Zellic

PancakeSwap Aptos

Smart Contract Security Assessment

November 17, 2022

Prepared for:

PancakeSwap

Prepared by:

Aaron Esau and Varun Verma

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for PancakeSwap from November 14th to November 17th, 2022.

Our general overview of the code is that it was well-organized and structured. The code coverage is reasonable, and tests are included for the majority of the functions. The documentation was adequate, although it could be improved. The code was easy to comprehend, and in most cases, intuitive.

We applaud PancakeSwap for their attention to detail and diligence in maintaining high code quality standards in the development of PancakeSwap Aptos.

Zellic thoroughly reviewed the PancakeSwap Aptos codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

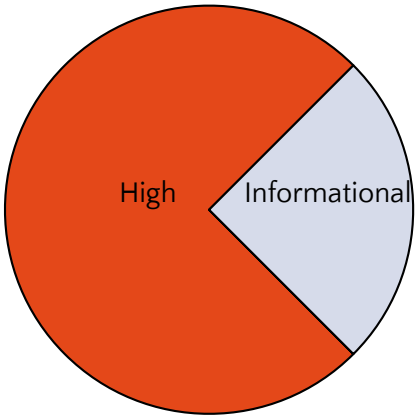
Specifically, taking into account PancakeSwap Aptos's threat model, we focused heavily on the issues of PancakeSwap's primary concern, including funds being drained by unauthorized outsiders, denial of service, funds being locked up due to incorrect configuration, and accounting bugs.

During our assessment on the scoped PancakeSwap Aptos contracts, we discovered four findings. Fortunately, no critical issues were found. Of the four findings, three were of high impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the audit for PancakeSwap's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	0
Low	0
Informational	1



2 Introduction

2.1 About PancakeSwap Aptos

PancakeSwap is the leading decentralized exchange on BNB Smart Chain with a suite of ecosystem products aiming to be the one-stop DeFi solution for every DeFi newcomer. PancakeSwap is now launching on the Aptos blockchain.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase in general. We

look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so forth.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their importance may differ. This varies based on numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

PancakeSwap Aptos Modules

Repository	https://github.com/chefrabbid/aptos-contracts
Versions	`889c0eba94e012a0b627470c349df3ee38d6300c`
Programs	• pancake::smart_chef.move • utils::math.move • utils::u256.move
Type	Move
Platform	Aptos

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of eight person-days. The assessment was conducted over the course of four calendar days.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Esau, Engineer
aaron@zellic.io

Varun Verma, Engineer
varun@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

November 14, 2022	Start of primary review period
November 17, 2022	End of primary review period

3 Detailed Findings

3.1 Precision factor is not precise enough

- **Target**: pancake::smart_chef
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The `precision_factor` used to avoid division precision errors is not large enough to mitigate truncation to zero errors.

Impact

The formula for `acc_token_per_share` is calculated by

```
acc_token_per_share = acc_token_per_share + (reward * precision_factor) /
    total_stake;
```

and the `precision_factor` is calculated by

```
let precision_factor = math::pow(10, (16 - reward_token_decimal));
```

In the case that `total_stake` is greater than `(reward * precision_factor)`, which can happen if the average user deposits 100 `StakeToken` coins of 12 decimals, or one factor smaller of a token of one decimal lower, `acc_token_per_share` can get truncated to zero via the division. This disables users from getting rewards, with the threat being highly likely for any coin greater than 11 decimals.

In a proof of concept, we recreated such a scenario by first minting some users an average of `100 coins` of a token of 12 decimals via a pseudo-random number generator and staking them.

```
while (i < 30) {
    let minted_amount = *vector::borrow_mut(&mut random_num_vec, i) *
    pow(10, coin_decimal_scaling);
    test_coins::register_and_mint<TestCAKE>(&coin_owner,
    vector::borrow<signer>(&signers_vec, i), minted_amount);
```

```
    i = i + 1;
};

while (i < 30) {
    deposit<TestCAKE, TestBUSD, U0>(vector::borrow<signer>(&signers_vec,
    i),
    coin::balance<TestCAKE>(signer::address_of(vector::borrow<signer>(&signe
    i))));
    i = i + 1;
};
```

We then increased the timestamp to allow rewards to accrue via the following:

```
timestamp::update_global_time_for_test_secs(start_time + 30);
```

And finally, we retrieved the reward for each staker through the following code,

```
while (i < 30) {
    let user_pending_reward = get_pending_reward<TestCAKE, TestBUSD,
    U0>(signer::address_of(vector::borrow<signer>(&signers_vec, i)));
    debug::print<u64>(&user_pending_reward);
    i = i + 1;
};
```

in which all `user_pending_reward` outputted a zero value, which indicated no users received any rewards.

