Audit Boutique

Grizzly.fi Smart Contract Audit Report

Produced for:
Grizzly.fi

Produced by:
Audit Boutique

January 27th 2022

Contents

1. Executive Summary

The audit has uncovered no high severity issues, five medium severity issues, and nine low severity issues. We also list six minor suggestions of how the contracts could be improved.

The Grizzly.fi Smart Contract suite implements complex liquidity mining and staking functionality behind a simplified external interface. Whereas this might increase the convenience for the user accessing the contracts, it also implies involved methods that orchestrate diverse functionality within monolithic invocations. Most of the medium severity issues that this audit report lists are connected to this logical and architectural complexity.[1]

As a consequence of this complexity, considerable uncertainty remains after the limited temporal scope of this audit regarding the correctness and security of the smart contract suite.

2. Scope

The review was performed on Git revision `62707aac48cfc92caf130cb866c56ac404d399d7` of the Grizzly.fi Smart Contracts repository.

In particular, the review pertained to the following smart contract source files, specified here with their `sha256` prefixes:

`contracts/Config/BaseConfig.sol`	`5578dcc9f57afe843fb6`
`contracts/DEX/DEX.sol`	`840b522c1365e0f6ac96`
`contracts/Grizzly.sol`	`745fb169a3cf991aa98b`
`contracts/HoneyBNBFarm.sol`	`588789af13c8165f5849`
`contracts/HoneyToken.sol`	`7b30c0e4ae303bdc4805`
`contracts/LaunchSale.sol`	`0683102d8257b67bc95d`
`contracts/Referral.sol`	`2c17fd63ffe46d7443e0`
`contracts/StakingPool.sol`	`f39b6ea86c6ad3da2316`
`contracts/Strategy/GrizzlyStrategy.sol`	`27a944f4d4d021e1adf9`
`contracts/Strategy/StableCoinStrategy.sol`	`83c25d2362b557fced23`
`contracts/Strategy/StandardStrategy.sol`	`a18683589d13228bb30c`

as well as the following auxiliary test and specification files:

`README.md`	`45cf43d3694454393a69`
`contracts/readme.md`	`a4ad04da24e52ed3eddd`
`test/integration/DepositWithdraw.ts`	`509cda71438816538eea`
`test/integration/Test1.ts`	`25ffddc8bd7e6ddc6a0c`
`test/integration/Test2.ts`	`1feebbb98ac00ac902ec`
`test/integration/Test4.ts`	`6e7aa0e321c1aecc78ad`
`test/integration/Test5.ts`	`c207cdbac07e22f1ab72`

[1] See in particular issues 6.2.2, 6.2.4 and 6.2.5.

`test/integration/Test6.ts`	`e6647c4b41158f46033c`
`test/integration/test3.ts`	`36c62257f866ca230395`
`test/unit/BaseConfig.ts`	`82a46ef56023a741a398`
`test/unit/DEX.ts`	`7ada48f5d347dd3477a9`
`test/unit/Grizzly.ts`	`2c7293fa79d948cc07d4`
`test/unit/GrizzlyStrategy.ts`	`b99159b2cd94e9bfea1c`
`test/unit/HoneyBNBFarm.ts`	`3a6532255ebf55012b30`
`test/unit/HoneyToken.ts`	`37ee1366b87162382dc1`
`test/unit/LaunchSale.ts`	`c12d9f9571e70b67b8b8`
`test/unit/Referral.ts`	`e82d7a8b67a4c2c2b63a`
`test/unit/StableCoinStrategy.ts`	`10e50bb8dcbbe3e6bf65`
`test/unit/StakingPool.ts`	`883c6faf333351528582`
`test/unit/StandardStrategy.ts`	`25623952768d1e3cbf13`

Imported contracts from well-known libraries such as Open-Zeppelin were not part of the review.

The review was constrained to the Solidity source files. Low-level assembly code generated thereof was not inspected.

3. Methodology

The review consisted of the following steps:

- Check for compliance with smart contract development best practices
- Check for compliance with specification, where available and applicable
- Manual inspection and analysis of the smart contract and test code
- Usage of static analysis tools

4. System description

4.1 Contracts

The Grizzly.fi smart contract suite is a set of contracts that provide a multitude of DeFi functionality. Its central purpose is to provide a user-friendly way of doing liquidity mining on a variety of decentralized exchange (DEX) pools. A large part of the codebase manages the reinvestment of the proceeds from this activity. To this end, the smart contract suite makes extensive use of DEX functions to convert between tokens and BNB, staking constructs to incentivize locking up of tokens, as well as a multitude of reward mechanisms centered around the Honey token.

