Zellic

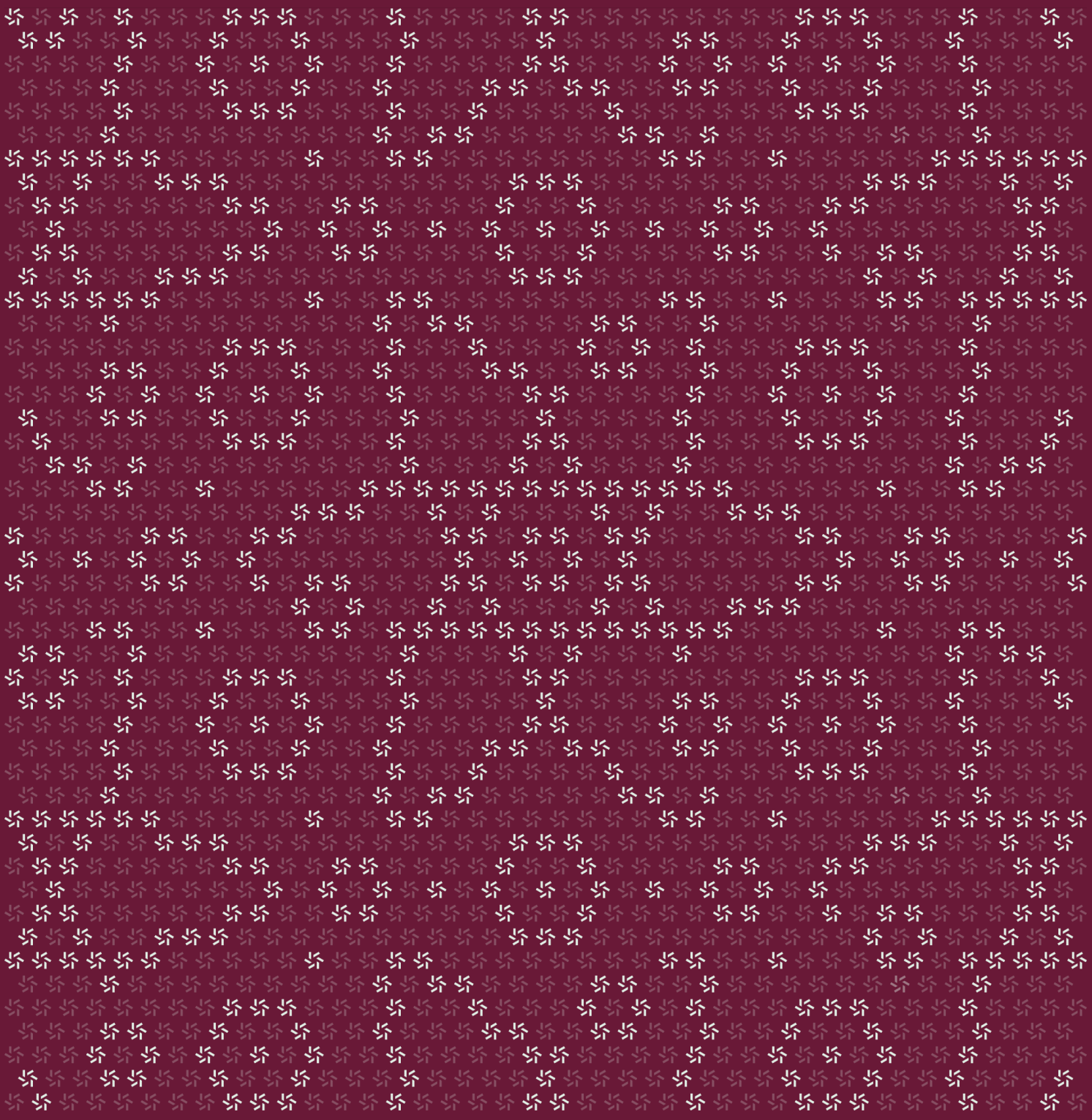
Prepared for
Koga
Dojoswap Labs, PTE

Prepared by
Ayaz Mammadov
Avraham Weinstock
Zellic

October 8, 2024

DojoSwap

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Dojoswap Labs, PTE from September 23rd to September 27th, 2024. During this engagement, Zellic reviewed DojoSwap's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are the rewards from staking correct?
- Are the allocations of launchpad sales correct?
- Can staked funds be lost?
- Are admin or governance operations correctly authenticated?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

