Zellic

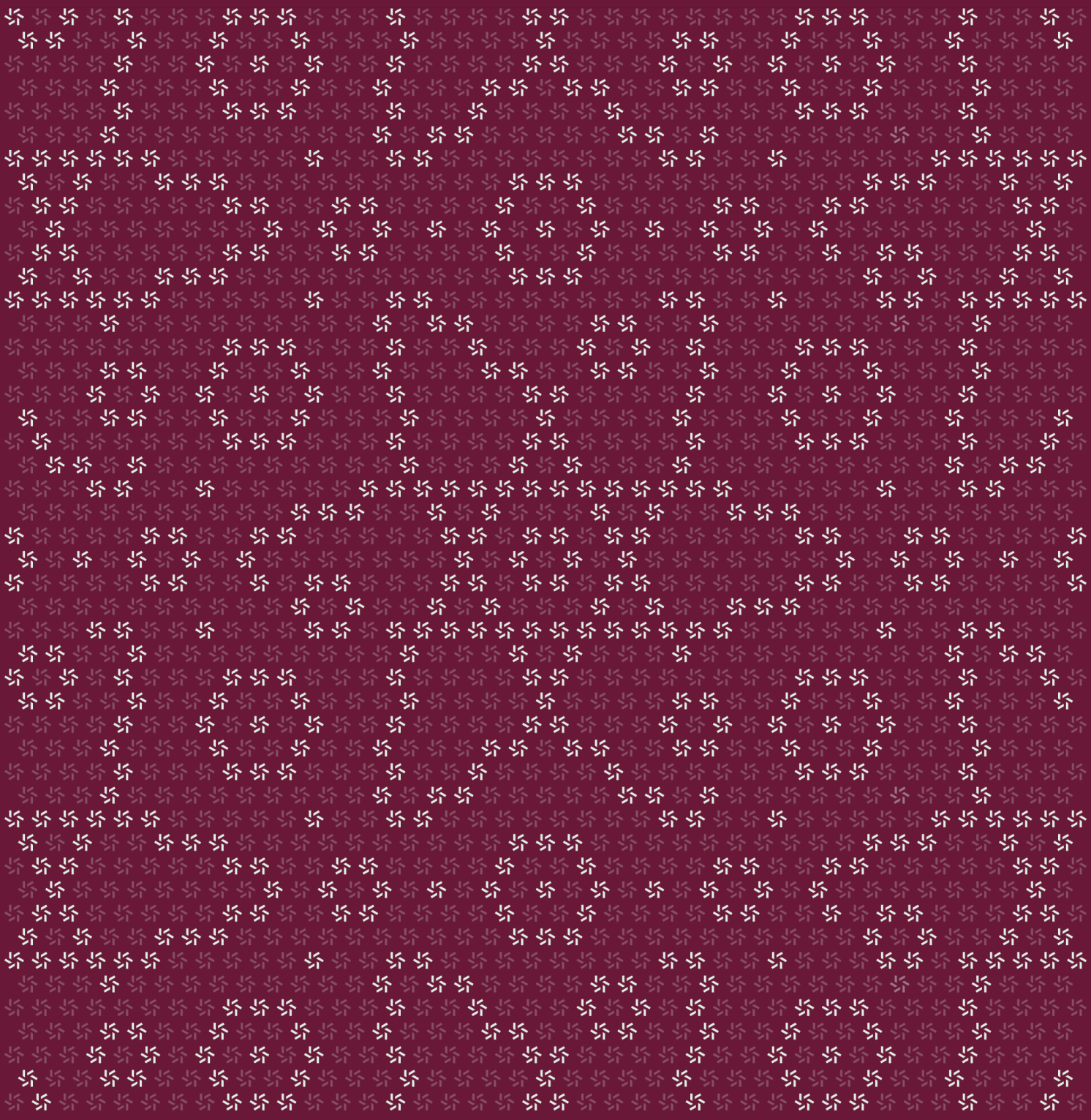
Prepared for
Jeremy Bokobza
TruFin

Prepared by
Frank Bachman
Nipun Gupta
Zellic

November 19, 2024

TruFin Injective Staker

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 TruFin from November 6th to November 15th, 2024. During this engagement, Zellic reviewed TruFin Injective Staker'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:

- Is it possible for a user to exploit the system to claim more INJ than they are legitimately entitled to?
- Could a malicious user cause protocol insolvency?
- Are there ways for someone to manipulate the exchange rate for personal profit?
- Could user stakes be blocked or made inaccessible in any way?

