SMART CONTRACT AUDIT REPORT

for

88MPH-V3

Prepared By: Shuxiao Wang

PeckShield
May 18, 2021

Document Properties

Client	88mph
Title	Smart Contract Audit Report
Target	88mph-v3
Version	1.0
Author	Xuxian Jiang
Auditors	Jian Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 18, 2021	Xuxian Jiang	Final Release
1.0-rc	May 18, 2021	Xuxian Jiang	Release Candidate
0.2	May 11, 2021	Xuxian Jiang	Add More Findings #1
0.1	May 4, 2021	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 **88mph-v3 Protocol** design document and related smart contract source code, 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 is well-designed. This document outlines our audit results.

1.1 About 88mph-v3 Protocol

`88mph` is an `Ethereum` protocol that allows users to lend their crypto assets at a fixed-interest rate. By doing so, users earn rewards in the protocol token `MPH` and the protocol's revenues. It is designed to be a deposit account with which users earn a fixed income and the bank rewards users with loyalty tokens (`MPH`) giving shareholders rights like cash dividends and governance power. More information about the audited `88mph-v3` protocol can be found at: `https://docs.88mph.app/`.

The basic information of the 88mph-v3 protocol is as follows:

Table 1.1: Basic Information of 88mph-v3

Item	Description
Name	88mph
Website	http://88mph.app/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 18, 2021

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

- https://github.com/88mphapp/88mph-contracts.git (93bbd88)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/88mphapp/88mph-contracts.git (24b9fcb)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [9]:

- 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) [8], 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 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), 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 Logic	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 88mph-v3 protocol design and 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	1	■
Medium	1	■
Low	2	■ ■
Informational	1	■
Total	5	

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 1 high-severity vulnerability, 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key 88mph-v3 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Proper Dividend Accounting In ERC1155DividentToken	Business Logic	Fixed
PVE-002	Low	Suggested setGovRewardMultiple() Counterpart	Code Practices	Fixed
PVE-003	Informational	Suggested Transfer Events For Wrapped-ERC1155Token Mint/Burn	Code Practices	Fixed
PVE-004	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed
PVE-005	Medium	Possible Costly xMPH From Improper Liquidity Initialization	Time and State	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Proper Dividend Accounting In ERC1155DividentToken

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: ERC1155DividentToken
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To facilitate the fixed-rate yield generation, the 88mph protocol makes use of the ERC1155DividentToken contract to allow for efficient distribution of dividends to all holders of an token ID. The specific token contract also supports multiple dividend tokens. In the following, we examine this token contract implementation.

To elaborate, we show below the _beforeTokenTransfer() routine in the ERC1155DividentToken token contract. This routine is invoked for every token transfer to properly keep track of due dividends for holders. Specifically, it discerns three different scenarios, i.e., Mint, Burn, and Transfer. It comes to our attention that the Burn-related handling logic is flawed.

```
    function _beforeTokenTransfer(
        address operator,
        address from,
        address to,
        uint256[] memory ids,
        uint256[] memory amounts,
        bytes memory data
    ) internal virtual override(ERC1155Base) {
        super._beforeTokenTransfer(operator, from, to, ids, amounts, data);

        if (from == address(0)) {
            // Mint
            for (uint256 i = 0; i < ids.length; i++) {
                uint256 tokenID = ids[i];
                uint256 amount = amounts[i];
```

```
                for (uint256 j = 1; j <= dividendTokenDataListLength; j++) {
                    DividendTokenData storage dividendTokenData =
                        dividendTokenDataList[j];
                    dividendTokenData.magnifiedDividendCorrections[tokenID][
                        to
                    ] -= (dividendTokenData.magnifiedDividendPerShare[tokenID] *
                        amount)
                        .toInt256();
                }
            }
        } else if (to == address(0)) {
            // Burn
            for (uint256 i = 0; i < ids.length; i++) {
                uint256 tokenID = ids[i];
                uint256 amount = amounts[i];

                for (uint256 j = 1; j <= dividendTokenDataListLength; j++) {
                    DividendTokenData storage dividendTokenData =
                        dividendTokenDataList[j];
                    dividendTokenData.magnifiedDividendCorrections[tokenID][
                        to
                    ] += (dividendTokenData.magnifiedDividendPerShare[tokenID] *
                        amount)
                        .toInt256();
                }
            }
        } else {
            // Transfer
            for (uint256 i = 0; i < ids.length; i++) {
                uint256 tokenID = ids[i];
                uint256 amount = amounts[i];

                for (uint256 j = 1; j <= dividendTokenDataListLength; j++) {
                    DividendTokenData storage dividendTokenData =
                        dividendTokenDataList[j];
                    int256 _magCorrection =
                        (dividendTokenData.magnifiedDividendPerShare[tokenID] *
                            amount)
                            .toInt256();
                    // Retain the rewards
                    dividendTokenData.magnifiedDividendCorrections[tokenID][
                        from
                    ] += _magCorrection;
                    dividendTokenData.magnifiedDividendCorrections[tokenID][
                        to
                    ] -= _magCorrection;
                }
            }
        }
    }
```

