b0rder1ess Native Assurance Protocol Audit Report

Version 1.1

Conducted by: Alin Mihai Barbatei (ABA)

March 21, 2024

Table of Contents

1 Introduction

1.1 About ABA

Alin Mihai BARBATEI, known as ABA, is an independent security researcher experienced in Solidity smart contract auditing. Having several solo and team smart contract security reviews, he consistently strives to provide top-quality security auditing services. He also serves as a smart contract auditor at Guardian Audits.

1.2 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

1.3 Risk classification

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

1.3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

1.3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

1.3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

1.3.4 Informational findings

Informational findings will also be included. These encompass recommendations to enhance code style, operations alignment with industry best practices, gas optimizations, adherence to documentation and EIP standards, and overall contract design.

Informational findings typically have minimal impact on code functionality or security risk.

Evaluating all vulnerability types, including informational ones, is crucial for a comprehensive security audit to ensure robustness and reliability.

2 Executive Summary

2.1 b0rder1ess Native Assurance Protocol

The Native Assurance Protocol (NAP) is a solution designed to enhance NFT valuation using liquidity backing, in other words a liquidity framework.

At its core, NAP allows users to exchange `ERC721` and `ERC1155` tokens for the backed `ERC20` class token, and vice versa. This is realized by integrating with Uniswap V2 for liquidity management and leveraging ERC20 tokens for assurance and collateral.

The NAP system core contracts are:

- `NAP721` and `NAP1155`: provide lifecycle management for NFT via activations and deactivations
  - on activation the user deposits LP tokens corresponding to the assurance-collateral pair and receives the backed NFT, of type in accordance with the interacted NAP contract
  - on deactivation, user exchanges his NFT for LP tokens
- `Factory`: sets-up an deploys the NAP contracts
- `Distributor`: distributes funds and fees through the NAP system
- several other peripheral contracts that help maintain the overall balance of the system.

The project will be first deployed on the Polygon PoS blockchain.

More on the protocol can be found on the official documentation page.

2.2 Overview

Project Name	b0rder1ess Native Assurance Protocol
Codebase	b01_testnet
Operating platform	Polygon PoS
Language	Solidity
Audited commits	[1] a307bdbcc881166acf92a7c4537ef2542dff546d [2] df2f19f5cc787800155f655131750b81119226cd [3] fd2905db795352796d24d9affc93a05398926e97 [4] dbd8d1f5dcc8fc4990cf0e78dfb5b9997c857671 [5] 3c032f7357fa92442a68e368830e43b41079cba5
Remediation commit	2e32db899166678b350ef37467a819ad231b0d3d
Audit methodology	Static analysis and manual review

2.3 Scope

Files and folders in scope
- contracts/B01.sol - contracts/BCPStaking.sol - contracts/Distributor.sol - contracts/Executor.sol - contracts/Factory.sol - contracts/MintableERC20.sol - contracts/NAP1155.sol - contracts/NAP721.sol - contracts/PFPMinter.sol - contracts/PFPNFT.sol - contracts/PFPSeasons.sol - contracts/implementations - contracts/interfaces - contracts/libraries

2.4 Issues Found

Severity	Total Found	Resolved	Partially Resolved	Acknowledged
Critical risk	4	4	0	0
High risk	3	3	0	0
Medium risk	8	8	0	0
Low risk	8	5	1	2
Informational	8	4	1	3

2.5 Findings & Resolutions

ID	Title	Severity	Status
C-01	Factory deployed contracts owner permanently blocked	Critical	Resolved
C-02	Anyone can call process rules	Critical	Resolved

ID	Title	Severity	Status
C-03	PFPNFT cannot be transferred, minted or burned	Critical	Resolved
C-04	Activations can be done at half price	Critical	Resolved
H-01	Deactivation discount not limited to B01 token	High	Resolved
H-02	Rule update collected token mechanism is broken	High	Resolved
H-03	Rule execution blocked due to incorrectly calculated amount	High	Resolved
M-01	Deactivation profit does not take B01 discount into account	Medium	Resolved
M-02	Deactivation profit does not account for liquidity removal	Medium	Resolved
M-03	Minimum profit check done before fees are taken	Medium	Resolved
M-04	Liquidity from only collateral cannot be added through Distributor	Medium	Resolved
M-05	Distributor swapV2 function fails because of allowance	Medium	Resolved
M-06	Anyone can add token IDs for a season	Medium	Resolved
M-07	Collateral for NAP, Distributor and Factory can differ	Medium	Resolved
M-08	Distributor can DoS B01 transfers	Medium	Resolved
L-01	NAP router upgradeFee has inconsistent logic	Low	Resolved
L-02	Treasury fee percent over 90% blocks withdrawals	Low	Resolved
L-03	Distributor rules can have allocated more them 100% amount	Low	Resolved
L-04	Inadequate season timeline checks	Low	Acknowledged
L-05	Default B01 pool fee of 100%	Low	Resolved
L-06	Seasons with 0 NFTs can be created	Low	Acknowledged
L-07	Incorrect amountInMax on Uniswap swap in activations	Low	Resolved
L-08	Dust token amounts may remain after activations	Low	Partially Resolved
I-01	Do not use _msgSender if meta transactions are not supported	Informational	Resolved
I-02	Add contingencies for emergency situations	Informational	Partially Resolved
I-03	Not all season NFT will reveal at the same time	Informational	Acknowledged
I-04	newERC20 can be a local variable	Informational	Resolved

ID	Title	Severity	Status
I-05	Use SafeERC20.forceApprove instead of custom implementation	Informational	Resolved
I-06	Redundant liquidity estimations	Informational	Acknowledged
I-07	Misleading UniswapPairCreated event	Informational	Resolved
I-08	generateRandomInRange returns pseudo-random values	Informational	Acknowledged

3 Findings

3.1 Critical Risk

3.1.1 Factory deployed contracts owner permanently blocked

Severity: *Critical risk (Resolved)*

Context: [⟲1] : ***Factory.sol***:*180-185,191-196*

Description

When the `Factory` contract deploys a `Distributor` or any NAP contract variations (`NAP721` or `NAP1155`) via the `deployNAPAndDistributor` function, it automatically is set as the owner of the distributor.

