Blaize.Security

March 29th, 2023 / V. 1.0

VERITTY

SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

Veritty contracts' source code was provided via Etherscan.

SCORE **9.9** /10

The scope of the project includes the **Veritty** set of contracts**:**

Ticket.sol
RaffleImpl.sol

Initial source code:

https://goerli.etherscan.io/address/0x0CE0E5676b7c6c26F6A9923cC2C054eB3FDBEcc2#code

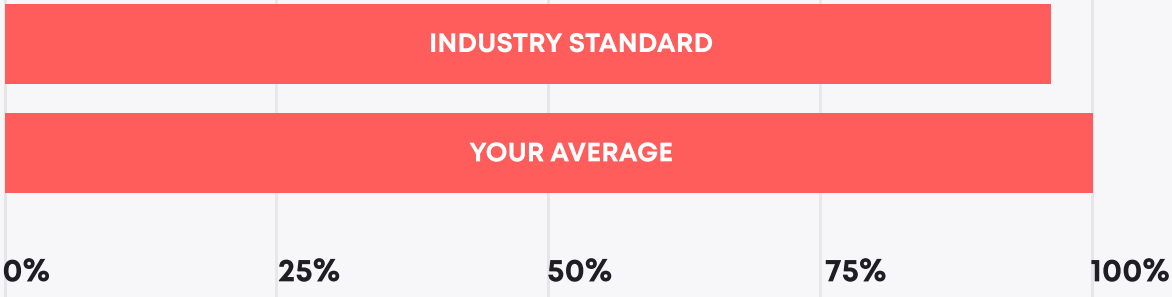
Final source code:

https://sepolia.etherscan.io/address/0x85f9A2BA28f13086285381023A6f759AEb65FFD9#code

TECHNICAL SUMMARY

During the audit, we examined the security of smart contracts for the Veritty protocol. Our task was to find and describe any security issues in the smart contracts of the platform. This report presents the findings of the security audit of the **Veritty** smart contracts conducted from **February 24th, 2023 - March 29th, 2023.**

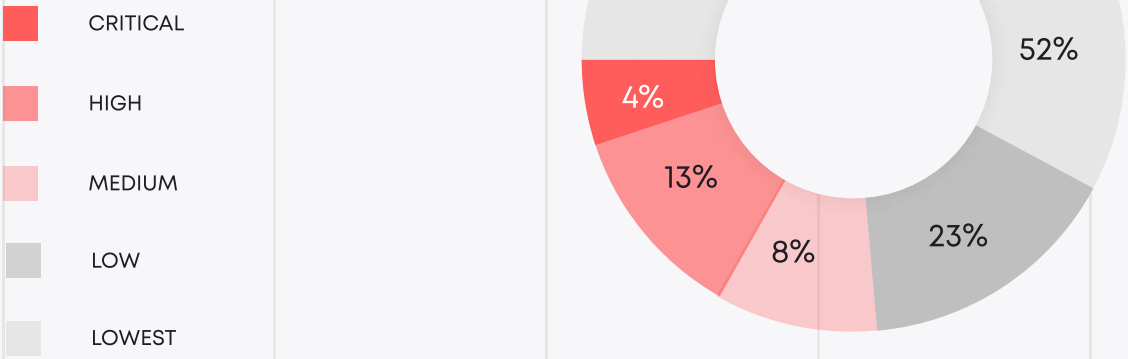
Testable code



The code is 100% testable, which is above the industry standard of 95%.

The audit scope includes the unit test coverage based on the smart contracts code, documentation, and requirements presented by the Veritty team. The coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies. However, to ensure the contract's security, the Blaize.Security team suggests the Veritty team launch a bug bounty program to encourage further active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. Of the 23 problems found, 22 were fixed or verified by the Veritty team.

	FOUND	FIXED/VERIFIED
Critical	1	1
High	3	3
Medium	2	2
Low	5	5
Lowest	12	11

SEVERITY DEFINITION

Critical

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

High

The system contains a couple of serious issues, which lead to unreliable work of the system and might cause a huge data or financial leak. Requires immediate fixes and a further check.

Medium

The system contains issues that may lead to medium financial loss or users' private information leaks. Requires immediate fixes and a further check.

Low

The system contains several risks ranked as relatively small with a low impact on the users' information and financial security. Requires fixes.

Lowest

The system does not contain critical security issues, but following best practices remains relevant.

