SMART CONTRACT AUDIT REPORT

for

88MPH

Prepared By: Shuxiao Wang

Hangzhou, China
January 11, 2020

Document Properties

Client	88mph
Title	Smart Contract Audit Report
Target	88mph
Version	1.0
Author	Xudong Shao
Auditors	Xudong Shao, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 11, 2020	Xudong Shao	Final Release

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 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 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 88mph Protocol

The 88mph protocol is a fixed-rate yield-generation protocol by pooling the deposits together. It puts the deposited DAI into a single pool, from which users can withdraw a deposit once its deposit period is over. The 88mph protocol also offers floating-rate bonds which allows someone to immediately fill up the debt of one or more deposits using their own money, and in exchange they would receive the yield generated by those deposits. This significantly reduces the risk of depositing into 88mph and provides a brand new financial product that allows users to long the interest rates of lending protocols.

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

Table 1.1: Basic Information of 88mph

Item	Description
Name	88mph
Website	http://88mph.app/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 11, 2020

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 (dd87a8a)

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 (dbcdd5d)

1.2 About PeckShield

PeckShield Inc. [7] 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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- 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.

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

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) [5], 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 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	0	
Medium	0	
Low	1	■
Informational	4	■ ■ ■ ■
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 low-severity vulnerability and 4 informational recommendations.

Table 2.1: Key 88mph Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Front-Running Resulting Losing Ownership	Business Logic	Confirmed
PVE-002	Informational	Redundant Check in Market::constructor()	Coding Practices	Fixed
PVE-003	Informational	Gas Optimization in DInterest::_deposit()	Coding Practices	Fixed
PVE-004	Informational	Unsafe Ownership Transition	Business Logic	Confirmed
PVE-005	Informational	Unused Events in MPHIssuanceModel01	Coding Practices	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 Possible Front-Running Resulting Losing Ownership

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MPHToken`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The 88mph protocol is a fixed-rate yield-generation protocol which pools the deposits together. It puts the deposited DAI into a single pool, from which users can withdraw a deposit once its deposit period is over. Users will receive MPH tokens after deposits from `MPHToken` contract.

After the `MPHToken` contract is deployed, the `init()` function will be called to initialize the contract and announce ownership.

```
    function init() public {
        require(!initialized, "MPHToken: initialized");
        initialized = true;

        _transferOwnership(msg.sender);
    }
```

Listing 3.1: MPHToken.sol

However, the `init` function is defined as `public` and anyone can call this function to take the ownership of `MPHToken`. As a result, right after the `MPHToken` contract is deployed, an attacker can use high gas fee to `init()` the contract first. This would cause front-running and no one is able to take back the ownership anymore.

Recommendation Use `onlyOwner` for `init()` function.

Status This issue has been confirmed by the team. However, the `MPHToken` contract has been deployed and initialized successfully. So the dev team decides to leave it as is.

3.2 Redundant Check in Market::constructor()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: AaveMarket, CompoundERC20Market, HarvestMarket, YVaultMarket
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

As we introduced in Section 3.1, 88mph pools the deposits together and deposits them to different markets to earn interests. Take `AaveMarket` for example, the deployer has to provide `_provider`, `_aToken` and `_stablecoin` to the constructor which will be used in `deposit` and `withdraw`. The constructor has to make sure these addresses are not `address(0)` and they are contracts.

```
    constructor(
        address _provider,
        address _aToken,
        address _stablecoin
    ) public {
        // Verify input addresses
        require(
            _provider != address(0) &&
                _aToken != address(0) &&
                _stablecoin != address(0),
            "AaveMarket: An input address is 0"
        );
        require(
            _provider.isContract() &&
                _aToken.isContract() &&
                _stablecoin.isContract(),
            "AaveMarket: An input address is not a contract"
        );

        provider = ILendingPoolAddressesProvider(_provider);
        stablecoin = ERC20(_stablecoin);
        aToken = ERC20(_aToken);
    }
```

Listing 3.2: AaveMarket.sol

However, `address(0)` won't pass the check of `isContract()`. So the first check on `address(0)` is redundant. Therefore, the first `require` could be safely removed.

The same problem exists in `CompoundERC20Market`, `HarvestMarket` and `YVaultMarket`.

Recommendation Remove the first check that the input address shouldn't be `address(0)`.

```
    constructor(
        address _provider,
        address _aToken,
        address _stablecoin
    ) public {
        require(
            _provider.isContract() &&
                _aToken.isContract() &&
                _stablecoin.isContract(),
            "AaveMarket: An input address is not a contract"
        );

        provider = ILendingPoolAddressesProvider(_provider);
        stablecoin = ERC20(_stablecoin);
        aToken = ERC20(_aToken);
    }
```

Listing 3.3: AaveMarket.sol

Status This issue has been fixed in the commit: 29c81174bd1652633821e517503bce1435bf08f9.

