Tron2Get

Smart Contract Review

Deliverable: Smart Contract Audit Report

Security Report

July 2021

Disclaimer

The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the Company. The content, conclusions and recommendations set out in this publication are elaborated in the specific for only project.

eNebula Solutions does not guarantee the accuracy of the data included in this study. All representations, warranties, undertakings and guarantees relating to the report are excluded, particularly concerning – but not limited to – the qualities of the assessed projects and products. Neither the Company nor any personating on the Company's behalf may be held responsible for the use that may be made of the information contained herein.

eNebula Solutions retains the right to display audit reports and other content elements as examples of their work in their portfolio and as content features in other projects with protecting all security purpose of customer. The report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all vulnerabilities fixed - upon a decision of the Customer.

Report Summary

Title	**Tron2Get Smart Contract Audit**		
Project Owner	Tron2Get Limited		
Type	Public		
Reviewed by	Vatsal Raychura	**Revision date**	09/07/2021
Approved by	eNebula Solutions Private Limited	**Approval date**	09/07/2021
		Nº Pages	**24**

Overview

Background

Tron2Get requested that eNebula Solutions perform an Extensive Smart Contract audit of their Smart Contract.

Project Dates

The following is the project schedule for this review and report:

- **July 9**: Smart Contract Review Completed *(Completed)*
- **July 9**: Delivery of Smart Contract Audit Report *(Completed)*

Review Team

The following eNebula Solutions team member participated in this review:

- Sejal Barad, Security Researcher and Engineer
- Vatsal Raychura, Security Researcher and Engineer

Coverage

Target Specification and Revision

For this audit, we performed research, investigation, and review of the smart contract of Tron2Get.

The following documentation repositories were considered in-scope for the review:

- Tron2Get Project: https://bscscan.com/address/0xd5097c5eC23E324bF1516501BEa34E8Cf078eafa#code

Introduction

Given the opportunity to review Tron2Get Project's smart contract source code, we in the report outline 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 is ready to launch after resolving the mentioned issues, there are no critical or high issues found related to business logic, security or performance.

About Tron2Get: -

Item	Description
Issuer	Tron2Get
Website	https://tron2get.com/
Type	BEP20
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 9, 2021

The Test Method Information: -

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open-source code, non-open-source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant effect on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project party should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.

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
	Business Logics Review

Advanced DeFi Scrutiny	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

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 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 ex pilotable 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.

Findings

Summary

Here is a summary of our findings after analyzing the Tron2Get's Smart Contract. During the first phase of our audit, we studied the smart contract sourcecode and ran our in-house static code analyzer through the Specific tool. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by 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	No. of Issues
Critical	0
High	0
Medium	1
Low	7
Total	8

We have so far identified that there are potential issues with severity of **0 Critical, 0 High, 1 Medium, and 7 Low**. Overall, these smart contracts are well- designed and engineered, though the implementation can be improved and bug free by common recommendations given under POCs.

Functional Overview

($) = payable function # = non-constant function	[Pub] public [Ext] external [Prv] private [Int] internal

```
+ [Lib] SafeMath
   - [Int] add
   - [Int] sub
   - [Int] sub
   - [Int] mul
   - [Int] div
   - [Int] div
   - [Int] mod
   - [Int] mod

 +  TRON2GetNEW
   - [Pub] <Constructor> #
   - [Pub] invest ($)
   - [Pub] withdraw #
   - [Pub] getContractBalance
   - [Pub] getContractBalanceRate
   - [Pub] getUserPercentRate
   - [Pub] getLeaderBonusRate
   - [Pub] getLeaderBonusRate_2
   - [Pub] getUserDividends
   - [Pub] getUserCheckpoint
   - [Pub] getUserReferrer
```

```
- [Pub] getUserDownlineCount
- [Pub] getUserReferralBonus
- [Pub] getUserAvailableBalanceForWithdrawal
- [Pub] isActive
- [Pub] getUserDepositInfo
- [Pub] getUserAmountOfDeposits
- [Pub] getUserTotalDeposits
- [Pub] getUserTotalWithdrawn
- [Int] isContract
- [Pub] getHoldBonus
```