This is an issue as the ownership must be changed to the rightful owner. As it is, the `Factory` contract is not upgradable or able to call the transferOwnership function. As such the entire Distributor for the newly deployed contract is bricked as all operations are gated by an `onlyOwner` modifier and all newly deployed NAP contracts have all sensitive operations, requiring owner action, blocked..

Recommendation

In the `deployNAPAndDistributor` function, after any `_deploy` call to create the distributor or NAP contract has been executed, transfer ownership to the **msg.sender**. Another solution is to change the contracts constructors to have the owner passed, instead of being set to the **msg.sender**.

Resolution: Resolved. The recommended fix was implemented in **6fd94a0**.

3.1.2 Anyone can call process rules

Severity: *Critical risk (Resolved)*

Context: [⟲1] : ***Distributor.sol***:*64*

Description

In the `Distributor` contract, only the owner can add rules but anyone can initiate the execution of all the set rules. Distributor can accumulate a variety of tokens over time, and the owner may wish to add liquidity with them, or transfer them to a different source.

If any of the set rules contains for example the collateral token, anyone can always call them and remove any collateral from the contract, even though the intention of the owner would of been to add them as liquidity if they amounted to a specific sum.

Also, if any burn token rule is present a user may call it until the full amount is burned, even if the intent of the rule was to burn only 10%, by calling it multiple times that can reach to near 100%.

Recommendation

Add access control on the `process` function to only be allowed to be called by the owner of the contract. Another solution is to create a separate role mechanism for allowed addresses to be able to call that function.

Resolution: Resolved. The recommended fix was implemented in **fd2905d** and **dbd8d1f**.

3.1.3 PFPNFT cannot be transferred, minted or burned

Severity: *Critical risk (Resolved)*

Context: [⊶2] : ***PFPNFT.sol***:189-194

Description

The `PFPNFT` contract overrides the `ERC721EnumerableUpgradeable` and `ERC721Upgradeable` contracts `_update` function without providing an implementation or passing execution to the already inherited code. This results in blocking all minting, transferring and burning of the PFP NFT.

Recommendation

Add a call to the inherited code via `super._update`.

```
    function _update(
        address to,
        uint256 tokenId,
        address auth
-    ) internal virtual override(ERC721EnumerableUpgradeable, ERC721Upgradeable)
    returns (address) {}
+    ) internal virtual override(ERC721EnumerableUpgradeable, ERC721Upgradeable)
    returns (address) {
+        return super._update(to, tokenId, auth);
+    }
```

Resolution: Resolved. The recommended fix was implemented in **dbd8d1f**.

3.1.4 Activations can be done at half price

Severity: *Critical risk (Resolved)*

Context: [⊶3] : ***NAP721.sol***:548-552 ***NAP1155.sol***:579-582

Description

The activate functions of the NAP contracts require users passing in which token they are passing in order to get the NFTs. A critical issue that exists is that the `tokenIn` address is not validated to be the assurance or collateral token.

```
    address token0 = lpAssurance.token0();
    address token1 = lpAssurance.token1();

    path[0] = tokenIn;
    path[1] = (tokenIn == token0) ? token1 : token0;
    uint256 amountInA = (tokenIn == token0) ? amount0 : amount1;
    uint256 amountInB = (tokenIn == token0) ? amount1 : amount0;
    uint256 totalAmountInA = amountInA + uniswapRouter.getAmountsIn(amountInB, path)
        [0];
```

The code presume that if the token is not the known first, then it must be by default the second.

An ill intended user can pass a newly created random token which was paired with the legitimate `token0` LP token and pass it to activation, resulting in an approximate half price of the NFT since

while the LP tokens amount is compared `require(liquidity >= (totalLiquidityAmount * 999)/ 1000, "Insufficient liquidity added");` the comparison in this case compares the LP received from adding to the newly created pool versus the liquidity calculated based on the NAP correct LP tokens.

Recommendation

Validate that the input token is one of token pair compromising the NAP LP pair. A solution ca be moving the validation logic from the getDeactivationProfit and reuse it here.

Another suggestion is to simply add a check `require(tokenIn == token0 || tokenIn == token1);`

Resolution: Resolved. The recommended fix was implemented in **dbd8d1f**.

3.2 High Risk

3.2.1 Deactivation discount not limited to B01 token

Severity: *High risk (Resolved)*

Context: [1] : ***NAP721.sol***:*215-217* ***NAP1155.sol***:*252-254*

Description

When deploying a NAP using the `Factory` contract, the address of the `B01` token is passed by the caller. Within the NAP721 and NAP1155 contracts, there is a deactivation discount mechanism that relies on caller owing a specific amount of tokens (threshold) and if so, to have the treasury fee deduced.

