Report
ERC20 Solidity Review for Grizzly

Who	Dr. Thomas Bocek Axelra AG Hagenholzstrasse 83 CH-8050 Zürich thomas.bocek@axelra.com
Date	17.01.2022 – 01.02.2022 (invested time for the review only ~27h)
Task	Review of the Grizzly ERC20 smart contract
Input	Files sent by Christian Killer <killer@grizzly.fi>, 16.01.2022, 21:51 Version: 745fb169a3cf991aa98b287d240795a0d7c67e64b6c5a07131796012b43c3e30 Grizzly.sol 588789af13c8165f584982641331a0fda15a25119adfeb785d3cf89664d44319 HoneyBNBFarm.sol 7b30c0e4ae303bdc4805aaeb97c74ed33180bd4f161c3117a9f14883bdad3471 HoneyToken.sol 0683102d8257b67bc95d470785b5f67c069de75523007a402502e38e7c11b665 LaunchSale.sol a4ad04da24e52ed3eddd474a96ee1d05a55e698d1501827649b550cede057147 readme.md 2c17fd63ffe46d7443e0a53cc957ee4c1a987f139b4e2b28f4d282b7bc989ad8 Referral.sol f39b6ea86c6ad3da231692388dfe9c7b0b731acef369439156babee429dbc144 StakingPool.sol 5578dcc9f57afe843fb60890d7cb9cee4c0fcbda9464e0f4f346b3ce3b1c07cf Config/BaseConfig.sol 840b522c1365e0f6ac96ced336c2eac8ac9d87fe812f60502dc1d1ccfcff429a DEX/DEX.sol 27a944f4d4d021e1adf9992b1b7acb9679e82ab00349366ec42685f08cb8f108 Strategy/GrizzlyStrategy.sol 83c25d2362b557fced2359405bde476c60ce56e029c559de3f92015abe4909cd Strategy/StableCoinStrategy.sol a18683589d13228bb30c188ca5da50238872b1fe094888ffd4d134acd1ba206a Strategy/StandardStrategy.sol

Analysis

The contract was reviewed with the remix IDE – https://remix.ethereum.org, offering static code analysis. Most findings were found with a manual review.

Contract Descriptions

Grizzly.sol is the main contract. On use-case is that a users can call deposit() and include BNBs. The contract swaps those BNBs to TokenA and TokenB, and adds liquidity via a DEX. TokenA and TokenB, are constant an set in BaseConfig. After adding liquidity, the resulting LP tokens are staked (MasterChef). In the traditional DEX, a user can then get the earned rewards from staking (MasterChef) and restake (compound) or withdraw it.

Grizzy restakes for you, and converts these rewards from BNBs into 70% LP tokens (MasterChef), 24% buy back Honey Tokens, 6% is newly minted, and those are then user for a strategy reward. 6% goes to the dev team. In case of a high honey price, 24% is added as liquidity to the Honey/BNB pool, and the resulting LP is added to the staking pool (HoneyPool) as reward. 30% is newly minted and this is added as reward, 6% goes to the dev team. The benefit is that Grizzly adds liquidity, stakes its LP token automatically. However, restaking the rewards is done manually by calling stakeRewards()

Due to time restrictions, the rest of this part is skipped.

Findings

The findings are categorized into 4 different levels: Critical, Major, Minor, and Comment. Critical issues are the issues that needs to be fixed immediately and poses a potential loss of funds. These are the issues that will prevent the product from working if it were released in that state. Major issues needs attention but do not necessarily pose a risk of loosing funds. Minor and Comment issues are usually reserved for "nice to haves" or specific comments.

Please note, this is a “first-pass” categorization. With more time, these points need to be rechecked and reevaluated. Sometimes, minor issues will go to major, sometimes major issues disappear, as further investigation shows that they do not apply.

Critical

- None

Major

- A withdrawEmergency() may be necessary anyway as you could update the role REWARDER_ROLE, than any withdrawal are locked.
- Found by 21Analytics → (block.timestamp + 300) is pointless. The value must come

from the user, thus exposing it.

