DEFIMOON

be secure

Smart Contract Audit Report

November, 2022

ThunderEV

DEFIMOON PROJECT

Audit and Development

CONTACTS

https://defimoon.org
audit@defimoon.org
defimoon_org
defimoonorg
defimoon
defimoonorg

DEFIMOON

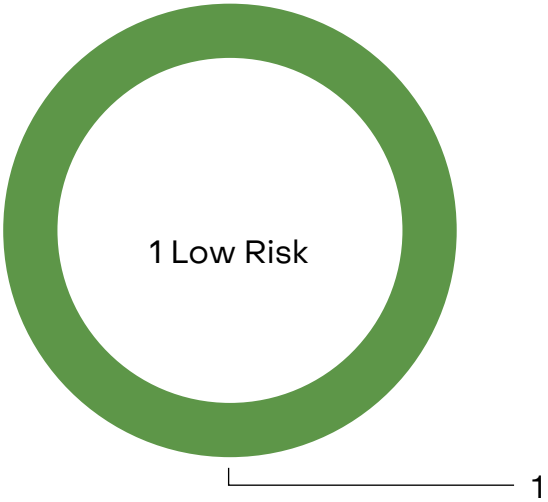
be secure

November 26th 2022

This reaudit report was prepared by Defimoon for Thunder EV

Audit information

Description	THEV ERC20 token
Audited files	Thev.sol IBEP20.sol Ownable.sol Address.sol SafeMath.sol Variables.sol DateTime.sol
Timeline	31st October 2022 - 26th November 2022
Audited by	Daniil Rashin
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Specification	Whitepaper
Docs quality	Low
Source code	Verified
Network	Ethereum mainnet
Status	Passed



	High Risk	A fatal vulnerability that can cause the loss of all Tokens / Funds.
	Medium Risk	A vulnerability that can cause the loss of some Tokens / Funds.
	Low Risk	A vulnerability which can cause the loss of protocol functionality.
	Informational	Non-security issues such as functionality, style, and convention.

Disclaimer

This audit is not financial, investment, or any other kind of advice and could be used for informational purposes only. This report is not a substitute for doing your own research and due diligence should always be paid in full to any project. Defimoon is not responsible or liable for any loss, damage, or otherwise caused by reliance on this report for any purpose. Defimoon has based this audit report solely on the information provided by the audited party and on facts that existed before or during the audit being conducted. Defimoon is not responsible for any outcome, including changes done to the contract/contracts after the audit was published. This audit is fully objective and only discerns what the contract is saying without adding any opinion to it. Defimoon has no connection to the project other than the conduction of this audit and has no obligations other than to publish an objective report. Defimoon will always publish its findings regardless of the outcome of the findings. The audit only covers the subject areas detailed in this report and unless specifically stated, nothing else has been audited. Defimoon assumes that the provided information and materials were not altered, suppressed, or misleading. This report is published by Defimoon, and Defimoon has sole ownership of this report. Use of this report for any reason other than for informational purposes on the subjects reviewed in this report including the use of any part of this report is prohibited without the express written consent of Defimoon. In instances where an auditor or team member has a personal connection with the audited project, that auditor or team member will be excluded from viewing or impacting any internal communication regarding the specific audit.

Audit Information

Defimoon utilizes both manual and automated auditing approach to cover the most ground possible. We begin with generic static analysis automated tools to quickly assess the overall state of the contract. We then move to a comprehensive manual code analysis, which enables us to find security flaws that automated tools would miss. Finally, we conduct an extensive unit testing to make sure contract behaves as expected under stress conditions.

In our decision making process we rely on finding located via the manual code inspection and testing. If an automated tool raises a possible vulnerability, we always investigate it further manually to make a final verdict. All our tests are run in a special test environment which matches the "real world" situations and we utilize exact copies of the published or provided contracts.

While conducting the audit, the Defimoon security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Defimoon assesses the risks and assigns a risk level to each section together with an explanatory comment.

Thev reAudit overview

No Major issues found at reaudit.
This conclusion is based on the problems described below.

- **DFM-1**

First and most critical issue is the lack of reusable code (OpenZeppelin libraries).
Greatest profit this approach brings to your project is security. It is much safer to inherit your ERC20 token from the most frequently used solution, which is well-known and trusted by millions of developers.
All the custom features should be added on top of that, so your code is easy to read and maintain.
Consider inhereting your contract from ERC20 instead of IBEP20, after all it refers to Binance Smart Chain, while THEV is deployed in Ethereum Mainnet.

