SMART CONTRACT AUDIT REPORT

for

SUSHISWAP

Prepared By: Shuxiao Wang

Hangzhou, China
September 3, 2020

Document Properties

Client	SushiSwap
Title	Smart Contract Audit Report
Target	SushiSwap
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Chiachih Wu, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	September 3, 2020	Xuxian Jiang	Final Release
0.2	September 2, 2020	Xuxian Jiang	Add More Findings
0.1	September 1, 2020	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **SushiSwap** 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 SushiSwap 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 SushiSwap

`SushiSwap` is designed as an evolutional improvement of `UniswapV2`, which is a major decentralized exchange (DEX) running on top of Ethereum blockchain. `SushiSwap` 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 Once they have withdrawn their portion of the pool, they no longer receive that reward.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 `SushiSwap`, `SUSHI` 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 SushiSwap is as follows:

Table 1.1: Basic Information of SushiSwap

Item	Description
Issuer	SushiSwap
Website	https://sushiswap.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 3, 2020

In the following, we show the Git repository of reviewed code and the commit hash value used in this audit:

- https://github.com/sushiswap/sushiswap (180bc9b)

1.2 About PeckShield

PeckShield Inc. [19] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact	*High*	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	*Medium*	*Low*
			Likelihood	

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [14]:

- 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

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) [13], 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.

1.4 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 SushiSwap implementation. During the first phase of our audit, we studied the smart contract source code and ran our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■ ■
Medium	3	■ ■ ■
Low	6	■ ■ ■ ■ ■ ■
Informational	2	■ ■
Total	13	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. 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), including 2 high-severity vulnerabilities, 3 medium-severity vulnerabilities, 6 low-severity vulnerabilities and 2 informational recommendations.

Table 2.1: Key SushiSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Potential Front-Running For Migration Blocking	Time and State	Partially Fixed
PVE-002	Low	Avoidance of Unnecessary (Small) Loss During Migration	Business Logics	Fixed
PVE-003	Medium	Duplicate Pool Detection and Prevention	Business Logics	Confirmed
PVE-004	Informational	Recommended Explicit Pool Validity Checks	Security Features	Confirmed
PVE-005	Informational	Incompatibility with Deflationary Tokens	Business Logics	Partially Fixed
PVE-006	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Confirmed
PVE-007	Medium	Improved Logic in getMultiplier()	Business Logics	Confirmed
PVE-008	Medium	Improved EOA Detection Against Front-Running of Revenue Conversion	Business Logics	Fixed
PVE-009	Low	No Pair Creation With Zero Migration Balance	Business Logics	Confirmed
PVE-010	Low	Full Charge of Proposal Execution Cost From Accompanying msg.value	Business Logics	Confirmed
PVE-011	Low	Improved Handling of Corner Cases in Proposal Submission	Error Conditions, Return Values, Status Codes	Confirmed
PVE-012	Low	Better Clarification of SUSHI Inflation	Business Logics	Confirmed
PVE-013	High	Non-Governance-Based Admin of TimeLock And Related Privileges	Security Features	Confirmed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Potential Front-Running For Migration Blocking

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `UniswapV2Pair`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [3]

Description

SushiSwap has developed unique tokenomics in two phases: In the first phase, traders stake the `UniswapV2`'s liquidity pools tokens for mining `SUSHI` tokens; and in the second phase, traders are meant to migrate those `UniswapV2`'s liquidity pools tokens for the underlying assets to the `SushiSwap` DEX. The migration might be incentivized by the different token distribution mechanics proposed by `SushiSwap`. Specifically, with the current `UniswapV2` configuration, 0.3% of all trading fees in any pool are proportionately distributed to the pool's liquidity providers. In comparison, `SushiSwap` allocates 0.25% directly to the active liquidity providers, but the remaining 0.05% are converted back to `SUSHI` and re-distributed to the `SUSHI` token holders.

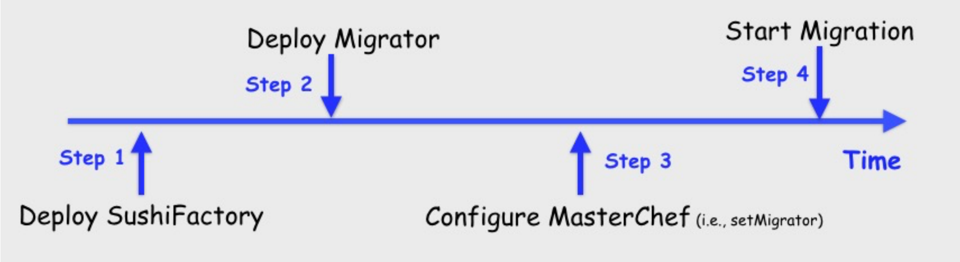

Figure 3.1: The Migration Procedure

Mechanically, the migration procedure can be divided into four distinct steps: `deploy sushiFactory` , `deploy migrator`, `configure MasterChef`, and `start migration`. Note these four steps need to be sequentially executed and the timing of their execution is crucial. In particular, if we examine the final step, i.e., `start migration`, the migration process is kicked off by invoking the `migrate()` routine, which has a final check in place after the migration, i.e., `require(bal == newLpToken.balanceOf(address(this)), "migrate: bad")`. For simplicity, we call this particular check as the `migration check`.

```
    // Migrate lp token to another lp contract. Can be called by anyone. We trust that
        migrator contract is good.
    function migrate(uint256 _pid) public {
        require(address(migrator) != address(0), "migrate: no migrator");
        PoolInfo storage pool = poolInfo[_pid];
        IERC20 lpToken = pool.lpToken;
        uint256 bal = lpToken.balanceOf(address(this));
        lpToken.safeApprove(address(migrator), bal);
        IERC20 newLpToken = migrator.migrate(lpToken);
        require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
        pool.lpToken = newLpToken;
    }
```