Detailed Results

Issues Checking Status

1. DoS With Block Gas Limit

- SWC ID:128
- Severity: Low
- Location: https://bscscan.com/address/0xd5097c5eC23E324bF1516501BEa34E8Cf078eafa#code
- Relationships: CWE-400: Uncontrolled Resource Consumption
- Description: Loop over unbounded data structure. Gas consumption in function "withdraw" in contract "TRON2GetNEW" depends on the size of data structures or values that may grow unboundedly. If the data structure grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

```
		for (uint256 i = 0; i < user.deposits.length; i++) {

			if (user.deposits[i].withdrawn < user.deposits[i].amount.mul(2)) {

				if (user.deposits[i].start > user.checkpoint) {
					dividends = (user.deposits[i].amount.mul(userPercentRate).div(PERCENTS_DIVIDER))
						.mul(block.timestamp.sub(user.deposits[i].start))
						.div(TIME_STEP);
				} else {
					dividends = (user.deposits[i].amount.mul(userPercentRate).div(PERCENTS_DIVIDER))
						.mul(block.timestamp.sub(user.checkpoint))
						.div(TIME_STEP);
				}

				if (user.deposits[i].withdrawn.add(dividends) > user.deposits[i].amount.mul(2)) {
					dividends = (user.deposits[i].amount.mul(2)).sub(user.deposits[i].withdrawn);
				}
				user.deposits[i].withdrawn = user.deposits[i].withdrawn.add(dividends); /// changing of storage data
				totalAmount = totalAmount.add(dividends);
			}
		}
```

- Remediations: Caution is advised when you expect to have large arrays that grow over time. Actions that require looping across the entire data structure should be avoided. If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions.

2. DoS With Block Gas Limit

- SWC ID:128
- Severity: Low
- Location: https://bscscan.com/address/0xd5097c5eC23E324bF1516501BEa34E8Cf078eafa#code
- Relationships: CWE-400: Uncontrolled Resource Consumption
- Description: Loop over unbounded data structure. Gas consumption in function "getUserDividends" in contract "TRON2GetNEW" depends on the size of data structures or values that may grow unboundedly. If the data structure grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose..

```
for (uint256 i = 0; i < user.deposits.length; i++) {

	if (user.deposits[i].withdrawn < user.deposits[i].amount.mul(2)) {

		if (user.deposits[i].start > user.checkpoint) {

			dividends = (user.deposits[i].amount.mul(userPercentRate).div(PERCENTS_DIVIDER))
				.mul(block.timestamp.sub(user.deposits[i].start))
				.div(TIME_STEP);

		} else {

			dividends = (user.deposits[i].amount.mul(userPercentRate).div(PERCENTS_DIVIDER))
				.mul(block.timestamp.sub(user.checkpoint))
				.div(TIME_STEP);

		}

		if (user.deposits[i].withdrawn.add(dividends) > user.deposits[i].amount.mul(2)) {
			dividends = (user.deposits[i].amount.mul(2)).sub(user.deposits[i].withdrawn);
		}

		totalDividends = totalDividends.add(dividends);

		/// no update of withdrawn because that is view function

	}

}
```

- Remediations: Caution is advised when you expect to have large arrays that grow over time. Actions that require looping across the entire data structure should be avoided. If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions.

3. DoS With Block Gas Limit

- SWC ID:128
- Severity: Low
- Location: https://bscscan.com/address/0xd5097c5eC23E324bF1516501BEa34E8Cf078eafa#code
- Relationships: CWE-400: Uncontrolled Resource Consumption
- Description: Loop over unbounded data structure. Gas consumption in function "getUserTotalDeposits" in contract "TRON2GetNEW" depends on the size of data structures or values that may grow unboundedly. If the data structure grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

```
for (uint256 i = 0; i < user.deposits.length; i++) {
    amount = amount.add(user.deposits[i].amount);
}
```

- Remediations: Caution is advised when you expect to have large arrays that grow over time. Actions that require looping across the entire data structure should be avoided. If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions.

