9inch

Smart Contract Security Assessment

Sep 10, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of the staking pools of the 9inch protocol. The staking pools combine a Masterchef contract for rewards distribution, and staking pools adapted from PancakeSwap staking contracts on BSC.

SETTING & CAVEATS

The protocol repository is currently private, at 9inchswap/9InchSM. We had previously performed a short audit over the delta of changes of the entire 9inch protocol relative to the underlying forked protocols. The report can be found here. Specifically for the changes to the staking pools, which are over two PancakeSwap public contracts (1,2) on BSC (and not in the PancakeSwap current public repository), we had (in our ealier audit) considered the diff between the original code and commit `0573a9e785d6d971c6f081ca7dd7eccba0052f09` of the 9inch repo. However, several protocol-level considerations were raised and a number of efficiency concerns were mentioned leading to the need to perform this second audit over the changes in the Staking Pools. The current audit was not focused on the delta changes only. We audited the Staking Pools from scratch based on commit `e262046798aec021091bb1a839bf4491c98a58d7`. Fixes were reviewed at commit `e87e2ae8cce6017598290dfff9511b0907313255`. (The latter fix commit also includes other changes that do not pertain to this report's scope.)

The Staking Pools of the previous audit were sharing the same contract for both the pure BBC (i.e. staking token and reward token are both BBC) and non-pure BBC (i.e. staking token is not BBC, while reward is BBC) pools. The changes in the current audit were mainly focused on splitting the pure and the non-pure pools into two separate contracts. Additionally, Flexible and Fixed-Term pools have the same underlying codebase, with the flexible pool contract overriding key parts of the functionality. In total, the auditing effort was over files:

```
contracts/pool/
├─ CakeFlexiblePool.sol
├─ CakePool.sol
├─ TokenFlexiblePool.sol
└─ TokenPool.sol
```

Several of the additions to the PancakeSwap staking contracts implement the ability to stake arbitrary tokens (or rather, more tokens than just the reward token)

Two auditors were commissioned to work on the codebase for 5 working days.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e., issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

PROTOCOL-LEVEL CONSIDERATIONS

General Pool Codebase Considerations

Our earlier audit "*advise[d] a redesign of the staking part of the protocol*" noting that "*[w]ith the current underlying contracts, confidence in the correctness of the final implementation will be low.*"

We also noted:

> *Accordingly, the test suite of the project needs to be substantially extended. Right now only basic interactions with the two staking pools are captured in tests. Extensive interactions, under all corner cases, should be fully tested. Both for pureBBC/non-pureBBC cases, all fees should be tested to be computed correctly (and transferred correctly to the appropriate parties), all balances should be checked in complex scenarios involving multiple stakers, all reward rates should be calculated analytically (in a spreadsheet) and the test cases should check that the code computes them correctly, and several conditions should be exercised (e.g., deposit of 0 amount, which is permitted in the code) to ensure that the computations are stable and do not confer an advantage or penalty to the staker.*

The latter recommendation (for more extensive testing) was not addressed in the revision considered during this audit. In fact, our testing did reveal serious issues (C1) by following exactly the testing strategy outlined above: trying equivalent stakes to ensure that the computations are stable and do not confer an advantage or penalty to the staker. (The issue turned out to be invalid and caused by other changes, but was not possible to dismiss without a more thorough test suite.)

Upon the final fix review (commit `e87e2ae8cce6017598290dfff9511b0907313255`), the test suite was extended, lending much more confidence to the validity of the staking operations.

Pools Deployment & Usage Considerations

There are several considerations when deploying and using the staking pools. We outline them below because they are easy to miss:

- The parent contracts, CakePool and CakeFlexiblePool should only be used with `token == bbc`, i.e., for pure BBC pools. For non-pure BBC pools, the children contracts, TokenPool and TokenFlexiblePool, should be used instead. (This property is not easily enforceable in the code, since the constructors of the parent

contracts are reused in the children.) Otherwise, using a CakePool with `token != bbc` would result in the users being unable to claim their earned BBC rewards since no claim functionality seems to be present and all transfers are made over the staking tokens only.

- The flexible pool contracts, CakeFlexiblePool and TokenFlexiblePool, should be whitelisted in the non-flexible pool contracts (CakePool and TokenPool correspondingly) so that they do not incur withdrawal fees. This is especially important for TokenFlexiblePool, otherwise, a key computation returns the wrong result. Namely, the following function blindly trusts that the credit that the flexible pool has in the parent pool is equal to the amount it has deposited in it.