4.1.1 `HoneyToken`

The Honey token (subsequently called "GHNY", its putative symbol) is the token around which the smart contract suite revolves. In multiple places, honey tokens are used as a reward mechanism. The Honey Token is a fairly standard ERC20 token. Tokens can be minted by any account or contract holding the corresponding role. As a distinguishing feature, the contract automatically mints additional tokens to a configured set of addresses. For every 100 tokens that are claimed by a minter account, four different accounts receive a total of 22 tokens. The only exception to this is the initially minted amount during contract creation.

4.1.2 `LaunchSale`

The `LaunchSale` contract facilitates the initial sale of GHNY to the end-user. A design goal is to achieve a notion of fairness where the sale price is uniform among users during a certain time window. To this end, the contract allows users to register their buying intent during an "open" phase with the contract by sending BNB to it. The amount of tokens that this will entitle the user to is not known beforehand. Before the administrator finishes the open phase, however, pledges may be withdrawn.

Two buy-phase features are optional and deactivated per default: A sale cap value can be instituted if desired, as well as a whitelisting state where any address that wishes to buy must first be whitelisted by an administrator.

The crucial step occurs when the administrator advances the state from "open" to "finished". To do this, he must provide liquidity in both BNB and GHNY. This liquidity is used to seed a Pancakeswap Pool for the WBNB-GHNY pair. Subsequently, and within the same transaction, the pooled user funds collected during the open phase are used to buy the GHNY token on behalf of the users from the liquidity pool.

Finally, users may withdraw tokens that were bought on their behalf. The full amount is not available immediately. Instead, the amounts vest linearly over a configured time period.

4.1.3 `Grizzly`

The Grizzly contract is the central contract of the Grizzly DeFi system. It is deployed once per desired token pair. At its core, the contract converts funds deposited by a user (either BNB or a third token that is first exchanged for BNB) into the token pair in question and provides them as liquidity to an external DEX pool. The liquidity pool token received in exchange for this is then deposited into a third-party staking pool.

The staking of the liquidity pool tokens accrues a reward denoted in a dedicated reward token. Those proceeds are automatically converted to BNB on a DEX and staked on the Grizzly contract. There are three strategies for this staking process, chosen by each user individually before he makes deposits (with a default "standard" strategy). The contract then automatically allocates the sale proceeds to the three strategies based on the shares of deposits belonging to each strategy. The three strategies are as follows.

4.1.3.1 Stablecoin strategy

This strategy converts 97% of the funds into the two tokens and provides them as liquidity to the DEX and stakes the liquidity pool tokens, representing a proper "reinvestment". Additionally, it registers rewards accruable to stablecoin strategy depositors in the amount of liquidity pool tokens bought. Finally, 3% of the funds are transferred to a configured "Dev Team" address.

4.1.3.2 Standard strategy

The standard strategy uses 70% of the proceeds for the aforementioned "Stablecoin Strategy" reinvestment approach. 6% of funds are transferred to a configured "Dev Team" address. The remaining 24% are allocated depending on the Honey token price as queried from a DEX.

If the price is lower than a configured threshold, 24% of the proceeds are used to buy the Honey token on the configured DEX. In addition, an amount of Honey tokens corresponding to 6% of the funds (based on the just queried price) are newly minted to the Grizzly contract. Finally, standard strategy investors accrue a reward denominated in Honey token amounts and equaling to the tokens bought from the DEX plus the tokens minted (this is just an internal bookkeeping function, no tokens are transferred).

If the price is higher than the threshold, 24% of the proceeds are used to provide liquidity to the BNB-Honey liquidity pool. The accrued liquidity pool tokens are staked in the StakingPool contract part of this contract suite (as opposed to token pair liquidity pool tokens which are staked on external staking pools). Here, Honey tokens corresponding to 30% of the funds are newly minted. The standard strategy investors reward equals the number of minted Honey tokens in this case.

4.1.3.3 Grizzly strategy

The Grizzly strategy also transfers 6% of the proceeds to a configured "Dev team". The rest is allocated based on the current price compared to a threshold, analogous to the Standard strategy.

If the price is lower than the threshold, 94% of the proceeds are used to buy the Honey token on the DEX. Additionally, Honey tokens corresponding to 6% of the proceeds are newly minted. The sum of those Honey tokens is staked on the internal staking contract.

If the price is higher than the threshold, 70% of the proceeds are used to buy the Honey token on the DEX. 24% of the proceeds are used identically to the over-threshold condition in the Standard strategy, i.e. they are used to provide liquidity to the BNB-Honey liquidity pool and the liquidity pool tokens are staked internally. Additionally, Honey tokens corresponding to 30% of the value of the proceeds are used to mint Honey tokens. Again, those together with the bought Honey tokens are staked internally.

