POLLENDAO REAUDIT

Distributed Lab

By Artem Chystiakov, JUNE 2022

Summary: production-ready.

The code is not production-ready due to issues found during this audit. Despite the decent code readability and transparency, there are 1 critical and 3 high severity bugs found that must be resolved. The overall code architecture is great and there are no problems discovered with the diamond pattern used. However, the code semantics suffer from the “childlike” errors that hint lack of proper code reviews during the development. Even though there is a great number of tests written, some of them just do dummy function calls and do not check any postconditions. The audit does not include significant recommendations for code optimization.

After conducting the reaudit of the contracts we can state that the project is still not mature enough to be deployed to the production. Even though many of the former issues have been fixed, the newly introduced logic brought many bugs and ways to break the intended logic of the system. Unfortunately, the overall code readability has decreased as well. We still highly recommend writing rigorous and extensive tests for the vital parts of the system.

During the second reaudit session, all of the aforementioned issues have been fixed and addressed. However, we were still able to spot several bugs that should be properly mitigated. Situation permitting, we recommend hosting a bugbounty program on one of the major platforms to ensure the contracts safety.

After the third reaudit completion we still think that the system is not ready to be deployed to production due to lack of proper testing (the coverage must be low because the changes in the core formulas do not proceed to the tests failing. We can’t check the coverage for ceratin due to the command simply failing) and found bugs during the reaudit proper. There is one low severity bug that is not fixed from the previous reaudit. Moreover, during this reaudit we have also found one high and one medium severity bugs that must be addressed.

The fourth reaudit included fixes to the bugs found during the preceding audit sessions. New logic was not added here. All of the issues found were fixed.

The audit is made regarding these commit hashes:

Vesting: c77ac77e937bf030b05c8f5edb754d11a94fd884
Pollen DAO: 574c5c7440cd2151cc4e1aa6e963d0b5dbe46adf

The reaudit is conducted considering this commit hash:

Pollen DAO: 2dd0e8782cfb483ff95f69eb069a75ae82042858

The second reaudit was made against this commit hash:

Pollen DAO: 9cacd6e857b832d93339ffa4acda85323a1fac63

The third reaudit references this commit hash:

Pollen DAO: 4ea40bb128360846869d1f0689eb5065ec13dc68

The fourth reaudit commit hash:

Pollen DAO: f866e790a610275a441ec9a7eccfdf4640f14192

STRUCTURE AND ORGANIZATION OF DOCUMENT

The information about the severity of the bugs is given below. The list starts with the description of the critical bugs in red and ends with the informational ones in blue.