4. DoS With Block Gas Limit

- SWC ID:128
- Severity: Low
- Location: https://bscscan.com/address/0xd5097c5eC23E324bF1516501BEa34E8Cf078eafa#code
- Relationships: CWE-400: Uncontrolled Resource Consumption
- Description: Loop over unbounded data structure. Gas consumption in function "getUserTotalWithdrawn" in contract "TRON2GetNEW" depends on the size of data structures or values that may grow unboundedly. If the data structure grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

```
for (uint256 i = 0; i < user.deposits.length; i++) {
    amount = amount.add(user.deposits[i].withdrawn);
}
```

- Remediations: Caution is advised when you expect to have large arrays that grow over time. Actions that require looping across the entire data structure should be avoided. If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions.

5. Message call with hardcoded gas amount

- SWC ID:134
- Severity: Low
- Location: https://bscscan.com/address/0xd5097c5eC23E324bF1516501BEa34E8Cf078eafa#code
- Relationships: CWE-655: Improper Initialization
- Description: Call with hardcoded gas amount. The highlighted function call forwards a fixed amount of gas. This is discouraged as the gas cost of EVM instructions may change in the future, which could break this contract's assumptions. If this was done to prevent reentrancy attacks, consider alternative methods such as the checks-effects-interactions pattern or reentrancy locks instead.

```
function invest(address referrer) public payable {
        require(msg.value >= INVEST_MIN_AMOUNT,"min amount is 0.0160 bnb");
        uint256 _amount=msg.value;
marketingAddress.transfer(_amount.mul(MARKETING_FEE).div(PERCENTS_DIVIDER));
        projectAddress.transfer(_amount.mul(PROJECT_FEE).div(PERCENTS_DIVIDER));
        emit FeePayed(msg.sender, _amount.mul(MARKETING_FEE.add(PROJECT_FEE)).div(PERCENTS_DIVIDER));
```

- Remediations: Avoid the use of transfer() and send() and do not otherwise specify a fixed amount of gas when performing calls. Use .call.value(...)("") instead. Use the checks-effects-interactions pattern and/or reentrancy locks to prevent reentrancy attacks.

6. Message call with hardcoded gas amount

- SWC ID:134
- Severity: Low
- Location: https://bscscan.com/address/0xd5097c5eC23E324bF1516501BEa34E8Cf078eafa#code
- Relationships: CWE-655: Improper Initialization
- Description:
- Call with hardcoded gas amount. The highlighted function call forwards a fixed amount of gas. This is discouraged as the gas cost of EVM instructions may change in the future, which could break this contract's assumptions. If this was done to prevent reentrancy attacks, consider alternative methods such as the checks-effects-interactions pattern or reentrancy locks instead.

```
        function invest(address referrer) public payable {
                require(msg.value >= INVEST_MIN_AMOUNT,"min amount is 0.0160 bnb");
                uint256 _amount=msg.value;
        marketingAddress.transfer(_amount.mul(MARKETING_FEE).div(PERCENTS_DIVIDER));
                projectAddress.transfer(_amount.mul(PROJECT_FEE).div(PERCENTS_DIVIDER));
                emit FeePayed(msg.sender, _amount.mul(MARKETING_FEE.add(PROJECT_FEE)).div(PERCENTS_DIVIDER));
```

- Remediations: Avoid the use of transfer() and send() and do not otherwise specify a fixed amount of gas when performing calls. Use .call.value(...)("") instead. Use the checks-effects-interactions pattern and/or reentrancy locks to prevent reentrancy attacks.

7. Message call with hardcoded gas amount

- SWC ID:134
- Severity: Low
- Location: https://bscscan.com/address/0xd5097c5eC23E324bF1516501BEa34E8Cf078eafa#code
- Relationships: CWE-655: Improper Initialization
- Description: Call with hardcoded gas amount. The highlighted function call forwards a fixed amount of gas. This is discouraged as the gas cost of EVM instructions may change in the future, which could break this contract's assumptions. If this was done to prevent reentrancy attacks, consider alternative methods such as the checks-effects-interactions pattern or reentrancy locks instead.

```
user.checkpoint = block.timestamp;

msg.sender.transfer(totalAmount);

totalWithdrawn = totalWithdrawn.add(totalAmount);

emit Withdrawn(msg.sender, totalAmount);

```

- Remediations: Avoid the use of transfer() and send() and do not otherwise specify a fixed amount of gas when performing calls. Use .call.value(...)("") instead. Use the checks-effects-interactions pattern and/or reentrancy locks to prevent reentrancy attacks.

