Fundraiser Security Review

Pashov Audit Group

Conducted by: Shaka, MrPotatoMagic, peanuts

November 30th - December 2nd

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **avierauy/fundraiser** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Fundraiser

Fundraiser smart contract manages a fundraising system where users can buy nodes using native currency, USDC, or BEAM tokens while applying discounts and referral rewards. It tracks deposits, applies pricing rules, and allocates funds to a treasury, allowing the owner to configure discounts, referral bonuses, and prices for each token type.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 9973f0ce627034d5e45baabd82ad0acd3dc5189e

fixes review commit hash - 4283c94614bfbebaf8c4fa1549d171cce870f729

Scope

The following smart contracts were in scope of the audit:

- `Fundraiser`

7. Executive Summary

Over the course of the security review, Shaka, MrPotatoMagic, peanuts engaged with Fundraiser to review Fundraiser. In this period of time a total of **8** issues were uncovered.

Protocol Summary

Protocol Name	Fundraiser
Repository	https://github.com/avierauy/fundraiser
Date	November 30th - December 2nd
Protocol Type	Fundraising Service

Findings Count

Severity	**Amount**
Medium	2
Low	6
Total Findings	**8**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Malicious users can exploit the referral system to always receive extra discounts	Medium	Resolved
[M-02]	Nodes can be purchased for extremely low price	Medium	Resolved
[L-01]	lastPriceUpdate should be updated in the constructor	Low	Resolved
[L-02]	Missing msg.value check for non-native deposits	Low	Resolved
[L-03]	Consider ensuring the sum of percentages is less than 10000	Low	Resolved
[L-04]	lastPriceUpdate resets when the protocol is paused	Low	Resolved
[L-05]	Referrals cannot be changed	Low	Resolved
[L-06]	Fixed prices can be arbitraged by large price movements	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Malicious users can exploit the referral system to always receive extra discounts

Severity

Impact: Low

Likelihood: High

Description

Functions `_processDepositNative()`, `_processDepositUSDC()` and `_processDepositBEAM()` process user deposits in their respective tokens. Let's go through the _processDepositBEAM() function below:

- Line 253 ensures we sent the correct amount of tokens for the number of nodes being purchased.
- Line 257 calls the _setReferrer() function. Let's say user A passes B as referrer. This would permanently set the user A's referrer as B.
- Line 260 calls the _isValidReferrer() function, which ensures that the referrer is a non-zero address and is not user A themselves. In the if block, we apply the referral bonus and discount to the original amount.
- Line 264 sends the referral bonus to user B and Line 270 sends the discounted amount to the treasury

```
File: Fundraiser.sol
252:     function _processDepositBEAM
  (uint256 quantity, uint256 amount, address referrer) internal {
253:         require(amount == (quantity * getPrice
  (TokenType.BEAM)), InvalidAmount());
254:
255:         uint256 discountedAmount = amount;
256:
257:         _setReferrer(msg.sender, referrer);
258:
259:
260:         if (_isValidReferrer(msg.sender, referrer)) {
261:             uint256 referralReward = _getReferralReward(amount);
262:             uint256 discountReward = _getDiscount(amount);
263:             discountedAmount = amount - (discountReward + referralReward);
264:             beam.safeTransferFrom(msg.sender, referrer, referralReward);
265:             referrals[referrer].totalBEAMRewarded += referralReward;
266:             referrals[referrer].totalDepositsReferred++;
267:             emit ReferralReward(TokenType.BEAM, referrer, referralReward);
268:         }
269:
270:         beam.safeTransferFrom(msg.sender, treasury, discountedAmount);
271:         _processDeposit
  (TokenType.BEAM, msg.sender, discountedAmount, quantity);
272:         totalBEAMDeposited += discountedAmount;
273:     }
```

As we know, referrals are usually intended to be a one-time reward for a specific user (in our case B). The issue is that user A can keep supplying address(B) as the referrer on every deposit. In this case, address(B) can simply be an alias address of user A. This is done to bypass the required check in the _setReferrer() function's else block and _isValidReferrer().

```
function _setReferrer(address user, address referrer) internal {
        if (users[user].referrer == address(0) && _isValidReferrer
          (user, referrer)) {
            users[user].referrer = referrer;
        } else {
            require(users[user].referrer == referrer || referrer == address
              (0), InvalidReferral());
        }
    }

    function _isValidReferrer
      (address user, address referrer) internal pure returns (bool) {
        return referrer != address(0) && referrer != user;
    }
```

Since users can keep on providing the same referrer (which can be their alias address) to gain discounts and always receive back the referral rewards, this indirectly means losses for the protocol team since they do not receive the expected amount for node purchases.