Finally, and common to all three strategies, Honey tokens equaling 1% of the reward amount share corresponding to deposits that came with a referral are minted and transferred to the Referral contract.

If a user wishes to withdraw his funds, the reverse process from the deposit will take place: The pair liquidity pool tokens are withdrawn from the staking contract and converted to tokens of the pair by withdrawing the corresponding liquidity from the pool. Finally, the two sets of tokens are sold for BNB on the DEX and the proceeds from this sale are transferred to the user.

4.1.4 `HoneyBNBFarm`

The HoneyBNBFarm contract is a staking contract allowing to stake liquidity pool tokens of the BNB-Honey liquidity pool. It pays out rewards in (newly minted) Honey tokens proportional to the amount of tokens staked and staking duration.

4.1.5 `DEX`

The DEX contract provides some utility functions to interact with a Pancakeswap instance, for example for exchanging BNB with a token, or for exchanging liquidity pool tokens. To provide a simplified interface, it often executes multiple DEX transactions within one function call, and hard-codes some of the DEX method arguments.

4.1.6 `Referral`

The referral contract manages referral information for all Grizzly contracts. When a user makes a Grizzly deposit, he can specify a referral address which will be able to claim a reward in Honey tokens proportional to the number of liquidity pool tokens the referred deposit resulted in. The rewards are inspired by EIP-1973.

4.1.7 `StakingPool`

The `StakingPool` contract is used to stake Honey tokens in the Grizzly strategy. It pays out rewards in both Honey tokens, as well as BNB-Honey-LP tokens. Periodically the contract is awarded GHNY from externally and LP tokens from the Standard and Grizzly strategies in a Grizzly contract. The contract keeps an internal account who staked how many coins. Only once the GHNY tokens are unstaked is a reward calculated and deducted from the contract's share of GHNY. LP tokens can be claimed separately and are converted to GHNY and BNB prior to payout.

4.2. Roles

Various contracts contain functions that are protected through privileged access. Throughout, OpenZeppelin's AccessControl contract is used to grant the administrator role to an address provided as a constructor argument.[2] Afterwards, the administrator may grant the relevant roles to other addresses or itself. The roles are:

- `Grizzly.sol`: An updater that may set a new bee efficiency level
- `HoneyBNBFarm.sol`: An updater that may set a new block reward
- `HoneyToken.sol`
    - A minter which may claim new tokens
    - An updater which may set new reward recipient addresses
- `LaunchSale.sol`: An updater that may advance the sale state, set some sale parameters and change the whitelisting state
- `Referral.sol`: A rewarder role which may deposit and withdraw LP reward tokens
- `StakingPool.sol`: An updater role which may set the parameters for the different reward phases

[2] The exception to this is `Grizzly.sol`, which grants the administrator role to the contract deployer, but see issue 6.3.2.

5. Best practices checklist

Non-adherence to the characteristics listed in the below list does not represent a security issue itself. However, following best practices is a good indicator of care and attention to detail, it allows the review process to be more focused and efficient, thus making it more likely for issues to be uncovered.

- ✓ The code was provided as a Git repository
- ○ There exists specification, covering the most important aspects of smart contract functionality (only the issuance schedule is specified)
- ✓ The development process is understandable through well-delineated, atomic commits
- ✓ Code duplication is minimal
- ✓ The smart contract source files are provided in an unflattened manner
- ✓ The code compiles with a recent Solidity version
- ✓ The code is consistently formatted
- ✓ Code comments are in line with the associated code (mostly)
- ✓ There are tests
- ✓ Tests are easy to run
- ✓ Tests provide a reasonably high coverage (see Appendix A)
- ✓ The code is well documented
- ✓ There is no commented code
- ✓ There is no unused code
- ○ The code follows standard Solidity naming conventions

6. Findings

We rank our findings according to their perceived severity (high, medium, low), where an issue's severity is understood to be the product of its likelihood of being triggered and the impact of its consequences. Where deemed appropriate, we also report issues that are not security-relevant per-se but impact things like transaction-cost effectiveness or user experience.

See section 6.4 for suggestions for improvements which have no discernible impact on any of the above.

6.1 High severity

No issues found.

6.2 Medium severity

6.2.1 Hardcoded minimalistic DEX safety arguments induce slippage and front-running risks

`DEX.sol` provides a simplified interface onto some of the exchange functions of the `UniswapV2Router` interface. One of the ways it achieves this is by hard-coding the safety parameters of exchange functions which control the maximal slippage the user is willing to accept. They are all set to a constant `1`, thus in principle accepting any non-zero exchange outcome (note that this amount usually corresponds to 10^{-18} tokens or 1 wei in BNB respectively).