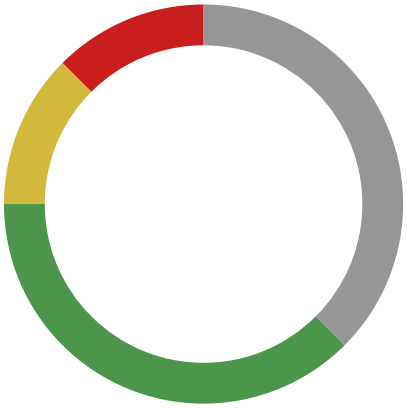
1.4. Results

During our assessment on the scoped DojoSwap crates, we discovered eight findings. One critical issue was found. One was of medium impact, three were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Dojoswap Labs, PTE in the Discussion section (4. ↗).

Breakdown of Finding Impacts

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



2. Introduction

2.1. About DojoSwap

Dojoswap Labs, PTE contributed the following description of DojoSwap:

> DojoSwap is the #1 Amm dex on Injective.

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 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. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the crates.

Nondeterminism. Nondeterminism is a leading class of security issues on Cosmos. It can lead to consensus failure and blockchain halts. This includes but is not limited to vectors like wall-clock times, map iteration, and other sources of undefined behavior (UB) in Go.

Arithmetic issues. This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

Complex integration risks. Several high-profile exploits have been the result of unintended consequences when interacting with the broader ecosystem, such as via IBC (Inter-Blockchain Communication Protocol). Zellic will review the project's potential external interactions and summarize the associated risks. If applicable, we will also examine any IBC interactions against the ICS Specification Standard to look for inconsistencies, flaws, and vulnerabilities.

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. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly 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 various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped crates itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

DojoSwap Crates

Type	Rust
Platform	CosmWasm
Target	dojoswap-contracts-public
Repository	https://github.com/dojo-trading/dojoswap-contracts-public ↗
Version	`4bef3041338a689bf8c510e752f10e102a52cbfa`

Programs	`dojoswap_factory`
	`dojoswap_pair`
	`dojoswap_router`
	`dojoswap_staking`
	`dojoswap_token`
	`launchpad`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.5 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov
Engineer
ayaz@zellic.io ↗

Avraham Weinstock
Engineer
avi@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 23, 2024	Start of primary review period
September 27, 2024	End of primary review period

3. Detailed Findings

3.1. Deposit amount is not validated against message funds

Target	launchpad/src/contract.rs		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In the launchpad contract's `deposit` function, `ExecuteMsg::Deposit`'s `amount` field is not validated against the amount of funds actually sent by the message.

Impact

Depositors can specify arbitrarily large amounts, obtaining an arbitrarily large fraction of the `offering_token`.

Recommendations

Validate the `amount` field against the message info's fund's `amount`.

```
     if info.funds.len() != 1 || info.funds[0].denom != state.raising_denom {
         return Err(StdError::generic_err("Wrong denom"));
     }

+ if info.funds[0].amount != amount {
+ return Err(StdError::generic_err("Wrong amount"));
+ }
```

Remediation

This issue has been acknowledged by Dojoswap Labs, PTE, and a fix was implemented in commit `ce55f60d` ↗.

3.2. The `FEE_COLLECTOR` address can drain `dojoswap_pair` contracts

Target	dojoswap_staking/src/contract.rs		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The `admin_configure` function permits the `FEE_COLLECTOR` address to set the `asset_infos` and `asset_decimals` of the pair arbitrarily. By setting one side of `asset_infos` to a worthless token that an attacker mints, an attacker can swap their worthless token to drain the deposits for the side that was left unmodified. This process can then symmetrically be used with the other side of the pair to drain its deposits as well.

Impact

If the keys for the `FEE_COLLECTOR` address are stolen, or if the `FEE_COLLECTOR` keys are misused, all the value stored in pair contracts can be drained.

Recommendations

Do not allow `admin_configure` to modify `asset_infos` or `asset_decimals` if there are any deposits for the pair on either side.

Remediation

This issue has been acknowledged by Dojoswap Labs, PTE, and a fix was implemented in commit `ce55f60d` ↗.

The patch removes the `AdminConfigure` message and `admin_configure` function from `dojoswap_pair`.

3.3. Final withdraw sends tokens to itself

Target	launchpad/contract.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

At the end of the `final_withdraw`, it is stated that the offering is sent to the admin, but it actually sends the final offering to the contract itself instead.

```
// Transfer offering tokens to admin
if offer_amount > Uint128::zero() {
    messages.push(CosmosMsg::Wasm(WasmMsg::Execute {
        contract_addr: state.offering_token.to_string(),
        msg: to_json_binary(&Cw20ExecuteMsg::Transfer {
            recipient: env.contract.address.to_string(),
            amount: offer_amount,
        })?,
        funds: vec![],
    }));
}
```

Impact

The raised offering tokens are not sent to the correct destination.