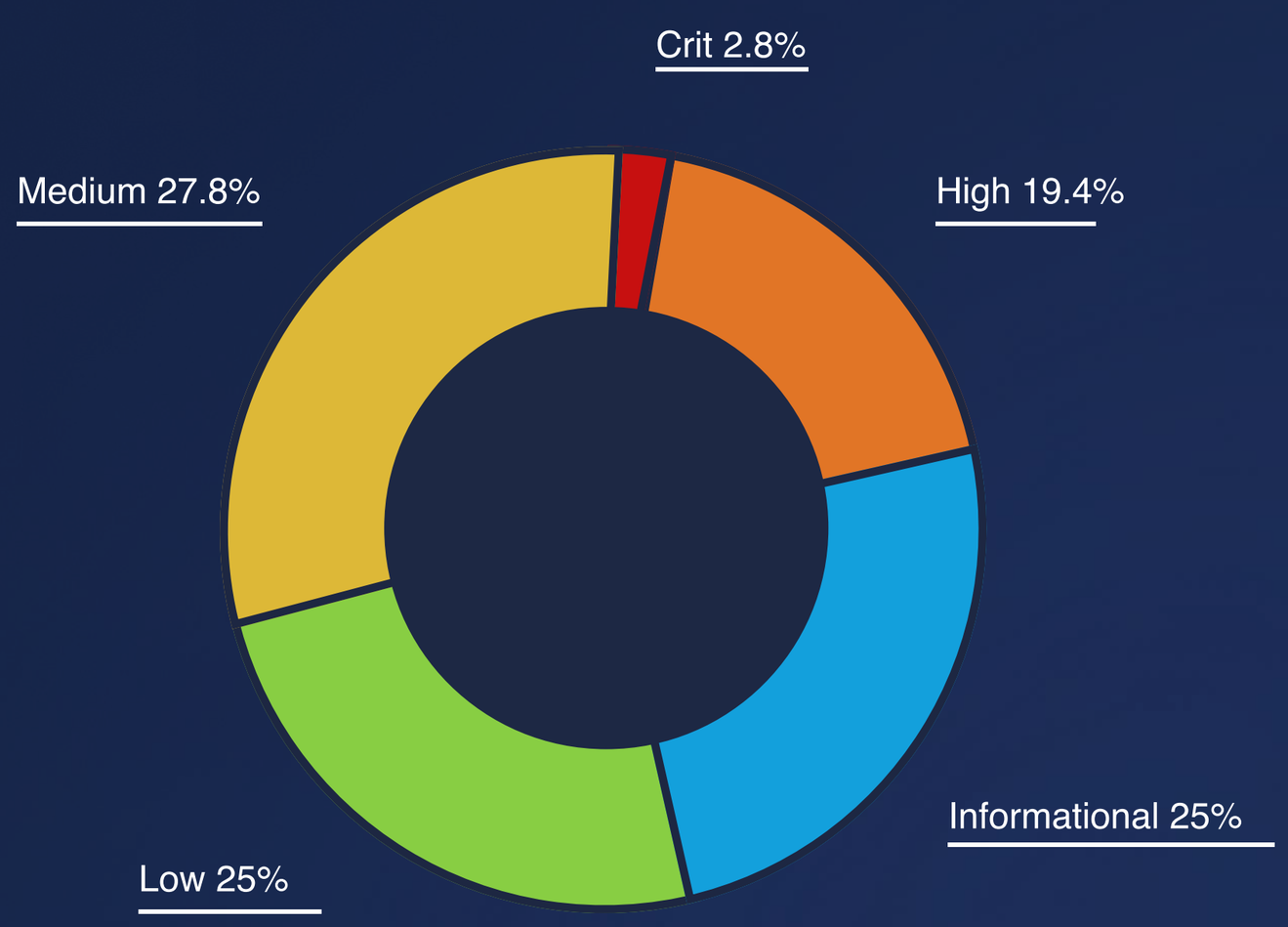
crit **Critical** - The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

high **High** - The issue affects the ability of the contract to compile or operate in a significant way.

med **Medium** - The issue affects the ability of the contract to operate in a way that doesn't significantly hinder its behavior.

low **Low** - The issue has minimal impact on the contract's ability to operate.

info **Informational** - The issue has no impact on the contract's ability to operate.



01 VESTINGVAULT.SOL

info

The `CLIFF_PERIOD` variable is unused

This variable should stand for the unlock period of the vesting term after which beneficiaries should be able to start withdrawing their funds. However, the variable is basically unused.

Recommendation:

Either remove the variable or integrate it into the vesting logic.

info

The contract does not work with `USDT` token

The vesting contract uses `token.transfer()` and `token.transferFrom()` functions in many of its methods and these functions are not compatible with `USDT` token.

Recommendation:

Substitute the `transfer` functions with the ones from `openzeppelin SafeERC20` library.

02 POLLENTOKEN.SOL

The reserve number may be written more explicitly

You can separate the reserve number in the constructor with underscores: `94_000_000_000_000_000_000_000_000`

Recommendation:

Add underscores to the reserve number.

03 LOCKEDPOLLEN.SOL

The inflation logic is not optimal and can be rewritten

Basically, in order to maintain the correct inflation rewards balance (the `reserve` variable), every function that manipulates the balances has to call the `processInflation()` in their first line. This creates unnecessary execution bulk because `processInflation()` function is quite expensive to call. There is another way of handling the inflation rewards logic without extra storage variables. The idea is to calculate the rewards on the fly by adjusting them to match the inflation rate.

Recommendation:

Remove all the logic accompanying `processInflation()` function including `inflationInfo` storage. Fix the `burn()` function bug. Add the following formula to the `unlock()` function after the for loop to adjust the `amount` value. The adjusted amount is the correct amount of tokens to be minted for a user including the inflation of tokens by paying out these rewards

```
uint256 futureBalance = amount + balance;
uint256 balancePercent = PRECISION * futureBalance / supply;

amount =
      amount *
      (PRECISION * supply / (PRECISION - balancePercent) - supply)
      / futureBalance;
```

What this formula does is that it calculates the percentage of user-owned tokens with their inflation rewards considering the current token supply, it adjusts the user-owned amount to maintain the same percentage after the rewards mintage and then proportionally increases the actual rewards to keep the same calculated percentage. This makes the user own the same percentage of tokens after the inflation and rewards mintage as they had before that.