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

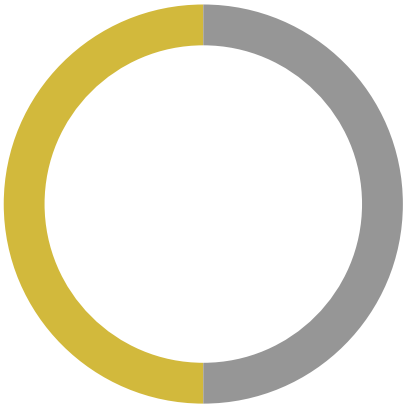
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped TruFin Injective Staker contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature.

Breakdown of Finding Impacts

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



2. Introduction

2.1. About TruFin Injective Staker

TruFin contributed the following description of TruFin Injective Staker:

> TruFin provides access to INJ staking for institutions featuring auto-compounding of rewards and the ability to allocate rewards to other addresses. The protocol is permissioned, ensuring that each user has passed know-your-customer (KYC) checks and that only whitelisted users can stake.

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 contracts.

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.

2.3. Scope

The engagement involved a review of the following targets:

TruFin Injective Staker Contracts

Type	Rust
Platform	Cosmos
Target	smart-contracts-inj-zellic
Repository	https://github.com/TruFin-io/smart-contracts-inj-zellic.git ↗
Version	`c3e4c242947a32f26fec95dbb220b0475f9c1f15`

Programs	`constants.rs`
	`contract.rs`
	`error.rs`
	`lib.rs`
	`msg.rs`
	`state.rs`
	`whitelist.rs`

2.4. Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Frank Bachman
Engineer
frank@zellic.io

Nipun Gupta
Engineer
nipun@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 6, 2024	Kick-off call
November 6, 2024	Start of primary review period
November 15, 2024	End of primary review period

3. Detailed Findings

3.1. Unstake could be blocked for certain users

Target	contract.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

When a user unstakes some amount of tokens, the tokens are unstaked from either the default validator or the validator provided by the user. If the validator does not have enough tokens available to unstake, the remaining tokens are transferred out of the `CONTRACT_REWARDS`. In certain cases, it might be possible that the entire unstake amount comes out of the contract rewards.

For example, consider a case where there is one validator and 10 users delegate 100 tokens each and after some time the rewards plus stakes become 120 for each user. For the last user to unstake, the entire amount should come from the rewards as the contract first removes from the validators and then from the rewards.

However, in that case, the `actual_amount_to_unstake` would be zero and the undelegate message will fail here: https://github.com/InjectiveLabs/cosmos-sdk/blob/master/x/staking/keeper/msg_server.go#L408.

```
if !msg.Amount.IsValid() || !msg.Amount.Amount.IsPositive() {
    return nil, errorsmod.Wrap(
        sdkerrors.ErrInvalidRequest,
        "invalid shares amount",
    )
}
```

Impact

Certain users might not be able to unstake.

Recommendations

We recommend only adding the unstake message if `actual_amount_to_unstake` is greater than zero.

Remediation

This issue has been acknowledged by TruFin, and a fix was implemented in commit 78e5d925 ↗.

3.2. Erroneous mint-fee message

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

Description

In the `internal_stake` and `internal_unstake` functions, the contract adds the `mint_treasury_fees` message even when there are no rewards.

```
let validator_total_rewards = deps
    .querier
    .query_delegation(staker_address, validator_addr.clone())?
    .and_then(|d| {
        d.accumulated_rewards
            .iter()
            .find(|coin| coin.denom == INJ)
            .cloned()
    })
    .map(|reward| reward.amount.u128())
    .unwrap_or(0);

CONTRACT_REWARDS.save(deps.storage, &validator_total_rewards.into())?;

// mint fees to the treasury for the liquid rewards on the validator
let treasury_shares_minted = mint_treasury_fees(
    &mut deps,
    &env,
    validator_total_rewards,
    fee,
    staker_info.treasury.clone(),
    share_price_num,
    share_price_denom,
)?;
```

Impact

If there are no rewards or the fee amount is zero, the contract does not need to add a message to mint the fee.

Recommendations

Only add the mint message for the fee, if the computed fee is nonzero.

Remediation

This issue has been acknowledged by TruFin, and a fix was implemented in commit `ef8518f8` ↗.

4. Threat Model

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

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.

4.1. Module: Injective-staker/contract.rs

Message: `ExecuteMsg::AddAgent`

This allows an existing agent to add a new agent to the whitelist.