Listing 3.1: MasterChef.sol

The actual bulk work of migration is performed by the `Migrator` contract in a function also named `migrate()` (we show the related code snippet below). It in essence burns the `UniswapV2`'s liquidity pool (or LP) tokens to reclaim the underlying assets and transfers them to `SushiSwap` for minting of the corresponding new pair's LP tokens.

```
26      function migrate(IUniswapV2Pair orig) public returns (IUniswapV2Pair) {
27          require(msg.sender == chef, "not from master chef");
28          require(block.number >= notBeforeBlock, "too early to migrate");
29          require(orig.factory() == oldFactory, "not from old factory");
30          address token0 = orig.token0();
31          address token1 = orig.token1();
32          IUniswapV2Pair pair = IUniswapV2Pair(factory.getPair(token0, token1));
33          if (pair == IUniswapV2Pair(address(0))) {
34              pair = IUniswapV2Pair(factory.createPair(token0, token1));
35          }
36          uint256 lp = orig.balanceOf(msg.sender);
37          if (lp == 0) return pair;
38          desiredLiquidity = lp;
39          orig.transferFrom(msg.sender, address(orig), lp);
40          orig.burn(address(pair));
41          pair.mint(msg.sender);
42          desiredLiquidity = uint256(-1);
43          return pair;
44      }
```

Listing 3.2: Migrator.sol

We emphasize that the staked `UniswapV2`'s LP tokens are transferred back to the `UniswapV2` pair for redemption of the underlying assets (lines $39-40$) and the redeemed underlying assets are then sent to the new pair in `SushiSwap` for minting (lines $40-41$).

The new `SushiSwap` pair's `mint()` function is shown below. Here comes the critical part: the migration process assumes the migrator is the first to mint the new LP tokens (of this particular trading pair). Otherwise, the migration will fail! This assumption essentially reflects the code logic in lines 126−128. In other words, if an actor is able to front-run it to become the first one in successfully minting the new LP tokens, the actor will successfully block this migration (of this specific trading pair or the pool in `MasterChef`).

```
115     function mint(address to) external lock returns (uint liquidity) {
116         (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
117         uint balance0 = IERC20Uniswap(token0).balanceOf(address(this));
118         uint balance1 = IERC20Uniswap(token1).balanceOf(address(this));
119         uint amount0 = balance0.sub(_reserve0);
120         uint amount1 = balance1.sub(_reserve1);
121 
122         bool feeOn = _mintFee(_reserve0, _reserve1);
123         uint _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update in _mintFee
124         if (_totalSupply == 0) {
125             address migrator = IUniswapV2Factory(factory).migrator();
126             if (msg.sender == migrator) {
127                 liquidity = IMigrator(migrator).desiredLiquidity();
128                 require(liquidity > 0 && liquidity != uint256(-1), "Bad desired liquidity");
129             } else {
130                 require(migrator == address(0), "Must not have migrator");
131                 liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
132             }
133            _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
134         } else {
135             liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul(_totalSupply) / _reserve1);
136         }
137         require(liquidity > 0, ’UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED’);
138         _mint(to, liquidity);
139 
140         _update(balance0, balance1, _reserve0, _reserve1);
141         if (feeOn) kLast = uint(reserve0).mul(reserve1); // reserve0 and reserve1 are up-to-date
142         emit Mint(msg.sender, amount0, amount1);
143     }
```

Listing 3.3: UniswapV2Pair.sol

Recall the above migration check that essentially states the new LP token amount should equal to the old LP token amount. If the migration transaction is not the first to mint new LP tokens, the first transaction that successfully mints the new LP tokens will lead to `_totalSupply != 0`. In other words, the migration transaction will be forced to take the execution path in lines 135, not the intended lines 126 − 128. As a result, the minted amount is unlikely to be the same as the old

UniswapV2's pool token amount before migration, hence failing the migration check!

To ensure a smooth migration process, we need to guarantee the first minting of new LP tokens is launched by the migration transaction. To achieve that, we need to prevent any unintended minting (of new LP tokens) between the first step deploy sushiFactory and the third step configure MasterChef. A natural approach is to complete the initial three steps within the same transaction, best facilitated by a contract-coordinated deployment.

Recommendation Deploy these contracts in a coherent fashion and avoid the above-mentioned front-running to guarantee a smooth migration.

Status This issue has been confirmed and largely addressed by streamlining the entire deployment script (without the need of actually revising the smart contract implementation). This is indeed the approach the team plans to take and exercise with extra caution when deploying these contracts (by avoiding unnecessary exposure of vulnerable time window for front-running).

3.2 Avoidance of Unnecessary (Small) Loss During Migration

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: UniswapV2Pair.sol
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [8]

Description

We have discussed the four distinct migration steps in Section 3.1 and highlighted the need of being the first one for the migrator to mint the new liquidity pool (LP) tokens. In this section, we further elaborate another issue in current migration logic that could unnecessarily lead to a (small) loss of assets.