We provided a full test for PancakeSwap Finance for reproduction.

Recommendations

We recommend using a higher precision factor such as in the EVM version or restricting the maximum decimal of the reward and stake coin to no greater than 10.

Remediation

PancakeSwap acknowledged the finding and resolved it in commits 19a751d7 and 390c8744.

3.2 Excessive rewards allocation leads to DOS

- **Target**: pancake::smart_chef
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: High

Description

First, understand the following variables in the `add_reward` function:

- `pool_info.historical_add_reward`: the total amount of reward LP that the admin has deposited.
- `pool_info.reward_per_second`: the maximum amount of reward LP the admin is allowed to deposit per second; the following assertion in `add_reward` requires that the admin's new deposit does not cause the historical average reward LP deposit per second to exceed `pool_info.reward_per_second`:

```
pool_info.historical_add_reward = pool_info.historical_add_reward +
    amount;
assert!(pool_info.reward_per_second * (pool_info.end_timestamp -
    pool_info.start_timestamp) ≥ pool_info.historical_add_reward,
    ERROR_REWARD_MAX);
```

When calculating `pool_info.acc_token_per_share` using the `cal_acc_token_per_share` function, we see that the reward-to-stake token ratio is based off of `reward_per_second`, which is the maximum reward LP deposit rate—not the actual deposit rate—multiplied by the period `multiplier`:

```
let reward = u256::from_u128((reward_per_second as u128) * (multiplier as
    u128));
```

Because the ratio is calculated using the maximum reward and the admin can deposit less than this amount, reward payouts may be too large, meaning the protocol can potentially be in deficit, leading to an underflow abort given enough withdrawals. This would require users to `emergency_withdraw` and forfeit rewards to save their funds.

The reward supply can be lower than it should be; the aforementioned `add_reward` assertion requires that the "limit >= actual", meaning the "actual supply is always <= the limit". For the reward supply to be sufficient, it must always be equal to the limit.

We created tests to prove the existence of this bug and provided them to the customer separately from this report.

Impact

Certain conditions may lead to users having to save funds by calling `emergency_withdraw`, forfeiting their rewards.

The following scenarios increase the likelihood of triggering the bug:

- smaller amounts deposited by an admin, at least less than the reward LP deposit limit (`pool_info.reward_per_second`).
- more users depositing at the pool start, fewer users depositing after pool start (`pool_info.start_timestamp`).
- more users withdrawing rewards later in the pool period.

Recommendations

Rather than using the reward token LP deposit limit when calculating `pool_info.acc_token_per_share`, use the actual reward token LP balance.

Note that admins may deposit reward token LP after the pool has started.

Remediation

PancakeSwap remediated the issue by taking out `pool_info.historical_add_reward` in commit 19a751d7.

3.3 Potential overflow in the `add_reward` function

- **Target**: pancake::smart_chef
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: High

Description

In the `add_reward` function, there exists the following assertion that checks that the admin is not depositing more reward LP than the `pool.historical_add_reward` limit:

```
assert!(pool_info.reward_per_second * (pool_info.end_timestamp -
    pool_info.start_timestamp) ≥ pool_info.historical_add_reward,
    ERROR_REWARD_MAX);
```

Note that multiplying these two u64 values may result in an integer overflow—especially since `pool_info.reward_per_second` will likely be a large number, particularly for reward tokens of larger decimals, and the maximum u64 value is only 20 decimals.

Impact

It is possible for an admin to configure a pool in a way that admins cannot deposit reward LP using the `add_reward` function.

