SMART CONTRACT AUDIT REPORT

for

TipsyCoin Gin

Prepared By: Xiaomi Huang

PeckShield

September 30, 2022

Document Properties

Client	TipsyCoin
Title	Smart Contract Audit Report
Target	TipsyCoin Gin
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 30, 2022	Shulin Bie	Final Release
1.0-rc	September 29, 2022	Shulin Bie	Release Candidate

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of TipsyCoin Gin, 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 TipsyCoin Gin

Gin ($gin) is a multi-chain utility token that users will be able to earn and spend all across the TipsyCoin ecosystem. Gin will feature as the premium currency in all the play-and-earn, blockchain backed apps and games. It also acts as the reward token for users who stake their assets in the TipsyStake platform. In particular, as a cross-chain native token, Gin can be bridged and spent on all the TipsyCoin supported blockchains.

Table 1.1: Basic Information of TipsyCoin Gin

Item	Description
Target	TipsyCoin Gin
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	September 30, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the OwnableKeepable.sol file is out of the audit scope.

- https://github.com/TipsyCoin/TipsyGin.git (aa6bdc6)

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

- https://github.com/TipsyCoin/TipsyGin.git (a165d4f)

1.2 About PeckShield

PeckShield Inc. [7] 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 [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.

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

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

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) 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 `TipsyCoin Gin` 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	1	■
Low	0	
Informational	1	■
Total	2	

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 1 medium-severity vulnerability and 1 informational recommendation.

Table 2.1: Key TipsyCoin Gin Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-002	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Redundant State/Code Removal

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Gin
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

In the Gin contract, the multisigMint() routine is designed to mint a certain amount (specified by the input amount parameter) of Gin token to the recipient (specified by the input to parameter). While examining its logic, we notice there is some redundant code that can be safely removed.

To elaborate, we show below the related code snippet of the Gin contract. Inside the multisigMint () routine, the privileged Minters' signatures will be validated and the requiredSigs specified amount of Minters' signatures are required. Additionally, the requirement of require(requiredSigs >= MIN_SIGS , "REQUIRED_SIGS_TOO_LOW") (line 131) is called to prevent the potential compromised key vulnerability. However, after further analysis, we notice there is the same protection logic inside the setRequiredSigs () routine (line 83) when the requiredSigs storage variable is configured. Given this, we suggest to remove the redundant protection in the multisigMint() routine.

```
82    function setRequiredSigs(uint8 _numberSigs) public onlyOwner returns (uint8) {
83        require(_numberSigs >= MIN_SIGS, "SIGS_BELOW_MINIMUM");
84        emit RequiredSigs(requiredSigs, _numberSigs);
85        requiredSigs = _numberSigs;
86        return _numberSigs;
87    }
```

Listing 3.1: Gin::setRequiredSigs()

```
129   function multisigMint(address minter, address to, uint256 amount, uint256 deadline,
          bytes32 _depositHash, bytes memory signatures) external whenNotPaused returns(
          bool) {
```

```
        require(deadline >= block.timestamp, "MINT_DEADLINE_EXPIRED");
        require(requiredSigs >= MIN_SIGS, "REQUIRED_SIGS_TOO_LOW");
        bytes32 dataHash;
        dataHash =
            keccak256(
                abi.encodePacked(
                    "\x19\x01",
                    DOMAIN_SEPARATOR(),
                    keccak256(
                        abi.encode(
                            keccak256(
                                "multisigMint(address minter,address to,uint256 amount,uint256 nonce,uint256 deadline,bytes signatures)"
                            ),
                            minter,
                            to,
                            amount,
                            nonces[minter]++,
                            deadline
                        )
                    )
                )
            );
        checkNSignatures(minter, dataHash, requiredSigs, signatures);
        _mint(to, amount);
        emit Withdrawal(to, amount, _depositHash);
        return true;
    }
```

Listing 3.2: `Gin::multisigMint()`

Recommendation Consider the removal of the redundant code.

Status The issue has been addressed by the following commit: `1a85050`.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Gin/SolMateERC20`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `TipsyCoin` `Gin` implementation, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., manage the privileged `Minter`). In the following, we show the representative functions potentially affected by the privilege of the owner account.

```
    function permitContract(address _newSigner) public onlyOwner returns (bool) {
        emit ContractPermission(_newSigner, true);
        return _addContractMinter(_newSigner);
    }

    function permitSigner(address _newSigner) public onlyOwner returns (bool) {
        emit SignerPermission(_newSigner, true);
        return _addMintSigner(_newSigner);
    }
```

Listing 3.3: Gin::permitContract()&&permitSigner()

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest a multi-sig account plays the privileged owner account to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status The issue has been confirmed by the team. The team intends to introduce multi-sig mechanism to mitigate this issue.

4 | Conclusion

In this audit, we have analyzed the `TipsyCoin Gin` design and implementation. `Gin` (`$gin`) is a multi-chain utility token that users will be able to earn and spend all across the `TipsyCoin` ecosystem. It will act as the reward token for users who stake their assets in the `TipsyStake` platform. In particular, as a cross-chain native token, `Gin` can be bridged and spent on all the `TipsyCoin` supported blockchains. 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 CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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