RD AUDITORS

BetSwirl Referral Smart Contract, Code Review and Security Analysis Report

Customer: BetSwirl
Prepared on: 12th March 2022
Platform: Polygon
Language: Solidity

rdauditors.com

Table of Contents

Disclaimer

This document may contain confidential information about its systems and intellectual property of the customer as well as information about potential vulnerabilities and methods of their exploitation.

The report containing confidential information can be used internally by the customer or it can be disclosed publicly after all vulnerabilities are fixed - upon the decision of the customer.

Document

Name	Smart Contract Code Review and Security Analysis Report of BetSwirl
Platform	Polygon/ Solidity
File 1	AccessControl.sol
MD5 hash	3FF9A51C75A42D39ABBEDEBAB938C18E
SHA256 hash	6F7CF91298F1BE78CFBFD1AAC87EEE6C4B0FB0AA04AD5D1E36F7A6DFC04B4F0E
File 2	Address.sol
MD5 hash	B89961E443500E3FD00BD2581776DBA1
SHA256 hash	D547CAB49A97D7F8FD633DB312B8A074EF816DD4544AF72E4208382E76391647
File 3	Context.sol
MD5 hash	C4B296FB9A98A645CA52CC72C3FBAE06
SHA256 hash	6DE5302543723D32C8EAF17BECC4525936E16D9C4551455C93D306B9B72C0799
File 4	ERC165.sol

MD5 hash	43815742616B32123FE16F92F896BD06
SHA256 hash	D57A87486B2A47DA36C0E094AA079D83BD6660EF2D17DAA41647B229156461C6
File 5	IAccessControl.sol
MD5 hash	DD6853D2027EA64F0B7985DDA5EEA6C6
SHA256 hash	84492B810C2ACCC1FD29BD9BAA00D1C5FCC5A2D8C2023AA81A88EBD9CC9D7621
File 6	IERC20.sol
MD5 hash	B440766B90EF49170F1315DB5F6A0C10
SHA256 hash	14B8D3DCC502B962DED23D820CCF338582B9FF4FDC1C86AAF80A9F804AB0012B
File 7	IERC165.sol
MD5 hash	0FB8A98C0DCBB3F0F693ED5E24E0F32F
SHA256 hash	92762629F91532D937E795CEEE7391D5E4E9DB0CA8EBA233DA3DD1E95CE9D792
File 8	Pausable.sol
MD5 hash	9AB2EFCAED3791A1C57B4B5290B1BC16

SHA256 hash	A96658EC5389FDFCAFF751A72BDACA6D34620DE99EC585D06D37440B0068FB08
File 9	Referral.sol
MD5 hash	C0D98A0EFD080A3B8EF5C1C2A8EAF7AC
SHA256 hash	F0E368374FAA3A4E5FB6F7FC481B94F6433D106F03371C7EE913D2E2CFDF3E16
File 10	SafeERC20.sol
MD5 hash	898F1F7533AE391DE23E193E9ABAE754
SHA256 hash	892DAF0652C27177692C0D79CCFD0E1F75EA174838748410C46F4C69E122219D
File 11	Strings.sol
MD5 hash	F90E6606083FA5398E2C8377DB1A4536
SHA256 hash	5258E2F47CA2C91476EEF42ADEA79C9D72448374D58B6DB877117188BBE0C9C55
Date	12/3/2022

Introduction

RD Auditors (Consultant) were contracted by BetSwirl (Customer) to conduct a Smart Contracts Code Review and Security Analysis. This report represents the findings of the security assessment of the customer`s smart contracts and its code review conducted between 3rd March 2022 - 12th March 2022.

This contract consists of eleven files.

Project Scope

The scope of the project is a smart contract. We have scanned this smart contract for commonly known and more specific vulnerabilities, below are those considered (the full list includes but is not limited to):

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Byte array vulnerabilities
- Style guide violation
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Unchecked external call - Unchecked math
- Unsafe type inference
- Implicit visibility level

Executive Summary

According to the assessment, the customer's solidity smart contract is **Well-Secured.**

Automated checks are with smartDec, Mythril, Slither and remix IDE. All issues were performed by our team, which included the analysis of code functionality, the manual audit found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the audit overview section. The general overview is presented in the AS-IS section and all issues found are located in the audit overview section.

