Redacted Cartel contest Findings & Analysis Report

2022-05-11

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Redacted Cartel smart contract system written in Solidity. The audit contest took place between February 15—February 17 2022.

Wardens

37 Wardens contributed reports to the Redacted Cartel contest:

1. csanuragjain
2. WatchPug (jtp and ming)
3. pauliax
4. hickuphh3

5. gzeon
6. cmichel
7. leastwood
8. cccz
9. danb
10. Omik
11. rfa
12. kirk-baird
13. 0x1f8b
14. SolidityScan (cyberboy and zombie)
15. hyh
16. Czar102
17. Dravee
18. ye0lde
19. kenta
20. kenzo
21. Jujic
22. llllllll
23. z3s
24. Ruhum
25. jayjonah8
26. defsec
27. robee
28. 0x0x0x
29. NoamYakov
30. peritoflores
31. 0xliumin
32. p4st13r4 (0x69e8 and 0xb4bb4)
33. d4rk

34. Tomio

This contest was judged by Alex the Entreprenerd. The judge also competed in the contest as a warden, but forfeited their winnings.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 18 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 18 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 25 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 18 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Redacted Cartel contest repository, and is composed of 4 smart contracts written in the Solidity programming language and includes 699 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges

- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

Medium Risk Findings (18)

[M-01] Manipulations of `setFee`

Submitted by pauliax

BribeVault.sol#L104-L113
BribeVault.sol#L164
BribeVault.sol#L213
BribeVault.sol#L256

If we consider that the fee variable is meaningfully applied, there will still be several problems with this:

1. Admin can setFee up to 100%. This is bad for users, fees should have a reasonable upper limit, e.g. 30% to prevent potential griefing.
2. Tokens are transferred in a separate function called transferBribes, which means that depositBribe txs have already settled. setFee can happen anytime, so an admin can change fees for already made deposits. I think this is again bad for users, as you need extra trust on an admin to not exploit this, and smart contracts should aim for as little external trust as possible.
3. Even if a fee would be applied in depositBribe, function setFee could frontrun user deposits. Consider using a timelock, so that users have time to react and adjust.

kphed (Redacted Cartel) disagreed with Medium severity and commented:

> We will likely set an upper bound as recommended, to ease user concerns. The admin being the Redacted multisig should also instill much trust and address most concerns.

Alex the Entreprenerd (judge) commented:

> The warden identified a potential admin privilege that would allow to set the fee to 100%.

> I agree with the finding and severity.

[M-02] DEPOSITOR_ROLE can be granted by the deployer of `BribeVault` and transfer briber's approved ERC20 tokens to `bribeVault` by specifying any `bribeIdentifier` and `rewardIdentifier`

Submitted by cccz

In the depositBribeERC20() function of the TokemakBribe contract, the briber can specify a proposal, and then call the depositBribeERC20 function of the bribeVault contract to deposit the reward.

```
function depositBribeERC20(
    address proposal,
    address token,
    uint256 amount
) external {
    uint256 currentRound = _round;
    require(
        proposalDeadlines[proposal] > block.timestamp,
        "Proposal deadline has passed"
    );
    require(token != address(0), "Invalid token");
    require(amount > 0, "Bribe amount must be greater than 0

    bytes32 bribeIdentifier = generateBribeVaultIdentifier(
        proposal,
        currentRound,
        token
    );
    bytes32 rewardIdentifier = generateRewardIdentifier(
        currentRound,
        token
    );
```

```
        IBribeVault(bribeVault).depositBribeERC20(
            bribeIdentifier,
            rewardIdentifier,
            token,
            amount,
            msg.sender
        );
```

But in the depositBribeERC20 function of the bribeVault contract, the address with DEPOSITOR_ROLE can transfer the briber's approved ERC20 tokens to bribeVault by specifying any bribeIdentifier and rewardIdentifier via safeTransferFrom.

```
    function depositBribeERC20(
        bytes32 bribeIdentifier,
        bytes32 rewardIdentifier,
        address token,
        uint256 amount,
        address briber
    ) external onlyRole(DEPOSITOR_ROLE) {
        require(bribeIdentifier.length > 0, "Invalid bribeIdenti
        require(rewardIdentifier.length > 0, "Invalid rewardIder
        require(token != address(0), "Invalid token");
        require(amount > 0, "Amount must be greater than 0");
        require(briber != address(0), "Invalid briber");

        Bribe storage b = bribes[bribeIdentifier];
        address currentToken = b.token;
        require(
            // If bribers want to bribe with a different token t
            currentToken == address(0) || currentToken == token,
            "Cannot change token"
        );

        // Since this method is called by a depositor contract,
        // that called the depositor contract - amount must be a
        IERC20(token).safeTransferFrom(briber, address(this), am
```

DEPOSITOR_ROLE can be granted by the deployer of BribeVault.

```
    function grantDepositorRole(address depositor)
```

```
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
    {
        require(depositor != address(0), "Invalid depositor");
        _grantRole(DEPOSITOR_ROLE, depositor);

        emit GrantDepositorRole(depositor);
    }
```

Proof of Concept

[BribeVault.sol#L164-L205]

Recommended Mitigation Steps

The depositBribeERC20 function of the TokemakBribe contract needs to first transfer the briber's tokens to the TokemakBribe contract, and then transfer the tokens to the bribeVault contract in the depositBribeERC20 function of the bribeVault contract. Make sure the first parameter of safeTransferFrom is msg.sender.

TokemakBribe.depositBribeERC20()

```
    function depositBribeERC20(
        address proposal,
        address token,
        uint256 amount
    ) external {
        uint256 currentRound = _round;
        require(
            proposalDeadlines[proposal] > block.timestamp,
            "Proposal deadline has passed"
        );
        require(token != address(0), "Invalid token");
        require(amount > 0, "Bribe amount must be greater than (

        bytes32 bribeIdentifier = generateBribeVaultIdentifier(
            proposal,
            currentRound,
            token
        );
        bytes32 rewardIdentifier = generateRewardIdentifier(
            currentRound,
            token
```

```
        );
+       IERC20(token).safeTransferFrom(msg.sender, address(this)

        IBribeVault(bribeVault).depositBribeERC20(
            bribeIdentifier,
            rewardIdentifier,
            token,
            amount,
            msg.sender
        );
```

bribeVault.depositBribeERC20()

```
    function depositBribeERC20(
        bytes32 bribeIdentifier,
        bytes32 rewardIdentifier,
        address token,
        uint256 amount,
        address briber
    ) external onlyRole(DEPOSITOR_ROLE) {
        require(bribeIdentifier.length > 0, "Invalid bribeIdenti
        require(rewardIdentifier.length > 0, "Invalid rewardIder
        require(token != address(0), "Invalid token");
        require(amount > 0, "Amount must be greater than 0");
        require(briber != address(0), "Invalid briber");

        Bribe storage b = bribes[bribeIdentifier];
        address currentToken = b.token;
        require(
            // If bribers want to bribe with a different token t
            currentToken == address(0) || currentToken == token,
            "Cannot change token"
        );

        // Since this method is called by a depositor contract,
        // that called the depositor contract - amount must be a
-       IERC20(token).safeTransferFrom(briber, address(this), am
+       IERC20(token).safeTransferFrom(msg.sender, address(this)
```

kphed (Redacted Cartel) disputed and commented:

> This isn't a concern since the "depositor" role can only be granted by admin (protocol multisig) - depositors will only be bribe contracts that we've written and deployed.

> In the future, we may grant the depositor role to contracts that are written and deployed by 3rd parties, but they would all be thoroughly vetted in some manner and need to conform to the BribeVault's interface.

> Thanks again for participating in our contest cccz, looking forward to more feedback/suggestions/comments.

Alex the Entreprenerd (judge) commented:

> While this may not be a concern for the sponsor, the smart contract is supposed to be given allowance, this allowance can then be used by the `DEPOSITOR_ROLE` to perform a transfer.

