Rubixi Contract

SMART CONTRACT SECURITY AUDIT

ANALAYSIS TYPE (STANDARD)

Blockstars Technology

9th August, 2022

Table of Contents

Declaration

This document is Blockstars' smart contract security audit report for Rubixi contract, and having confidential information. It has the analysis results of smart contract programs which contains vulnerabilities and malicious code which could be used to malform the project. Until the issues are resolved or mitigated, this report is not public.

1. Introduction

This audit report contains confidential information of audit summary of Rubixi smart contract. It analyses security vulnerabilities, smart contract best practices, and possible attacks, using popular automated tests and manual audits. We outlined our systematic approach to evaluate potential security issues in Rubixi.sol and provide audit summary with remedies for mitigating the vulnerability findings.

2. Project Context

This section explains the client's project in detail with scope.

Item	Description
Issuer	Blockstars Technology
Website	https://blockstars.com.au/
Source	Smart contract programs
Language	Solidity
Blockchain	Ethereum
Git Repository	N/A
Audit type	Standard
Analysis Methods	Static, Dynamic analysis, manual
Audit Team	Pura, Faizin, and Marcus
Approved By	Pura
Timeline	From: 08/08/2022 To: 10/08/2022
Change logs	V 0.1

3. Audit Scope

Scope of this project is to identify smart contract vulnerabilities to improve the coding practice that are implemented in the Rubixi contract.

Audit Method: Standard

Standard method covers following audits:

- Automated testing using analysis tools which includes static and dynamic
- Manual audit with code review.

Repository	https://etherscan.io/address/0xe82719202e5965Cf5D9B6673B7503a3b92DE20be#code
Commit Id	NA
Branch	NA
Technical Documentation	NA
Contract	Rubixi.sol
Contract Address	0xe82719202e5965Cf5D9B6673B7503a3b92DE20be

4. Severity Definitions

Severity	Value	Description
Critical	0 - 1.9	Critical vulnerabilities are easily exploited by attackers and they lead to potential assets loss (Example: Tokens, Cryptocurrency)
High	2 - 3.9	High level vulnerabilities are difficult to exploit, however they also have significant impact on smart contract execution due to lack of secured access control. (Example: Public access to crucial functions and data)
Medium	4 - 5.9	Medium vulnerabilities do not lead to loss of assets or data, but it is important to fix those issues.
Moderate	6 - 7.9	Moderate vulnerabilities lead to potential risks of errors when external programs call the contract. Otherwise, contract works as intended.
Low	8 - 8.9	Low level vulnerabilities are related to out-dated, un-used code snippets, and, they don't have significant impact on contract execution.
Informational	9 - 10	It requires best practices, code standards and documentary code. Contract is not vulnerable.

5. Audit Summary

1. Documentation Quality: (0-10)
   N/A (*Documentation Quality Checklist*)

   It covers the quality of the business documentations provided and how it matches with code implementations (*Appendix 9.2*).

2. Code Quality: (0-10)
   6/10

   This includes the use of best practices, coding standards, coding readability, and proper comments in the code (*Appendix 9.3*).

3. Architecture Quality: (0-10)
   8/10

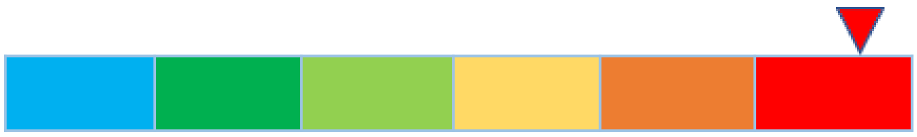
   The process of execution of the contracts in detail such as development environment, and compilation, deployment, and execution detail (*Appendix 9.4*).

4. Security Score: (0-10)
   0/10

   This summarizes the security audit results from automated, manual audits (*Analysis Statistics*).

Summary Score

According to the audit result, we summarize that Rubixi contract is "**Vulnerable to hacks**".



9.5. Summary Score Calculations

Standard: The vulnerabilities that are not identified from automated analysis tools, the audit team have done manual code review for each smart contract code.