We found the following;

Total Issues	0
Critical	0
High	0
Medium	0
Low	0
Very Low	0

Code Quality

Please note that within this report IBEP20, Context, Ownable are taken from the popular OpenZeppelin library.

The libraries within this smart contract are part of a logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned to a specific address and its properties/methods can be reused many times by other contracts.

The BetSwirl team has provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.

Documentation

We were given the BetSwirl code as a link:

https://polygonscan.com/address/0xd0B6D4503207588B677F79876cf12724bAFF70f8#code

The hash of that file is mentioned in the table. As mentioned above, It's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic.

Comments are very helpful in understanding the overall architecture of the protocol. It also provides a clear overview of the system components, including helpful details, like the lifetime of the background script.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure. Those were based on well known industry standard open source projects and even core code blocks that are written well and systematically.

AS-IS Overview

BetSwirl

File And Function Level Report

File: Pausable.sol

Contract: Pausable

Observation: Passed

Test Report: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	paused	read	Passed	All Passed	No Issue	Passed
2	_pause	internal				
3	_unpause	internal				

Contract: Referral

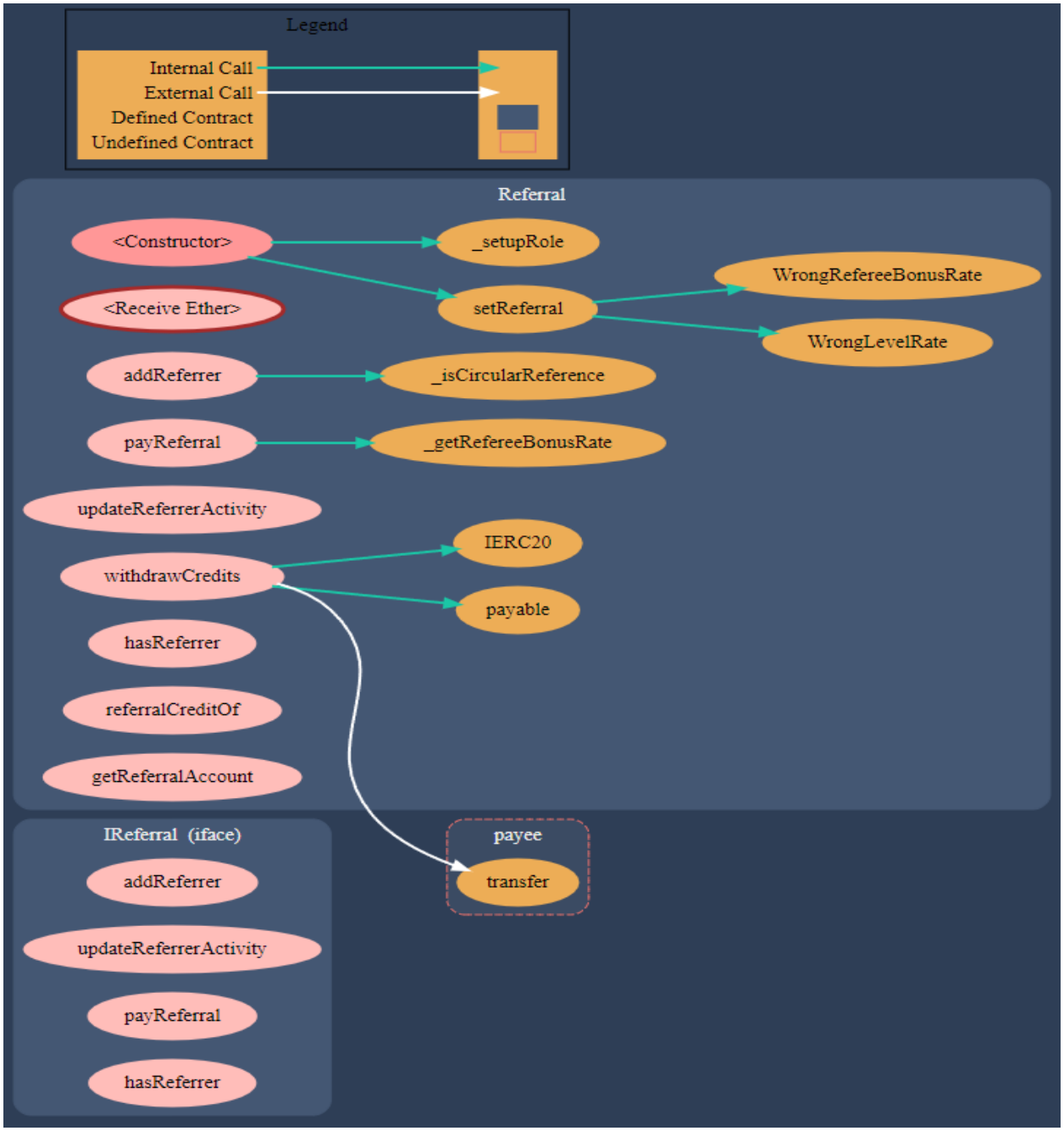
Inherit: AccessControl, Pausable

Observation: Passed

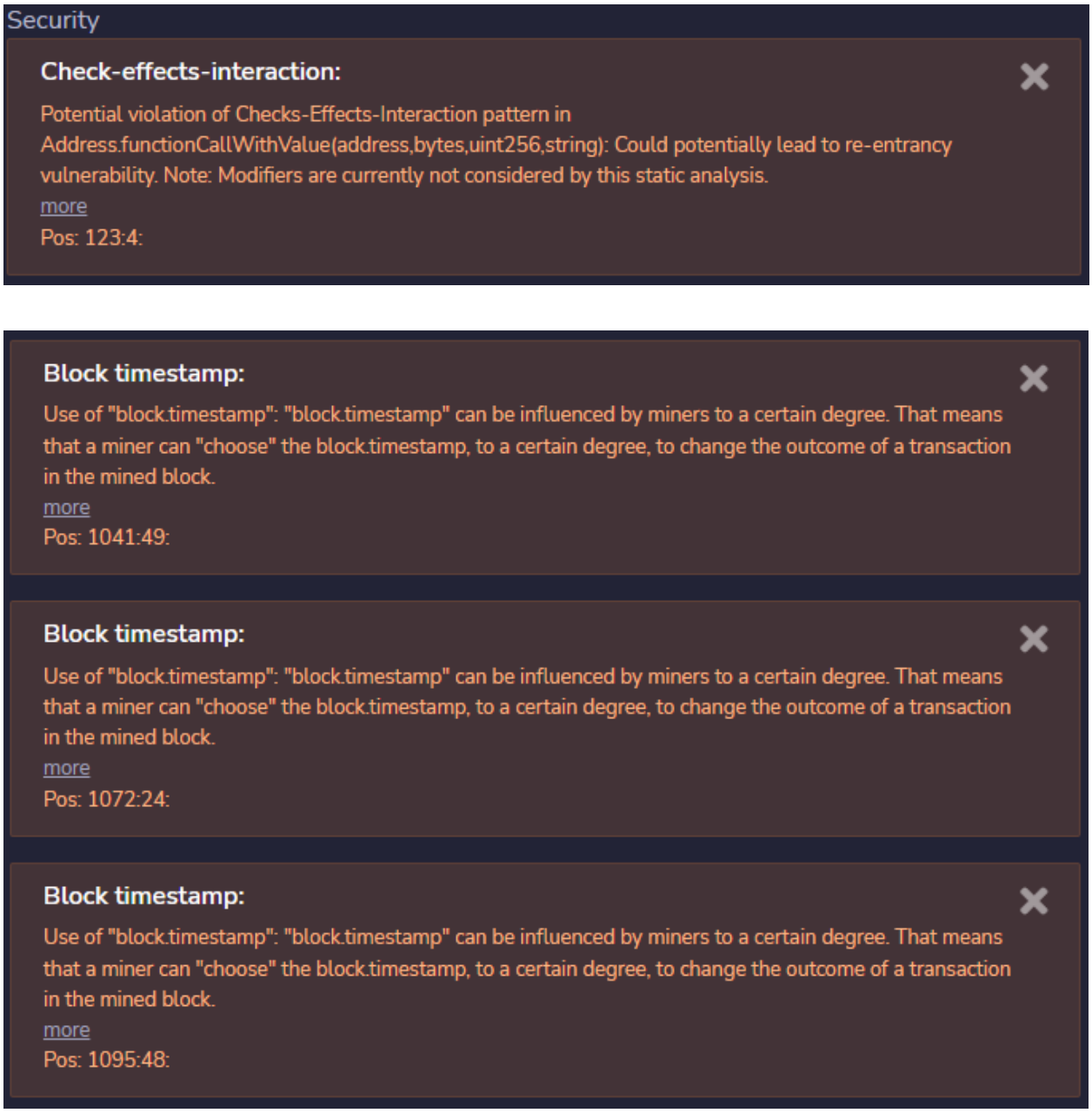
Test Report: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	_getRefereeBonusRate	read	Passed	All Passed	No Issue	Passed
2	_isCircularReference	read	Passed	All Passed	No Issue	Passed
3	setReferral	write	Passed	All Passed	No Issue	Passed
4	addReferrer	external	Passed	All Passed	No Issue	Passed
5	payReferral	write	Passed	All Passed	No Issue	Passed
6	updateReferrerActivity	external	Passed	All Passed	No Issue	Passed
7	withdrawCredits	external	Passed	All Passed	No Issue	Passed
8	hasReferrer	read	Passed	All Passed	No Issue	Passed
9	referralCreditOf	read	Passed	All Passed	No Issue	Passed
10	getReferralAccount	read	Passed	All Passed	No Issue	Passed

