Custodia Security

BAOs.fun Review

Conducted By: Ali Kalout, Ali Shehab

Contents

1. Disclaimer

A smart contract security review cannot ensure the absolute absence of vulnerabilities. This process is limited by time, resources, and expertise and aims to identify as many vulnerabilities as possible. We cannot guarantee complete security after the review, nor can we assure that the review will detect every issue in your smart contracts. We strongly recommend follow-up security reviews, bug bounty programs, and on-chain monitoring.

2. Introduction

Custodia conducted a security assessment of BAOs.fun's smart contract ensuring its proper implementation.

3. About BAOs.fun

The BAOs.fun platform enables DAO and project fundraising with multi-token contribution support, precise equity tracking, and robust fund management capabilities. The platform is built on Solidity with Foundry testing framework and integrates Pyth Network for accurate price feeds.

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 substantial loss of assets within the protocol or significantly impacts a group of users.
- Medium: Causes a minor loss of funds (such as value leakage) or affects a core functionality of the protocol.
- Low: Leads to any unexpected behavior in some of the protocol's functionalities, but is not critical.

4.2. Likelihood

- High: The attack path is feasible with reasonable assumptions that replicate on-chain conditions, and the cost of the attack is relatively low compared to the potential funds that can be stolen or lost.
- Medium: The attack vector is conditionally incentivized but still relatively likely.
- Low: The attack requires too many or highly unlikely assumptions, or it demands a significant stake by the attacker with little or no incentive.

4.3. Action required for severity levels

- Critical: Must fix as soon as possible
- High: Must fix
- Medium: Should fix
- Low: Could fix

5. Security Assessment Summary

Duration: 16/04/2025 - 22/04/2025
Repository: beradigm/bao-contracts
Commit: 84c0bce580e6531f154aea0728c555fcc4be6d43

- src/*

6. Executive Summary

Throughout the security review, Ali Kalout and Ali Shehab engaged with BAOs.fun's team to review BAOs.fun. During this review, 13 issues were uncovered.

Findings Count

Severity	**Amount**
Critical	N/A
High	4
Medium	6
Low	3
Total Finding	**13**

Summary of Findings

ID	Title	Severity	Status
H-01	BAO owner should not be able to change the protocol admin in BAOs and EquityNFTs, the protocol admin wouldn't receive NFT royalties	High	Resolved
H-02	Royalties can be drained by continously calling `EquityNFT::claimRoyalties`	High	Resolved
H-03	refund reverts if the contributor has an OTC contirbution, as it doesn't handle `address(1)` token	High	Resolved
H-04	`totalRaised` and `contributions[user].amount` become stale and inaccurate over time	High	Resolved
M-01	EquityNFT doesn't support royalties in ERC20 tokens	Medium	Resolved
M-02	Sending ETH to the sender may fail if the caller is a contract, because of the `.transfer` usage	Medium	Resolved
M-03	Funds would be stuck if the BAO owner decided not to call `finalizeFundraising`	Medium	Resolved
M-04	`recordOtcContribution` lacks max contribution validation	Medium	Disputed
M-05	Users can’t claim their contribution NFT if they refunded earlier	Medium	Resolved
M-06	`refund` is not decreasing the `totalRaised` amount, leading to wrong contribution proportions	Medium	Resolved
L-01	`tokenURI` shows the BERA contribution as a whole number in `_formatEther`	Low	Resolved
L-02	Deprecated use of `Pyth.getPrice`	Low	Resolved
L-03	There's no way to refund an OTC contribution	Low	Resolved

7. Findings

7.1. High Findings

[H-01] BAO owner should not be able to change the protocol admin in BAOs and EquityNFTs, the protocol admin wouldn't receive NFT royalties

Severity:
High

Description:
Both the BAO owner and the protocol admin are expected to receive NFT royalty. However, the BAO owners could block the protocol admin from receieving those royalties by setting it as a differeent address.

