SMART CONTRACT AUDIT REPORT

for

Vabble

Prepared By: Xiaomi Huang

PeckShield

October 15, 2023

Document Properties

Client	Vabble
Title	Smart Contract Audit Report
Target	Vabble
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Jianzhuo Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 15, 2023	Xuxian Jiang	Final Release
1.0-rc1	October 3, 2023	Xuxian Jiang	Release Candidate #1

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 `Vabble` 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 audited protocol 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 Vabble

`Vabble` provides `WEB3` projects and filmmakers with an easy way to finance and distribute `WEB3` content such as `Films`, `Series`, and `Animations`. Navigating multiple infrastructures, high upfront costs, and extensive networking make funding `WEB3` films and series difficult. `Vabble` has built a unified `WEB3` content financing platform by streamlining the funding process, eliminating upfront costs for creators, studios, and filmmakers, and letting them concentrate on what they do best. Moreover, it developed a `WEB3` content distribution infrastructure that brings value to investors and offers a fresh content experience for communities and consumers. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Vabble

Item	Description
Name	Vabble
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 15, 2023

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

- https://github.com/Vabble/dao-sc.git (2392bc8)

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

- https://github.com/Vabble/dao-sc.git (9a8f446)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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) [11], 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) 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.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Vabble` 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 logic, 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	4	■■■■
Low	3	■■■
Undetermined	1	■
Total	8	

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 4 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Vabble Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Reentrancy Risk Avoidance in FactoryFilmNFT/FactorySubNFT	Time And State	Resolved
PVE-002	Medium	Possible Sandwich/MEV For Reduced Returns	Time And State	Resolved
PVE-003	Low	No ETH Support in FactoryFilmNFT::mintToBatch()	Business Logic	Resolved
PVE-004	Medium	Collusion-Based Revenue Collection With Just-in-Time Film NFTs	Business Logic	Resolved
PVE-005	Low	Enforcement of One-Time Initialization in FactoryFilmNFT	Coding Practices	Resolved
PVE-006	Medium	Improper Update on Film Fund Raise in FactoryFilmNFT	Business Logic	Resolved
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-008	Undetermined	Possibly Out-of-Sync Reward Boost in StakingPool	Business Logic	Resolved

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 Reentrancy Risk Avoidance in FactoryFilmNFT & FactorySubNFT

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [15] exploit, and the `Uniswap/Lendf.Me` hack [14].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `FactoryFilmNFT` as an example, the `mint()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 209) start before effecting the update on internal states, hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
180     function mint(
181         uint256 _filmId,
182         address _to,
183         address _payToken
```

```
    ) public payable {
        if(_payToken != IOwnablee(OWNABLE).PAYOUT_TOKEN() && _payToken != address(0)) {
            require(IOwnablee(OWNABLE).isDepositAsset(_payToken), "mint: not allowed asset");
        }
        require(mintInfo[_filmId].maxMintAmount > 0, "mint: no mint info");
        require(mintInfo[_filmId].maxMintAmount > getTotalSupply(_filmId), "mint: exceed mint amount");

        __handleMintPay(_filmId, _payToken);

        VabbleNFT t = filmNFTContract[_filmId];
        uint256 tokenId = t.mintTo(_to);
        filmNFTTokenList[_filmId].push(tokenId);

        emit FilmERC721Minted(address(t), tokenId, _to, block.timestamp);
    }

    function __handleMintPay(
        uint256 _filmId,
        address _payToken
    ) private {
        uint256 expectAmount = getExpectedTokenAmount(_payToken, mintInfo[_filmId].price);
        // Return remain ETH to user back if case of ETH and Transfer Asset from buyer to this contract
        if(_payToken == address(0)) {
            require(msg.value >= expectAmount, "handlePay: Insufficient paid");
            if (msg.value > expectAmount) {
                Helper.safeTransferETH(msg.sender, msg.value - expectAmount);
            }
        } else {
            Helper.safeTransferFrom(_payToken, msg.sender, address(this), expectAmount);
        }
        ...
    }
```

Listing 3.1: `FactoryFilmNFT::mint()`

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`. Note the same issue is also applicable another contract `FactorySubNFT`.

Status This issue has been fixed in the following commit: `9a8f446`.

3.2 Possible Sandwich/MEV For Reduced Returns

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [4]

Description

To facilitate the user engagement, the `Vabble` protocol supports a variety of payment tokens. Because of that, there is a constant need of swapping one asset to another. With that, the protocol has provided a dedicated helper routine to facilitate the asset conversion. Our analysis shows this swap routine exposes a potential MEV risk.