Listing 3.1: ERC1155DividentToken::_beforeTokenTransfer()

Specifically, in the Burn case, the related state of magnifiedDividendCorrections[tokenID] for the from should be updated, instead of the current to! A non-updated magnifiedDividendCorrections[tokenID] for the from account may result in potential loss for the sender.

Recommendation Revise the above _beforeTokenTransfer() logic to properly update dividend correction for associated parties in all cases.

Status This issue has been fixed in this commit: a672cb6.

3.2 Suggested setGovRewardMultiple() Counterpart

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MPHIssuanceModel02
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

As part of the built-in incentive mechanism, the 88mph protocol has an issuance model that supports customized multipliers that are applied when minting the governance token MPH for a pool's depositor reward. There are also two specific multipliers used for calculating dev and gov rewards respectively. These two specific multipliers are saved in two different state variables, i.e., devRewardMultiplier and govRewardMultiplier.

Our analysis shows that the MPHIssuanceModel02 contract has the proper setter support for devRewardMultiplier. However, there is no related setter for govRewardMultiplier. Therefore, it is suggested to add the setGovRewardMultiplier() counterpart.

```
    function setDevRewardMultiplier(uint256 newMultiplier) external onlyOwner {
        require(
            newMultiplier <= PRECISION,
            "MPHIssuanceModel: invalid multiplier"
        );
        devRewardMultiplier = newMultiplier;
        emit ESetParamUint(
            msg.sender,
            "devRewardMultiplier",
            address(0),
            newMultiplier
        );
    }
```

Listing 3.2: MPHIssuanceModel02::setDevRewardMultiplier()

Recommendation Add the `setGovRewardMultiplier()` counterpart to be consistent with another existing one, i.e., `setDevRewardMultiplier()`.

Status This issue has been fixed in this commit: `d96ac70`.

3.3 Suggested Transfer Events For WrappedERC1155Token Mint/Burn

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `WrappedERC1155Token`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `WrappedERC1155Token` contract as an example. This contract is designed to wrap their `88mph NFTs` in an `ERC-20` token. While examining the events that reflect the token dynamics, we notice the `Transfer` event is not emitted when the token is being minted or burned.

To elaborate, we show below its `_beforeTokenTransfer()` routine, which is invoked for every token transfer. Note that it indeed emits the ERC20-compliant `Transfer` event for actual transfers, but misses the same event for `Mint` and `Burn` actions. According to the ERC20 specification, there is also a need to emit `Transfer` for `Mint` and `Burn` operations.

```
    function _beforeTokenTransfer(
        address operator,
        address from,
        address to,
        uint256[] memory ids,
        uint256[] memory amounts,
        bytes memory data
    ) internal virtual override {
        super._beforeTokenTransfer(operator, from, to, ids, amounts, data);

        if (from == address(0)) {
            // Mint
            if (!deployWrapperOnMint) {
                return;
```

```
            }
            for (uint256 i = 0; i < ids.length; i++) {
                _deployWrapper(ids[i]);
            }
        } else if (to != address(0)) {
            // Transfer
            for (uint256 i = 0; i < ids.length; i++) {
                address wrapperAddress = tokenIDToWrapper[ids[i]];
                if (wrapperAddress != address(0)) {
                    ERC20Wrapper wrapper = ERC20Wrapper(wrapperAddress);
                    wrapper.emitTransferEvent(from, to, amounts[i]);
                }
            }
        }
    }
```

Listing 3.3: WrappedERC1155Token::_beforeTokenTransfer()

Recommendation Properly emit the `Transfer` event in all cases. This is very helpful for external analytics and reporting tools.

Status This issue has been fixed in this commit: `58250f5`.