The audit analysis identified the following result for each smart contracts:

- 5 critical / 3 high / 2 low level vulnerabilities were found in Rubixi.sol

6. Analysis Statistics

6.1. Programming Issues

#	Description	Type	Severity	Location	Status
1	Unchecked return value from low-level external calls	SWC-104	Medium	L: 67, 78, 88, 96	Failed
2	Functional visibility is not set (prior to solidity 0.5.0)	SWC-100	Low	L: 16, 32, 75, 82, 92, 101, 105, 111, 118, 123, 128, 133, 137, 141, 145, 149	N/A
3	Floating pragma is set	SWC-103	Low	L: 2	Failed
	Error Handling and logging are implemented	Custom	Medium		N/A
4	Spot prices should not be used as a source for price oracles	Custom	Medium		Passed
5	State variable should not be used without being initialized	Custom	Medium		Passed
6	Is inheritance used properly	SWC-125	High		N/A

7	External components used insecurely	Custom	High		N/A
8	Functions that loop over unbounded data structures	Custom	Critical (1)	L: 65-71	Failed
9	Msg.value should not be used in a loop	Custom	Critical		Passed

6.2. Code Specifications and Best practices

#	Description	Type	Severity	Location	Status
1	Use of the "constant" state mutability modifier is deprecated.	SWC-111	Low	L: 118, 123, 128, 133, 137, 141, 145, 149, 76, 86, 93	N/A
2	Use of the "throw" state mutability modifier is deprecated.	SWC-111	Low	L: 76, 86, 93, 106, 112	N/A
3	Strict equalities should not render the function to be unusable	Custom	Moderate		Passed
4	Use of best Practices	Custom	Low		Failed
5	Business logic is implemented as per	Custom	High		N/A

	the documents provided				

6.3. Gas optimization

#	Description	Type	Severity	Location	Status
1	Message call with hardcoded gas amount	SWC-134	Medium		Passed
2	Check for gas usage and minimize gas consumption.	Custom	Low		N/A

6.4. Risk to attacks

#	Description	Type	Severity	Location	Status
1	Code contains suicidal instructions	SWC-106	High		Passed
2	Contract is Haltable	Custom	Medium		Passed
3	Adopt checks-effects-interactions patterns for any transactions of value	Custom	Critical (5)	L: 58-62, 69-74, 82-83, 92-93, 100-101	Failed

4	Reduce and remove unnecessary code to reduce attack surface area.	Custom	Low		Passed
5	Timestamps should not be used to execute critical functions.	SWC-116	Medium		Passed
6	Sensitive data in normal form should not be stored on-chain	Custom	Medium		Passed
7	Vulnerable to Integer over-flow and under-flow	SWC-101	High (7)	All write functions	Failed

7. Manual Audit Result

Functions Overview of Rubixi.sol

#	Function	Type	Observation	Status
1	DynamicPyramid()	Constructor	Mismatch constructor and contract name.	Safe
2	onlyOwner()	Modifier	Because of invalid constructor, this modifier is nullified.	Not safe
3	Function()	Fallback function	Passed	Safe
4	Init()	Private	Passed	Safe
5	addPayout()	Private	Missing check-effects-interactions and loops through unbounded data structure	Not safe
6	collectAllFees()	Write	Access onlyOwner. Violates checks-effects-interaction	Not safe
7	collectFeesInEther()	Write	Access onlyOwner. Violates checks-effects-interaction	Not safe
8	collectPercentOfFees()	Write	Access	Not safe

			onlyOwner. Violates checks-effects-interaction	
9	changeOwner()	Write	Access onlyOwner.	Not safe
10	changeMultiplier()	Write	Access onlyOwner. Unsafe arithmetic calculations	Not safe
11	changeFeePercentage()	Write	Access onlyOwner. Unsafe arithmetic calculations	Not safe
12	currentFeePercentage()	Read	Passed	Safe
13	currentPyramidBalanceApproximately()	Read	Passed	Safe
14	nextPayoutWhenPyramidBalanceTotalsApproximately()	Read	Passed	Safe
15	feesSeperateFromBalanceApproximately()	Read	Passed	Safe
16	totalParticipants()	Read	Passed	Safe
17	numberOfParticipantsWaitingForPayout()	Read	Passed	Safe
18	participantDetails()	Read	Passed	Safe

8. Audit Findings in Detail

8.1. Critical

1. **Issue:** No modifiers are used and no constructor implemented

Function name: DynamicPyramid()

```
            //Sets creator
            function DynamicPyramid() {
                  creator = msg.sender;
            }
```

Description:

No modifiers are used in this function. Anyone can call this function and become the creator. Secure way of doing this is via a constructor.