Additionally, there is a second impact to this issue i.e. it also increases the `totalDepositsReferred` variable for the referrer B though it's the same user A depositing again and again. Another way to look at this

`totalDepositsReferred` issue is, let's say user A is using referrer B's address for a referral. User A decides to buy 10 nodes. Instead of buying 10 nodes all at once, user A buys them one by one and passes the same referrer B every time. This means that `totalDepositsReferred` for the referrer would be 10 instead of 1. This logic is clearly flawed since if user A had bought all 10 nodes at once, the `totalDepositsReferred` would've been 1. So there's a clear discrepancy in behavior arising from the root cause pointed out.

Coded POC

How to use this POC:

- Integrate foundry as guided here
- Run both tests using `forge test --mt testReferralExploit` and `forge test --mt testTotalDepositsReferredIssue`. You can use the `-vvvvv` flag to view traces in the terminal.
- The first test proves how user A is always able to use its alias address to receive back the referral bonus amount + receive an additional discount.
- The second test proves how user B's (the referrer's) `totalDepositsReferred` value is different depending on how it is called by user A.
- Both the above impacts arise from the same root cause. The solution has been provided under `Recommendations`.

```
// SPDX-License-Identifier: BUSL-1.1

pragma solidity ^0.8.28;

import {Test, console2} from "forge-std/Test.sol";
import {Fundraiser} from "../contracts/Fundraiser.sol";

contract TestReferralExploit is Test {

    Fundraiser fundRaiser;

    address treasury = makeAddr("treasury");

    address userA = makeAddr("userA");
    address userA_Alias = makeAddr("userA_Alias");
    address userB = makeAddr("userB");

    function setUp() public {

        // Deploy fund raiser with 1e18 as ETH price (i.e. 1 node costs 1 ETH).
        fundRaiser = new Fundraiser(treasury, address(0), address
          (0), 1e18, 0, 0);
    }

    function testReferralExploit() public {
        fundRaiser.start();

        // Set ETH balance of User A as 10 ETH
        vm.deal(userA, 10e18);

        // User A's first 10 ETH deposit with referrer as user A's alias address
        vm.prank(userA);
        fundRaiser.deposit{value: 10e18}
          (Fundraiser.TokenType.NATIVE, 10, 10e18, userA_Alias);

        // Assert:
        // 1. User A's alias address received 5% of 10e18
        // 2. User A still holds 5% of 10e18 (discounted amount)
        // 3. Treasury received 90% of 10e18
        assertEq(userA_Alias.balance, 500/10000 * 10e18);
        assertEq(userA.balance, 500/10000 * 10e18);
        assertEq(treasury.balance, 90/100 * 10e18);

        // Mint 10 ETH to User A
        //(existing balance + mint 10 ETH due to how vm.deal works)
        vm.deal(userA, userA.balance + 10e18);

        // User A's second 10 ETH deposit with same referrer being passed as
        // parameter
        vm.prank(userA);
        fundRaiser.deposit{value: 10e18}
          (Fundraiser.TokenType.NATIVE, 10, 10e18, userA_Alias);

        // Assert:
        // 1. User A's alias address received 5% of 10e18 + previous first
        // deposit amount
        // 2. User A still holds 5% of 10e18
        //(discounted amount) + previous first deposit amount
        // 3. Treasury received 90% of 10e18 + previous first deposit amount
        // Note - We just multiply by 2 to consider first deposit amount as
        // well.
        assertEq(userA_Alias.balance, (500/10000 * 10e18) * 2);
        assertEq(userA.balance, (500/10000 * 10e18) * 2);
        assertEq(treasury.balance, (90/100 * 10e18) * 2);
    }

    function testTotalDepositsReferredIssue() public {
```

```
        fundRaiser.start();

        vm.deal(userA, 10e18);

        // Buying 10 nodes at one time
        vm.prank(userA);
        fundRaiser.deposit{value: 10e18}
          (Fundraiser.TokenType.NATIVE, 10, 10e18, userB);

        (, , , uint256 totalDepositsReferred) = fundRaiser.referrals(userB);
        assertEq(totalDepositsReferred, 1);

        vm.deal(userA, 10e18);

        // Buying 1 node at a time upto 10 nodes
        for(uint256 i; i < 10; i++) {
            vm.prank(userA);
            fundRaiser.deposit{value: 1e18}
              (Fundraiser.TokenType.NATIVE, 1, 1e18, userB);
        }

        (, , , totalDepositsReferred) = fundRaiser.referrals(userB);
        assertEq
        //(totalDepositsReferred, 10 + 1); // Adding one from previous 10 nodes at one
    }
}
```