Inputs

- `info.sender`
  - **Validation**: The `add_agent` function verifies that the `info.sender` is a valid agent.
  - **Impact**: Ensures that only authorized agents can modify the whitelist.
- `new_agent`
  - **Validation**: The address is validated to ensure proper formatting, and it must not already exist in the whitelist. Additionally, the new agent cannot be the contract owner.
  - **Impact**: This is the address of the new agent to be added.

Branches and code coverage (including function calls)

Intended branches

- If `new_agent` is valid and not already an agent, it is added to the whitelist.
  - ☑ Test coverage

Negative behavior

- Fail if the `info.sender` is not an existing agent.
  - ☑ Negative test
- Fail if `new_agent` is the contract owner.
  - ☑ Negative test
- Fail if `new_agent` is already in the whitelist.
  - ☑ Negative test

Message: `ExecuteMsg::AddUserToBlacklist`

This allows an agent to add a user to the blacklist.

Inputs

- `info.sender`
  - **Validation**: The `add_user_to_blacklist` function checks if the `info.sender` is an authorized agent.
  - **Impact**: N/A.
- `user`
  - **Validation**: The `user` address is validated to ensure it is a valid address format.
  - **Impact**: If the user is not already blacklisted, they are added to the blacklist.

Branches and code coverage (including function calls)

Intended branches

- If the `user` address is valid and not already blacklisted, it is added to the blacklist.
  - ☑ Test coverage

Negative behavior

- Fail if the `info.sender` is not an authorized agent.
  - ☑ Negative test
- Fail if the `user` address is invalid.
  - ☑ Negative test
- Fail if the `user` is already blacklisted.
  - ☑ Negative test

Message: `ExecuteMsg::AddUserToWhitelist`

This adds a user to the whitelist.

Inputs

- `info.sender`
  - **Validation**: The `add_user_to_whitelist` function verifies that the `info.sender` is an agent.
  - **Impact**: Ensures only authorized agents can add users to the whitelist.
- `user`
  - **Validation**: The `user` address is validated using `deps.api.addr_validate`. Additionally, it ensures the user is not already whitelisted by checking their current status.

- **Impact**: Specifies the user to be added to the whitelist, changing their status.

Branches and code coverage (including function calls)

Intended branches

- If the `user` address is valid and not already whitelisted, they are added to the whitelist.
  - ☑ Test coverage

Negative behavior

- Fail if the `info.sender` is not an agent.
  - ☑ Negative test
- Fail if the `user` address is invalid.
  - ☑ Negative test
- Fail if the `user` is already whitelisted.
  - ☑ Negative test

Message: `ExecuteMsg::AddValidator`

This allows the admin to add a new validator that can be staked to.

Inputs

- `info.sender`
  - **Validation**: The `add_validator` function verifies that the `info.sender` is the contract owner.
  - **Impact**: N/A.
- `validator_addr`
  - **Validation**: The address is verified to ensure it exists in the current validator set and is not already in the contract's validator list.
  - **Impact**: This is the address of the validator to be added.

Branches and code coverage (including function calls)

Intended branches

- If `validator_addr` is valid, it is added to the list of enabled validators in the contract state.
  - ☑ Test coverage

Negative behavior

- Fail if `validator_addr` already exists in the validator list.
  - ☑ Negative test
- Fail if `validator_addr` is not part of the current validator set.

- ☑ Negative test

Message: `ExecuteMsg::Allocate`

This allocates INJ staking rewards from the sender to a specified recipient.

Inputs

- `info.sender`
  - **Validation**: Verified to ensure the sender is whitelisted. Confirmed that the contract is not paused.
  - **Impact**: Acts as the allocator of the funds.
- `recipient`
  - **Validation**: Ensures the recipient address is valid. Cannot be the same as `info.sender`.
  - **Impact**: Receives the allocated amount.
- `amount`
  - **Validation**: Must be at least `ONE_INJ`.
  - **Impact**: The amount of INJ allocated to the recipient.

Branches and code coverage (including function calls)

Intended branches

- Create a new allocation if none exists.
  - ☑ Test coverage
- Update an existing allocation with the new amount and share price.
  - ☑ Test coverage

Negative behavior

- Fail if the contract is paused.
  - ☑ Negative test
- Fail if the sender is not whitelisted.
  - ☑ Negative test
- Fail if the recipient is the same as the sender.
  - ☑ Negative test
- Fail if the amount is less than `ONE_INJ`.
  - ☑ Negative test
- Fail if the recipient address is invalid.
  - ☑ Negative test

Message: `ExecuteMsg::Claim`

This allows a user to withdraw all their expired claims.

