SMART CONTRACT AUDIT REPORT

for

YFETH

Prepared by: O Bari
September 26, 2020

Document Properties

Client	Alex
Title	Smart Contract Audit Report
Target	YFETH
Version	1.0
Auditor	O Bari
Classification	Confidential

Contents

1 Introduction

Given the opportunity to review the **YFETH** smart contract source code, we in the report outline our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given branch of YFETH can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About YFETH

`YFETH` is designed as an evolutionary improvement of `UniswapV2`, which is a major decentralized exchange (DEX) running on top of Ethereum blockchain. `YFETH` used `UniswapV2`'s core design, but extended with features such as liquidity provider incentives and community-based governance. We note that with `UniswapV2`, liquidity providers only earn the pool's trading fees when they are actively providing the pool liquidity. Once they have withdrawn their portion of the pool, they no longer receive that reward. With `YFETH`, `YFE` holders will be entitled to continue to earn a portion of the protocol's trading fee, even though she no longer participates in the liquidity provision.
The basic information of YFETH is as follows:

Table 1.1: Basic Information of YFETH

Item	Description
Issuer	YFETH
Website	
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 26, 2020

In the following, we show the Git repository of reviewed code and the commit hash value used in his audit:
https://github.com/gupt-blip/YFETH

1.2 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: H, M and L, i.e., $high$, $medium$ and low respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., $Critical$, $High$, $Medium$, Low shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Critical	High	Medium
High	Medium	Low
Medium	Low	Low

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) , which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.3 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the YFETH implementation

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	0	
Informational	1	■
Total	2	

More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1

Table 2.1: Key YFETH Audit Findings

ID	Severity	Title	Category	Status
1	Informational	Incompatibility with Deflationary Tokens	Business Logics	Partially Fixed
2	Medium	Improved Logic in getMultiplier()	Business Logics	Confirmed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Incompatibility With Deflationary Tokens

- ID: 1
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Business Logics
- CWE subcategory: CWE-708

Description

In `YFETH`, the `MasterChef` contract operates as the main entry for interaction with staking users. The staking users `deposit` `UniswapV2`'s LP tokens into the `YFETH` pool and in return get proportionate share of the pool's rewards. Later on, the staking users can `withdraw` their own assets from the pool. With assets in the pool, users can earn whatever incentive mechanisms proposed or adopted via governance.

Naturally, the above two functions, i.e., `deposit()` and `withdraw()`, are involved in transferring users' assets into (or out of) the `YFETH` protocol. Using the `deposit()` function as an example, it needs to transfer deposited assets from the user account to the pool. When transferring standard ERC20 tokens, these asset-transferring routines work as expected: namely the account's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts (lines 242 * 264).

```
// Deposit LP tokens to MasterChef for YFE allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
```

```
            uint256 pending = user
                .amount
                .mul(pool.accYFEPerShare)
                .div(1e12)
                .sub(user.rewardDebt);
            if (pending > 0) {
                safeYFETransfer(msg.sender, pending);
            }
        }
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(
                address(msg.sender),
                address(this),
                _amount
            );
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accYFEPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.11: MasterChef.sol

However, in the cases of deflationary tokens, as shown in the above code snippets, the input amount may not be equal to the received amount due to the charged (and burned) transaction fee. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary tokens. Apparently, these balance inconsistencies are damaging to accurate portfolio management of `MasterChef` and affects protocol-wide operation and maintenance.

One mitigation is to query the asset change right before and after the asset-transferring routines. In other words, instead of automatically assuming the amount parameter in `transfer()` or `transferFrom ()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()`/`transferFrom()` is expected and aligned well with the intended operation. Though these additional checks cost additional gas usage, we feel that they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `MasterChef` pools. With the single entry of adding a new pool (via `add()`), `MasterChef` is indeed in the position

to effectively regulate the set of assets allowed into the protocol.

Fortunately, the `UniswapV2`'s LP tokens are not deflationary tokens and there is no need to take any action in `YFETH`. However, it is a potential risk if the current code base is used elsewhere or the need to add other tokens arises (e.g., in listing new DEX pairs). Also, the current code implementation, including the `UniswapV2`'s path-supported `swap()` and thus `YFETH`'s similar `swap()`, is indeed not compatible with deflationary tokens.

Recommendation Regulate the set of LP tokens supported in `YFETH` and, if there is a need to support deflationary tokens, add necessary mitigation mechanisms to keep track of accurate balances.

3.2 Improved Logic in getMultiplier()

- ID: 2
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Status Codes
- CWE subcategory: CWE-682

Description

YFETH incentives early adopters by specifying an initial list of 13 pools into which early adopters can stake the supported `UniswapV2`'s LP tokens. The earnings were started at block 10, 750, 000 in

Ethereum. For every new block, there will be 100 new `YFE` tokens minted and these minted tokens will be accordingly redistributed to the stakers of each pool. For the first 100, 000 blocks (lasting about 2 weeks), the amount of `YFE` tokens produced will be multiplied by 10, resulting in 1, 000 `YFE` tokens being minted per block.

The early incentives are greatly facilitated by a helper function called `getMultiplier()`. This function takes two arguments, i.e., `_from` and `_to`, and calculates the reward multiplier for the given block range (`[_from, _to]`).

