



Smart Contract Audit Report for Quiiiz

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Table of Contents	2
Management Summary	2
Vulnerabilities by Risk	2
Approach	4
Introduction	4
Scope Overview	4
Scope Validation	4
Threat Model	5
Security Evaluation	5
Audit Findings	12
Merkle Tree Verification of Wrong Data	12
Incorrect Payout Logic	13
payOut() Can Be Called Multiple Times	14
Gas Refunds	14
Answers Visible in the Mempool	16
Wrong Order of Arguments	16
jackpotTime Unreasonably Long	18
Extra Fee Is not Refunded	19
Return Value of Transfer not Checked	19
Unused Code	20
Inconsistency in Requires	21
Usage of Solidity's transfer()	22
Bonus Point Logic	23
Unnecessary Function	25
Centralization Risk in serverError()	25

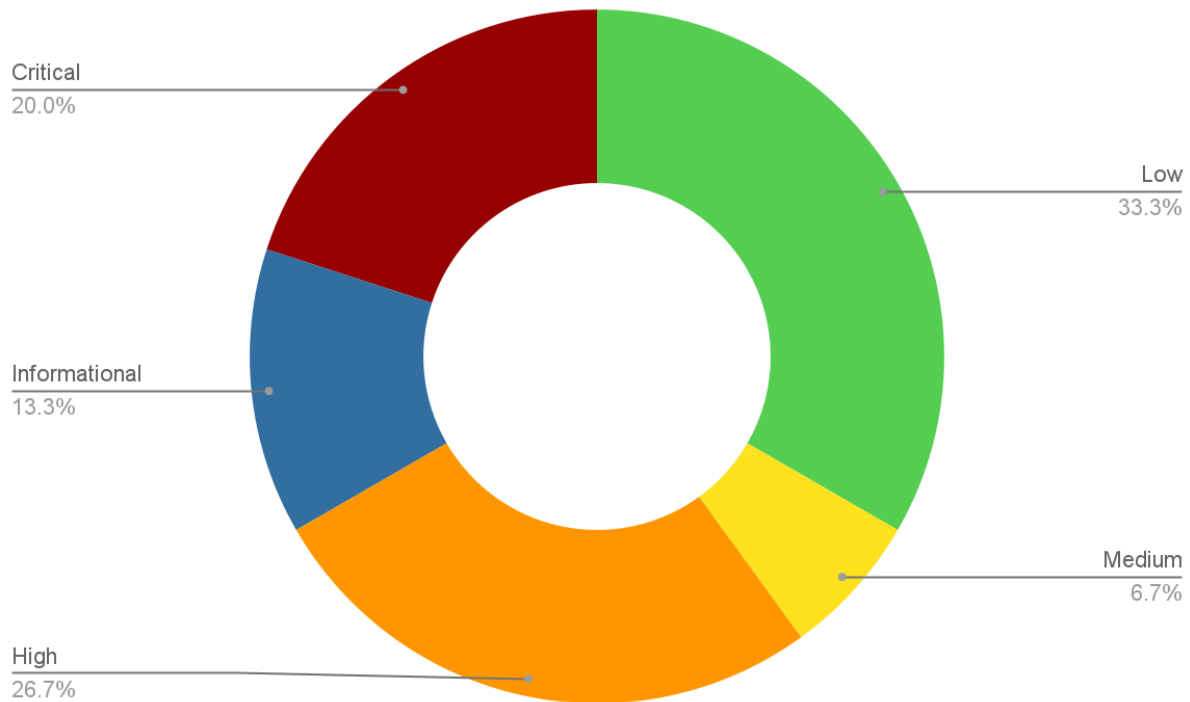
Management Summary

Quiiiz contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for Quiiiz smart contracts.

Over the audit period of 3 weeks, we discovered 15 vulnerabilities in the contract, There are 3 critical and 4 high-risk vulnerabilities.

Vulnerabilities by Risk



Risk	Informational	Low	Medium	High	Critical
# of issues	2	5	1	4	3

- **Critical** - Immediate or ongoing part of the business being exploited with direct key business losses.
- **High** - Direct threat to key business processes.
- **Medium** - Indirect threat to key business processes or partial threat to business processes.
- **Low** - No direct threat exists. The vulnerability may be exploited using other vulnerabilities.
- **Informational** - This finding does not indicate vulnerability, but states a comment that notifies about design flaws and improper implementation that might cause a problem in the long run.