8. DoS With Block Gas Limit

- SWC ID:128
- Severity: Low
- Location: https://bscscan.com/address/0xd5097c5eC23E324bF1516501BEa34E8Cf078eafa#code
- Relationships: CWE-400: Uncontrolled Resource Consumption
- Description: Implicit loop over unbounded data structure. Gas consumption in function "getUserDownlineCount" in contract "TRON2GetNEW" depends on the size of data structures that may grow unboundedly. The highlighted statement involves copying an array inside "users[userAddress]" from "storage" to "memory". When copying arrays from "storage" to "memory" the Solidity compiler emits an implicit loop.If the array grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

```
function getUserDownlineCount(address userAddress) public view returns(uint256, uint256, uint256,uint256, uint256, uint256,uint256, uint256, uint256,uint256) {
User memory _user=users[userAddress];
        return (_user.level1, _user.level2, _user.level3,
        _user.level4, _user.level5, _user.level6,
        _user.level7, _user.level8, _user.level9,
        _user.level10
        );
}
```

- Remediations: Caution is advised when you expect to have large arrays that grow over time. Actions that require looping across the entire data structure should be avoided. If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions.

Basic Coding Bugs

1. **Constructor Mismatch**
    - Description: Whether the contract name and its constructor are not identical to each other.
    - Result: PASSED
    - Severity: Critical
2. **Ownership Takeover**
    - Description: Whether the set owner function is not protected.
    - Result: PASSED
    - Severity: Critical
3. **Redundant Fallback Function**
    - Description: Whether the contract has a redundant fallback function.
    - Result: PASSED
    - Severity: Critical
4. **Overflows & Underflows**
    - Description: Whether the contract has general overflow or underflow vulnerabilities
    - Result: PASSED
    - Severity: Critical
5. **Reentrancy**
    - Description: Reentrancy is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
    - Result: PASSED
    - Severity: Critical
6. **MONEY-Giving Bug**
    - Description: Whether the contract returns funds to an arbitrary address.
    - Result: PASSED
    - Severity: High

7. **Blackhole**
   - Description: Whether the contract locks ETH indefinitely: merely in without out.
   - Result: PASSED
   - Severity: High

8. **Unauthorized Self-Destruct**
   - Description: Whether the contract can be killed by any arbitrary address.
   - Result: PASSED
   - Severity: Medium

9. **Revert DoS**
   - Description: Whether the contract is vulnerable to DoS attack because of unexpected revert.
   - Result: PASSED
   - Severity: Medium

10. **Unchecked External Call**
    - Description: Whether the contract has any external call without checking the return value.
    - Result: PASSED
    - Severity: Medium

11. **Gasless Send**
    - Description: Whether the contract is vulnerable to gasless send.
    - Result: PASSED
    - Severity: Medium

12. **Send Instead of Transfer**
    - Description: Whether the contract uses send instead of transfer.
    - Result: PASSED
    - Severity: Medium

13. Costly Loop

- Description: Whether the contract has any costly loop which may lead to Out-Of-Gas exception.
- Result: PASSED
- Severity: Medium

14. (Unsafe) Use of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: PASSED
- Severity: Medium

15. (Unsafe) Use of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: PASSED
- Severity: Medium

16. Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: PASSED
- Severity: Medium

17. Deprecated Uses

- Description: Whether the contract use the deprecated tx.origin to perform the authorization.
- Result: PASSED
- Severity: Medium

Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: PASSED
- Severity: Critical

Conclusion

In this audit, we thoroughly analyzed Tron2Get's Smart Contract. The current code base is well organized but there are promptly some Medium-level and low-level issues found in the first phase of Smart Contract Audit.

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.

About eNebula Solutions

We believe that people have a fundamental need to security and that the use of secure solutions enables every person to more freely use the Internet and every other connected technology. We aim to provide security consulting service to help others make their solutions more resistant to unauthorized access to data & inadvertent manipulation of the system. We support teams from the design phase through the production to launch and surely after.

The eNebula Solutions team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities & specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code & networks and build custom tools as necessary.

Although we are a small team, we surely believe that we can have a momentous impact on the world by being translucent and open about the work we do.

For more information about our security consulting,
please mail us at – contact@enebula.in