The current discount mechanism allows any token to be sent but presumes that it has 18 decimals.

```
    uint256 callerB01Balance = b01Token.balanceOf(msg.sender);
    uint256 threshold = 100 * (10 ** 18);
```

If the `B01` token is not passed and a different token is used, that does not have 18 decimals, then the user would either need significantly more or less then 100 units.

The underlying issue is that any token can be used, since currently there is no way to constrain that the `b01Token` token is actually the B01 b0rder1ess token. This results in loss of incentive for the original B01 token and if tokens with different decimals then 18 are used, users do not benefit or over benefit from the discount.

Recommendation

If this is intentional, use the `b01Token.decimals()` function to correctly get any decimals.

If this is not intentional, in the NAP contracts, create a function that sets the `b01Token`, callable only once (require `b01Token` to be zero address to be called), callable by owner, which is called from the `Factory` contract, before giving the ownership to the caller. The `Factory` contract will hold the original `b01Token` address.

Also consider creating a setter for the threshold amount which is callable only by the owner of the`B01` token. Limiting this way allows the team control over the discount. Currently it is hardcoded to 100 but economical factors may lead to it not being appropriate.

Resolution: Resolved. The recommended fix was implemented in **6fd94a0** and **dbd8d1f**.

3.2.2 Rule update collected token mechanism is broken

Severity: *High risk (Resolved)*

Context: [1] : ***Distributor.sol***:276-281

Description

In the `Distributor` contract, the owner can have a set of rules which he executes. These rules are token related and can transfer or burn tokens. Either operations reduce the current balance of those tokens.

Whenever a rule is executed, a call to `_updateCollectedToken` is done. Here, incorrectly, a new token amount is calculated as the difference between the last snapshot and the current balance.

```
    function _updateCollectedToken(address token) private {
        uint256 balance = ERC20(token).balanceOf(address(this));
        uint256 newTokenAmount = balance - _collectedTokensAmounts[token];
        _collectedTokensAmounts[token] = balance;
        _indexesAmounts[token] += newTokenAmount;
    }
```

Because the balance decreases on any rule action, the subtraction `balance - _collectedTokensAmounts[token]` will underflow for any operation that is not done when the current balance is higher then the old balance.

Example scenario:

- a rule is set to transfer 50% tokens
- at 1100 token balance it is called. `_collectedTokensAmounts[token]` is set to 1100 then the transfer happens
- after transfer, balance is 550
- now, up until the contract holds 1100 tokens, no rule can be executed because of the revert.

Recommendation

Rethink the underlying collected token mechanism and if it is needed. At a minimum consider the delta in variation, positive or negative, and take into account the possibility of a 0 balance.

Resolution: Resolved. The recommended fix was implemented in **dbd8d1f**.

3.2.3 Rule execution blocked due to incorrectly calculated amount

Severity: *High risk (Resolved)*

Context: [1] : ***Distributor.sol***:74

Description

In the `Distributor` contract, when a rule is executed, it works with a specific amount. Each rule has a preset percentage and the amount that it can work with is calculated using that percentage.

Currently the amount is incorrectly calculate because of mistakenly reversing the rule percentage with the 100% equivalent (100000):

```
    uint256 amount = (_indexesAmounts[rule.token] * 100000) / rule.percentage;
```

With the exception when a rule has 100% and the amount is the exact `_indexesAmounts` (witch on first execution is the entire balance), this results in the rule execution reverting due to attempting to transfer or burn more tokens then owned.

Recommendation

Change the way the amount is calculated to:

```
    uint256 amount = _indexesAmounts[rule.token] * rule.percentage / 100_000;
```

Resolution: Resolved. The recommended fix was implemented in **fd2905d**.

3.3 Medium Risk

3.3.1 Deactivation profit does not take B01 discount into account

Severity: *Medium risk (Resolved)*

Context: [1] : ***NAP721.sol***:*451-452* ***NAP1155.sol***:*469-470*

Description

Calculating the received deactivation profit via `getDeactivationProfit` returns less then intended. This amount is crucial because the it is passed as the minimum deactivation profit when deactivating, acting as a slippage protection against standing attacks and reverting the transaction if this minimum is not achieved.

The `getDeactivationProfit` function logic currently does not take into consideration the discount users get if they hold the B01 token over a specific threshold.

This results in an under estimated profit amount which users will use to estimate their gains and call the deactivation function with. This small difference also leaves an opportunity for sandwich attacks. If the treasury fee percentage, which is not actually deduced, to be gained by a MEV bot is enough to offset fees, then users will lose funds.

Recommendation

Consider the treasury fee only if the user does not have the require B01 balance.

Resolution: Resolved. The recommended fix was implemented in **dbd8d1f**.

3.3.2 Deactivation profit does not account for liquidity removal

Severity: *Medium risk (Resolved)*

Context: [1] : ***NAP721.sol***:*449,200* ***NAP1155.sol***:*467,222*

Description

Users wanting to deactivate their NFTs and get a profit will call the NAP `getDeactivationProfit` function to have an estimate before calling the deactivate function. This is crucial because the amount is

passed as the minimum deactivation profit when deactivating, acting as a slippage protection against standing attacks and reverting the transaction if this minimum is not achieved.