Approach

Introduction

Quiiiz contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the aforementioned contracts.

Scope Overview

Together with the Quiiiz team we define that the scope of the project will be:

1. The audit was done on code shipped up to commit
469877537e765370647ba2efaf00e518ae1a14d4

Our tests were performed in January 2023.

Scope Validation

We began by ensuring that the scope defined to us by the client was technically logical. Deciding what scope is right for a given system is part of the initial discussion.

Threat Model

We defined that the largest current threat to the system is the ability of malicious users to abuse the protocol functionality.

Security Evaluation

The following test cases were the guideline while auditing the system. This checklist is a modified version of the [SCSVS v1.2](#), with improved grammar, clarity, conciseness, and additional criteria. Where there is a gap in the numbering, an original criterion was removed. Criteria that are marked with an asterisk were added by us.

Architecture, Design and Threat Modeling	Test Name
G1.2	Every introduced design change is preceded by threat modeling.
G1.3	The documentation clearly and precisely defines all trust boundaries in the contract (trusted relations with other contracts and significant data flows).
G1.4	The SCSVS, security requirements or policy is available to all developers and testers.
G1.5	The events for the (state changing/crucial for business) operations are defined.
G1.6	The project includes a mechanism that can temporarily stop sensitive functionalities in case of an attack. This mechanism should not block users' access to their assets (e.g. tokens).
G1.7	The amount of unused cryptocurrencies kept on the contract is controlled and at the minimum acceptable level so as not to become a potential target of an attack.
G1.8	If the fallback function can be called by anyone, it is included in the threat model.
G1.9	Business logic is consistent. Important changes in the logic should be applied in all contracts.
G1.10	Automatic code analysis tools are employed to detect vulnerabilities.
G1.11	The latest major release of Solidity is used.
G1.12	When using an external implementation of a contract, the most recent version is used.
G1.13	When functions are overridden to extend functionality, the super keyword is used to maintain previous functionality.
G1.14	The order of inheritance is carefully specified.
G1.15	There is a component that monitors contract activity using events.
G1.16	The threat model includes whale transactions.
G1.17	The leakage of one private key does not compromise the security of the entire project.

Policies and Procedures	Test Name
----------------------------	-----------

G2.2	The system's security is under constant monitoring (e.g. the expected level of funds).
G2.3	There is a policy to track new security vulnerabilities and to update libraries to the latest secure version.
G2.4	The security department can be publicly contacted and that the procedure for handling reported bugs (e.g., thorough bug bounty) is well-defined.
G2.5	The process of adding new components to the system is well defined.
G2.6	The process of major system changes involves threat modeling by an external company.
G2.7	The process of adding and updating components to the system includes a security audit by an external company.
G2.8	In the event of a hack, there's a clear and well known mitigation procedure in place.
G2.9	The procedure in the event of a hack clearly defines which persons are to execute the required actions.
G2.10	The procedure includes alarming other projects about the hack through trusted channels.
G2.11	A private key leak mitigation procedure is defined.

Upgradability	Test Name
G2.2	Before upgrading, an emulation is made in a fork of the main network and everything works as expected on the local copy.
G2.3	The upgrade process is executed by a multisig contract where more than one person must approve the operation.
G2.4	Timelocks are used for important operations so that the users have time to observe upcoming changes (please note that removing potential vulnerabilities in this case may be more difficult).
G2.5	<i>initialize()</i> can only be called once.
G2.6	<i>initialize()</i> can only be called by an authorized role through appropriate modifiers (e.g. <i>initializer</i> , <i>onlyOwner</i>).
G2.7	The update process is done in a single transaction so that no one can front-run it.
G2.8	Upgradeable contracts have reserved gap on slots to prevent overwriting.
G2.9	The number of reserved (as a gap) slots has been reduced appropriately if new variables have been added.
G2.10	There are no changes in the order in which the contract state variables are declared, nor their types.
G2.11	New values returned by the functions are the same as in previous versions of the contract (e.g. <i>owner()</i> , <i>balanceOf(address)</i>).
G2.12	The implementation is initialized.
G2.13	The implementation can't be destroyed.