```
    function swapAsset(bytes calldata _swapArgs) external transferHandler(_swapArgs) returns (uint256 amount_) {
        (
            uint256 depositAmount,
            address depositAsset,
            address incomingAsset
        ) = abi.decode(_swapArgs, (uint256, address, address));

        (address router, , address weth, address[] memory path) = __checkPool(depositAsset, incomingAsset);
        require(router != address(0), "swapAsset: No Pool");

        // Get payoutAmount from depositAsset on Uniswap
        uint256 expectAmount = IUniswapV2Router(router).getAmountsOut(depositAmount, path)[1];

        if(path[0] == weth) {
            amount_ = __swapETHToToken(depositAmount, expectAmount, router, path)[1];
        } else {
            amount_ = __swapTokenToToken(depositAmount, expectAmount, router, path)[1];
        }
    }
```

Listing 3.2: UniHelper::swapAsset()

To elaborate, we show above the related helper routine. We notice the conversion is routed to `UniswapV2/SushiSwap` in order to swap one asset to another. While the swap operation does specify the expected amount for slippage control, the expected amount calculation does not present to be manipulation-resistant and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of conversion.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of farming users. Note this issue affects a number of routines that rely on the above token swap routine.

Status This issue has been fixed in the following commit: `9a8f446`.

3.3 No ETH Support in FactoryFilmNFT::mintToBatch()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FactoryFilmNFT`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Vabble` protocol has a `FactoryFilmNFT` contract to allow for user funding via file-related `NFT` mints. In the process of reviewing current mint logic, we notice the batch minting does not support the `ETH` payment.

To elaborate, we show below the implementation of the `mintToBatch()` routine. This routine is designed to mint a given batch of `NFTs`. Note actual mint of each individual `NFT` is delegated to the `mint()` routine. However, it comes to our attention that this `mintToBatch()` routine does not have the `payable` modifier, while the `mint()` counterpart does have it. In other words, current implementation of `mintToBatch()` does not support `ETH` as the payment token.

```
    function mintToBatch(
        uint256[] calldata _filmIdList,
        address[] calldata _toList,
        address _payToken
    ) external {
        require(_toList.length > 0, "mintBatch: zero item length");
        require(_toList.length == _filmIdList.length, "mintBatch: bad item length");

        for(uint256 i; i < _toList.length; i++) {
            mint(_filmIdList[i], _toList[i], _payToken);
        }
```

```
    }

    function mint(
        uint256 _filmId,
        address _to,
        address _payToken
    ) public payable {
        if(_payToken != IOwnablee(OWNABLE).PAYOUT_TOKEN() && _payToken != address(0)) {
            require(IOwnablee(OWNABLE).isDepositAsset(_payToken), "mint: not allowed asset");
        }
        require(mintInfo[_filmId].maxMintAmount > 0, "mint: no mint info");
        require(mintInfo[_filmId].maxMintAmount > getTotalSupply(_filmId), "mint: exceed mint amount");

        __handleMintPay(_filmId, _payToken);

        VabbleNFT t = filmNFTContract[_filmId];
        uint256 tokenId = t.mintTo(_to);
        filmNFTTokenList[_filmId].push(tokenId);

        emit FilmERC721Minted(address(t), tokenId, _to, block.timestamp);
    }
```

Listing 3.3: `FactoryFilmNFT::mintToBatch()`

Recommendation Revise the above routine to allow for the use of `ETH` as the payment token.

Status This issue has been fixed in the following commit: `9a8f446`.

3.4 Collusion-Based Revenue Collection With Just-in-Time Film NFTs

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VabbleDAO`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [2]

Description

To facilitate the protocol management, the `Vabble` protocol has a built-in `VabbleDAO` support. To encourage user participation, `VabbleDAO` is designed to transfer certain revenue amount to the user if the user funds this film throughout `NFT` mints. In the process of reviewing the revenue refund logic, we notice the implementation can be improved.

To elaborate, we show below the implementation of the related `__payRevenue()` function. This function counts the total number of owned `NFT` mints in `nftCountOwned` and then computes the revenue as `revenueAmount = nftCountOwned * _payout * revenuePercent / 1e10` (line 369). This logic may be susceptible to a collusion attack where multiple users may share their `NFT` mints so that each may maximize his/her revenue return.

