SMART CONTRACT AUDIT REPORT

for

Surfswap

Prepared By: Xiaomi Huang

PeckShield

July 30, 2022

Document Properties

Client	Surfswap
Title	Smart Contract Audit Report
Target	Surfswap
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 30, 2022	Jing Wang	Final Release
1.0-rc	July 29, 2022	Jing Wang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Surfswap` protocol, we outline in the report 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 version of smart contracts 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 Surfswap

`Surfswap` is a decentralized exchange with an automated market maker for the support of liquidity provision and peer-to-peer transactions on `Kava`. It is designed to provide an one-stop shop for the crypto community. The implementation is based on the popular `Uniswap-v2` protocol with customization on certain swaps as well as the associated fee. The basic information of the `Surfswap` protocol is as follows:

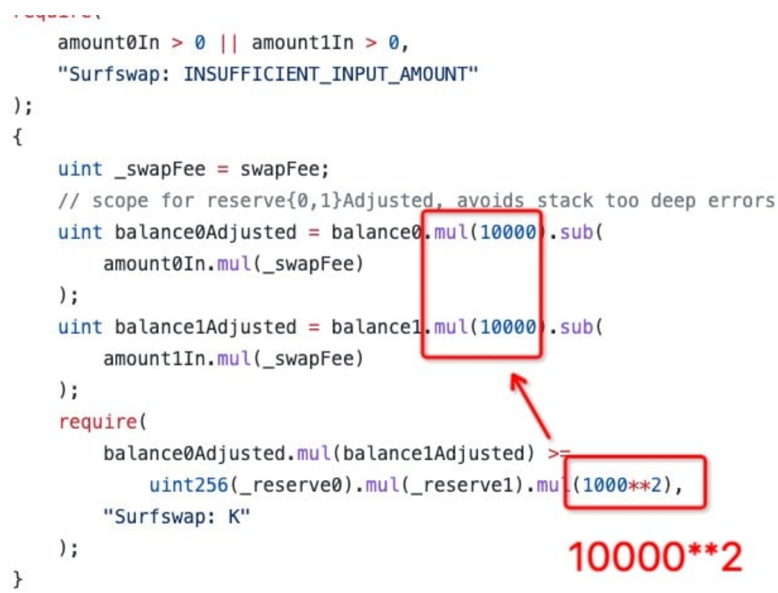
Table 1.1: Basic Information of The `Surfswap` Protocol

Item	Description
Name	Surfswap
Website	https://Surfswap.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 30, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. This audit covers the smart contracts under the specific `uniswapv2` directory.

- https://github.com/BeamSwap/surfswap-contracts/tree/main/contracts/uniswapv2 (13bf57c)

Figure 1.1: K Invariant Screenshot



```
require(
    amount0In > 0 || amount1In > 0,
    "Surfswap: INSUFFICIENT_INPUT_AMOUNT"
);
{
    uint _swapFee = swapFee;
    // scope for reserve{0,1}Adjusted, avoids stack too deep errors
    uint balance0Adjusted = balance0.mul(10000).sub(
        amount0In.mul(_swapFee)
    );
    uint balance1Adjusted = balance1.mul(10000).sub(
        amount1In.mul(_swapFee)
    );
    require(
        balance0Adjusted.mul(balance1Adjusted) >=
            uint256(_reserve0).mul(_reserve1).mul(1000**2),
        "Surfswap: K"
    );
}
```


Note the team has alrady fixed the invalid `K invariant` bug in the following commit: `13bf57c`.

1.2 About PeckShield

PeckShield Inc. [9] 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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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 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) [7], 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.

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

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.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, 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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Surfswap` implementation. During the first phase of our audit, we study the smart contract source code and run 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	0	
Medium	0	
Low	2	■ ■
Informational	2	■ ■
Total	4	

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 that 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 low-severity vulnerabilities and 2 informational recommendations.

Table 2.1: Key Surfswap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Confirmed
PVE-002	Informational	Fork Resistant Domain Separator In SurfswapERC20 Implementation	Business Logic	Confirmed
PVE-003	Informational	Potential Inconsistent Fee Calculation Between SurfswapLibrary And Surfswap-Pair	Business Logics	Confirmed
PVE-004	Low	Trust Issue of Admin Keys	Coding Practices	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SurfswapRouter
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [2]

Description

In the SurfswapRouter contract, the addLiquidity() routine (see the code snippet below) is provided to add amountADesired amount of tokenA and amountBDesired amount of tokenB into the pool as liquidity via the SurfswapRouter::addLiquidity() routine. To elaborate, we show below the related code snippet.

```
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    )
        external
        virtual
        override
        ensure(deadline)
        returns (
            uint256 amountA,
            uint256 amountB,
            uint256 liquidity
        )
    {
        (amountA, amountB) = _addLiquidity(
```

```
            tokenA,
            tokenB,
            amountADesired,
            amountBDesired,
            amountAMin,
            amountBMin
        );
        address pair = SurfswapLibrary.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = ISurfswapPair(pair).mint(to);
    }
```