Whereas this increases the convenience of accessing the API, it induces non-overridable slippage risks on the users of those functions.[3] They are thus in extensive danger of being front-run on the DEX. This is exacerbated by the complexity described in issue 6.2.2, as well as the ineffectual deadlines described in issue 6.3.5 (although most, but not all, of the users of those functions have their own deadline functionality).

The `checkSlippage` function in principle allows the user to craft arguments such that some guarantees about slippage could be made. However, this is highly susceptible to accidental misuse, for example by forgetting one of the token pairs involved. It also forces the user to provide a universal slippage parameter for all trades involved. Furthermore, it puts a very large burden on the user for situations where the same liquidity pools are accessed in the same direction within one transaction, as described in issue 6.2.2.

6.2.2 Exceedingly complex intra-transaction DEX interactions

The Grizzly contract orchestrates complex token interactions behind a relatively simple interface for the user. However, this convenience comes at the cost of high complexity of single method calls, most easily demonstrated by the calls into the `UniswapV2Router`-like DEX.

As an extreme but not atypical example, the `Grizzly.depositFromToken` function exhibits a code path with the following DEX calls (for convenience we list the `DEX.sol` functions, as well as the actual DEX calls one level indented):

- `convertTokenToEth`
    - `swapExactTokensForETH` (user-specified token)
- `convertTokenToEth`
    - `swapExactTokensForETH` (staking reward token)
- `convertEthToPairLP`
    - `swapExactETHForTokens` (token A)
    - `swapExactETHForTokens` (token B)
    - `addLiquidity`
- `convertEthToTokenLP`
    - `swapExactETHForTokens` (Honey token)
    - `addLiquidityETH`
- `convertEthToToken`
    - `swapExactETHForTokens` (Honey token)
- `convertEthToTokenLP`
    - `swapExactETHForTokens` (Honey token)
    - `addLiquidityETH`
- `convertEthToPairLP`
    - `swapExactETHForTokens` (token A)
    - `swapExactETHForTokens` (token B)

[3] Compare https://docs.uniswap.org/protocol/V2/guides/smart-contract-integration/providing-liquidity

  - `addLiquidity`
- `convertEthToPairLP`
  - `swapExactETHForTokens` (token A)
  - `swapExactETHForTokens` (token B)
  - `addLiquidity`

There are multiple problems stemming from this complexity. Firstly, a back-of-the-envelope calculation suggests that this results in non-negligible fees even for the relatively low BSC gas prices currently in place. [4]

More severely, however, it makes it downright impossible for the transaction initiator to predict the outcome of a transaction. It is well-known that Uniswap-like pools with their inherent slippage induce a large amount of uncertainty because the state of the liquidity pool when the transaction will execute cannot be predicted. This is exacerbated here because assets are exchanged multiple times in the same direction within the same transaction. In the above execution trace, all of the Honey token, token A and token B are exchanged for BNB three times.

As mentioned in issue 6.2.1, fine-grained slippage control is deactivated in a non-overrdiable manner in the DEX contract. Instead, the single `slippage` parameter and input and output amounts are checked a single time at the beginning of methods like `depositFromToken`. The guarantees about the end state derived from this single check are virtually guaranteed to be very weak,[5] which leaves the method vulnerable to undesired outcomes and prone to accidental misuse.

Unfortunately, the test suite is not of use in alleviating this issue, because the `MockUniswapV2Router01` contract used as a mock DEX does not exhibit DEX behaviour (for valid reasons) that can and must be expected to occur in a production environment. To the degree that testnet DEXs exhibit similar behaviour as the mainnet ones, manual and/or automated tests on testnets could potentially somewhat mitigate this shortfall.

6.2.3 Potential danger of stuck funds on Grizzly contracts

It is a common occurrence that users erroneously transfer funds to smart contracts involved in DeFi systems. Instantiations of the Grizzly contract allow this to happen via the `receive()` function inherited by `DEX.sol`. This is necessary to perform token-BNB exchanges with the DEX itself. However, this introduces the risk that funds that are sent erroneously to the contract remain forever stuck there.[6]

Indeed, all occurrences where BNB is transferred away from the contract merely perform a “passing on” of funds that were received earlier in the transaction. However, it is also the case that the funds are

