0.7

SushiSwap 0.5 Process Quality Review

Score: 57%

This is a SushiSwap Process Quality Audit completed on 1 September 2020. Based on improvements by SushiSwap, a revision of the report was conducted on 22 September. Revised sections will have REVISED and the old comments in italics.

It was performed using the Process Audit process (version 0.5) and is documented here. The audit was performed by ShinkaRex of Caliburn Consulting. Check out our Telegram.

The revised final score of the audit is 57%, much improved from the previous 19%. The breakdown of the scoring is in Scoring Appendix.

Summary of the Process

Very simply, the audit looks for the following declarations from the developer's site. With these declarations, it is reasonable to trust the smart contracts.

1. **Here is my smart contract on the blockchain**
2. **You can see it matches a software repository used to develop the code**
3. **Here is the documentation that explains what my smart contract does**
4. **Here are the tests I ran to verify my smart contract**
5. **Here are the audit(s) performed to review my code by third party experts**

Disclaimer

Executing Code Verification

This section looks at the code deployed on the Mainnet that gets audited and its corresponding software repository. The document explaining these questions is here. This audit will answer the questions;

1. Is the executing code address(s) readily available? (Y/N)
2. Is the code actively being used? (%)
3. Are the Contract(s) Verified/Verifiable? (Y/N)
4. Does the code match a tagged version in the code hosting platform? (%)
5. Is the software repository healthy? (%)

Is the executing code address(s) readily available? (Y/N)

> ✓ Answer: Yes

REVISED: New text; All smart contract addresses are in the Medium article.

Of the eight or more contracts in the GitHub (counting Uniswap as one), only one contract address is listed in the medium article. I found the address for the sushi token contract by analyzing the master chef contract. All the other addresses are unknown.

They are available at Address 0xc2EdaD668740f1aA35E4D8f227fB8E17dcA888Cd as indicated in the Appendix. This Audit only covers the contract *MasterChef*.

Is the code actively being used? (%)

> ✓ Answer: 100&

Activity is 15k transactions a day, as indicated in the Appendix.

Percentage Score Guidance

100%	More than 10 transactions a day
70%	More than 10 transactions a week
40%	More than 10 transactions a month
10%	Less than 10 transactions a month
0%	No activity

Are the Contract(s) Verified/Verifiable? (Y/N)

Answer: Yes

0xc2EdaD668740f1aA35E4D8f227fB8E17dcA888Cd is the Etherscan verified contract address.

How to improve this score

Ensure that the deployed code is verified as described in this article for Etherscan or ETHPM. Improving this score may require redeployment.

Does the code match a tagged version on a code hosting platform? (%)

Answer: 60%

REVISED: New text: Of the contracts found, the sushi token and master chef contracts matched. Sushi updated the contracts since last review and the files matched.

Of the contracts found, the sushi token and master chef contracts matched. However the contracts on the main for the other contracts, the majority, were not found. For this reason a 0% score is given.

Guidance:

100%	All code matches and Repository was clearly labelled
60 %	All code matches but no labelled repository. Repository was found manually
30%	Almost all code does match perfectly and repository was found manually
0%	Most matching Code could not be found

GitHub address : https://github.com/sushiswap/sushiswap

Deployed contracts in the following file;

Sushi2_Deployed.rar 39KB
Binary

Matching Repository: https://github.com/sushiswap/sushiswap/tree/master/contracts

How to improve this score

Ensure there is a clearly labelled repository holding all the contracts, documentation and tests for the deployed code. Ensure an appropriately labeled tag exists corresponding to deployment dates. Release tags are clearly communicated.

Is development software repository healthy? (%)

⚠ Answer: 0%

The repository was created on August 23 and the master chef contract was deployed on August 26. It only had 48 commits and one branch when audited on 1 September. This is not a healthy repo.

How to improve this score

Ensure there is a clearly labelled repository holding all the contracts, documentation and tests for the deployed code. Continue to test and perform other verification activities after deployment, including routine maintenance updating to new releases of testing and deployment tools.

Documentation

This section looks at the software documentation. The document explaining these questions is here.