The loss is caused in the mint() function of the revised UniswapV2Pair contract. In particular, the first-time minting (with _totalSupply == 0) will take the then branch (line 124) that executes code statements in lines 126 – 128, followed by _mint(address(0), MINIMUM_LIQUIDITY) in line 133.

```
115     function mint(address to) external lock returns (uint liquidity) {
116         (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
117         uint balance0 = IERC20Uniswap(token0).balanceOf(address(this));
118         uint balance1 = IERC20Uniswap(token1).balanceOf(address(this));
119         uint amount0 = balance0.sub(_reserve0);
120         uint amount1 = balance1.sub(_reserve1);
121
122         bool feeOn = _mintFee(_reserve0, _reserve1);
123         uint _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update in _mintFee
124         if (_totalSupply == 0) {
125             address migrator = IUniswapV2Factory(factory).migrator();
126             if (msg.sender == migrator) {
127                 liquidity = IMigrator(migrator).desiredLiquidity();
128                 require(liquidity > 0 && liquidity != uint256(-1), "Bad desired liquidity");
129             } else {
130                 require(migrator == address(0), "Must not have migrator");
131                 liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
132             }
133            _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
134         } else {
135             liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul(_totalSupply) / _reserve1);
136         }
137         require(liquidity > 0, 'UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED');
138         _mint(to, liquidity);
139 
140         _update(balance0, balance1, _reserve0, _reserve1);
141         if (feeOn) kLast = uint(reserve0).mul(reserve1); // reserve0 and reserve1 are up-to-date
142         emit Mint(msg.sender, amount0, amount1);
143     }
```

Listing 3.4: UniswapV2Pair.sol

To understand why current migration logic will cause a small bit of loss, we need to understand the purpose of minting of `MINIMUM_LIQUIDITY` to `address(0)`. It may look strange as it essentially burns `MINIMUM_LIQUIDITY` of LP tokens (and thus introduces the loss). It turns out that it is in place to prevent an early liquidity provider to make the LP token too costly for other liquidity providers to enter, hence blocking the early liquidity provider from monopolizing the liquidity pool. However, since our migration is the early liquidity provider (with likely a large amount of minting), this case will not occur! With that, we can safely move the `MINIMUM_LIQUIDITY` burning operation into the `else` branch (lines 129 − 132). The intention is the burning of `MINIMUM_LIQUIDITY` only occurs in other pairs that are not involved in the migration.

Recommendation Avoid the unnecessary small loss during migration. A quick fix is suggested as below.

```
    function mint(address to) external lock returns (uint liquidity) {
        (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
        uint balance0 = IERC20Uniswap(token0).balanceOf(address(this));
        uint balance1 = IERC20Uniswap(token1).balanceOf(address(this));
        uint amount0 = balance0.sub(_reserve0);
        uint amount1 = balance1.sub(_reserve1);

        bool feeOn = _mintFee(_reserve0, _reserve1);
        uint _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update in _mintFee
        if (_totalSupply == 0) {
            address migrator = IUniswapV2Factory(factory).migrator();
            if (msg.sender == migrator) {
                liquidity = IMigrator(migrator).desiredLiquidity();
                require(liquidity > 0 && liquidity != uint256(-1), "Bad desired liquidity");
            } else {
                require(migrator == address(0), "Must not have migrator");
                liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
                _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
            }
        } else {
            liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul(_totalSupply) / _reserve1);
        }
        require(liquidity > 0, 'UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED');
        _mint(to, liquidity);

        _update(balance0, balance1, _reserve0, _reserve1);
        if (feeOn) kLast = uint(reserve0).mul(reserve1); // reserve0 and reserve1 are up-to-date
        emit Mint(msg.sender, amount0, amount1);
    }
```

Listing 3.5: UniswapV2Pair.sol (revised)

Status The issue has been fixed by this commit: d76898b603aed60a776fc0ac529b199e1a6c8c9e.

3.3 Duplicate Pool Detection and Prevention

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: MasterChef
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [8]

Description

SushiSwap provides incentive mechanisms that reward the staking of UniswapV2 LP tokens with SUSHI tokens. The rewards are carried out by designating a number of staking pools into which UniswapV2 LP tokens can be staked. Each pool has its allocPoint*100%/totalAllocPoint share of scheduled rewards and the rewards these stakers in a pool will receive are proportional to the amount of LP tokens they have staked in the pool versus the total amount of LP tokens staked in the pool.

As of this writing, there are 13 pools that share the rewarded `SUSHI` tokens and 5 more have been scheduled for addition (after voting approval). To accommodate these new pools, SushiSwap has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate `UniswapV2` LP token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`) and the supported `governance` can be leveraged to ensure a duplicate LP token will not be added, it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accSushiPerShare: 0
        }));
    }
```

Listing 3.6: MasterChef.sol

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accSushiPerShare: 0
        }));
    }
```

Listing 3.7: MasterChef.sol (revised)

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers. Worse, it will also bring great troubles for the planned migration!

Status We have discussed this issue with the team and the team is aware of it. Since the `MasterChef` contract is already live (with a huge amount of assets), the team prefers not modifying the code for the duplicate prevention, but instead takes necessary off-chain steps and exercises with extra caution to block duplicates when adding a new pool.

3.4 Recommended Explicit Pool Validity Checks

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MasterChef`
- Category: Security Features [9]
- CWE subcategory: CWE-287 [2]

Description

SushiSwap has a central contract – `MasterChef` that has been tasked with not only the migration (Section 3.1), but also the pool management, staking/unstaking support, as well as the reward distribution to various pools and stakers.