Business Logic	Test Name
G4.2	The contract logic and protocol parameters implementation corresponds to the documentation.
G4.3	The business logic proceeds in a sequential step order and it is not possible to skip steps or to do it in a different order than designed.
G4.4	The contract has correctly enforced business limits.
G4.5	The business logic does not rely on the values retrieved from untrusted contracts (especially when there are multiple calls to the same contract in a single flow).
G4.6	The business logic does not rely on the contract's balance (e.g., <i>balance == 0</i>).
G4.7	Sensitive operations do not depend on block data (e.g., <i>block hash</i> , <i>timestamp</i>).
G4.8	The contract uses mechanisms that mitigate transaction-ordering (front-running) attacks (e.g. pre-commit schemes).
G4.9	The contract does not send funds automatically, but lets users withdraw funds in separate transactions instead.

Access Control	Test Name
G5.2	The principle of the least privilege is upheld. Other contracts should only be able to access functions and data for which they possess specific authorization.
G5.3	New contracts with access to the audited contract adhere to the principle of minimum rights by default. Contracts should have a minimal or no permissions until access to the new features is explicitly granted.
G5.4	The creator of the contract complies with the principle of the least privilege and their rights strictly follow those outlined in the documentation.
G5.5	The contract enforces the access control rules specified in a trusted contract, especially if the dApp client-side access control is present and could be bypassed.
G5.6	Calls to external contracts are only allowed if necessary.
G5.7	Modifier code is clear and simple. The logic should not contain external calls to untrusted contracts.
G5.8	All user and data attributes used by access controls are kept in trusted contracts and cannot be manipulated by other contracts unless specifically authorized.
G5.9	the access controls fail securely, including when a revert occurs.
G5.10	If the input (function parameters) is validated, the positive validation approach (whitelisting) is used where possible.

Communication	Test Name
G6.2	Libraries that are not part of the application (but the smart contract relies on to operate) are identified.

G6.3	Delegate call is not used with untrusted contracts.
G6.4	Third party contracts do not shadow special functions (e.g. revert).
G6.5	The contract does not check whether the address is a contract using <i>extcodesize</i> opcode.
G6.6	Re-entrancy attacks are mitigated by blocking recursive calls from other contracts and following the Check-Effects-Interactions pattern. Do not use the <i>send</i> function unless it is a must.
G6.7	The result of low-level function calls (e.g. <i>send</i> , <i>delegatecall</i> , <i>call</i>) from other contracts is checked.
G6.8	Contract relies on the data provided by the right sender and does not rely on tx.origin value.

Arithmetic	Test Name
G7.2	The values and math operations are resistant to integer overflows. Use SafeMath library for arithmetic operations before solidity 0.8.*.
G7.3	the unchecked code snippets from Solidity $\geq 0.8.*$ do not introduce integer under/overflows.
G7.4	Extreme values (e.g. maximum and minimum values of the variable type) are considered and do not change the logic flow of the contract.
G7.5	Non-strict inequality is used for balance equality.
G7.6	Correct orders of magnitude are used in the calculations.
G7.7	In calculations, multiplication is performed before division for accuracy.
G7.8	The contract does not assume fixed-point precision and uses a multiplier or store both the numerator and denominator.

Denial of Service	Test Name
G8.2	The contract does not iterate over unbound loops.
G8.3	Self-destruct functionality is used only if necessary. If it is included in the contract, it should be clearly described in the documentation.
G8.4	The business logic isn't blocked if an actor (e.g. contract, account, oracle) is absent.
G8.5	The business logic does not disincentivize users to use contracts (e.g. the cost of transaction is higher than the profit).
G8.6	Expressions of functions assert or require have a passing variant.
G8.7	If the fallback function is not callable by anyone, it is not blocking contract functionalities.
G8.8	There are no costly operations in a loop.
G8.9	There are no calls to untrusted contracts in a loop.
G8.10	If there is a possibility of suspending the operation of the contract, it is also

	possible to resume it.
G8.11	If whitelists and blacklists are used, they do not interfere with normal operation of the system.
G8.12	There is no DoS caused by overflows and underflows.

Blockchain Data	Test Name
G9.2	Any saved data in contracts is not considered secure or private (even private variables).
G9.3	No confidential data is stored in the blockchain (passwords, personal data, token etc.).
G9.4	Contracts do not use string literals as keys for mappings. Global constants are used instead to prevent Homoglyph attack.
G9.5	Contract does not trivially generate pseudorandom numbers based on the information from blockchain (e.g. seeding with the block number).