Code Flow Diagram - BetSwirl



Solidity Static Analysis



Gas & Economy

Gas costs:

Gas requirement of function Referral.getRoleAdmin is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 684:4:

Gas costs:

Gas requirement of function Referral.updateReferrerActivity is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 1092:4:

Gas costs:

Gas requirement of function Referral.withdrawCredits is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 1103:4:

Gas costs:

Gas requirement of function Referral.referralCreditOf is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 1133:4:

For loop over dynamic array:

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Additionally, using unbounded loops incurs in a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.
more
Pos: 1105:8:

Miscellaneous

Constant/View/Pure functions:

Address.functionStaticCall(address,bytes) : Is constant but potentially should not be. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 142:4:

Similar variable names:

Referral._isCircularReference(address,address) : Variables have very similar names "referrer" and "referee". Note: Modifiers are currently not considered by this static analysis.
Pos: 948:25:

Similar variable names:

Referral._isCircularReference(address,address) : Variables have very similar names "referrer" and "referee". Note: Modifiers are currently not considered by this static analysis.
Pos: 955:26:

Similar variable names:

Referral.payReferral(address,address,uint256) : Variables have very similar names "_credits" and "credit". Note: Modifiers are currently not considered by this static analysis.
Pos: 1074:20:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.

more

Pos: 589:8:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.

more

Pos: 730:8:

Data truncated:

Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.

Pos: 1074:37:

Data truncated:

Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.

Pos: 1074:39:

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to lost tokens etc.
High	High level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g. public access to crucial functions.
Medium	Medium level vulnerabilities are important to fix; however, they cannot lead to lost tokens.
Low	Low level vulnerabilities are most related to outdated, unused etc. These code snippets cannot have a significant impact on execution.
Lowest Code Style/ Best Practice	Lowest level vulnerabilities, code style violations and information statements cannot affect smart contract execution and can be ignored.

Audit Findings

Critical

No critical severity vulnerabilities were found.

High:

No high severity vulnerabilities were found.

Medium:

No medium severity vulnerabilities were found.

Low:

No low severity vulnerabilities were found.

Very Low:

No very low severity vulnerabilities were found.

Conclusion

We were given a contract file and have used all possible tests based on the given object. The contract is written systematically, so it is ready to go for production.

We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

The security state of the reviewed contract is now “well-secured”.

Note For Contract Users

There are several owner only functions. Those can be called by the owner's wallet only. So, if the owner's wallet is compromised, then it carries the risk of the contract becoming vulnerable.

Owner has full control over the smart contract. Thus, technical auditing does not guarantee the project's ethical side.

Please do your due diligence before investing. Our audit report is never an investment advice.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are 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.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included 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, skim open issue tickets, and generally 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 generally 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 analyse the feasibility of an attack in a live system.

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 scrutinised 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.

Disclaimers

RD Auditors Disclaimer

The smart contracts given for audit have been analysed in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Because the total number of test cases are unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

Email: info@rdauditors.com

Website: www.rdauditors.com