Recommendations

Ensure the offering tokens are sent to the correct destination.

Remediation

This issue has been acknowledged by Dojoswap Labs, PTE, and a fix was implemented in commit ce55f60d ↗.

The relevant section of the patch is:

```
        messages.push(CosmosMsg::Wasm(WasmMsg::Execute {
            contract_addr: state.offering_token.to_string(),
            msg: to_json_binary(&Cw20ExecuteMsg::Transfer {
- recipient: env.contract.address.to_string(),
+ recipient: info.sender.to_string(),
                amount: offer_amount,
            })?,
            funds: vec![],
```

3.4. Address-string comparison

Target	launchpad/src/contract.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Addresses are compared using string comparisons. In CosmWasm, there are two valid representations of addresses, uppercase and lowercase addresses. The sender may be the correct authorized address; however, the comparison may fail.

```
// Check if the sender is the admin
if info.sender.to_string() != state.admin {
    return Err(StdError::generic_err("Unauthorized: not admin"));
}
```

Impact

The correct authorized sender may not be allowed, and the transaction may revert.

Recommendations

Use the correct canonicalized representation of the addresses.

Remediation

This was remediated in commit 5e6ce5b5 ↗ by comparing the canonicalized version of the addresses.

3.5. Fewer tokens sent than required in `migrate_staking`

Target	dojoswap_staking/src/contract.rs		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `migrate_staking` function distributes tokens to the new staking contract so that the new contract has enough funds. It does this by calculating how many funds have already been distributed, and then, subtracting that from the total amount of funds, the remaining amount is sent to the new contract.

However, it assumes all distribution schedules have had some bond in them and consequently some reward distributed; if a distribution schedule had no bond, then it would not have distributed any rewards, and thus more funds could be retained.

```
pub fn migrate_staking(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    new_staking_contract: String,
) -> StdResult<Response> {
    ...
    let mut distributed_amount = Uint128::zero();
    for s in config.distribution_schedule.iter_mut() {
        if s.1 < block_time {
            // all distributed
            distributed_amount += s.2;
        } else {
            // partially distributed slot
            let whole_time = s.1 - s.0;
            let distribution_amount_per_second:
    Decimal = Decimal::from_ratio(s.2, whole_time);

            let passed_time = block_time - s.0;
            let distributed_amount_on_slot =
                distribution_amount_per_second *
    Uint128::from(passed_time as u128);
            distributed_amount += distributed_amount_on_slot;

            // modify distribution slot
```

```
            s.1 = block_time;
            s.2 = distributed_amount_on_slot;
        }
    }
    ...
    let remaining_anc =
    total_distribution_amount.checked_sub(distributed_amount)?;
    ...
}
```

Impact

Fewer funds than expected would be sent to the new staking contract, resulting in a nonzero balance of `dojo_tokens` in the old staking contract.

Recommendations

Ensure there is always some bond when deploying a schedule, or account for the cumulative sum of undistributed rewards when `total_bond_amount` is zero in `compute_reward`.

Remediation

This issue has been acknowledged by Dojoswap Labs, PTE.

Dojoswap Labs, PTE provided the following response:

> Noted on this. Currently, all staking contracts do indeed have bonds resulting in distribution.

3.6. Possible overflow in calculations

Target	dojoswap_staking/contract.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

There is a possible overflow in this calculation:

```
let pending_reward = (staker_info.bond_amount *
    state.global_reward_index).checked_sub(staker_info.bond_amount *
    staker_info.reward_index)?;
```

Impact

Calculations could be inaccurate and result in a loss of rewards.

Recommendations

Adjust the calculation to the equivalent one below:

```
let pending_reward = staker_info.bond_amount *
    state.global_reward_index.checked_sub(staker_info.reward_index)?;
```

Remediation

This issue has been acknowledged by Dojoswap Labs, PTE, and a fix was implemented in commit ce55f60d ↗.

