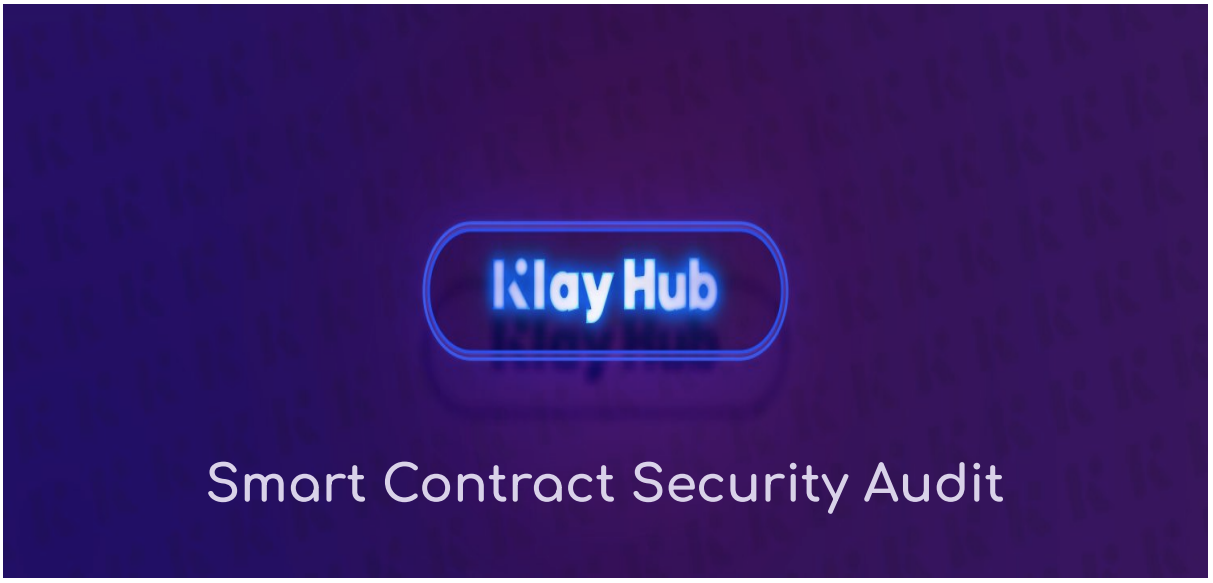

Prepared by

S2W
Safe and Secure World

S2W

Contents

Executive Summary 3

Introduction 4

About KlayHub

Scope

Disclaimer

Methodology 6

Audit Items

Risk Rating

Findings 8

Summary

Recommendations

Conclusion 10

Executive Summary

As requested by the KlayHub team, S2W conducted an audit to evaluate the security of KlayHub's token contracts from Nov 8 to 12. During the audit, we have carefully analyzed the workflows of all contracts in the scope to understand how the contracts were designed. The state of the art methods for code analysis(e.g., symbolic execution, fuzzing) are used to discover security vulnerabilities, expose possible semantic inconsistencies between the implementation and the design of smart contracts. In this audit, we have found no significant security issues except a few minor recommendations that focus on improving code safety and optimizing gas usage.

Introduction

About KlayHub

KlayHub is a DeFi platform which aims to provide various financial services(e.g., swaps with an Automated Market Maker) and it launched on the Klaytn blockchain. With the KlayHub, users can easily trade their virtual assets and also participate in a variety of profit opportunities such as NFT minting. Now, KlayHub team has published basic token contracts for both FT and NFT as the beginning of the big picture. The contracts have been implemented based on several open-source libraries(e.g., klaytn-contracts, https://github.com/klaytn/klaytn-contracts).

Project Information

Name	KlayHub
Platform	Klaytn
Contract Type	KIP7 & KIP17(compatible with ERC20 & ERC721)
Language	Solidity
Official Website	https://www.klayhub.com/

Scope

The audit covered token contracts of KlayHub including both FT and NFT for the Klaytn blockchain.

Repository Information

Location(URL)	https://github.com/klayhub/contract
Commit ID	7f0c0e829a8255d7c8d935cc9d4739f6b0f358aa
sha256(.zip)	9ede09195e601c9c560454b6a96fb87b506edd75d4f519375d535af13a593738

Contract Name	Location
KIP7	contracts/token/KIP7.sol
KIP17	contracts/token/KIP17.sol

Disclaimer

While we ensure that this audit is conducted in professional and sophisticated ways, it does not guarantee that we uncover all of the security vulnerabilities on the given smart contracts. In general, an assessment from a single party is not considered as comprehensive so long as it could be confirmed to discover all potential security issues. Therefore, S2W recommends performing multiple independent assessments or even bug bounty programs to minimize security risks. Finally, there is no reason to consider this audit report as investment advice.

Methodology

S2W performs code auditing for smart contracts via automated testing and manual analysis by experts in cyber security. The audit according to the following procedures:

- **General Code Analysis(Basic):** Static and dynamic code analysis of smart contracts is conducted with instructions that S2W provided to identify coding bugs and potential vulnerabilities.
- **Correctness Assessment(Advanced):** Further in-depth analysis is manually performed to evaluate the correctness of smart contracts(e.g., finding logic bugs). The assessment covers workflows, business logics and the actual behaviors of smart contracts to uncover possible pitfalls or bugs.
- **Best Practices Review(Additional):** Smart contracts are reviewed to improve efficiency, effectiveness and security based on the standards and recommendations of established industry(e.g., financial circle).