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

We have scanned this smart contract for commonly known and more specific vulnerabilities:

- Unsafe type inference;
- Timestamp Dependence;
- Reentrancy;
- Implicit visibility level;
- Gas Limit and Loops;
- Transaction-Ordering Dependence;
- Unchecked external call - Unchecked math;
- DoS with Block Gas Limit;
- DoS with (unexpected) Throw;
- Byte array vulnerabilities;
- Malicious libraries;
- Style guide violation;
- ERC-20 API violation;
- Uninitialized state/storage/ local variables;
- Compile version not fixed.

Procedure

We checked the contract for the following parameters:

- Whether the contract is secure;
- Whether the contract corresponds to the documentation;
- Whether the contract meets the best practices in the efficient use of gas, and code readability.

Automated analysis:

Contract scanning by several publicly available automated analysis tools such as Mythril, Solhint, Slither, and Smartdec. Manual verification of all the issues found with tools.

Manual audit:

Manual analysis of smart contracts for security vulnerabilities. We checked smart contract logic and compared it with the one described in the documentation.

EXECUTIVE SUMMARY

We audited a set of smart contracts provided by the Veritty team. The protocol represents a raffle smart contract and NFT tickets, allowing users to purchase tickets with ETH for a chance to win multiple prizes in USD tokens. Our goal was to analyze the security of the smart contracts, validate the business logic and the flow of funds, audit smart contracts against common vulnerabilities and the Blaize.Security internal security checklist, and review gas consumption optimization.

During the manual audit, we discovered several issues with different threat severity. One issue was that randomness was provided with information that miners could predict or manipulate. We suggested several approaches to mitigate this vulnerability. Consequently, the Veritty team integrated the Chainlink VRF Oracle to provide truly random numbers for the protocol.

Other issues we found were an unsafe transfer of ETH, verification of price calculations, inability to withdraw small fractions of ETH, and absence of validations and logic verification. We also proposed several approaches to optimize the user sorting algorithm. Again, the Veritty team has successfully fixed or verified all the issues.

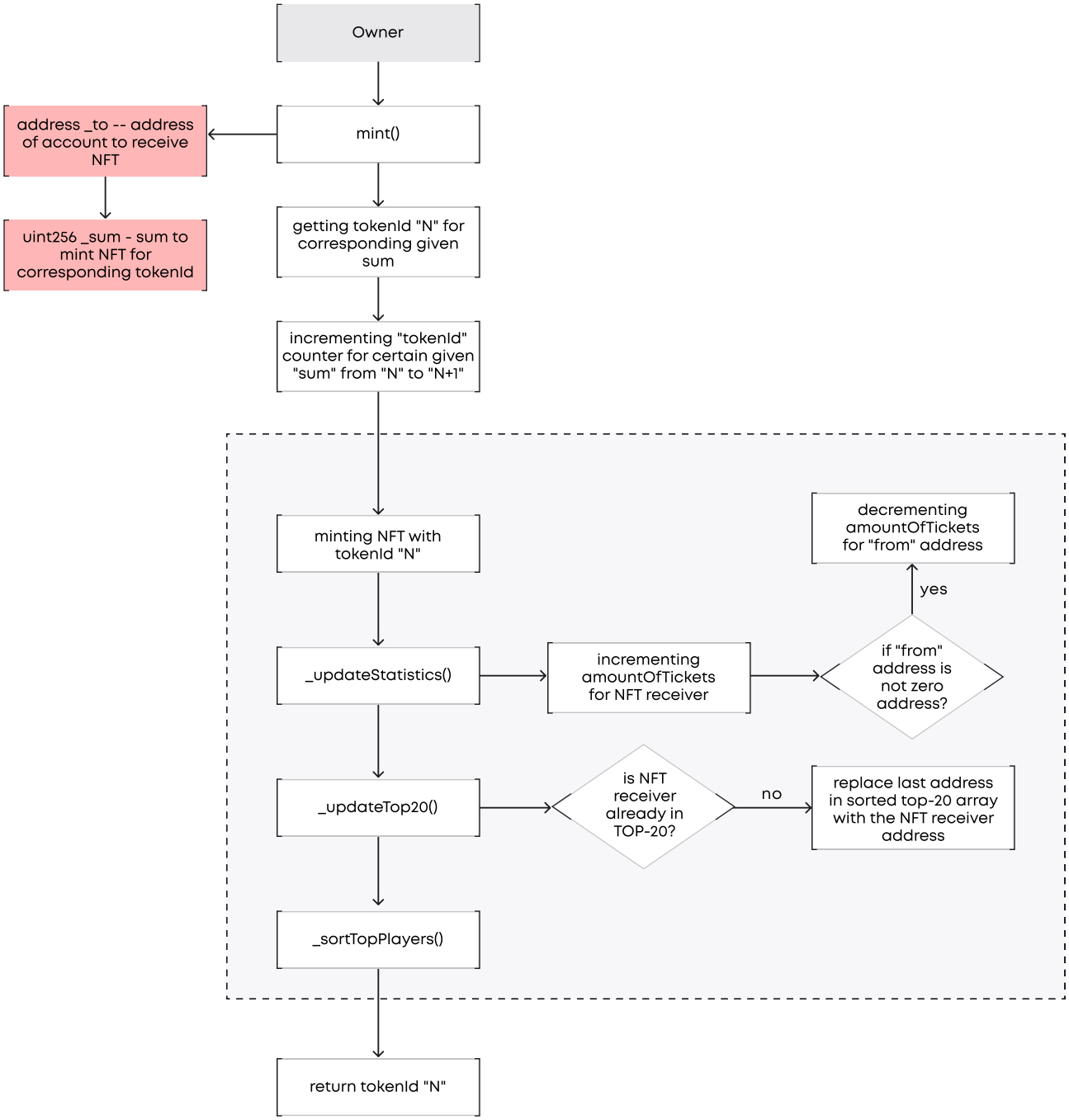
Our team also prepared a set of unit tests and additional testing scenarios to validate smart contracts security, including checks against the lately added referral fee functionality. After the Veritty team has fixed all the issues, smart contracts have passed all the security tests.