Gas Usage and Limitations	Test Name
G10.2	Gas usage is anticipated, defined and has clear limitations that cannot be exceeded. Both code structure and malicious input should not cause gas exhaustion.
G10.3	Function execution and functionality does not depend on hard-coded gas fees (they are bound to vary).

Clarity and Readability	Test Name
G11.2	The logic is clear and modularized in multiple simple contracts and functions.
G11.3	Each contract has a short 1-2 sentence comment that explains its purpose and functionality.
G11.4	Off-the-shelf implementations are used, this is made clear in comment. If these implementations have been modified, the modifications are noted throughout the contract.
G11.5	The inheritance order is taken into account in contracts that use multiple inheritance and shadow functions.
G11.6	Where possible, contracts use existing tested code (e.g. token contracts or mechanisms like <i>ownable</i>) instead of implementing their own.
G11.7	Consistent naming patterns are followed throughout the project.
G11.8	Variables have distinctive names.
G11.9	All storage variables are initialized.
G11.10	Functions with specified return type return a value of that type.

G11.11	All functions and variables are used.
G11.12	<i>require</i> is used instead of <i>revert</i> in <i>if</i> statements.
G11.13	The <i>assert</i> function is used to test for internal errors and the <i>require</i> function is used to ensure a valid condition in input from users and external contracts.
G11.14	Assembly code is only used if necessary.

Test Coverage	Test Name
G12.2	Abuse narratives detailed in the threat model are covered by unit tests.
G12.3	Sensitive functions in verified contracts are covered with tests in the development phase.
G12.4	Implementation of verified contracts has been checked for security vulnerabilities using both static and dynamic analysis.
G12.5	Contract specification has been formally verified.
G12.6	The specification and results of the formal verification is included in the documentation.

Decentralized Finance	Test Name
G14.1	The lender's contract does not assume its balance (used to confirm loan repayment) to be changed only with its own functions.
G14.2	Functions that change lenders' balance and/or lend cryptocurrency are non-re-entrant if the smart contract allows borrowing the main platform's cryptocurrency (e.g. Ethereum). It blocks the attacks that update the borrower's balance during the flash loan execution.
G14.3	Flash loan functions can only call predefined functions on the receiving contract. If it is possible, define a trusted subset of contracts to be called. Usually, the sending (borrowing) contract is the one to be called back.
G14.4	If it includes potentially dangerous operations (e.g. sending back more ETH/tokens than borrowed), the receiver's function that handles borrowed ETH or tokens can be called only by the pool and within a process initiated by the receiving contract's owner or another trusted source (e.g. multisig).
G14.5	Calculations of liquidity pool share are performed with the highest possible precision (e.g. if the contribution is calculated for ETH it should be done with 18 digit precision - for Wei, not Ether). The dividend must be multiplied by the 10 to the power of the number of decimal digits (e.g. dividend * 10 ¹⁸ / divisor).
G14.6	Rewards cannot be calculated and distributed within the same function call that deposits tokens (it should also be defined as non-re-entrant). This protects from momentary fluctuations in shares.
G14.7	Governance contracts are protected from flash loan attacks. One possible

	mitigation technique is to require the process of depositing governance tokens and proposing a change to be executed in different transactions included in different blocks.
G14.8	When using on-chain oracles, contracts are able to pause operations based on the oracles' result (in case of a compromised oracle).
G14.9	External contracts (even trusted ones) that are allowed to change the attributes of a project contract (e.g. token price) have the following limitations implemented: thresholds for the change (e.g. no more/less than 5%) and a limit of updates (e.g. one update per day).
G14.10	Contract attributes that can be updated by the external contracts (even trusted ones) are monitored (e.g. using events) and an incident response procedure is implemented (e.g. during an ongoing attack).
G14.11	Complex math operations that consist of both multiplication and division operations first perform multiplications and then division.
G14.12	When calculating exchange prices (e.g. ETH to token or vice versa), the numerator and denominator are multiplied by the reserves (see the <i>getInputPrice</i> function in the <i>UniswapExchange</i> contract).

Audit Findings

Merkle Tree Verification of Wrong Data

Status	Open
Risk	Critical
Location	Factory.sol - <i>function verifyParticipant</i>
Tools	Manual testing
Description	<p><i>verifyParticipant</i> in the factory hashes <i>msg.sender</i> and checks that for inclusion in the Merkle tree:</p> <pre>function verifyParticipant(bytes32[] calldata proof) external view returns (bool) { bytes32 leaf = keccak256(abi.encodePacked(msg.sender)); require(MerkleProof.verify(proof, merkleRoot, leaf), "Invalid proof"); return true; }</pre>
	<p>However, this function is called by the game contracts, so the <i>msg.sender</i> is going to be these contracts and not the participants.</p>
Mitigation	Verify the address of a participant, not <i>msg.sender</i> .