In the following, we show the key `pool` data structure. Note all added pools are maintained in an array `poolInfo`.

```
    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocation points assigned to this pool. SUSHIs to distribute per block.
        uint256 lastRewardBlock;  // Last block number that SUSHIs distribution occurs.
        uint256 accSushiPerShare; // Accumulated SUSHIs per share, times 1e12. See below.
    }
    ...
    // Info of each pool.
    PoolInfo[] public poolInfo;
```

Listing 3.8: MasterChef.sol

When there is a need to add a new pool, set a new allocPoint for an existing pool, stake (by depositing the supported UniswapV2's LP tokens), unstake (by redeeming previously deposited UniswapV2's LP tokens), query pending SUSHI rewards, or migrate the pool assets, there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range [0, poolInfo.length-1]. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say validatePool. This new modifier essentially ensures the given _pool_id or _pid indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not!

```
    // Deposit LP tokens to MasterChef for SUSHI 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.accSushiPerShare).div(1e12).sub(user.
                rewardDebt);
            safeSushiTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.9: MasterChef.sol

We highlight that there are a number of functions that can be benefited from the new pool-validating modifier, including set(), migrate(), deposit(), withdraw(), emergencyWithdraw(), pendingSushi() and updatePool().

Recommendation Apply necessary sanity checks to ensure the given _pid is legitimate. Accordingly, a new modifier validatePool can be developed and appended to each function in the above list.

```
     modifier validatePool(uint256 _pid) {
        require(_pid < poolInfo.length, "chef: pool exists?");
        _;
    }

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

```
212            uint256 pending = user.amount.mul(pool.accSushiPerShare).div(1e12).sub(user.
                   rewardDebt);
213            safeSushiTransfer(msg.sender, pending);
214        }
215        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
216        user.amount = user.amount.add(_amount);
217        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(1e12);
218        emit Deposit(msg.sender, _pid, _amount);
219    }
```

Listing 3.10: MasterChef.sol

Status We have discussed this issue with the team. For the same reason as outlined in Section 3.3, because the MasterChef contract is already live (with a huge amount of assets), any change needs to be deemed absolutely necessary. In this particular case, the team prefers not modifying the code as the compiler-generated bounds-checking is already in place.

3.5 Incompatibility With Deflationary Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: MasterChef
- Category: Business Logics [10]
- CWE subcategory: CWE-708 [5]

Description

In SushiSwap, the MasterChef contract operates as the main entry for interaction with staking users. The staking users deposit UniswapV2's LP tokens into the SushiSwap 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 SushiSwap protocol. Using the deposit() function as an example, it needs to transfer deposited assets from the user account to the pool (line 210). 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 211 – 212).

```
201    // Deposit LP tokens to MasterChef for SUSHI allocation.
202    function deposit(uint256 _pid, uint256 _amount) public {
203        PoolInfo storage pool = poolInfo[_pid];
204        UserInfo storage user = userInfo[_pid][msg.sender];
```

```
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accSushiPerShare).div(1e12).sub(user.rewardDebt);
            safeSushiTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).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 `SushiSwap`. 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 `SushiSwap`'s similar `swap()`, is indeed not compatible with deflationary tokens.

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

Status This issue has been confirmed. As there is a central place to regulate the assets that can be introduction in the pool management, the team decides no change for the time being.

3.6 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [22] exploit, and the recent `Uniswap/Lendf.Me` hack [20].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `MasterChef` as an example, the `emergencyWithdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 234) starts before effecting the update on internal states (lines 236 – 237), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching `re-entrancy` via the very same `emergencyWithdraw()` function.

```
230     // Withdraw without caring about rewards. EMERGENCY ONLY.
231     function emergencyWithdraw(uint256 _pid) public {
232         PoolInfo storage pool = poolInfo[_pid];
233         UserInfo storage user = userInfo[_pid][msg.sender];
234         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
235         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
236         user.amount = 0;
237         user.rewardDebt = 0;
238     }
```

Listing 3.12: MasterChef.sol

Another similar violation can be found in the `deposit()` and `withdraw()` routines within the same contract.

```
    // Deposit LP tokens to MasterChef for SUSHI 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.accSushiPerShare).div(1e12).sub(user.rewardDebt);
            safeSushiTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accSushiPerShare).div(1e12).sub(user.rewardDebt);
        safeSushiTransfer(msg.sender, pending);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(1e12);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.13: MasterChef.sol