Inputs

- `info.sender`
  - **Validation**: The claim function verifies that the `info.sender` is a whitelisted address.
  - **Impact**: The user that withdraws their claims.

Branches and code coverage (including function calls)

Intended branches

- If the amount of claimed assets is greater than zero and the contract has the rewards available, then it transfers the assets to the user.
  - ☑ Test coverage

Negative behavior

- Revert if there is nothing to claim.
  - ☑ Negative test
- Revert if the contract does not have a sufficient amount to transfer to the user.
  - ☑ Negative test

Message: `ExecuteMsg::ClaimOwnership`

This allows the pending owner to claim ownership of the contract, transferring privileges from the current owner.

Inputs

- `info.sender`
  - **Validation**: The `claim_ownership` function ensures that the `info.sender` matches the pending owner set in the contract state.
  - **Impact**: The sender becomes the new owner upon successful execution.

Branches and code coverage (including function calls)

Intended branches

- If the `info.sender` matches the pending owner, ownership is transferred and the pending owner is cleared.

- ☑ Test coverage

Negative behavior

- Fail if no pending owner is set.
  - ☑ Negative test
- Fail if the `info.sender` does not match the pending owner.
  - ☑ Negative test

Message: `ExecuteMsg::AddUserToWhitelist`

This adds a user to the whitelist.

Inputs

- `info.sender`
  - **Validation**: The `add_user_to_whitelist` function verifies that the `info.sender` is an agent.
  - **Impact**: Ensures only authorized agents can add users to the whitelist.
- `user`
  - **Validation**: The `user` address is validated using `deps.api.addr_validate`. Additionally, it ensures the user is not already whitelisted by checking their current status.
  - **Impact**: Specifies the user to be added to the whitelist, changing their status.

Branches and code coverage (including function calls)

Intended branches

- If the `user` address is valid and not already whitelisted, they are added to the whitelist.
  - ☑ Test coverage

Negative behavior

- Fail if the `info.sender` is not an agent.
  - ☑ Negative test
- Fail if the `user` address is invalid.
  - ☑ Negative test
- Fail if the `user` is already whitelisted.
  - ☑ Negative test

Message: `ExecuteMsg::CompoundRewards`

This restakes rewards from all validators and sweeps contract rewards back into the default validator while handling treasury fees.

Inputs

- No explicit user inputs — the function operates based on the contract's internal state and validators.

Branches and code coverage (including function calls)

Intended branches

- Successfully restake rewards and sweep contract rewards when rewards exist.
  - ☑ Test coverage
- Calculate fees based on `staker_info.fee` and mint treasury shares if calculated fees are greater than zero.
  - ☑ Test coverage

Negative behavior

- Return early if no rewards are available (`total_rewards == 0`).
  - ☑ Negative test

Message: `ExecuteMsg::Deallocate`

This deallocates INJ staking rewards from a specified recipient.

Inputs

- `info.sender`
  - **Validation**: Verified to ensure the sender is whitelisted. Confirms that the contract is not paused.
  - **Impact**: Acts as the deallocator of the funds.
- `recipient`
  - **Validation**: Ensures the recipient address is valid. Checks that the recipient has an existing allocation from the sender.
  - **Impact**: Has their allocation reduced or removed.
- `amount`
  - **Validation**: Must not exceed the current allocation to the recipient. Remaining allocation must either be zero or at least `ONE_INJ`.
  - **Impact**: The amount of INJ removed from the allocation.

Branches and code coverage (including function calls)

Intended branches

- Deallocate the `amount` from the recipient.
  - ☑ Test coverage
- Remove the allocation entirely if the remaining amount is zero.
  - ☑ Test coverage
- Update the allocation with the reduced amount while preserving the share price.
  - ☑ Test coverage

Negative behavior

- Fail if the contract is paused.
  - ☑ Negative test
- Fail if the sender is not whitelisted.
  - ☑ Negative test
- Fail if the recipient address is invalid.
  - ☑ Negative test
- Fail if no allocation exists between the sender and recipient.
  - ☑ Negative test
- Fail if the `amount` exceeds the current allocation.
  - ☑ Negative test
- Fail if the remaining allocation is nonzero but less than `ONE_INJ`.
  - ☑ Negative test

Message: `ExecuteMsg::DisableValidator`

This allows the admin to disable a previously enabled validator. Disabled validators cannot accept new stakes, but existing stakes can still be unstaked and withdrawn as usual.

Inputs

- `info.sender`
  - **Validation**: The `disable_validator` function verifies that the `info.sender` is the contract owner.
  - **Impact**: N/A.
- `validator_addr`
  - **Validation**: Ensures the validator exists in the contract's state and is currently enabled.
  - **Impact**: This validator address gets disabled.

