Bountive Security Review

Reviewers
m4k2
Greed
Mikb

November 8, 2024

1 Executive Summary

Over the course of 14 days in total, Bountive engaged with Beyond for a security review.

We found a total of 14 issues with Bountive.

Summary

Type of Project	Raffle Protocol
Timeline	Oct 12, 2024 - Oct 28, 2024
Methods	Manual Review

Total Issues

Critical Risk	0
High Risk	1
Medium Risk	7
Low Risk	6
Informational	2

Contents

2 Beyond

Beyond is a team of Web3 security engineers unified by a common objective: to ensure the continued security of blockchain technology. Our collective expertise encompasses the full spectrum of Web3 technologies, from protocol architecture to smart contract implementation. We specialize in comprehensive security assessments across diverse blockchain environments, with expertise in Solidity and EVM-based protocols, Rust, and Cairo. Our assessments combine a rigorous methodology with practical insights, leveraging our expertise to provide thorough coverage of potential vulnerabilities while maintaining the highest standards of quality.

3 Introduction

Bountive is a decentralized, non-custodial prize savings protocol on Starknet, empowering users with no-loss prize games.

The protocol is based on the proven concept of Prize-linked savings accounts from traditional finance, which is used by banks in the United States to help boost savings.

Key features of the protocol include:

- User-Friendliness: Seamless and effortless participation experience for users of all backgrounds
- No Risk Of Loss: Users can participate in prize games without risking their initial investment
- Non-Custodial: Users maintain full ownership and control over their funds at all times
- Instant Fund Access: Funds can be withdrawn at any time, providing ultimate control over assets

Disclaimer: Please note that this security review does not guarantee against a hack. It represents a snapshot in time of the code, as reviewed by our team. Any modifications to the code will require a new security review.

4 Scope of the audit

The audit was conducted on the following commit hash: 46631c7a723ab792d21d8ef4f1e51d16d0e84c7c

```
components
   oracle.cairo
   raffle.cairo
core
   ducks_raffle.cairo
   factory.cairo
   nostra_raffle.cairo
   raffle.cairo
   reserve.cairo
   sbt.cairo
   vesu_raffle.cairo
   winner_manager.cairo
   zklend_raffle.cairo
lib.cairo
tokens
   erc20_mintable_burnable.cairo
utils
    constants.cairo
    errors.cairo
    storage_access.cairo
```

5 Findings classification

In order to fairly evaluate the severity of an issue, it is correlated with the following components:

- the `likelihood/impact` matrix
- the core issue definition
- the protocol's design and functionality

It is assumed that privileged parties (admins, bots...) will act for the benefit of the protocol and not deliberately perpetrate any malicious actions against it.

Moreover, it is the protocol's responsability to securely manage their accounts to prevent eventual compromissions.

Thus, restricted functions requiring privileges will be assumed to be called with legitimate values.

5.1 Likelihood and Impact matrix

The following matrix facilitates the evaluation of an issue based on its likelihood and impact.

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

5.2 Core issue definition

This section details the severity of an issue based upon its damage for the protocol.

It is broken down into :

- a brief introduction of the issue's damage at its core
- non-exhaustive examples of such issue

Critical

- Empty or freeze the contract's holdings with no pre-condition, very low capital investment and no time constraint.
- Disrupt the system with impossibility to recover.

  Examples:

  - Infinitely repeatable risk-free trades
  - Draining the system capital with profit
  - Continuous Denial Of Service (DoS) against the protocol's essential functionalities
  - Prevent users from withdrawing their funds or interacting with them

High

- Considerable amount of funds are lost.
- System can be disrupted with difficulty to recover.

  Examples:

  - User position can be driven towards liquidation by a malicious actor
  - Functionality is disrupted for more than an hour
  - Protocol's fundamental functionality is flawed

    - Unauthorized party can act on behalf of another user

Medium

- Funds are lost under certain conditions.
- System can be temporarily disrupted but can recover.
- Functionality is missing or incorrectly implemented.
- Non-compliance with standards meaningful for the protocol.

  Examples:

  - Lack of slippage
  - Fee calculations are flawed resulting in users/protocol losing a relatively low percentage of their funds
  - Admin-restricted transaction damages a user's position
  - Functionality is disrupted for less than an hour
  - System not compliant with EIP-712

Low

- Funds are not at risk but the handling of the state might be incorrect or not handled appropriately.

  Examples:

  - User mistakes that could have been prevent
  - Issues that may occur in a future implementation of the protocol
  - Weird token behaviors (fee-on-transfer, rebase, blacklist...) unless strong will for the protocol to integrate them
  - Incorrect event emission unless the system's functionalities heavily depend on off-chain listeners.

Informational

- Insights on design choices that do not represent a security risk.

  Examples:

  - Code optimization
  - Irrelevant checks
  - Unreachable code that do not break the system

5.3 Protocol's design and functionality

Every protocol is designed to provide its own set of functionalities and services.

