



SMART CONTRACT AUDIT REPORT

for

THAW.bet



Prepared By: Xiaomi Huang

PeckShield
September 15, 2025

Document Properties

Client	THAW.bet
Title	Smart Contract Audit Report
Target	THAW.bet
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 15, 2025	Xuxian Jiang	Final Release
1.0-rc	September 3, 2025	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About THAW.bet	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Possible Out-of-Bound Access in VideoPoker::entropyCallback()	12
3.2	Lack of Caller Validation in Mines::Mines_setMultiplier()	13
3.3	Trust Issue of Admin Keys	14
3.4	Revisited Cleanup Logic in FishPrawnCrab::FishPrawnCrab_Refund()	16
3.5	Improved Validation on Function Arguments	17
3.6	Possible totalPool Inflation With Reduced Share in BankrollAndStaking	18
3.7	Inconsistent Member Fields in Various Games Play Events	20
4	Conclusion	22
	References	23

1 | Introduction

Given the opportunity to review the design document and related source code of the THAW.bet protocol, we outline in the report 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 can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About THAW.bet

THAW.bet is a blockchain-powered gaming platform that delivers a complete casino experience through decentralized smart contracts. Utilizing Pyth Network's Entropy protocol for provably fair randomness, the platform operates on a unified treasury system with integrated staking mechanisms that optimize liquidity distribution and employ Kelly Criterion risk management for exposure control. Players can wager with various ERC-20 tokens or native assets across games that feature comprehensive security implementations including access validation, reentrancy protection, and automatic refunds for entropy failures. The modular architecture enables independent game contracts to share core infrastructure for randomness generation, payment processing, and treasury management, creating a scalable and secure gaming ecosystem. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of THAW.bet

Item	Description
Name	THAW.bet
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	September 15, 2025

In the following, we show the Git repository of reviewed files and the commit hash value used in

this audit.

- <https://github.com/Narwhal-Finance/carnival-contracts-audit.git> (6275141)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/Narwhal-Finance/carnival-contracts-audit.git> (210a54e)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Table 1.3: 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
Advanced DeFi Scrutiny	Business Logics Review
	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	

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.4: 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 of 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 exploitable 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.




comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.



2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the THAW.bet protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our 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	# of Findings	
Critical	0	
High	0	
Medium	3	
Low	3	
Informational	1	
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational issue.

Table 2.1: Key THAW.bet Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Out-of-Bound Access in VideoPoker::entropyCallback()	Coding Practices	Resolved
PVE-002	Medium	Lack of Caller Validation in Mines::Mines_setMultiplier()	Security Features	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Low	Revisited Cleanup Logic in FishPrawn-Crab_Refund()	Business Logic	Resolved
PVE-005	Low	Improved Validation of Function Arguments	Coding Practice	Resolved
PVE-006	Low	Possible totalPool Inflation With Reduced Share in BankrollAndStaking	Time And State	Resolved
PVE-007	Informational	Inconsistent Member Fields in Various Games Play Events	Coding Practice	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contract is being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Out-of-Bound Access in VideoPoker::entropyCallback()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: VideoPoker
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

The THAW.bet protocol supports a so-called `VideoPoker` game where players get dealt a 5 card hand and can replace any number of cards to form winning combinations. While analyzing the game logic, we notice current implementation may lead to possible out-of-bounds access.

In the following, we show the code snippet of the related `entropyCallback()` function, which is called by `PYTH Entropy` with random numbers. We notice the deck has an initial array of 52 cards. The following code snippet in essence removes the cards in hold from the initial array. However, the second loop (line 307) still iterates from 0 (inclusive) to 52 (exclusive) and the last (few) tries may lead to out-of-bounds access of the deck array (after being downsized).

```
306     for (uint256 g = 0; g < 5; g++) {
307         for (uint256 j = 0; j < 52; j++) {
308             if (
309                 game.cardsInHand[g].number == deck[j].number &&
310                 game.cardsInHand[g].suit == deck[j].suit
311             ) {
312                 deck[j] = deck[deck.length - 1];
313                 assembly {
314                     mstore(deck, sub(mload(deck), 1))
315                 }
316                 break;
317             }
318         }
```

319

}

Listing 3.1: VideoPoker::entropyCallback()