Resolution:

This function should be a constructor, and the name should be equivalent to the contract's name as "Rubixi". Then when the contract is created the creator is set to the msg.sender.

2. **Issue:** State access after external call

Function Name: addPayout()

```
//Function called for valid tx to the contract
      function addPayout(uint _fee) private {
            participants.push(Participant(msg.sender, (msg.value *
            pyramidMultiplier) / 100));

            if (participants.length == 10) pyramidMultiplier = 200;
            else if (participants.length == 25) pyramidMultiplier = 150;

            balance += (msg.value * (100 - _fee)) / 100;
            collectedFees += (msg.value * _fee) / 100;

            while (balance > participants[payoutOrder].payout) {
                  uint payoutToSend = participants[payoutOrder].payout;

                  participants[payoutOrder].etherAddress.send(payoutToSe
                  nd);
```

```
                        balance -= participants[payoutOrder].payout;
                        payoutOrder += 1;
            }
    }
```

Description:

This function violates the check-effects-interactions which makes it vulnerable to re-entrancy attacks. In the while loop, the state variable balance is set after the external call using send(payoutTosend). This leads to a Re-entrancy attack.

Resolution:

Use check-effects-interactions pattern to avoid re-entrancy attack. It is recommended to change the state variable before the external call.

3. **Issue:** State access after external call

Function Name: collectAllFees ()

```
//Fee functions for creator
      function collectAllFees() onlyowner {
            if (collectedFees == 0) throw;

            creator.send(collectedFees);
            collectedFees = 0;
      }
```

Description: onlyOwner modifier is used. However, the way constructor is defined, this modifier has no effect. It sends the collectedFees and after change the state variable collectedFees. It violates checks-effects-interaction. It is vulnerable to re-entrancy attacks.

Resolution: Use check-effects-interactions pattern to avoid re-entrancy attack. It is recommended to change the state variable before the external call.

4. **Issue:** State access after external call

Function Name: collectFeesInEther ()

```
function collectFeesInEther(uint _amt) onlyowner {
              _amt *= 1 ether;
              if (_amt > collectedFees) collectAllFees();

              if (collectedFees == 0) throw;

              creator.send(_amt);
              collectedFees -= _amt;
       }
```

Description: onlyOwner modifier is used. However, the way constructor is defined, this modifier has no effect. This function sends the _amt first and modifies the state variable collectedFees after the send execution. It violates checks-effects-interaction. Vulnerable to re-entrancy attacks.

Resolution: Use check-effects-interactions pattern to avoid re-entrancy attack. It is recommended to change the state variable before the external call.

5. **Issue:** State access after external call

Function Name: collectPercentOfFees ()

```
function collectPercentOfFees(uint _pcent) onlyowner {
              if (collectedFees == 0 || _pcent > 100) throw;

              uint feesToCollect = collectedFees / 100 * _pcent;
              creator.send(feesToCollect);
              collectedFees -= feesToCollect;
       }
```

Description: onlyOwner modifier is used. However, the way constructor is defined, this modifier has no effect. This function sends the feesToCollect before changing the state variable collectedFees. It violates checks-effects-interaction. Vulnerable to re-entrancy attacks.

Resolution: Use check-effects-interactions pattern to avoid re-entrancy attack. It is recommended to change the state variable before the external call.

8.2. High

1. **Issue**: Modifier has no effect

Function Name: chanegOwner (), changeMultiplier(), changeFeePercentage(), collectAllFees(), collectFeesInEther(), collectPercentOfFees()

```
        function changeOwner(address _owner) onlyowner {
                        creator = _owner;
                }
```

Description: onlyOwner modifier is used in all of the mentioned functions, as above in the changeOwner function. However, the way the constructor is defined, this modifier has no effect. Anyone can call the “DynamicPyramic()” function and get ownership of the contract and able to execute these functions.