The `getDeactivationProfit` function returns an incorrect amount, greater then the actual amount calculated and returned by the `_removeLiquidityAndSwap` function. The underlying issue is that the function correctly simulates a liquidity withdrawal but when it then calculates the amount of token received from converting one of the tokens from the removed liquidity is uses `UniswapV2Router02.getAmountsOut` function. This function simulates a token swap while taking into consideration the current LP token reserves, but in the `getDeactivationProfit` simulation, we need to account for the fact that we already simulated the liquidity removal.

This results in the `getAmountsOut` call to over estimate how much tokens are received. The issues is currently mitigated for small deactivations by manually reducing the deactivationProfitAmountMin.

```
        if (deactivationProfitAmount < (deactivationProfitAmountMin * 997) / 1000) {
```

This workaround will fail if a high enough number of liquidity, respectively LP, is removed from the LP pool. As such, for large deactivations, which offset the LP balance and cause a deviation more then 0.3% (as hardcoded by the `* 997/1000`) users relying on `getDeactivationProfit` will have their transaction revert.

Recommendation

Implement a custom `getAmountsOut` that simulates token swap while also taking into consideration any simulated liquidity withdrawal.

Afterwards, when deactivating tokens and comparing the profit, remove the `* 997/1000` workaround.

Example implementation:

```
diff --git a/contracts/NAP721.sol b/contracts/NAP721.sol
index b564472..1cdf26e 100644
--- a/contracts/NAP721.sol
+++ b/contracts/NAP721.sol
@@ -446,8 +446,9 @@ contract NAP721 is Ownable(msg.sender), INAP721, ReentrancyGuard
     {
             revert("Invalid tokenOut");
         }

-        uint256[] memory amountsOut = uniswapRouter.getAmountsOut(amountIn, path);
-        amountOut = initialAmountOut + amountsOut[amountsOut.length - 1]; /// Apply
     the 0.3% Uniswap commission fee
+        // Apply the 0.3% Uniswap commission fee and take into consideration
    removed liquidity
+        amountOut = initialAmountOut + _liquidityRemovedAwareGetAmountsOut(amountIn
    , token0Amount, token1Amount);
+
         uint256 deactivationFee = (amountOut * (deactivationFeePercentage +
             treasuryFeePercentage)) /
             DENOMINATOR;
         uint256 deactivationAmount = amountOut - deactivationFee;
@@ -455,6 +456,20 @@ contract NAP721 is Ownable(msg.sender), INAP721,
    ReentrancyGuard {
         return amountOut;
     }
```

```
+    function _liquidityRemovedAwareGetAmountsOut(
+        uint amountIn,
+        uint256 removedFromReserveIn,
+        uint256 removedFromReserveOut
+    ) internal view returns (uint256) {
+
+        (uint reserveIn, uint reserveOut, ) = lpAssurance.getReserves();
+
+        return uniswapRouter.getAmountOut(
+            amountIn,
+            reserveIn - removedFromReserveIn,
+            reserveOut - removedFromReserveOut);
+    }
+
     /**
      * @dev Calculates the required amounts of tokens for activating NFTs based on
         the current price.
     * This function calculates how much of the ‘tokenIn‘ is needed to activate a
         given number of NFTs,
```

Resolution: Resolved. The recommended fix was implemented in **f28b3d0**.

3.3.3 Minimum profit check done before fees are taken

Severity: *Medium risk (Resolved)*

Context: [1] : ***NAP721.sol***:200-206 ***NAP1155.sol***:222-229

Description

When a deactivation happens, user passes in a minimum profit variable, which functions as a slippage mechanism protecting from sandwich attacks or abrupt price fluctuations.

Users wanting to deactivate their NFTs and get a profit will call the NAP `getDeactivationProfit` function to have an estimate before calling the deactivate function. The `getDeactivationProfit` function, correctly takes into consideration fees.

The deactivations themselves incorrectly compare the profit to the amount before fees, not after.

```
        if (deactivationProfitAmount < (deactivationProfitAmountMin * 997) / 1000) {
            revert DeactivationProfitAmountLessThanMin(
                deactivationProfitAmount,
                deactivationProfitAmountMin
            );
        }
        deactivationAmount = _processDeactivation(deactivationProfitAmount, tokenOut
            , to);
```

Since users get the `deactivationAmount`, not the `deactivationProfitAmount` amount, this results in users getting less then the indicated minimum amount.

Recommendation

Move the `_processDeactivation` call before the minimum profit check and check the `deactivationProfitAmountMin` against the `deactivationAmount`.

Resolution: Resolved. The recommended fix was implemented in **fd2905d**.

3.3.4 Liquidity from only collateral cannot be added through Distributor

Severity: *Medium risk (Resolved)*

Context: [⟲1] : ***Distributor.sol***:*119-129*

Description

In the `Distributor` contract, the function `addLiquidityToV2Pool` allows the owner to add liquidity to a collateral token - ERC20 pair. It facilitates the option to not pass any other token then the collateral one and use that for liquidity adding.

The issue is that the function adds a checks that the collateral balance after the token swap must be greater then the one before the swap, but the swap is not done if the passed token is equal to the collateral token, resulting in a revert and the inability to purely call the function with the collateral token.

```
    /// Swap tokenIn for collateralToken if they are not the same
    if (tokenIn != collateralToken) {
        _safeApproveAndSwap(tokenIn, collateralToken, amountIn, minAmountOut,
            uniswapDeadline);
    }

    /// Calculate new collateral balance and validate increase
    uint256 newCollateralBalance = IERC20(collateralToken).balanceOf(address(this));
    require(
        newCollateralBalance > initialCollateralBalance,
        "Insufficient collateral token balance after swap"
    );
```

Recommendation

Do not check the balance increase if the token in is the collateral token.