3.7. Commission-amount attribute is incorrect

Target	dojoswap_pair/contract.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Fees are deducted from the `commission_asset`; however, when it is added as an attribute, the `fees` are not deducted.

```
// CONTRACT - a user must do token approval
#[allow(clippy::too_many_arguments)]
pub fn swap(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    sender: Addr,
    offer_asset: Asset,
    belief_price: Option<Decimal>,
    max_spread: Option<Decimal>,
    to: Option<Addr>,
    deadline: Option<u64>,
) -> Result<Response, ContractError> {
    ...
    let fees = commission_amount
        .checked_div(Uint128::from(2u128))
        .ok()
        .unwrap();
    // Sends half of commissions as fees to fee collector
    if !fees.is_zero() {
        let commission_asset = Asset {
            info: ask_pool.info.clone(),
            amount: fees,
        };

    messages.push(commission_asset.into_msg(Addr::unchecked(FEE_COLLECTOR))?);
    }
    ...
    Ok(Response::new().add_messages(messages).add_attributes(vec![
    ("commission_amount", &commission_amount.to_string()),
}
```

Impact

The attributes of this transaction will be inaccurate.

Recommendations

Deduct the fees from the `commission_asset` before adding it as an attribute.

Remediation

This issue has been acknowledged by Dojoswap Labs, PTE.

Dojoswap Labs, PTE provided the following response:

> Acknowledged. Attribute is currently not used anywhere else other than as logs.

3.8. Launchpad's migrate sets the contract to the current version and not the target version

Target	launchpad/src/contract.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The function `migrate_version` is responsible for migrating the contract; however, it treats the `target_contract_version` as the version in the past (that the migration is away from) and the `version` as the version to migrate to.

```
pub fn migrate_version(
    deps: DepsMut,
    target_contract_version: &str,
    name: &str,
    version: &str,
) -> StdResult<()> {
    ...
    if prev_version.version != target_contract_version {
        return Err(StdError::generic_err(format!(
            "invalid contract version. target {}, but source is {}",
            target_contract_version, prev_version.version
        )));
    }
    set_contract_version(deps.storage, name, version)?;

    Ok(())
}
```

```
const CONTRACT_NAME: &str = "crates.io:launchpad";
const CONTRACT_VERSION: &str = env!("CARGO_PKG_VERSION");
const TARGET_CONTRACT_VERSION: &str = "0.1.2";

pub fn migrate(deps: DepsMut, _env: Env, _msg: MigrateMsg) ->
    Result<Response, ContractError> {
    migrate_version(
    deps,
    TARGET_CONTRACT_VERSION,
```

```
    CONTRACT_NAME,
    CONTRACT_VERSION,
    )?;

    Ok(Response::default())
}
```

Impact

The misnaming of source and target may lead to mistakes when updating the contract for deployment, costing gas for failed migrations.

Recommendations

Rename `TARGET_CONTRACT_VERSION` to `EXPECTED_PREVIOUS_CONTRACT_VERSION` in launchpad, and rename `target_contract_version` to `previous_contract_version` and `version` to `updated_contract_version` in `migrate_version`.

Remediation

This issue has been acknowledged by Dojoswap Labs, PTE, and a fix was implemented in commit `ce55f60d` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Composability

A user who bonds using CW20 `send_from` to send a message with the allowance of another account/contract will always be the sender in the call. As a result, it may be difficult to build on top of DojoSwap and take advantage of the bonding functionality if it has to be done through other contracts.

Here is a practical edge case: a fund-type contract that gives allowance to members and expects to safekeep all intermediary tokens will not receive the bond tokens if the operator is using `send_from`.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the crates 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.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Crate: dojoswap_staking

The dojoswap_staking contract is responsible for handling requests to bond and unbond stake and for computing and distributing staking rewards.

Its state includes

- A governance address, which is capable of updating the reward-distribution schedule, updating the governance address, and migrating the staking contract to a new address/version
- The CW20 token that rewards are denominated in, `dojo_token`
- The CW20 token that can be staked, `staking_token`
- The distribution schedule, a vector of `(u64, u64, Uint128)`s (interpreted as start time, end time, and tokens per second)
- The most recent block time that rewards were computed for, `last_distributed`
- The current total amount of bonded stake, `total_bond_amount`
- The cumulative sum of the ratio of rewarded tokens per bonded stake, `global_reward_index`, used for computing individual stakers' rewards on demand

as well as the following, per staking address:

- A snapshot of `global_reward_index` as of the last time rewards were computed for this particular staker, `reward_index`
- The amount of bonded `staking_token`s for this staker, `bond_amount`
- The amount of `dojo_token`s that have been accumulated for this staker, but not yet withdrawn, `pending_reward`

Function: `instantiate`

The `instantiate` function configures the initial state of the contract. The sender of the instantiate message is stored as the governance address. The `dojo_token`, `staking_token`, and initial `distribution_schedule` are set from the contents of the instantiate message. Additionally, `last_distributed` is set to the current block time, and `total_bond_amount` and `global_reward_index` are initialized to zero.