[4] A cursory survey of recent confirmed example transactions suggests average gas costs of approximately 140’000 for the DEX functions listed above. Together with a gas price of 7 Gwei and a BNB price of \$500, 16 of those invocations as exhibited by the above trace result in transaction fees of 140’000 * 16 * 7 * 10^{-9} * \$500 = \$8. Of course, this is by no means the total cost of the transaction, but only of its DEX interactions.
[5] For example, the user would need to make sure to sum all of the (partially as of yet unknown) input and output amounts if a particular pair is traded more than once, which is the case for at least three pairs as shown above.
[6] Another source of (albeit tiny amounts of) funds stuck on a DEX contract stems from the two functions `convertEthToTokenLP` and `convertEthToTokenLP`, which each twice convert half of BNB funds received into tokens. If an odd amount of BNB wei were received, one wei will remain on the contract. While insignificant, this illustrates the issue, and a BNB recovery function may protect against other more significant occurrences of this pattern.

transferred *completely*, and the inter-transaction BNB balance of a Grizzly contract should therefore always be zero under normal operations. Thus, it would be possible to include a method with suitable access controls that recovers erroneously received funds on Grizzly contracts.

6.2.4 Functions with large re-entrancy attack surface

All of `Grizzly.deposit`, `Grizzly.depositFromToken`, and `Grizzly.withdrawToToken` contain numerous external function calls, as well as updates to state variables thereafter. This considerable re-entrancy attack surface could be neutered by marking those functions `nonReentrant`, similar to other entry-points in `Grizzly.sol`. Another possibility to mitigate this scenario is to strictly follow the Checks-Effects-Interaction pattern.[7]

We give one example of a re-entrancy scenario here, which does not necessarily result in an advantageous outcome for the attacker, but clearly induces incorrect logical behaviour. It is, however, not unlikely that an exploitative scenario exists, given the large number of candidate entry-points, external calls and state updates. As a reference, Appendix B lists the maximal attack surface of the `_deposit` function that is exemplified below.

Consider the `Grizzly._deposit` function, which is called from both `deposit` and `depositFromToken`, which are, as mentioned above, not re-entrancy protected. It calls `_stakeRewards`, which divides up the shares of profit to be re-invested according to the three different strategies. Afterwards, `_deposit` actually deposits the value in the corresponding strategy, and updates state variables, for example increasing the total deposit counts for the relevant strategy, which will affect subsequent reward staking. However, during `_stakeRewards`, there are numerous occurrences where execution control leaves the Grizzly contract and enters functions on the token A and token B contracts (to use the least predictable scenario), for example when they are exchanged on the DEX. If execution flow re-enters `_deposit` and `_stakeRewards` then, the shares attributable to the different strategies are computed under the old state, without taking the first `_deposit` invocation into account.

6.2.5 Test suite heavily skewed towards integration tests

The codebase features a test suite with 100% coverage. However, this attractive metric hides some issues, which are tightly bound to the monolithic nature of most of the `Grizzly.sol` methods discussed above. Specifically, most of the test suite must be considered to consist of integration tests, even the ones located in the `tests/unit/` folder. This has two concrete reasons.

First and most obviously, most of the important methods are so big, rely on so much state and induce so many side-effects that they simply cannot be unit-tested. There is considerable potential to split mutating and non-mutating functionality into more easily unit-testable methods.

The second, and more easily fixed, problem is that existing potential for unit-tests is frequently unexploited. By example, the `getHoneyMintRewardsInRange` function in `StakingPool.sol` would be a perfect candidate for unit-tests: It computes a pure function based only on its arguments, requires no contract state and induces no side-effects. However, its testing coverage exclusively stems from the fact that it is called by other smart contract functions. Unfortunately, for the reader it is

[7] https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html

orders-of-magnitudes harder to convince himself that the function behaves correctly by way of background invocations in complex integration tests, compared to unit-tests that concern themselves with this function exclusively.

As another illustrative example for the second point, the following (externally accessible) functions in `Grizzly.sol` all have return values which are never checked in tests even though the functions are sometimes invoked in dozens of test locations:

- `deposit`
- `depositFromToken`
- `withdraw`
- `withdrawAll`
- `stakeRewardsForBounty`

This issue is considered medium severity because tight integration tests are much more likely to uncover issues than monolithic integration tests. As another drawback, a larger degree of integration greatly hinders the review process efficiency.

6.3 Low severity

6.3.1 Use of deprecated `_setupRole` function

The constructors of multiple contracts use the `AccessControl._setupRole` function to grant the initial `DEFAULT_ADMIN_ROLE`. As of version 4.4.0 of the OpenZeppelin smart contract suite, this function is deprecated.[8] Instead, `_grantRole` should be called directly, to which `_setupRole` in any case delegates. This concerns the constructors of the following contracts:

- `Grizzly.sol`
- `HoneyBNBFarm.sol`
- `HoneyToken.sol`
- `LaunchSale.sol`
- `Referral.sol`
- `StakingPool.sol`

6.3.2 Unnecessary requirement to deploy contract from multisig contract

The comment in `Grizzly.sol:L16` claims that `DEFAULT_ADMIN_ROLE` will be a 2-of-3 multisig contract. Furthermore, `DEFAULT_ADMIN_ROLE` is granted to `msg.sender` in the constructor. This would imply that the contract is being deployed from a multisig contract, a very cumbersome and expensive procedure. Instead, the role could be granted to an address provided as a constructor argument, like it is done in all other contracts.

