BlockApex

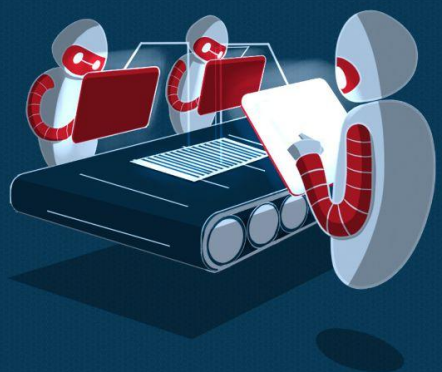
SMART CONTRACT SECURITY ANALYSIS REPORT

```
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

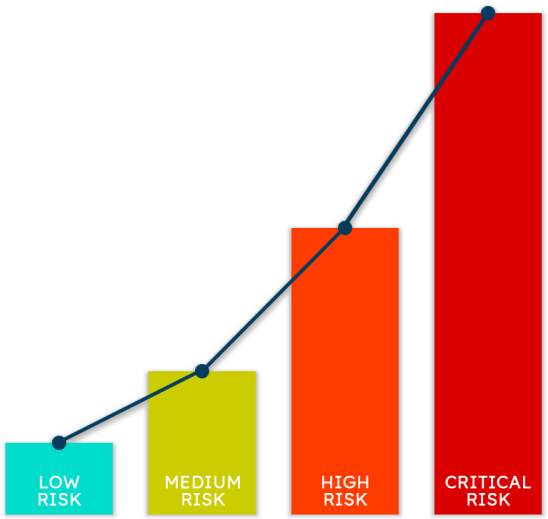
PREFACE

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

PREFACE **2**
- Objectives 2
- Key understandings 2

TABLE OF CONTENTS **3**

INTRODUCTION **4**
- Scope 5
- Project Overview 6
- System Architecture 6
- Methodology & Scope 6

AUDIT REPORT **7**
- Executive Summary 8
- Findings 9
  - Critical-risk issues 10
  - High-risk issues 13
  - Medium-risk issues 13
  - Low-risk issues 17
  - Informatory issues and Optimization 19

DISCLAIMER **20**

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 25th Feb 2022 .

Name
Unipilot-Farming-V2
Auditor
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-farming-v2
White paper/ Documentation
-
Document log
Initial Audit: 4th March 2022 (Complete)
Quality Control: 5th - 8th March 2022
Final Audit: 10th 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 yield farming incentivizes Liquidity Providers to earn $PILOT tokens by staking their Unipilot LP tokens of whitelisted pools.

System Architecture

Unipilot yield farming has 1 main smart contract; ***UnipilotFarm.sol:*** which allows Liquidity Providers to earn ***$PILOT*** token, an ***$ALT*** token or both by staking their Unipilot LP tokens. ***UnipilotFarm*** linearly distributes the ***$PILOT*** according to ***rewardPerBlock*** and ***rewardMultiplier.***

Contracts Description Table

Contract	Type	Bases		
└	**Function Name**	**Visibility**	**Mutability**	**Modifiers**
UnipilotFarm	Implementation	IUnipilotFarm, ReentrancyGuard		
└	<Constructor>	Public ❗	🛑	NO❗
└	initializer	External ❗	🛑	onlyGovernance
└	stakeLp	External ❗	🛑	nonReentrant
└	unstakeLp	External ❗	🛑	nonReentrant
└	claimReward	Public ❗	🛑	NO❗
└	blacklistVaults	External ❗	🛑	onlyGovernance
└	updateRewardPerBlock	External ❗	🛑	onlyGovernance
└	updateMultiplier	External ❗	🛑	onlyGovernance
└	updateAltMultiplier	External ❗	🛑	onlyGovernance
└	updateGovernance	External ❗	🛑	onlyGovernance
└	vaultListed	Public ❗		NO❗
└	updateRewardType	External ❗	🛑	onlyGovernance
└	migrateFunds	External ❗	🛑	onlyGovernance
└	updateFarmingLimit	External ❗	🛑	onlyGovernance
└	toggleBooster	External ❗	🛑	NO❗