Resolution: Resolved. The recommended fix was implemented in **fd2905d**.

3.3.5 Distributor swapV2 function fails because of allowance

Severity: *Medium risk (Resolved)*

Context: [⟲1] : ***Distributor.sol***:*156-174*

Description

The `Distributor` contract supplies a `swapV2` function to allow the owner to swap contract tokens to other tokens. This function will always fail because it does not give an allowance to the passed router contract. Swapping within the distributor contract can still be done using the `swapOrTransferToken` function but in a limited manner (has parameters constraints).

Recommendation

Use the `SafeERC20.forceApprove` function to approve the `amount` for the provided router in the `swapV2` function.

Resolution: Resolved. The recommended fix was implemented in **fd2905d**.

3.3.6 Anyone can add token IDs for a season

Severity: *Medium risk (Resolved)*

Context: [-o-2] : ***PFPNFT.sol***:*61-64*

Description

In the `PFPNFT ERC721` contract, the `addTokenIds` is used to populate the list of unminted tokens from which minting will select a random subset. Calling it works until all season allocated tokens are populated into the unminted token list.

The `addTokenIds` function lacks access control, allowing anyone to call it. Because of this, sub-batching minting into smaller groups is no longer feasible. Say the team initially wanted to allow the minting of 10% of tokens from a season and called the `addTokenIds` with that amount. Since anyone can call it, it would instantly cap to the maximum allowed season minting cap.

Recommendation

If this is intended then document the behavior, otherwise allow only the owner to add token IDs by adding the `onlyOwner` modifier on the `addTokenIds` function.

Resolution: Resolved. Fix was implemented in **dbd8d1f**.

Issue was also found by the team during testing

3.3.7 Collateral for NAP, Distributor and Factory can differ

Severity: *Medium risk (Resolved)*

Context: [-o-3] : ***Factory.sol***:*175-243*

Description

When a new NAP and Distributor is deployed via the `Factory` contract, there is no check that the passed collateral token to the 2 contracts is the same as the one currently set in the `Factory`. If a different token is used, the send LP from the factory is lost as it is.

Recommendation

Either validate that the `napParams.napConstructorParams._collateralToken` and `distributorParams.distributorConstructorParams._collateralToken` address are identical to the `collateralTokenAddress` or simply enforce it.

Resolution: Resolved. The recommended fix was implemented in **dbd8d1f**.

3.3.8 Distributor can DoS B01 transfers

Severity: *Medium risk (Resolved)*

Context: [-o-4] : ***B01.sol***:79-81

Description

With the current implementation, whenever a `B01` transfer of an amount greater then `DISTRIBUTOR_HOOK_THRESHOLD` is done, then the distributor `process` function is called. The function executes the rules set by the distributor owner.

There are several issues with this approach that can lead to transfers reverting:

- if the `Distributor` is frozen, then the `B01` transfer reverts and users would need to transfer a lower amount or wait until the team changes the distributor
- if there are significant rules in the `distributor` the transaction may OOG (out of gas) or add an unfair gas cost to users transfers.
- the `DISTRIBUTOR_HOOK_THRESHOLD` value is hardcoded and may be inappropriate in relation to future market conditions

Recommendation

Move the `Distributor.process` call outside of the `_update` function into it's own dedicated function. Allow this function to be called by whitelisted contracts, similar to using a Keeper type logic, where an off-chain logic determines if the `process` function should be called and calls it (e.g. at regular intervals). Also consider if there should be the possibility of calling an arbitrary distributor via the same mechanism.

At a minimum, guard the call to the `Distributor.process` function with a low-level `.call` and if it is not successful emit an event to be able to be notified and investigate if the situations arise.

Resolution: Resolved. The recommended fix was implemented in **3c032f7**.

3.4 Low Risk

3.4.1 NAP router upgradeFee has inconsistent logic

Severity: *Low risk (Resolved)*

Context: [-o-1] : ***RouterNAP.sol***:38-47

Description

The `upgradeFee` function from the `RouterNAP` contract implements an if-else statement logic that has the same outcome

```
        if (executeFeeAmount == 0) {
            executeFeeAmount += amount;
        } else {
            executeFeeAmount = amount;
        }

        require(executeFeeAmount == amount, "createNewFee");
```

- if the `executeFeeAmount` is 0, then to it we add the amount, thus becoming `amount`
- if the `executeFeeAmount` is not 0, then it becomes `amount`

Recommendation

Implement the initially intended behavior or simply the function as:

```
    function upgradeFee(uint8 amount) public onlyOwner {
        executeFeeAmount = amount;
        emit ChangedFee(executeFeeAmount);
    }
```

Also add validations/constraints if needed.

Resolution: Resolved. The recommended fix was implemented in **fd2905d**.

3.4.2 Treasury fee percent over 90% blocks withdrawals

Severity: *Low risk (Resolved)*

Context: [-o-1] : ***NAP721.sol***:236-238 ***NAP1155.sol***:347-349

Description

When a deactivation is initiated, a user can be a deactivation fee and a treasury fee. The deactivation fee is limited to 10% of the profit amount but the treasury fee has no limit.

Setting a treasury fee percent of over 90% will result in deactivations being blocked due to the underflow when deducing the fees

```
deactivationAmount = _deactivationProfitAmount - treasuryFee - feeRecipientFee;
```

Recommendation

Change the `setTreasuryFeePercentage` function to not allow any value over 9000 (90%). Consider also limiting it to a lower value.