```
356     function __payRevenue(
357         address _user,
358         address _vabToken,
359         uint256 _filmId,
360         uint256 _payout
361     ) private {
362         uint256 nftCountOwned;
363         uint256[] memory nftList = IFactoryFilmNFT(FILM_NFT_FACTORY).getFilmTokenIdList(
                _filmId);
364         for(uint256 i = 0; i < nftList.length; i++) {
365             if(IERC721(FILM_NFT_FACTORY).ownerOf(nftList[i]) == _user) nftCountOwned +=
                    1;
366         }
367
368         ( , , , , uint256 revenuePercent, ,) = IFactoryFilmNFT(FILM_NFT_FACTORY).
                getMintInfo(_filmId);
369         uint256 revenueAmount = nftCountOwned * _payout * revenuePercent / 1e10;
370         if(_payout >= revenueAmount && revenueAmount > 0) {
371             require(StudioPool >= revenueAmount, "revenue: insufficient studio pool");
372             require(IERC20(_vabToken).balanceOf(address(this)) >= revenueAmount, "
                    revenue: insufficient balance");
373
374             Helper.safeTransfer(IOwnablee(OWNABLE).PAYOUT_TOKEN(), _user, revenueAmount)
                    ;
375             StudioPool -= revenueAmount;
376         }
377     }
```

Listing 3.4: `VabbleDAO::__payRevenue()`

Recommendation Revisit the above routine to properly compute the revenue reward in a robust manner.

Status This issue has been fixed in the following commit: `9a8f446`.

3.5 Enforcement of One-Time Initialization in FactoryFilmNFT

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FactoryFilmNFT`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [2]

Description

As mentioned earlier, the `Vabble` protocol has a `FactoryFilmNFT` contract to allow for user funding via file-related `NFT` mints. While reviewing the logic to initialize the film-related mint information, we notice the initialization routine may be repeatedly invoked, which should be revised to be invoked once.

To elaborate, we show below the implementation of the `setMintInfo()` routine. This routine is protected to be invoked only by the film `owner`. However, it comes to our attention that it may be repeatedly invoked. With that, we suggest to take necessary measures to ensure it will be only invoked once for each film.

```
    function setMintInfo(
        uint256 _filmId,
        uint256 _tier,
        uint256 _amount,
        uint256 _price,
        uint256 _feePercent,
        uint256 _revenuePercent
    ) external {
        require(_amount > 0 && _price > 0 && _tier > 0, "setMint: Zero value");
        require(_feePercent <= IProperty(DAO_PROPERTY).maxMintFeePercent(), "setMint: over max mint fee");
        require(_revenuePercent < 1e10, "setMint: over 100%");

        address owner = IVabbleDAO(VABBLE_DAO).getFilmOwner(_filmId);
        require(owner == msg.sender, "setMint: not film owner");

        (uint256 raiseAmount, , uint256 fundType) = IVabbleDAO(VABBLE_DAO).getFilmFund(_filmId);
        if(fundType > 0) { // case of funding film
            require(_amount * _price * (1e10 - _feePercent) / 1e10 > raiseAmount, "setMint: many amount");
        }

        Mint storage mInfo = mintInfo[_filmId];
        mInfo.tier = _tier;                       // 1, 2, 3, , ,
        mInfo.maxMintAmount = _amount;            // 100
        mInfo.price = _price;                     // 5 usdc = 5 * 1e6
        mInfo.feePercent = _feePercent;           // 2% = 2 * 1e8(1% = 1e8, 100% = 1e10)
```