Recommendations

Cast the multiplier and multiplicand values to u256 before the operation:

```
assert!(pool_info.reward_per_second * (pool_info.end_timestamp -
    pool_info.start_timestamp) ≥ pool_info.historical_add_reward,
    ERROR_REWARD_MAX);
assert!(u256::as_u128(u256::mul(
        u256::from_u64(pool_info.reward_per_second),
        u256::sub(
            u256::from_u64(pool_info.end_timestamp),
            u256::from_u64(pool_info.start_timestamp)
        )
    )) ≥ pool_info.historical_add_reward as u128, ERROR_REWARD_MAX);
```

Remediation

PancakeSwap remediated the issue by taking out `pool_info.historical_add_reward` in commit 19a751d7.

3.4 Type checking is unnecessarily complicated

- **Target**: pancake::smart_chef
- **Category**: Gas Optimization
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The following type comparison in the `create_pool` function is unnecessarily complicated:

```
let stake_token_type_name_bytes =
    *string::bytes(&type_info::type_name<StakeToken>());
let reward_token_type_name_bytes =
    *string::bytes(&type_info::type_name<RewardToken>());
assert!(!comparator::is_equal(&comparator::compare_u8_vector(
    stake_token_type_name_bytes, reward_token_type_name_bytes)),
    ERROR_SAME_TOKEN);
```

Impact

The transaction may consume more gas than necessary, the code may be less readable, and a more complicated operation may introduce more risk of failure.

Recommendations

Replace the assertion with the following simpler, more comprehensible code:

```
let stake_token_type_name_bytes = *string::bytes(
    &type_info::type_name<StakeToken>());
let reward_token_type_name_bytes = *string::bytes(
    &type_info::type_name<RewardToken>());
assert!(!comparator::is_equal(&comparator::compare_u8_vector(
    stake_token_type_name_bytes, reward_token_type_name_bytes)),
    ERROR_SAME_TOKEN);
assert!(type_info::type_name<StakeToken>()
    ≠ type_info::type_name<RewardToken>(), ERROR_SAME_TOKEN);
```

Remediation

PancakeSwap resolved the issue in commit 19a751d7.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Pseudo fuzz testing

In order to test the protocol in a more comprehensive manner, a form of pseudo fuzz testing was applied. In our example, we generated 30 random signers and numbers

```
let signers_vec: vector<signer> = vector::empty();
let random_num_vec: vector<u64> = vector::empty();
let i = 0u64;
let randomization_seed = 0x4333447;
let coin_decimal_scaling = 8;
while (i < 30) {
    let (signer, _) = account::create_resource_account(dev,
    std::bcs::to_bytes<u64>(&(i+randomization_seed)));
    let vec =
    hash::sha3_256(std::bcs::to_bytes<u64>(&(i+randomization_seed)));
    let pseudo_random = (vector::pop_back<u8>(&mut vec) as u64);
    vector::push_back<u64>(&mut random_num_vec, pseudo_random);
    vector::push_back<signer>(&mut signers_vec, signer);
    i = i + 1;
};
```

and used them to deposit and withdraw at randomized time intervals

```
timestamp::update_global_time_for_test_secs(start_time + 15);
i = 0;
while (i < 15) {
    deposit<TestCAKE, TestBUSD, U0>(vector::borrow<signer>(&signers_vec,
    i),
    coin::balance<TestCAKE>(signer::address_of(vector::borrow<signer>(&signers_vec,
    i))));
    i = i + 1;
};
```

```
timestamp::update_global_time_for_test_secs(start_time + 16);

while (i < 15) {
    let stake_amount = get_user_stake_amount<TestCAKE, TestBUSD,
    U0>(signer::address_of(vector::borrow<signer>(&signers_vec, i)));
    withdraw<TestCAKE, TestBUSD, U0>(vector::borrow<signer>(&signers_vec,
    i), stake_amount);
    i = i + 1;
};
let (_, expected_total_reward_end, _, _, _, _, _) =
    get_pool_info<TestCAKE, TestBUSD, U0>();
assert!(expected_total_reward_start > expected_total_reward_end, 1);
```

while asserting that core invariants were still withheld. For instance, we asserted that the expected total reward and expected total stake matched accordingly to the user's actions of depositing and withdrawing.

The full test has been provided to PancakeSwap Finance.

