our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Yeet Cup

SECURITY REVIEW

Date: 20 May 2024

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Yeet Cup

Yeet It is a fully on-chain game with no dominant game theoretic strategy to win. Players can make money in multiple ways.

The game is played by depositing `$BERA` into the pool, and the winner is rewarded the pool of assets once the finish condition is met. The game also has simple token economics attached to it where we will be giving out daily rewards in the form of `$YEET` tokens to players who deposit assets into the pool during that period. The rewards are distributed each epoch (24 hours) and are done so pro-rata to each player according to how much `$BERA` they deposited in the epoch that has just elapsed.

Anyone can also stake their `$YEET` tokens to earn a share of the assets that are deposited into the pool by players.

The core premise of the game is:

1. Players yeet `$BERA` into a pool
2. The last player to make a yeet, wins **80 percent** of the whole pool
   - **20 percent** is split among 10 random players
3. You can yeet as much as you want as long as it is equal to or greater than **0.5 percent** of the current pool value (this ensures that the game eventually ends and that there is a winner)
4. You earn `$YEET` tokens by making yeets
5. **10 percent** of every yeet is taxed and is split among:
   - **7 percent** to **$YEET** stakers
   - **1 percent** to the **$BERA/$YEET** liquidity pool - to be placed strategically in the v3 pool
   - **2 percent** to BERA8 - retroactive ecosystem and public goods funding by The Honey Jar

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 1 week with a total of 200 hours dedicated to the audit by five researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying several Critical and High severity issues, concerning wrong calculations and loss of rewards, among other issues with lower severity.

The code lacks tests of any kind which hindered the auditing process to a certain extent. Nevertheless, the Yeet team's responsiveness and extensive feedback to all code and logic-related questions made up for this.

5.1 Protocol Summary

Project Name	**Yeet Cup**
Repository	yeet-cup
Type of Project	Social GameFI Project
Audit Timeline	7 days
Review Commit Hash	f43ad283290293e18e5d9ab0c9d56e29bffa3eb3
Fixes Review Commit Hash	a89aae41116904eea4e1b5293b9d0be219a6bbd5

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
src/Stake.sol	113
src/Yeet.sol	290
src/NFTVesting.sol	57
src/Reward.sol	94
src/YeetToken.sol	10
src/YeetGameSettings.sol	69
src/RewardSettings.sol	17
src/Yeetback.sol	91
src/INFTContract.sol	3
Total	**744**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **4**
- **Medium** issues: **4**
- **Low** issues: **6**

ID	**Title**	**Severity**	**Status**
[C-01]	Direct Loss of Rewards on Re-Staking	Critical	Fixed
[C-02]	User Claimable Reward Tokens Can Be Locked	Critical	Fixed
[H-01]	Re-staking Mechanism Can Lead to Contract Insolvency	High	Fixed
[H-02]	Yeet Functionality Flaw Upon Contract Deployment or Restart Due to Wrong Calculation	High	Fixed
[M-01]	Missing Vesting Period Verification in `unstake()` Method Puts Users at Risk	Medium	Fixed
[M-02]	Significantly Decreased Entropy in `draftWinners()` Method	Medium	Fixed
[M-03]	Dysfunctional Instantiation of the `Yeetback.sol` Contract	Medium	Fixed

[M-04]	Incomplete Validation	Medium	Fixed
[L-01]	The `latestHistoryArr` Storage Array Is Ever Growing	Low	Fixed
[L-02]	`call()` Should Be Used Instead of `transfer()` on an `address payable`	Low	Fixed
[L-03]	`yeet()` Function Should Be Pauseable	Low	Acknowledged
[L-04]	Potentially Stuck Funds	Low	Acknowledged
[L-05]	Wrong Event Value Causes Confusion	Low	Fixed
[L-06]	Avoid Drafting the Same Winner Multiple Times by the Increasing Pseudo-Randomness in `Yeetback.sol`	Low	Fixed

7. Findings

[C-01] Direct Loss of Rewards on Re-Staking

Severity

Critical Risk

Description

The `reStake()` method of the `DiscreteStakingRewards` contract intends to add a user's (`msg.sender`) earned rewards to their staked balance instead of claiming those rewards. This is done by clearing the user's earned rewards and forwarding them to the contract's `stake()` method.
However, the `stake()` method is called externally and therefore the `msg.sender` will be the contract itself (`address(this)`) during the stake call.