Resolution: Resolved. The recommended fix was implemented in **dbd8d1f**.

3.4.3 Distributor rules can have allocated more them 100% amount

Severity: *Low risk (Resolved)*

Context: [-o-1] : ***Distributor.sol***:88

Description

When a rule is set in the `Distributor` it can be set with more then 100% percent, which will cause a revert in rule execution.

```
    require(newRules[i].percentage > 0, "Invalid percentage"); /// Validate
        percentage
```

Recommendation

Also validated that rule percentage is no more then 100% equivalent.

Resolution: Resolved. Fix was implemented in **2be2de5**.

2be2de5ea23bdf7ac0fa434b85aabe39dd4bddde

3.4.4 Inadequate season timeline checks

Severity: *Low risk (Acknowledged)*

Context: [⟲2] : ***PFPSeasons.sol***:*56-83*

Description

When a PFP minting season is created, there are no checks that:

- the seasons does not start or end in the past
- the end date can be set before the start date
- there are no overlapping seasons

Overlapping seasons may lead to external integrators that rely on `getCurrentSeason` to have incorrect results, as the function finds the first season where the `block.timestamp` is within its timeline.

Recommendation

In the `PFPSeasons.createSeason` function, add each new season ID to a seasons list and before creating it, check that the current season `startAt` time is greater (not equal) to the `endAt` of the previous element in the list.

Also consider not allowing any of the dates to be in the past and validated that the end date is after the start date.

Resolution: Acknowledged by the team.

team: this is the functionality we want for our business logic

3.4.5 Default B01 pool fee of 100%

Severity: *Low risk (Resolved)*

Context: [⟲3] : ***B01.sol***:*26*

Description

The `B01` token can apply a tax on transfers for specific addresses. This is intended as a tax on sales.

The current default of this tax is 100% which will lead to a complete loss of funds for the marked address.

Recommendation

Change the default value to less then 100%, e.g. 5%. Also consider using basis points for managing the tax. As it is, (from 1 to 100) it is impossible to have less then 1% fee granularity, example setting a 1.5% fee or a 0.5% fee.

Resolution: Resolved. The recommended fix was implemented in **3c032f7**.

3.4.6 Seasons with 0 NFTs can be created

Severity: *Low risk (Acknowledged)*

Context: [-o-3] : ***PFPSeasons.sol***:*56-83*

Description

Current PFP seasons implementation allows the infinite creation of seasons with 0 NFT as amounts. This can happen even if the maximum of 10,000 have been allocated, seasons can still be created.

A season with 0 NFT amount does not impact system as it is but this situation should be avoided.

Recommendation

Validate that the `tokensAmount` which is passed to the `createSeason` function is different then 0.

Resolution: Acknowledged by the team.

3.4.7 Incorrect amountInMax on Uniswap swap in activations

Severity: *Low risk (Resolved)*

Context: [-o-5] : ***NAP721.sol***:*136* ***NAP1155.sol***:*193*

Description

When an activation is done, only one token is used for paying the required liquidity. An estimation `totalAmountInA` is determine which contains both the amount needed to be swapped into the pairing token for the LP adding and the original token amount.

This amount is incorrectly used as the amountInMax on the Uniswap `swapTokensForExactTokens` execution call

```
        uniswapRouter.swapTokensForExactTokens(
            data.amountInB,
            data.totalAmountInA,
            data.path,
            address(this),
            uniswapDeadline
        );

        _addLiquidity(data.amountInA, data.amountInB, data.path, uniswapDeadline);
```

The `data.totalAmountInA` is contains both the `amountInA` and the equivalent amount of token A to be swapped to get the `amountInB`

```
uint256 totalAmountInA = amountInA + uniswapRouter.getAmountsIn(amountInB, path)[0];
```

As such, the maximum that can be used from it is `data.totalAmountInA - data.amountInA`, since any more then that will make the add liquidity call in `_addLiquidity` revert.

Recommendation

In activations, in the `swapTokensForExactTokens` call, change the `amountInMax` argument to `data.totalAmountInA - data.amountInA` instead of `data.totalAmountInA`.

The situation where more then the difference is taken should not appear implicitly because the exact amounts are calculated before calling the swap.

Resolution: Resolved. Fix was implemented in **2be2de5**.

2be2de5ea23bdf7ac0fa434b85aabe39dd4bddde

3.4.8 Dust token amounts may remain after activations

Severity: *Low risk (Partially Resolved)*

Context: [⊶5] : ***NAP721.sol***:116-150 ***NAP1155.sol***:162-213

Description

When an NAP activation is done, liquidity is added to a Uniswap V2 pool. The exact amounts are calculated on-chain to ensure it is as optimal as possible.

When liquidity is added to a Uniswap V2 pool, the optimal token amount is calculated for each token, while taking into consideration the imposed constraints. In some cases, it may result in not the fully allowed/passed token amount to be consumed for both tokens.

Because of how adding liquidity works in activations, after activations there may remain dust amounts of tokens within the NAP contracts.

The issue is also present in the `Factory` contract when deploying a new NAP contract.

Recommendation

At the end of activations or factory NAP deployment, check if there is a remaining token balance on the contract and if so, send it back to the caller.