> The smart can then allow the `DEFAULT_ADMIN_ROLE` to withdraw the funds.

> Ultimately the ability to deposit being permissioned and it's ability to pull unlimited funds is a strong admin privilege, which I'd recommend the sponsor to remove.

> A similar deposit flow with less strict permissions can be found in most Yield Farming Vaults, see Badger Vaults for example: Badger-Finance/Vault.sol#L671.

[M-03] `DEFAULT_ADMIN_ROLE` of `BribeVault` can steal tokens from users' wallets

Submitted by WatchPug, also found by danb, Dravee, Alex the Entreprenerd, gzeon, IIIIIII, jayjonah8, kenzo, pauliax, cmichel, csanuragjain, and z3s

The current design/implementation allows the `DEFAULT_ADMIN_ROLE` of `BribeVault` to steal funds from any address that approved this contract up to allowance:

As a `DEFAULT_ADMIN_ROLE`, the attack is simply do the following steps:

1. `grantDepositorRole()` to self;
2. `BribeVault#depositBribeERC20()` and transfer funds from vict
3. `emergencyWithdrawERC20()`.

This can be effectively used as a backdoor/attack vector for a malicious/compromised wallet with `DEFAULT_ADMIN_ROLE` of `BribeVault` to steal all the tokens from users' wallets for these users who have approved `BribeVault`.

BribeVault.sol#L164-L187

```
function depositBribeERC20(
    bytes32 bribeIdentifier,
    bytes32 rewardIdentifier,
    address token,
    uint256 amount,
    address briber
) external onlyRole(DEPOSITOR_ROLE) {
    require(bribeIdentifier.length > 0, "Invalid bribeIdentifier
    require(rewardIdentifier.length > 0, "Invalid rewardIdentifi
    require(token != address(0), "Invalid token");
    require(amount > 0, "Amount must be greater than 0");
    require(briber != address(0), "Invalid briber");

    Bribe storage b = bribes[bribeIdentifier];
    address currentToken = b.token;
    require(
        // If bribers want to bribe with a different token they
        currentToken == address(0) || currentToken == token,
        "Cannot change token"
    );

    // Since this method is called by a depositor contract, we m
    // that called the depositor contract - amount must be apprc
    IERC20(token).safeTransferFrom(briber, address(this), amount
    ...
```

BribeVault.sol#L80-L88

```
function grantDepositorRole(address depositor)
    external
```

```
        onlyRole(DEFAULT_ADMIN_ROLE)
    {
        require(depositor != address(0), "Invalid depositor");
        _grantRole(DEPOSITOR_ROLE, depositor);

        emit GrantDepositorRole(depositor);
    }
```

Proof of Concept

Given:

- Alice (the victim) has approved `BribeVault` to spend `WBTC`;
- Alice has `100e8 WBTC` in their wallet balance.

A malicious/compromised `DEFAULT_ADMIN_ROLE` of `BribeVault` can do the following to steal tokens from users' wallets.

1. `grantDepositorRole()` to self;
2. `depositBribeERC20()` with: `token = WBTC`, `amount = 100e8`, and `briber` = Alice;
3. `emergencyWithdrawERC20()` with: `token = WBTC`, `amount = 100e8`.

As a result, the `100e8 WBTC` belongs Alice is now stolen by the Hacker.

The steps above can be repeated for all tokens and users, effectively stealing all the token balances from all the wallets that approved `BribeVault` up to the allowance limit, which usually is unlimited.

Recommended Mitigation Steps

1. Consider using `TokemakBribe` instead of `BribeVault` to hold users' allowances;
2. Consider making sure that the `from` parameter of `transferFrom` can only be `msg.sender`;
3. Consider using a multi-sig for the `DEFAULT_ADMIN_ROLE` of `BribeVault`.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> Fully agree with the finding and appreciate the level of detail.

> Because the exploit is contingent on a malicious owner, I believe Medium Severity to be more appropriate.

Alex the Entreprenerd (judge) commented:

> The `emergencyWithdrawERC20` without any check is a rug vector, protected exclusively by the multisig.
> While depositors may opt into this system, that doesn't mean that it's trust is fully reliant on the mulsitig, which means the code has trust assumptions by design.

> These trust assumptions make medium severity appropriate.

[M-04] Send ether with call instead of transfer

Submitted by kenta, also found by Dravee, hyh, Jujic, leastwood, and z3s

Use call instead of transfer to send ether. And return value must be checked if sending ether is successful or not. Sending ether with the transfer is no longer recommended.

Proof of Concept

RewardDistributor.sol#L181

Recommended Mitigation Steps

(bool result,) = payable(_account).call{value: _amount}(""); require(result, "Failed to send Ether");

kphed (Redacted Cartel) confirmed

Alex the Entreprenerd (judge) commented:

> I believe the function would actually work with most Smart Contract Wallets and proxies. However this could change in the future.

> Agree with the finding.

[M-05] Wrong slippage check

Submitted by cmichel, also found by danb, Alex the Entreprenerd, hickuphh3, hyh, and WatchPug

The `ThecosomataETH.addLiquidity` function computes the `expectedAmount` and then subtracts a slippage percentage from it.

```
function addLiquidity(uint256 ethAmount, uint256 btrflyAmount) i
    uint256[2] memory amounts = [ethAmount, btrflyAmount];
    uint256 expectedAmount = ICurveCryptoPool(CURVEPOOL).calc_to
        amounts
    );
    uint256 minAmount = expectedAmount - ((expectedAmount * slip

    ICurveCryptoPool(CURVEPOOL).add_liquidity(amounts, minAmount
}
```

According to the Curve docs 21.4, this amount is already exact and takes the slippage into account (but not fees).

If the pool is imbalanced, the `calc_token_amount` will already return a wrong amount and the additional slippage check on the wrong amount is unnecessary (except for the fees).

Recommended Mitigation Steps

Consider computing the minimum expected LP tokens off-chain and pass them to the `performUpkeep` function as a parameter to prevent sandwich attacks.

drahrealm (Redacted Cartel) confirmed and commented:

> Thanks for the finding. Confirmed that this is not the right way for handling slippage. Will be updating the flow a little bit to allow externally sourced data for the expected amount.

kphed (Redacted Cartel) disputed and commented:

> Changing to “sponsor disputed” since using values derived off-chain doesn’t prevent sandwich attacks and could make it easier to get sandwiched: using the off-chain calculation method, a MEV operator would only need to parse the tx input when deciding to front run us (vs. needing to simulate the tx if we were to do our calculations on-chain).

> Additionally, we’re not using the `StableSwap` contract referenced in the warden’s comment.

Alex the Entreprenerd (judge) commented:

> I think the sponsor’s perspective is interesting in that I believe any MEV researcher could write a simple algorithm to check for the tx inputs to detect a slippage check.

> However in practice they’d still have to run a simulation as you could input the off-chain price with any variation (different decimals, as ETH, as USD, as BTC, multiply by 2 or 5 or w/e)

> Additionally while there can be arguments made as to how to mitigate, the finding is still valid.

> Asking Curve for the `calc_token_amount` will return whatever price the pool can offer at that time, because tx are atomic that means that any front-running or price manipulation would have already happened in a tx before the request.

> This means that at worst you could directly use the output from `calc_token_amount` (multiplication has no impact).

> What the finding also implies, is that if the pool were to be completely imbalances (99% of in asset, 1% of out asset) the price you’d get would be very low, and the code wouldn’t be able to detect it (the code is effectively same as having 0 * .95)

> Because the finding has to do with potential value extraction, I believe the finding to be valid and of medium severity.