This could be done in a couple of places:
1. `BaosFactory::deployDao` allows the deployer to override the BAO's protocol admin upon deployment:

```
// If protocolAdmin is not set in config, use the factory's protocolAdmin
if (updatedConfig.protocolAdmin == address(0)) {
    updatedConfig.protocolAdmin = protocolAdmin;
}
```

2. `EquityNFT::setProtocolAdmin` allows the BAO owner to override the protocol admin in the Equity NFT address:

```
/**
 * @dev Set new protocol admin
 * @param newProtocolAdmin New protocol admin address
 */
function setProtocolAdmin(address newProtocolAdmin) external onlyOwner {
    require(newProtocolAdmin != address(0), "Invalid protocol admin");
    protocolAdmin = newProtocolAdmin;
}
```

Recommendations:
Remove these code snippents, to block the BAO owner from overriding the protocol admin.

[H-02] Royalties can be drained by continously calling `EquityNFT::claimRoyalties`

Severity:
High

Description:
Both the BAO owner and the protocol admin are expected to receive NFT royalty. They could be claimed by calling `EquityNFT::claimRoyalties` by either tha BAO owner or the protocol admin, it calculates the caller's cut, and send it. However, it doesn't either send the roylaties to the other side, nor saves the claim. This allows either the BAO owner or the protocol admin to drain all royalties and steal other party's royalties.

Proof of Concept:
```
function test_DrainAllRoyalties() public {
    uint256 payment = 0.5 ether;
    vm.deal(marketplaceUser, payment);

    vm.prank(marketplaceUser);
    (bool sent1, ) = address(nft).call{value: payment}("");
    assertTrue(sent1);

    uint256 protocolAdminBalanceBefore = address(protocolAdmin).balance;

    for (uint256 i = 0; i < 100; i++) {
        vm.prank(protocolAdmin);
        nft.claimRoyalties();
    }

    assertEq(
        address(protocolAdmin).balance - protocolAdminBalanceBefore,
        payment - 1
    );

    uint256 daoManagerBalanceBefore = address(daoManager).balance;

    vm.prank(daoManager);
    nft.claimRoyalties();

    assertEq(address(daoManager).balance - daoManagerBalanceBefore, 0);
}
```

Recommendations:
Send both royalties on every claim call.

[H-03] refund reverts if the contributor has an OTC contirbution, as it doesn't handle `address(1)` token

---

Severity:
High

Description:
The `recordOtcContribution()` function marks OTC (off-chain or manual) contributions using `address(1)` in the `TokenContribution` struct:

```
TokenContribution({
    token: address(1),
    amount: 0,
    usdValue: ...
});
```

However, the `refund()` function does not handle `address(1)` explicitly, and treats it as a normal ERC20 token. When it reaches:

```
IERC20(contrib.token).safeTransfer(msg.sender, contrib.amount);
```

…it attempts to call `transfer()` on `IERC20(address(1))`, which is not a real contract and causes the transaction to revert.

This completely blocks refunds for any contributor who has at least one OTC contribution, even if they also contributed via ETH or ERC20.

Proof of Concept:

```
function test_noRefundIfUserGetOtcContribution() public {
    vm.prank(daoManager);
    bao.addSupportedToken(ibgtToken, ibgtUsdPriceId);

    // First add users to the whitelist
    address[] memory addresses = new address[](2);
    addresses[0] = user1;
    addresses[1] = user2;

    vm.prank(daoManager);
    bao.addToWhitelist(addresses);

    // Deal tokens to users for testing if not done already in setUp
    deal(ibgtToken, user1, 100 * 10 ** 18);

    // User1 approves and contributes 100 iBGT
    vm.startPrank(user1);
    IERC20(ibgtToken).approve(address(bao), 100 * 10 ** 18);

    // Empty update data since we're using the price from setup
    bytes[] memory updateData = new bytes[](0);
```

```
    bao.contributeWithToken(ibgtToken, 100 * 10 ** 18, updateData);
    vm.stopPrank();

    // Record an OTC contribution for user3 worth $800
    vm.prank(daoManager);
    bao.recordOtcContribution(user1, 100 * 10 ** 18, "");

    vm.warp(block.timestamp + 31 days);

    vm.prank(user1);
    vm.expectRevert();
    bao.refund();
}
```