In the meantime, we should mention that the UniswapV2's LP tokens implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions best practice. The above three functions can be revised as follows:

```
    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 _amount=user.amount
        user.amount = 0;
        user.rewardDebt = 0;
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit EmergencyWithdraw(msg.sender, _pid, _amount);
    }

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

        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accSushiPerShare).div(1e12).sub(user.rewardDebt);

        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(1e12);


        safeSushiTransfer(msg.sender, pending);
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        emit Deposit(msg.sender, _pid, _amount);
    }


    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accSushiPerShare).div(1e12).sub(user.rewardDebt);

        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(1e12);

        safeSushiTransfer(msg.sender, pending);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.14: MasterChef.sol (revised)

Status This issue has been confirmed. Due to the same reason as outlined in Section 3.3, the team prefers not modifying the live code and leaves the code as it is.

3.7 Improved Logic in getMultiplier()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Status Codes [12]
- CWE subcategory: CWE-682 [4]

Description

SushiSwap 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 SUSHI tokens minted (more in Section 3.12) 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 SUSHI tokens produced will be multiplied by 10, resulting in 1,000 SUSHI tokens (again more in Section 3.12) 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., pendingSushi() 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) {
            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 3.16: MasterChef.sol

Status This issue has been confirmed. Due to the same reason as outlined in Section 3.3, the team prefers not modifying the live code and leaves the implementation as it is. As discussed earlier, the current callers provide the arguments that have been similarly verified to always obtain correct reward multipliers. Meanwhile, the team has been informed about possible misleading results as external inquiries on the `getMultiplier()` routine may provide arbitrary arguments that do not take into account the initial block, i.e., `startBlock`.

3.8 Improved EOA Detection Against Front-Running of Revenue Conversion

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `SushiMaker`
- Category: Status Codes [12]
- CWE subcategory: CWE-682 [4]

Description

SushiSwap has a rather unique tokenomics around `SUSHI` tokens. In this section, we explore the logic behind `SushiMaker` and `SushiBar`. `SushiMaker` collects possible revenues (in terms of `SushiSwap` pairs' LP tokens), convert collected revenues into `SUSHI` tokens, and then send them to `SushiBar`. `SUSHI` holders can stake their `SUSHI` assets to `SushiBar` to earn more `SUSHI`.

```
    function convert(address token0, address token1) public {
        // At least we try to make front-running harder to do.
        require(!Address.isContract(msg.sender), "do not convert from contract");
        IUniswapV2Pair pair = IUniswapV2Pair(factory.getPair(token0, token1));
        pair.transfer(address(pair), pair.balanceOf(address(this)));
        pair.burn(address(this));
        uint256 wethAmount = _toWETH(token0) + _toWETH(token1);
        _toSUSHI(wethAmount);
    }
```

Listing 3.17: SushiMaker.sol

The conversion of collected revenues into `SUSHI` is implemented in `convert()`. Due to possible revenues into `SushiMaker`, this routine could be a target for front-running (and further facilitated by flash loans) to steal the majority of collected revenues, resulting in a loss for current stakers in `SushiBar`.

As a defense mechanism, `SushiMaker` takes a pro-active measure by only allowing EOA accounts when the revenues are being converted. The detection of whether the transaction sender is an EOA or contract is based on the `isContract()` routine borrowed from the `Address` library (shown below).

```
    /**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // This method relies in extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        // solhint-disable-next-line no-inline-assembly
        assembly { size := extcodesize(account) }
        return size > 0;
    }
```

Listing 3.18: Address.sol

The current `isContract()` could achieve its goal in most cases. However, as mentioned in the library documentation, *"it is unsafe to assume that an address for which this function returns false is an externally-owned account (EOA) and not a contract."* Considering the specific context `SushiMaker`, we need a reliable method to detect the `convert()` transaction sender is an externally-owned account, i.e., `EOA`. With that, we can simply achieve our goal by `require(msg.sender==tx.origin, "do not convert from contract")`.

Recommendation Apply the improved detection logic in the `convert()` routine as follows.

```
    function convert(address token0, address token1) public {
        // At least we try to make front-running harder to do.
        require(msg.sender==tx.origin, "do not convert from contract");
        IUniswapV2Pair pair = IUniswapV2Pair(factory.getPair(token0, token1));
        pair.transfer(address(pair), pair.balanceOf(address(this)));
        pair.burn(address(this));
        uint256 wethAmount = _toWETH(token0) + _toWETH(token1);
        _toSUSHI(wethAmount);
    }
```

Listing 3.19: SushiMaker.sol

Status This issue has been confirmed and accordingly fixed by this commit: `84243d745ed68d76c85964eb4a160211cec`
.

3.9 No Pair Creation With Zero Migration Balance

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Migrator`
- Category: Business Logics [10]
- CWE subcategory: CWE-770 [6]

Description

As discussed in Section 3.1, the actual bulk work of migration is performed by the `Migrator` contract, specifically its `migrate()` routine (we show the related code snippet below). This specific routine basically burns the `UniswapV2`'s LP tokens to reclaim the underlying assets and then transfers them to `SushiSwap` for the minting of the corresponding new pair's LP tokens.

```
26     function migrate(IUniswapV2Pair orig) public returns (IUniswapV2Pair) {
27         require(msg.sender == chef, "not from master chef");
28         require(block.number >= notBeforeBlock, "too early to migrate");
29         require(orig.factory() == oldFactory, "not from old factory");
30         address token0 = orig.token0();
31         address token1 = orig.token1();
32         IUniswapV2Pair pair = IUniswapV2Pair(factory.getPair(token0, token1));
33         if (pair == IUniswapV2Pair(address(0))) {
34             pair = IUniswapV2Pair(factory.createPair(token0, token1));
35         }
36         uint256 lp = orig.balanceOf(msg.sender);
37         if (lp == 0) return pair;
38         desiredLiquidity = lp;
39         orig.transferFrom(msg.sender, address(orig), lp);
40         orig.burn(address(pair));
41         pair.mint(msg.sender);
42         desiredLiquidity = uint256(-1);
43         return pair;
44     }
```

Listing 3.20: Migrator.sol

In the unlikely situation when the migrated pool does have any balance for migration, `migrate()` routine is expected to simply return. However, it is interesting to notice that the `return` (line 37) does not happen until the new SushiSwap pair is created. As the SushiSwap DEX is based on the UniswapV2, a new pair creation may cost more than $2,000,000$ gas. Considering the current congested Ethereum blockchain and the relatively prohibitive gas cost, it is inappropriate to spend the gas cost to create a new pair when the balance is zero and no migration actually occurs.

Recommendation Move the balance detection logic earlier so that we can simply return without migration and new pair creation if the balance is zero , i.e., orig.balanceOf(msg.sender)== 0. An example adjustment is shown below.