└	setStake	External ❗	🛑	onlyGovernance
└	currentReward	Public ❗		NO❗
└	updateLastBlock	Private 🔐	🛑	
└	getGlobalReward	Private 🔐		
└	updateVaultState	Private 🔐	🛑	
└	updateAltState	Private 🔐	🛑	
└	insertVault	Private 🔐	🛑	
└	verifyLimit	Private 🔐		
└	<Receive Ether>	External ❗	💰	NO❗
└	<Fallback>	External ❗	💰	NO❗

Methodology & Scope

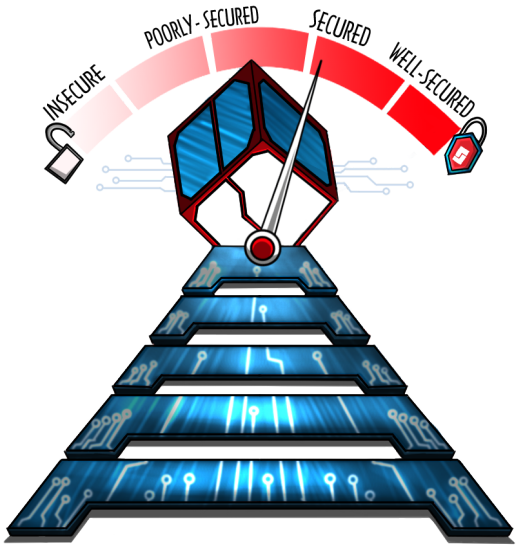
The codebase was audited in an iterative process. Fixes were applied on the way and updated contracts were examined for more bugs. We used a combination of static analysis tool (slither) and testing framework (hardhat) which indicated some of the critical bugs like reentrancy in the code. We also did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

AUDIT REPORT

Executive Summary

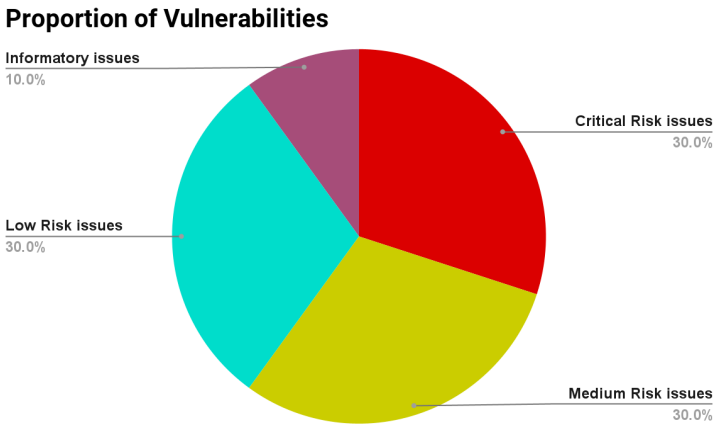
The analysis indicates that some of the functionalities in the contracts audited are **working properly**.

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
3	Critical Risk issue(s)
0	High Risk issue(s)
3	Medium Risk issue(s)
3	Low Risk issue(s)
1	Informatory issue(s)



Findings

#	Findings	Risk	Status
1.	First block reward is wrong in Only ***ALT*** and ***Dual*** Case.	Critical	Fixed
2.	Consecutive change in reward type will disable the ***stakeLp()*** functionality.	Critical	Fixed
3.	Miscalculation in pilot reward	Critical	Fixed
4.	***RewardToken*** validity should be check	Medium	Fixed
5.	The ***UpdateRewardPerBlock()*** function will manipulate the last reward for every user.	Medium	Fixed
6.	Should allow Parameters to send Zero to reopen farming	Medium	Fixed
7.	***RewardToken*** Zero Address check	Low	Fixed
8.	Necessary checks in ***updateFarmingLimit()*** function	Low	Fixed
9.	Emergency exit for users	Low	Fixed
10.	Should update ***Multiplier*** and ***RewardType*** while calling init function.	Informatory	Fixed

Critical-risk issues

1. **First block reward is wrong in Only ALT and Dual Case.**

Description:
If the reward type is changed from only ***Pilot*** to only ***ALT*** or only ***Pilot*** to ***Dual***, the code updates the ***lastBlockReward*** variable first for ***Pilot*** then for ***ALT***, calculating block difference as 2 instead of 0. Hence, the user gets triple reward for just the first block.

