QuillAudits

Audit Report
November, 2020

Contents

Introduction

During the period of November 11th, 2020 to November 14th, 2020 – Quillhash Team performed a security audit for **Yearn Finance Passive Income (YFPI)** smart contract. The code for audit was taken from following:
https://etherscan.io/address/0x05D27CdD23E22ca63e7f9c7C6D1B79ede9C4fCF5#code

Overview of YFPI

YEARN FINANCE PASSIVE INCOME (YFPI) is a Governance token. Users who stake YFPI tokens can vote for potential product upgrades, releases, and parameter fixes. Users will be able to withdraw their funds at any time with no delay. Rewards are calculated and distributed weekly on every Sunday to user wallets between 7 PM and 8PM UTC. YFPI Token is an experimental token in beta phase. Do Your Own Research (DYOR) before investing in this venture, as Crypto-currencies are subject to high market risk and volatility.
Official Website: https://yfpi.finance/

Scope of Audit

The scope of this audit was to analyze and document YFPI smart contract codebase for quality, security, and correctness.

Check Vulnerabilities

We have scanned YFPI smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Transaction-Ordering
- Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality

- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level
- Address hardcoded
- Using delete for arrays
- Integer overflow/underflow
- Locked money
- Private modifier
- Revert/require functions
- Using var
- Visibility
- Using blockhash
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contracts care was taken to ensure:

- Overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per intended behavior mentioned in whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis
In this step we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.
SmartCheck.

Static Analysis
Static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step a series of automated tools are used to test security of smart contracts.

Code Review / Manual Analysis
Manual Analysis or review of code was done to identify new vulnerability or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of automated analysis were manually verified.

Gas Consumption
In this step we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Remix IDE, Truffle, Truffle Team, Ganache, Solhint, Mythril, Slither, SmartCheck.

Issue Categories

Every issue in this report has been assigned with a severity level. There are four levels of severity and each of them has been explained below.

High severity issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality and we recommend these issues to be fixed before moving to a live environment.

Medium level severity issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems and they should still be fixed.

Low level severity issues

Low level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity four issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

	High	Medium	Low	Informational
Open	0	0	6	4
Closed	0	0	4	1

Issues Found – Code Review / Manual Testing

High severity issues

None.

Medium severity issues

None.

Low level severity issues

1. **Compiler version should be fixed**
   Solidity source files indicate the versions of the compiler they can be compiled with. It's recommended to lock the compiler version in code, as future compiler versions may handle certain language constructions in a way the developer did not foresee. Also, it's recommended to use the latest compiler version.
   Auditor Remarks: Done. Closed.

2. **Coding Style Issues**
   Coding style issues influence code readability and in some cases may lead to bugs in future. Smart Contracts have naming convention, indentation and code layout issues. It's recommended to use Solidity Style Guide to fix all the issues. Consider following the Solidity guidelines on formatting the code and commenting for all the files. It can improve the overall code quality and readability.
   Auditor Remarks: **Done. Closed.**

3. **Order of layout**
   The order of functions as well as the rest of the code layout does not follow solidity style guide.
   Layout contract elements in the following order:
   - Pragma statements
   - Import statements
   - Interfaces
   - Libraries
   - Contracts

   Inside each contract, library or interface, use the following order:
   - Type declarations
   - State variables
   - Events
   - Functions

   Please read following documentation links to understand the correct order:
   - https://solidity.readthedocs.io/en/v0.5.3/style-guide.html#order-of-layout
   - https://solidity.readthedocs.io/en/v0.5.3/style-guide.html#order-of-functions

   Auditor Remarks: Done. Closed.

4. **Give preference for 'bytes32' over 'string'**
   For constant values smaller than 32 bytes, give preference for a bytes32 type, since they are much cheaper than string types, which are dynamically sized.

5. **View functions should not modify the state[#182-188]**
   Using inline assembly that contains certain opcodes is considered as modifying the state. Functions that modify the state should not be declared as view functions. Do not declare functions that change the state as view.