The overall security of smart contracts is high enough. The Veritty team has fixed all the issues, implemented random numbers with Chainlink VRF, and applied gas optimization suggestions. Furthermore, the smart contracts are well-written and tested, though the code quality rating is decreased due to the absence of documentation.

	RATING
Security	10
Gas usage and logic optimization	10
Code quality	9.8
Test coverage	10
Total	9.9

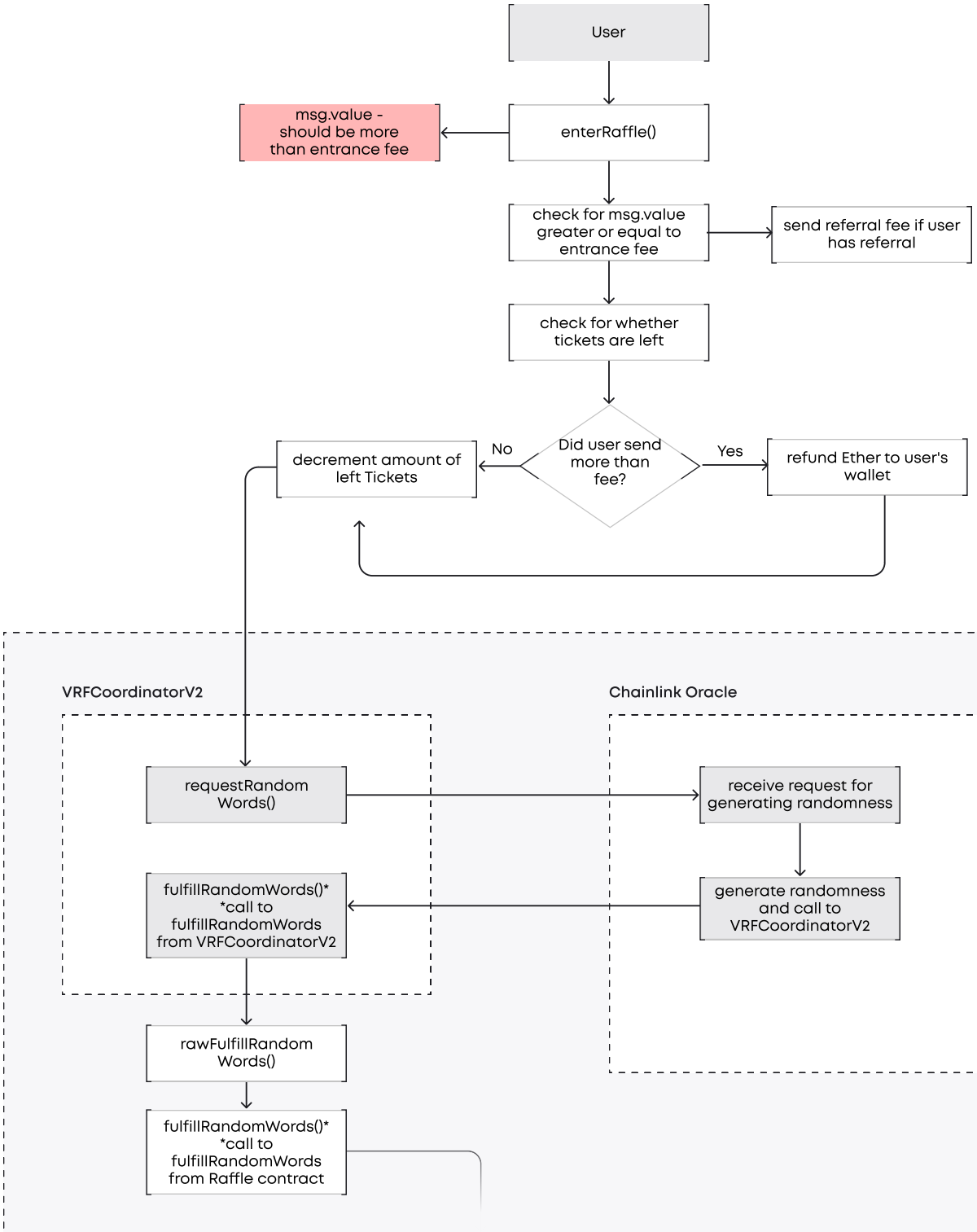
TICKET.SOL

Minting flow



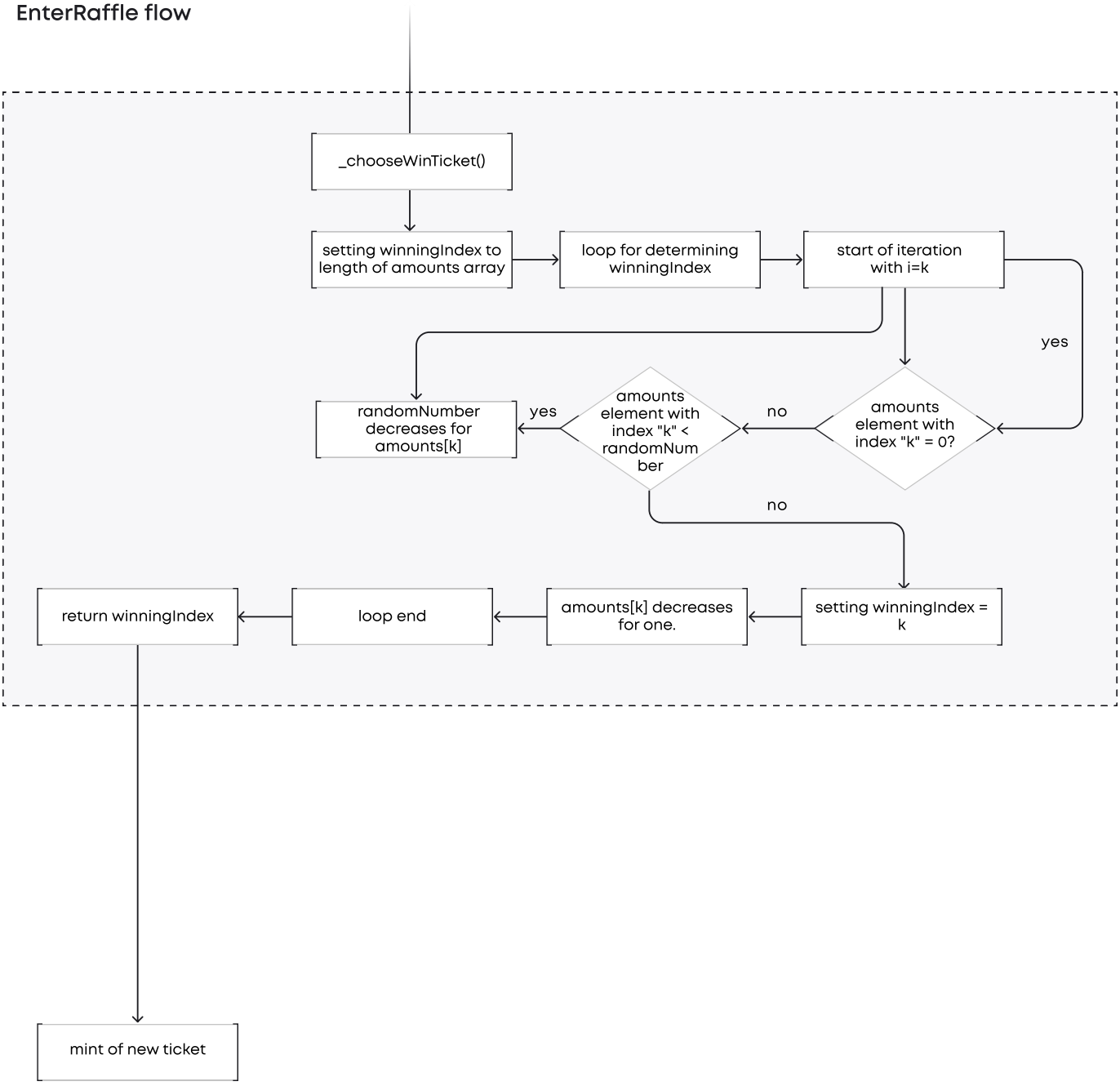
RAFFLEIMPL.SOL

EnterRaffle flow



RAFFLEIMPL.SOL

EnterRaffle flow



COMPLETE ANALYSIS

CRITICAL-1	✔ Resolved

Miners are able to manipulate the result of a random number.