Example of function to send back any remaining dust:

```
    function _sendBackDust(address to) internal {
        IERC20 token0 = IERC20(lpAssurance.token0());
        IERC20 token1 = IERC20(lpAssurance.token1());

        uint256 balance0 = token0.balanceOf(address(this));
        if (balance0 > 0) {
            token0.safeTransfer(to, balance0);
        }

        uint256 balance1 = token1.balanceOf(address(this));
        if (balance1 > 0) {
            token1.safeTransfer(to, balance1);
        }
    }
```

The factory contract, because of a different reason, already sends the assurance token balance back. Implement the same logic for the collateral token also.

Resolution: Partially resolved by the team in **2be2de5**

Issue was resolved only for the Factory contract.

3.5 Informational Findings

3.5.1 Do not use _msgSender if meta transactions are not supported

Severity: *Informational (Resolved)*

Context: [⊸1] : ***B01.sol***:58

Description

The use of the `_msgSender` sender instead of the `msg.sender` is specifically related to supporting meta transactions. Meta transactions are also referred to as gasless transactions where not the caller pays the gas fee but third party. Services like Biconomy offer this.

The `burn` function from the `B01` contract takes the sender from the `_msgSender` function. This is redundant and should be changed to `msg.sender`.

Recommendation

Change the `_msgSender` call to the direct `msg.sender`.

Resolution: Resolved. The recommended fix was implemented in **6fd94a0**.

3.5.2 Add contingencies for emergency situations

Severity: *Informational (Partially Resolved)*

Context: Global

Description

In case of unforeseen issues with the contract, exploits or any other negative scenario, crisis mitigation mechanisms must exist.

Recommendation

- Add pause/unpausing mechanisms for activations and deactivations. Deactivations can already be fully blocked by the owner by setting the deactivation count to 0. This mechanism is a straightforward way and it only adds further blocking on activations, which do not benefit users to be executed. The owner is considered trusted.
- Add an emergency deactivation mechanism for users. This should be usable only in an emergency and should bypass as much logic as it can from the normal execution flow. An example implementation for the `NAP721` contract:

```
    function emergencyDeactivation(
        uint256[] calldata tokenIds,
        address receiver
    )
        external nonReentrant onlyInAnEmergency
    {
        require(receiver != address(0), "receiver can't be zero address");
        uint256 tokenCount = tokenIds.length;
        require(tokenCount != 0, "no ids were passed");
```

```
        // do not use safeTransferFrom since this function should not have any means
            of reverting
        // once an emergency situation is initiated by the team and do not send them
            to address(this)
        // since the contract might be in an undesired state
        for (uint256 i = 0; i < tokenCount; i++) {
            IERC721Enumerable(nft).transferFrom(
                msg.sender,
                owner(),
                tokenIds[i]
            );
        }
        uint256 totalOwnedLiquidity = tokenCount * getNFTPrice();
        lpAssurance.safeTransfer(msg.sender, totalOwnedLiquidity);

        emit EmergencyDeactivation(msg.sender, tokenIds, receiver,
            totalOwnedLiquidity);
    }
```

- Add an `emergencyWithdraw` for any possible stray token that reached the contract, or even stuck tokens from the the deactivations/activations. Normally the NAP contracts should hold only LP tokens, so those must not be allowed to be withdrawn to protect users against abuse. Example implementation for the `NAP721` contract:

```
    function emergencyWithdraw(address asset, address receiver) external onlyOwner {
        require(asset != address(lpAssurance), "asset can't be LP token");
        require(asset != address(0), "asset token invalid");
        require(receiver != address(0), "receiver can't be zero address");

        uint256 amount = IERC20(asset).balanceOf(address(this));

        IERC20(asset).safeTransfer(receiver, amount);
        emit EmergencyWithdraw(msg.sender, receiver, amount);
    }
```

Resolution: Partially resolved by the team in **2be2de5**

ABA: the team implemented only the emergency withdraw mechanism

team: partially done since we are still discussing pausable condition, otherwise new owner can block the functions or create rugpull scenario for NFT's and LP tokens and we don't want that

3.5.3 Not all season NFT will reveal at the same time

Severity: *Informational (Acknowledged)*

Context: [⊶2] : ***PFPNFT.sol***:95

Description

The reveal date for minted PFP NFTs is set to 2 weeks since the time they are minted. This behavior is uncommon and coupled with allowed batch minting via `addTokenIds` may lead to situations where not all tokens are minted, the season is still running and some tokens are revealing while others are not minted.

Recommendation

Clearly document this behavior or implement a fixed reveal time for the entire season (e.g. 2 weeks from season end).

Resolution: Acknowledged by the team.

team: this is the functionality we want for our business logic

3.5.4 newERC20 can be a local variable

Severity: *Informational (Resolved)*

Context: [-o-3] : ***Factory.sol***:*78*

Description

In the `Factory` contract, the address of the assurance token is passed in a storage variable `newERC20`. This storage variable is not used outside of the `deployNAPAndDistributor` function, as such it can be set to a local variable (**`memory`**) within the function to save gas.

Recommendation

Change the `newERC20` from a storage variable to a local variable.

Resolution: Resolved. The recommended fix was implemented in **2be2de5**.

3.5.5 Use SafeERC20.forceApprove instead of custom implementation

Severity: *Informational (Resolved)*

Context: [-o-3] : ***Distributor.sol***:*197,222-223,237-247,266*

Description

In the `Distributor` contract there is a `_safeApprove` function which is used to odd allowance cases and approve to the Uniswap router. This function is redundant and can be changed with existing library code.