Thus, the threat model that corresponds to a particular protocol might differ from another. In this manner, the same issue can be considered damageful for a particular protocol and irrelevant for another.

This disparity is taken into account when defining the severity of an issue, making it fit the protocol's specific design.

6 Findings

6.1 High Risk

6.1.1 Incorrect weight calculation in `reduce_user_weight` function leads to potential underflow

Severity: High

Context: `raffle.cairo#L694-L723`

Description:

The `reduce_user_weight()` function in the `RaffleComponent` is designed to decrease a user's weight in the raffle proportionally to the value of their withdrawal. However, the current implementation has a critical flaw in its weight calculation logic.

The function calculates the weight to be reduced based on the multiplier associated with the user's first deposit timestamp. This approach fails to account for scenarios where a user has made multiple deposits over time, each with potentially different multipliers based on the day of deposit.

The core issue lies in the following code snippet from `reduce_user_weight()`:

```
fn reduce_user_weight(
            ...
        ) {
            ...
                let opening_time = self.raffle_opening_time.read();
                let user_fist_deposit = self.raffle_user_first_deposit.entry(caller_address).read();
                let elapsed_time = user_fist_deposit - opening_time;
                let index = elapsed_time / 86400;

            ...

                let multiplier: u256 = index_multiplier.into();
@>              let weight = amount * multiplier;

            ...
            }
        }
```

This calculation always uses the multiplier from the first deposit, regardless of when subsequent deposits were made. As a result, the weight being subtracted can be larger than the user's actual weight, leading to an underflow condition.

Proof of Concept:

The technical details and demonstration of this vulnerability are provided in the appendix. The test performs the following steps (note that this explanation doesn't take into account the fees on deposit and withdrawal, which doesn't change the outcome of the vulnerability):

1. User deposits 1000 tokens on day 1 with a multiplier of 7, resulting in a weight of 7000.
2. User deposits another 1000 tokens on day 2 with a multiplier of 6, increasing the total weight to 13000.
3. User attempts to withdraw all shares, which triggers the `reduce_user_weight()` function.
4. The function incorrectly calculates the weight to reduce as 2000 * 7 = 14000, which is greater than the actual total weight of 13000.
5. This leads to an underflow when subtracting the calculated weight from the user's actual weight, causing the function to revert.

You will find the technical PoC in Appendix.

Recommendation:

To address this issue, the `reduce_user_weight()` function should be redesigned to accurately track and calculate user weights based on multiple deposits with varying multipliers. Here are some potential approaches:

1. Maintain a detailed record of each deposit, including its amount and associated multiplier.
2. When reducing weight, calculate the proportion of each deposit being withdrawn and apply the correct multiplier for each portion.

This approach ensures that the weight reduction is calculated accurately based on the specific multipliers used for each deposit, preventing potential underflows and maintaining the integrity of the raffle system.

Project: Fixed. This has had no impact on the Raffle draws, as there is verification in the backend to recalculate the weights and check their integrity.

Beyond: Acknowledged.

6.2 Medium Risk

6.2.1 Incorrect price conversion for stablecoins leads to significant undervaluation

Severity: Medium

Context: `oracle.cairo#L61-L73`

Description:

The `convert_to_usd()` function in the Oracle component is responsible for converting asset amounts to their USD equivalent. However, the current implementation contains a flaw that results in significant undervaluation for certain stablecoins, particularly USDT and USDC.

The root cause of this issue lies in the hardcoded division by `10^8` at the end of the `convert_to_usd()` function:

```
fn convert_to_usd(
    self: @ComponentState<TContractState>,
    amount: u256,
    unit_price: u128,
    offchain_decimals: u32,
    current_decimals: u8
) -> u256 {
    let converted_amount = amount
        / pow(10, current_decimals.into() - offchain_decimals.into());
    let total_price = converted_amount * unit_price.into();
@>  total_price / pow(10, 8)
}
```

This hardcoded division assumes that all assets have 8 decimal places for their price feed. However, according to the Pragma documentation, some stablecoins, including USDT and USDC, use 6 decimal places instead of 8.

As a result, when converting USDT or USDC to USD, the function returns a price that is 100 times lower than it should be. This undervaluation has severe implications for various critical functions that rely on `estimate_fees()` which relies on `convert_to_usd()`, including `withdraw()`, `deposit()`, `refund()`, and `claim_winning_prize()`.

The impact of this issue is a direct loss of funds for users. The contract consistently underestimates amount, thus overestimating fees calculated as `SHRIMP_BASIS_POINTS` (0.14%) for these stablecoins, leading to significant discrepancies in fee collection.

Recommendation:

To address this issue, the `convert_to_usd()` function should use the `offchain_decimals` parameter instead of the hardcoded value. Additionally, the `estimate_fees()` function should be updated to use the correct number of decimals.