> As for mitigation, there are 2 viable options:

1. Use Chainlink Price Feed to get an accurate price
2. Provide the price as a parameter

> For option 2, I don't believe that argument to be valid for the examples above (just shift, multiply or obfuscate the param) Additionally, while you may never get a guarantee of perfect pricing, providing a price will give you a guarantee of a minimum price, this ensures you can opt-in into the slippage you'd be willing to tolerate.

> To give further details, let's look at using Flashbots (Flashbots RPC or a private mempool, either is a great idea).

> By using a private tx with the code provided for this contest, in the case of low liquidity, you'd still loose a considerable amount of value. No front-run needs to happen as in asking the price to the pool, you'll always get a valid response.

> This has happened to Yield Farming Aggregators (last I remember was yearn with StakeDAO token or similar). To summarize: Asking the price to the pool in the same tx is the same as having a 0 slippage check, which means you can loose value even without being front-run.

> Now let's add the idea of being front-run while using Flashbots RPC:
> -> You have calculate the off-chain Price, which means there' a require that will revert if the tx will fail, which means (because Flashbots is awesome) the tx won't be mined unless the tx goes through (miner get's a tip).
> This means you can be extremely strict with your slippage check, providing you with as much MEV protection as possible.

> For these reasons I believe the finding to be valid and I recommend you do explore:

- Flashbots (private TXs)
- Price as parameter
- Chainlink Price Feeds

kphed (Redacted Cartel) commented:

> I think the sponsor's perspective is interesting in that I believe any MEV researcher could write a simple algorithm to check for the tx inputs to detect a slippage

> check.
> That was my point, that it lowers the difficulty threshold.

> Thanks, the warden's recommended solution makes more sense now after your elaboration. We were planning on using Flashbots Protect and will look into your other suggestions as well.

[M-06] `SafeERC20.sol` is imported but not used in the `transferBribes()` function

Submitted by jayjonah8, also found by cccz, cmichel, Dravee, gzeon, hyh, lllllll, leastwood, NoamYakov, and Omik

In BribeVault.sol the transferBribes() function uses token.transfer() instead of token.safeTransfer. Tokens that don't correctly implement the latest EIP20 spec, like USDT, will be unusable in the protocol as they revert the transaction because of the missing return value. The fact that the SafeERC20.sol library is imported at the top of the BribeVault.sol implies that safeTransfer should be being used but may have been forgotten.

Proof of Concept

BribeVault.sol#L296

Recommended Mitigation Steps

It's recommended to use OpenZeppelin's SafeERC20 versions with the safeTransfer and safeTransferFrom functions that handle the return value check as well as non-standard-compliant tokens.

kphed (Redacted Cartel) confirmed and commented:

> Good catch!

> Thanks again for participating in our contest jayjonah8, looking forward to more feedback/suggestions/comments.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> Agree with the finding, because this is contingent on the specific token failing. I believe Medium severity to be more appropriate.

[M-07] Changing `bribeVault` in `RewardDistributor.sol` will Lock Current ETH Rewards

Submitted by kirk-baird, also found by WatchPug

[RewardDistributor.sol#L178-#L182](#)
[RewardDistributor.sol#L65-#L73](#)

Claiming of the ETH native currency requires `token` to be set to `bribeVault`. If the `bribeVault` is modified in `setBribeVault()` then users who have ETH rewards will now be considered to have `ERC20(bribeVault)` tokens. Since `bribeVault` is not an ERC20 token the `transfer()` call will fail and the users will not be able to claim their funds.

Recommended Mitigation Steps

Consider removing the functionality to change the `bribeVault` or ensuring all funds have been withdraw i.e. `balanceOf(address(this)) == 0` before changing the `bribeVault`.

kphed (Redacted Cartel) confirmed and commented:

> Good find, we're going to address this by making `bribeVault` immutable and removing the setter.

> Thanks again for participating in our contest kirk-baird, looking forward to more feedback/suggestions/comments.

Alex the Entreprenerd (judge) commented:

> Agree with the finding, ultimately preventing `bribeVault` from changing will provide users further security guarantees.

> An alternative solution would be to use a different code for ETH (I've seen protocols use address(0) or perhaps 0xeeeeeeee).

> However I believe that making `bribeVault` immutable will provide the stronger guarantees.

[M-08] Admin Privilege - Owner can rug via `ThecosomataETH.withdraw`

Submitted by Alex the Entreprenerd, also found by gzeon

Due to the generalized nature of `withdraw` the function is a clear rug-vector, allowing the `owner` to steal all funds.

Ideally, you should add some validation logic to limit the tokens or the amounts that the owner can withdraw.

Additionally, it's important that you disclose the level of admin privilege and the risk it can cause to your users and depositors.

Recommended Mitigation Steps

Disclose the admin privilege in your docs.
Refactor the code to reduce it.

kphed (Redacted Cartel) disputed and commented:

> The `owner` is our protocol multisig which has proven itself to be a trustworthy steward of funds (e.g. manages the Redacted treasury funds).

> The `withdraw` method is simply a utility to remove any ERC20 tokens that are unintentionally received. There won't be any funds to steal since it's not intended for the Thecosomata contract to custody funds for any extended period of time: our keepers will constantly poll the contract so that any BTRFLY received gets paired with ETH and added to our Curve LP immediately - any excess is burned.

Alex the Entreprenerd (judge) closed as Invalid and commented:

> It should be noted that I have submitted the finding, and in being judge of the contest am forfeiting my potential winnings.

> Personally, I don't believe a multisig gives any particular security guarantee to depositors beside the fact that it takes X amount of people to agree on how to move funds.

> The sponsor is making it clear that the `owner` in this case is also the depositor of funds.
> This means that the multi-sig is self custodying the funds into the contract.

> As such, the finding doesn't prove any additional security risk beside those that comes with a multi-sig.

> For those reasons, the finding is invalid.

kphed (Redacted Cartel) commented:

> Thanks for following up with your thoughts @Alex the Entreprenerd.

> NOTE: Mistakenly made comment below because I thought this was referring to the BribeVault contract.

> The sponsor is making it clear that the owner in this case is also the depositor of funds. This means that the multi-sig is self custodying the funds into the contract.

> Just to clear up any miscommunication or misunderstandings, we've never stated that the owner is the depositor of funds - the funds are deposited by bribers. The owner/admin only whitelists contracts that have permission to call the BribeVault's deposit methods but those contracts do not custody funds beyond the deposit transactions (this is also only the case when a briber deposits a native token).

Alex the Entreprenerd (judge) reopened as Valid and commented:

> Thank you for the clarification @kphed.

> If the bribers are not the same as the owner then the owner technically has the ability of withdrawing funds at any time, which puts the depositors under the risk

> of the owner rugging.

> Typically a Vault Protocol (Yearn, Badger) would have a check for "protectedTokens", in this case BTRFLY and WETH to prevent taking that type of operation.

> As it stands, the multisig can move the funds at any time, technically can frontrun the `keeper` and steal the funds.

> Also notice that you said that there will be a keeper for `performUpkeep` but the modifier is `onlyOwner` which either means you'll have an EOA as the owner, or you may want to change the access control checker (or remove it as Chainlink docs would require you to).

> With the information I have, I'm inclined to revert back to medium severity.

> While there's always the counter-argument that the multisig or governance will not rug, the only guarantee for it is the inability to rug by structuring the smart contract in a way that makes it impossible to move funds (e.g. add a check against moving BTRFLY and WETH, allow sweeping of other "random" tokens)

kphed (Redacted Cartel) commented:

> Sorry, disregard my last comment, I mistakenly read your comment as one directed towards BribeVault (which also has a method to withdraw tokens). You're correct, BTRFLY is minted by our protocol for ThecosomataETH. That said, we still don't consider the possibility of admin-rugging a real concern.

> Typically a Vault Protocol (Yearn, Badger) would have a check for "protectedTokens", in this case BTRFLY and WETH to prevent taking that type of operation.

> This is a potential idea, thanks, I'll share it with the team.