Incorrect Payout Logic

Status	Open
Risk	Critical
Location	SoftMode.sol - <i>function payOut</i>
Tools	Manual testing
Description	<p>The function <i>payOut</i> has the following check:</p> <pre>users[winners[winners.length - 1]].points == topPoints</pre> <p>However, after the first winner is popped off, the value of <i>winners[winners.length - 1]</i> will no longer be equal to <i>topPoints</i> (except in edge cases with equal points), so there is no way for the other users to get their allocation.</p> <p>More generally, this design where the end is always popped off and a new dispute period are started is a bit strange.</p>
Mitigation	Could the system have one dispute period where anyone can submit fraud proofs and one payout period?

payOut() Can Be Called Multiple Times

Status	Open
Risk	Critical
Location	SquidMode.sol - <i>function payOut</i>
Tools	Manual testing
Description	<p>When there are multiple winners, <i>SquidMode.payOut</i> can be called multiple times for the same address.</p> <pre>else if(winners > 1) { uint calc = (address(this).balance / winners); winners --; payable(winner).transfer(calc); }</pre> <p>This means that one winner can steal the funds of all other winners by repeatedly calling this function.</p>
Mitigation	Make sure the winner can not double claim the payout

Gas Refunds

Status	Open
Risk	High
Location	SoftMode.sol & SquidMode.sol - <i>functions claimGas, claimGasBack, escape & countGasSpent</i>
Tools	Manual testing
Description	<p><i>claimGasBack</i> requires that all gas claim-backs were processed and then transfers the balance of the contract.</p> <pre>function claimGasBack() external onlyOwner{ require(totalGasSpent == 0, "Gas refunds not finished"); require(address(this).balance > 0, "No balance"); payable(jackpot).transfer(address(this).balance); }</pre> <p>However, no gas fees will be included in the accounts balance. The same assumption was also made while writing <i>escape</i>. <i>totalGasSpent</i> is subtracted from the refund, but gas is not included in the balance:</p> <pre>payable(msg.sender).transfer(refund - totalGasSpent);</pre> <p>Because gas fees are not included in the smart contract's balance, this refund system needs to be rethought.</p>
Mitigation	Something like meta transactions should be looked into, but this requires a more involved system.

Answers Visible in the Mempool

Status	Open
Risk	High
Location	SoftMode.sol & SquidMode.sol - <i>function answerQuestion</i>
Tools	Manual testing
Description	<p><i>Need validation from Quiiiz's team.</i></p> <p>Submitted answers are recorded in each user's object.</p> <pre>users[msg.sender].answer = answer;</pre> <p>They are therefore visible in the mempool. A malicious user can perhaps deduce the correct answers by seeing what others responded to.</p>
Mitigation	Architectural rethinking.

Wrong Order of Arguments

Status	Open
Risk	High
Location	Factory.sol - <i>function createGameContract</i>
Tools	Manual testing
Description	<p>In <i>createGameContract</i> we see</p> <pre>ISquidMode(clone).initialize(timeOfRound, answers, corpFee, jackpotFee, players, entryFee, jackpotMode);</pre> <p>But the signature of <code>SquidMode.initialize</code> is</p> <pre>function initialize(uint _timeOfRound, bytes32[] calldata _answers, uint _fee, uint _maxPlayers, uint _jackpotFee, uint _entryFee, bool _jackpotMode)</pre> <p>Argument 4 is <i>_maxPlayers</i>, not <i>_jackpotFee</i>.</p>
Mitigation	Switch the order of <i>players</i> and <i>jackpotFee</i> .

jackpotTime Unreasonably Long

Status	Open
Risk	High
Location	Factory.sol - <i>function createGameContract</i> & <i>function createSoftGame</i>
Tools	Manual testing
Description	<p>In the constructor <i>jackpotTime</i> is defined as</p> <pre>jackpotTime = block.number + _jackpotTime;</pre> <p>However in line 104 and line 140 we see.</p> <pre>if (jackpotMode == true) { require(block.number >= jackpotTime, "Jackpot time has expired"); jackpotTime = block.number + jackpotTime; }</pre> <p>Because <i>jackpotTime</i> is already a previous block number, the new time will be more or less double the old time. This will mean no jackpots for years to come.</p>
Mitigation	If this was not intended, come up with a more reasonable jackpot time calculation.