It should be noted that despite this formula working correctly there are 2 caveats: it might output unexpectedly big numbers as `futureBalance` approaches `supply` and when all the supply of `PLN` tokens is locked by a single user, the formula divides by zero.

03 LOCKEDPOLLEN.SOL

Useless `Ownable` inheritance

The contract inherits from the `Ownable` contract and never uses the inherited functionality.

```
contract LockedPollen is ERC20, Ownable {
      . . .
}
```

Recommendation:

Remove `Ownable` from the inheritance list.

The inflation formula might revert

The inflation formula in the `processInflation()` function reverts if the total locked amount of `PLN` matches the `PLN` total supply.

Recommendation:

Do not allow users to lock the full supply of pollen into the contract.

Users of the contract can be DOSed

The `lock()` function is used to issue `vePLN` tokens in exchange for locking regular `PLN` tokens. This function is implemented in a way that allows anyone to issue tokens for someone else. However, this creates a problem. A malicious actor could lock a miserable amount of tokens (or even zero) for 4 years for someone he would like DDOS, disabling the possibility for the victim to withdraw the locked tokens for 4 years. It is worth mentioning that this attack only works on the accounts that do not have any tokens locked.

Recommendation:

Remove the feature of locking tokens on user’s behalf.

03 LOCKEDPOLLEN.SOL

User deposits are not decreased nor are the rewards

The `burn()` function burns the `vePLN` tokens of the user, but it does not update the `LockDetail` balances of one. This bug might influence the rewards of the user by minting them tokens more than anticipated.

```
function burn(address account, uint256 amount) external {
      _burn(account, amount);
      ERC20(plnAddress).transfer(dao, amount);
}
```

Recommendation:

Loop through all the `LockDetails` of the user and decrease the deposited amounts proportionally to the overall amount locked and the amount to be burned.

The functions miss the inflation update function call

Both `burn()` and `increaseLock()` functions manipulate with `vePLN` balance, which affects the inflation rewards directly. In order to function correctly, the inflation rewards have to be recalculated every time the balances change. These functions fail to do so.

Recommendation:

The `burn()` and `increaseLock()` function have to call `processInflation()` function on their first execution line.

The rewards supply can exceed maximum amount

The prorata formula in the updateCurve() function is wrong which might result in a excess of user rewards. The formula outputs wrong results when y + `c.offsetY` is more than `MAX_REWARDS_FUNDS`.

```
function updateCurve() private view returns (RewardCurve memory c) {
. . .

uint256 y = c.rate * (ts - c.offsetX);
      y = y + c.offsetY > MAX_REWARDS_FUNDS ? MAX_REWARDS_FUNDS : y;
      . . .
      c.offsetY += y;
}
```

Recommendation:

Correct the formula to be `y = min(y, MAX_REWARDS_FUNDS - c.offsetY)`.

04 POLLENDAO.SOL

Useless `require` statement

The `require` statement in the `removeModule()` function that checks a selector's belonging to the implementation contract can be removed. The removal is safe because the sanity hash already gets checked before that. Moreover, the addition of a new selector is validated in the `addModule()` function which ensures the safeness of the module removal.

```
function removeModule(address implementation, bytes4[] calldata selectors)
      private
      onlyAdmin
{
      . . .

      for (uint256 i = 0; i < selectors.length; i++) {
            require(
                  ds.implementations[selectors[i]] == implementation,
                  "PollenDAO: unregistered selector"
            );

            ds.implementations[selectors[i]] = address(0);
      }

      emit ModuleRemoved(implementation, selectors);
}
```

05 GOVERNANCE.SOL

The quorum is wrong

The quorum value in the `submitProposal()` function gets clipped and only 64 least significant bits are left. The quorum itself is a big number and this clip makes no sense because it is fairly unpredictable and shadows the quorum precision logic.

```
function submitProposal(address executer) external {
        . . .

        Proposal memory proposal = Proposal({
            submitter: msg.sender,
            executer: executer,
            yes: 0,
            no: 0,
            quorum: uint64(gs.quorum * supply),
            expires: uint64(block.timestamp + gs.votingPeriod),
            timeLock: uint64(block.timestamp + gs.votingPeriod +
            gs.timeLock)
        });

        . . .
}
```

Recommendation:

Store the quorum in a `uint256` variable.