Here's the recommended fix:

```
fn convert_to_usd(
    self: @ComponentState<TContractState>,
    amount: u256,
    unit_price: u128,
    offchain_decimals: u32,
    current_decimals: u8
) -> u256 {
     let converted_amount = amount
        / pow(10, current_decimals.into() - offchain_decimals.into());
     let total_price = converted_amount * unit_price.into();
-    total_price / pow(10, 8)
+    total_price / pow(10, offchain_decimals)
}

// In the estimate_fees function:
- ... / pow(10, 8)
+ ... / pow(10, output.decimals)
```

These changes ensure that the conversion and fee calculations are accurate for all supported assets, regardless of their decimal representation in the price feed.

Project: Fixed.

Beyond: Acknowledged.

6.2.2 NFT ownership check vulnerability allows bypassing raffle participation restrictions

Severity: Medium

Context: `ducks_raffle.cairo#L206-L240`, `raffle.cairo#L628-L637`

Description:

The vulnerability exists in both the `DucksRaffle` contract and the `RaffleComponent`. In the `RaffleComponent`, the `deposit()` function also performs a collection check.

The `DucksRaffle` contract implements a raffle system where participation is restricted to holders of specific NFTs (Ducks Everywhere or Ducks Frens). This mechanism is intended to limit raffle entries to a selected group of users.

The current implementation checks NFT ownership only at the time of deposit, creating a vulnerability where users can circumvent the participation restriction. A malicious actor could exploit this by using a flash loan mechanism to borrow an eligible NFT, participate in the raffle, and return the NFT, all within a single atomic transaction. This process could involve:

1. Initiating a flash loan to borrow an eligible NFT
2. Depositing funds into the raffle
3. Returning the borrowed NFT
4. Repeating the process with multiple accounts or in multiple transactions

This allows a single NFT to be used to enter the raffle multiple times without the risk of holding the NFT between transactions, effectively bypassing the intended participation restriction. The atomic nature of the transaction means the attacker never actually holds the NFT, making the exploit even more accessible and less risky.

The root cause of this vulnerability lies in the `provide_liquidity()` function, which only checks NFT ownership at the time of deposit:

```
fn provide_liquidity(
     ref self: RaffleComponent::ComponentState<ContractState>, assets: u256
) {
    let caller_address = get_caller_address();
    let contract_state = RaffleComponent::HasComponent::get_contract(@self);
    if caller_address != self.raffle_reserve_contract.read() {
        let is_ducks_frens_holder = DucksRaffleHelper::is_ducks_frens_holder(
            contract_state, caller_address
        );
        let is_ducks_everywhere_holder = DucksRaffleHelper::is_ducks_everywhere_holder(
                contract_state, caller_address
            );
        assert(
            is_ducks_frens_holder || is_ducks_everywhere_holder,
            Errors::NOT_COLLECTION_HOLDER
        );
        DucksRaffleHelper::supply(contract_state, assets);
    } else {
        DucksRaffleHelper::non_player_supply(contract_state, assets);
    };
}
```

This vulnerability undermines the fairness and integrity of the raffle system, potentially allowing a small group of users to dominate the participant pool.

Recommendation:

To mitigate this vulnerability, consider implementing one or more of the following solutions:

1. *BEST* Consider locking the NFTs in the raffle contract for the duration of participation. This would prevent transfers and ensure continued ownership throughout the raffle:

```
fn deposit(ref self: ContractState, assets: u256) {
    // ... existing checks ...
    let nft_contract = IERC721Dispatcher { contract_address: self.nft_address.read() };
    nft_contract.transferFrom(get_caller_address(), get_contract_address(), token_id);
    // ... proceed with deposit ...
}

fn withdraw(ref self: ContractState, shares: u256) {
    // ... existing logic ...
    let nft_contract = IERC721Dispatcher { contract_address: self.nft_address.read() };
    nft_contract.transferFrom(get_contract_address(), get_caller_address(), token_id);
    // ... proceed with withdrawal ...
}
```

2. Check NFT ownership on every interaction with the raffle, not just at deposit time:

```
fn check_nft_ownership(ref self: ContractState, user: ContractAddress) {
    let is_ducks_frens_holder = DucksRaffleHelper::is_ducks_frens_holder(self, user);
    let is_ducks_everywhere_holder = DucksRaffleHelper::is_ducks_everywhere_holder(self, user);
    assert(
        is_ducks_frens_holder || is_ducks_everywhere_holder,
        Errors::NOT_COLLECTION_HOLDER
    );
}
```

Call this function at the beginning of every user interaction (deposit, withdraw, claim, etc.).

For Ducks Frens NFTs, implementing a mapping to track used NFT IDs would be complex due to the ERC1155 nature of the collection, where multiple tokens of the same ID exist. This approach could help prevent flash loan attacks, as the nft flash loaned for the deposit could be sold the time the hacker want to withdraw.

Note that for Ducks Everywhere NFTs, option 1 (locking the NFTs) would be the most effective solution to prevent the vulnerability.