Resolution: Modify the constructor name to the contract name “Rubixi”.

2. **Issue:** Integer overflow and underflow
Function Name: addPayout()

Description:
Payout is calculated by multiplying msg.value with pyramidMultiplier and divided by 100. Calculations are vulnerable to overflows and underflows attack. List of participants is dynamic and can be of any limit. If the length of the participant list is huge, the looping execution will run out of gas and cause a DOS attack. Looping check is based on arithmetic calculation. Since safeMath is not used, it is vulnerable to DOS due to underflow/overflow attacks.

Resolution:
Use of SafeMath library from Openzeppelin to avoid arithmetic issues.

3. **Issue**: Integer overflow and underflow

Function Name: collectPercentOfFees ()

Description: Arithmetic calculations are not designed to handle underflows and overflows. It is vulnerable to integer overflow and underflow attacks.

Resolution: Use of SafeMath library from Openzeppelin to avoid arithmetic issues.

9. Conclusion

We received the Rubixi smart contract code from Blockstars Technology to supply a standard security audit. We used automated audit methods using smart contract security analysis tools and manual code review by our professional smart contract auditing team. The report constraints no statements or warranties on the identified issue and vulnerabilities. Do not consider this report as the final statement for smart contract security.

We also recommend proceeding with independent audits and public bug bounty program to ensure the security of the smart contracts.

We identified 5 critical, 3 high, and 2 low levels of smart contract vulnerabilities in the Rubixi contract.

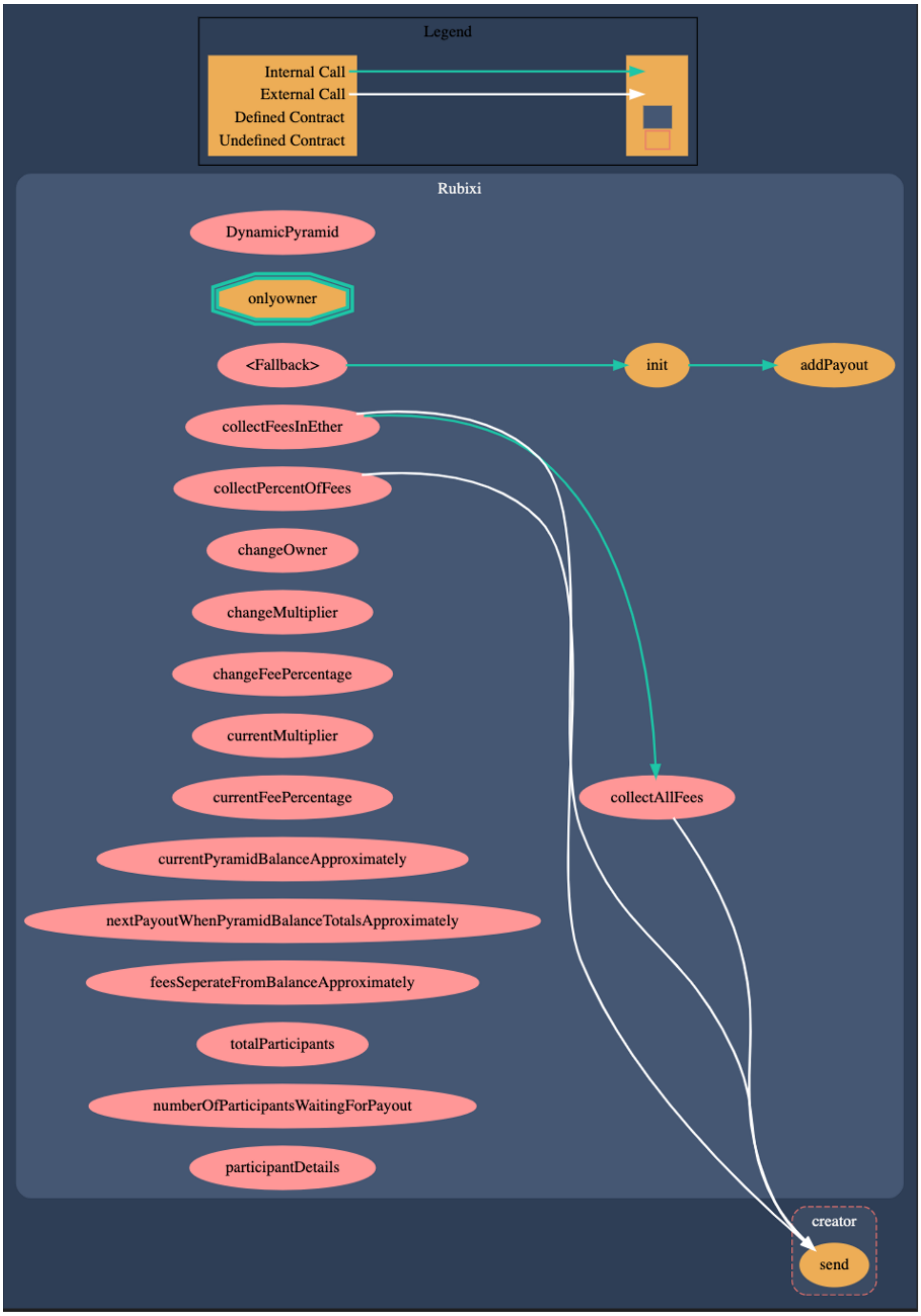
It is strongly recommended to remediate the identified issues in order to avoid.