Consequently, the user's earned rewards are re-staked on behalf of the contract instead of the user and are therefore lost. Furthermore, the re-staked rewards become stuck, since there is no way for the contract to redeem those "self-staked" amounts.

Location of Affected Code

File: src/Stake.sol#L163

```
/// @notice The function used to re-stake the rewards without claiming
    them, bypassing the vesting period
/// @dev The rewards are staked again
function reStake() external {
    _updateRewards(msg.sender);
    uint reward = earned[msg.sender];
    require(reward > 0, "No rewards to claim");

    earned[msg.sender] = 0;
    this.stake(reward); // @audit msg.sender will be address(this) in
       stake call
}
```

Recommendation

Directly increase the user's (`msg.sender`) staked balance:

```
function reStake() external {
    _updateRewards(msg.sender);
    uint reward = earned[msg.sender];
    require(reward > 0, "No rewards to claim");

    earned[msg.sender] = 0;

+   balanceOf[msg.sender] += reward;
+   totalSupply += reward;
-   this.stake(reward);
}
```

Team Response

Fixed as suggested.

[C-02] User Claimable Reward Tokens Can Be Locked

Severity

Critical Risk

Description

In the `Reward.sol` file, the `currentEpoch` variable is defined as a `uint8`, which means it can only hold values from 0 to 255. Since each epoch lasts for a day, the maximum epoch possible is indeed 255. However, this setup could potentially lead to a Denial of Service (DoS) scenario for the entire protocol. This issue arises because when the epoch reaches 255, any attempt to increment it further, like during the `addYeetVolume()` function that calls `_endEpoch()` internally upon `yeeting`, will cause an overflow, resulting in a revert. Therefore, ending the 255th epoch is impossible.

The `getClaimableAmount()` function loops through each epoch. However, it is important to note that although the maximum value of `currentEpoch` is 255, no amounts can be claimed beyond that point because no new epochs can be finalized.

Attack Scenario

To understand better the vulnerability let's look at the following example:

1. The `currentEpoch` is `255`.
2. Alice yeets and now participates in epoch `255` and `userYeetVolume[currentEpoch][user] += amount;` mapping will update Alice's data for the current epoch.
3. Once Epoch 255 concludes, the protocol attempts to increment the value, but hits a roadblock—it's unable to proceed. Consequently, the protocol stalls, unable to advance to the next epochs.
4. Alice tries to claim her rewards for epoch `255` but the tx will return 0, because `epoch < currentEpoch` is going to be `false`.

Location of Affected Code

File: src/Reward.sol#L135

```
/// @notice Calculate the amount of tokens that a user can claim
/// @dev if no one yeets, the epoch will never end, and the user can
   never claim (should not really be a problem)
function getClaimableAmount(address user) public view returns (uint256) {
    uint256 totalClaimable = 0;

// Fixed-point arithmetic for more precision
    uint256 scalingFactor = 1e18;

@>  for (uint256 epoch = lastClaimedForEpoch[user] + 1; epoch <
   currentEpoch; epoch++) {
        if (totalYeetVolume[epoch] == 0) continue; // Avoid division by
           zero

        uint256 userVolume = userYeetVolume[epoch][user];
        uint256 totalVolume = totalYeetVolume[epoch];

        uint256 userShare = (userVolume * scalingFactor) / totalVolume;

        uint256 maxClaimable = (epochRewards[epoch] / rewardsSettings.
           MAX_CAP_PER_WALLET_PER_EPOCH_FACTOR());
        uint256 claimable = (userShare * epochRewards[epoch]) /
           scalingFactor;

        if (claimable > maxClaimable) {
            claimable = maxClaimable;
        }

        totalClaimable += claimable;
    }

    return totalClaimable;
}
```