> Also notice that you said that there will be a keeper for performUpkeep but the modifier is onlyOwner which either means you'll have an EOA as the owner, or you may want to change the access control checker (or remove it as Chainlink docs would require you to).

> Tagging @drahrealm as he's implementing ThecosomataETH. Your comment about `onlyOwner` is a good one though - it does appear to be a mistake or can be improved tremendously (e.g. use AccessControl and add a role limited to calling this and not the withdraw method).

Alex the Entreprenerd (judge) commented:

> Would highly recommend limiting the withdrawal to specific tokens (ideally exclude important tokens), this would provide strong security guarantees against a rug.

> Also limiting roles can help reduce trust, however, it wouldn't address the underlying issue that "someone" can move the funds.

> With the information I have, I believe Medium Severity to be appropriate, and believe the sponsor has set motion to minimize trust as well as add additional security guarantees.

[M-09] Improper control over the versions of distributions' metadata may lead to repeated claims of rewards

Submitted by WatchPug

BribeVault.sol#L317-L324

```
    function updateRewardsMetadata(Common.Distribution[] calldata di
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
    {
        require(distributions.length > 0, "Invalid distributions");
        IRewardDistributor(distributor).updateRewardsMetadata(distri
    }
```

In the current implementation, `DEFAULT_ADMIN_ROLE` of `BribeVault` can call `updateRewardsMetadata()` to update the rewards metadata for the specified identifiers.

When a distribution's metadata is updated, it will also increase the `updateCount` and reset the claimed tracker.

[RewardDistributor.sol#L97-L119]

```
    function updateRewardsMetadata(
            Common.Distribution[] calldata _distributions
        ) external {
            require(msg.sender == bribeVault, "Invalid access");
            require(_distributions.length > 0, "Invalid _distributic

            for (uint256 i = 0; i < _distributions.length; i++) {
                // Update the metadata and also increment the update
                Reward storage reward = rewards[_distributions[i].re
                reward.token = _distributions[i].token;
                reward.merkleRoot = _distributions[i].merkleRoot;
                reward.proof = _distributions[i].proof;
                reward.updateCount += 1;

                emit RewardMetadataUpdated(
                    _distributions[i].rewardIdentifier,
                    _distributions[i].token,
                    _distributions[i].merkleRoot,
                    _distributions[i].proof,
                    reward.updateCount
                );
            }
        }
```

However, when the network is congested, `DEFAULT_ADMIN_ROLE` of `BribeVault` may mistakenly send 2 `updateRewardsMetadata()` txs, and the transactions can be packaged into different blocks.

Let's say there 2 `updateRewardsMetadata()` tx with the same calldata, if someone claims rewards in between the two txs, then they can claim again after the second transaction.

Proof of Concept

Given:

- `distributionA`'s proof is set wrong in `transferBribes()`
- `Alice` is eligible for rewards in `distributionA`
- the network is congested
- current block number = 10000
- `DEFAULT_ADMIN_ROLE` of `BribeVault` tries to call `updateRewardsMetadata()` and update `distributionA`'s proof;
- After a while, since the prev tx is stucked, `DEFAULT_ADMIN_ROLE` of `BribeVault` calls `updateRewardsMetadata()` again with same calldata;
- The first tx got packed into block 10010;
- `Alice` calls `claim()` and got the reward;
- The 2nd tx got packed into block 10020;
- `Alice` calls `claim()` again and get the reward again.

Recommended Mitigation Steps

Change to:

```
struct UpdateDistribution {
    bytes32 rewardIdentifier;
    address token;
    bytes32 merkleRoot;
    bytes32 proof;
    uint256 prevUpdateCount;
}

function updateRewardsMetadata(
    Common.UpdateDistribution[] calldata _distributions
) external {
    require(msg.sender == bribeVault, "Invalid access");
    require(_distributions.length > 0, "Invalid _distributions")

    for (uint256 i = 0; i < _distributions.length; i++) {
        require(reward.updateCount == _distributions[i].prevUpda
        // Update the metadata and also increment the update to
        Reward storage reward = rewards[_distributions[i].rewarc
        reward.token = _distributions[i].token;
        reward.merkleRoot = _distributions[i].merkleRoot;
        reward.proof = _distributions[i].proof;
```

```
            reward.updateCount += 1;

            emit RewardMetadataUpdated(
                _distributions[i].rewardIdentifier,
                _distributions[i].token,
                _distributions[i].merkleRoot,
                _distributions[i].proof,
                reward.updateCount
            );
        }
    }
```

[kphed (Redacted Cartel) disputed and commented]:

> However, when the network is congested, DEFAULT*ADMIN*ROLE of BribeVault may mistakenly send 2 updateRewardsMetadata() txs, and the transactions can be packaged into different blocks.

> The tx is executed via a multisig - we won't accidentally call it twice.

> After a while, since the prev tx is stucked, DEFAULT*ADMIN*ROLE of BribeVault calls updateRewardsMetadata() again with same calldata;

> In the scenario where we wanted to call `updateRewardsMetadata` again with the same calldata, we would use the same nonce as the stuck transaction.

[Alex the Entreprenerd (judge) decreased severity to Medium and commented]:

> I agree with both the warden and the sponsor in that I believe that:
>
> 1. the `DEFAULT_ADMIN_ROLE` can set metadata back to allow further (arbitrary) claims
> 2. This can be used by the admin to grief or alter the claimable rewards

> At the same time I have to agree that the Sponsor wouldn't be calling this "accidentally".

> I believe this finding to be valid, and to shed light into Admin Privilege, in that the Admin can set the metadata to whatever they want, allowing or denying claims at

> their will.

> As such I believe the finding to be valid, and Medium Severity to be more appropriate.

[M-10] Distributions must not match actual bribes

Submitted by cmichel

The `BribeVault.transferBribes` transfers tokens for distribution.
All parameters (`amounts` , `distributions`) are blindly accepted by the function and never verified to match the actual bribes that were deposited for the `distributions[i].rewardIdentifier` .

The `distributions[i].token` must not match the `distributions[i].rewardIdentifier` 's token (included in the reward identifier hash), and the `amounts[i]` (and fees) must not match the `bribes[bribeIdentifier].amount` .

The admin can submit arbitrary values and create distributions that don't reflect the bribe the distribution is actually for. It's easy to under-or overreport amounts for a bribe, take amounts from a different bribe, or steal all amounts from users by using 100% fees, distribute the same bribe over and over, etc.

Recommended Mitigation Steps

Reduce the trust that users need to have in the admin by validating the `Common.Distribution[] calldata distributions` , `uint256[] calldata amounts` , `uint256[] calldata fees` parameters against the deposited bribes.

For example:

- Check that the `distributions[i].token` matches the `distributions[i].rewardIdentifier`
- The amount + fees equal the `bribes[bribeIdentifier].amount` , then reset the `bribes[bribeIdentifier].amount` .

kphed (Redacted Cartel) disagreed with High severity and commented:

> We will be adding validation to the `transferBribes` method to provide peace of mind to our users, however, we consider this low-risk for the reasons below.

> All parameters (amounts, distributions) are blindly accepted by the function and never verified to match the actual bribes that were deposited for the distributions[i].rewardIdentifier. ...

> Compilation and thorough validation of the data necessary for calling `transferBribes` will be done off-chain using a publicly auditable set of scripts in our repo.

> Additionally, since the method can only be called by the protocol multisig (i.e. admin), signers will have the opportunity to review the data prior to submitting their signature. In conjunction with the above, they can generate their own data using the script and compare it against what is to be submitted.

> Thanks again for participating in our contest cmichel, looking forward to more feedback/suggestions/comments.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> I agree with the finding, the math for onChain bribes is not validated, while the math for bribes distribution is blindly trusted.

> This finding highlights a type of admin privilege, in which the admin can set arbitrary claims which do not reflect the actual state of the system.