```
    function migrate(IUniswapV2Pair orig) public returns (IUniswapV2Pair) {
        require(msg.sender == chef, "not from master chef");
        require(block.number >= notBeforeBlock, "too early to migrate");

        uint256 lp = orig.balanceOf(msg.sender);
        if (lp == 0) return pair;

        require(orig.factory() == oldFactory, "not from old factory");
        address token0 = orig.token0();
        address token1 = orig.token1();
        IUniswapV2Pair pair = IUniswapV2Pair(factory.getPair(token0, token1));
        if (pair == IUniswapV2Pair(address(0))) {
            pair = IUniswapV2Pair(factory.createPair(token0, token1));
        }
        orig.transferFrom(msg.sender, address(orig), lp);
        orig.burn(address(pair));

        desiredLiquidity = lp;
        pair.mint(msg.sender);
        desiredLiquidity = uint256(-1);
        return pair;
    }
```

Listing 3.21: Migrator.sol

Status This issue has been confirmed.

3.10 Full Charge of Proposal Execution Cost From Accompanying msg.value

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: GovernorAlpha
- Category: Business Logics [10]
- CWE subcategory: CWE-770 [6]

Description

Sushi adopts the governance implementation from Compound by adjusting its governance token and related parameters, e.g., quorumVotes() and proposalThreshold(). The original governance has been successfully audited by OpenZeppelin.

In the following, we would like to comment on a particular issue regarding the proposal execution cost. Notice that the actual proposal execution is kicked off by invoking the governance's execute() function. This function is marked as payable, indicating the transaction sender is responsible for supplying required amount of ETHs as each inherent action (line 215) in the proposal may require accompanying certain ETHs, specified in proposal.values[i], where i is the i^{th} action inside the proposal.

```
210     function execute(uint proposalId) public payable {
211         require(state(proposalId) == ProposalState.Queued, "GovernorAlpha::execute:
                proposal can only be executed if it is queued");
212         Proposal storage proposal = proposals[proposalId];
213         proposal.executed = true;
214         for (uint i = 0; i < proposal.targets.length; i++) {
215             timelock.executeTransaction.value(proposal.values[i])(proposal.targets[i],
                    proposal.values[i], proposal.signatures[i], proposal.calldatas[i],
                    proposal.eta);
216         }
217         emit ProposalExecuted(proposalId);
218     }
```

Listing 3.22: GovernorAlpha.sol

Though it is likely the case that a majority of these actions do not require any ETHs, i.e., proposal.values[i] = 0, we may be less concerned on the payment of required ETHs for the proposal execution. However, in the unlikely case of certain particular actions that do need ETHs, the issue of properly attributing the associated cost arises. With that, we need to better keep track of ETH charge for each action and ensure that the transaction sender (who initiates the proposal execution) actually pays the cost. In other words, we do not rely on the governance's balance of ETH for the payment.

Recommendation Properly charge the proposal execution cost by ensuring the amount of accompanying ETH deposit is sufficient. If necessary, we can also return possible leftover in `msgValue` back to the sender.

```
    function execute(uint proposalId) public payable {
        require(state(proposalId) == ProposalState.Queued, "GovernorAlpha::execute:
            proposal can only be executed if it is queued");
        Proposal storage proposal = proposals[proposalId];
        proposal.executed = true;
        uint msgValue = msg.value;
        for (uint i = 0; i < proposal.targets.length; i++) {
            inValue = sub256(msgValue, proposal.values[i])
            timelock.executeTransaction.value(proposal.values[i])(proposal.targets[i],
                proposal.values[i], proposal.signatures[i], proposal.calldatas[i],
                proposal.eta);
        }
        emit ProposalExecuted(proposalId);
    }
```

Listing 3.23: GovernorAlpha.sol

Status This issue has been confirmed.

3.11 Improved Handling of Corner Cases in Proposal Submission

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `GovernorAlpha`
- Category: Business Logics [10]
- CWE subcategory: CWE-837 [7]

Description

As discussed in Section 3.10, `Sushi` adopts the governance implementation from `Compound` by accordingly adjusting its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold()`. Previously, we have examined the payment of proposal execution cost. In this section, we elaborate one corner case during a proposal submission, especially regarding the proposer qualification.

To be qualified to be proposer, the governance subsystem requires the proposer needs to obtain a sufficient number of votes, including from the proposer herself and other voters. The threshold is specified by `proposalThreshold()`. In `SushiSwap`, this number requires the votes of 1% of `SUSHI` token's total supply, i.e., `SushiToken(sushi).totalSupply()`.