Recommendations:

Explicitly handle `address(1)` (OTC marker) in the `refund()` loop:

```
for (uint256 i = 0; i < tokenContribs.length; i++) {
    TokenContribution storage contrib = tokenContribs[i];

    if (contrib.token == address(1)) {
        // OTC contribution, no refund needed
        continue;
    }

    if (contrib.token == address(0)) {
        payable(msg.sender).transfer(contrib.amount);
        emit Refund(msg.sender, address(0), contrib.amount);
    } else {
        IERC20(contrib.token).safeTransfer(msg.sender, contrib.amount);
        emit Refund(msg.sender, contrib.token, contrib.amount);
    }
}
```

[H-04] `totalRaised` and `contributions[user].amount` become stale and inaccurate over time

Severity:

High

Description:

The `BAO` contract tracks `totalRaised` and each user's `contributions[user].amount` in USD (18 decimals). However, these values are only updated during the moment of contribution using a snapshot of token prices. This leads to inconsistencies due to:

1. Token price changes: USD values become outdated when the token price changes after contribution.
2. Token removals: If a token is removed via `removeSupportedToken`, its contributions are still counted in `totalRaised`.
3. Refunds: Refunded contributions do not decrement `totalRaised`, inflating the fundraising progress.
4. Manual `setGoalReached`: The owner can mark goal as reached with incorrect `totalRaised`, allowing `finalizeFundraising()` with inflated numbers.

This causes multiple downstream issues:

- Misleading `goalReached` status.
- Incorrect share distribution in `finalizeFundraising`.
- Wrong contribution proportions shown in the NFTs (`claimNFT()` uses stale USD).
- Inability to determine accurate eligibility for refunds or further contributions.

Recommendations:

Implement a dynamic, on-demand recalculation model:

- Replace fixed `contributions[user].amount` with token-level records (`TokenContribution[]`) only, we can still have a static record for OTC contributions.
- Calculate `totalRaised` and each user's USD value on-the-fly using current Pyth prices.
- Introduce a permissionless function to recalculate `totalRaised` based on latest prices.
- Add a similar helper: `getCurrentUsdContribution(address user)` to compute real-time value of each user’s contributions.
- Replace usages of `contributions[user].amount` with this dynamic calculation where accuracy is important (e.g. gating logic, refunds, NFT proportions).

Once fundraising is finalized via `finalizeFundraising()`, it’s safe to cache the current USD values permanently:

- Capture each user's final USD value and the total at that moment.
- These values can then be stored and used for NFT minting, token URI rendering, share distribution, etc.

- This avoids recalculation post-finalization and reduces gas costs for `claimNFT()` calls.

This hybrid model ensures precision during fundraising, and performance afterward.

7.2. Medium Findings

[M-01] EquityNFT doesn't support royalties in ERC20 tokens

Severity:
Medium

Description:
The `EquityNFT` contract currently supports receiving and claiming royalties only in the native token (e.g., ETH or BERA), via:

1. `receive()` function to accumulate royalties
2. `claimRoyalties()` to allow the daoManager and protocolAdmin to withdraw their share

However, many modern NFT marketplaces support ERC20 tokens (e.g., USDC, DAI) as payment options. In such cases, royalty payments in ERC20s will not be detected, tracked, or claimable through the current contract.

This breaks the expectation of full royalty support and could result in lost or inaccessible royalty revenue.

Recommendations:
Support royalties in ERC20 tokens.

[M-02] Sending ETH to the sender may fail if the caller is a contract, because of the `.transfer` usage

---

Severity:
Medium

Description:

This can revert if `msg.sender` is a contract, because `.transfer` only forwards **2300 gas**, which is not enough for contracts with non-trivial fallback logic or no `receive()` function.