> For this reason, as well as the eloquence of the warden, am going to mark this as valid and of medium severity.

[M-11] Depositor can spend funds of another Depositor

Submitted by csanuragjain

1. One depositor can spend funds of another depositor
2. Depositor can deposit in expired proposal
3. rewardIdentifier and bribeIdentifier can point to different rounds/tokens

Proof of Concept

One depositor can spend funds of another depositor:

1. Malicious depositor can call depositBribeERC20 at BribeVault.sol#L164 with briber as User B (Malicious user can generate bribeIdentifier with his own proposal)
2. Assume this User B has approved amount x to this contract
3. BribeVault.sol#L187 (IERC20(token).safeTransferFrom(briber, address(this), amount);) will transfer this amount x to the contract due to call at step 1. All this happen without User B knowledge

Expired Proposal:

1. Malicious depositor can generate bribeIdentifier of an expired proposal (proposalDeadlines[proposal] < block.timestamp) using generateBribeVaultIdentifier at TokemakBribe.sol#L166
2. Malicious depositor can then simply call depositBribeERC20 at BribeVault.sol#L164
3. Since there is no deadline check and this function blindly trusts bribeIdentifier, user deposit will be success even though the associated proposal already expired

RewardIdentifier and bribeIdentifier can point to different rounds/tokens:

1. Similar to expired proposal, depositor can generate bribeIdentifier and RewardIdentifier with different tokens and rounds.
2. Depositor now calls epositBribeERC20 at BribeVault.sol#L164 with the generated bribeIdentifier and RewardIdentifier
3. rewardToBribes[rewardIdentifier].push(bribeIdentifier); will update reward for round x and bribeIdentifier will point to round y which is incorrect

Recommended Mitigation Steps

depositBribeERC20 at **BribeVault.sol#L164** should only be allowed to be called via TokemakBribe.sol

kphed (Redacted Cartel) disputed and commented:

> depositBribeERC20 at BribeVault.sol#L164 should only be allowed to be called via TokemakBribe.sol

> Only those with the “depositor” role can call the deposit bribe methods (`depositBribeERC20` and `depositBribe`). We only grant the role to bribe contracts we own such as TokemakBribe.sol.

> Thanks again for participating in our contest csanuragjain, looking forward to more feedback/suggestions/comments.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> While the finding is similar to M-02, I believe this more eloquently shows the specific types of admin privileges that the `DEPOSITOR_ROLE` has for the function `depositBribeERC20` .

> Ultimately the finding is highlighting how things can go wrong and how the `DEPOSITOR_ROLE` provides a high level of admin privilege.

> Because this is contingent on a malicious admin, I believe Medium Severity to be more appropriate.

[M-12] Users Can Frontrun Calls to `updateRewardsMetadata()` And Claim Tokens Twice

Submitted by leastwood

The `updateRewardsMetadata()` function is called by the `BribeVault` contract by the admin role. The function will take a list of distributions which are used to update the associated reward metadata. It is expected that the merkle root will be updated to correctly identify which claimers have already claimed tokens.

`reward.updateCount` is incremented to reset the claimed tracker, allowing users that may have previously claimed, to claim their updated reward. However, there is potential for mis-use if users frontrun calls to `updateRewardsMetadata()` and claim their reward after the new merkle root has been calculated and updated by the

admin role. This may allow the claimer to double claim their rewards or lead to a loss in rewards if the reward metadata completely replaces the previous list of claimers.

Proof of Concept

RewardDistributor.sol#L97-L119
RewardDistributor.sol#L127-L209

Recommended Mitigation Steps

Consider implementing a delay where users cannot claim rewards before a call to `updateRewardsMetadata()` is made. This should ensure the admin role can construct a merkle tree based on the most up-to-date and correct data.

kphed (Redacted Cartel) confirmed and commented:

> After speaking with leastwood via Discord, I now believe this issue to be meaningfully different from issue M-09 and is a valid attack vector. His recommended solution above inspired a fix which we both agreed would solve the problem (i.e. set a "blank" merkle root, evaluate the users who claimed with the previous merkle root, and construct a new one accordingly).

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> I appreciate the nuance from the sponsor over this finding.
>
> Ultimately when using a merkle proof system, the new proof is calculated at a certain time. (ProofX)
>
> If UserA didn't claim when the proof was generated, they technically have time between when the proof is generated and the proof is published to claim for the first time.
>
> Because the new Merkle Proof (ProofX) was built to allow UserA to claim, they will be able to claim again.
>
> The only way I can think of to avoid this is to always only have one proof per set of claims, as to avoid getting front-run.

> There is merit to make this finding separate, although ultimately the reason why this is possible is because of the Admin ability to change the proofs at any time.

> So I'm going to suggest that this finding is similar to M-09, it's mitigation should be basically the same, however I'll mark as separate to give credit where it's due.

> Because the finding is contingent on external conditions (owner getting frontrun or owner being malicious), I believe medium severity to be appropriate.

> A mitigation could be to push new proofs via Flashbots, and use a snapshot like system to check that no claims where made in the time between the proof generation and the proof being set.

[M-13] Reentrancy in `depositBribeERC20` function

Submitted by Czar102, also found by 0x1f8b and SolidityScan

[BribeVault.sol#L164-L205]

`depositBribeERC20` function in `BriveVault` is reentrant in line 187, where an address supplied by the caller is called.

A bad actor that has `DEPOSITOR_ROLE` and is a contract can execute a folowing attack:

1. Create a dummy token contract, reentrant in the transferFrom() function. All tokens are approved to the `BriveVault` and the attacker contract has unlimited tokens. Reentrancy aims back to a function in the attacker contract, which calls `depositBribeERC20` again.
2. The first call by the contract must use a novel `bribeIdentifier`. `token` is set to a dummy contract and `amount` to `uint(-2)`.
3. All checks pass, `transferFrom` is called, which calls attacker contract, which can call `depositBribeERC20` again, this time will transfer 1 wei of a valuable token, using the same `bribeIdentifier`. All checks pass as the previous token hasn't been registered yet. Then, a valid transfer happens. After that, the amount is set to 1 wei and the token is saved. Event is emitted and the function returns value. Then, attacker function returns and dummy token returns. The operation

is to increment amount in storage by the transfer value, which increases `b.amount` to the maximum integer. The token is nonzero, so the if statement is passed.

Thus, an attacker can grant any amount of tokens from `BriveVault` to a certain bribe, stealing all the funds once the bribe will be withdrawn.

Recommended Mitigation Steps

Set bribe token before the transfer is made.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> I do believe re-entrancy is possible, so I recommend the sponsor to add the `nonReentrant` modifier to the deposit function.

> I'll keep the finding separate [from M-02] as this deals with reEntrancy. Mitigation would be to enforce a bribeIdentifier to be used for a specific token (and it being enforced), as well as adding `nonReentrant`.

> Because the function is permissioned, I believe medium severity to be more appropriate.

[M-14] transferBribes could transfer before proposal deadline + Input validation

Submitted by csanuragjain, also found by gzeon and WatchPug

It seems that Admin can call transferBribes even when proposals belonging to this rewardIdentifier have not expired. Also due to lack of input validation, token in distributions[i].rewardIdentifier might differ from distributions[i].token and also amount is not validated

Proof of Concept

transferBribes could transfer before proposal deadline:

1. Observe the transferBribes function at BribeVault.sol#L256

2. This is directly distributing the rewards for a rewardIdentifier even though proposal (bribeIdentifier) linked to this rewardToBribes[rewardIdentifier] might not have expired. This means users are still depositing and Admin transferred reward early

Input Validation:

1. Observe the transferBribes function at BribeVault.sol#L256
2. amounts object is directly passed by Admin and there is no verification to see that sum amount of all proposals under rewardToBribes[rewardIdentifier] is equal to amount provided by admin in argument
3. distributions[i].token is directly passed by Admin and there is no verification to see that token under distributions[i].rewardIdentifier is equal to the one provided by admin in argument

Recommended Mitigation Steps

Perform input validation.

kphed (Redacted Cartel) disagreed with Medium severity and commented:

> transferBribes could transfer before proposal deadline:
> ...