Recommendations

There need to be two solutions for this:

1. Remove the first condition from _setReferrer() function's else block. This would ensure that user's cannot exploit the referral mechanism with alias addresses and receive extra discounts.

```
function _setReferrer(address user, address referrer) internal {
        if (users[user].referrer == address(0) && _isValidReferrer
          (user, referrer)) {
            users[user].referrer = referrer;
        } else {
+           require(referrer == address(0), InvalidReferral());
        }
    }
```

2. Preventing the use of alias addresses cannot be solved on the contract level since it is a sybil problem. To resolve this, consider using a signature validation based mechanism on the contract level which ensures that referrer's being used are always validated by the protocol's signer. The frontend would need to attempt to implement an anti-sybil mechanism. Examples such as Guild and/or Zealy can be used to increase sybil protection.

[M-02] Nodes can be purchased for extremely low price

Severity

Impact: High

Likelihood: Low

Description

The code snippet below is from the `deploy-multichain.js` file. As we can see, we use `usdcBasePrice` (parsed mwei value) during deployment instead of the `finalUSDCBasePrice` (parsed ether value) variable. USDC has 18 decimals on BSC mainnet while on other chains it is 6. Since the price is deployed with 6 decimals instead of 18, buying nodes would be extremely cheap (close to zero) on BSC mainnet, causing a loss to the protocol during the launch.

```
if (networkName === "bscMainnet") {
      finalUSDCBasePrice = ethers.parseUnits(usdcBasePrice, "ether");
    }

    const contract = await Fundraiser.deploy
      (treasury, usdc, beam, nativeBasePrice, usdcBasePrice, beamBasePrice);
```

While the `deploy-multichain.js` is out-of-scope for this review, it does include an important bug that should be fixed.

Recommendations

Implement the fix below:

```
+ const contract = await Fundraiser.deploy
+ (treasury, usdc, beam, nativeBasePrice, finalUSDCBasePrice, beamBasePrice);
```

8.2. Low Findings

[L-01] lastPriceUpdate should be updated in the constructor

When the contract is created, all the native prices are set and the contract is set to `paused` mode. Only the owner can unpause the contract by calling `start()`.

`start()` will set `lastPriceUpdate = block.timestamp`.

In the period between the creation of the contract and `start()`, users might be interested in the price, and call `getPrice()` while waiting. If they called in before `start()` is called, the `priceMultiplier` will be counted and the price returned will be wrong because `lastPriceUpdate` is still zero.

```
function getPrice(TokenType tokenType) public view returns (uint256) {
        bool weekElapsed = block.timestamp > (lastPriceUpdate + 7 days);

        if (weekElapsed) {
            if (tokenType == TokenType.NATIVE) {
                return nativeBasePrice * priceMultiplier;
            } else if (tokenType == TokenType.USDC) {
                return usdcBasePrice * priceMultiplier;
            } else {
                return beamBasePrice * priceMultiplier;
            }
```

1. Protocol creates the contract and sets Beam price at $0.5. The protocol intends to start at a specific time, 2 days later after the contract is created
2. Users want to query the price of beam and call `getPrice()` instead of directly accessing the public `beamBasePrice` variable.
3. Since `lastPriceUpdate` is zero, `weekElapsed` is true, and the price becomes 0.5 * 3 = $1.5.

In the constructor, set `lastPriceUpdate` to `block.timestamp`. `lastPriceUpdate` will be updated again when `start()` is called.

[L-02] Missing msg.value check for non-

native deposits

The deposit() function allows users to deposit native tokens, USDC or BEAM to buy nodes. The function is marked payable which means that anyone can supply msg.value (i.e. native tokens) with the call always. The issue is that for the two else-if cases (i.e. USDC and BEAM), we do not ensure that msg.value is not sent.