```
// Return reward multiplier over the given _from to _to block.
function getMultiplier(uint256 _from, uint256 _to)
    public
    view
    returns (uint256)
{
    if (_to <= bonusEndBlock) {
        return _to.sub(_from).mul(BONUS_MULTIPLIER);
    } else if (_from >= bonusEndBlock) {
        return _to.sub(_from);
    } else {
        return
            bonusEndBlock.sub(_from).mul(BONUS_MULTIPLIER).add(
                _to.sub(bonusEndBlock)
            );
    }
}
```

Listing 3.15: MasterChef.sol

For elaboration, the helper's code snippet is shown above. We notice that this helper does not take into account the initial block (`startBlock`) from which the inventive rewards start to apply. As a result, when a normal user gives arbitrary arguments, it could return wrong reward multiplier! A correct implementation needs to take `startBlock` into account and appropriately re-adjusts the starting block number, i.e., `_from = _from >= startBlock ? _from : startBlock`.

We also notice that the helper function is called by two other routines, e.g., `pendingYFE()` and

`updatePool()`. Fortunately, these two routines have ensured `_from >= startBlock` and always use the correct reward multiplier for reward redistribution.

Recommendation Apply additional sanity checks in the `getMultiplier()` routine so that the internal `_from` parameter can be adjusted to take `startBlock` into account.

```
// Return reward multiplier over the given _from to _to block .
function getMultiplier(uint256 _from, uint256 _to) public view returns(uint256) {
    _from = _from >= startBlock ? _from : startBlock;
    if (_to <= bonusEndBlock) {
        ret urn _to.sub(_from).mul(BONUS_MULTIPLIER);
    }
    else
    if (_from >= bonusEndBlock) {
        return _to.sub(_from);
    }
    else {
        return bonusEndBlock.sub(_from).mul(BONUS_MULTIPLIER).add(
            _to.sub(bonusEndBlock)
        );
}
```

Listing 3.2: MasterChef.sol

3.3 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.0;` instead of `pragma solidity >=0.6.0;`. Also io would suggest to use latest LTS version 0.7.1

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet.

report can be accessed in the following
link: https://blog.openzeppelin.comcompound-alpha-governance-system-audit.

4 | Conclusion

In this audit, we thoroughly analyzed the YFETH design and implementation. Overall, YFETH presents an evolutional improvement based on Uniswap and provide extra incentives to liquidity providers. Our impression is that the current code base is well organized.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities .
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` excep- tion.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.

  Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()`, which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[5] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

[6] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

[7] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[8] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[9] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[10] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[11] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[12] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[13] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699. html.

[14] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[15] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.

[16] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.

End of report