```
    function propose(address[] memory targets, uint[] memory values, string[] memory
        signatures, bytes[] memory calldatas, string memory description) public returns
        (uint) {
```

```
155         require(sushi.getPriorVotes(msg.sender, sub256(block.number, 1)) > proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal threshold");
156         require(targets.length == values.length && targets.length == signatures.length && targets.length == calldatas.length, "GovernorAlpha::propose: proposal function information arity mismatch");
157         require(targets.length != 0, "GovernorAlpha::propose: must provide actions");
158         require(targets.length <= proposalMaxOperations(), "GovernorAlpha::propose: too many actions");

160         uint latestProposalId = latestProposalIds[msg.sender];
161         if (latestProposalId != 0) {
162           ProposalState proposersLatestProposalState = state(latestProposalId);
163           require(proposersLatestProposalState != ProposalState.Active, "GovernorAlpha::propose: one live proposal per proposer, found an already active proposal");
164           require(proposersLatestProposalState != ProposalState.Pending, "GovernorAlpha::propose: one live proposal per proposer, found an already pending proposal");
165         }
166         ...
167     }
```

Listing 3.24: GovernorAlpha.sol

If we examine the `propose()` logic, when a proposal is being submitted, the governance verifies upfront the qualification of the proposer (line 155): `require(sushi.getPriorVotes(msg.sender, sub256(block.number, 1))> proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal threshold")`. Notice that the number of prior votes is strictly higher than `proposalThreshold()`.

However, if we check the proposal cancellation logic, i.e., the `cancel()` function, a proposal can be canceled (line 225) if the number of prior votes (before current block) is strictly smaller than `proposalThreshold()`. The corner case of having an exact number prior votes as the threshold, though unlikely, is largely unattended. It is suggested to accommodate this particular corner case as well.

```
220     function cancel(uint proposalId) public {
221         ProposalState state = state(proposalId);
222         require(state != ProposalState.Executed, "GovernorAlpha::cancel: cannot cancel executed proposal");

224         Proposal storage proposal = proposals[proposalId];
225         require(msg.sender == guardian || sushi.getPriorVotes(proposal.proposer, sub256(block.number, 1)) < proposalThreshold(), "GovernorAlpha::cancel: proposer above threshold");

227         proposal.canceled = true;
228         for (uint i = 0; i < proposal.targets.length; i++) {
229             timelock.cancelTransaction(proposal.targets[i], proposal.values[i], proposal.signatures[i], proposal.calldatas[i], proposal.eta);
230         }
```

```
        emit ProposalCanceled(proposalId);
    }
```

Listing 3.25: GovernorAlpha.sol

Recommendation Accommodate the corner case by also allowing the proposal to be successfully submitted when the number of proposer's prior votes is exactly the same as the required threshold, i.e., `proposalThreshold()`.

```
    function propose(address[] memory targets, uint[] memory values, string[] memory signatures, bytes[] memory calldatas, string memory description) public returns (uint) {
        require(sushi.getPriorVotes(msg.sender, sub256(block.number, 1)) >= proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal threshold");
        require(targets.length == values.length && targets.length == signatures.length && targets.length == calldatas.length, "GovernorAlpha::propose: proposal function information arity mismatch");
        require(targets.length != 0, "GovernorAlpha::propose: must provide actions");
        require(targets.length <= proposalMaxOperations(), "GovernorAlpha::propose: too many actions");

        uint latestProposalId = latestProposalIds[msg.sender];
        if (latestProposalId != 0) {
          ProposalState proposersLatestProposalState = state(latestProposalId);
          require(proposersLatestProposalState != ProposalState.Active, "GovernorAlpha::propose: one live proposal per proposer, found an already active proposal");
          require(proposersLatestProposalState != ProposalState.Pending, "GovernorAlpha::propose: one live proposal per proposer, found an already pending proposal");
        }
        ...
    }
```

Listing 3.26: GovernorAlpha.sol

Status This issue has been confirmed.

3.12 Inconsistency Between Documented and Implemented SUSHI Inflation

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MasterChef
- Category: Business Logics [10]
- CWE subcategory: CWE-837 [7]

Description

According to the documentation of SushiSwap [24], *"At every block, 100 SUSHI tokens will be created. These tokens will be equally distributed to the stakers of each of the supported pools."*

As part of the audit process, we examine and identify possible inconsistency between the documentation/white paper and the implementation. Based on the smart contract code, there is a system-wide configuration, i.e., `sushiPerBlock`. This particular parameter is initialized as 100 when the contract is being deployed and it can only be changed at the contract's constructor. The initialized number of 100 seems consistent with the documentation and `sushiPerBlock` is fixed forever (and cannot be adjusted even via a governance process).

A further analysis about the `SUSHI` inflation logic (implemented in `updatePool()`) shows certain inconsistency that needs to better articulated and clarified. For elaboration, we show the related code snippet below.