Due to this, any native tokens sent along with the deposit() call will be locked in the contract permanently. The severity of this issue is low since it relies on user mistakes. Nonetheless, the contract should protect against user mistakes whenever possible.

```
function deposit
      (TokenType assetType, uint256 quantity, uint256 amount, address referrer)
        external
        payable
        whenNotPaused
    {
        require(amount > 0, InvalidAmount());
        require(quantity > 0, InvalidQuantity());
        if (assetType == TokenType.NATIVE) {
            require(amount == msg.value, InvalidAmount());
            _processDepositNative(quantity, amount, referrer);
        } else if (assetType == TokenType.USDC) {
            _processDepositUSDC(quantity, amount, referrer);
        } else if (assetType == TokenType.BEAM) {
            require(block.chainid == ETHEREUM_MAINNET_CHAIN_ID, InvalidChain());
            _processDepositBEAM(quantity, amount, referrer);
        } else {
            revert InvalidToken();
        }
    }
```

Recommended Mitigation Steps

```
function deposit
      (TokenType assetType, uint256 quantity, uint256 amount, address referrer)
        external
        payable
        whenNotPaused
    {
        require(amount > 0, InvalidAmount());
        require(quantity > 0, InvalidQuantity());
        if (assetType == TokenType.NATIVE) {
            require(amount == msg.value, InvalidAmount());
            _processDepositNative(quantity, amount, referrer);
        } else if (assetType == TokenType.USDC) {
+           require(msg.value == 0, InvalidAmount());
            _processDepositUSDC(quantity, amount, referrer);
        } else if (assetType == TokenType.BEAM) {
            require(block.chainid == ETHEREUM_MAINNET_CHAIN_ID, InvalidChain());
+          require(msg.value == 0, InvalidAmount());
            _processDepositBEAM(quantity, amount, referrer);
        } else {
            revert InvalidToken();
        }
    }
```

[L-03] Consider ensuring the sum of percentages is less than 10000

The referral bonus and discount percentages are applied during deposits. The issue is that we individually ensure these to be less than or equal to 10000 instead of ensuring their sum is not greater than or equal to 10000.

```
function setDiscountPercent(uint256 _discountPercent) external onlyOwner {
        require(_discountPercent <= PERCENT_DENOMINATOR, InvalidPercent());
        discountPercent = _discountPercent;
        emit PercentUpdated(PercentType.DISCOUNT, _discountPercent);
    }

    function setReferralBonusPercent
      (uint256 _referralBonusPercent) external onlyOwner {
        require(_referralBonusPercent <= PERCENT_DENOMINATOR, InvalidPercent());
        referralBonusPercent = _referralBonusPercent;
        emit PercentUpdated(PercentType.REFERRAL_BONUS, _referralBonusPercent);
    }
```

Due to this, while processing deposits, we could underflow in the subtraction (in the code snippet below) since the sum discountReward + referralReward would be greater than the amount.

```
uint256 referralReward = _getReferralReward(amount);
            uint256 discountReward = _getDiscount(amount);
            discountedAmount = amount - (discountReward + referralReward);
```

Recommended Mitigation Steps

While this issue indirectly relies on an admin misconfiguration, if the checks are anyway being implemented in the setters, consider implementing them correctly to ensure such a bug does not arise.

```
function setDiscountPercent(
      uint256_discountPercent,
      uint256_referralBonusPercent
    ) external onlyOwner {
+        require
+ (_discountPercent + _referralBonusPercent <= PERCENT_DENOMINATOR, InvalidPercent());

        discountPercent = _discountPercent;
        emit PercentUpdated(PercentType.DISCOUNT, _discountPercent);

+        referralBonusPercent = _referralBonusPercent;
+       emit PercentUpdated(PercentType.REFERRAL_BONUS, _referralBonusPercent);
    }
```

[L-04] lastPriceUpdate resets when the protocol is paused

The start() function allows the owner to unpause the contract (that was initially paused on deployment). Unpausing the contract allows users to deposit native tokens, USDC or BEAM to buy nodes. Nodes are bought at the price set by the owner in the contract. After the first 7 days of launch, the price of the node multiplies by `priceMultiplier`.

```
File: Fundraiser.sol
118:     function start() external onlyOwner {
119:         _unpause();
120:
122:         lastPriceUpdate = block.timestamp;
123:         emit Start(block.timestamp);
124:     }
```