Required questions are;

1. Is there a whitepaper? (Y/N)
2. Are the basic application requirements documented? (Y/N)
3. Do the requirements fully (100%) cover the deployed contracts? (%)
4. Are there sufficiently detailed comments for all functions within the deployed contract code (%)
5. Is it possible to trace software requirements to the implementation in code (%)

Is there a whitepaper? (Y/N)

✓ Answer: Yes

This article fulfills the requirements for a white paper.

Location: https://medium.com/sushiswap/the-sushiswap-project-c4049ea9941e

Are the basic application requirements documented? (Y/N)

⚠ Answer: No

No documentation on code requirements are evident.

How to improve this score

Write the document based on the deployed code. For guidance, refer to the SecurEth System Description Document.

Do the requirements fully (100%) cover the deployed contracts? (%)

> ⚠ Answer: 0%

With no documentation on code requirements, obviously zero coverage is the result.

How to improve this score

This score can improve by adding content to the requirements document such that it comprehensively covers the requirements. For guidance, refer to the SecurEth System Description Document . Using tools that aid traceability detection will help.

Are there sufficiently detailed comments for all functions within the deployed contract code (%)

> ⓘ Answer: 40%

Good commenting in Masterchef, but other contracts quite light

Code examples are in the Appendix. As per the SLOC, there is 24% commenting to code.

How to improve this score

This score can improve by adding comments to the deployed code such that it comprehensively covers the code. For guidance, refer to the SecurEth Software Requirements.

Is it possible to trace requirements to the implementation in code (%)

> ⚠ Answer: 0%

With no documentation on code requirements, obviously zero traceability is the result.

Guidance:
100% - Clear explicit traceability between code and documentation at a requirement level for all code
60% - Clear association between code and documents via non explicit traceability
40% - Documentation lists all the functions and describes their functions
0% - No connection between documentation and code

How to improve this score

This score can improve by adding traceability from requirements to code such that it is clear where each

requirement is coded. For reference, check the SecurEth guidelines on traceability.

Testing

This section looks at the software testing available. It is explained in this document. This section answers the following questions;

1. Full test suite (Covers all the deployed code) (%)
2. Code coverage (Covers all the deployed lines of code, or explains misses) (%)
3. Scripts and instructions to run the tests (Y/N)
4. Packaged with the deployed code (Y/N)
5. Report of the results (%)
6. Formal Verification test done (%)
7. Stress Testing environment (%)

Is there a Full test suite? (%)

> Answer: 100%

Test suite looks good.

Code coverage (Covers all the deployed lines of code, or explains misses) (%)

> Answer: 50%

No indication of coverage results.

Guidance:
100% - Documented full coverage
99-51% - Value of test coverage from documented results
50% - No indication of code coverage but clearly there is a reasonably complete set of tests
30% - Some tests evident but not complete
0% - No test for coverage seen

How to improve this score

This score can improve by adding tests achieving full code coverage. A clear report and scripts in the software repository will guarantee a high score.

Scripts and instructions to run the tests (Y/N)

⚠ Answer: No

No instructions visible.

How to improve this score

Add the scripts to the repository and ensure they work. Ask an outsider to create the environment and run the tests. Improve the scripts and docs based on their feedback.

Packaged with the deployed code (Y/N)

✓ Answer: Yes

Report of the results (%)

⚠ Answer: 0%

No report of test results visible.

How to improve this score

Add a report with the results. The test scripts should generate the report or elements of it.

Formal Verification test done (%)

⚠ Answer: 0%

No formal verification visible.

Stress Testing environment (%)

⚠ Answer: 0%

No evidence of testnets. They might have been there, but nothing is shown.

Audits

> Answer: 70%

REVISED: New Text. Two audits were performed after deployment from Quantstamp and PeckShield. Some changes were implemented while others will be dealt with by the team without contract modification as the contracts are in heavy use. No critical flaws found. Score as per below 70%.

OLD: Contracts addresses on mainnet not available, so 0%, even if an audit was done. No audit done.