RaffleImpl.sol: enterRaffle().
In order to determine winningIndex, a pseudo-random number is generated based on the hash of the previous block msg.sender and a nonce. Miners could manipulate the transaction and some blocks to control the outcomes of the id to mint.

This article describes the issue.

Recommendation:

Consider using off-chain oracles for generating random numbers.

Post-audit:
Chainlink VRF was integrated to provide random numbers.

HIGH-1	✔ Verified

Users can pay less than the entrance fee.

RaffleImpl.sol: enterRaffle().
Users are able to provide their other addresses as referrals to get a refund and pay fewer entrance fees. With current logic, payment to the referrals is performed with protocol funds, and no extra referral fee is charged.

Recommendation:
Consider adding a referral validation either on Frontend or directly in contracts, for example, through a whitelist of accounts that can be used as referrals.

Post-audit:
According to the Veritty team, such functionality corresponds to the logic of the protocol.

HIGH-2	✔ Resolved

ETH might not be calculated correctly from USDT.

RaffleImpl: pay()
According to the documentation, the 'amountInUsdt' parameter should be passed without decimals (e.g., 1, 2, 5). The value 'answer' returned by priceFeed represents the ETH price in USD with 8 decimals (e.g., 163236000ETH), but it is not correctly calculated from USD.
If we want to withdraw $5 worth of ETH and pass 5 as 'amountInUsdt', the result of multiplying it by 'answer' will be incorrect. To correctly withdraw ETH, the following expression should be used: amountInUSDT * 10**(8 + 18) / answer.
This will return the correct value in ETH units corresponding to the provided USD value.

Recommendation:

Change the calculation of the ETH amount OR consider simplifying function by providing the value ETH, which should be withdrawn.

Post-audit:
According to the team, the contract will interact with Chainlink Pricefeed Oracle for USDT/ETH. Price is calculated correctly in the current implementation with this particular oracle.

HIGH-3	✔ Resolved

Obsolete method for transferring ETH.

RaffleImpl.sol: enterRaffle()
Function utilizes .send() method for ETH transfer. The enterRaffle() function referral shares ETH with the referrer, which may be set to the multisig account. In this case, transfer() may revert since it forwards not enough gas. Therefore, funds may be stuck on the contract.
Since .transfer() and .send() methods became obsolete after the Istanbul Ethereum update, it is recommended to use .call() for funds transfer. This issue is marked as high because the receiver address is provided by users, and a smart contract account might be potentially used.

Recommendation:

Use .call() for ETH transfer.

MEDIUM-1	✔ Resolved

ETH withdrawal might be blocked.

RaffleImpl.sol: pay().
The owner of the contract is able to withdraw accumulated ETH through the pay() method. While calling this method, the owner specifies an amount of ETH to withdraw, equivalent to a particular USDT amount. The ETH amount is determined by Chainlink's PriceFeed, which is set during contract deployment. Thus, the withdrawal of ETH might be blocked if PriceFeed is hacked or damaged in some way. However, the contract has no setter method on the PriceFeed address. The issue is classified as Medium because withdrawal relies on an external protocol.

Recommendation:

Add a setter on the PriceFeed address.

Post-audit:
The setter was added.

MEDIUM-2	✔ Acknowledged

USD amount with fraction can't be used.

RaffleImpl.sol: pay().
According to the provided documentation, the parameter 'amountInUsdt' should be passed as a value without decimals (e.g., 1, 2, 5). Thus, if the ETH amount left on the contract is worth less than $1, it can't be withdrawn, as only values without decimals can be passed. Thus a small amount of ETH will be stuck on the contract.

Recommendation:

Allow to provide USDT value with decimals to pass fractional part (e.g., $1 = 1000000 with 6 zeros, since USDT has 6 decimals).

Post-audit:
According to the Veritty team, the loss of USDT <$1 is not relevant.

LOW-1	✔ Resolved

Contracts can be initialized with wrong parameters.

RaffleImpl.sol: constructor().
The _priceFeed and _ticketNft variables are not validated for not being zero-address. Thus, there are no setters for these variables.

Recommendation:

Validate parameters in the initialization of contracts.

LOW-2	✔ Resolved

No Ether refund.