The issue is that if the protocol pauses/unpauses again, `lastPriceUpdate` is reset to the current block.timestamp in the start() function as seen above. Due to this, when users buy nodes, the getPrice() function would return the base prices without the multiplier applied to them. This would mean an indirect loss to the protocol team.

```
function getPrice(TokenType tokenType) public view returns (uint256) {
        bool weekElapsed = block.timestamp > (lastPriceUpdate + 7 days);

        if (weekElapsed) {
            if (tokenType == TokenType.NATIVE) {
                return nativeBasePrice * priceMultiplier;
            } else if (tokenType == TokenType.USDC) {
                return usdcBasePrice * priceMultiplier;
            } else {
                return beamBasePrice * priceMultiplier;
            }
        } else {
            if (tokenType == TokenType.NATIVE) {
                return nativeBasePrice;
            } else if (tokenType == TokenType.USDC) {
                return usdcBasePrice;
            } else {
                return beamBasePrice;
            }
        }
    }
```

The severity of this issue is low since:

1. The protocol still receives the base price amount of tokens.
2. The team can always configure the base prices themselves to reflect any multipliers into them (assuming this bug was noticed).

Recommended Mitigation Steps The solution below ensures the lastPriceUpdate is only updated for the initial case. For future pauses/unpauses, it remains unchanged.

```
function start() external onlyOwner {
        _unpause();

+        if (lastPriceUpdate == 0) lastPriceUpdate = block.timestamp;

        emit Start(block.timestamp);
    }
```

[L-05] Referrals cannot be changed

When a user deposits a token, he sets the referrer as well. The referrer allows the user to get a discount, and the referrer can only be set once.

```
function _setReferrer(address user, address referrer) internal {
        if (users[user].referrer == address(0) && _isValidReferrer
          (user, referrer)) {
            users[user].referrer = referrer;
        } else {
            require(users[user].referrer == referrer || referrer == address
              (0), InvalidReferral());
        }
    }
```

However, the referrer can get blacklisted from USDC, or revert any low-level call if it's a contract, resulting in a failed transaction for the user. The user cannot change the referrer except by setting the referrer to `address(0)`, which will forfeit his discount rewards.

A potential scenario:

- Alice changes upon this protocol from a big-time user, so she uses his address as the referrer address.
- The protocol has a big sale and has a 25% discount, so Alice continuously buys a small sum to reap the benefit.
- The referrer suddenly got his USDC blacklisted, so USDC cannot be transferred to him.
- Alice cannot use the referrer's address anymore, only `address(0)`. This means that she cannot have the discount anymore.

It is recommended to allow the depositor to change the referrer address anytime.

[L-06] Fixed prices can be arbitraged by large price movements

The protocol sets a fixed price for NATIVE, USDC and BEAM.

```
function setNativeBasePrice(uint256 _nativeBasePrice) external onlyOwner {
        nativeBasePrice = _nativeBasePrice;
        emit BasePriceUpdated(TokenType.NATIVE, _nativeBasePrice);
    }
```

In case of a flash crash, the prices will change much faster than the owner can update. Depositors can gain by frontrunning the oracle update because the price will be higher than expected.

- Let's say one Node is worth 10 USD. This means the base price is 10 USDC or 500 BEAM (BEAM is $0.02 per token).
- BEAM suddenly crashed to $0.005 per token.
- Before the oracle updates, users still need 500 BEAM to buy a node, and now the node is worth $2.50 in BEAM.
- The oracle updates to 2000 BEAM per Node, but some people already bought their Node for $2.50.

If the native token price drops and BEAM price drops (USDC can depeg and price can drop too) at once, the owner has to update the price one by one, and depositors can look for the best arbitrage opportunity to get the node at the lowest cost.

Users will get to buy a node at a cheaper price.

It is recommended that an oracle be used to query prices.

Otherwise, the best practice is to ensure that all prices are changed in one atomic transaction to prevent any sandwich attacks.

```
function setAllPrices(
  uint256_nativeBasePrice,
  uint256_usdcBasePrice,
  uint256_beamBasePrice
) external onlyOwner {
        nativeBasePrice = _nativeBasePrice;
        usdcBasePrice = _usdcBasePrice;
        beamBasePrice = _beamBasePrice;
        emit BasePriceUpdated(TokenType.NATIVE, _nativeBasePrice);
        emit BasePriceUpdated(TokenType.USDC, _usdcBasePrice);
        emit BasePriceUpdated(TokenType.BEAM, _beamBasePrice);
    }
```