  TokenFlexiblePool::balanceOf():205

  ```
  function balanceOf() public override view returns (uint256) {
      return totalStakedAmount;
  }
  ```

  We generally advise revising the above function so that it is less brittle: the balance of a flexible pool in terms of the staking token should be exactly what the parent pool believes it is, not some quantity maintained by the flexible pool alone.

- Users can stake with a zero lock duration directly in the parent (non-flexible, i.e., CakePool) pool instead of using the flexible pool (i.e., CakeFlexiblePool). These two staking approaches are not equivalent but subtly different. Additionally, neither one matches the published PancakeSwap documentation of flexible pools, exhibiting small differences.

  Specifically, the non-flexible pools charge a withdrawal fee when someone withdraws their principal earlier than 3 days from their last deposit, except when the user has been whitelisted inside `freeWithdrawFeeUsers`.

  CakePool::withdrawOperation():493

```
uint256 public withdrawFeePeriod = 72 hours; // 3 days

function withdrawOperation(
    uint256 _shares,
    uint256 _amount
) internal virtual {
    ...
    // Calculate withdraw fee
    if (!freeWithdrawFeeUsers[msg.sender] &&
        (block.timestamp < user.lastDepositedTime + withdrawFeePeriod)
    ) {
        uint256 currentWithdrawFee = (currentAmount * withdrawFee) /
            FEE_RATE_SCALE;
        token.safeTransfer(treasury, currentWithdrawFee);
        currentAmount -= currentWithdrawFee;
    }
    token.safeTransfer(msg.sender, currentAmount);
    ...
}
```

Per the earlier bullet item, flexible pools should be whitelisted in the parent pools (non-flexible pools) so that they do not incur withdrawal fees which could affect their internal accounting (see TokenFlexiblePool description).

Notably, the PancakeSwap documentation of flexible pools does not match the behavior in the code. (*9inch did not change that logic in the implementation.*) The documentation of flexible pools states:

Unstaking Fee

- **0.1% if you unstake (withdraw) within 72 hours.**
- Only applies within 3 days of manually staking.
- After 3 days, you can unstake with **no fee**.
- The 3-day timer resets every time you manually stake more CAKE in the pool.

- This fee only applies to manual unstaking: it does not apply to automatic compounding.

As discussed, flexible pools should not incur a withdrawal fee (and do not in the current PancakeSwap setup either).

Furthermore, the current configuration of **non-flexible** pools also incurs no withdrawal fee! The minimum lock duration is 7 days, rendering the withdrawal fee useless, since at withdraw time more than 3 days from staking must have elapsed.

All the above comments lead to the conclusion that this withdrawal fee becomes redundant as it will not be used at any point. The non-flexible pool users can't be charged as the minimum lock duration (7 days) exceeds the 3-day period that the fee defines and the flexible pools are whitelisted in the parent pools meaning that they are not charged any withdrawal fees either.

So, some protocol-level decisions should be made here in order to decide which is the desired behavior and take care of implementing it properly without introducing issues and edge cases like the ones described above.

- Attention is also needed if, in the future, the protocol needs to be refactored and the `MIN_LOCK_DURATION` changes. Having in mind the above, this constant variable in the non-flexible pools code should not be set to anything less than the value of the `withdrawFeePeriod` variable. Otherwise, there could be cases where the non-flexible pool stakers would have to pay withdrawal fees when the locking period ends and they attempt to withdraw their principal and rewards.

  For example, assume a `MIN_LOCK_DURATION` of 2 days and a `withdrawFeePeriod` of 3 days. Then if someone locks his stake for the minimum possible duration (2 days), then he will lose some of his principal and rewards when withdrawing.