RaffleImpl.sol: enterRaffle().
In the enterRaffle() function, the first operation is to check whether the user sent not less ETH than required by entrance fee, and if there is more ETH than needed, it is not being returned to his account.

Recommendation:

Check the value to be equal to the entrance fee or return funds back to the user account **OR** verify that it is normal behavior that the user can send more than the entrance fee without a refund.

Post-audit:
Refunding was added.

LOW-3	✔ Resolved

Unsafe transfer of Ether.

RaffleImpl.sol: enterRaffle().
When sending an ETH refund to the user, funds are transferred with the .transfer() method, which is considered unsafe for transferring ETH. A smart contract now is taking a hard dependency on gas costs by forwarding a fixed amount of gas: 2300. Thus, we will suggest changing .transfer in favor of .call or .sendValue from OpenZeppelin's 'Address' library.
However, when transferring ETH via .call or .sendValue, you should consider the reentrancy issue and do safety checks before ETH transfer as mentioned in the checks-effects-interactions pattern **OR** by adding a non-reentrant modifier.

Recommendation:

Change method for sending ETH refund.

Post-audit:
The .transfer() was replaced with .call().

LOW-4	✔ Resolved

Missing validation checks in setters.

RaffleImpl.sol, Ticket.sol.
Such variables as priceFeed, keyHash, and subscriptionId are being validated for zero-address in the constructor, but there are no safety checks on any of these parameters on the setters.

Recommendation:

Add validations on the setters.

Post-audit:
According to the Veritty team, validations won't be added not to increase gas consumption.

LOW-5	✔ Resolved

Referrer's reward might be more than the entrance fee.

RaffleImpl.sol: enterRaffle()
In the setter for the referrerSares variable, there is no validation for new referrerSares not being greater than the 'FLOOR' variable. When the referrerSares is set more than FLOOR, the referrer's becomes greater than the Ticket price.

Recommendation:

Validate the referrer's shares to be less than 'FLOOR'.

Post-audit:
Validation for the referrer's share to be less than the 'FLOOR' variable was added.

LOWEST-1	✔ Resolved

Lack of events.

RaffleImpl.sol: multiple methods.
To track the historical changes of essential storage variables, emitting events in setters on every variable change is recommended. Thus the following setter function should emit an event.

1. enterRaffle()
2. pay()
3. finalGame()
4. stopRaffle()
5. openRaffle()

Recommendation:

Emit events in setter functions.

Post-audit: The necessary events were added.

LOWEST-2	✔ Resolved

Gas optimization suggestions.

Ticket.sol: _sortTopPlayers().
After every Ticket token transfer, the _afterTokenTransfer hook updates statistics and sorts the array of top players according to the new token transfer. Choosing the most efficient sorting method in terms of complexity is recommended for the sake of gas usage. Also, the array is almost sorted after every token transfer. The implemented sorting algorithm is **Bubble sort**, which is not a preferable algorithm in terms of complexity for sorting a given type of array (almost sorted).
Bubble sort is arguably the simplest sorting algorithm, and its time complexity of $O(n^2)$ in the average and worst cases and $O(n)$ in the best case. Bubble sort does not benefit at all from how the data is organized when counting the number of element comparisons it makes.

Bubble sort always performs the exact same amount of comparisons for a certain amount of data regardless of how the data is organized. Bubble sort always performs n-1 passes through the data (where n is the amount of elements), and it always performs (n-1)+(n-2)+...+1 comparisons regardless of how the data is organized to begin with. In contrast, **Insertion sort** also has time complexity $O(n^2)$ in the worst-case scenario, but it does benefit if the data is already almost sorted. The best case scenario is when the data is already fully sorted, in which case insertion sort goes through the array once and performs n-1 comparisons, and that's it.

If we apply this to possible scenarios for sorting top players - we have 2 different options for changing the state of the array:

1. Mint of Ticket. In this case, only one change in the array should be made to keep it sorted: Assuming we have an array of addresses {k1, k2, k3, ..., k20}, where amountOfTickets[k1] == amountOfTickets[k2]. Let's suppose a new Ticket was minted to address 'k2'. In this scenario, we need 1 swap in the array to keep it sorted. Bubble sort will perform **n + (n-1) comparisons** in that case. Insertion sort will perform just **n comparisons.**
2. Transfer of Tickets between users from top-20. In this case, a maximum of 2 swaps should be performed in the next possible scenario: Assuming we have an array of addresses {k1, k2, k3, ..., k20}, where amountOfTickets[k1] == amountOfTickets[k2] and amountOfTickets[k1] - amountOfTickets[k3] = 1.