Function: `receive_cw20`

The `receive_cw20` function is called in response to being sent CW20 tokens. It rejects tokens of any type other than `staking_token`, and it accepts `staking_token`s, passing the sender and amount to `bond`.

Function: `bond`

The `bond` function increases a staker's bonded tokens, initializing the tracking thereof if the staker previously had no stake.

Prior to increasing the staker's bond amount, it updates the global state with `compute_reward` and the per-staker state with `compute_staker_reward` so that the rewards up to the current block use the previous bonding amount. It increases the staker's bond amount with `increase_bond_amount`, then stores the updated per-staker and global state in storage.

Function: `compute_reward`

The `compute_reward` function updates `last_distributed` to the current block time.

If there is a nonzero amount of total bonded stake, it additionally iterates through the schedule entries to calculate a cumulative sum of the rewards accumulated since the last block in which `compute_reward` was called and adds it (divided by the total amount of stake) to `global_reward_index`.

Function: `compute_staker_reward`

The `compute_staker_reward` function updates the reward allocated to an individual staker. The difference between the current `global_reward_index` and the staker's `reward_index`, each multiplied by the staker's amount of bonded stake, is added to the staker's `pending_reward`, and its `reward_index` is updated to the current `global_reward_index`. This calculation can be made more overflow resistant; see Finding 3.6. ↗.

Function: `unbond`

The `unbond` function decreases a staker's bonded tokens, sending them to the staker.

It enforces that a staker cannot withdraw more than the amount of tokens they have staked.

Prior to decreasing the staker's bond amount, it updates the global and per-staker state via `compute_reward` and `compute_staker_reward` so that the rewards up to the current block use the previous bond amount; it then decreases the staker's bond amount with `decrease_bond_amount`.

If the staker has no pending reward to withdraw and decreased their bond amount all the way to zero, it removes the staking entry for the staker entirely (which saves storage space); otherwise, it stores the modified state for the staker.

It then updates the global state and sends the unbonded staking tokens to the staker.

Function: `withdraw`

The `withdraw` function allows a staker to withdraw their pending rewards.

It updates the global and per-staker state via `compute_reward` and `compute_staker_reward`, reads the updated `pending_reward`, then zeroes it in the to-be-updated state.

If the staker has zero bond amount (i.e., they unbonded previously, potentially in the same block), it removes their staking entry from the state; otherwise, it stores their modified entry in the state.

It then updates the global state and sends the withdrawn Dojo tokens to the staker.

Function: `update_config`

The `update_config` function allows the governance address to append entries to the distribution schedule.

It validates that the message sender is the governance address. It validates the schedule by calling the `assert_new_schedules` function; it then copies the validated schedule and the previous configuration's `dojo_token` and `staking_token` to a configuration object that it writes to the state.

Function: `assert_new_schedules`

The `assert_new_schedules` function validates that a proposed distribution schedule does not remove any schedule entries that have started nor adds schedule entries that would have retroactively started.

Function: `update_gov`

The `update_gov` function allows the governance address to set a replacement governance address.

It validates that the message sender is the current governance address. It does not check that the replacement governance address is a valid address, with similar impact to Finding 3.4. ↗.

Function: `migrate_staking`

The `migrate_staking` function allows the governance address to migrate the staking contract to a new, potentially updated instance. It is intended to send the reward tokens that have not yet been distributed to the new contract and leave the staking tokens and pending rewards with the current contract to be withdrawn by stakers.

It validates that the message sender is the governance address. It updates the global reward state via `compute_reward` (but does not update the per-staker rewards; those will be updated on subse-

quent calls to `unbond`/`withdraw`).

It clears all future (not yet started) schedule entries from the state, then iterates all past and current schedule entries to account for how many rewards have been distributed, but this overestimates schedule entries during which there was zero total bonded stake; see Finding 3.5. ↗.

It writes the updated schedule and global rewards to the state, then sends the reward tokens that have not yet been distributed to the new contract.

5.2. Crate: `launchpad`

The launchpad contract implements a two-phase batch sale of tokens.

Function: `instantiate`

The `instantiate` function configures the initial state of the contract.