05 GOVERNANCE.SOL

The voting comparison with a quorum is wrong due to precision

Due to this bug, it is not possible to execute proposals successfully. The `quorum` variable is set as a percentage of total pollen supply and has a precision of 3 decimal places. In order to compare variables against the quorum, one has to multiply them by 1000 to match the precision. The `executeProposal()` function fails to do so and compares the variables directly.

```
function executeProposal(uint256 id) external {
        . . .

        require(proposal.yes > proposal.no && proposal.yes >
proposal.quorum,"Not passed");
        . . .
}
```

Recommendation:

Either multiply the power of the proposal by 1000 in `voteProposal()` function after all the tokens are transferred or bring all the variables to the same precision in place.

The same proposal can be executed an arbitrary number of times

The `executeProposal()` function in the contract does not state the success of the proposal execution, thus exploiting the function to be run any number of times. The “double execution” logic should not be lain onto the executor’s shoulders as the governance contract is responsible for the validity of the proposals.

Recommendation:

Update the state of the governance contract in a way that would not make the proposal be executed twice.

06 QUOTER.SOL

Useless code in the `quotePrice()` function

The following lines do nothing in the `quotePrice()` function because `the _updatedAt` and `rate` values will be returned eventually.

```
function quotePrice(RateBase rateBase, address asset)
        public
        view
        returns (uint256 rate, uint256 updatedAt)
{
        . . .

        (, int256 answer, , uint256 _updatedAt, ) =
              priceFeed.latestRoundData();
        updatedAt = _updatedAt;
        rate = decimals == RATE_DECIMALS
              ? uint256(answer)
            : uint256(answer) * (10**uint256(RATE_DECIMALS -
                  decimals));

      return (rate, _updatedAt);
}
```

Recommendation:

Remove the `return` statement.

07 PORTFOLIO.SOL

Wrong comparison sign

The `amount` in `createPortfolio()` function is compared with “>” sign, which might cause problems with providing the correct `amount` from the front end.

```
function createPortfolio(
      uint256 amount,
      uint256[] memory weights,
      bool tokenType
) external {
      require(amount > ps.minBalancePollinator, "Insufficient amount");

      . . .
}
```

Recommendation:

Change the sign to “>=”.

The event emits wrong value

The `rebalancePortfolio()` function emits `PortfolioRebalances` event in the very end of its execution. As it stands, the `benchMarkRef` value is wrong to be broadcast because the currect value is calculated via weighted average and might be different.

```
function rebalancePortfolio(
      uint256[] calldata weights,
      uint256 amount,
      bool tokenType
) external {
      . . .

      emit PortfolioRebalanced(
            msg.sender,
            weights,
            value,
            benchMarkRef,
            amount,
            tokenType
      );
}
```

Recommendation:

Let the event emit correct benchmark value.

07 PORTFOLIO.SOL

Wrong benchmark value saved upon portfolio rebalance

The `rebalancePortfolio()` function completely overwrites the previously saved benchmark value with the new value on each portfolio rebalance. This is a broken concept because the rewards mechanism depends on the difference between the saved and current benchmark values. Basically, the pollinator can rebalance the portfolio just before claiming the rewards and escape the rewards punishment if the benchmark portfolio increased in price.

```
function rebalancePortfolio(
      uint256[] calldata weights,
      uint256 amount,
      bool tokenType
) external {
      . . .

      uint256 benchMarkRef = getBenchMarkValue();
      p.benchMarkRef[msg.sender] = benchMarkRef;

      . . .
}
```

Recommendation:

Instead of mindlessly saving a completely new value, the portfolio probably needs to calculate a weighted average of the saved benchmark value and the new one, with the weights being the amount of currently deposited tokens and the tokens amount passed as a new deposit.

07 PORTFOLIO.SOL

The function ignores delegatee 20% reward fee

The `getTotalReward()` function totally ignores the 20% delegatee reward fee when calculating delegator rewards.

Recommendation:

Add proper if clauses to account for that case.