3.3 Gas Optimization in DInterest::_deposit()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DInterest`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

Users can deposit DAI into `DInterest` contract for fixed-rate APY. And `MPHToken` contract will mint MPH tokens to users according to the amounts they deposit. The owner of `DInterest` contract sets the `MinDepositAmount` and `MaxDepositAmount`. When users deposits by calling the `deposit()` function, the `amount` should be bigger than `MinDepositAmount` and less than `MaxDepositAmount`.

```
    function deposit(uint256 amount, uint256 maturationTimestamp)
        external
        nonReentrant
    {
        _deposit(amount, maturationTimestamp);
    }
```

Listing 3.4: DInterest.sol

```
    function _deposit(uint256 amount, uint256 maturationTimestamp) internal {
        // Cannot deposit 0
        require(amount > 0, "DInterest: Deposit amount is 0");

        // Ensure deposit amount is not more than maximum
        require(
            amount >= MinDepositAmount && amount <= MaxDepositAmount,
            "DInterest: Deposit amount out of range"
        );

        // Ensure deposit period is at least MinDepositPeriod
        uint256 depositPeriod = maturationTimestamp.sub(now);
        require(
            depositPeriod >= MinDepositPeriod &&
                depositPeriod <= MaxDepositPeriod,
            "DInterest: Deposit period out of range"
        );

        // Update totalDeposit
        totalDeposit = totalDeposit.add(amount);

        // Update funding related data
        uint256 id = deposits.length.add(1);
        unfundedUserDepositAmount = unfundedUserDepositAmount.add(amount);

        // Calculate interest
        uint256 interestAmount = calculateInterestAmount(amount, depositPeriod);
        require(interestAmount > 0, "DInterest: interestAmount == 0");

        // Update totalInterestOwed
        totalInterestOwed = totalInterestOwed.add(interestAmount);

        // Mint MPH for msg.sender
        uint256 mintMPHAmount = mphMinter.mintDepositorReward(
            msg.sender,
            amount,
            depositPeriod,
            interestAmount
        );

        // Record deposit data for `msg.sender`
        deposits.push(
            Deposit({
                amount: amount,
                maturationTimestamp: maturationTimestamp,
                interestOwed: interestAmount,
                initialMoneyMarketIncomeIndex: moneyMarket.incomeIndex(),
                active: true,
                finalSurplusIsNegative: false,
                finalSurplusAmount: 0,
                mintMPHAmount: mintMPHAmount,
                depositTimestamp: now
```

```
            })
        );

        // Transfer `amount` stablecoin to DInterest
        stablecoin.safeTransferFrom(msg.sender, address(this), amount);

        // Lend `amount` stablecoin to money market
        stablecoin.safeIncreaseAllowance(address(moneyMarket), amount);
        moneyMarket.deposit(amount);

        // Mint depositNFT
        depositNFT.mint(msg.sender, id);

        // Emit event
        emit EDeposit(
            msg.sender,
            id,
            amount,
            maturationTimestamp,
            interestAmount,
            mintMPHAmount
        );
    }
```

Listing 3.5: DInterest.sol

However, every time the users deposits, the `_deposit()` routine also checks that the `amount` should be bigger than 0. This check can be safely removed if `setMinDepositAmount()` checks that the `newValue` is bigger than 0.

```
    function setMinDepositAmount(uint256 newValue) external onlyOwner {
        require(newValue <= MaxDepositAmount, "DInterest: invalid value");
        MinDepositAmount = newValue;
        emit ESetParamUint(msg.sender, "MinDepositAmount", newValue);
    }
```

Listing 3.6: DInterest.sol

Recommendation Check the `newValue` is bigger than 0.

```
    function setMinDepositAmount(uint256 newValue) external onlyOwner {
        require(newValue <= MaxDepositAmount, "DInterest: invalid value");
        require(newValue > 0, "DInterest: invalid value");
        MinDepositAmount = newValue;
        emit ESetParamUint(msg.sender, "MinDepositAmount", newValue);
    }
```

Listing 3.7: DInterest.sol

Status This issue has been fixed in the commit: e61e51ea47a4b759aa48688eab8a6773fce3d89e.

3.4 Unsafe Ownership Transition

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Ownable`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In 88mph, the `Ownable` contract is widely used for ownership management in many contracts such as `DInterest`, `NFT`, etc. When the contract owner needs to transfer the ownership to another address, she could invoke the `transferOwnership()` function with a `newOwner` address.

```
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
```

Listing 3.8: Ownable.sol

However, if the `newOwner` is not the exact address of the new owner (e.g., due to a typo), nobody could own that contract anymore.

Recommendation Implement a two-step ownership transfer mechanism that allows the new owner to claim the ownership by signing a transaction.

```
function transferOwnership(
    address newOwner
)
    external
    onlyOwner
{
    require(newOwner != address(0), "Owned: Address must not be null");
    require(candidateOwner != newOwner, "Owned: Same candidate owner");
    candidateOwner = newOwner;
}

function claimOwner()
    external
{
    require(candidateOwner == msg.sender, "Owned: Claim ownership failed");
    owner = candidateOwner;
    emit OwnerChanged(candidateOwner);
}
```

Listing 3.9: Ownable.sol

Status This issue has been confirmed by the team. However, the contracts have been deployed and initialized successfully. So the dev team decides to leave it as is.

3.5 Unused Events in MPHIssuanceModel01

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MPHIssuanceModel01`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

The `MPHIssuanceModel01` contract is used to compute the MPH amount to reward to depositors and funders. In this contract, there is an unused event, `ESetParamAddress`, which could be safely removed.

```
    event ESetParamAddress(
        address indexed sender,
        string indexed paramName,
        address newValue
    );
```

Listing 3.10: MPHIssuanceModel01.sol

Recommendation Remove the unused event.

Status This issue has been confirmed by the team. However, the `MPHIssuanceModel01` contract has been deployed and initialized successfully. So the dev team decides to leave it as is.

4 Conclusion

In this audit, we have analyzed the 88mph design and implementation. The system is a fixed-rate yield-generation protocol on Ethereum that allows users to deposit assets, earn fixed-rate interests, and farm for MPH tokens. During the audit, we notice that the current code base is well structured and neatly organized, and 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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