```
        mInfo.revenuePercent = _revenuePercent; // any %(1% = 1e8, 100% = 1e10)
        mInfo.studio = msg.sender;

        emit MintInfoSetted(msg.sender, _filmId, _tier, _amount, _price, _feePercent,
            _revenuePercent, block.timestamp);
    }
```

Listing 3.5: `FactoryFilmNFT::setMintInfo()`

Recommendation Only allow for one-time initialization of the above `setMintInfo()` routine for each film. Also, the same issue is also applicable to another `deployFilmNFTContract()` routine.

Status This issue has been fixed in the following commit: `9a8f446`.

3.6 Improper Update on Film Fund Raise in FactoryFilmNFT

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FactoryFilmNFT`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.3, `Vabble` allows users to mint `Film NFTs` to fund the film production. Naturally, the `FactoryFilmNFT` contract keeps track of the raised funds for each film. While examining internal accounting logic, we notice the current implementation does not properly keep track of the fund raise amount for each film.

To elaborate, we show below the implementation of the related `mint()` routine. It has a rather straightforward logic in collecting the payment and minting the associated film `NFT`. However, it forgot to update the accumulated fund raise with the following statement: `fileFundRaisedByNFT[_fileId] += mintInfo[_filmId].price`.

```
    function mint(
        uint256 _filmId,
        address _to,
        address _payToken
    ) public payable {
        if(_payToken != IOwnablee(OWNABLE).PAYOUT_TOKEN() && _payToken != address(0)) {
            require(IOwnablee(OWNABLE).isDepositAsset(_payToken), "mint: not allowed
                asset");
        }
        require(mintInfo[_filmId].maxMintAmount > 0, "mint: no mint info");
        require(mintInfo[_filmId].maxMintAmount > getTotalSupply(_filmId), "mint: exceed
            mint amount");
```

```
        __handleMintPay(_filmId, _payToken);

        VabbleNFT t = filmNFTContract[_filmId];
        uint256 tokenId = t.mintTo(_to);
        filmNFTTokenList[_filmId].push(tokenId);

        emit FilmERC721Minted(address(t), tokenId, _to, block.timestamp);
    }
```

Listing 3.6: `FactoryFilmNFT::mint()`

Recommendation Revisit the above routine to properly update the accumulated fund raise for each film.

Status This issue has been fixed in the following commit: `9a8f446`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the `Vabble` protocol, there is a privileged `auditor` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration and sensitive operation execution). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Ownablee` contract as an example and show the representative functions potentially affected by the privileges of the `auditor` account.

```
    function setup(
        address _vote,
        address _dao,
        address _stakingPool
    ) external onlyAuditor {
        require(_vote != address(0), "setupVote: bad Vote Contract address");
        VOTE = _vote;
        require(_dao != address(0), "setupVote: bad VabbleDAO contract address");
        VABBLE_DAO = _dao;
        require(_stakingPool != address(0), "setupVote: bad StakingPool contract address");
        STAKING_POOL = _stakingPool;
    }
```

```
    function transferAuditor(address _newAuditor) external onlyAuditor {
        require(_newAuditor != address(0) && _newAuditor != auditor, "Ownablee: Zero newAuditor address");
        auditor = _newAuditor;
    }

    function replaceAuditor(address _newAuditor) external onlyVote {
        require(_newAuditor != address(0) && _newAuditor != auditor, "Ownablee: Zero newAuditor address");
        auditor = _newAuditor;
    }

    function addDepositAsset(address[] calldata _assetList) external onlyAuditor {
        require(_assetList.length > 0, "addDepositAsset: zero list");

        for(uint256 i = 0; i < _assetList.length; i++) {
            if(allowAssetToDeposit[_assetList[i]]) continue;

            depositAssetList.push(_assetList[i]);
            allowAssetToDeposit[_assetList[i]] = true;
        }
    }

    function removeDepositAsset(address[] calldata _assetList) external onlyAuditor {
        require(_assetList.length > 0, "removeDepositAsset: zero list");

        for(uint256 i = 0; i < _assetList.length; i++) {
            if(!allowAssetToDeposit[_assetList[i]]) continue;