Recommendation

There are two possible solutions which should be considered:

1. The first solution is quite straightforward, consider changing the current type of `uint8 public currentEpoch` to `uint256 public currentEpoch`, however, bear in mind the potential gas costs for future users. For instance, if the current epoch represents a significant timeframe, e.g. `10 years = 3650 epochs` or `20 years = 7300 epochs`, gas expenses for new users could become quite high. This occurs because initially, a new user's `lastClaimedForEpoch[user]` value would be 0, causing the for loop to iterate from 0 to the current epoch (e.g., 3650 or 7300), resulting in considerable gas consumption.
2. Another solution with a higher complexity would be refactoring the code logic so there is a single mapping `address => uint256` being updated when a user has accrued any rewards without recording the epoch in the mapping.

Team Response

Fixed with the first suggestion.

[H-01] Re-staking Mechanism Can Lead to Contract Insolvency

Severity

High Risk

Description

In the `DiscreteStakingRewards` contract, the staking rewards are paid in `BERA` (native as a token on Berachain) while the locked staking tokens are `YEET`.

However, the current re-staking mechanism allows to increase the staked `YEET` with `BERA` instead of claiming it as a reward.

As a consequence of re-staking, the `totalSupply` of `YEET` tokens in the contract will be higher than the actual supply which can potentially lead to insolvency of the contract in terms of `YEET` tokens.

Furthermore, treating `BERA` equivalently to `YEET` on re-staking comes with an economic disbalance since their respective prices are likely not equal at all times.

Location of Affected Code

File: src/Stake.sol#L157

```
/// @notice The function used to re-stake the rewards without claiming
   them, bypassing the vesting period
/// @dev The rewards are staked again
function reStake() external {
    _updateRewards(msg.sender);
    uint reward = earned[msg.sender];
    require(reward > 0, "No rewards to claim");

    earned[msg.sender] = 0;
    this.stake(reward);
}
```

Recommendation

`YEET` tokens should be bought with the `BERA` rewards before re-staking to prevent misaccounting of `BERA` as `YEET` and resolve the economic disbalance.

Team Response

Fixed by temporarily removing the re-staking functionality that will be re-designed.

[H-02] Yeet Functionality Flaw Upon Contract Deployment or Restart Due to Wrong Calculation

Severity

High Risk

Description

After deploying the Yeet contract or executing the `restart()` function, the `_potToWinner` value remains 0 until either the `receive()` or `fallback()` function is invoked. Meanwhile, the `yeet()` function is payable and aims to enforce a minimum amount of Yeet points to be passed, calculated through the `_minimumYeetPoint()` function. This calculation involves dividing the `totalPot` by `POT_DIVISION` and it will yield 0 due to the absence of funds in the pot.

Consequently, a user could successfully execute `yeet()` with 0 wei, bypassing the intended minimum Yeet point requirement.

Due to each game round's initial `minimumYeetPoint` of 0, an adversary can `yeet` an arbitrary amount of times with `0 <= msg.value < POT_DIVISION` and thereby gain an unfair advantage over other participants concerning the `Yeetback` winnings distribution. This is possible since the adversary will be added again to the `yeetsInRound` array of the `Yeetback` contract on each `yeet` and therefore increase their probability of being among the 10 winners during the final draw.

Location of Affected Code

File: src/Yeet.sol#L154

```
/// @notice yeet is the main function of the game, users yeet the native
    token in the pot
/// @notice The pot is divided into a winner pot and a yeetback pot
function yeet() external payable {
     uint timestamp = block.timestamp;
     require(timestamp < _endOfYeetTime, "Yeet: Yeet time has passed, game
         is over");

     uint256 minimumYeetPoint = _minimumYeetPoint(_potToWinner);
     require(msg.value >= minimumYeetPoint, "Yeet: Yeet not big enough");

// code
}
```

Recommendation

Consider re-designing the `_minimumYeetPoint()` function logic to handle the case when `totalPot` and still return a feasible minimum Yeet points amount, for example, require a minimum of `msg.value >= POT_DIVISION` on each `yeet` and therefore also resolve the integer division issue in `_minimumYeetPoint`.

