. The team was very supportive and was open to suggestions and discussion. They even pointed out

to us that part of the code where they feel something can go wrong. This helped a lot in auditing.

AUDIT REPORT

Executive Summary



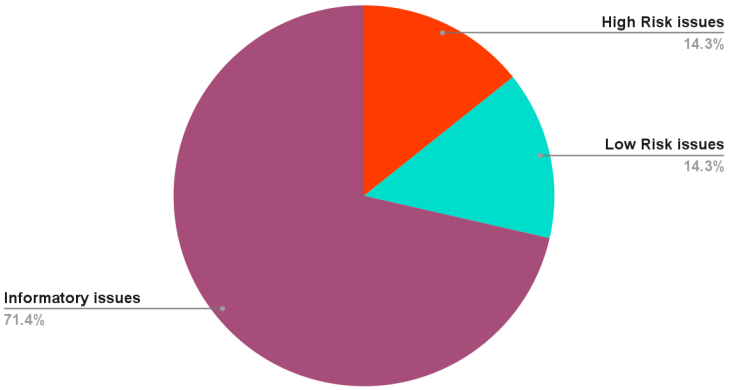
The analysis indicates that some of the functionalities in the contracts audited are **highly secured**.

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 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)
	High Risk issue(s)
4	Medium Risk issue(s)
	Low Risk issue(s)
2	Informatory issue(s)

Proportion of Vulnerabilities



Findings

Critical-risk issues

1. **Ether balance of contract can be withdrawn**

File: PeripheryPayments.sol

Description:

Unipilot conducted a testnet bug bounty. In that some users reported they were unable to collect their fees. By investigating that with the dev team, they founded that ERC20/WETH pool was misbehaving due to the following code snippet

```
if (balanceWETH9 > 0) {

IWETH9(WETH).withdraw(balanceWETH9); // contract balance is transferred

TransferHelper.safeTransferETH(recipient, balanceWETH9);}
```

Remedy: The dev team suggested to remove this method and pass proper amount to be withdrawn

Status: Fixed

High-risk issues

No issues were found

Medium-risk issues

1. **Maths Precision issues**

File: UniswapLiquidityManger.sol && ULMState.sol
Description
Unipilot uses 1e18 precision for calculating the Liquidity shares and feeGrowth when adding liquidity and fees. This causes some small amounts to be locked forever in the UniswapLiquiditymanager.
See the following code snippet

```
position.feeGrowthGlobal0 += FullMath.mulDiv(
    collect0Base + collect0Range,
    1e18, // precision
    position.totalLiquidity
);
```

Remedy: We suggest using FixedPoint128.Q128 from uniswap. We fuzzed against this precision and we got more precise results

Status: Fixed

2. **Economic Attacks**

   a. **Price oracle manipulation**
   File: UniswapLiquidityManger.sol

   Unipilot relies on TWAP when minting new unipilot either in fees or in readjustment of the pool. Tokens are priced wrt to ether and then from etherAmount the pilotAmount is calculated.
   Attack Scenario
   Consider users who want to get $PILOT in fees instead of underlying tokens. Users can create a pool with $WETH, do some swaps, and inflate the tokens

against $WETH. Unipilot will get the etherAmount of tokens from the pool and convert ethers to pilot. Hence the user gets the $PILOT at a manipulated price.

Note: To mitigate this attack, devs already have implemented whitelisting of pools when distributing fees. But they had not implemented whitelisting during readjustments

Remedy: Add whitelisting to readjustment

Status: Fixes in progress

b. Put a limit on tx.gasPrice

File: UniswapLiquidityManger.sol

Description

When a user readjust the pool, his transaction fees is compensated plus a Premium is given to the user in $PILOT.

```
 b.gasUsed = tx.gasprice.mul(initialGas.sub(gasleft()));

// tx.gasprice can be adjusted high enough to mint more $PILOT

b.pilotAmount = IOracle(_oracle).ethToAsset(PILOT, 3000,b.gasUsed);
```

Remedy: Add a limit to tx.gasprice

Status: Fixed

c. **Probable Slippage Attacks**
While readjusting the pool, the unipilot swap X amount of tokens to make the pool in range. The swap percent (how much should be swapped) and the slippage is hardcoded. This incentivizes the sandwich attacks
Remedy: The swap amount and slippage should be configurable according to favorable conditions
Status: Fixed

Low-risk issues

No issues were found

Informatory issues and Suggestions

Contract should be puasable if something goes wrong
Contracts should have rescue funds methods

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.