Project: Fixed

Beyond: Acknowledged.

6.2.3 Incorrect weight reduction logic

Severity: Medium

Context: `raffle.cairo#L694-L723`

Description:

The `RaffleComponent` contract implements a raffle system where users can deposit assets and receive shares, with their chances of winning being determined by a weight system. The weight calculation takes into account both the amount deposited and the timing of deposits.

When users withdraw their shares, the contract attempts to reduce their weight accordingly through the `reduce_-user_weight()` function. However, there is a wrong assumption in how this reduction is implemented.

The issue occurs in the following sequence:

1. In `withdraw()`, the user's shares are first reduced via `InternalImpl::withdraw()`
2. Then `reduce_user_weight()` is called, which reads the user's shares AFTER they've been reduced
3. The weight reduction logic checks `if user_shares == amount` to determine if this is a full withdrawal

Since the shares are already reduced when this check happens, `user_shares` will always be less than the withdrawal `amount`, causing the full withdrawal logic to be skipped. Instead, it executes the partial withdrawal logic.

Impact:

- The weight reduction logic does not work as intended
- The code fails to properly handle full withdrawals, leading to incorrect weight calculations
- The system cannot accurately track user participation weights as designed

Recommendation:

Modify the condition responsible for determining if the user attempts to perform a full withdrawal like such:

```
fn reduce_user_weight(
    ...
) {
    let user_shares = self.raffle_share_holder.entry(caller_address).read();
-   if user_shares == amount {
+   if user_shares == 0 {
        self.raffle_total_weight.write(
            self.raffle_total_weight.read() -
            self.raffle_user_weight.entry(caller_address).read()
        );
        self.raffle_user_weight.entry(caller_address).write(0);
        self.raffle_user_first_deposit.entry(caller_address).write(0);
    } else {
        // ... partial withdrawal logic
    }
}
```

Project: Fixed. This has had no impact on the Raffle draws, as there is verification in the backend to recalculate the weights and check their integrity.

Beyond: Acknowledged.

6.2.4 Last Ducks Frens ID check skipped in holder verification

Severity: Medium

Context: `ducks_raffle.cairo#L220-L240`

Description:

The `is_ducks_frens_holder()` function in the `DucksRaffle` contract is responsible for verifying if a given address holds any of the whitelisted Ducks Frens NFTs. This verification is crucial for determining eligibility for participation in the raffle system.

The function contains a logical error in its loop termination condition that causes it to skip checking the last ID in the `ducks_frens_ids` array. The current implementation breaks the loop when `idx` equals `ducks_frens_ids_len - 1`, which means the last element is never processed:

```
if (idx == ducks_frens_ids_len - 1) {
    break;
}
```

This premature loop termination could allow users who only hold the last NFT ID in the array to be incorrectly identified as non-holders, denying them legitimate access to the raffle system.

Impact:

- Users holding only the last Ducks Frens NFT ID in the array will be incorrectly identified as non-holders
- These users will be unable to participate in the raffle despite having legitimate holdings
- This could lead to user frustration and reduced participation in the protocol

Proof of Concept:

Consider a scenario where `ducks_frens_ids` contains `[1, 2, 3]`:

1. A user holding only ID 3 attempts to participate
2. The loop checks ID 1 (index 0), then ID 2 (index 1)
3. When `idx == 2`, the loop breaks before checking ID 3
4. The function returns `false` even though the user holds a valid NFT

Recommendation:

Modify the loop condition to ensure all IDs are checked. There are two possible solutions:

```diff
- if (idx == ducks_frens_ids_len - 1) {
+ if (idx == ducks_frens_ids_len) {
    break;
}
```

Or alternatively:

```diff
- if (idx == ducks_frens_ids_len - 1) {
+ if (idx > ducks_frens_ids_len - 1) {
    break;
}
```

Project: Fixed.

Beyond: Acknowledged.

6.2.5 Incorrect max_users check prevents participants from depositing

Severity: Medium

Context: `raffle.cairo#L296-L302`

Description:

The `deposit()` function in the `RaffleComponent` contains logic to enforce a maximum number of users that can participate in the raffle. However, once the maximum user limit is reached, participants are unable to increase their position through additional deposits.

The function first checks if the maximum users limit would be exceeded by adding a new participant:

```
if self.raffle_max_users.read() != 0 {
    assert(
        self.raffle_max_users.read() >= self.raffle_participant_count.read()
            + 1_u256,
        Errors::MAX_USERS_EXCEEDED
    );
};
```

Then it checks if the caller is a new participant and increments the counter if so:

```
if (self.raffle_share_holder.entry(caller_address).read() == 0) {
    self.raffle_participant_count.write(self.raffle_participant_count.read() + 1);
}
```

The issue is that the "max users" check is performed unconditionally, even for existing participants who are just adding to their position. This means that once the maximum number of users is reached, existing participants cannot increase their deposits because the check will always revert, even though they wouldn't actually increase the participant count.