> Much of the validation will be handled off-chain at the time we compute the proofs and merkle roots (we still need to write the script for that). The reason for that approach is because we will have many similar contracts (but not exactly alike - e.g. some may not have deadlines) to TokemakBribe that will interact with BribeVault. It wouldn't be feasible to validate all the different constraints on-chain. That said, the admin is a multisig and the signers will have to agree on timing and correctness of the data to prevent this from happening.

> Input Validation:
> ...

> Thanks, we'll implement #2 and #3 to provide peace of mind to our users.

> Thanks again for participating in our contest csanuragjain, looking forward to more feedback/suggestions/comments.

Alex the Entreprenerd (judge) commented:

> While I understand the sponsor's reasoning, any validation that is not enforced by the Smart Contract can't be verified. It requires trust which could be minimized, if not removed, if the contract enforced those conditions.

> `bribeIdentifier`, it's relation with `token`, the fact that a bribe was transferred or not, all these events can be tracked onChain, offering clear paths for funds, which ultimately give more security guarantees to end users.

> Because this ultimately is a trust issue, I believe medium severity to be appropriate.

[M-15] Fees can be any amount

Submitted by danb, also found by pauliax

In `transferBribes`, the fees are user input, rather than calculation using `fee` (state var).
Currently, `fee` is unused: BribeVault.sol#L23.

Therefore the fees amounts might be wrong.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> I don't believe M-14 mentions validation of fees, as such will mark this finding as unique.

> Ultimately the function trusts the Admin input instead of using the storage variable, giving less security guarantees as to the fairness of the Distribution of the Bribes.

> Because this is contingent on a malicious admin, I believe medium severity to be appropriate.

[M-16] `DEPOSITOR_ROLE` can manipulate `b.amount` value

Submitted by rfa, also found by Omik

[BribeVault.sol#L187]

Malicious `DEPOSITOR_ROLE` can doing self transfer and manipulate `b.amount`

Proof of Concept

In case malicious `DEPOSITOR_ROLE` inputing `WETH` address and putting `briber` == address(this) in `safeTransferFrom` argument (which is self transfering). Therefore, it is posible to increase `b.amount` without any cost.

WETH token contract:

```
//Line 62 WETH contract
function transferFrom(address src, address dst, uint wad)
        public
        returns (bool)
    {
        require(balanceOf[src] >= wad);

        if (src != msg.sender && allowance[src][msg.sender] != u
            require(allowance[src][msg.sender] >= wad);
            allowance[src][msg.sender] -= wad; // <---------- t
        }

        balanceOf[src] -= wad;
        balanceOf[dst] += wad;

        Transfer(src, dst, wad);

        return true;
    }
```

If the condition didn't pass (in this case msg.sender != `src`), the transaction will treated like a transfer (doesn't need an allowance), Therefore its possible to do self transfer

Recommended Mitigation Steps

I recommend to validate that bribe != Address(this)

Alex the Entreprenerd (judge) commented:

> I'm marking the finding as unique because of the interesting mechanic.
> Some ERC20 will revert on trying to transfer to yourself, however the warden showed a specific exploit, using WETH, that could be used against the vault.

[M-17] ThecosomataETH: Oracle price can be better secured (freshness + tamper-resistance)

Submitted by hickuphh3

ThecosomataETH.sol#L94-L110

The `ThecosomataETH` contract adds ETH and BTRFLY tokens as liquidity into the ETH-BTRFLY curve crypto pool. The `calculateAmountRequiredForLP()` function relies on the `price_oracle` value returned by the pool to calculate the ETH and BTRFLY amounts to be added as liquidity. It is therefore important to ensure that `price_oracle` is accurate.

At the time of writing, the pool has about $5M in liquidity, which is comparable to that of the liquidity provided on UniswapV3. Flash loan attacks are therefore possible, but ineffective (explained further later).

In the curve v2 whitepaper, the price oracle mechanism is explained briefly in the "Algorithm for repegging" section. It is reproduced below for convenience.

> Internally, we have a price oracle given by an exponential moving average (EMA) applied in N-dimensional price space. Suppose that the last reported price is `pLast`, and the update happened `t` seconds ago while the half-time of the EMA is `T1/2`. Then the oracle price `p_new` is given as:

```
α = 2^(- t / T1/2);
p_new = pLast * (1 - α) + α * p_old // p_old = current price_ora
```

Impact

With oracles (curve pool now, to be switched to chainlink based oracle as per comment in L27), there is an inverse correlation between freshness and tamper-resistance.

We can expect `price_oracle` to be relatively fresh as trades will occur whenever arbitrage opportunities arise against the UniV3 pool which has comparable liquidity. Note that the `ETH-BTRFLY` pool has a half-time of 10 minutes (`T1/2 = 600`). This means that after exactly 10 mins, both `pLast` and `p_old` have equal weightage.

It is unclear how resistant the EMA oracle is against manipulation. Flash loan attacks, while possible, will be ineffective because `t` will be zero (`pLast` will be ignored in the update). However, a sophisticated attacker could possibly skew the price oracle by inflating the price of `BTRFLY` a couple of blocks before the `performUpkeep()` transaction to get the treasury to deposit more `ETH` / burn more `BTRFLY` than necessary.

Recommended Mitigation Steps

In my opinion, both freshness and tamper-resistance can be better secured.

This can be done by:

1. Ensuring that the price was updated within a certain limit.

```
// eg. last price update / trade must have been executed within
uint256 lastPricesTimestamp = ICurveCryptoPool(CURVEPOOL).last_p
require(block.timestamp - lastPricesTimestamp <= 1 hours, 'stale
```

2. Checking that the last reported price `pLast` has not deviated too far from the current oracle price `p_old` . One can argue that it would be safer to add liquidity when the market isn't volatile.

```
uint256 lastPrice = ICurveCryptoPool(CURVEPOOL).last_prices();
uint256 oraclePrice = ICurveCryptoPool(CURVEPOOL).price_oracle()
uint256 percentDiff;
// eg. require difference in prices to be within 5%
```

```
    if (lastPrice > oraclePrice) {
      percentDiff = (lastPrice - oraclePrice) * 1e18 / oraclePrice;
    } else {
      percentDiff = (oraclePrice - lastPrice) * 1e18 / oraclePrice;
    }
    require(percentDiff <= 5e16, 'volatile market');
```

drahrealm (Redacted Cartel) commented:

> Idem with M-05, we will proceed with doing calculating the min token amount off-chain, then specify it when calling `performUpKeep`.

> Thanks for the finding.

Alex the Entreprenerd (judge) commented:

> Agree that solution is based off of M-05.
> While I believe simpler solutions where highlighted, I feel the warden put in the extra effort to make a valuable submission.

> As such, I'll mark the finding as unique.

> Personally I would not trust Curve Pricing model over a Price Feed at this time, however am happy to be proven wrong.

[M-18] Rewards can be lost

Submitted by csanuragjain

Reward can be lost if bribeVault calls the updateRewardsMetadata on same rewardIdentifier again before user can claim his reward (since merkleRoot and proof will get updated).

Proof of Concept

1. bribeVault calls the updateRewardsMetadata at RewardDistributor.sol#L97 using rewardIdentifier X
2. Assume _distributions[i].proof contains merkle proof for User A

3. User A fails to call claim function
4. bribeVault again calls the updateRewardsMetadata at RewardDistributor.sol#L97 using rewardIdentifier X updating _distributions[i].proof which might not contain merkle proof of User A now. So User A loses his rewards

Recommended Mitigation Steps

bribeVault should only make second call to updateRewardsMetadata on same rewardIdentifier when all claimers have made their claims.

kphed (Redacted Cartel) disputed and commented:

> We would only call `updateRewardsMetadata` again if there was an issue with the originally-set merkle root(s). The recommended mitigation steps above would block us from setting the correct merkle roots until after claimers claimed the wrong amounts.