Team Response

Fixed.

[M-01] Missing Vesting Period Verification in `unstake()` Method Puts Users at Risk

Severity

Medium Risk

Description

The `DiscreteStakingRewards` contract has a `rageQuit()` method which allows users to unstake before the vesting period has ended. Thereby, the user is at a (partial) loss which is intended and expected on a "rage quit".
However, the `unstake()` method, which can be reasonably assumed to only allow unstaking once the vesting period has ended, is equivalent to the `rageQuit()` method and neglects to check the vesting period.

Consequently, users are at risk of having their stake (partially) burnt when calling `unstake()` which is not expected and therefore cannot be considered a careless user error.

```
/// @notice The function used to unstake tokens
/// @param index The index of the vesting to unstake
function unstake(uint256 index) external {
    _unstake(index);
}

/// @notice The function used to rage quit
/// @notice Rage quit is used to unstake before the vesting period ends,
   will be called in the dApp
/// @param index The index of the vesting to unstake
function rageQuit(uint256 index) external {
    _unstake(index);
}
```

Location of Affected Code

File: src/Stake.sol#L101

```
/// @notice The function used to unstake tokens
/// @param index The index of the vesting to unstake
function unstake(uint256 index) external {
    _unstake(index);
}
```

Recommendation

Consider adding a vesting period check:

```
function unstake(uint256 index) external {
+   require(block.timestamp >= vestings[msg.sender][index].end, "Vesting
   period has not ended");
    _unstake(index);
}
```

Team Response

Fixed as suggested.

[M-02] Significantly Decreased Entropy in `draftWinners()` Method

Severity

Medium Risk

Description

In the `draftWinners` method of the `Yeet` contract, the initial entropy `randomNumber` is meant to be split 10 times into `smallNumbers` which are subsequently hashed to `randomData`. This `randomData` is meant to provide randomness for determining the winner in each iteration. However, it's currently unused and `randomNumber` is directly used without hashing.

Consequently, the entropy for drafting 10 winners is significantly reduced compared to the intended implementation. Currently, the only measure preventing all 10 winners from being the same is that the `randomNumber` is downshifted by 16 bits in each iteration, i.e. randomNumber »= 16.

Location of Affected Code

File: src/Yeetback.sol#L67

```
/// @notice Drafts the winners for the round using the random number
   generated by the entropy contract
/// @dev The random number is used to select the winners from the yeets
   in the round
/// @dev We don't want to get 10 random numbers from the entropy contract
    because it's expensive, so we use a single random number and seed it
   with some entropy
/// @dev We hash the smaller number so that its distribution is more
   uniform
function draftWinners(uint256 randomNumber, uint256 round) private {
// code

// Seed a new random number with some entropy from the large random
   number
    bytes32 randomData = keccak256(abi.encodePacked(smallNumbers));
    uint randomDataNumber = uint(randomNumber);

// code
}
```

Recommendation

Use `randomNumber` instead of `randomData` in `Yeet`:

```
// Seed a new random number with some entropy from the large random
   number
  bytes32 randomData = keccak256(abi.encodePacked(smallNumbers));
- uint randomDataNumber = uint(randomNumber);
+ uint randomDataNumber = uint(randomData);
```

Team Response

Fixed as suggested.

[M-03] Dysfunctional Instantiation of the `Yeetback.sol` Contract

Severity

Medium Risk

Description

The `Yeetback` contract is expected to be constructed with an `entropy` contract, which implements the IEntropy interface, and an `entropyProvider` contract as follows:

```
constructor(
    address _entropy,
    address _entropyProvider
) Ownable(msg.sender) {
    entropy = IEntropy(_entropy);
    entropyProvider = _entropyProvider;
}
```

However, the `Yeet` contract itself (`address(this)`) is provided on construction which does not implement any of the `IEntropy` methods required by the **Yeetback** contract, nor does it serve with any `entropyProvider` functionality.