CakePool::withdrawOperation:495
TokenPool::withdrawOperation:343

```
// Calculate withdraw fee
if (!freeWithdrawFeeUsers[msg.sender] &&
    (block.timestamp < user.lastDepositedTime + withdrawFeePeriod)
) { ... }
```

The variables of interest that need to be aligned for this to be prevented are the following:

- MAX_WITHDRAW_FEE_PERIOD
- MIN_LOCK_DURATION
- withdrawFeePeriod

If the code maintains the following condition then the problem would be prevented as the withdrawFeePeriod can be set only up to the MAX_WITHDRAW_FEE_PERIOD value.

```
MAX_WITHDRAW_FEE_PERIOD <= MIN_LOCK_DURATION
```

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or

	cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

ID	Description	STATUS
C1	Rewards are not correct, can be manipulated via staging of stake/unstake operations	**DISMISSED (invalid issue, upon further testing)**
The rewards yielded by the staking contracts vary greatly and can confer an unfair profit to a manipulator, at the expense of others. For instance, in the test suite of the repository itself (`cakepools.js`) one can replace the deposit action:		

```
await track(
    CakeVault.connect(alice), 'deposit', parseEther('10'), 86400 * 7)
```

with ten individual deposits of 1 BBC each:

```
await track(
    CakeVault.connect(alice), 'deposit', parseEther('1'), 86400 * 7)

for (let i = 0; i < 9; i++) {
      await track(CakeVault.connect(alice), 'deposit', parseEther('1'), 0)
}
```

The result for the staker is 244.7 instead of 139.1 BBC upon withdrawing.

The same potential for manipulation can be observed by making many small withdrawals instead of a single big one. The effect persists when one changes the staking period and block advancement to more realistic numbers, or when trying larger deposit quantities.

In general, the rewards boosting mechanism of `MasterChefV2` seems to not account correctly for the claimed rewards. It is not clear without further inspection (mostly of code outside the audit scope) what is the source of the error, but it seems to be an error of omission, with the most likely culprit being the removal of function `updateBoostContractInfo` (as well as the calls to it). Via this function, the boost contract adjusts the boost factor (calling the appropriate update function in MasterChef) when a deposit or withdrawal takes place.

```
function updateBoostContractInfo(address _user) internal {
    if (boostContract != address(0)) {
        UserInfo storage user = userInfo[_user];
        uint256 lockDuration = user.lockEndTime - user.lockStartTime;
        IBoostContract(boostContract).onCakePoolUpdate(
            _user,
            user.lockedAmount,
```

```
            lockDuration,
            totalLockedAmount,
            DURATION_FACTOR
        );
    }
}
```

HIGH SEVERITY:

ID	Description	STATUS
H1	Precision is lost for tokens with > 18 decimals	**RESOLVED (check for >= 18 decimals added in commit 2929d7e)**

The revised codebase made several adjustments to address the possibility of tokens with more or fewer than 18 decimal digits. There is still, however, an insidious issue for tokens with > 18 decimals. Both in `TokenPool` and in `TokenFlexiblePool`, the code makes `shares` be denominated in the same decimals as the staking token. The main code that converts staking amounts to shares is:

`TokenPool::depositOperation():227`

```
if (totalShares != 0) {
    ...
} else {
    currentShares = _amount;
}
```

However, this means that the number of shares has the same decimals as the decimals of the staking token. But then, calculations like this lose precision:

```
bbcPerShare += (claimedAmount * 1 ether) / totalShares;
```

E.g., if the staking token has 36 decimals, then `bbcPerShare` will have `0` decimals, i.e., will lose all precision.

Further quantities will then have the wrong number of decimals, since `bbcPerShare` is no longer really denominated in BBC precision. However, notably, even if a quantity derived from `bbcPerShare` happens to have the correct number of decimals, it will have lost precision, likely being rounded to zero.
The issue seems to be addressable by just converting all staking token amounts to 18-decimal-precision before they become share amounts. However, full resolution of the issue also requires adding to the test suite scenarios with tokens with both more and fewer than 18 decimals. The tests should examine whether the final amounts are correct, and not merely whether they have the right number of decimals.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	`CakePool::unlock` should best be `nonReentrant`	**RESOLVED (commit 7136b90)**

The function `CakePool::unlock` allows any user to call it, if operating on the user's own account. The function performs a `depositOperation`, which checks and updates several storage variables. To eliminate the possibility of reentrancy, this function should be declared `nonReentrant`, just like other entry points of `depositOperation`. We have not identified a specific attack vector, however.