```
} else {
    if (_rewardType⬆ == RewardType.Alt) {
        altState.lastRewardBlock = blockNumber;
        updateVaultState(_vault⬆);
    } else {
        altState.lastRewardBlock = blockNumber;
    }
}
emit RewardStatus(
    _vault⬆,
    vaultInfo[_vault⬆].reward,
    vaultInfo[_vault⬆].reward = _rewardType⬆,
    _altToken⬆
);
```

Remedy:
When updating the reward type ***lastRewardBlock*** should be set to the current block number.

```
else {
  if (_rewardType⬆ == RewardType.Alt) {
      altState.lastRewardBlock = blockNumber;
      vaultState.startBlock = blockNumber;
      updateVaultState(_vault⬆);
  } else {
      altState.lastRewardBlock = blockNumber;
  }
```

Status:
Fixed as per BlockApex recommendation.

2. **Consecutive change in reward type will disable the stakeLp functionality.**

Description:

If the ***governance*** changes reward type from ***only Pilot*** to ***only Alt*** and then changes it again from ***only Alt*** to ***Dual***, no user can stake their ***LPs.***
Scenario: Before staking starts, if the last reward type is changed twice and is finally set to ***Dual***, it causes the last reward block check, inside the ***stakeLp()*** function, to break, i.e., to not equal to start block, causing the ***getGlobalReward()*** function to be invoked, which in turn causes the ***mulDiv()*** error as the value of ***totalLpLocked*** never changed from zero.

```
function getGlobalReward(
    address _vault↑,
    uint256 _blockDiff↑,
    uint256 _multiplier↑,
    uint256 _lastGlobalReward↑
) private view returns (uint256 _globalReward↑) {


    if (vaultWhitelist[_vault↑]) {
        _globalReward↑ = FullMath.mulDiv(rewardPerBlock, _multiplier↑, 1e18);
        _globalReward↑ = FullMath
            .mulDiv(
                _blockDiff↑.mul(_globalReward↑),
                1e18,
                vaultInfo[_vault↑].totalLpLocked
            )
            .add(_lastGlobalReward↑);
    } else {
        _globalReward↑ = vaultInfo[_vault↑].globalReward;
    }
}
```

Remedy:

If the ***totalLpLocked*** is zero, the previous state of global reward should return the value of its previous state. In case zero returns, the function should not calculate the new global reward.

```
function getGlobalReward(
    address _vault,
    uint256 _blockDiff,
    uint256 _multiplier,
    uint256 _lastGlobalReward
) private view returns (uint256 _globalReward) {
    if (vaultWhitelist[_vault]) {
        if (vaultInfo[_vault].totalLpLocked > 0) {
            _globalReward = FullMath.mulDiv(
                rewardPerBlock,
                _multiplier,
                1e18
            );