As a consequence, core methods, i.e. `restart` , `addYeetback` & `getEntropyFee` , of the `Yeet` and `Yeetback` contracts will always fail to work as expected.

Location of Affected Code

File: src/Yeet.sol#L142

```
constructor(
    YeetToken _token,
    Reward _reward,
    DiscreteStakingRewards _staking,
    YeetGameSettings _gameSettings,
    address _publicGoodsAddress,
    address _lpStakingAddress,
    address _teamAddress
) Pauseable(msg.sender) {
// code

    yeetback = new Yeetback(
        address(this),
        address(this)
    );

// code
```

Recommendation

Provide the correct `entropy` and `entropyProvider` contract addresses according to the Pyth Network contract documentation.

Team Response

Fixed as suggested.

[M-04] Incomplete Validation

Severity

Medium Risk

Description

In the `NFTVesting.sol` contract the following validation checks are missing:

- In `constructor()` if `_startTime` is set in the past the `getTimePassed()` function will revert on every call and there `claim()` will be unusable. The same applies for `_vestingPeriod` and `_tokenAmount` to be greater than 0.
- In `claim()` function check whether the vesting period has started and whether the vesting period has ended. A zero-value check for `claimable` should be applied as well.

In the `Stake.sol` contract the following validation checks are missing:

- `stake()` and `startUnstake()` functions are missing zero-value checks.

In the `OnlyYeetContract.sol` contract the following validation checks are missing:

- `setYeetContract()` function missing zero address check for `_yeetContract` parameter.

In the `Reward.sol` contract the following validation checks are missing:

- `addYeetVolume()` function missing zero address check for `user` parameter.
- In the `addYeetVolume()` function there is a `require` statement which must not allow zero amount to be sent for the tx but the comparison allows zero value to pass.

In the `Yeet.sol` contract the following validation checks are missing:

- `constructor()` missing zero address check for `_publicGoodsAddress`, `_lpStakingAddress` and `_teamAddress`.
- `restart()` function missing empty bytes check for `userRandomNumber` parameter.
- `setPublicGoodsAddress()`, `setLpStakingAddress()`, `setTreasuryRevenueAddress()`, `setYeetardsNFTsAddress()` functions are missing zero address checks.

In the `Yeetback.sol` contract the following validation checks are missing:

- `constructor()` missing zero address check for `_entropy` and `_entropyProvider` parameters.
- `addYeetsInRound()` function missing zero-value check for `round` and zero address check for `user` parameters.
- `addYeetback()` function missing empty bytes check for `userRandomNumber` and zero-value checks for `round` and `amount` parameters.
- `claim()` function zero-value check for the `round` parameter.

Location of Affected Code

File: src/NFTVesting.sol

File: src/Stake.sol

File: src/OnlyYeetContract.sol

File: src/Reward.sol

File: src/Yeet.sol

File: src/Yeetback.sol

Recommendation

Implement the following validation checks:

- File: `NFTVesting.sol`

```
constructor(IERC20 _token, INFTContract _nftContract, uint256
    _tokenAmount, uint256 _nftAmount, uint256 _startTime, uint256
    _vestingPeriod) {
+   require(_startTime >= block.timestamp, "NFTVesting: startTime should
    be in the future");
+   require(_vestingPeriod != 0, "NFTVesting: vestingPeriod should be
    larger than 0");
+   require(_tokenAmount != 0, "NFTVesting: _tokenAmount should be larger
    than 0");
}

function claim(uint256 tokenId) public {
+   require(claimable != 0, "Nothing to claim");
}
```

- File: `Stake.sol`

```
function stake(uint256 amount) external {
+    require(amount != 0, "Invalid stake amount");
}

function startUnstake(uint256 unStakeAmount) external {
+    require(unStakeAmount != 0, "Invalid unstake amount");
}
```

- File: `OnlyYeetContract.sol`

```
function setYeetContract(address _yeetContract) external onlyOwner {
+    require(_yeetContract != address(0), "Invalid yeet contract address")
    ;
}
```