[8] See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.4.0/contracts/access/AccessControl.sol#L183

6.3.3 Intended undeployable contract can be made `abstract`

The comment on `BaseConfig.sol:L14` states that the `BaseConfig` contract should always be inherited from and not deployed by itself. Marking the contract as `abstract` is an easy measure to enforce this and avoid accidental deployments.

6.3.4 Unchecked return values on `approve` calls on unknown tokens

The `BaseConfig` constructor performs multiple ERC20 approvals which are necessary for the correct functioning of the `Grizzly` mechanisms which involve token transfers. The `approve` invocations don't check the boolean return value from approve. This is not an issue for tokens that are under the project's control, and known to follow the OpenZeppelin implementation which reverts upon unsuccessful approvals. However, `approve()` is also called on a-priori unknown ERC20 implementations (TokenA and TokenB), which may signal an unsuccessful approval via a `false` return value.

6.3.5 Ineffectual deadlines passed to DEX functions

`UniswapV2Router` functions commonly allow a `deadline` argument, after which the trade will not be executable anymore. The `StakingPool.sol` and `DEX.sol` contracts pass `block.timestamp + 300` as a deadline, presumably with the intention of creating a three-minute window where the trade may be executed. However, `block.timestamp` will always equal the timestamp of the block in which the transaction is mined. Hence, the passed deadline will always be fulfilled, regardless of how long the transaction remained in the mempool. This can be made clear to the reader by passing `block.timestamp` as a deadline instead.

This concerns the following locations:

- `StakingPool.sol:L297`
- `DEX.sol:L42, L47, L58, L87, L96, L123, L135, L145, L168, L199`

6.3.6 Unexpected partial withdraw behaviour

The Standard and Grizzly strategies accrue rewards in Honey and Honey-BNB-Liquidity-Pool tokens proportional to the deposited amount. When a user withdraws parts of his funds (through the `Grizzly.withdraw` function), however, all of their accrued rewards are withdrawn and transferred to the user, hence not accruing compound returns anymore.

This may well be intended behaviour, as it may be prohibitively expensive to maintain the necessary book-keeping to support partial reward withdrawals. For the user, however, it remains somewhat surprising and should be documented.

6.3.7 Granting of unlimited ERC20 allowances

`BaseConfig.sol` prepares the `Grizzly.sol` contract for future interactions with various other contracts by granting them unlimited ERC20 allowances to transfer tokens on its behalf. Whereas this is a convenient way of ensuring future interactions will pass through, and thus slightly saves on gas costs, it introduces other risks if the external contracts are malicious or contain vulnerabilities themselves.

A more secure way of handling this would be to check for the current allowance in-place and to increase it if necessary. This is already done in `DEX.sol:L191`, but unfortunately not to the minimum number necessary, but to the maximum.

Besides the constructor of `BaseConfig.sol`, this concerns `DEX.sol: L191` and `StakingPool.sol:L62`.

This issue is considered low-severity because the contracts that are granted an allowance are either contracts controlled by this project or well-used and presumably audited.

6.3.8 Unnecessary double role check on `claimAdditionalTokens`

The internal function `claimAdditionalTokens` in `HoneyToken.sol` is only called by `claimTokens()`, which already checks that the caller has the `MINTER_ROLE`. The repeated check at `claimAdditionalTokens` is therefore unnecessary and results in increased gas costs.

6.3.9 Wrong error message when updating reward phase parameters

The error message in `StakingPool.sol:L345` wrongly mentions phase one twice.

6.4 Comments

6.4.1 Deferred `require` checks

In general, `require()` checks should be executed as early as possible to keep wasted fees to a minimum. The following `require` checks can be moved further up, usually to the beginning of functions:

- `Grizzly.sol:L334`
- `StakingPool:L104`
- `StableCoinStrategy.sol:L43`
- `StandardStrategy.sol:54`

6.4.2 Unused private variable in `StandardStrategy.sol`

The variable `totalHoneyRewards` in `StandardStrategy.sol` is private and never read. It can be removed.

6.4.3 Missed reuse of variable

The expression on `Grizzly.sol:L469` has already been defined as `currentRewards` above, which can be reused.