```
    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 sushiReward = multiplier.mul(sushiPerBlock).mul(pool.allocPoint).div(
            totalAllocPoint);
        sushi.mint(devaddr, sushiReward.div(10));
        sushi.mint(address(this), sushiReward);
        pool.accSushiPerShare = pool.accSushiPerShare.add(sushiReward.mul(1e12).div(
            lpSupply));
        pool.lastRewardBlock = block.number;
    }
```

Listing 3.27: MasterChef.sol

The sushiPerBlock parameter indeed controls the number of SUSHI rewards that are distributed to various pools (line 196). However, it further adds another 10% of the calculated sushiReward to the development team-controlled account (line 195). With that, the number of new SUSHI rewards per block should be 110, not 100!

Recommendation Clarify the inconsistency by clearly stating the number of new SUSHI tokens is 110, and the development team will be receiving about $1/11 = 9.09\%$ of total SUSHI distribution.

Status This issue has been confirmed.

3.13 Non-Governance-Based Admin of TimeLock And Related Privileges

- ID: PVE-013
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Timelock
- Category: Security Features [9]
- CWE subcategory: CWE-287 [2]

Description

In SushiSwap, the governance contract, i.e., GovernorAlpha, plays a critical role in governing and regulating the system-wide operations (e.g., pool addition, reward adjustment, and migrator setting). It also has the privilege to control or govern the life-cycle of proposals and enact on them regarding their submissions, executions, and revocations.

With great privilege comes great responsibility. Our analysis shows that the governance contract is indeed privileged, but it currently has NOT been deployed yet to govern the MasterChef contract that is the central to SushiSwap. In the following, we examine the current state of privilege assignment in SushiSwap.

Specifically, we kept track of the current deployment of various contracts in SushiSwap and the results are shown in Table 3.1.

Table 3.1: Current Contract Deployment of SushiSwap

Contract	Address	Owner/Admin
SUSHIToken	0x6b3595068778dd592e39a122f4f5a5cf09c90fe2	0xc2edad668740f1aa35e4d8f227fb8e17dca888cd
MasterChef	0xc2edad668740f1aa35e4d8f227fb8e17dca888cd	0x9a8541ddf3a932a9a922b607e9cf7301f1d47bd1
Timelock	0x9a8541ddf3a932a9a922b607e9cf7301f1d47bd1	0xf942dba4159cb61f8ad88ca4a83f5204e8f4a6bd
Deployer/DevAddr	0xf942dba4159cb61f8ad88ca4a83f5204e8f4a6bd	
Migrator	0x0000000000000000000000000000000000000000	

To further elaborate, we draw the admin chain based on the current deployment of SushiSwap in Figure 3.2. We emphasize that the SUSHI token contract is properly administrated by the MasterChef contract that is authorized to mint new SUSHI tokens per block. The MasterChef contract is administrated by the Timelock contract and this administration is also appropriate as the Timelock contract is indeed authorized to configure various aspects of MasterChef, including the addition of new pools, the share adjustment of each existing pool (if necessary), and the setting of the upcoming migrator contract.

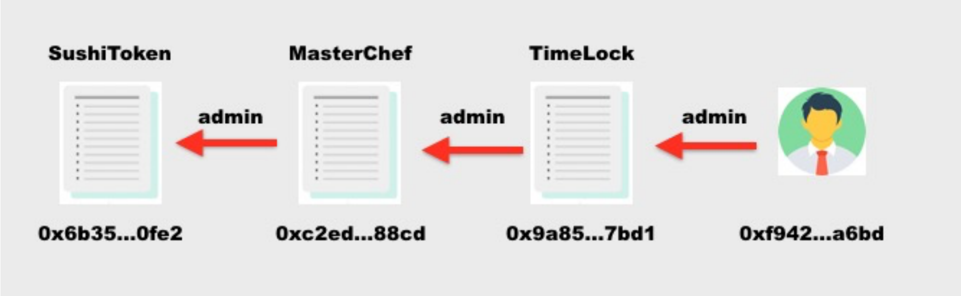

Figure 3.2: The Current Admin Chain of SushiSwap

However, it is worrisome that Timelock is not governed by the GovernorAlpha governance contract. Our analysis shows that the current Timelock control is controlled by an externally-owned account (EOA) address, i.e., 0xf942dba4159cb61f8ad88ca4a83f5204e8f4a6bd. This EOA address happens to be the same deployer address of SushiSwap and also configured as the development team address, i.e., devaddr. With a proper community-based on-chain governance, its admin chain should be depicted as follows:

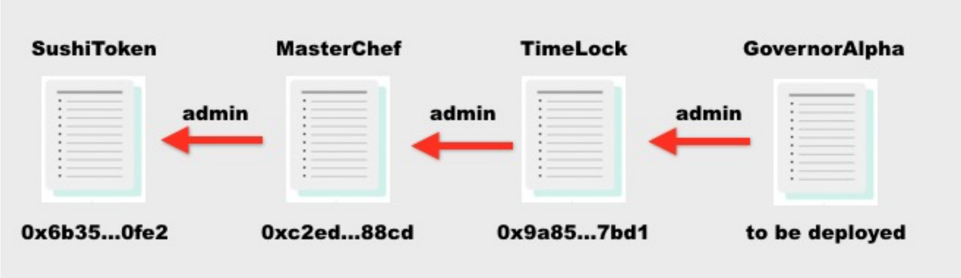

Figure 3.3: The Expected Admin Chain of SushiSwap

In the meantime, we notice the GovernorAlpha contract has a special guardian that has certain privilege, including the cancellation of ongoing proposals that has not been executed yet. However, since this contract has not been deployed and this part of logic is directly borrowed from Compound without any modification, we do not expand further.[1]

[1] Interested readers are referred to the original GovernorAlpha audit report conducted by OpenZeppelin and the

Recommendation Promptly transfer the `admin` privilege of `Timelock` to the intended `GovernorAlpha` governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed.

3.14 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;`.

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 SushiSwap design and implementation. Overall, SushiSwap 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 and those identified issues are promptly confirmed and fixed. The main concern, however, is related to the current deployment as its privilege management is not under the control of community-based governance.

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 [15, 16, 17, 18, 21].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [23] 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` exception.
- 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()` [1], 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] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[16] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[17] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[18] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

[19] PeckShield. PeckShield Inc. https://www.peckshield.com.

[20] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[21] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

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

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

[24] SushiSwap Team. The SushiSwap Project: An Evolution of Uniswap With SUSHI Tokenomics. https://medium.com/sushiswap/the-sushiswap-project-c4049ea9941e.