Minor

- deposit() in Grizzly checks the slippage in checkSlippage() However, the addresses are not used further in the _deposit() function. You could set a wrong token, get a good slippage, do _deposit() with the right token and have a terrible slippage. So the front end must supply the correct tokens.
    - SwapRouter functions in DEX.sol (and removeLiquidity* in StakingPool.sol) sets the minimum amount to 1. (since you check the slippage before). You could pass the limits to those functions. This could make the code more readable, however, if you have checked the code that its working, its fine.
- IStakingPool.sol – stakerAmounts do not match the params
- StakingPool.sol, rewardLP has a TODO. Please resolve
- If the price is below efficiency, the 24% Honey Token will be kept in circulation (as strategy reward), and 6% are created/minted, so with constant demand for the Honey Token, the price will go down, right?
    - From your answer, I deduct this is a yes, as there will be never a reduction of the supply. It will be less minted, 6% instead of 30%. So the token is designed to be inflationary. In order to keep the price at a certain level, should the contract also burn the tokens and make it deflationary?

Comments

- Checked: DEX.sol splits the amount in half in several places. So here we may have problems due to rounding issues. However, all occurrences are fine, and no rounding issues occur.
- Deadline for swaps is hardcoded for 300s (5min). This could be parameterized
    - This would become obsolete if time parameter is provided by the user.
- Interfaces not implement ed, e.g., with HoneyToken.sol does not implement IHoney.sol. Same with Grizzly, StakingPool, Referral.
- LaunchSale. The state from PENDING to OPEN seems unnecessary, from PENDING, you can only do OPEN, not even cancel.
- The Honey Tokens and BNBs for liquidity is provided by you. If sale is finished, the BNB from the investors gets all swapped to Honey Tokens, which can be claimed in claimTokens() proportionally to the locktime. If much more BNBs were provided by investors than by you, then this would skyrocket the Honey Token price. Please make

sure that the saleCapValue is set correctly.

- Nice, everyone can call statkeRewardForBounty(), and claim unused tokens. But why would anyone call stakeRewards(), just call stakeRewardsForBounty()? Put some ifs before the transfer, if the amount is 0, then you could remove stakeRewards().
    - Discussion with Roger: Grizzly is calling stakeRewards(), but this could be incentivized to make other users this call this as he gets a bounty.
- Rentrancy Guards
    - Function withdrawToToken() in Grizzly.sol needs a reentrancy guard, as it calls _stakeRewards(), that does a call (transfers ETH/BNBs)
    - Function deposit() in Grizzly.sol needs a reentracncy guard, as it calls _deposit(), which calls _stakeRewards(), that does a call (transfers ETH/BNBs)
    - Function depositFromToken() in Grizzly.sol needs a reentracncy guard, as it calls _deposit(), which calls _stakeRewards(), that does a call (transfers ETH/BNBs)
    - Please note, I did not put these issues as major, as the receiver of those calls is the DevTeam address, which I believe is under the control of the founders. Thus, an attack would need to be launched from the DevTeam address. Also, you seem to be aware of that as you mention in the comment that you are only concerned with msg.sender: “The Contract uses ReentrancyGuard from openzeppelin for all transactions that transfer bnbs to the msg.sender”
    - Confirmed by Roger, the DevTeam address is under control.
- finishSale() first transfers the honey tokens to the launch contract, then to the DEX. This could be made directly. Also you would not need to send tokens a back, as you never sent it in the first place (on line 169 – 172). (See minor issue 1: also here the ReentrancyGuard is not required, as its a non public function call back to your address)
    - 21analytics also checked the contract and had similar inputs. Since the contract is checked, I would leave the functionality in place.
- Use safeTransfer from OpenZeppelin, which also handles tokens that does not return a boolean.

Automated Checks reported by Remix IDE

- Block timestamp: 18 times warning for use of block.timestamp.
    - Found by 21Analytics → (block.timestamp + 300) is pointless (see major issue)
- Low level calls: Remix reports that “call” should be avoided whenever possible. However, Consensys best practices states that sending Ethers should be done with a “call” plus a reentrancy check, plus check if returned value was true. All occurrences of

this have been checked and are according to Consensys best practices.