- File: `Reward.sol`

```
function addYeetVolume(address user, uint256 amount) external
   onlyYeetOwner {
-   require(amount >= 0, "Amount must be greater than 0");
+   require(amount != 0, "Amount must be greater than 0");
+   require(user != address(0), "Invalid user address");
}
```

- File: `Yeet.sol`

```
constructor(YeetToken _token, Reward _reward, DiscreteStakingRewards
   _staking, YeetGameSettings _gameSettings, address _publicGoodsAddress,
    address _lpStakingAddress, address _teamAddress) Pauseable(msg.sender
   ) {
+   require(_publicGoodsAddress != address(0), "Invalid public goods
   address");
+   require(_lpStakingAddress != address(0), "Invalid lp staking address
   ");
+   require(_teamAddress != address(0), "Invalid team address");
}

function restart(bytes32 userRandomNumber) payable external whenNotPaused
    {
+   require(userRandomNumber != bytes32(0), "Invalid number");
}

function setPublicGoodsAddress(address _publicGoodsAddress) external
   onlyOwner {
+   require(_publicGoodsAddress != address(0), "Invalid public goods
   address");
}

function setLpStakingAddress(address _lpStakingAddress) external
   onlyOwner {
+   require(_lpStakingAddress != address(0), "Invalid lp staking address
   ");
}

function setTreasuryRevenueAddress(address _treasuryRevenueAddress)
   external onlyOwner {
+   require(_treasuryRevenueAddress != address(0), "Invalid treasury
   revenue address");
}

function setYeetardsNFTsAddress(address _yeetardsNFTsAddress) external
   onlyOwner {
+   require(_yeetardsNFTsAddress != address(0), "Invalid NFTs contract
   address");
}
```

- File: `Yeetback.sol`

```
constructor(address _entropy, address _entropyProvider) Ownable(msg.
    sender) {
+   require(_entropy != address(0), "Invalid entropy address");
+   require(_entropyProvider != address(0), "Invalid entropy provider
    address");
}

function addYeetsInRound(uint256 round, address user) public onlyOwner {
+   require(round != 0, "Invalid round");
+   require(user != address(0), "Invalid user address");
}

function addYeetback(bytes32 userRandomNumber, uint256 round, uint256
    amount) payable public onlyOwner {
+   require(userRandomNumber != bytes32(0), "Invalid number");
+   require(round != 0, "Invalid round");
+   require(amount != 0, "Invalid amount");
}

function claim(uint256 round) public {
+   require(round != 0, "Invalid round");
}
```

Team Response

Fixed as suggested.

[L-01] The `latestHistoryArr` Storage Array Is Ever Growing

Severity

Low Risk

Description

The `latestHistoryArr` storage array is intended to only record the last `LATEST_HISTORY_LENGTH` yeets, see code below:

```
if (latestHistoryArr.length >= LATEST_HISTORY_LENGTH) {
    delete latestHistoryArr[0];
}
```

```
latestHistoryArr.push(History(
    msg.sender,
    timestamp,
    _potToWinner - valueToPot,
    _potToWinner,
    _yeetTimeInSeconds,
    _minimumYeetPoint(_potToWinner),
    _nrOfYeets,
    roundNumber
));
```

However, the `latestHistoryArr` array is ever-growing since the `delete latestHistoryArr[0]` statement only clears the storage slot at array index 0, but does not reduce the array's length. Therefore, the `latestHistoryArr` array effectively contains the whole history except for the very first entry.

Location of Affected Code

File: src/Yeet.sol#L211-L213

```
/// @notice yeet is the main function of the game, users yeet the native
   token in the pot
/// @notice The pot is divided into a winner pot and a yeetback pot
function yeet() external payable {
// code

    if (latestHistoryArr.length >= LATEST_HISTORY_LENGTH) {
        delete latestHistoryArr[0];
    }

// code
}
```

Recommendation

For the purpose of keeping the last `LATEST_HISTORY_LENGTH` entries, it is recommended to implement a ring buffer mechanism, which is especially favourable since it avoids unnecessary storage/-gas overhead because the relocation of other array elements is not needed.