This can block participation, leading to loss of funds or broken integrations

Recommendations:

Replace `.transfer()` with `.call{value: refund}("")` for safe and gas-flexible transfers:

```
(bool sent, ) = payable(msg.sender).call{value: refund}("");
require(sent, "ETH refund failed");
```

[M-03] Funds would be stuck if the BAO owner decided not to call `finalizeFundraising`

Severity:

Medium

Description:

If the fundraising goal is reached but the DAO owner (contract owner) decides not to call `finalizeFundraising()`, then all contributed funds—whether ETH or ERC20—become permanently stuck in the contract:

- Contributors cannot call `refund()`, because the goal was reached.
- `finalizeFundraising()` is restricted to `onlyOwner`, so only the DAO manager can initiate equity NFT minting and unlock fund withdrawal.
- `emergencyEscape()` is restricted to `protocolAdmin`, but it only transfers tokens from `supportedTokensList`. The owner can front-run the `protocolAdmin` and remove supported tokens using `removeSupportedToken()` before `emergencyEscape()` is called.

This creates a **centralized griefing vector**, where the DAO owner can block contributors from getting shares and block the protocol from recovering the funds.

Proof of Concept:

```
function test_neverFinalizingBug() public {
    vm.prank(daoManager);
    bao.addSupportedToken(ibgtToken, ibgtUsdPriceId);

    // First add users to the whitelist
    address[] memory addresses = new address[](2);
    addresses[0] = user1;
    addresses[1] = user2;

    vm.prank(daoManager);
    bao.addToWhitelist(addresses);

    // Deal tokens to users for testing if not done already in setUp
    deal(ibgtToken, user1, 15000 * 10 ** 18);
    deal(ibgtToken, user2, 15000 * 10 ** 18);

    // User1 approves and contributes 20 iBGT (worth $140 at $7 per iBGT)
    vm.startPrank(user1);
    IERC20(ibgtToken).approve(address(bao), 15000 * 10 ** 18);

    // Empty update data since we're using the price from setup
    bytes[] memory updateData = new bytes[](0);
    bao.contributeWithToken(ibgtToken, 15000 * 10 ** 18, updateData);
    vm.stopPrank();

    // User2 also contributes with iBGT
    vm.startPrank(user2);
    IERC20(ibgtToken).approve(address(bao), 15000 * 10 ** 18);
    bao.contributeWithToken(ibgtToken, 15000 * 10 ** 18, updateData);
    vm.stopPrank();

    //dao manager dont want to finalize
    //so the protocolAdmin decided to do emergencyEscape to save the money
    //but the owner so that and decided to front-run and remove the supported token
    //so that it will not be possible
    vm.prank(daoManager);
    bao.removeSupportedToken(ibgtToken);

    //now emergency escape won't work
    uint256 amountBefore = IERC20(ibgtToken).balanceOf(address(bao));
    vm.prank(protocolAdmin);
    bao.emergencyEscape();
    uint256 amountAfter = IERC20(ibgtToken).balanceOf(address(bao));
    assertEq(amountBefore, amountAfter);
}
```

Recommendations:

Allow the `protocolAdmin` to finalize fundraising if the goal is reached and the deadline passed.

[M-04] `recordOtcContribution` lacks max contribution validation

Severity:
Medium

Description:
The `recordOtcContribution()` function is used to manually record off-chain (OTC) contributions, such as a user sending an NFT or an asset outside the protocol. However, the function does not validate against contribution limits, such as `maxWhitelistAmount`.

This allows users to bypass whitelist contribution caps by:

1. Contributing up to their `maxWhitelistAmount` using `contribute()` or `contributeWithToken()`
2. Then sending an NFT or asset off-chain and having the DAO manager record an OTC contribution
3. The resulting total contribution exceeds the cap without reversion

This is especially dangerous in private or allowlist rounds, where the DAO wants to enforce strict per-user limits.

This allows users to bypass per-user contribution limits during private rounds.