- Gas costs: Gas cost issues have been checked and the warnings by Remix can be ignored
- For loop over dynamic array: For loop in DEX.sol, line 281, is fine as the length of the fromToken array depends on the user input
- ERC20 contract's "decimals" function should have "uint8" as return type : can be ignored, as it returns uint8
- Similar variable names: can be ignored
- No return: can be ignored, those are all interfaces
- Guard conditions: require is fine in those cases
- Data truncated: no issues found, I checked those cases

Checking Interfaces

- IMasterChef.sol: please add a link to the contract: https://bscscan.com/address/0x73feaa1ee314f8c655e354234017be2193c9e24e#code
- IERC20.sol: correct: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol
- IGrizzly.sol matches the implementation
- IHoney.sol does not match the implementation, overlaps with IERC20.sol, and mint() is missing
- HoneyToken.sol – claimAdditionalToken() → works as described, does not need onlyRole, as its internal
- IReferral.sol - matches the implementation
- IStakingPool.sol – stakerAmounts do not match the params
- IUniswapV2Pair.sol – matches https://github.com/Uniswap/v2-core/blob/master/contracts/interfaces/IUniswapV2Pair.sol
- IUniswapV2Router01.sol - matches https://github.com/Uniswap/v2-periphery/blob/master/contracts/interfaces/IUniswapV2Router01.sol

Advice: remove redundancy (ER20 overlap), and remove unneeded interfaces, and if needed make sure the params are matching.

Remix Warnings

No Remix warnings were seen, kudos!

Testcases

There are testcases, but this was not part of this review.

Answered Questions (by Roger Staubli via Slack)

- Is totalPendingReward() correct in DEX.sol? For me it seems like it calculates the liquidity of either TokenA or TokenB for the given pending rewards.
    - From you answer, this should be correct, as it should return the reward in terms of LP tokens.
- You have three staking pools for the honey token: HoneyBNBFarm.sol, StakingPool.sol, and the staking contract in MasterChef. MasterChef staking is when depositing and receiving LP Tokens from MasterChef. When staking the rewards, then StakingPool is used for the reward tokens. When is HoneyBNBFarm used? (When reviewing the contracts, this HoneyBNBFarm was not well integrated. What LPToken is supported here?)
    - Your answer: This is for Honey-BNB-LP tokens can be staked and rewards are being paid out. HoneyBNBFarm.sol will only support Honey-BNB-LP tokens.
    - Suggestion: you could add this into a comment. That should make it clear.
- HoneyBNBFarm.sol → You inflate your supply even more with the creation of HoneyTokens in claimRewards(). Should this not be transfer rather than a mint? (Please note that due to minting, you don't need a function unstakeEmergency, in case of a transfer, you would need one.)
    - Your answer: Because we were worried that only a few people will use the plattform and many many will buy and stake the token, we decided to add an additional mint in the Honeypot.

Questions from Grizzy.fi

- Are there any potential issues or more advanced threats (such as backrunning / sandwich attacks) possible in the referral logic / Referral.sol?
    - Something that may need investigation is that the contract needs funding from outside. There is a approve in place, but for the withdraw, HoneyToken.transfer(msg.sender, _amount) is used. So the contract needs Honey Tokens. If there are no honey tokens on the contract, withdraw fails. Since no SwapRouter is involved, I don't see issues with sandwich/backrunning attacks.
- Are there any potential attacks possible that exploit the new minting of tokens? Is the minting of tokens uniformly correctly implemented?
    - The problem with minting is that you have unlimited tokens. One solution could be don't mint, but transfer the tokens from a pre-funded reserve address. Only have a portion of tokens in that reserve address. In that case, if something happens, only the reserve gets depleted. This may have consequences for the contract, and functions may start to fail, due to unavailable funds. But maybe in such a case, this is desired.

    - Also the minting depends on beeEfficiencyLevel, which is the Honey Price. With a flash loans, this could be changed, leading to a high or low amount of minted tokens.

- Are the token rewards correctly calculated at all moments in time and can the calculation function be exploited? How could we eliminate the risk of this happening?
    - The calculation looks fine, however, I could not check all corner cases.

- How can we assure upgradeability to a new version, without introducing major potential bug sources? Or even worse, opening up to flashloan attacks (e.g., as has happened in the past with Pancakeswap) This will most likely be with the Hives.
    - Try to avoid bugs in the first place :) Since, this is difficult, you could either use proxies (who has control?) or deploy new contracts (V1, V2, V3). Also a possibility could be to “import” by reading the state from a V1 contract and importing it into a V2, marking this V1 as imported. It could help if important states are accessible.

- Do we need some type of „emergency“ function that would pay back immediately all investors fairly, in case of some event / condition being met?
    - Yes, see major issue.

Disclaimer

The audit makes no statements or warrantee about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

Conclusions

The smart contract is in a good shape with a medium code complexity. I could not spot critical bugs, but those major issues need to be addressed and discussed. The comments should be looked at and/or discussed.