`CakePool::unlock():269`

```
function unlock(
    address _user
) public onlyOperatorOrBBCOwner(_user) whenNotPaused {
    UserInfo storage user = userInfo[_user];
    require(
        user.locked && user.lockEndTime < block.timestamp,
        "Cannot unlock yet"
    );
    depositOperation(0, 0, _user);
}
```

LOW SEVERITY:

[No low severity issues]

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	Protocol owner/admins should be trusted	**OPEN**
The protocol owner has privileges for, e.g., pausing the protocol, disabling staking (report item A1), but also withdrawing all reward tokens, in the case of non-pure pools:		

```
function inCaseTokensGetStuck(address _token) public onlyAdmin {
    require(
        _token != address(token),
        "Token cannot be same as deposit token"
    );

    uint256 amount = IERC20(_token).balanceOf(address(this));
    IERC20(_token).safeTransfer(msg.sender, amount);
}
```

It is not straightforward for the protocol owner to appropriate staking tokens, but there may be a combination of actions to do so. Given that the owner is already privileged, we have not considered scenarios of attack-by-owner further.

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them to be sure that they have been taken into account and not merely overlooked.

ID	Description	STATUS
A1	Staking state is finalized upon an `emergencyWithdraw`	**INFO**

In `CakeFlexiblePool`, the `staking` flag cannot be reset to true once set to `false`.

`CakeFlexiblePool::emergencyWithdraw():193`

```
function emergencyWithdraw() public onlyAdmin {
    require(staking, "No staking bbc");
    staking = false;
    parentPool.withdrawAll();
}
```

This seems to be by design and changing it requires more revisions (e.g., to prevent a user from losing their share by trying to unstake when `staking` is false).

A2	Inconsistencies in names and comments	**INFO**

Below is a list of observed inconsistencies or typos in the code:

`CakePool`:

- Inconsistent comment

```
uint256 public UNLOCK_FREE_DURATION = 2 weeks; // 1 week
```

- There are several different scales for fees. E.g., overdue fees are scaled at `e10`, other fees are scaled at `e2` and others at the `FEE_RATE_SCALE` below. (Also this variable could be made immutable, for gas savings.)

```
uint256 public FEE_RATE_SCALE = 10000;
```

- Bad English (also in original):

```
 * @notice Update user share When need to unlock or charges a fee.
```

- The word “fee” is used to mean both “absolute fee” and “fee scale” in this line:

```
uint256 currentWithdrawFee = (currentAmount * withdrawFee)
```

- The functions setOverdueFeeUser and setWithdrawFeeUser do not follow the naming style used in set**Free**PerformanceFeeUser. All these, are setters that whitelist users from being charged by these fee categories.

```
function setFreePerformanceFeeUser(...) { ... }
function set[Free]OverdueFeeUser  (...) { ... }
function set[Free]WithdrawFeeUser (...) { ... }
```

`TokenFlexiblePool`:

- Inconsistent comment (copy-paste):

```
 * @notice Withdraws funds from the BBC Flexible Pool
```

A3	Staking pools should best not be used with tokens that perform call-backs to the sender	**INFO**

The current implementation of deposit operations on all staking pools (`CakePool`, `TokenPool`, `CakeFlexiblePool`, `TokenFlexiblePool`) violates the checks-effects-interactions (CEI) pattern if a `transferFrom` makes a call-back to an untrusted sender of funds. E.g.,

```
    if (_amount > 0) {
        token.safeTransferFrom(_user, address(this), _amount);
        currentAmount = _amount;
    }

    // Calculate lock funds
    if (user.shares > 0 && user.locked) {
        userCurrentLockedBalance = (pool * user.shares) / totalShares;
    ... // many more checks and effects
```

We have not found a reentrancy threat based on this potential attack vector, especially since all key functions seem well-protected by the `nonReentrant` modifier. However, our recommendation is to not use the pools with the (very few) tokens (mainly ERC777 implementations) that make such unusual call-backs to the *sender* of funds.

Recent experience with read-only reentrancies in mature protocols (e.g., Balancer) shows that fully trusting `nonReentrant` flags is not wise: the checks-effects-interactions pattern is the best protection against reentrancy and violating it may result in attacks that are extremely hard to detect, as evident by recent practice.