Listing 3.1: SurfswapRouter::addLiquidity()

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin
    ) internal virtual returns (uint256 amountA, uint256 amountB) {
        // create the pair if it doesn't exist yet
        if (ISurfswapFactory(factory).getPair(tokenA, tokenB) == address(0)) {
            ISurfswapFactory(factory).createPair(tokenA, tokenB);
        }
        (uint256 reserveA, uint256 reserveB) = SurfswapLibrary.getReserves(
            factory,
            tokenA,
            tokenB
        );
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint256 amountBOptimal = SurfswapLibrary.quote(
                amountADesired,
                reserveA,
                reserveB
            );
            if (amountBOptimal <= amountBDesired) {
                require(
                    amountBOptimal >= amountBMin,
                    "SurfswapRouter: INSUFFICIENT_B_AMOUNT"
                );
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint256 amountAOptimal = SurfswapLibrary.quote(
                    amountBDesired,
                    reserveB,
                    reserveA
                );
                assert(amountAOptimal <= amountADesired);
```

```
                require(
                    amountAOptimal >= amountAMin,
                    "SurfswapRouter: INSUFFICIENT_A_AMOUNT"
                );
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Listing 3.2: `SurfswapRouter::_addLiquidity()`

It comes to our attention that the `SurfswapRouter` has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two amounts: `amountXDesired` and `amountXMin`. The first amount `amountXDesired` determines the desired amount for adding liquidity to the pool and the second amount `amountXMin` determines the minimum amount of used assets. There are two implicit conditions, i.e., `amountADesired >= amountAMin` and `amountBDesired >= amountBMin`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on `SurfswapRouter` may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of `amountADesired >= amountAMin` and `amountBDesired >= amountBMin` explicitly in the `addLiquidity()` function.

Status This issue has been confirmed.

3.2 Fork Resistant Domain Separator In SurfswapERC20 Implementation

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SurfswapERC20`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

While examining the logics of the `SurfswapERC20` contract, we observe it implements the `EIP2612` specification by providing the `permit()` support. In the `SurfswapERC20` contract, it comes to our attention that the state variable `DOMAIN_SEPARATOR` is only assigned in the `constructor()` function (lines 29).

```
    constructor() public {
        uint256 chainId;
        assembly {
            chainId := chainid
        }
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256(
                    "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
                ),
                keccak256(bytes(name)),
                keccak256(bytes("1")),
                chainId,
                address(this)
            )
        );
    }
```

Listing 3.3: `SurfswapERC20::constructor()`

The `DOMAIN_SEPARATOR` is used in the `permit()` function, which allows users to modify the allowance mapping using a signed message, instead of through `msg.sender`. When analyzing this `permit()` function, we notice the current implementation can be improved by recalculating the value of `DOMAIN_SEPARATOR` in `permit()` function. Suppose there is a hard-fork, since `DOMAIN_SEPARATOR` is immutable, a valid signature for one chain could be replayed on the other.

```
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        require(deadline >= block.timestamp, "Surfswap: EXPIRED");
        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                DOMAIN_SEPARATOR,
                keccak256(
                    abi.encode(
                        PERMIT_TYPEHASH,
                        owner,
                        spender,
                        value,
                        nonces[owner]++,
                        deadline
                    )
                )
            )
```

```
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(
            recoveredAddress != address(0) && recoveredAddress == owner,
            "Surfswap: INVALID_SIGNATURE"
        );
        _approve(owner, spender, value);
    }
```

Listing 3.4: `SurfswapERC20::permit()`

Recommendation Recalculate the value of `DOMAIN_SEPARATOR` inside the `permit()` function.

Status This issue has been confirmed.

3.3 Potential Inconsistent Fee Calculation Between SurfswapLibrary And SurfswapPair

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SurfswapLibrary`, `SurfswapPair`
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [3]