Meanwhile, there is an arithmetic underflow issue in `Dice::entropyCallback()`, `CoinFlip::entropyCallback()`, and `Roulette::entropyCallback()`, which may lead to revert and abort the game. In the following `Dice` example, an arithmetic underflow occurs when `gamePayout` is smaller than `game.wager` (lines 230 and 237).

```

227     for (i = 0; i < game.numBets; i++) {
228         diceOutcomes[i] = getRandomWithEntropy(randomNumber,i) % 10000000;
229         if (diceOutcomes[i] >= numberToRollOver && game.isOver == true) {
230             totalValue += int256(gamePayout - game.wager);
231             payout += gamePayout;
232             payouts[i] = gamePayout;
233             continue;
234         }

236         if (diceOutcomes[i] <= winChance && game.isOver == false) {
237             totalValue += int256(gamePayout - game.wager);
238             payout += gamePayout;
239             payouts[i] = gamePayout;
240             continue;
241         }

243         totalValue -= int256(game.wager);
244     }

```

Listing 3.2: Dice::entropyCallback()

Recommendation Revise the above-mentioned routines to resolve possible out-of-array access violation and arithmetic underflows.

Status This issue has been fixed in the following commit: 4143f2f.

3.2 Lack of Caller Validation in Mines::Mines_setMultiplier()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Mines
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

The `THAW`.bet protocol has a built-in support of the `Mines` game: a player has 25 tiles where mines are hidden, and players flip tiles until they cash out or reveal a mine in which case they lose. We notice

this game has a public function named `Mines_setMultiplier()`, which is not guarded. In other words, it allows any one to set up the game parameters, i.e., `minesMultipliers`.

```
400     function Mines_SetMultipliers(uint256 numMines) external {
401         if (numMines == 0 || numMines >= 25) {
402             revert();
403         }
404         if (minesMultipliers[numMines][1] != 0) {
405             revert();
406         }
407
408         for (uint256 g = 1; g <= 25 - numMines; g++) {
409             uint256 multiplier = 1;
410             uint256 divisor = 1;
411             for (uint256 f = 0; f < g; f++) {
412                 multiplier *= (25 - numMines - f);
413                 divisor *= (25 - f);
414             }
415             minesMultipliers[numMines][g] =
416                 (edgeFactor * (10 ** 9)) /
417                 ((multiplier * (10 ** 9)) / divisor);
418         }
419     }
```

Listing 3.3: `Mines::Mines_setMultiplier()`

Recommendation Revise the above routine to validate the caller to be same as `Bankroll.getOwner()`.

Status This issue has been fixed in the following commit: [4143f2f](#).

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `THAW.bet` protocol, there is a privileged account `owner` that plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, whitelist quotas, collect funds, and pause/upgrade protocol). The account also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized.

In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
189     function setTokenAddress(  
190         address tokenAddress,  
191         bool isValid  
192     ) external onlyOwner {  
193         isTokenAllowed[tokenAddress] = isValid;  
194         emit Bankroll_Token_State_Changed(tokenAddress, isValid);  
195     }  
196     ...  
197     function setWrappedAddress(address wrapped) external onlyOwner {  
198         require(wrapped != address(0), "Invalid wrapped token address");  
199         wrappedToken = wrapped;  
200     }  
201  
202     function setFeeInfo(address _receiver, uint256 _rate) external onlyOwner {  
203         require(_receiver != address(0), "Invalid receiver address");  
204         feeReceiver = _receiver;  
205         feeRate = _rate;  
206  
207         emit BankRoll_Fee_Info(_receiver, _rate);  
208     }  
209  
210     function setNarStakingContract(address _narStakingContract) external onlyOwner {  
211         require(_narStakingContract != address(0), "Invalid staking contract address");  
212         narStakingContract = _narStakingContract;  
213         emit NewNarStakingContract(_narStakingContract);  
214     }  
215  
216     function setWhitelistQuota(address _user, address _token, uint256 _quota) external  
217         onlyOwner {  
218         whitelistQuota[_user][_token] = _quota;  
219         emit NewWhiteQuota(_user, _token, _quota);  
220     }  
221  
222     function setQuotaRatio(address _token, uint256 _ratio) external onlyOwner {  
223         quotaRatio[_token] = _ratio;  
224         emit NewQuotaRatio(_token, _ratio);  
225     }  
226  
227     function setMinLockPeriod(uint256 _duration) external onlyOwner {  
228         MIN_LOCK_PERIOD = _duration;  
229         emit NewMinLock(_duration);  
230     }
```

Listing 3.4: Example Privileged Functions in BankrollAndStaking

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

In the meantime, the protocol contracts are deployed as proxies to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team makes use of a multisig to act as the privileged owner.

3.4 Revisited Cleanup Logic in `FishPrawnCrab::FishPrawnCrab_Refund()`

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FishPrawnCrab`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

THAW.bet supports a number of casino games. One specific game is `FishPrawnCrab` game, which is traditional Asian dice game using three dice with fish, prawn, and crab symbols. Players bet on specific symbol combinations with varying payout multipliers. While examining the refund logic, we notice an issue that does not properly clean up all game-related state.

To elaborate, we show below the implementation of the related `FishPrawnCrab_Refund()` routine that basically refunds the game if VRF request fails. For each ongoing `FishPrawnCrab` game, there are two states, i.e., `fishPrawnCrabIDs` and `fishPrawnCrabGames`. Our analysis shows that current refund logic only cleans up the `fishPrawnCrabGames` state (line 151), not the `fishPrawnCrabIDs` state.