This creates an unfair restriction where early participants are blocked from increasing their position once the "max users" limit is reached, even though allowing them to do so would not violate the intended constraint of limiting unique participants.

Proof of Concept:

1. Raffle is created with `raffle_max_users = 100`
2. Alice deposits 1 STRK
3. 99 other users deposit, reaching the max users limit
4. Alice tries to deposit more STRK to increase her position
5. The transaction reverts due to the max users check, even though Alice is an existing participant

Recommendation:

The max users check should only be performed for new participants. Move the check inside the condition that handles new participants:

```
+   if (self.raffle_share_holder.entry(caller_address).read() == 0) {
        if self.raffle_max_users.read() != 0 {
            assert(
                self.raffle_max_users.read() >= self.raffle_participant_count.read()
                    + 1_u256,
                Errors::MAX_USERS_EXCEEDED
            );
        };
        self.raffle_participant_count.write(self.raffle_participant_count.read() + 1);
+   }
```

This ensures that existing participants can still increase their position while maintaining the limit on unique participants.

Project: Fixed.

Beyond: Acknowledged.

6.2.6 Missing Defi Spring claim functionality in Nostra and Vesu raffles

Severity: Medium

Context: `vesu_raffle.cairo`, `nostra.cairo`

Description:

The ZkLend and Ducks raffle contracts implement functionality to claim Defi Spring rewards through their respective `claim_defi_spring()` functions, but this capability is missing from the Vesu and Nostra raffle implementations.

For example, in `zklend_raffle.cairo`, the claim functionality is implemented as:

```
fn claim_defi_spring(
    ref self: ContractState,
    reward_contract_address: ContractAddress,
    claim_id: u64,
    claim_claimee: ContractAddress,
    claim_amount: u128,
    proof: Span<felt252>
) {
    self.ownable.assert_only_owner();
    self.claim(reward_contract_address, claim_id, claim_claimee, claim_amount, proof);
    self.emit(
        Event::DefiSpringRewardClaimed(
            DefiSpringRewardClaimed {
                recipient: self.ownable.owner(),
                amount: claim_amount
            }
        )
    );
}
```

The absence of this functionality in Vesu and Nostra raffles means that any Defi Spring rewards generated through these protocols cannot be claimed, leading to lost rewards.

Recommendation:

Add similar claim functionality to both Vesu and Nostra raffle contracts. The implementation should follow the same pattern as seen in ZkLend and Ducks raffles.

Also ensure to:

1. Add the necessary event definitions
2. Implement the supporting `claim()` helper function
3. Update the respective interfaces to include the new functionality

This will ensure consistent reward claiming capabilities across all raffle types.

Project: We are still talking to the Vesu team about endpoints for retrieving data. At the moment, there's no implementation for retrieving DeFi Spring from a Vesu Raffle contract but it can be retrieved by other means.

Beyond: Acknowledged.

6.2.7 Raffles may be subjected to Denial Of Service

Severity: Medium

Context: `raffle.cairo#L376-L385`

Description:

The `RaffleComponent::deposit()` function increments a participant count when a new user deposits into the raffle. There is a check to ensure this count doesn't exceed a maximum limit on raffle that depends on the `max_users` value:

```
if self.raffle_max_users.read() != 0 {
    assert(
        self.raffle_max_users.read() >= self.raffle_participant_count.read()
            + 1_u256,
        Errors::MAX_USERS_EXCEEDED
    );
};
if (self.raffle_share_holder.entry(caller_address).read() == 0) {
    self.raffle_participant_count.write(self.raffle_participant_count.read() + 1);
}
```

This implementation is vulnerable to a denial-of-service attack where an attacker can fill up the raffle with minimal deposits (1 wei) from multiple addresses, preventing legitimate users from participating.

While Bountive can increase the `max_users` limit, an attacker can continuously execute this strategy by monitoring the raffle and reacting to any changes.

Proof of Concept:

An attacker could:

1. Create multiple addresses
2. Deposit 1 wei from each address until the `raffle_max_users` limit is reached
3. Effectively block other users from participating

This attack remains profitable even with a non-zero minimum deposit, as the attacker can withdraw their deposits after the raffle ends.

Recommendation:

Several measures can be taken to mitigate this vulnerability:

1. Implement a dynamic weighting system that favors larger deposits and longer holding periods. This could make it unprofitable for an attacker to spread small deposits across multiple accounts.
2. Consider implementing a vetting process for participants or requiring them to hold a certain amount of a specific token (e.g., a governance token) to participate.

Project: Bountive systematically implements a minimum deposit on their raffles which drastically mitigates the issue.

Beyond: Acknowledged.