Let's suppose an address 'k3' transferring Ticket to address 'k1'.
In this scenario, we need 2 swaps in the array to keep it sorted. First, swapping k1 and k2, and the second swap is between k1 and k3. Bubble sort will perform **n + (n-1) comparisons** in that case. Insertion sort will perform just **n comparisons.**
This article provides an example of the implementation of the insertion sorting algorithm.

Recommendation: Consider picking a more efficient sorting method for updating top players.

Post-audit: Insertion sorting algorithm is used now.

LOWEST-3	✔ Resolved

Variables' visibility can be changed to immutable.

RaffleImpl.sol: sums, finalGameSums.
Arrays 'sums' and 'finalGameSums' are being initialized in the constructor and couldn't be set again with no setters on it, so marking this variable as immutable is preferable.

Recommendation:
Consider marking the isStable variable as immutable.

LOWEST-4	Unresolved

Lack of documentation.

RaffleImpl.sol, Ticket.sol.
The contract's documentation is barely presented and consists of a few lines of comments. Writing comments in a conventional style to document the code is recommended.

Recommendation:

Add documentation to contracts.

LOWEST-5	✔ Resolved

Fixed Solidity version should be used.

RaffleImpl.sol, Ticket.sol.
For now, all contracts use solidity ^0.8.17 - floating version, which does not correspond to best practices. Thus the compilation may include unstable versions of compilers with major bugs introduced during intermediate releases. The contracts should be deployed with the same compiler version and options with which they have been most tested. Locking the pragma version helps ensure the contract is not accidentally deployed using a different version. Thus, it is recommended to use the last stable version - which is now 0.8.19.

Recommendation:
Use the last stable Solidity version.

LOWEST-6	✔ Resolved

Unused function

RaffleImpl.sol: _mixTop20Players().
Function is internal and is never used within the smart contract. An unused function may indicate unfinished logic.

Recommendation:
Remove or use the function **OR** verify that it is necessary for use contract which will inherit this function.

Post-audit:
The function was removed.

LOWEST-7	✔ Resolved

Unnecessary variables.

1. RaffleImpl.sol: enterRaffle() line 82 variable “openPositionsLeft”. There is no need to create extra local variable to pass value of ticketsLeft to expression in line 84.

2. RaffleImpl.sol: constructor() line 41 amounts = new uint256[] (length)
There is no need to create an empty array inside a variable to fill it with values. Assigning an array directly to a variable without extra steps is possible.

Recommendation:

1. Use value of ticketsLeft directly in expression.
2. Assign array to variable directly without filling variable with empty array.

LOWEST-8	✔ Verified

Exchange rate volatility risk.

Users pay in ETH to participate, but the rewards are paid in USDT. e.g., if the prize pool is 90% and the ETH price drop is 20%, paying the rewards in full would be impossible.

Recommendation:

State rewards in ETH or make sure that the reward currency has been chosen intentionally and you have reserves to cover ETH price volatility.

Post-audit:

According to the Veritty team, such volatility risk is acceptable, and rewards will be distributed corresponding to the actual exchange rate.

LOWEST-9	✔ Resolved

Typo

1. RaffleImpl.sol: constructor()
2. _vffCoordinatorV2 -> _vrfCoordinatorV2

Recommendation:
Fix typo.

LOWEST-10	✔ Resolved

Function marked as virtual but not overridden by any other function.

RaffleImpl.sol: fulfillRandomWords().
The fulfillRandomWords function overrides a function from the VRFConsumerBaseV2 contract but is not itself overridden. However, it is marked as 'virtual'.

Recommendation:
Remove the 'virtual' keyword.

Post-audit: The keyword was removed.

LOWEST-11	✔ Resolved

Unused function parameter.

Ticket.sol: _updateStatistics().
Parameter 'sum' is not used in the function.

Recommendation:
Remove unused function parameter.

Post-audit: The parameter was removed.

LOWEST-12	✔ Resolved

Users are able to set themselves as their referrers.

RaffleImpl.sol: enterRaffle()
In the enterRaffle() function, there is no validation for a given referrer's address not to be equal to the msg.sender address. Thus, users can set themselves as referrers and benefit according to referrer shares.

Recommendation:

When entering Raffle, validate the referrer's address not to be msg.sender address.

Post-audit:
Validation for the referrer's address not to be msg.sender was implemented.