4.2 Admin transfer fail-safe

The setting of a new admin is enabled via the following code:

```
public entry fun set_admin(sender: &signer, new_admin: address) acquires
    SmartChefMetadata {
    let sender_addr = signer::address_of(sender);
    let metadata
    = borrow_global_mut<SmartChefMetadata>(RESOURCE_ACCOUNT);
    assert!(sender_addr == metadata.admin, ERROR_ONLY_ADMIN);
    metadata.admin = new_admin;
}
```

However, in the case that a wrong `new_admin` address is supplied, the admin is lost forever. We suggest making this a two-step process where the `new_admin` must claim their admin role to avoid such a scenario.

4.3 General coding practices

There are a few minor instances where better general coding practices can be applied. For example, the following line of code is used numerously throughout the contract for admin access-control and could be refactored to its own function:

```
let metadata = borrow_global<SmartChefMetadata>(RESOURCE_ACCOUNT);
assert!(sender_addr == metadata.admin, ERROR_ONLY_ADMIN);
```

In addition, the code contains a custom math library `pancake::math`; however, a standard math library is provided by the Aptos framework that supports popular `U128` and `U64` math functions, which would be preferable to use over a custom implementation.

More so, variables such as start_timestamp, end_timestamp, pool_limit_per_user, and time_for_user_limit are in the unit of measurement of seconds, and it would be better practice to reflect this in their variable names, for example `pool_limit_per_user_seconds`.

Another instance is that the setter functions check that the instance of a pool exists with

```
assert!(exists<PoolInfo<StakeToken, RewardToken, UID>>(RESOURCE_ACCOUNT),
    ERROR_POOL_NOT_EXIST);
```

However, the getter functions such as `get_pool_info` do not have this check, which could be helpful from a code consistency standpoint.

Lastly, the code involves the use of a custom `U256` datatype, which is necessary as Aptos does not have built-in support for this data type yet. However, the Aptos team says support for `U256` is coming soon, and we recommend using it when it is released instead a custom implementation.

4.4 Presence of emergency reward withdrawal function

Note that there exists an emergency function `emergency_reward_withdraw` for the admin to withdraw rewards. This function may be called at any time—including during the pool's "open" period (between `pool_info.start_timestamp` and `pool_info.end_timestamp`).

If an admin called this function, no staking token LPs would receive rewards for their tokens unless the admin redeposited the reward token, and the staking LPs would have to withdraw using `emergency_withdraw` because the `withdraw` function would no

longer work since it cannot pay out rewards.

5 Formal Verification

The Move prover allows for formal specifications to be written on Move code, which can provide guarantees on function behavior as these specifications are exhaustive on every possible input case.

During the audit period, we provided PancakeSwap with Move prover specifications, a form of formal verification.

The following specifications were provided.

5.1 pancake::smart_chef

Verifies setter function of `update_reward_per_second`:

```
spec update_reward_per_second {
    ensures borrow_global_mut<PoolInfo<StakeToken, RewardToken,
    UID>>(RESOURCE_ACCOUNT).reward_per_second == reward_per_second;
}
```

Verifies setter function of `update_pool_limit_per_user`:

```
spec update_pool_limit_per_user {
    ensures time_for_user_limit ==> pool_limit_per_user ==
    borrow_global_mut<PoolInfo<StakeToken, RewardToken,
    UID>>(RESOURCE_ACCOUNT).pool_limit_per_user;

    ensures !time_for_user_limit ==> 0 ==
    borrow_global_mut<PoolInfo<StakeToken, RewardToken,
    UID>>(RESOURCE_ACCOUNT).pool_limit_per_user && 0 ==
    borrow_global_mut<PoolInfo<StakeToken, RewardToken,
    UID>>(RESOURCE_ACCOUNT).time_for_user_limit;

}
```

Verifies permissions on contract upgrading:

```
spec upgrade_contract {
```

```
    pragma aborts_if_is_partial = true; // some abort conditions on
    code::publish_package_txn
    aborts_if !exists<SmartChefMetadata>(RESOURCE_ACCOUNT);
    aborts_if signer::address_of(sender) ≠
    borrow_global<SmartChefMetadata>(RESOURCE_ACCOUNT).admin;
}
```

Verifies coin of type X exists after usage of the `check_or_register_coin_store` function:

```
spec check_or_register_coin_store {
    ensures exists<coin::CoinStore<X>>(signer::address_of(sender));
}
```

Verifies permissions on setting a new admin:

```
spec set_admin {
    aborts_if !exists<SmartChefMetadata>(RESOURCE_ACCOUNT);
    aborts_if signer::address_of(sender) ≠
    borrow_global<SmartChefMetadata>(RESOURCE_ACCOUNT).admin;
}
```

6 Threat Model

The purpose of this section is to provide a full threat model description for each function.