            _globalReward = FullMath
                .mulDiv(
                    _blockDiff.mul(_globalReward),
                    1e18,
                    vaultInfo[_vault].totalLpLocked
                )
                .add(_lastGlobalReward);
        } else {
            _globalReward = vaultInfo[_vault].globalReward;
        }
    } else {
        _globalReward = vaultInfo[_vault].globalReward;
    }
}
```

Status:

Fixed as per BlockApex recommendation.

3. **Miscalculation in pilot reward.**

   Description:
   In an empty farm, if the reward type is changed from only ***Pilot*** to only ***ALT*** or only ***Pilot*** to ***Dual***, the first user who stakes gets a reward of all the empty blocks since the reward type has been updated.

   Remedy:
   Modify the ***updateRewardType()*** function to check if there are no staked LPs in the farm, only then should the start block be updated to set as the current block number.

   Status:
   Fixed as per BlockApex recommendation.

High-risk issues

No issues were found

Medium-risk issues

1. **RewardToken validity should be checked.**

   Description:

   - While calling the ***initializer()*** function, if the user sends invalid reward token there is no check to validate whether that token exists in the contract or not.
   - Same problem is confirmed in the ***UpdateRewardType()*** function, which also requires a token validity check..

```
if (!vaultWhitelist[_vault⬆[i]] && vaultState.totalLpLocked == 0) {
    insertVault(_vault⬆[i], _multiplier⬆[i]);
} else {
    require(!vaultWhitelist[_vault⬆[i]], "AI");
    if (vaultState.reward == RewardType.Dual) {
        vaultState.lastRewardBlock = blockNum;
        vaultAltState.lastRewardBlock = blockNum;
    } else if (vaultState.reward == RewardType.Alt) {
        vaultAltState.lastRewardBlock = blockNum;
    } else {
        vaultState.lastRewardBlock = blockNum;
    }
}
```

   Remedy:

   initializer() and ***UpdateRewardType()*** functions should check in the vault contract if the token actually exists by calling the ***balanceOf()*** function.

```
if (!vaultWhitelist[_vault⬆[i]] && vaultState.totalLpLocked == 0) {
    if (
        _rewardType⬆[i] == RewardType.Alt ||
        _rewardType⬆[i] == RewardType.Dual
    ) {
        require(
            IERC20(_rewardToken⬆[i]).balanceOf(address(this)) > 0,
            "NEB"
        );
        vaultAltState.multiplier = _multiplier⬆[i];
        vaultAltState.startBlock = blockNum;
        vaultAltState.lastRewardBlock = blockNum;
        vaultAltState.rewardToken = _rewardToken⬆[i];
    }
```

Status:
Fixed as per BlockApex recommendation.

2. **The UpdateRewardPerBlock function will manipulate the last reward for every user.**

Description:

The ***event*** in ***updateRewardPerBlock()*** function ***emits*** the old reward value as well as updating and sending the new reward value, the problem is that this event ***emits*** at the start of the function which manipulates the last block reward for every user in the vault.

```
function updateRewardPerBlock(uint256 _value⬆)
    external
    override
    onlyGovernance
{
    emit RewardPerBlock(rewardPerBlock, rewardPerBlock = _value⬆);
    require(_value⬆ > 0, "IV");
```

Remedy:

Event should be fired after the calculation at the end of the function.

```
function updateRewardPerBlock(uint256 _value)
    external
    override
    onlyGovernance
{
    require(_value > 0, "IV");
    address[] memory vaults = vaultListed();
    for (uint256 i = 0; i < vaults.length; i++) {
        if (vaultWhitelist[vaults[i]]) {
            if (vaultInfo[vaults[i]].totalLpLocked != 0) {
                if (vaultInfo[vaults[i]].reward == RewardType.Dual) {
                    updateVaultState(vaults[i]);
                    updateAltState(vaults[i]);
                } else if (vaultInfo[vaults[i]].reward == RewardType.Alt) {
                    updateAltState(vaults[i]);
                } else {
                    updateVaultState(vaults[i]);
                }
            }
        }
    }
    emit RewardPerBlock(rewardPerBlock, rewardPerBlock = _value);
}
```

Status:
Fixed as per BlockApex recommendation

3. Should allow Parameters to send Zero to reopen farming

Description:
There is a check in ***updateFarmingLimit()*** function which does not allow sending zero value in parameter, but it contradicts with the functionality. If the gov sets the farming limit to a specific block and they want to reopen or update the limit they have to send zero value to the ***updateFarmingLimit()*** function which will not work if zero value check is placed.