```
139     function FishPrawnCrab_Refund() external nonReentrant {
140         address msgSender = _msgSender();
141         FishPrawnCrabGame storage game = fishPrawnCrabGames[msgSender];
142         if (game.requestID == 0) {
143             revert NoRequestPending();
144         }
145         if (game.blockNumber + 200 > block.number) {
146             revert BlockNumberTooLow(block.number, game.blockNumber + 200);
147         }
148
149         uint256 totalWager = game.totalWager;
150         address tokenAddress = game.tokenAddress;
```

```

151     delete (fishPrawnCrabGames[msgSender]);
152     if (tokenAddress == address(0)) {
153         (bool success, ) = payable(msgSender).call{value: totalWager}("");
154         if (!success) {
155             revert TransferFailed();
156         }
157     } else {
158         IERC20(tokenAddress).safeTransfer(msgSender, totalWager);
159     }
160     emit FishPrawnCrab_Refund_Event(msgSender, totalWager, tokenAddress);
161 }

```

Listing 3.5: FishPrawnCrab::FishPrawnCrab_Refund()

Recommendation Revise the above routine to properly clean up game-related state once it is refunded. Note the same issue also affects other games, including Baccarat, Mines, Crash, and Roulette.

Status This issue has been fixed in the following commit: 4143f2f.

3.5 Improved Validation on Function Arguments

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Slots
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

Among the supported games, there is a slots game where players put in a wager and receive payout depending on the slots outcome. Within this game, there is a setter routine `_setSlotsMultipliers()`, which is used to configure multipliers used in the game. This routine can only be called at deploy time and may be improved with strengthened input validation.

To elaborate, we show below the implementation of this `_setSlotsMultipliers()` routine. Our analysis shows it can be improved by validating the given arrays of `_multipliers` and `_outcomeNum` share the same length and the length is equal to the given third parameter of `_numOutcomes`.

```

282     function _setSlotsMultipliers(
283         uint16[] memory _multipliers,
284         uint16[] memory _outcomeNum,
285         uint16 _numOutcomes
286     ) internal {
287         for (uint16 i = 0; i < numOutcomes; i++) {
288             delete (slotsMultipliers[i]);

```

```

289     }
290
291     numOutcomes = _numOutcomes;
292     for (uint16 i = 0; i < _multipliers.length; i++) {
293         slotsMultipliers[_outcomeNum[i]] = _multipliers[i];
294         if (_multipliers[i] > maxMultiplier) {
295             maxMultiplier = _multipliers[i];
296         }
297     }
298 }

```

Listing 3.6: Slots::_setSlotsMultipliers()

Recommendation Improve the parameters validation in the above `_setSlotsMultipliers()` routine.

Status This issue has been fixed in the following commit: 4143f2f.

3.6 Possible totalPool Inflation With Reduced Share in BankrollAndStaking

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: BankrollAndStaking
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

The THAW.bet protocol has a core `BankrollAndStaking` that is responsible for keeping the bankroll and distribute payouts. While reviewing the current logic to compute the share amount based on the deposit amount, we notice an issue that may reduce the share amount to return if the pool is not properly bootstrapped.

To elaborate, we show below the related `deposit()` routine. As the name indicates, this routine is used to deposit the supported tokens and gain respective share of the pool. However, it comes to our attention that if the pool is just initialized, the first user may simply deposit 1 WEI, followed up by a donation of huge amount of the same token. By doing so, the second user may result in zero share if the amount is smaller than the donated amount.