6.3 Low Risk

6.3.1 Withdrawal check bypass allows deposits below minimum threshold

Severity: Low

Context: `raffle.cairo#L282-L288`

Description:

The raffle contract implements a minimum deposit requirement through the `raffle_min_deposit` storage variable, which is enforced during the `deposit()` function if the value is not zero. However, this protection can be bypassed through the `withdraw()` function, which does not verify that the remaining balance stays above the minimum threshold.

When users deposit funds, the contract correctly validates that their total position (current balance + new deposit) meets the minimum deposit requirement:

```
if self.raffle_min_deposit.read() != 0 {
    assert(
        self.raffle_share_holder.entry(caller_address).read()
            + assets >= self.raffle_min_deposit.read(),
        Errors::MIN_DEPOSIT
    );
};
```

However, the `withdraw()` function only checks that:

1. The user has sufficient shares to withdraw
2. The withdrawal amount is greater than 0
3. The raffle is still active

This allows users to partially withdraw funds and maintain a position below `raffle_min_deposit`, effectively circumventing the minimum deposit requirement. This undermines the purpose of having a minimum deposit threshold in the first place.

Impact:

The minimum deposit requirement becomes ineffective as users can bypass it through partial withdrawals, potentially disrupting the raffle's economic model and security assumptions.

Proof of Concept:

1. User deposits 100 tokens when `raffle_min_deposit` is set to 100
2. User withdraws 99 tokens
3. User maintains a position of 1 token, well below the minimum requirement
4. Contract state becomes inconsistent with its design goals

Recommendation:

Modify the `withdraw()` function to validate that either:

1. The withdrawal amount equals the user's entire balance (full exit), or
2. The remaining balance after withdrawal stays above the minimum deposit threshold

```
fn withdraw(ref self: ComponentState<TContractState>, shares: u256) {
    // ... existing checks ...

    let caller_shares = self.raffle_share_holder.entry(caller_address).read();
    let remaining_shares = caller_shares - shares;

+   // Allow either full withdrawal or maintain minimum deposit
+   assert(
+       remaining_shares == 0
+           || remaining_shares >= self.raffle_min_deposit.read(),
+       Errors::BELOW_MIN_DEPOSIT
+   );

    // ... rest of the function ...
}
```

Project: Bountive wants to offer flexibility regarding user deposits. This means the finding is a design choice.

Beyond: Acknowledged.

6.3.2 Missing external `Ownable` functionality

Severity: Low

Context: `reserve.cairo`

Description:

The `reserve` contract integrates the `OwnableComponent` from Openzeppelin. This component allows to define an `owner` that has access to the restricted functions of the contract. One feature of the component is to transfer the ownership to another contract using `transfer_ownership()`. However, the `reserve` contract does not expose the `OwnableComponent` external interface which prevents the current owner to transfer ownership of the contract.

Recommendation:

Expose the external interface of the `OwnableComponent` by adding the following in `reserve.cairo`:

```
#[abi(embed_v0)]
impl OwnableImpl = OwnableComponent::OwnableImpl<ContractState>;
```

Project: Fixed.

Beyond: Acknowledged.

6.3.3 Lack of token transfer success check

Severity: Low

Context: `reserve.cairo`

Description:

The protocol has implemented a way too ensure token transfers have succeeded by comparing the recipient's balance before the transfer and after the transfer. However, some functions lack this verification which could lead to unexpected behaviors depending on the token implementation. Below are the relevant functions affected by the issue:

- `reserve.cairo::withdraw_deposited_capital()`
- `reserve.cairo::withdraw_assets()`
- `raffle.cairo::claim_yield_prize_multiple_winner()`

Recommendation:

Make sure the token transfers have succeeded by comparing the recipient's balance before and after the transfer in the affected functions.

Project: Fixed.

Beyond: Acknowledged.

6.3.4 Funds can be locked due to blacklisted tokens

Severity: Low

Context: `raffle.cairo`

Description:

The functions `refund()`, `withdraw()`, `claim_winning_prize()` allows a user to retrieve the tokens deposited after the raffle has been drawn. In case the user's address is blacklisted, his funds end up locked on the contract.

Recommendation:

Consider giving the user the ability to choose at which address to withdraw the tokens by adding a `ContractAddress` parameter in the relevant functions.

Project: Fixed.

Beyond: Acknowledged.

6.3.5 Low decimals tokens are not compatible

Severity: Low

Context: `oracle.cairo#L68-L71`

Description:

The `convert_to_usd()` function reverts in case the token decimals is lower than the oracle's decimals due to an underflow.

```
fn convert_to_usd(
    self: @ComponentState<TContractState>,
    amount: u256,
    unit_price: u128,
    offchain_decimals: u32,
    current_decimals: u8
) -> u256 {
    let converted_amount = amount
@>      / pow(10, current_decimals.into() - offchain_decimals.into());
    let total_price = converted_amount * unit_price.into();
    total_price / pow(10, 8)
}
```

Every core functionality of the protocol depends on the result of this function meaning the protocol cannot operate if it reverts.