Team Response

Fixed.

[L-02] `call()` Should Be Used Instead of `transfer()` on an `address payable`

Severity

Low Risk

Description

The `transfer()` and `send()` functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase in the SLOAD instruction.

Location of Affected Code

File: src/Stake.sol#L173 File: src/Yeet.sol File: src/Yeetback.sol#L124

Scenario

Additionally, the use of the deprecated `transfer()` function for an address will inevitably make the transaction fail when:

- The claimer smart contract does not implement a payable function.
- The claimer smart contract does implement a payable fallback which uses more than 2300 gas units.
- The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through a proxy, raising the call's gas usage above 2300.
- Additionally, using more than 2300 gas might be mandatory for some multisig wallets.

Recommendation

Use `call()` instead of `transfer()`, but be sure to respect the CEI pattern and/or add re-entrance guards, as several hacks already happened in the past due to this recommendation not being fully understood.

Team Response

Fixed as suggested.

[L-03] `yeet()` Function Should Be Pauseable

Severity

Low Risk

Description

The `yeet()` function is missing to implement the `whenNotPaused` modifier. Even though this addition would add centralization it could be useful in case of an emergency, to prevent further damage or loss of funds by temporarily halting all executions.

Location of Affected Code

File: src/Yeet.sol#L154

Recommendation

Consider adding the `whenNotPaused` to the `yeet()` function.

Team Response

Acknowledged.

[L-04] Potentially Stuck Funds

Severity

Low Risk

Description

The users interacting with the protocol are able to call the `depositReward()` function by mistake and lose their funds forever. The function is specifically designated for use by the protocol team or sponsors for supplementing rewards or making donations. Restricting access to this function aligns with best practices in UX, ensuring the prevention of potential cases where users encounter locked funds.

Location of Affected Code

File: src/Stake.sol#L50

Recommendation

Consider allowing only a whitelisted set of addresses to be able to call the `depositReward()` function.

Team Response

Acknowledged.

[L-05] Wrong Event Value Causes Confusion

Severity

Low Risk

Description

The `Reward._endEpoch()` function is meant to end an epoch if the current `block.timestamp` is greater than `epochEnd`, set the starting of a new epoch and calculate the rewards. The problem occurs when the `EpochEnded` event is emitted.

In this context, it's vital for the event to emit both the epoch that just concluded and its corresponding timestamp. However, a challenge arises because the epoch parameter passed to the event undergoes incrementation just before emission. Consequently, the event emits the upcoming epoch rather than the one that has ended, potentially causing confusion for users and contracts relying on this information.

Location of Affected Code

File: src/Reward.sol#L110

Recommendation

Consider emitting the event before setting the start of the new epoch.

Team Response

Fixed as suggested.

[L-06] Avoid Drafting the Same Winner Multiple Times by the Increasing Pseudo-Randomness in `Yeetback.sol`

Severity

Low Risk

Description

In the `draftWinners()` method, 10 winners are drafted in a loop based on `randomData` which is the `keccak256` hash of `smallNumbers`. Each iteration's `smallNumbers` are derived from the base entropy given by `randomNumber` and could potentially be the same in multiple iterations leading to drafting the same winner multiple times.

Location of Affected Code

File: src/Yeetback.sol#L66

```
bytes32 randomData = keccak256(abi.encodePacked(smallNumbers));
```

Recommendation

In order to increase the pseudo-randomness based on the entropy given by `randomNumber`, it is recommended to hash further data like the loop counter `i` and the `block.timestamp`.

```
- bytes32 randomData = keccak256(abi.encodePacked(smallNumbers));
+ bytes32 randomData = keccak256(abi.encodePacked(smallNumbers, i, block.
   timestamp));
```

Team Response

Fixed as suggested.

our shielding · Your smart contracts, our shielding · Your smart

shieldify

Thank you!