- **DFM-2**

There is a duplicate variable _owner which is already implemented in Ownable.sol, moreover this leads to unexpected behaviour after transferOwnership from Ownable is called since burnToken compares _owner balance.
Consider removing this variable.

- **DFM-3**

Another critical issue is the misuse of time literals in cases of token lock duration.
next_release_time in wallet_details uses classic epoch timestamp, while such values as _director_lock_days and _investor_lock_days are treated as days without using the correct literal, which is later multiplied by seconds. So instead of locking tokens for n days, it is done for n seconds, which is definitely unwanted.
Consider adding days literal to _investor_lock_days, _investor_release_every_days_after_locking, _director_lock_days, _director_release_every_days_after_locking.

- **DFM-4**

Function addInvestorWallet checks if account is already a director instead of investor.
Consider changing is_director to is_investor in the require statement.

- **DFM-5**

burnToken function substitutes amount of token from _owner (DFM-2) balance, shrinking _total_supply by amount * 10**18.

- **DFM-6**

_contribution_airdrop has to for loops with same conditions, which is highly gas-ineffective.
Consider using one loop.

- **DFM-7**

updateLockingConditioins has incorrect logic because of next_release_time (DFM-3) incorrect values.

Variables Audit overview

No major issues were found.

Library holding essential values.

DateTime Audit overview

No major issues were found.

DateTime functionality implementation.

Address Audit overview

No major issues were found.

Address functionality implementation.
It is recommended to use OpenZeppelin implementation.

Ownable Audit overview

No major issues were found.

Ownable functionality implementation.

It is recommended to use OpenZeppelin implementation.

SafeMath Audit overview

No major issues were found.

SafeMath library implementation.

It is recommended to use OpenZeppelin implementation.

IBEP20 Audit overview

No major issues were found.

It is always recommended to reuse industry standard libraries while implementing common functionality.
In this case, IERC20 interface should be used.
Although it is an interface with common ERC20 functions, name IBEP20 may confuse the user.
Using OpenZeppelin interface instead is considered more secure and effective way of implementing ERC20 tokens.
Also adding custom events to standard interface makes it pointless since the interface is modified at this point.
getOwner() functionality is already implemented in Ownable.sol and does not fit into interface definition as well.

Summary of findings

According to the standard audit assessment, the audited solidity smart contracts are fairly secure and are could be used for production, however it is recommended to implement the recommendations fully to secure and optimize the codebase.

ID	Description	Severity	Status
DFM-1	ERC20 inheritance	Low Risk	Resolved
DFM-2	Duplicate _owner	High Risk	Resolved
DFM-3	Incorrect timestamp calculation	Medium Risk	Resolved
DFM-4	addInvestor incorrect require condition	Low Risk	Resolved
DFM-5	burnToken math	High Risk	Mitigated
DFM-6	contribution_airdrop for loops	Low Risk	Open
DFM-7	updateLockingConditions logic problems	High Risk	Resolved
DFM-8	OpenZeppelin libraries	Low Risk	Mitigated

Application security checklist

Compiler errors	Passed
Possible delays in data delivery	Passed
Timestamp dependence	Passed
Integer Overflow and Underflow	Not Passed
Race Conditions and Reentrancy	Passed
DoS with Revert	Passed
DoS with block gas limit	Not Passed
Methods execution permissions	Passed
Private user data leaks	Passed
Malicious Events Log	Passed
Scoping and Declarations	Passed
Uninitialized storage pointers	Passed
Arithmetic accuracy	Not Passed
Design Logic	Passed
Cross-function race conditions	Passed

Detailed Audit Information

Contract Programming

Solidity version not specified	Passed
Solidity version too old	Passed
Integer overflow/underflow	Not Passed
Function input parameters lack of check	Passed
Function input parameters check bypass	Passed
Function access control lacks management	Passed
Critical operation lacks event log	Passed
Human/contract checks bypass	Passed
Random number generation/use vulnerability	Passed
Fallback function misuse	Passed
Race condition	Passed
Logical vulnerability	Passed
Other programming issues	Passed

Code Specification

Visibility not explicitly declared	Passed
Variable storage location not explicitly declared	Passed
Use keywords/functions to be deprecated	Passed
Other code specification issues	Passed

Gas Optimization

Assert () misuse	Passed
High consumption ‘for/while’ loop	Not Passed
High consumption ‘storage’ storage	Passed
“Out of Gas” Attack	Passed
Public function could be external	Passed

Findings

DFM-1 "ERC20 inheritance"

Severity: Low Risk

Status: Resolved

Description:
Contract inherits IBEP20 interface which is used on Binance Smart Chain, also interface does not implement any functions, so it is done in the THEV contract.