Proof of Concept:
```
function test_bypassMaxWhitelistAmount() public {
    vm.prank(daoManager);
    bao.addSupportedToken(ibgtToken, ibgtUsdPriceId);

    // First add users to the whitelist
    address[] memory addresses = new address[](2);
    addresses[0] = user1;
    addresses[1] = user2;

    vm.prank(daoManager);
    bao.addToWhitelist(addresses);

    //maxWhitlist 1000$
```

```
    // Deal tokens to users for testing if not done already in setUp
    deal(ibgtToken, user1, 600 * 10 ** 18);

    // User1 approves and contributes 100 iBGT
    vm.startPrank(user1);
    IERC20(ibgtToken).approve(address(bao), 100 * 10 ** 18);

    // Empty update data since we're using the price from setup
    bytes[] memory updateData = new bytes[](0);
    bao.contributeWithToken(ibgtToken, 100 * 10 ** 18, updateData);
    vm.stopPrank();

    // Record an OTC contribution for user3 worth $800
    vm.prank(daoManager);
    bao.recordOtcContribution(user1, 800 * 10 ** 18, "");

    (uint256 user1Amount, ) = bao.contributions(user1);
    uint256 maxWhitelistAmount = bao.maxWhitelistAmount();

    assertGt(user1Amount, maxWhitelistAmount);
}
```

Recommendations:

Add validation logic inside `recordOtcContribution()`:

```
if (maxWhitelistAmount > 0) {
    require(whitelist[contributor], "Not whitelisted");
    require(
        contributions[contributor].amount + usdValue <= maxWhitelistAmount,
        "Exceeds max whitelist amount"
    );
} else if (maxPublicContributionAmount > 0) {
    require(
        contributions[contributor].amount + usdValue <= maxPublicContributionAmount,
        "Exceeds max public contribution amount"
    );
}
```

This mirrors the checks already present in `contribute()` and `contributeWithToken()`.

[M-05] Users can’t claim their contribution NFT if they refunded earlier

---

Severity:

Medium

Description:

When a contributor calls `refund()`, the contract sets:

```
claimed[msg.sender] = true;
```

Later, if:

1. The fundraising deadline is extended,
2. The contributor contributes again,
3. And the fundraising is finalized…

…the same user cannot claim their NFT, because `claimNFT()` includes the check:

```
require(!claimed[msg.sender], "Already claimed");
```

This logic incorrectly assumes that a "claim" or "refund" is terminal, even though the contributor may re-enter via a new contribution after a deadline extension.

Leading to contributors who refunded but later rejoined **cannot claim their NFT**.

Proof of Concept:

```
function test_CantClaimAfterRefund() public {
    address[] memory addresses = new address[](3);
    addresses[0] = user1;
    addresses[1] = user2;
    addresses[2] = user3;

    vm.prank(daoManager);
    bao.addToWhitelist(addresses);

    vm.deal(user1, 200 ether);
    vm.deal(user2, 200 ether);
    vm.deal(user3, 200 ether);

    vm.prank(user1);
    bao.contribute{value: 200 ether}();

    vm.prank(user2);
    bao.contribute{value: 200 ether}();

    vm.prank(user3);
    bao.contribute{value: 200 ether}();

    vm.warp(block.timestamp + 31 days);

    vm.mockCall(
        address(mockPyth),
```

```
        abi.encodeWithSelector(mockPyth.getPrice.selector, beraUsdPriceId),
        abi.encode(
            PythStructs.Price({
                price: BERA_USD_PRICE,
                conf: uint64(10000),
                expo: EXPO,
                publishTime: uint(block.timestamp)
            })
        )
    );

    vm.prank(user3);
    bao.refund();

    vm.prank(daoManager);
    bao.extendFundraisingDeadline(block.timestamp + 30 days);

    vm.prank(user3);
    bao.contribute{value: 200 ether}();

    vm.warp(block.timestamp + 31 days);

    vm.startPrank(daoManager);
    bao.setGoalReached();
    bao.finalizeFundraising(
        "myNFT",
        "MFT",
        "https://api.bao.fun/nft/metadata/"
    );
    vm.stopPrank();

    vm.prank(user1);
    bao.claimNFT();

    vm.prank(user2);
    bao.claimNFT();

    vm.prank(user3);
    vm.expectRevert(bytes("Already claimed"));
    bao.claimNFT();
}
```

Recommendations:

In `refund()`, remove the `claimed[msg.sender] = true;` flag entirely. Instead:

```
delete tokenContributions[msg.sender];
```

This safely resets the contributor state and allows them to re-enter.