Description

The `SurfswapPair` contract adds `swapFee` and `devFee` to customize the fee distribution in the contract. The `feeToSetter` could set the fees via the `setDevFee()` and the `setSwapFee()` routines in the `SurfswapFactory` contract. The fees will be charged in the `SurfswapPair` contract. To elaborate, we show below the related code snippet of the `SurfswapPair::swap()` routine.


```
    function swap(
        uint256 amount0Out,
        uint256 amount1Out,
        address to,
        bytes calldata data
    ) external lock {
        require(
            amount0Out > 0  amount1Out > 0,
            "Surfswap: INSUFFICIENT_OUTPUT_AMOUNT"
        );
        (uint112 _reserve0, uint112 _reserve1, ) = getReserves(); // gas savings
        require(
            amount0Out < _reserve0 && amount1Out < _reserve1,
            "Surfswap: INSUFFICIENT_LIQUIDITY"
        );
```

```

        uint256 balance0;
        uint256 balance1;
        {
            // scope for _token{0,1}, avoids stack too deep errors
            address _token0 = token0;
            address _token1 = token1;
            require(to != _token0 && to != _token1, "Surfswap: INVALID_TO");
            if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer tokens
            if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer tokens
            if (data.length > 0)
                ISurfswapCallee(to).SurfswapCall(
                    msg.sender,
                    amount0Out,
                    amount1Out,
                    data
                );
            balance0 = IERC20(_token0).balanceOf(address(this));
            balance1 = IERC20(_token1).balanceOf(address(this));
        }
        uint256 amount0In = balance0 > _reserve0 - amount0Out
            ? balance0 - (_reserve0 - amount0Out)
            : 0;
        uint256 amount1In = balance1 > _reserve1 - amount1Out
            ? balance1 - (_reserve1 - amount1Out)
            : 0;
        require(
            amount0In > 0  amount1In > 0,
            "Surfswap: INSUFFICIENT_INPUT_AMOUNT"
        );
        {
            uint _swapFee = swapFee;
            // scope for reserve{0,1}Adjusted, avoids stack too deep errors
            uint balance0Adjusted = balance0.mul(10000).sub(
                amount0In.mul(_swapFee)
            );
            uint balance1Adjusted = balance1.mul(10000).sub(
                amount1In.mul(_swapFee)
            );
            require(
                balance0Adjusted.mul(balance1Adjusted) >=
                    uint256(_reserve0).mul(_reserve1).mul(10000**2),
                "Surfswap: K"
            );
        }

        _update(balance0, balance1, _reserve0, _reserve1);
        emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
    }
```

Listing 3.5: `SurfswapPair::swap()`

In the above `swap()` routine implementation, the reserved swap fee is `_swapFee`‰ (line 273) of the `amountIn`. However, this is inconsistent with the fee rate (50‰ – lines 85 and 87) in the `SurfswapLibrary` contract where swap operations are performed by the router. The code snippet of the `getAmountOut()` routine in the `SurfswapLibrary` contract is shown as below.

```
75     function getAmountOut(
76         uint256 amountIn,
77         uint256 reserveIn,
78         uint256 reserveOut
79     ) internal pure returns (uint256 amountOut) {
80         require(amountIn > 0, "SurfswapLibrary: INSUFFICIENT_INPUT_AMOUNT");
81         require(
82             reserveIn > 0 && reserveOut > 0,
83             "SurfswapLibrary: INSUFFICIENT_LIQUIDITY"
84         );
85         uint256 amountInWithFee = amountIn.mul(995);
86         uint256 numerator = amountInWithFee.mul(reserveOut);
87         uint256 denominator = reserveIn.mul(1000).add(amountInWithFee);
88         amountOut = numerator / denominator;
89     }
```

Listing 3.6: `SurfswapLibrary::getAmountOut()`

The inconsistent fee calculation between the `SurfswapPair` and the `SurfswapLibrary` will not block the token swap if the `SurfswapPair` fee is smaller than the `SurfswapLibrary` fee, but it will make the amount of the target token smaller than expected.

Recommendation Be consistent on the fee calculation between `SurfswapPair` and `SurfswapLibrary` .

Status The issue has been confirmed. The team clarifies they will re-deploy the router with new fees and update old pair contracts with new fees.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Surfswap` protocol, there is a special administrative account, i.e., `feeToSetter`. This `feeToSetter` account plays a critical role in governing and regulating the system-wide operations (e.g., setting feeTo, setting _devFee and _swapFee, etc.). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the setDevFee() function in the SurfswapFactory contract, which allows the feeToSetter to change the devFee.

```
    function setDevFee(address _pair, uint8 _devFee) external {
        require(msg.sender == feeToSetter, "Surfswap: FORBIDDEN");
        require(_devFee > 0, "Surfswap: FORBIDDEN_FEE");
        SurfswapPair(_pair).setDevFee(_devFee);
    }
```

Listing 3.7: SurfswapFactory::setDevFee()

It is worrisome if the privileged feeToSetter account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team clarifies the privileged account will be passed to multi-sig account for all contracts and timelock for token contract.

4 Conclusion

In this audit, we have analyzed the `Surfswap` protocol design and implementation. `Surfswap` is a decentralized exchange with an automated market maker for the support of liquidity provision and peer-to-peer transactions on `Kava`. It is designed to provide an one-stop shop for the crypto community and the implementation is based on the popular `Uniswap-v2` protocol with customization on certain swaps as well as the associated fee. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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.

References

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

[2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

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

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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

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