Recommendation:

There are 2 cases to consider:

- `current_decimals` is greater than or equal to `offchain_decimals` (which is handled by the current design)
- `current_decimals` is less than `offchain_decimals`. In this case `converted_amount` must be scaled up to match the offchain decimals.

*The below takes issue **H-02** recommendation into account*

```
if(current_decimals < offchain_decimals) {
    let converted_amount = amount
        * pow(10, offchain_decimals.into() - current_decimals.into());
    let total_price = converted_amount * unit_price.into();
    total_price / pow(10, offchain_decimals.into())
}
```

Project: Fixed.

Beyond: Acknowledged.

6.3.6 Limited number of collections

Severity: Low

Context: `raffle.cairo`

Description:

The protocol has the ability to restrict the raffle to participants that own an NFT of a particular collection. The allowed collections are stored in the `raffle_authorized_collection` mapping. However, this mapping maps a `u8` to a `ContractAddress`

```
raffle_authorized_collection: Map<u8, ContractAddress>;
```

This means only up to 255 collection can be allowed in a raffle if this feature is used.

Recommendation:

Consider increasing the number of collections allowed in a raffle by updating the relevant storage and local variables to at least `u16`.

Project: Fixed.

Beyond: Acknowledged.

6.4 Informational

6.4.1 Loop can `break` directly

In `ducks_raffle.cairo::is_ducks_frens_holder()`, once `is_holder` is set to `true`, the loop can be broken directly.

```
if caller_balance > 0 {
    is_holder = true;
+   break;
}
```

6.4.2 Shadow declarations

1. In `vesu_raffle.cairo::draw()`, the `asset_contract` variable is shadowed but holds the same value. Thus, the shadow declaration can be removed.

```
fn draw(ref self: RaffleComponent::ComponentState<ContractState>) {
    // ----- snip -----
    let contract_address = get_contract_address();
@>  let asset_contract = IERC20CamelDispatcher {
        contract_address: contract_state.asset_address()
    };
    assert(
        asset_contract.balanceOf(contract_address) >= self.total_deposited_assets(),
        Errors::INSUFFICIENT_YIELD_GENERATED
    );

-   let asset_contract = IERC20CamelDispatcher {
-       contract_address: self.raffle_asset_contract.read()
-   };
```

2. In `raffle.cairo::claim_winning_prize()`, the `caller_address` variable is shadowed but holds the same value. Thus, the shadow declaration can be removed.

```
fn claim_winning_prize(ref self: ComponentState<TContractState>) {
    assert(self.raffle_closing_time.read() <= get_block_timestamp(), Errors::NOT_CLOSE);
    assert(self.raffle_drawn.read(), Errors::WAITING_DRAW);

@>  let caller_address = get_caller_address();
    let winner_dispatcher = IWinnerManagerDispatcher {
        contract_address: self.raffle_winner_contract.read().try_into().unwrap()
    };
    let winner_id = winner_dispatcher.get_winner_id(get_contract_address(), caller_address);
-   let caller_address = get_caller_address();
```

6.5 Positive design choices

6.5.1 Checks on token transfer success

Bountive verifies token transfers have succeeded by checking the recipient's token balance after the transfer. In case the balance differs from the expected the transaction reverts.

6.5.2 Non-transferable shares

All raffles could have been DoS'ed by sending 1 wei of shares to the raffle contract itself however since the shares are not transferrable, the attack cannot be performed.

Consideration: In order to avoid the above situation in a future implementation, consider modifying the assertion in the `withdraw_all_non_player_capital()` function of **every raffle contract**.

```
assert(
-    shares_contract.balance_of(contract_address) == non_player_capital,
+    shares_contract.balance_of(contract_address) >= non_player_capital,
    Errors::NOT_ENOUGH_SHARES
);
```