6.4.4 Subtle undocumented behaviour of `StakingPool.getHoneyMintRewardsInRange fromBlock` argument

`getHoneyMintRewardsInRange` accepts a range and outputs the rewards accrued within that range. Given its use, it is clear that `fromBlock` is exclusive and `toBlock` is inclusive. It would be helpful for the reader to document this.

6.4.5 Convoluted initial purchase of Honey token during `finishSale`

In the `LaunchSale` contract, the `finishSale` function provides the initial liquidity in both Honey tokens and BNB used to establish a liquidity pool[9] of that pair and use the proceeds from the “open” phase to purchase the initial tokens from that pool. Thus, the caller completely determines the initial purchase price of the Honey token, but through a somewhat convoluted route via the initial liquidity amounts and the Automated-Market-Maker equation `k=x*y`.[10]

The same effect could be achieved more simply by providing the liquidity at the desired ratio in the very end, and not routing the initial sale tokens from the caller to the `LaunchSale` contract, to the pool, and back to the `LaunchSale` contract. The correct `liquidityTokenAmount` provided would ensure that the initial purchasers effectively achieved the initial pool price.

6.4.6 Fallback to dev team referral-giver can be simplified

The code block in `Referral.sol:L83-93` can be simplified substantially by just assigning `DevTeam` to `_referralGiver` if the latter is the zero address.

7. Limitations

Even though the code has been reviewed carefully on a best-effort basis, undiscovered issues can not be excluded. This report does not consist in a guarantee that no undeclared issues remain, nor should it be interpreted as such.

[9] The fact that this call establishes the liquidity pool and doesn’t merely add liquidity to an already existing one is not enforced on a smart contract level. If it already exists, the arguments to `finishSale` need to be carefully chosen for the call to `addLiquidityETH` not to be wasteful

[10] Of course, because no party can provide liquidity on the opposing side, the initial price can be driven up arbitrarily high by the caller.

Appendix A: Test Coverage[11]

The test coverage is 100%.

Appendix B: Re-entrancy attack surface exhibit

The listing below is one of the numerous potential re-entrancy vulnerabilities detected by the Slither static analysis tool.[12] Many of the listed external calls and state updates are benign, for example because the contracts in question and their source code are controlled by the project, and known not to be re-entrant. However, this is not the case for all items, and the listing is intended to demonstrate the considerable attack surface present in Grizzly's complex deposit and withdrawal functions.[13] See issue 6.2.4 for a discussion of this problem. For convenience, we have highlighted one of the many potential pairs of external calls and state updates discussed there.

External calls:

```
- _stakeRewards() (contracts/Grizzly.sol#259)
        - StakingPool.stake(amount) (contracts/Strategy/GrizzlyStrategy.sol#65)
        - StakingContract.deposit(PoolID,0) (contracts/Grizzly.sol#462)
        - HoneyToken.claimTokens(tokens) (contracts/Grizzly.sol#699)
        - tokenAmount = SwapRouter.swapExactETHForTokens{value:
amount}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#164-169)
        - tokenValue = SwapRouter.swapExactETHForTokens{value: amount /
2}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#85-87)
        - tokenAValue = SwapRouter.swapExactETHForTokens{value: amount /
2}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#40-42)
        - tokenBValue = SwapRouter.swapExactETHForTokens{value: amount /
2}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#45-47)
        - tokenInstance.approve(address(SwapRouter),2 ** 256 - 1) (contracts/DEX/DEX.sol#191)
        - (usedToken,usedEth,lpValue) = SwapRouter.addLiquidityETH{value: amount /
2}(token,tokenValue,1,1,address(this),block.timestamp + 300) (contracts/DEX/DEX.sol#89-97)
        - (transferSuccess) = address(DevTeam).call{value: bnbReward - honeyBuybackShare}()
(contracts/Grizzly.sol#624-626)
        - StakingContract.deposit(PoolID,tokenPairLpAmount) (contracts/Grizzly.sol#543)
        - StakingContract.deposit(PoolID,pairLpAmount) (contracts/Grizzly.sol#680)
        - ethAmount = SwapRouter.swapExactTokensForETH(amount,1,pairs,address(this),block.timestamp
+ 300)[1] (contracts/DEX/DEX.sol#194-200)
        - (usedTokenA,usedTokenB,lpValue) =
SwapRouter.addLiquidity(tokenA,tokenB,tokenAValue,tokenBValue,1,1,address(this),block.timestamp +
300) (contracts/DEX/DEX.sol#49-59)
        - (transferSuccess) = address(DevTeam).call{value: bnbReward - pairLpShare}()
(contracts/Grizzly.sol#683-685)
        - StakingPool.rewardLP(honeyBnbLpAmount) (contracts/Grizzly.sol#643)
        - (transferSuccess) = address(DevTeam).call{value: bnbReward - tokenPairLpShare -
honeyBuybackShare}() (contracts/Grizzly.sol#567-569)
```