Audit Items

The following list represents items that were checked during the audit but not limited to:

- Reentrancy(on a single or cross-function)
- Arithmetic: integer overflow and underflow
- Unchecked Low Level Call
- Denial of Service(DoS)
- Access Control
- Bad Randomness
- Front Running
- Timestamp Dependence
- Oracle Security: improper oracle usage
- Business Logic(contradicting the specification)
- Gas Usage
- Best Practices Violation

Risk Rating

Estimating the severity of security risks is as important as discovering vulnerabilities. We define risk ratings for the security issues of smart contracts based on the well-known OWASP Risk Rating Methodology to carefully decide the severity of risks. The risk is commonly determined as the combination of impact factor and likelihood which could be categorized into three levels: **Low**, **Medium** and **High**.

- **Impact** defines how bad damages can get from the attack scenario.
- **Likelihood** defines the probable rate at which the vulnerabilities may occur.

Determining final severity of security risks can be figured out by combining both impact and likelihood. We have categorized it into four levels: **Low**, **Medium**, **High** and **Critical**. Also, if any vulnerability is discovered with no likelihood or impact, then it would be listed as an additional minor level: **Recommendation**.

Overall Risk Severity

Factor	Likelihood			
Impact	Level	Low	Medium	High
	Low	Low	Low	Medium
	Medium	Low	Medium	High
	High	Medium	High	Critical

Findings

Summary

On the whole, the given smart contracts are well reasoned and engineered. But then, most of them originated from the libraries that have been already published on the official github organization of Klaytn(ref. https://github.com/klaytn/klaytn-contracts) to help building blockchain applications on Klaytn. In addition, some implementations were derived from openzepplin contracts which is one of the most popular open-source libraries for secure smart contract development.

From the assessments, S2W has uncovered no security issue that may have correctness or security impacts but, there are two minor considerations for better gas usage and code safety which does not impose any immediate security impacts.

Recommendations

ID	Category	Description	Importance
#1	Optimization	Reduce gas usage for reverted token transfers	Minor
#2	Safety	Consider using non-standard functions of ERC20 implementation	Minor

Recommendations #1: Reduce gas usage for reverted token transfers

Anyone who wants to make a token transfer could use the *transferFrom* function to send some tokens from sender to recipient with three parameters: sender's and receiver's addresses and token amount to transfer. The function works by conducting two internal function calls. First, the contract literally made a token transfer by conducting *_transfer* function call which decreases sender's balance and increases receiver's balance. After the *_transfer* function call is completed properly, the contract deducts the allowance of the caller for sender's tokens as much as the spent amount by means of *_approve* function.

Assume that the sender has enough token balances to send(i.e., more than the amount specified as a parameter), but she has approved the caller to spend her tokens only fewer than the amount or even zero. In this scenario, when the caller calls *transferFrom* function to transfer sender's tokens, the internal *_transfer* function call is executed correctly. Which means that the balances of sender's and receiver's addresses would be changed and also a transfer log would be written.

However, during the *_approve* function call, the overall transaction would be reverted with an exception occurred by *SafeMath* contract(note that, there is no sufficient allowance to the caller to spend sender's tokens). As a result, the gas used in *_transfer* function call would be consumed meaninglessly.

To get rid of this kind of gas waste, we recommend that the contract confirm whether the caller has sufficient allowance for the sender's token to transfer before it makes *_transfer* function call. Also, if the external contract checks the allowance before who is willing to conduct the *transferFrom* function of this KIP7 contract, the waste is no longer occurring.

```
function transferFrom(address sender, address recipient, uint256 amount) public returns (bool) {
    require(_allowances[sender][msg.sender] >= amount, "KIP7: transfer by unapproved caller")
    _transfer(sender, recipient, amount);
    _approve(sender, msg.sender, _allowances[sender][msg.sender].sub(amount));
    return true;
}
```

Recommendations #2: Consider using non-standard functions of ERC20 implementation

We would suggest using two non-standard functions(i.e., *decreaseAllowance* and *increaseAllowance*) on ERC20 implementation to mitigate the well-known race condition problems around setting allowances. Using these functions instead of *approve* function can help to resolve such issues.

Assume that Alice has approved Bob to spend 'n' of her tokens. For some reasons, Alice made a decision to decrease the allowance to 'm' tokens and thus she submits a function call to change her approval to m. At this time, Bob submits a transfer request sending n of her tokens to whom with much higher gas price than the her transaction. Thus, Bob's transaction executes first and sets Bob's approval to zero and then, after Alice's transaction was executed, Bob gains an additional allowance m. In this scenario, using *decreaseAllowance* prevents approval of an additional allowance m to Bob.

Detail description about this issues can be found at *IERC20.approve* of following:

- https://docs.openzeppelin.com/contracts/2.x/api/token/erc20

Conclusion

In this audit, S2W has reviewed the design and implementation of KlayHub token contracts. Some implementations we analyzed come from common popular open source libraries and they are well structured and neatly organized as well. While there are a few minor recommendations, we have not found any critical security issues that would be a threat to the safety of token economies.

Note that the scope of this audit is not a major part of the KlayHub DeFi services. We suggest conducting audits for the rest of the system as well to lessen the security risks.

End of Document