Extra Fee Is not Refunded

Status	Open
Risk	Medium
Location	SoftMode.sol & SquidMode.sol - <i>function enter</i>
Tools	Manual testing
Description	<p>In <i>enter</i> function, it is required that the provided ETH is equal to or greater than the <i>entryFee</i>:</p> <pre>require(msg.value >= entryFee, "Not enough ether sent");</pre> <p>But additional ETH is not reimbursed, which can lead to situations where users pay too much.</p>
Mitigation	Either requires that the amount is equal to <i>entryFee</i> or reimburses the user for sending extra funds.

Return Value of Transfer not Checked

Status	Open
Risk	Low
Location	Factory.sol - <i>function createGameContract</i>
Tools	Manual testing
Description	On line 225, The return value of <i>transfer</i> is not checked. While this is not a significant issue because USDC reverts on failure, it is best practice to check return values when interacting with ERC20 tokens.
Mitigation	Check the return value of <i>transfer</i> .

Unused Code

Status	Open
Risk	Low
Location	Factory.sol - <i>function setGasRefund</i>
Tools	Manual testing
Description	<p>The variable <i>maxGasRefund</i> (line 50) and the associated function <i>setGasRefund</i> are never used other than in comments. They can be removed.</p> <pre>function setGasRefund(uint256 _maxGasRefund) external onlyOwner { maxGasRefund = _maxGasRefund; }</pre>
Mitigation	Remove the unused variables.

Inconsistency in Requires

Status	Open
Risk	Low
Location	Factory.sol - <i>function createSoftGame</i>
Tools	Manual testing
Description	<p>There is a small inconsistency between the functions <i>createSoftGame</i> and <i>createGameContract</i>.</p> <p><i>createSoftGame</i> requires that</p> <pre>require (block.number > jackpotTime, "Jackpot time has expired");</pre> <p>Whereas <i>createGameContract</i> requires that</p> <pre>require(block.number >= jackpotTime, "Jackpot time has expired");</pre>
Mitigation	Choose to either use >= or > in both requires.

Usage of Solidity's *transfer()*

Status	Open
Risk	Low
Location	—
Tools	Manual testing
Description	The system uses <i>transfer</i> in many places. This is discouraged . Solidity's <i>transfer()</i> uses a hardcoded gas amount, 2300. This means that using this function introduces a hard dependency on this stipend. but gas costs for EVM opcodes are sometimes changed.
Mitigation	<code>.call.value(...)(<i>""</i>)</code> should be used instead, but more robust reentrancy prevention should be employed.

Bonus Point Logic

Status	Open
Risk	Low
Location	SoftMode.sol & SquidMode.sol - <i>function evaluateQuestion</i>
Tools	Manual testing
Description	<p>The following logic is used to award bonus points:</p> <pre>users[msg.sender].points += users[msg.sender].correct * users[msg.sender].responseTime;</pre> <p>This multiplies <i>responseTime</i> (which only tracks the response time per round) by the <i>correct</i> counter (which tracks the number of correct replies).</p> <p>It would mean that responding quickly in the last round will be much more beneficial (because it is multiplied by the number of correct answers) than responding quickly in the beginning.</p>
Mitigation	If this is not intended, figure out a more sensible way to award bonus points.

Unnecessary Function

Status	Open
Risk	Information
Location	SoftMode.sol & SquidMode.sol - <i>function initiateFinalEval</i>
Tools	Manual testing
Description	The function <i>initiateFinalEval</i> may be redundant. <i>passSalt</i> also sets the <i>state</i> to <i>states.finalEval</i> (and adds the last salt, which is very important for <i>evalOnLastRound</i>), so <i>passSalt</i> could always be used.
Mitigation	Remove <i>initiateFinalEval</i> .

Centralization Risk in *serverError()*

Status	Open
Risk	Information
Location	SoftMode.sol & SquidMode.sol - <i>function serverError</i>
Tools	Manual testing
Description	<i>serverError</i> carries a significant centralization risk, as the owner can always call this function to end the auction.
Mitigation	Consider more decentralized alternatives.