7 Appendix

7.0.1 Incorrect weight calculation in reduce_user_weight function leads to potential underflow

```
use bountive::interfaces::raffle::{IRaffleComponentDispatcher, IRaffleComponentDispatcherTrait};
use bountive::interfaces::factory::{IRaffleFactoryDispatcher, IRaffleFactoryDispatcherTrait};
use bountive::interfaces::erc20_mintable_burnable::{IERC20MintableBurnableDispatcher,
↪  IERC20MintableBurnableDispatcherTrait};
use core::option::OptionTrait;
use core::traits::{TryInto};

use openzeppelin::token::erc20::interface::{IERC20Dispatcher, IERC20DispatcherTrait};
use openzeppelin::utils::serde::SerializedAppend;

use snforge_std::{
    declare, ContractClassTrait, DeclareResultTrait, start_cheat_caller_address,
    ↪  stop_cheat_caller_address,
    start_cheat_caller_address_global, stop_cheat_caller_address_global,
    ↪  start_cheat_block_timestamp_global
};
use starknet::ContractAddress;
use starknet::get_block_timestamp;

// Constants
const ADMIN_ADDRESS: felt252 = 0x017e7dac0da1b79e08659fcf7bcb5ae30c707f0fee941233691f7684e7bfc78f;
const USER_ADDRESS: felt252 = 0x059a943ca214c10234b9a3b61c558ac20c005127d183b86a99a8f3c60a08b4ff;
const FACTORY_ADDRESS: felt252 = 0x05f42c38730f9619962c8bf7f44828f994bab0d6e8d106269bc6970fdac35e0f;
const ETH_ADDRESS: felt252 = 0x049d36570d4e46f48e99674bd3fcc84644ddd6b96f7c741b1562b82f9e004dc7;
const ONE_DAY_PLUS_ONE_SECOND: u64 = 86401;

// Helper functions
fn admin() -> ContractAddress {
    ADMIN_ADDRESS.try_into().unwrap()
}

fn user() -> ContractAddress {
    USER_ADDRESS.try_into().unwrap()
}

/// Sets up the test environment by:
/// 1. Deploying an ERC20 token for shares
/// 2. Deploying a ZkLendRaffle contract through the factory
/// Returns the raffle dispatcher
fn __setup__() -> IRaffleComponentDispatcher {
    let factory_address: ContractAddress = FACTORY_ADDRESS.try_into().unwrap();
    start_cheat_caller_address_global(admin());

    // Deploy shares token
    let erc20_contract = declare("ERC20MintableBurnable").unwrap().contract_class();
    let name: ByteArray = "BountiveToken";
    let symbol: ByteArray = "BoETH";
    let mut erc20_calldata = array![];
    erc20_calldata.append_serde(name);
    erc20_calldata.append_serde(symbol);
    erc20_calldata.append_serde(18_u8);
    erc20_calldata.append_serde(admin());
    let (shares_address, _) = erc20_contract.deploy(@erc20_calldata).unwrap();

    // Configure shares token
    IERC20MintableBurnableDispatcher { contract_address: shares_address }
        .set_factory_contract(factory_address);
    // Deploy raffle
    let class_hash_raffle = declare("ZkLendRaffle").unwrap().contract_class();
    let raffle_address = IRaffleFactoryDispatcher { contract_address: factory_address }
        .deploy_new_raffle(
            *class_hash_raffle.class_hash,
            ETH_ADDRESS.try_into().unwrap(),
            shares_address,
            604_800, // 7 days
            'zklend',
            3,
            19514442401534788
        );

    // Verify deployment
    assert(
        IRaffleFactoryDispatcher { contract_address: factory_address }.exists(raffle_address),
        'Raffle should exist'
    );

    stop_cheat_caller_address_global();
    IRaffleComponentDispatcher { contract_address: raffle_address }
}

#[test]
#[should_panic(expected: 'u256_sub Overflow')]
#[fork("MAINNET")]
fn test_weight_index() {
    // Setup
    let dispatcher = __setup__();
    let amount = 100_u256;

    // Configure weight multipliers as admin
    start_cheat_caller_address_global(admin());
    dispatcher.set_weight_multiplier(array![
        0x31,0x302e3835,0x302e3731,0x302e3537,0x302e3432,0x302e3238,0x302e3134
    ]);
    stop_cheat_caller_address_global();

    // Setup user permissions
    start_cheat_caller_address(ETH_ADDRESS.try_into().unwrap(), user());
    start_cheat_caller_address(dispatcher.contract_address, user());

    // Approve token spending
    IERC20Dispatcher { contract_address: ETH_ADDRESS.try_into().unwrap() }
        .approve(dispatcher.contract_address, amount * 2);
    stop_cheat_caller_address(ETH_ADDRESS.try_into().unwrap());

    start_cheat_block_timestamp_global(get_block_timestamp() + ONE_DAY_PLUS_ONE_SECOND);
    // Day 1: Initial deposit with weight multiplier of 0x302e3835
    dispatcher.deposit(amount);
    let user_weight_day1 = dispatcher.get_user_weight(user());
    println!("User weight day1: {}", user_weight_day1);

    // Day 2: Second deposit with weight multiplier of 0x302e3731
    start_cheat_block_timestamp_global(get_block_timestamp() + ONE_DAY_PLUS_ONE_SECOND);
    dispatcher.deposit(amount);
    let user_weight_day2 = dispatcher.get_user_weight(user());
    println!("User weight day2: {}", user_weight_day2);

    // Calculate and verify weights
    let user_shares = dispatcher.shares(user());
    let weight_to_reduce = 0x302e3835 * user_shares;
    println!("Weight calculated to reduce the position: {}", weight_to_reduce);
    println!("Total user weight: {}", user_weight_day2);
    println!("over calculated diff {}", weight_to_reduce - user_weight_day2);
    println!("Weight reduction <= total user weight: {}", weight_to_reduce <= user_weight_day2);

    // Attempt withdrawal - should fail with underflow
    dispatcher.withdraw(user_shares);
}
```