```

268     function deposit(address _token, uint256 _amount) external payable nonReentrant {
269         require(isTokenAllowed[_token], "not whitelist token");
270         require(_amount > 0, "Amount must be greater than 0");
271         uint256 newShares;
272         uint256 totalPool;

```

```
273     uint256 userAmount;
274     if(_token == address(0)){
275         totalPool = address(this).balance - _amount;
276     }else{
277         totalPool = IERC20(_token).balanceOf(address(this));
278     }
279     if (tokenTotalShares[_token] == 0) {
280         newShares = _amount;
281         userAmount = 0;
282     } else {
283         newShares = (_amount * tokenTotalShares[_token]) / totalPool;
284         userAmount = (userShares[msg.sender][_token] * totalPool) / tokenTotalShares
                [_token];
285     }
286     require(userAmount + _amount <= getUserQuota(msg.sender, _token), "Staking amount
        exceeds limit");
287
288     if(_token == address(0)){
289         if (msg.value < _amount) {
290             revert InvalidValue(_amount, msg.value);
291         }
292     } else {
293         IERC20(_token).safeTransferFrom(
294             msg.sender,
295             address(this),
296             _amount
297         );
298     }
299     userShares[msg.sender][_token] += newShares;
300     tokenTotalShares[_token] += newShares;
301     unlockTimestamp[msg.sender][_token] = block.timestamp;
302     emit BankRoll_Stake(msg.sender, _token, _amount, newShares, userShares[msg.sender][
        _token], tokenTotalShares[_token], totalPool);
303 }
```

Listing 3.7: BankrollAndStaking::deposit()

Recommendation Revise the above deposit logic to reliably compute the pool share. The best practice may simply perform the very first deposit upon the contract deployment by the protocol team.

Status This issue has been fixed in the following commit: 210a54e.

3.7 Inconsistent Member Fields in Various Games Play Events

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

In Ethereum, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `CoinFlip` contract as an example. This contract has a public entry function that is used to start the game, which basically takes the user wager, saves bet parameters, and makes a request to the VRF. However, it comes to our attention that the emitted `CoinFlip_Play_Event` event, by its definition, has a last field named `VRFfee`, which is emitted with current request sequence number. With that, there is a need to revise its definition to use the sequence number, not `VRFfee`. Note the same issue also affects a number of contracts, including `Plinko`, `RockPaperScissors`, and `Slots`.

```
103     function CoinFlip_Play(  
104         uint256 wager,  
105         address tokenAddress,  
106         bool isHeads,  
107         uint32 numBets  
108     ) external payable nonReentrant {  
109         address msgSender = _msgSender();  
110         if (coinFlipGames[msgSender].requestID != 0) {  
111             revert AwaitingVRF(coinFlipGames[msgSender].requestID);  
112         }  
113         if (!(numBets > 0 && numBets <= wagerNumber)) {  
114             revert InvalidNumBets(wagerNumber);  
115         }  
116         _kellyWager(wager, tokenAddress);  
117         uint256 fee = getRandomFee();  
118         _transferWager(tokenAddress, wager * numBets, fee, msgSender);  
  
120         uint256 id = _requestRandomWords(keccak256(abi.encodePacked(block.timestamp,  
121             block.number, msgSender)), fee);  
  
122         coinFlipGames[msgSender] = CoinFlipGame(  
123             wager,
```

```
124         id,
125         tokenAddress,
126         uint64(block.number),
127         numBets,
128         isHeads
129     );
130     coinIDs[id] = msgSender;

132     emit CoinFlip_Play_Event(
133         msgSender,
134         wager,
135         tokenAddress,
136         isHeads,
137         numBets,
138         id
139     );
140 }
```

Listing 3.8: CoinFlip::CoinFlip_Play()

Recommendation Accurately emit the respective play event when the new game is started.

Status This issue has been fixed in the following commit: 4143f2f.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the THAW.bet protocol, which is a blockchain-powered gaming platform that delivers a complete casino experience through decentralized smart contracts. Utilizing Pyth Network's Entropy protocol for provably fair randomness, the platform operates on a unified treasury system with integrated staking mechanisms that optimize liquidity distribution and employ Kelly Criterion risk management for exposure control. Players can wager with various ERC-20 tokens or native assets across games that feature comprehensive security implementations including access validation, reentrancy protection, and automatic refunds for entropy failures. The modular architecture enables independent game contracts to share core infrastructure for randomness generation, payment processing, and treasury management, creating a scalable and secure gaming ecosystem. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based 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.

References

- [1] MITRE. CWE-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [3] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [4] MITRE. CWE-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [6] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [7] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [8] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [9] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.html>.

[10] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.

[11] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[12] PeckShield. PeckShield Inc. <https://www.peckshield.com>.