```
function updateFarmingLimit(uint256 _blockNumber)
    external
    override
    onlyGovernance
{
   require(_blockNumber > 0, "IV");
    emit UpdateFarmingLimit(
        farmingGrowthBlockLimit,
        farmingGrowthBlockLimit = _blockNumber
    );
    updateLastBlock();
}
```

Remedy:
Zero value check should be removed from the function.

```
function updateFarmingLimit(uint256 _blockNumber)
    external
    override
    onlyGovernance
{
    emit UpdateFarmingLimit(
        farmingGrowthBlockLimit,
        farmingGrowthBlockLimit = _blockNumber
    );
    updateLastBlock();
}
```

Status:
Fixed as per BlockApex recommendation

Low-risk issues

1. **RewardToken validity check**

   Description:

   No zero address check placed for ***RewardToken*** while calling the ***initializer()*** function.

```
function initializer(
    address[] calldata _vault⬆,
    uint256[] calldata _multiplier⬆
) external override onlyGovernance {
    require(_vault⬆.length == _multiplier⬆.length, "LNS");
    uint256 blockNum = block.number;
    for (uint256 i = 0; i < _vault⬆.length; i++) {
        VaultInfo storage vaultState = vaultInfo[_vault⬆[i]];
        AltInfo storage vaultAltState = vaultAltInfo[_vault⬆[i]];
```

   Remedy:

   Zero address check should be Placed.

   Status:

   Fixed as per BlockApex recommendation

2. **Necessary checks in updateFarmingLimit() function.**

Description:

If the ***updateFarmingLimit()*** function is called with the same value of the block number in which it is going to be executed (or the past block number), the tx will be mined but it will not limit the farming as expected.

```
function updateFarmingLimit(uint256 _blockNumber)
    external
    override
    onlyGovernance
{
    emit UpdateFarmingLimit(
        farmingGrowthBlockLimit,
        farmingGrowthBlockLimit = _blockNumber
    );
    updateLastBlock();
}
```

Remedy:

A check should be placed in the ***updateFarmingLimit()*** function to ensure that block number never equals to current block or past block.

Status:

Fixed as per BlockApex recommendation

3. **Emergency exit for users.**

   Description:

   In event of any mishap with $ALT or $PILOT reward, a user won't be able to withdraw their LP funds. A user calls the ***unstakeLp()*** function, the contract will throw the ***"Insufficient balance"*** error.

   Remedy:

   Contract should have emergency ***withdraw()*** function to withdraw user's staked LPs

   Status:

   Fixed as per BlockApex recommendation

Informatory issues and Optimization

1. **Should update Multiplier and RewardType while calling init function.**

   Description:

   It's extra work for the ***governance***, for the first time if they want to set a ***vault*** for only ***ALT*** reward they have to call 3 different functions, this work can be done by calling only one function.

```
function initializer(
    address[] calldata _vault⬆,
    uint256[] calldata _multiplier⬆
) external override onlyGovernance {
    require(_vault⬆.length == _multiplier⬆.length, "LNS");
    uint256 blockNum = block.number;
    for (uint256 i = 0; i < _vault⬆.length; i++) {
        VaultInfo storage vaultState = vaultInfo[_vault⬆[i]];
        AltInfo storage vaultAltState = vaultAltInfo[_vault⬆[i]];
```

   Remedy:

   RewardType and ***RewardToken*** should be set by Calling ***Initializer()*** function and later it can be handled by individual functions.

```
function initializer(
    address[] calldata _vault⬆,
    uint256[] calldata _multiplier⬆,
    RewardType[] calldata _rewardType⬆,
    address[] calldata _rewardToken⬆
) external override onlyGovernance {
```

   Status:

   Fixed as per BlockApex recommendation

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.