Wrong calculation of the delegated amount

The `_delegatePollen()` function miscalculates the amount of delegated pollen by only taking into account the type of the token passed the function. This makes the delegated limit twice as much. Moreover, the formula used does not consider the passed `amount` of tokens to be delegated as delegated, thus enabling users to delegate any number of tokens once.

```
function _delegatePollen(
      address delegate,
      uint256 amount,
      bool tokenType
) private {
      . . .

      uint256 delegated = p.totalDeposited -
            p.deposits[tokenType][delegate];
      require(ps.maxDelegation >= delegated, "Delegation max exceeded");

      . . .
}
```

Recommendation:

First of all, subtract the deposits of the other token as well. Second of all, add the passed `amount` to the value if `msg.sender` is not an owner:

```
uint256 delegated = p.totalDeposited - p.deposits[true][delegate]
- p.deposits[false][delegate] + delegate == msg.sender ? 0 :
amount;
```

07 PORTFOLIO.SOL

Wrong benchmark value saved upon delegation

Just like in the bug one above, the `_delegatePollen()` function simply overwrites the new benchmark value over the saved one. This logic can be broken by a delegator who might delegate pollen just before claiming the rewards to get away with not reduced rewards.

```
function _delegatePollen(
      address delegate,
      uint256 amount,
      bool tokenType
) private {
      . . .

      p.benchMarkRef[msg.sender] = getBenchMarkValue();

      . . .
}
```

Recommendation:

Here the portfolio would need to calculate the weighted average of the values with the weights being the old delegation amount (the saved one) and the new one.

07 PORTFOLIO.SOL

Useless array memory allocations

There are 4 useless memory allocations for `assetPrices` array. The required memory already gets allocated in the `getPrices()` function making the highlighted allocation a waste of resources. The problem escalates to the functions: `createPortfolio()`, `rebalancePortfolio()`, `getTotalReward()` and `_delegatePollen()`. The snipped of only the first function is provided.

```
function createPortfolio(uint256 amount, uint256[] calldata weights)
      external
{
      . . .

      uint256[] memory assetPrices = new
            uint256[](weights.length);
      assetPrices = getPrices(weights, ps.assets.elements);

      . . .
}

function getPrices(uint256[] memory isValidAsset, address[] memory assets)
      public
      view
      returns (uint256[] memory prices)
{
      prices = new uint256[](isValidAsset.length);

      . . .
}
```

Recommendation:

Remove extra memory allocations.

07 PORTFOLIO.SOL

`totalAssets` is a useless variable

In the `isValidWeightsAndAssets()` function the `totalAssets` variable can be completely removed. Basically, the function would either revert and make this variable unused or the `totalAssets` variable would be equal to `weights.length`.

Recommendation:

Remove the `totalAssets` variable and return `weights.length` instead.

`maxDelegation` might never be reached

The `function _delegatePollen()` compares wrong values: `p.totalDeposited` and `p.balances[]`. On the one hand, the `totalDeposited` variable is non-normalized and represents the actual amount of pollen tokens deposited into the portfolio. On the other hand, the `balances[]` array stores normalized values of deposited pollen and knows nothing about the actual pollen balance of a portfolio creator. This fact increases the difference between these two variables and makes “total delegation” rise faster if the value of a portfolio is > `1.0`.

```
function _delegatePollen(address delegate, uint256 amount) private {
      . . .
      uint256 delegated = p.totalDeposited - p.balances[delegate];
      require(ps.maxDelegation >= delegated, "Delegation max
            exceeded");

      . . .
}
```

Recommendation:

Calculate `delegated` as `p.totalDeposited - p.deposits[delegate]`.

08 MINTER.SOL

The duplicate function

The function `getBenchMarkValue()` is a duplicate of the same function in the Portfolio contract.

```
function getBenchMarkValue() private view returns (int256) {
      . . .
}
```

Recommendation:

Remove the function from the `Minter` contract and make it `public` in the `Portfolio` contract.

The pollen supply can exceed the maximum allocation

The `mintInflationReimburse()` function that is called from the `vePLN` contract does not check that the minted amount of `PLN` does not exceed the maximum allocation of 200m tokens.

Recommendation:

Add the proper logic to the function and mint the minimum available amount of tokens.