3.4 Accommodation of approve() Idiosyncrasies

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
```

```
    * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses'
        //  allowance to zero by calling 'approve(_spender, 0)' if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.4: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `AaveMarket::deposit()` routine as an example. This routine is designed to deposit supported assets into an integrated money market. To accommodate the specific idiosyncrasy, for each `safeIncreaseAllowance()` (line 67), there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
58    function deposit(uint256 amount) external override onlyOwner {
59        require(amount > 0, "AaveMarket: amount is 0");
60
61        ILendingPool lendingPool = ILendingPool(provider.getLendingPool());
62
63        // Transfer 'amount' stablecoin from 'msg.sender'
64        stablecoin.safeTransferFrom(msg.sender, address(this), amount);
65
66        // Approve 'amount' stablecoin to lendingPool
67        stablecoin.safeIncreaseAllowance(address(lendingPool), amount);
68
69        // Deposit 'amount' stablecoin to lendingPool
70        lendingPool.deposit(
71            address(stablecoin),
72            amount,
73            address(this),
74            REFERRALCODE
75        );
76    }
```

Listing 3.5: AaveMarket::deposit()

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been fixed in this commit: `0ac3924`.

3.5 Possible Costly xMPH From Improper Liquidity Initialization

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `xMPH`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

The `88mph-v3` protocol allows users to stake supported `MPH` and get in return `xMPH` tokens to represent the pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool token extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()/withdraw()` routines. The `deposit()` routine is used for participating users to deposit the supported asset (e.g., `MPH`) and get respective `xMPH` pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function deposit(uint256 _mphAmount)
        external
        returns (uint256 shareAmount)
    {
        require(_mphAmount > 0, "xMPH: 0 amount");
        shareAmount = _mphAmount.decdiv(getPricePerFullShare());
        _mint(msg.sender, shareAmount);
        mph.transferFrom(msg.sender, address(this), _mphAmount);
    }

    /**
        @notice Withdraw MPH using xMPH
        @dev The amount can't be 0
        @param _shareAmount The amount of xMPH to burn
```

```
86          @return mphAmount The amount of MPH withdrawn
87       */
88      function withdraw(uint256 _shareAmount)
89          external
90          returns (uint256 mphAmount)
91      {
92          require(_shareAmount > 0, "xMPH: 0 amount");
93          mphAmount = _shareAmount.decmul(getPricePerFullShare());
94          _burn(msg.sender, _shareAmount);
95          mph.transfer(msg.sender, mphAmount);
96      }
97
98      /**
99          @notice Compute the amount of MPH that can be withdrawn by burning
100                 1 xMPH. Increases linearly during a reward distribution period.
101         @dev Initialized to be PRECISION (representing 1 MPH = 1 xMPH)
102         @return The amount of MPH that can be withdrawn by burning
103                 1 xMPH
104      */
105     function getPricePerFullShare() public view returns (uint256) {
106         uint256 totalShares = totalSupply();
107         uint256 mphBalance = mph.balanceOf(address(this));
108         if (totalShares == 0 || mphBalance == 0) {
109             return PRECISION;
110         }
111         uint256 _lastRewardAmount = lastRewardAmount;
112         uint256 _currentUnlockEndTimestamp = currentUnlockEndTimestamp;
113         if (
114             _lastRewardAmount == 0 ||
115             block.timestamp >= _currentUnlockEndTimestamp
116         ) {
117             // no rewards or rewards fully unlocked
118             // entire balance is withdrawable
119             return mphBalance.decdiv(totalShares);
120         } else {
121             // rewards not fully unlocked
122             // deduct locked rewards from balance
123             uint256 _lastRewardTimestamp = lastRewardTimestamp;
124             uint256 lockedRewardAmount =
125                 (_lastRewardAmount *
126                     (_currentUnlockEndTimestamp - block.timestamp)) /
127                     (_currentUnlockEndTimestamp - _lastRewardTimestamp);
128             return (mphBalance - lockedRewardAmount).decdiv(totalShares);
129         }
130     }
```

Listing 3.6: xMPH::deposit()/withdraw()

Specifically, when the pool is being initialized (line 108), the share value directly takes the value of `_mphAmount.decdiv(PRECISION)` (line 77), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shareAmount = 1 WEI`. With that, the actor can further deposit a huge amount of `MPH` with the goal of making the `xMPH` pool token extremely expensive.

An extremely expensive `xMPH` pool token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `getPricePerFullShare()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status This issue has been confirmed. The team will exercise extra caution in properly initializing the pool.

4 Conclusion

In this audit, we have analyzed the 88mph-v3 design and implementation. The system presents a unique, robust offering as a decentralized non-custodial lending platform allowing users to lend crypto assets at a fixed-interest rate. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Furthermore, 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

[5] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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