6. **Use external function modifier instead of public**
   The public functions that are never called by contract should be declared external to save gas. Following functions can be declared external:
   -ERC20.totalSupply()[#61-63]
   -ERC20.balanceOf(address)[#64-66]
   -ERC20.transfer(address,uint256)[#67-70]
   -ERC20.allowance(address,address)[#71-73]
   -ERC20.approve(address,uint256)[#74-77]
   -ERC20.transferFrom(address,address,uint256)[#78-82]
   -ERC20.increaseAllowance(address,uint256)[#83-86]
   -ERC20.decreaseAllowance(address,uint256)[#87-90]
   -ERC20Detailed.name()[#132-134]
   -ERC20Detailed.symbol()[#135-137]
   -YFPI.burn(uint256)[#235-237]
   Auditor Remarks: Done. Closed.

Informational

These are severity four issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

1. **Do not use EVM assembly. Use of assembly is error-prone and should be avoided.**

2. **Be explicit about which `uint` the code is using**
   `uint` is an alias for `uint256`, but using the full form is preferable. Be consistent and use one of the forms.
   Auditor Remarks: Done. Closed.

3. **Low level calls [#209-220]**
   Low-level calls do not check for code existence or call success. Use of low-level calls should be avoided as they are error-prone.

4. Approve function of ERC-20 is vulnerable [#74-77]

The Multiple Withdrawal Attack is a known attack on ERC20 that allows an approved spender to transfer more than allowed by another user.

More details can be in the below link:
https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit#heading=h.q96g0auxmi6a

The above document explains in detail on how to fix the issue using three different approaches:

- Atomic "Compare And Set" Approve Method
- Separate Log Message for "TransferFrom" Transfers
- Four-Args Approval Event

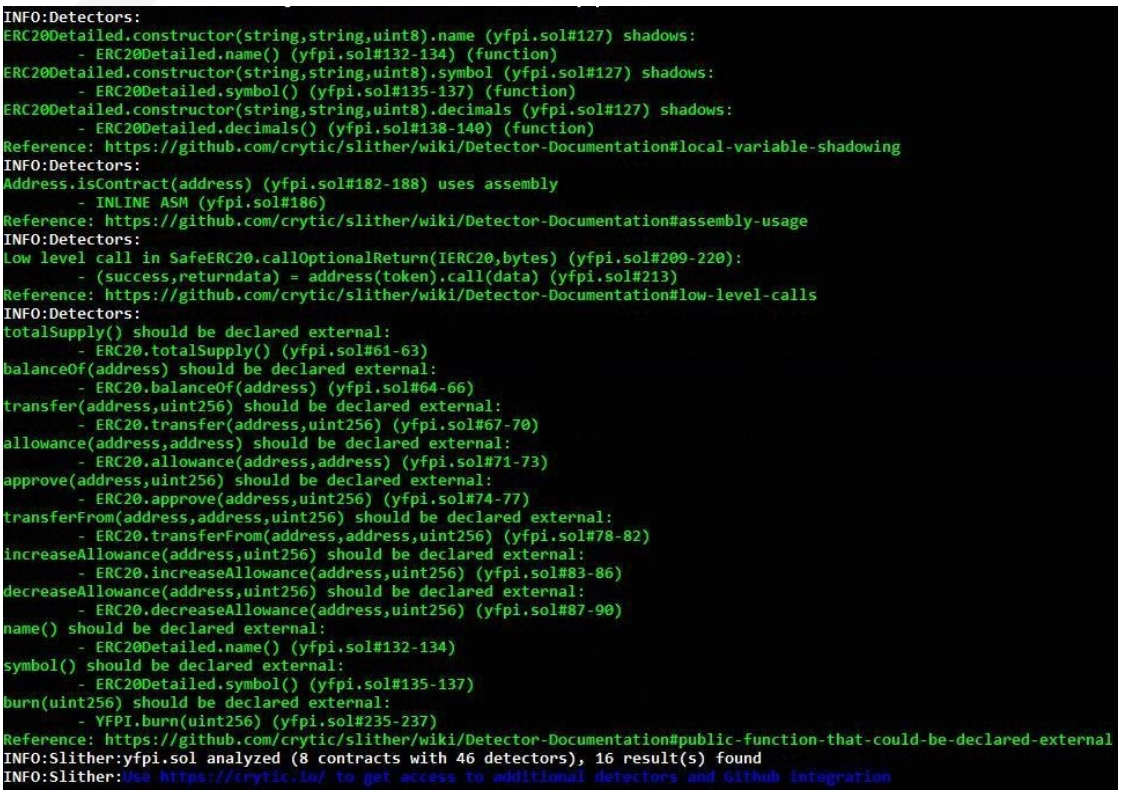
For more details on how to resolve the multiple withdrawal attack check the following document for details:
https://drive.google.com/file/d/1mr7Mw_161ZxK4jBfDRbuzsXKPiaVGkjY/view?usp=sharing