Recommendation

Remove the `_safeApprove` function and replace the calls to it in with calls to `SafeERC20.forceApprove`, example:

```
ERC20(token).forceApprove(uniswapRouter, amount);
```

Resolution: Resolved. The recommended fix was implemented in **dbd8d1f**.

3.5.6 Redundant liquidity estimations

Severity: *Informational (Acknowledged)*

Context: [-o-4] : ***NAP1155.sol***:*206-208,321-326,494-507,544-552,577* ***NAP721.sol***:*148,322-327,498-511,518-526,558*

Description

Within the NAP contracts there are various functions used to estimate operations working with Uniswap V2 liquidity. The `_getAddLiquidityIn`, `_calculateTokensForLiquidity` and initial stub from the `calculateAmountOut` have particular unnecessary operations that overly complicate the implementation.

In particular, the use of the LP balance of the NAP contract is redundant and mathematically it is simplified in the end case.

The `_getAddLiquidityIn` can be rewritten as the following completely removing the `_calculateTokensForLiquidity` call with no side effect as:

```
    function _getAddLiquidityIn(
        uint256 liquidityOut
    ) internal view returns (uint256 amount0, uint256 amount1) {
        uint256 totalSupply = lpAssurance.totalSupply();
        (uint112 reserve0, uint112 reserve1, ) = lpAssurance.getReserves();

        amount0 = ((reserve0 * liquidityOut / totalSupply) * 1001) / 1000;
        amount1 = ((reserve1 * liquidityOut / totalSupply) * 1001) / 1000;
    }
```

Also, the first part of the `calculateAmountOut` function, respectively:

```
    function calculateAmountOut(
        uint256 totalLiquidityAmount,
        address tokenOut
    ) internal view returns (uint256 amountOut) {
        (uint256 reserve0, uint256 reserve1) = _calculateTokensForLiquidity();
        uint256 initialAmountOut;
        uint256 amountIn;
        uint256 lpTotalSupply = lpAssurance.balanceOf(address(this));
        uint256 token0Amount = (totalLiquidityAmount * reserve0) / lpTotalSupply;
        uint256 token1Amount = (totalLiquidityAmount * reserve1) / lpTotalSupply;
```

can be rewritten as:

```
    function calculateAmountOut(
        uint256 totalLiquidityAmount,
        address tokenOut
    ) internal view returns (uint256 amountOut) {
        uint256 totalSupply = lpAssurance.totalSupply();
        (uint112 reserve0, uint112 reserve1, ) = lpAssurance.getReserves();

        uint256 token0Amount = (reserve0 * totalLiquidityAmount / totalSupply;
        uint256 token1Amount = (reserve1 * totalLiquidityAmount / totalSupply;
```

Which is represents the logic as the `_getAddLiquidityIn` without the `* 1001 / 1000` increase. The specific increase in the `_getAddLiquidityIn` was introduced as an artificial workaround to compensate that `calculateAmountOut` returns a lesser amount than liquidity needed in practice returns.

In both cases `_calculateTokensForLiquidity` and subsequently the dependency on NAP contract balance is not needed.

Recommendation

Remove the `_calculateTokensForLiquidity` function, create a `_liquidityToTokens` function with the implementation:

```
    function _liquidityToTokens(
        uint256 liquidityAmount
    ) internal view returns (uint256 amount0, uint256 amount1) {
        uint256 totalSupply = lpAssurance.totalSupply();
        (uint112 reserve0, uint112 reserve1, ) = lpAssurance.getReserves();

        amount0 = reserve0 * liquidityAmount / totalSupply;
        amount1 = reserve1 * liquidityAmount / totalSupply;
    }
```

Use this function in `_getActivationData` in place of `_getAddLiquidityIn` and in `calculateAmountOut` remove the entire `token0Amount` / `token1Amount` calculations and replace it with a call to `_liquidityToTokens`.

`_liquidityToTokens` works for estimating in both remove and adding liquidity since the Uniswap V2 burn/mint LP operations evaluate tokens the equivalently same.

Since this removes the `* 1001 / 1000` increase in estimated liquidity, the workaround in the activation function `require(liquidity >= (totalLiquidityAmount * 999)/ 1000, "Insufficient liquidity added");` should be lowered even more to compensate until a better estimation function is determined.

Implement the changes in both NAP contracts.

Resolution: Acknowledged by the team.

3.5.7 Misleading UniswapPairCreated event

Severity: *Informational (Resolved)*

Context: [⊶4] : ***Factory.sol***:*227*

Description

In the `Factory` contract, the `UniswapPairCreated` event is fired on any call to the function, regardless if a pair was created or not.

Recommendation

Remove the `UniswapPairCreated` event as it does not seem to have any benefit.

Resolution: Resolved. The recommended fix was implemented in **3c032f7**.

3.5.8 generateRandomInRange returns pseudo-random values

Severity: *Informational (Acknowledged)*

Context: [⊶5] : ***Random.sol***:*5-24*

Description

The `generateRandomInRange` function from the `Random` library uses a pseudo random calculation which leads to users, that go to extremes lengths, to identify what token they might receive and not mint until they have a specific one.

Recommendation

Comment that this is known on the `generateRandomInRange` function. Since no advantage can be obtained by users (the reveal data is not known on mint), there is no advantage in using a Chainlink VRF (Verifiable Random Function).

Resolution: Acknowledged by the team.

team: this is not relevant for us as no damage or leak of value can be done