[11] Measured with the "solidity-coverage" hardhat plugin (version 0.7.16)
[12] https://github.com/crytic/slither
[13] Note that, as mentioned, this listing concerns only one of numerous potential re-entrancy targets

```
        - (transferSuccess) = address(DevTeam).call{value: bnbReward - honeyBuybackShare_scope_0 -
honeyBnbLpShare}() (contracts/Grizzly.sol#655-657)
        - StakingPool.rewardLP(honeyBnbLpAmount) (contracts/Grizzly.sol#580)
        - (transferSuccess) = address(DevTeam).call{value: bnbReward - tokenPairLpShare -
honeyBnbLpShare}() (contracts/Grizzly.sol#592-594)
        - Referral.referralUpdateRewards(mintedHoney) (contracts/Grizzly.sol#508)
- (lpValue,unusedTokenA,unusedTokenB) = convertEthToPairLP(amount,address(TokenA),address(TokenB))
(contracts/Grizzly.sol#261-265)
        - tokenAValue = SwapRouter.swapExactETHForTokens{value: amount /
2}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#40-42)
        - tokenBValue = SwapRouter.swapExactETHForTokens{value: amount /
2}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#45-47)
        - (usedTokenA,usedTokenB,lpValue) =
SwapRouter.addLiquidity(tokenA,tokenB,tokenAValue,tokenBValue,1,1,address(this),block.timestamp +
300) (contracts/DEX/DEX.sol#49-59)
```

External calls sending eth:

```
- _stakeRewards() (contracts/Grizzly.sol#259)
        - tokenValue = SwapRouter.swapExactETHForTokens{value: amount /
2}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#85-87)
        - tokenAmount = SwapRouter.swapExactETHForTokens{value:
amount}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#164-169)
        - tokenAValue = SwapRouter.swapExactETHForTokens{value: amount /
2}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#40-42)
        - tokenBValue = SwapRouter.swapExactETHForTokens{value: amount /
2}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#45-47)
        - (usedToken,usedEth,lpValue) = SwapRouter.addLiquidityETH{value: amount /
2}(token,tokenValue,1,1,address(this),block.timestamp + 300) (contracts/DEX/DEX.sol#89-97)
        - (transferSuccess) = address(DevTeam).call{value: bnbReward - honeyBuybackShare}()
(contracts/Grizzly.sol#624-626)
        - (transferSuccess) = address(DevTeam).call{value: bnbReward - pairLpShare}()
(contracts/Grizzly.sol#683-685)
        - (transferSuccess) = address(DevTeam).call{value: bnbReward - tokenPairLpShare -
honeyBuybackShare}() (contracts/Grizzly.sol#567-569)
        - (transferSuccess) = address(DevTeam).call{value: bnbReward - honeyBuybackShare_scope_0 -
honeyBnbLpShare}() (contracts/Grizzly.sol#655-657)
        - (transferSuccess) = address(DevTeam).call{value: bnbReward - tokenPairLpShare -
honeyBnbLpShare}() (contracts/Grizzly.sol#592-594)
- (lpValue,unusedTokenA,unusedTokenB) = convertEthToPairLP(amount,address(TokenA),address(TokenB))
(contracts/Grizzly.sol#261-265)
        - tokenAValue = SwapRouter.swapExactETHForTokens{value: amount /
2}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#40-42)
        - tokenBValue = SwapRouter.swapExactETHForTokens{value: amount /
2}(1,pairs,address(this),block.timestamp + 300)[1] (contracts/DEX/DEX.sol#45-47)
```

State variables written after the call(s):

```
- grizzlyStrategyDeposit(lpValue) (contracts/Grizzly.sol#273)
        - grizzlyStrategyDeposits += amount (contracts/Strategy/GrizzlyStrategy.sol#44)
- stablecoinStrategyDeposit(lpValue) (contracts/Grizzly.sol#275)
```

```
	- stablecoinStrategyDeposits += currentBalance - currentAmount + amount
(contracts/Strategy/StableCoinStrategy.sol#31)
- standardStrategyDeposit(lpValue) (contracts/Grizzly.sol#271)
	- standardStrategyDeposits = standardStrategyDeposits + currentDeposit -
participantData[msg.sender].amount + amount (contracts/Strategy/StandardStrategy.sol#38-42)
- totalUnusedTokenA += unusedTokenA (contracts/Grizzly.sol#267)
- totalUnusedTokenB += unusedTokenB (contracts/Grizzly.sol#268)
```