As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

6.1 File/Module: `pancake:smart_chef`

Function: `create_pool()`

Intended behavior:

Creating a staking pool, callable by admin only, based on a StakeToken, RewardToken and UID based on some configurable params.

Branches and code coverage:

Intended branches:

- Pool configuration is valid and therefore created
  - ☑ Test coverage

Negative behavior:

- pool already exists
  - ☑ Negative test?
- start-time stamp less than end-time stamp, and start timestamp is greater than now
  - ☑ Negative test?
- Stake or reward token not initialized
  - ☑ Negative test?
- Creator is not admin
  - ☑ Negative test?
- reward token equal to stake token
  - ☑ Negative test?

Preconditions:

- SmartChefMetadata exists for the `RESOURCE_ACCOUNT`

Inputs:

- `time_for_user_limit`:
  - **Control**: None
  - **Authorization**: N/A
  - **Impact**: This value can be configured to be anything by the admin. Perhaps should have a lower bound and upper bound threshold
- `pool_limit_per_user`:
  - **Control**: If `time_for_user_limit` > 0 and then `pool_limit_per_user` must be greater than zero
  - **Authorization**: N/A
  - **Impact**: Configured in accordance to `time_for_user_limit`
- `start_timestamp`:
  - **Control**: Must be greater than now and less than the end timestamp
  - **Authorization**: N/A
  - **Impact**: Only valid start and end times can be configured
- `reward_per_second`:
  - **Control**: N/A
  - **Authorization**: N/A
  - **Impact**: Reward per second could possibly be misconfigured to be high or low, should have some form of upper and lower bound threshold
- `admin: &signer`:
  - **Control**: Owns the private key corresponding to the address of the signer
  - **Authorization**: Address of admin must be the metadata's admin
  - **Impact**: Only admins can call this function

External call analysis

- `account::create_signer_with_capability()`
  - **What is controllable?** nothing
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable
  - **What happens if it reverts, reenters, or does other unusual control flow?** Pool is not created under revert case, does not do any unusual control flow because this function was created by Aptos Core

—

Function: `add_reward()`

Intended behavior:

Add a reward amount to a specific pool.

Branches and code coverage:

Intended branches:

- Admin is able to add reward
  ☑ Test coverage

Negative behavior:

- Reward max cannot be added
  ☑ Negative test?
- Non admin cannot add reward
  ☑ Negative test?

Preconditions:

- SmartChefMetaData is initialized
- Pool of that specific StakeToken, RewardToken and UID is initialized

Inputs:

- `amount`:
  - **Control**: `pool_info.reward_per_second * (pool_info.end_timestamp - pool_info.start_timestamp ≥ pool_info.historical_add_reward`
  - **Checks**: N/A
  - **Impact**: Admin is rate limited to much how reward they can deposit
- `signer`:
  - **Control**: N/A
  - **Checks**: Admin auth only
  - **Impact**: Only admin can call this function

Function call analysis

- `transfer_in()`
  - **What is controllable?** Amount
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?**

Reward is not added under revert condition, no unusual control flow

—

Function: `emergency_reward_withdraw()`

Intended behavior:

Transfer all the reward out to the admin.

Branches and code coverage:

Intended branches:

- Admin can successfully withdraw reward
  ☑ Test coverage

Negative behavior:

- Non admin cannot withdraw reward
  ☑ Negative test?

Preconditions:

- SmartChefData is initialized
- Pool exists

Inputs:

- `admin`:
  - **Control**: N/A
  - **Checks**: Admin Auth only
  - **Impact**: Only an admin can call this function

Function call analysis

- `transfer_out()`
  - **What is controllable?** Nothing, all retrieved from State data
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Would only revert if the reward function param was greater than the value of `total_reward_token`, an impossible scenario to occur. No unusual control flow

—

Function: `stop_reward()`

Intended behavior:

Set the pool's end timestamp to now.