Automated Testing

Slither

Slither is an open-source Solidity static analysis framework. This tool provides rich information about Ethereum smart contracts and has the critical properties. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses.

- Detects vulnerable Solidity code with low false positives
- Identifies where the error condition occurs in the source code
- Easily integrates into continuous integration and Truffle builds
- Built-in 'printers' quickly report crucial contract information
- Detector API to write custom analyses in Python
- Ability to analyze contracts written with Solidity >= 0.4
- Intermediate representation (SlithIR) enables simple, high-precision analyses
- Correctly parses 99.9% of all public Solidity code
- Average execution time of less than 1 second per contract



Slither didn't raise any critical issue with YFPI smart contract. The contract was well tested and all the minor issues that were raised have been documented in the report. All other vulnerabilities of importance have already been covered in the manual audit section of the report.

SmartCheck

SmartCheck is a tool for automated static analysis of Solidity source code for security vulnerabilities and best practices. SmartCheck translates Solidity source code into an XML-based intermediate representation and checks i t against XPath patterns. SmartCheck shows significant improvements over existing alternatives in terms of false discovery rate (FDR) and false negative rate (FNR). It gave the following result for the YFPI contract:

https://tool.smartdec.net/scan/4a1cbf0217c04c188c7b6458f3d08810

SmartCheck did not detect any high severity issue. All the considerable issues raised by SmartCheck are already covered in the code review section of this report.

Mythril

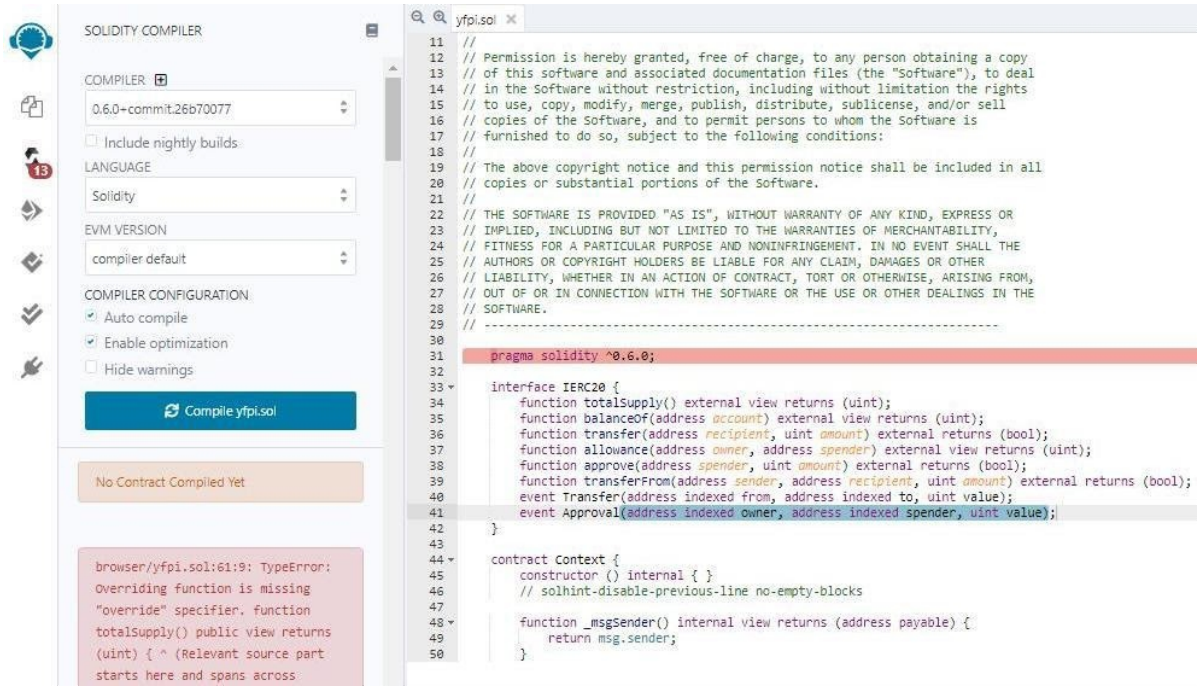
Mythril is a security analysis tool for EVM bytecode. It detects security vulnerabilities in smart contracts built for Ethereum. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities.