The instantiate message specifies the following:

- An `admin` address that can call `update_config` (the `admin` address has no well-formedness checks)
- A native token `raising_denom`, in which purchases are denominated
- A CW20 token `offering_token` that the contract sells
- A `start_time` and `end_time` in seconds that the sale will take place during
- An amount `raising_amount` that is the expected amount of `raising_denom` that must be met
- An amount `offering_amount` of total `offering_tokens` to sell

The `instantiate` function verifies that `end_time` is after `start_time`, initializes the state from the above fields, additionally sets `total_amount` to zero and `allow_claim` to false, and persists the state.

Function: `deposit`

The `deposit` function processes deposits from purchasers of tokens.

It checks if all of the following hold:

- The current block time is within the start and end time.
- The deposit amount specified as part of the message is nonzero.
- Exactly one coin type was transferred in by the message, and its denom matches the `raising_denom`.

It does not check that the deposit amount equals the amount in the message's funds; see Finding 3.1. ↗.

If all of the above hold, the per-user `amount` and `total_amount` are increased by the deposit amount

and the state is updated.

Function: `harvest`

The `harvest` function allows purchasers of tokens to obtain their share of the offering tokens.

It checks if all of the following hold:

- The current block time is strictly after the end time.
- The user deposited a nonzero amount.
- The user has not already called `harvest` (tracked by the `claimed` field).
- The state's `allow_claim` field must be true.

If all of the above hold, the amounts of the `offering_token` and refund of the `raising_denom` are calculated.

If the `raising_amount` was met,

- The user gets `(amount / total_amount) * offering_amount` of the `offering_token`.
- The user gets `amount - (raising_amount * amount / total_amount)` of the `raising_denom` as a refund.

If the `raising_amount` was not met,

- The user gets `(amount / total_amount) * (offering_amount / raising_amount)` of the `offering_token`.
- The user gets no refund of the `raising_denom`.

It transfers the relevant amounts of the `offering_token` and `raising_denom` to the user and sets their `claimed` flag in the state.

Function: `update_config`

The `update_config` function allows the `admin` address to update any of the following: `raising_denom`, `offering_token`, `offering_amount`, `start_time`, and `end_time`.

It enforces that the message sender is the `admin` address (but see Finding 3.4. ↗), that `start_time` is not after `end_time`, and that no user has called `deposit` yet (via checking `total_amount`).

Function: `final_withdraw`

The `final_withdraw` function allows the `admin` address to withdraw any remaining `offering_token`s as well as the `raising_denom`s.

It enforces that the message sender is the `admin` address (but see Finding 3.4. ↗) and that the specified withdrawal amounts are less than or equal to the contract's balance of the relevant tokens; if the amounts are nonzero, it sends the `raised_denom` value to the message sender and the offering

tokens to the launchpad contract itself. See Finding 3.3. ↗.

Function: `flip_allow_claim`

The `flip_allow_claim` function allows the `admin` address to negate the global `allow_claim` field. No user is able to `harvest` until this is called.

It enforces that the message sender is the `admin` address (but see Finding 3.4. ↗) and then persists the negation of `allow_claim` to the state.

Functions: `migrate` / `migrate_version`

The `migrate` function calls `migrate_version` with constants `TARGET_CONTRACT_VERSION`, `CONTRACT_NAME`, and `CONTRACT_VERSION`.

The `migrate_version` function parses the current contract's name and version, and it requires that the current name matches the provided name and that the current version matches the provided target version. If both match, it sets the contract's version to the provided version. See Finding 3.8. ↗.

5.3. Crate: dojoswap_pair

The dojoswap_pair contract allows swaps and liquidity provisioning for a pair of tokens. It is essentially unmodified from `terraswap_pair`, except for the following additions:

- When swapping, half the commission is sent as a fee to the `FEE_COLLECTOR` address.
- An `admin_configure` option, only callable by the `FEE_COLLECTOR` address, permits modifying the assets that a particular pair instance manages, as well as where the decimal point for those assets is.

Function: `admin_configure`

The `admin_configure` function allows the `FEE_COLLECTOR` to modify which assets the pair contract swaps.

It enforces that the sender is the `FEE_COLLECTOR` address. It then loads the contract's `PairInfoRaw`, copies its `liquidity_token`, recomputes the `contract_addr` to be the same as in `initialize`, and overwrites `asset_infos` and `asset_decimals` with the provided values.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Injective Mainnet.

During our assessment on the scoped DojoSwap crates, we discovered eight findings. One critical issue was found. One was of medium impact, three were of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

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 code added to the project after the version reviewed during 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 samples 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.