Guidance:

1. Multiple Audits performed before deployment and results public and implemented or not required (100%)
2. Single audit performed before deployment and results public and implemented or not required (90%)
3. Audit(s) performed after deployment and no changes required. Audit report is public. (70%)
4. No audit performed (20%)
5. Audit Performed after deployment, existence is public, report is not public and no improvements deployed OR smart contract address' not found, question 1 (0%)

Appendices

Author Details

The author of this audit is Rex of Caliburn Consulting.

Email : rex@defisafety.com Twitter : @defisafety

I started with Ethereum just before the DAO and that was a wonderful education. It showed the importance of code quality. The second Parity hack also showed the importance of good process. Here my aviation background offers some value. Aerospace knows how to make reliable code using quality processes.

I was coaxed to go to EthDenver 2018 and there I started SecuEth.org with Bryant and Roman. We created guidelines on good processes for blockchain code development. We got EthFoundation funding to assist in their development.

Process Quality Audits are an extension of the SecurEth guidelines that will further increase the quality processes in Solidity and Vyper development.

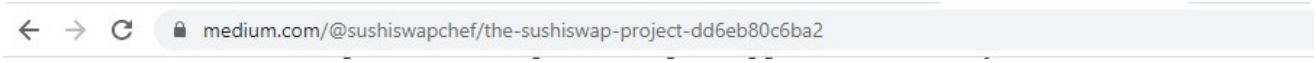
Career wise I am a business development manager for an avionics supplier.

Scoring Appendix

	Total	SushiSwap	
PQ Audit Scoring Matrix (v0.4 and 0.5)			

PQ Audit Scoring Matrix (v0.4 and 0.5)	Points	Answer	Points
Total	240		137.5
Executing Code Verification			**57%**
1. Is the executing code address(s) readily available? (Y/N)	30	Y	30
2. Is the code actively being used? (%)	5	100%	5
3. Are the Contract(s) Verified/Verifiable? (Y/N)	5	Y	5
4. Does the code match a tagged version on a code hosting platform? (%)	20	60%	12
5. Is development software repository healthy? (%)	10	0%	0
Code Documentation			
1. Is there a whitepaper? (Y/N)	5	Y	5
2. Are the basic application requirements documented? (Y/N)	10	N	0
3. Do the requirements fully (100%) cover the deployed contracts? (%)	15	0%	0
4. Are there sufficiently detailed comments for all functions within the deployed contract code (%)	10	40%	4
5. Is it possible to trace requirements to the implementation in code (%)	5	0%	0
Testing			
1. Full test suite (Covers all the deployed code) (%)	20	100%	20
2. Code coverage (Covers all the deployed lines of code, or explains misses) (%)	5	50%	2.5
3. Scripts and instructions to run the tests? (Y/N)	5	N	0
4. Packaged with the deployed code (Y/N)	5	Y	5
5. Report of the results (%)	10	0%	0
6. Formal Verification test done (%)	5	0%	0
7. Stress Testing environment (%)	5	0%	0
Audits			
Audit done	70	70%	49
Section Scoring			
Executing Code Verification	70	74%	
Documentation	45	20%	
Testing	55	50%	
Audits	70	70%	

Executing Code Appendix



licenses. The followings are the list of the contracts with rough explanation:

🍣 **SushiToken**: The token contract, with COMP/YAM voting functionality.
👨‍🍳 **MasterChef**: Deposit LPs tokens to farm SUSHI.
🔪 **SushiMaker**: Collect revenues, convert to SUSHI, and send to SushiBar.
🍸 **SushiBar**: Stake SUSHI to earn more SUSHI 📈.
🏭 **Migrator**: Migrate MasterChef LP tokens from Uniswap to SushiSwap.
🏛 **GovernorAlpha** + **Timelock**: Governance stuff from Compound.
🦄 **UniswapV2**: UniswapV2 contracts with small modification for Migration.