08 MINTER.SOL

Storage variables are not in the storage contract

Minter facet storage should be entirely in the dedicated storage contract, however, there are two storage variables that are located in the minter contract directly. This bug does not affect the operability of the contracts yet due to the absence of other direct storage variables. However, it must be fixed to prevent unexpected behavior of the system.

```
contract Minter is
      PollenDAOStorage,
      PortfolioModuleStorage,
      MinterModuleStorage
{
      uint256 private constant RATE_DECIMALS = 18;
      uint256 private constant TIME_LIMIT_200M = 1673458433;

      uint256 private rewardMultiplier;
      uint256 private penaltyMultiplier;
            . . .
}
```

Recommendation:

Move the storage variables to the appropriate storage contract.

Useless array memory allocation

Similar to the portfolio's issue, there is a useless memory allocation before `getPrices()` function call.

```
function _withdraw(address owner) private {
      . . .

      uint256[] memory assetPrices = new uint256[]
            (p.assetAmounts.length);

      Portfolio portfolio = Portfolio(address(this));

      assetPrices = portfolio.getPrices(p.assetAmounts,
            ps.assets.elements);

      . . .
}
```

Recommendation:

Remove extra memory allocation.

08 MINTER.SOL

Unreachable `if` statement

There is an `if` statement in the `_withdraw()` function that is unreachable.

```
function _withdraw(address owner, uint256 amount) private {
      (      ,
            uint256 deposited,
            uint256 balance,
            uint256 r,
            bool isPositive
      ) = preprocessWithdrawal(owner);

      if (msg.sender == owner && p.isOpen) {
            require(
              ps.minBalancePollinator >= deposited - amount,
                  "Min balance reached"
            );
      }
}
```

There are 2 problems with the highlighted code. The first one is that the exact same if statement gets checked in the revert expression in the `preprocessWithdrawal()` function call. This means that the call either reverts or passes making the given if unreachable. The second problem is that in order to fulfill the require statement, the contract has to lock some user tokens forever and there will not be any conventional way to recover these tokens back.

Recommendation:

Delete the highlighted code and add appropriate pollinator logic to the functions that rebalance the portfolio. Make the portfolio “unrebalanceable” unless the user has enough tokens deposited.

08 MINTER.SOL

The user is unable to withdraw his tokens after the rate expiration

There are 2 `require` statements in the `processReward()` function that revert in case of the rate expiration and the supply excess. These `reverts` disable the option for the users to get their rewards and initially staked tokens after the `PLN` minting process finishes. The function should at least payout the staked `PLN` (the total supply does not change here) instead of locking the tokens inside the contract.

```
function processReward(
      uint256 reward,
      uint256 deposited,
      address portfolioOwner
) private {
      DAOStorage storage ds = getPollenDAOStorage();
      MinterStorage storage ms = getMinterStorage();

      IssuanceInfo memory sc = ms.schedule;
      require(sc.maxTime > block.timestamp, "Rate expired");

      uint256 maxSupply = sc.rate * (block.timestamp - sc.offsetX) +
            sc.offsetY;

      IPollen pln = IPollen(ds.pollenToken);
      uint256 totalSupply = pln.totalSupply();
      uint256 totalRewards = reward + (reward * rewardMultiplier) / 100;

      require(
            maxSupply >= totalSupply + totalRewards,
            "Insufficient balance for reward"
      );

      . . .

      if (amount > 0) {
            pln.transfer(msg.sender, amount);
      }
}
```

Recommendation:

Take the minimum of the `block.timestamp` and `sc.maxTime` to calculate the `maxSupply` value. In order to remove the second `require`, payout “free” rewards (that will not exceed the token’s total supply). This will ensure the ability of the users to withdraw tokens after the rate expiration.

08 MINTER.SOL

Users are underpaid and might lose their deposit due to a lack of precision

The `preprocessWithdrawal()` function calculates `r_` value with only 10**2 precision digits (which means a 1% error margin) and it is definitely not enough. If the deposited value is really big (let's say 100000 tokens with 18 decimals), with a margin of error of 1%, the user would lose 10000 pollen tokens.

Recommendation:

Calculate `r_` value and other related values with more precision digits. The conventional value would be 10**25.

08 MINTER.SOL

The rewards are miscalculated

The `_withdrawProfit()` function is responsible for updating `p.balances[]` variable to a value that would represent the current portfolio balance. However, the formula is wrong.

```
function _withdrawProfit(address owner) private {
      (
            uint256 currentValue,
            uint256 deposited,
            ,
            uint256 r,
            bool isPositive
      ) = preprocessWithdrawal(owner);

      require(isPositive, "Porfolio returns are negative");
      PortfolioStorage storage ps = getPortfolioStorage();
      PortfolioInfo storage p = ps.portfolios[owner];

      p.balances[msg.sender] -= (deposited * 1e18) / (currentValue);

      . . .
}
```

There Let’s prove that the formula is wrong with an example.