	Ticket.sol RaffleImpl.sol
✔ Re-entrancy	Pass
✔ Access Management Hierarchy	Pass
✔ Arithmetic Over/Under Flows	Pass
✔ Delegatecall Unexpected Ether	Pass
✔ Default Public Visibility	Pass
✔ Hidden Malicious Code	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass
✔ External Contract Referencing	Pass
✔ Short Address/Parameter Attack	Pass
✔ Unchecked CALL Return Values	Pass
✔ Race Conditions/Front Running	Pass
✔ General Denial Of Service (DoS)	Pass
✔ Uninitialized Storage Pointers	Pass
✔ Floating Points and Precision	Pass
✔ Tx.Origin Authentication	Pass
✔ Signatures Replay	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

Raffle

- ✓ enterRaffle when some ticket amount minted compelty (189ms)
- ✓ enterRaffle when some ticket amount minted compelty (177ms)
- ✓ constructor cannot be called with argument mismatch
- ✓ enterRaffle tickets distributed with correct amount in correct quantity
- ✓ user cannot enterRaffle if he paid not enough ether
- ✓ user cannot enterRaffle if raffle is not opened
- ✓ user cannot enterRaffle if all tickets are sold (1255441ms)
- ✓ only owner call pay function
- ✓ cannot pay when no ether present at contract
- ✓ owner should be able to pay the ETH to users (2240ms)

Setters and getters

- ✓ should open raffle
- ✓ only owner can open raffle
- ✓ should close raffle
- ✓ only owner can close raffle
- ✓ setEntranceFee
- ✓ only owner can setEntranceFee
- ✓ getTicketsLeft
- ✓ getTopPlayers (3499ms)
- ✓ finalGame (3464ms)
- ✓ cannot call finalGame if raffle is open (3523ms)
- ✓ getTicketsLeftBySum (38ms)

Ticket

- ✓ supportsInterface
- ✓ win
- ✓ getTopPlayer (3502ms)
- ✓ tokenURI (240ms)
- ✓ only owner can mint nft
- ✓ _updateStatistics during nft transfer (265ms)

- ✓ user should enterRaffle -> oracle's mock should triger fulfilling of random words (111ms)
- ✓ user should get the Ether refund if he sends more value than entranceFee (103ms)
- ✓ user should enterRaffle (145ms)
- ✓ referrer should be paid according to refferal shares when his referal is entering raffle (64ms)
- ✓ should revert when user sets himself/herself as own referrer
- ✓ should revent when deploying Raffle with argument's length mismatch
- ✓ should revent when deploying Raffle with zero address as price feed
- ✓ should revent when deploying Raffle with zero address as Ticket NFT
- ✓ should revent when deploying Raffle with zero address as VRFCoordinatorV2
- ✓ should revent when deploying Raffle with requestConfirmations=0
- ✓ should revent when deploying Raffle with zero bytes32 as keyHash
- ✓ should revent when deploying Raffle with subscriptionId=0
- ✓ user cannot enterRaffle if he paid not enough ether
- ✓ user cannot enterRaffle if raffle is not opened
- \- user cannot enterRaffle if all tickets are sold
- ✓ only owner can call pay function
- ✓ cannot pay when no ether present at contract
- ✓ owner should be able to pay the Ether to users (848ms)

Setters and getters

- ✓ should open raffle
- ✓ only owner can open raffle
- ✓ should close raffle
- ✓ only owner can close raffle
- ✓ should set entrance fee
- ✓ should set new chainlink subscription id
- ✓ only owner can set entrance fee
- ✓ should set new price feed

- ✓ should set new key hash
- ✓ should set baseUri
- ✓ should revert when setting zero address as price feed
- ✓ getTicketsLeft
- ✓ getTopPlayers (1143ms)
- ✓ finalGame (1165ms)
- ✓ cannot call finalGame if raffle is open (1547ms)
- ✓ getTicketsLeftBySum (45ms)

Ticket

- ✓ supportsInterface
- ✓ user should get 'win' status if his sum > 0
- ✓ user should get 'win' status if his sum = 0
- ✓ getTopPlayer (2220ms)
- ✓ should decrease Tickets amount in sender balance on NFT transfer (120ms)
- ✓ should revert when not owner tries to mint NFT
- ✓ should revert when deploying Ticket NFT with arguments' length mismatch (57ms)

Scenarios

- ✓ 2 users scenario: user1 enters Raffle -> user2 enters Raffle -> user1 enters Raffle -> user2 enters Raffle 2 times -> getTopPlayer() (317ms)
- ✓ 3 users scenario: user1 enters Raffle -> user2 enters Raffle -> user3 enters Raffle -> user3 enters Raffle -> user2 enters Raffle -> user1 enters Raffle -> user1 enters Raffle -> getTopPlayers() (432ms)
- ✓ scenario: 20 users enter Raffle for one time, user21 enters Raffle but he is outta top-20 (1399ms)

72 passing

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
Ticket.sol	100	100	100
RaffleImpl.sol	100	100	100

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.