Recommendation:
Inherit THEV contract from OpenZeppelin ERC20 contract, which would save much effort and remove the need to implement core functions.

Reautdit comments:
Though contract is not inhereted from OpenZeppelin implementation, interface was renamed so that it matches the blockchain the contract was deployed to.

DFM-2 "Duplicate _owner"

Severity: High Risk

Status: Resolved

Description:
THEV contract has _owner and Ownable's owner, which is both used in contract logic, after Ownable's transferOwnership is called, THEV's _owner remains the same.

Recommendation:
Remove _owner and rely on owner from Ownable library.

Reautdit comments:
Ownable functionality was moved to the main contract.

DFM-3 "Incorrect timestamp calculation"

Severity: Medium Risk

Status: Resolved

Description:
wallet_details.next_release_time is epoch timestamp, but treated as days literal while updated in different functions.

Recommendation:
Add correct time in update_locking_conditions, addInvestorWallet, addDirectorWallet to this variable.

Reautdit comments:
Time literals were updated so that variables are treated properly.

DFM-4 "addInvestor incorrect require condition"

Severity: Low Risk

Description:
Function checks if provided address is already a director instead of investor.

Recommendation:
Change is_director to is_investor in the first require statement in addInvestor function.

Reautdit comments:
Require conditions now checks if address is already an investor.

DFM-5 "burnToken math"

Severity: High Risk

Status: Mitigated

Description:
burnToken function substitutes different values from _owner balance and total_balance, which besides from token loss may lead to integer overflow.

Recommendation:
Use Ownable's owner and remove digits multiplication in burnToken function.

Reautdit comments:
Input value is still treated as non-bignumber, that is multiplied inside the function, which may lead to integer overflows.
However, client states that the problems described will not occur, as the code only interacts with the function through controlled interface, only sending correct values — this still leaves the large surface for possible value fuzzing attacks, if control of the interface is compromised and it is recommended to resolve the issue following the recommendations for the safer usage.

DFM-6 "contribution_airdrop for loops"

Severity: Low Risk

Status: Open

Description:
Both for loops in contribution_airdrop function has same conditions, which is gas-ineffective.

Recommendation:
Remove redundant for loop.

Reautdit comments:
Loops with same conditions still exist which affects gas usage.

DFM-7 "updateLockingConditions logic problems"

Severity: High Risk

Status: Resolved

Description:
DFM-3 leads to unexpected behaviour in updateLockingConditions function since locking conditions are in seconds.

Recommendation:
After fixing DFM-3 look through the logic again to make sure timestamp is always treated the same way.

Reautdit comments:
Issue is resolved.

DFM-8 "OpenZeppelin libraries"

Severity: Low Risk

Status: Mitigated

Description:
All imported contracts are already implemented by OpenZeppelin

Recommendation:
Use well-known and already tested solutions instead of rewriting the same code again.

Reautdit comments:
Though OpenZeppelin libraries are still not used, all the conflicts were resolved.

Automated Analyses

Slither

Slither has reported 98 findings. These results were either related to false positives or have been integrated in the findings or best practices of this report.

Adherence to Best Practices

1. Use OpenZeppelin solutions.
2. Use correct time literals.
3. Use correct decimals multiplications.

Methodology

Manual Code Review

We prefer to work with a transparent process and make our reviews a collaborative effort. The goal of our security audits is to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, review open issue tickets, and investigate details other than the implementation.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system to make a final decision.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Appendix A — Finding Statuses

Resolved	Contracts were modified to permanently resolve the finding
Mitigated	The finding was resolved by other methods such as revoking contract ownership or updating the code to minimize the effect of the finding
Acknowledged	Project team is made aware of the finding
Open	The finding was not addressed