[M-06] `refund` is not decreasing the `totalRaised` amount, leading to wrong contribution proportions

---

Severity:
Medium

Description:
The `BAO` contract tracks `totalRaised` to calculate proportional share distribution during `finalizeFundraising()`. However, when a user calls `refund()` (after the fundraising deadline and without goal being reached), their refunded amount is not deducted from `totalRaised`.

This leads to a mismatch: when the DAO owner later calls `setGoalReached()` and `finalizeFundraising()`, the proportions are calculated using the outdated `totalRaised` value, which includes refunded funds that no longer exist in the contract.

This results in inflated denominators in the share calculation and incorrect equity allocations.

Proof of Concept:
```
function test_settingGoalReachedManuallyAfterDeadline_wrongProportions()
    public
{
    address[] memory addresses = new address[](3);
    addresses[0] = user1;
    addresses[1] = user2;

    vm.prank(daoManager);
    bao.addToWhitelist(addresses);

    vm.deal(user1, 200 ether);
    vm.prank(user1);
    bao.contribute{value: 200 ether}();

    vm.deal(user2, 200 ether);
    vm.prank(user2);
    bao.contribute{value: 200 ether}();

    vm.warp(block.timestamp + 31 days);

    vm.prank(user1);
    bao.refund();

    vm.prank(daoManager);
```

```
    bao.setGoalReached();

    vm.prank(daoManager);
    bao.finalizeFundraising(
        "myNFT",
        "MFT",
        "https://api.bao.fun/nft/metadata/"
    );

    vm.prank(user2);
    bao.claimNFT();

    // proportions should is 50%
    assertEq(
        EquityNFT(payable(bao.contributorNFT())).tokenURI(
            bao.contributorNFTIds(user2)
        ),

"https://api.bao.fun/nft/metadata/2?contribution=8000000000000000000&proportion=5000&shares=
10000000000000000000000000"
    );
}
```

Recommendations:

Update `refund()` to decrement `totalRaised`:

```
// After summing refunds and before setting claimed
totalRaised -= contributedAmountInUsd;
```

7.3. Low Findings

[L-01] `tokenURI` shows the BERA contribution as a whole number in `_formatEther`

Severity:

Low

Description:

`EquityNFT::_formatEther` only returns the contributed BERA as a whole number:

```
function _formatEther(
    uint256 weiAmount
) internal pure returns (string memory) {
    // Simple implementation - convert to ether by dividing by 10^18
    uint256 ether_value = weiAmount / 1 ether;
```

```
    return ether_value.toString();
}
```

This leads to misleading NFT's token URIs, where it would show lower contribution, for example, both 1.0 and 1.9 show up as 1.0.

Recommendations:

Similar to `_formatBasisPoints`, make sure to atleast show 2 decimal places.

[L-02] Deprecated use of `Pyth.getPrice`

Severity:

Low

Description:

The `BAO` contract uses `pythOracle.getPrice()` to fetch asset prices from the Pyth Network.

However, per Pyth's official EVM documentation, the `getPrice()` function is deprecated. Instead, the recommended approach is to use `getEmaPriceNoOlderThan()` or `getPriceNoOlderThan()`, both of which internally perform timestamp validation.

Additionally, the current implementation performs a manual staleness check:

```
require(
    (block.timestamp - uint256(price.publishTime)) <= maxPriceAgeSecs,
    "ETH price feed stale or invalid"
);
```

This becomes redundant when switching to `get*NoOlderThan()` variants.

Using deprecated API may lead to unexpected breakage in future Pyth upgrades

Recommendations:

Update all price queries to:

```
PythStructs.Price memory ethPrice = pythOracle.getEmaPriceNoOlderThan(
    ID,
    maxPriceAgeSecs
);
```

[L-03] There's no way to refund an OTC contribution

Severity:
Low

Description:
The `recordOtcContribution()` function allows DAO managers to register off-chain contributions (like NFTs sent manually), but these contributions are non-refundable. If a contributor wants to reclaim their OTC asset, the current contract provides no way to track or return those.

Recommendations:
Implement a way to allow the BAO owner to remove the OTC contribution.