            for(uint256 k = 0; k < depositAssetList.length; k++) {
                if(_assetList[i] == depositAssetList[k]) {
                    depositAssetList[k] = depositAssetList[depositAssetList.length - 1];
                    depositAssetList.pop();

                    allowAssetToDeposit[_assetList[i]] = false;
                }
            }

        }
    }
```

Listing 3.7: Example Privileged Operations in `Ownablee`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged account may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the administrative privileges to the intended DAO-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team confirmed the `owner` will be transferred to a multi-sig account.

3.8 Possibly Out-of-Sync Reward Boost in StakingPool

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StakingPool`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [2]

Description

To facilitate the protocol engagement, the `Vabble` protocol has a built-in `StakingPool` support. Users may stake `VAB` tokens to receive rewards. Moreover, to encourage users to participate in the protocol governance, the reward may be boosted with the governance involvement. In the process of reviewing the reward boost logic, we notice the current approach needs to be revisited.

To elaborate, we show below the implementation of the related `calcRewardAmount()` function. This function calculates the staking reward amount for the given user. We notice the computed reward amount is amplified with a factor, i.e., `rewardAmount * voteCount / proposalCount`, where `voteCount` represents the total number of voted protocols by the user and `proposalCount` denotes the number of protocols which are active during the user staking time period. As a result, these two states `voteCount` and `proposalCount` may be out-of-sync with respect to the staker lifetime.

```
    function calcRewardAmount(address _customer) public view returns (uint256 amount_) {
        Stake memory si = stakeInfo[_customer];
        require(si.stakeAmount > 0, "calcRewardAmount: Not staker");

        uint256 minAmount = 10**IERC20Metadata(IOwnablee(OWNABLE).PAYOUT_TOKEN()).
            decimals() / 100;
        require(si.stakeAmount > minAmount, "calcRewardAmount: less amount than 0.01");

        // Get proposal count started in withdrawable period of customer
        uint256 proposalCount = 0;
        for(uint256 i = 0; i < proposalCreatedTimeList.length; i++) {
            if(proposalCreatedTimeList[i] > si.stakeTime && proposalCreatedTimeList[i] <
                 si.withdrawableTime) {
                proposalCount += 1;
            }
        }

        uint256 rewardPercent = __rewardPercent(si.stakeAmount); // 0.0125*1e8 = 0.0125%
```

```

        // Get time with accuracy(10**4) from after lockPeriod
        uint256 period = (block.timestamp - si.stakeTime) * 1e4 / 1 days;
        uint256 rewardAmount = totalRewardAmount * rewardPercent * period / 1e10 / 1e4;

        // if no proposal then full rewards, if no vote for 5 proposals then no rewards,
             if 3 votes for 5 proposals then rewards*3/5
        if(proposalCount > 0) {
            if(si.voteCount == 0) {
                rewardAmount = 0;
            } else {
                uint256 countVal = (si.voteCount * 1e4) / proposalCount;
                rewardAmount = rewardAmount * countVal / 1e4;
            }
        }

        // If customer is film board member, more rewards(25%)
        if(IProperty(DAO_PROPERTY).isBoardWhitelist(_customer) == 2) {
            rewardAmount = rewardAmount + rewardAmount * IProperty(DAO_PROPERTY).
                boardRewardRate() / 1e10;
        }

        amount_ = rewardAmount;
    }
```

Listing 3.8: `StakingPool::calcRewardAmount()`

Recommendation Revisit the above routine to properly compute the boost factor for the staking reward.

Status This issue has been fixed in the following commit: `9a8f446`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Vabble` protocol, which provides `WEB3` projects and filmmakers with an easy way to finance and distribute `WEB3` content such as `Films`, `Series`, and `Animations`. Navigating multiple infrastructures, high upfront costs, and extensive networking make funding `WEB3` films and series difficult. `Vabble` has built a unified `WEB3` content financing platform by streamlining the funding process, eliminating upfront costs for creators, studios, and filmmakers, and letting them concentrate on what they do best. Moreover, it developed a `WEB3` content distribution infrastructure that brings value to investors and offers a fresh content experience for communities and consumers. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[10] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

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

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