The current security state of the Rubixi smart contract, based on "Standard" audit scope is "Vulnerable to Hacks". The consultant cannot guarantee the explicit security of the audited smart contracts.

Approved By	Name	Signature
Team Leader	Pura	
Development Head:	Nilanga	

10. Appendix

10.1: Rubixi.sol flow



10.2 Documentation Quality Checklist

#	Documentation Quality	Status
1	UML diagram containing all use cases	N/A
2	System flow chart containing functionalities	N/A
3	System Requirements and Specifications (SRS)	N/A
4	Business logic documentation	N/A

Score = N/A

10.3 Code Quality Checklist

#	Code Quality	Status
1	Code readability	Passed
2	Use of comments with function explanations (What logic, input parameters, expected outputs, accessibilities)	Failed
3	Error handling and logging (Use assert(), require(), revert() properly	N/A
4	Use modifiers only for checks	Failed
5	Beware rounding with integer division	Failed
6	Be aware of the tradeoffs between abstract contracts and interfaces	N/A
7	Keep fallback functions simple	Passed
8	Check data length in fallback functions	N/A
9	Explicitly mark payable functions and state variables	Passed
10	Explicitly mark visibility in functions and state variables	Passed
11	Lock pragmas to specific compiler version	Passed
12	Use events to monitor contract activity	N/A
13	Be aware that 'Built-ins' can be shadowed	N/A
14	Avoid using tx.origin	N/A
15	Avoid using block Timestamp manipulation	Passed
16	Use interface type instead of the address for type safety	N/A

Score = (5/9)*10 = **5.6**

10.4 Architecture Quality Checklist

#	Architecture Quality	Status
1	Check if the contract is upgradable	Passed
2	If upgradable, is proper access control used?	N/A
3	Use of latest compiler version	Passed
4	Avoid nightly build compiler version	Passed
5	Check if contract is haltable	Passed
6	If contract is haltable, make sure the accessibility of other emergency functions is available	Passed
7	Use of proper constructor	Failed
8	Proper access control used	Failed

Score = (5/7)*10 = 7

10.5. Summary Score Calculations

#	Score Type	Weight	Score
1	Documentation Quality	2%	N/A
2	Code Quality	8%	6/10
3	Architecture Quality	15%	7/10
4	Smart contract security	75%	0/10 (All write functions are vulnerable)
	Summary Calculation		(0.08*6/10) + (0.15*7/10) + (0.75*0/10)
			Final Score: 0.048 + 0.105 + 0 = 0.153 = 1.5/10