> Thanks again for participating in our contest csanuragjain, looking forward to more feedback/suggestions/comments.

Alex the Entreprenerd (judge) commented:

> The finding highlights the consequences of admin privilege, in that the admin can use `updateRewardsMetadata` to deny claims.

> While I believe the warden could have done a better job at expressing the risks involved for users, I believe the finding to be valid.

Low Risk and Non-Critical Issues

For this contest, 25 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by warden **hickuphh3** received the top score from the judge.

The following wardens also submitted reports: **ye0lde**, **kenzo**, **pauliax**, **Ruhum**, **WatchPug**, **cmichel**, **0x0x0x**, **csanuragjain**, **defsec**, **gzeon**, **kenta**, **SolidityScan**, **cccz**, **IIIIIII**, **peritoflores**, **0x1f8b**, **0xliumin**, **hyh**, **Omik**, **robee**, **Dravee**, **jayjonah8**, **p4st13r4**, *and* **danb**.

Codebase Impressions & Summary

This audit scope consisted of 4 contracts. Overall, the code quality is great. Inline comments and documentation provided was adequate. Various parties / roles and contract interactions were well explained.

Most issues raised are minor improvements to improve the security of the contracts. The only notable findings made had to do with the usage of the curve crypto pool's price oracle, and protection against sandwich attacks when adding liquidity.

In addition, I made a suggestion regarding the syncing of Tokemak's rounds with the `TokemakBribe` contract.

Note that I refrained raising issues regarding FoT tokens because I assume they are not meant to be supported.

[L-01] RewardDistributor: Change `payable(account).transfer()` to `.call()` for native fund transfers

Line References

RewardDistributor.sol#L181

Description

`BribeVault` uses `.call()` for native fund transfers, but `RewardDistributor` uses `.transfer()`. They should be standardized to `.call()`, the currently recommended method since `.transfer()` fowards 2300 gas whereas `.call()` forwards all / set gas.

Recommended Mitigation Steps

```
(bool sentAccount, ) = _account.call{value: _amount}("");
require(sentAccount, "Failed to transfer to _account");
```

[L-02] BribeVault: Use `safeTransfer` for tokens

Line References

[BribeVault.sol#L296-L297](#)

[BribeVault.sol#L337](#)

Description

Some ERC20 tokens like ZRX don't revert if the transfer fails. Since the `SafeERC20` has already been imported and the `safeTransferFrom` method used, the same should be done for token transfers.

Recommended Mitigation Steps

Replace `transfer` with `safeTransfer`.

[L-03] RewardDistributor: Limit native fund transfers to `bribeVault`

Line References

[RewardDistributor.sol#L58-L59](#)

Description

Since the only source of native fund transfers is expected to be the `bribeVault` contract, it would be good to restrict incoming fund transfers from other sources to prevent accidental transfers.

Recommended Mitigation Steps

```
receive() external payable {
  require(msg.sender == bribeVault, 'only bribeVault');
}
```

[L-04] TokemakBribe: Sync rounds with Tokemak's manager instead of manually setting rounds via `setRound()`

Line References

TokemakBribe.sol#L104-L110

Description

Instead of manually setting rounds, consider fetching the round number directly from Tokemak's manager contract via `[manager.currentCycleIndex()](https://etherscan.io/address/0xa86e412109f77c45a3bc1c5870b880492fb86a14#readProxyContract)`. While I initially wrote an issue about being able to set previous round numbers, after having chatted with the sponsor, it is intended to be a feature, not a bug.

Recommended Mitigation Steps

```
// TODO: change _round to getRound() wherever it is called in ot
function getRound() public view returns (uint256) {
  // if round is overridden, return set value
  if (_round != 0) return _round;
  // otherwise, if value is 0, use Tokemak's currentCycleIndex()
  // Tokemak manager at 0xa86e412109f77c45a3bc1c5870b880492fb86a
  return manager.currentCycleIndex();
}
```

[N-01] TokemakBribe: `getBribe()` has incorrect description

Line References

TokemakBribe.sol#L188-L194

Description

- Missing `round` param
- `bribeAmount` has incorrect description

Recommended Mitigation Steps

```
    /**
             @notice Get bribe from BribeVault
             @param  proposal            address  Proposal
             @param  round               uint256  Round
             @param  token               address  Token
             @return bribeToken          address  Bribe token addre
             @return bribeAmount         uint256  Bribe token amour
    */
```

[N-02] Emit relevant events in constructor methods when variables are set, or abstract to internal functions

Line References

[BribeVault.sol#L59-L74]

[RewardDistributor.sol#L51-L56]

Description

Some variables are set in the constructor method but do not emit events, unlike their setter counterparts. For instance, `bribeVault` in the `RewardDistributor` contract fails to emit the `SetBribeVault` event, but this is emitted in the `setBribeVault()` function.

Recommended Mitigation Steps

Either emit the events in the constructor, or make the setter functions public and have the constructor call it.

kphed (Redacted Cartel) confirmed and commented:

> Overall, the code quality is great. Inline comments and documentation provided was adequate. Various parties / roles and contract interactions were well explained.

> Thanks for the compliment and the thorough code review! Both are greatly appreciated.

> [L-01] RewardDistributor: Change payable(account).transfer() to .call() for native fund transfers
> [L-02] BribeVault: Use safeTransfer for tokens
> [L-03] RewardDistributor: Limit native fund transfers to bribeVault
> [N-01] TokemakBribe: getBribe() has incorrect description
> [N-02] Emit relevant events in constructor methods when variables are set, or abstract to internal functions

> Thank you, we're planning on implementing all of the above.

> [L-04] TokemakBribe: Sync rounds with Tokemak's manager instead of manually setting rounds via setRound()

> This was an option we've considered but we opted for setting the round manually since our schedule may not always be in lockstep with Tokemak's (e.g. there may be off-chain activities - governance, disputes, etc. - that may result in us taking delayed action). That said, your recommended implementation is a great middle ground solution, thank you for that.

kphed (Redacted Cartel) commented:

> Your comment has inspired a potentially more streamlined solution without needing `round`: using a proposal's deadline to segregate the token deposits for a general time period (i.e. voting round). Thank you!

Alex the Entreprenerd (judge) commented:

> The report is great, gives some general considerations as well as specific advice to implement.
> Great submission.

> Only negative note is the warden missed re-entrancy, beside that, this is how I think a QA report should be done.

> Score: 7/10

Gas Optimizations

For this contest, 18 reports were submitted by wardens detailing gas optimizations. The report highlighted below by warden team **WatchPug** received the top score from the judge.

The following wardens also submitted reports: csanuragjain, 0x1f8b, Jujic, ye0lde, hickuphh3, IIIIIII, pauliax, kenta, robee, gzeon, Omik, rfa, z3s, d4rk, SolidityScan, Tomio, *and* defsec.

[G-01] Adding unchecked directive can save gas

Note: minor optimation, the amount of gas saved is minor, change when you see fit.

For the arithmetic operations that will never over/underflow, using the unchecked directive (Solidity v0.8 has default overflow/underflow checks) can save some gas from the unnecessary internal over/underflow checks.

For example:

ThecosomataETH.sol#L118-L118

[G-02] Using immutable variable can save gas

Note: Suggested optimation, save a decent amount of gas without compromising readability.

TokemakBribe.sol#L28-L28

```
    address public bribeVault;
```

TokemakBribe.sol#L60-L65

```
    constructor(address _bribeVault) {
        require(_bribeVault != address(0), "Invalid bribeVault")
        bribeVault = _bribeVault;

        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
```

```
}
```

Considering that `bribeVault` will never change, changing it to immutable variable instead of storage variable can save gas.