A4	Thoughts on checks-effects-interactions in withdrawals	INFO

In both `CakePool` and `CakeFlexiblePool`, the `withdraw` operation results in a transfer to an untrusted party (`msg.sender`). For some tokens (more than in A3) this results in a callback to the untrusted party. The code violates the checks-effects-interactions pattern because a few checks and effects take place afterwards:

```
token.safeTransfer(msg.sender, currentAmount);

if (user.shares > 0) {
    user.lastUserActionAmount =
      (user.shares * balanceOf()) /
      totalShares;
} else {
    user.lastUserActionAmount = 0;
}
user.lastUserActionTime = block.timestamp;
```

In this case (unlike in A3), it seems easy to reorder the code so that the transfer operation is last, thus eliminating all possibilities of reentrancy/read-only reentrancy. If such reordering is to take place, the expression "`balanceOf()`" should be replaced with "`balanceOf() - currentAmount`" and there should be a check that the amount transferred was indeed `currentAmount`. (This suggestion should be tested thoroughly if implemented.)

However, we have thoroughly inspected the code that accesses the storage variables checked and updated above, and all current access is in `nonReentrant` functions. Therefore, the current code appears safe.

A5	Storage variable should be declared immutable	INFO

In the `TokenFlexiblePool`, the variable `tokenDecimals` is set only once in the constructor and is supposed to hold the decimals of the staking token. It is then used inside the functions that convert values from or to 18 decimals precision.

This variable should be declared immutable to avoid all SLOADs that otherwise would incur in the conversion functions adding unnecessary gas consumption.

TokenFlexiblePool::tokenDecimals:15

```
// Dedaub: tokenDecimals should be made immutable to avoid the SLOADs in
//         toEther and fromEther functions
uint8 private tokenDecimals;

function toEther(uint256 _amount) internal view returns(uint256) {
    if(tokenDecimals < 18)
        return _amount * 10 ** (18 - tokenDecimals);
    else if(tokenDecimals > 18)
        return _amount / 10 ** (tokenDecimals - 18);
    return _amount;
}

function fromEther(uint256 _amount) internal view returns(uint256) {
    if(tokenDecimals < 18)
        return _amount / 10 ** (18 - tokenDecimals);
    else if(tokenDecimals > 18)
        return _amount * 10 ** (tokenDecimals - 18);
    return _amount;
}
```

A6	Gas inefficiencies that are easy to address	**INFO**

There are some instances in the code where gas inefficiencies arise and can be addressed easily.

`CakePool`:

- The second line makes a storage load of a value just stored one line above:

```
user.lockedAmount = userCurrentLockedBalance;
totalLockedAmount += user.lockedAmount;
```

- The second `require` can be put in an `else` branch of the `if`, so as to avoid loading `user.shares` unnecessarily if `_shares` is zero:

```
if(_shares==0 && _amount > 0)
      require(_amount > MIN_WITHDRAW_AMOUNT,
        "Withdraw amount must be greater than MIN_WITHDRAW_AMOUNT");
// else
require(_shares <= user.shares, "Withdraw amount exceeds balance");
```

A7	Code can be streamlined	**DISMISSED (invalid)**

In both `CakePool` and `TokenPool`, the following code in `withdrawOperation` can be improved from:

```
uint256 currentShare = _shares;
uint256 sharesPercent = (_shares * PRECISION_FACTOR_SHARE) /
            user.shares;
...  // A
if (_shares == 0 && _amount > 0) {
      ...  // B
} else {
      currentShare = (sharesPercent * user.shares)/PRECISION_FACTOR_SHARE;
}
```

to just:

```
uint256 currentShare = _shares;
...  // A
if (_shares == 0 && _amount > 0) {
      ...  // B
```

```
}
```

A8	Unused variables, functions	INFO

The following variables or functions appear unused. (Note that the list may not be exhaustive. Also, other variables/functions are not read in the code but may be accessed externally, since they are set in the code and are publicly readable.)

- `TokenFlexiblePool::feeDebt`
- `TokenFlexiblePool::payFee`

A9	Function may return a result that is not realistic	INFO

The `CakePool::calculateWithdrawFee` function returns a result even when the user is still in `locked` mode. This means that the result may not correspond to any fee that the user will observe in practice. Care should be taken to not cache or otherwise rely upon this result in off-chain code.

A10	Magic constant	INFO

Our recommendation is for all numeric constants to be given a symbolic name at the top of the contract, instead of being interspersed in the code. In `TokenFlexiblePool::withdraw`:

```
withdrawAmount = (withdrawAmount * withdrawAmountBooster) / 10000;
```

A11	Compiler bugs	INFO

The code is compiled with Solidity `0.8.19`. This version has some known bugs, which we do not believe affect the correctness of the contracts.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.