`Balances` = 900, `deposited` = 1000, `currentValue` = 20000$.
According to the formula, `balances` = 900 - 1000 * 1000 / 20000 = 850.
Let’s calculate the `purchasedValue` then. It would equal 1000 * 1000 / 850 = 1176$. This means that we have collected the reward with the current portfolio value being 20000$ and have set the purchased value of tokens to 1176$. As a result, we can now collect even more rewar

Recommendation:

To prevent this exploit, change the formula to the following one (remove the ‘-’ sign):

```
p.balances[msg.sender] = (deposited * 1e18) /
(currentValue);
```

08 MINTER.SOL

Users are underpaid and might lose their deposit due to a lack of precision

The `preprocessWithdrawal()` function calculates `r_` value with only 10**2 precision digits (which means a 1% error margin) and it is definitely not enough. If the deposited value is really big (let's say 100000 tokens with 18 decimals), with a margin of error of 1%, the user would lose 10000 pollen tokens.

Recommendation:

Calculate `r_` value and other related values with more precision digits. The conventional value would be 10**25.

The important function might revert, temporarily locking the contract

The `calculateMaxAllocation()` function might revert with an underflow exception, temporarily locking the `Minter` contract for the funds withdrawal. The case might happen when the pollen supply reaches the maximum allowed supply by the price curve and the inflation reserves are high.

```
function calculateMaxAllocation() private view returns (uint256) {
      DAOStorage storage ds = getPollenDAOStorage();

      uint256 maxSupply = getIssuanceCurve();
      uint256 plnTotalSupply = IPollen(ds.pollenToken).totalSupply();
      uint256 reservedPln = ILockedPollen(ds.vePollenToken)
            .inflationInfo()
            .reserved;
      return maxSupply - plnTotalSupply - reservedPln;
}
```

Recommendation:

Check that `maxSupply` is greater than `plnTotalSupply + reservedPln` before subtraction or return `0`.

09 LAUNCHER.SOL

The DAO gets deployed without a treasury module

There are no bugs involved into deploying the pool without the treasury module, however it is strange to see unused module in the code.

Recommendation:

Either deploy the pool with the treasury module, or remove it from the project. You can also move it to the “mock” directory.

The contract can be dosed and DAO pool might never be deployed

Basically if the first voting campaign fails, the DAO pool might never be deployed. This exploit is possible because one can call `validateCampaign()` function on future campaigns that do not yet exist. The `validateCampaign()` will set `executed` flag to `true` and `passed` flag to `false` of these future campaigns. Then the exploiter can call `startCampaign()` function on such campaigns to instantly start a new one (surpassing the expiration period). So the DOS is the following: exploiter calls `validateCampaign()` function on many future campaigns (ids from 1 to infinity). This costs him pretty much nothing because the Avalanche is cheap. Then if the very first voting campaign fails, the second campaing starts and as soon as the voting begins, the exploiter calls `startCampaign()` function to start a new campaign and enialate the voting results, blocking the deployment of the DAO.

Recommendation:

Allow validation of existing campaigns only: add

```
require(campaignId <= currentCampaign);
```

to the `validateCampaign()` function.

AFTERTHOUGHTS

- ~~Although the architecture of the project is really flexible and modular, there are a lot of places where the code is duplicated. For example, the functions `setIssuanceRate()` and `initializeIssuanceInfo()` in the `Minter` contract share 90% of their code. The functions `createPortfolio()` and `rebalancePortfolio()` in the `Portfolio` contract suffer from the same code bulk. The refactoring has to be made and the code that does the same logic has to be moved into the separate functions to be reused.~~

- ~~There are numerous “magic numbers” in the contracts that have to be declared as `constants`.~~

- ~~There are several events that do not follow the naming conventions.~~

- ~~There are several `external` functions that accept `memory` variables. These variables can be changed to the `calldata` ones.~~

- It is impossible to run the test coverage. The `npm run coverage` command fails with the execution error.

With great appreciation and respect.

Distributed Lab

JUNE 2022