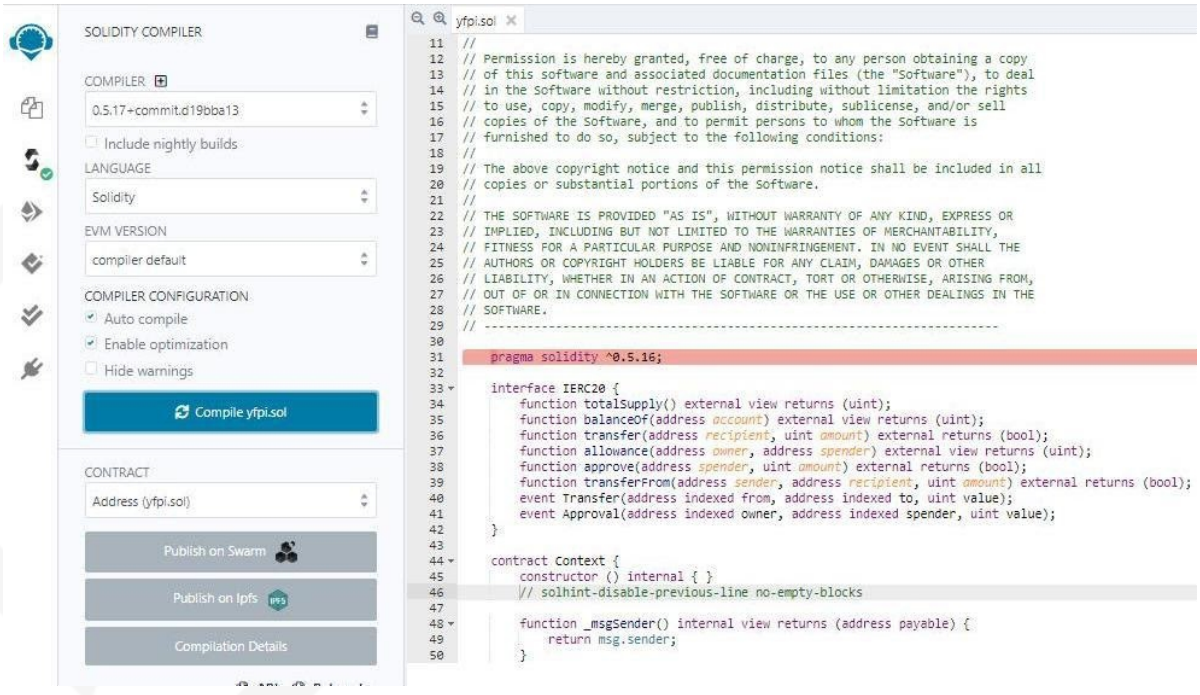
Mythril did not detect any high severity issue. All the considerable issues raised by Mythril are already covered in the code review section of this report.

Remix IDE

Remix is a powerful, open source tool that helps you write Solidity contracts straight from the browser. Written in JavaScript, Remix supports both usage in the browser and locally.

The YFPI smart contract was tested for various compiler versions. The smart contract compiled without any error on explicitly setting compiler version 0.5.17 to 0.6.12. The contract fails to compiler when compiler version is set to 0.6.0 or higher.





It is recommended to use any fixed compiler versions >= 0.5.17 <=0.6.0
The contract threw some warnings and these warnings have been reported in the code review or manual testing section of this report. There was no critical or major issue reported by Remix.

Closing Summary

Overall, the smart contracts are very well written and adhere to ERC-20 guidelines. Several issues of low severity were found during the audit. There were no critical or major issues found that can break the intended behaviour.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the YFPI platform. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the YFPI Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

hello@quillhash.com