[G-03] Remove redundant access control checks can save gas

Note: suggested optimation, save a decent amount of gas without compromising readability.

[TokemakBribe.sol#L125-L135]

```
function setProposal(address proposal, uint256 deadline)
    public
    onlyAuthorized
{
    require(proposal != address(0), "Invalid proposal");
    require(deadline >= block.timestamp, "Deadline must be i

    proposalDeadlines[proposal] = deadline;

    emit SetProposal(proposal, deadline, _round);
}
```

[TokemakBribe.sol#L142-L157]

```
function setProposals(
    address[] calldata proposals,
    uint256[] calldata deadlines
) external onlyAuthorized {
    require(proposals.length > 0, "Need at least 1 proposal'
    require(
        proposals.length == deadlines.length,
        "Must be equal # of proposals and deadlines"
    );

    for (uint256 i = 0; i < proposals.length; i += 1) {
        setProposal(proposals[i], deadlines[i]);
    }
```

```
        emit SetProposals(proposals, deadlines, _round);
    }
```

`setProposal()` already got `onlyAuthorized` check, and `setProposals()` will check it again multiple times.

Consider creating `_setProposal()` private function without access control and call it inside the public functions.

Recommended Mitigation Steps

Change to:

```
    function _setProposal(address proposal, uint256 deadline)
        private
    {
        require(proposal != address(0), "Invalid proposal");
        require(deadline >= block.timestamp, "Deadline must be i

        proposalDeadlines[proposal] = deadline;
    }

    /**
        @notice Set a single proposal
        @param  proposal  addresss Proposal address
        @param  deadline  uint256  Proposal deadline
     */
    function setProposal(address proposal, uint256 deadline)
        public
        onlyAuthorized
    {
        _setProposal(proposal, deadline);
        emit SetProposal(proposal, deadline, _round);
    }

    /**
        @notice Set multiple proposals
        @param  proposals  address[]  Proposal addresses
        @param  deadlines  uint256[]  Proposal deadlines
     */
    function setProposals(
```

```
        address[] calldata proposals,
        uint256[] calldata deadlines
    ) external onlyAuthorized {
        require(proposals.length > 0, "Need at least 1 proposal'
        require(
            proposals.length == deadlines.length,
            "Must be equal # of proposals and deadlines"
        );

        for (uint256 i = 0; i < proposals.length; i += 1) {
            _setProposal(proposals[i], deadlines[i]);
        }

        emit SetProposals(proposals, deadlines, _round);
    }
```

[G-04] Validation can be done earlier to save gas

Note: suggested optimation, save a decent amount of gas without compromising readability.

Check if `ethLiquidity > 0 && btrflyLiquidity > 0` earlier can avoid unnecessary external call (`IRedactedTreasury(TREASURY).manage(WETH, ethLiquidity);`) when this check failed.

ThecosomataETH.sol#L124-L155

```
    function performUpkeep() external onlyOwner {
        require(checkUpkeep(), "Invalid upkeep state");

        uint256 btrfly = IBTRFLY(BTRFLY).balanceOf(address(this)
        uint256 ethAmount = calculateAmountRequiredForLP(btrfly,
        uint256 ethCap = IERC20(WETH).balanceOf(TREASURY);
        uint256 ethLiquidity = ethCap > ethAmount ? ethAmount :

        // Use BTRFLY balance if remaining capacity is enough, c
        uint256 btrflyLiquidity = ethCap > ethAmount
            ? btrfly
            : calculateAmountRequiredForLP(ethLiquidity, false);

        IRedactedTreasury(TREASURY).manage(WETH, ethLiquidity);
```

```
        // Only complete upkeep only on sufficient amounts
        require(ethLiquidity > 0 && btrflyLiquidity > 0, "Insuff
        // ...
    }
```

Recommended Mitigation Steps

Change to:

```
    function performUpkeep() external onlyOwner {
        require(checkUpkeep(), "Invalid upkeep state");

        uint256 btrfly = IBTRFLY(BTRFLY).balanceOf(address(this)
        uint256 ethAmount = calculateAmountRequiredForLP(btrfly,
        uint256 ethCap = IERC20(WETH).balanceOf(TREASURY);
        uint256 ethLiquidity = ethCap > ethAmount ? ethAmount :

        // Use BTRFLY balance if remaining capacity is enough, c
        uint256 btrflyLiquidity = ethCap > ethAmount
            ? btrfly
            : calculateAmountRequiredForLP(ethLiquidity, false);

        // Only complete upkeep only on sufficient amounts
        require(ethLiquidity > 0 && btrflyLiquidity > 0, "Insuff

        IRedactedTreasury(TREASURY).manage(WETH, ethLiquidity);

        // ...
    }
```

[G-05] `type(uint256).max` is more gas efficient than `2**256 - 1`

Note: minor optimation, the amount of gas saved is minor, change when you see fit.

ThecosomataETH.sol#L68-L69

[G-06] `10e18` is more gas efficient than `10**18`

Note: minor optimation, the amount of gas saved is minor, change when you see fit.

ThecosomataETH.sol#L102-L108

[G-07] Cache array length in for loops can save gas

Note: suggested optimation, save a decent amount of gas without compromising readability.

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Instances include:

TokemakBribe.sol#L147-L152

BribeVault.sol#L261-L275

RewardDistributor.sol#L80-L82

[G-08] Avoid unnecessary storage read can save gas

Note: Suggested optimation, save a decent amount of gas without compromising readability

BribeVault.sol#L213-L248

```
    function depositBribe(
        bytes32 bribeIdentifier,
        bytes32 rewardIdentifier,
        address briber
    ) external payable onlyRole(DEPOSITOR_ROLE) {
        require(bribeIdentifier.length > 0, "Invalid bribeIdenti
        require(rewardIdentifier.length > 0, "Invalid rewardIder
        require(briber != address(0), "Invalid briber");
        require(msg.value > 0, "Value must be greater than 0");

        Bribe storage b = bribes[bribeIdentifier];
        address currentToken = b.token;
        require(
```

```
            // For native tokens, the token address is set to th
            // overwriting storage - the address can be anythinc
            currentToken == address(0) || currentToken == addres
            "Cannot change token"
        );

        b.amount += msg.value; // Allow bribers to increase brik

        // Only set the token address and update the reward-to-k
        if (currentToken == address(0)) {
            b.token = address(this);
            rewardToBribes[rewardIdentifier].push(bribeIdentifie
        }

        emit DepositBribe(
            bribeIdentifier,
            rewardIdentifier,
            b.token,
            msg.value,
            b.amount,
            briber
        );
    }
```

Based on L224~~L230, L235~~L236, we know that `b.token == address(this)`, therefore at L243 `b.token` can be replaced with `address(this)`.

Use `address(this)` directly can avoid unnecessary storage read of `b.token` and save some gas.

Recommended Mitigation Steps

Replace:

```
    emit DepositBribe(
        bribeIdentifier,
        rewardIdentifier,
        b.token,
        msg.value,
        b.amount,
        briber
    );
```

with:

```
    emit DepositBribe(
        bribeIdentifier,
        rewardIdentifier,
        address(this),
        msg.value,
        b.amount,
        briber
    );
```

drahrealm (Redacted Cartel) confirmed and commented:

> Some new gas optimization tricks confirmed 👍

Alex the Entreprenerd (judge) commented:

> Submission is really good.

> Adding the exact gas savings would be the cherry on top.

> Additionally adding a list of all the places in which to apply the optimization would have made this the best finding.

> Pretty good.

Alex the Entreprenerd (judge) commented:

> Best submission 8/10

> To improve:

- Actual Gas Savings math (sort findings by impact)
- List of all spots to fix (So the sponsor can implement instead of it being a puzzle)

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top