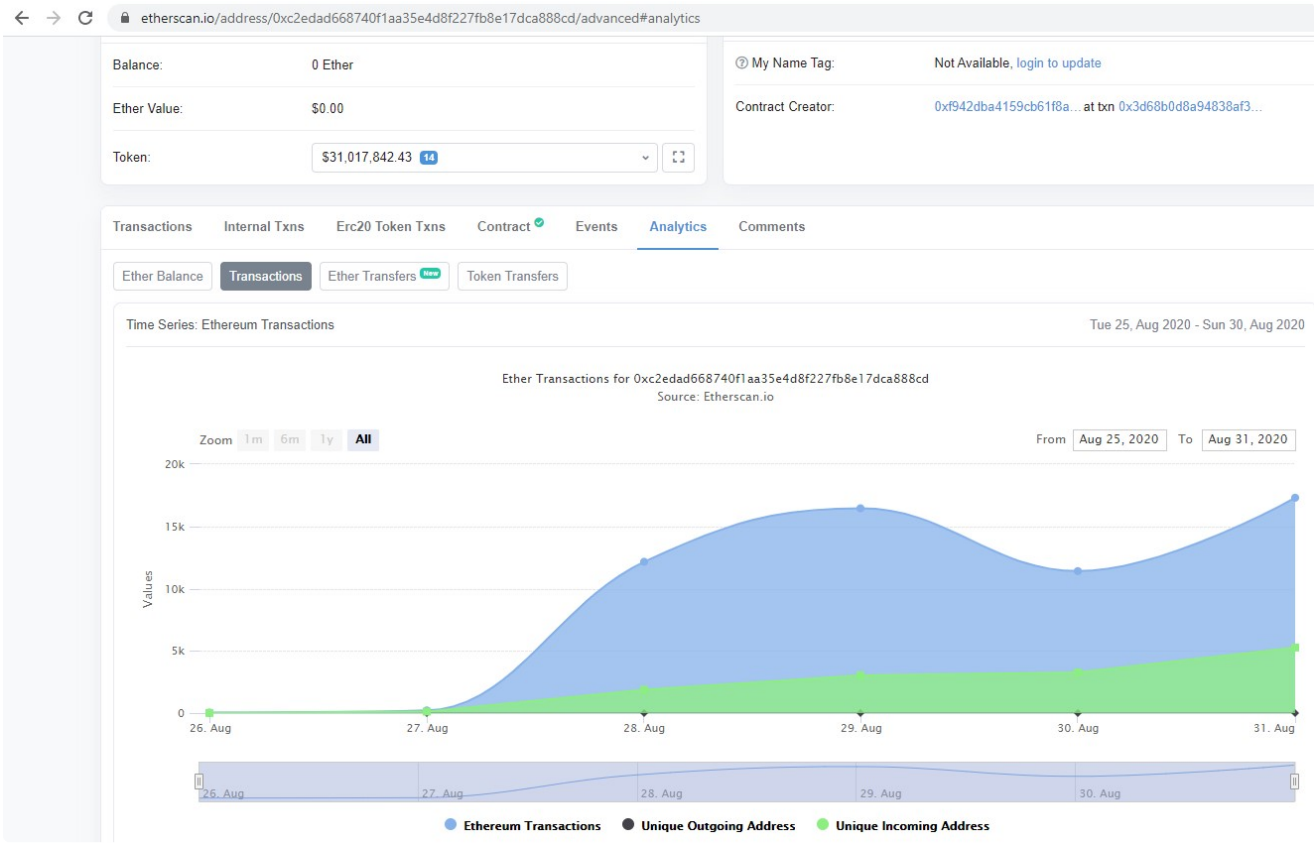
SUSHI Token 0x6B3595068778DD592e39A122f4f5a5cF09C90fE2
MasterChef 0xc2EdaD668740f1aA35E4D8f227fB8E17dcA888Cd
Factory 0xC0AEe478e3658e2610c5F7A4A2E1777cE9e4f2Ac

Router 0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F
SushiBar 0x8798249c2E607446EfB7Ad49eC89dD1865Ff4272
SushiMaker 0x6684977bBED67e101BB80Fc07fCcfba655c0a64F

PeckShield :
https://github.com/peckshield/publications/blob/master/audit_reports/PeckShield-Audit-Report-SushiSwap-v1.0.pdf