Branches and code coverage:

Intended branches:

- Admin calls function and pool is ended
  - ☑ Test coverage

Negative behavior:

- Non-existent pool is attempted to be ended
  - ☐ Negative test?
- Non admin calls function and fails
  - ☐ Negative test?

Preconditions:

- SmartChefMeta is initialized for the RESOURCE_ACCOUNT

Inputs:

- `signer`:
  - **Control**: N/A
  - **Checks**: Admin only
  - **Impact**: Function is callable by admin only

Function call analysis

—

Function: `emergency_withdraw()`

Intended behavior:

Allow the user to withdraw their staked tokens and abort any reward accrual.

Branches and code coverage:

Intended branches:

- Staked tokens are transferred back to the user
  ☑ Test coverage

Negative behavior:

- User cannot emergency withdraw for a pool they are not a participant of
  ☐ Negative test?
- User cannot emergency withdraw for another user's staked tokens
  ☐ Negative test?

Preconditions:

- SmartChefMeta is initialized for the RESOURCE_ACCOUNT

Inputs:

- `signer`:
  - **Control**: N/A
  - **Checks**: User holds the private key corresponding to the address of the signer
  - **Impact**: Only the address that corresponds to the signer's staked amount is withdrawn

Function call analysis

- `transfer_out()`
  - **What is controllable?** Nothing, the account comes from the signer
  - **If return value controllable, how is it used and how can it go wrong?** N/A
  - **What happens if it reverts, reenters, or does other unusual control flow?** Funds are not transferred back to the user under a revert scenario and would only revert if `pool_info.total_staked_token` does not have enough coins, which should never be the case

—

Function: `deposit()`

Intended behavior:

Allow the user to deposit StakeTokens into a pool.

Branches and code coverage:

Intended branches:

- User can deposit into a specific pool of type <StakeToken, RewardToken, UID>
  ☑ Test coverage

Negative behavior:

- User cannot deposit above pool_limit if that configuration is applied
  ☑ Negative test?
- User cannot deposit into pool that doesn't exist
  ☑ Negative test?

Preconditions:

- Specific pool is initialized

Inputs:

- `amount`:
  - **Control**: User has a balance of that amount of stake token
  - **Checks**: N/A
  - **Impact**: User can deposit any amount of tokens that they hold of a type into the pool
- `account`:
  - **Control**: Any user
  - **Authorization**: None
  - **Impact**: Permissible for anybody to deposit into the pool

Function call analysis

- `reward_debt()`
  - **What is controllable?** amount
  - **If return value controllable, how is it used and how can it go wrong?** It can go wrong if the value outputted is smaller than the real calculation, which would therefore indicate the user received more rewards than they were credited for.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No unusual control flow
- `transfer_in()`
  - **What is controllable?** Amount
  - **If return value controllable, how is it used and how can it go wrong?** No re-

turn value, can go wrong if the amount is transferred in was not the amount the user supplied, that is not the case
  - **What happens if it reverts, reenters, or does other unusual control flow?** Total_stake is not updated under revert condition, no unusual control flow
- `transfer_out()`
  - **What is controllable?** Pending reward by proxy of amount
  - **If return value controllable, how is it used and how can it go wrong?** It can go wrong if it computes the reward incorrectly, and therefore transferring the wrong amount of funds to the user
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reward is given to the user under revert condition and no unusual control flow
- `cal_pending_reward()`
  - **What is controllable?** user_info.amount
  - **If return value controllable, how is it used and how can it go wrong?** It can go wrong if the user transfer's enough funds to make the total_stake be greater than reward * precision, causing a division truncation error
  - **What happens if it reverts, reenters, or does other unusual control flow?** User cannot deposit under revert condition. The while loops in the U256 library present the majority of the complexity control flow
- `update_pool()`
  - **What is controllable?** The specific `pool_info` corresponding to the StakeToken/RewardToken/UID that the user wants to deposit into
  - **If return value controllable, how is it used and how can it go wrong?** No return value, can go wrong if the accounting `cal_acc_token_per_share` is incorrect
  - **What happens if it reverts, reenters, or does other unusual control flow?** Pool is not updated under revert condition, no unusual control flow

7 Audit Results

At the time of our audit, the code was deployed to mainnet.

During our audit, we discovered four findings. Of these, three were high risk and one was a suggestion (informational). PancakeSwap acknowledged all findings and implemented fixes.

7.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.