Quantstamp:
https://github.com/quantstamp/sushiswap-security-review

Code Used Appendix



Example Code Appendix

```
pragma solidity 0.6.12;


import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/utils/EnumerableSet.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
```

```
import "./SushiToken.sol";


interface IMigratorChef {
    // Perform LP token migration from legacy UniswapV2 to SushiSwap.
    // Take the current LP token address and return the new LP token address.
    // Migrator should have full access to the caller's LP token.
    // Return the new LP token address.
    //
    // XXX Migrator must have allowance access to UniswapV2 LP tokens.
    // SushiSwap must mint EXACTLY the same amount of SushiSwap LP tokens or
    // else something bad will happen. Traditional UniswapV2 does not
    // do that so be careful!
    function migrate(IERC20 token) external returns (IERC20);
}

// MasterChef is the master of Sushi. He can make Sushi and he is a fair guy.
//
// Note that it's ownable and the owner wields tremendous power. The ownership
// will be transferred to a governance smart contract once SUSHI is sufficiently
// distributed and the community can show to govern itself.
//
// Have fun reading it. Hopefully it's bug-free. God bless.
contract MasterChef is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    // Info of each user.
    struct UserInfo {
        uint256 amount;     // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt. See explanation below.
        //
        // We do some fancy math here. Basically, any point in time, the amount of SUSHIs
        // entitled to a user but is pending to be distributed is:
        //
        //   pending reward = (user.amount * pool.accSushiPerShare) - user.rewardDebt
        //
        // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
        //   1. The pool's `accSushiPerShare` (and `lastRewardBlock`) gets updated.
        //   2. User receives the pending reward sent to his/her address.
        //   3. User's `amount` gets updated.
        //   4. User's `rewardDebt` gets updated.
    }

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

    // The SUSHI TOKEN!
```

```
62      SushiToken public sushi;
63      // Dev address.
64      address public devaddr;
65      // Block number when bonus SUSHI period ends.
66      uint256 public bonusEndBlock;
67      // SUSHI tokens created per block.
68      uint256 public sushiPerBlock;
69      // Bonus muliplier for early sushi makers.
70      uint256 public constant BONUS_MULTIPLIER = 10;
71      // The migrator contract. It has a lot of power. Can only be set through governance (ow
72      IMigratorChef public migrator;
73
74      // Info of each pool.
75      PoolInfo[] public poolInfo;
76      // Info of each user that stakes LP tokens.
77      mapping (uint256 => mapping (address => UserInfo)) public userInfo;
78      // Total allocation poitns. Must be the sum of all allocation points in all pools.
79      uint256 public totalAllocPoint = 0;
80      // The block number when SUSHI mining starts.
81      uint256 public startBlock;
82
83      event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
84      event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
85      event EmergencyWithdraw(address indexed user, uint256 indexed pid, uint256 amount);
86
87      constructor(
88          SushiToken _sushi,
89          address _devaddr,
90          uint256 _sushiPerBlock,
91          uint256 _startBlock,
92          uint256 _bonusEndBlock
93      ) public {
94          sushi = _sushi;
95          devaddr = _devaddr;
96          sushiPerBlock = _sushiPerBlock;
97          bonusEndBlock = _bonusEndBlock;
98          startBlock = _startBlock;
99      }
100
101     function poolLength() external view returns (uint256) {
102         return poolInfo.length;
103     }
104
105     // Add a new lp to the pool. Can only be called by the owner.
106     // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do
107     function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
108         if (_withUpdate) {
109             massUpdatePools();
110         }
111         uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
112         totalAllocPoint = totalAllocPoint.add(_allocPoint);
113         poolInfo.push(PoolInfo({
114             lpToken: _lpToken,
```

```
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accSushiPerShare: 0
        }));
    }
```

SLOC Appendix

Solidity Contracts

Language	Files	Lines	Blanks	Comments	Code	Complex
Solidity	4	613	75	105	433	61

Comments to Code 105/ 433 = 24%

Javascript Tests

Language	Files	Lines	Blanks	Comments	Code	Complex
JavaScript	7	614	31	42	541	5

Tests to Code 541 / 433 = 125%