Branches and code coverage (including function calls)

Intended branches

- If `validator_addr` exists and is enabled, it is marked as disabled in the contract state.
  - ☑ Test coverage

Negative behavior

- Fail if `validator_addr` does not exist in the validator list.
  - ☑ Negative test
- Fail if `validator_addr` is already disabled.
  - ☑ Negative test

Message: `ExecuteMsg::DistributeRewards`

This distributes staking rewards from the sender to a specified recipient based on the current allocation and share price.

Inputs

- `info.sender`
  - **Validation**: Verified to ensure the sender is whitelisted. Confirmed to have at least one allocation.
  - **Impact**: Acts as the distributor of rewards.
- `recipient`
  - **Validation**: Ensures the recipient address is valid. Confirms the sender has an allocation to the specified recipient.
  - **Impact**: Receives distributed rewards.
- `in_inj`
  - **Validation**: N/A.
  - **Impact**: Determines if the rewards should be distributed in INJ.
- `info.funds`
  - **Validation**: If provided, ensures it is handled correctly, either as additional rewards or refunded if not required.
  - **Impact**: May be redistributed or refunded to the sender.

Branches and code coverage (including function calls)

Intended branches

- Distribute rewards to the recipient based on the allocation and share price.
  - ☑ Test coverage
- Handle cases where no rewards are available for distribution.
  - ☑ Test coverage

- Refund unused INJ back to the sender.
  - ☑ Test coverage

Negative behavior

- Fail if the contract is paused.
  - ☑ Negative test
- Fail if the sender is not whitelisted.
  - ☑ Negative test
- Fail if the recipient address is invalid.
  - ☑ Negative test
- Fail if no allocations exist for the sender.
  - ☑ Negative test
- Fail if there is no allocation to the specified recipient.
  - ☑ Negative test

Message: `ExecuteMsg::DistributeAll`

This distributes staking rewards from the sender to all recipients based on the sender's allocations. Rewards can be distributed in INJ or TruINJ.

Inputs

- `info.sender`
  - **Validation**: Verified to ensure the sender is whitelisted. Confirmed to have at least one allocation.
  - **Impact**: Acts as the distributor of rewards.
- `in_inj`
  - **Validation**: N/A.
  - **Impact**: Determines whether rewards are distributed in INJ or TruINJ.
- `info.funds`
  - **Validation**: If provided, ensures it is handled correctly, either as additional rewards or refunded if not required.
  - **Impact**: May be redistributed or refunded to the sender.

Branches and code coverage (including function calls)

Intended branches

- Distribute rewards for all allocations in either INJ or TruINJ.
  - ☑ Test coverage
- Refund remaining INJ back to the sender after processing all allocations.
  - ☑ Test coverage

Negative behavior

- Fail if the contract is paused.
  - ☑ Negative test
- Fail if the sender is not whitelisted.
  - ☑ Negative test
- Fail if the sender has no allocations.
  - ☑ Negative test
- Handle cases where no rewards are available for distribution, refunding any attached INJ to the sender.
  - ☑ Negative test

Message: `ExecuteMsg::SetDistributionFee`

This allows the admin to set the treasury fee charged on rewards distribution.

Inputs

- `info.sender`
  - **Validation**: The `set_distribution_fee` function verifies that the `info.sender` is the owner.
  - **Impact**: N/A.
- `new_distribution_fee`
  - **Validation**: The fee is verified to be lower than 100%.
  - **Impact**: The new fee is set to be this value.

Branches and code coverage (including function calls)

Intended branches

- If the fee is lower than 100%, it is updated in the state.
  - ☑ Test coverage

Negative behavior

- Fail if the fee is larger than 100%.
  - ☑ Negative test

Message: `ExecuteMsg::EnableValidator`

This allows the admin to re-enable a previously disabled validator, permitting new stakes to the validator.

Inputs

- `info.sender`
  - **Validation**: The `enable_validator` function verifies that the `info.sender` is the contract owner.
  - **Impact**: N/A.
- `validator_addr`
  - **Validation**: Ensures the validator exists in the contract's state and is currently disabled.
  - **Impact**: This is the validator address that is enabled.

Branches and code coverage (including function calls)

Intended branches

- If `validator_addr` exists and is disabled, it is marked as enabled in the contract state.
  - ☑ Test coverage

Negative behavior

- Fail if `validator_addr` does not exist in the validator list.
  - ☑ Negative test
- Fail if `validator_addr` is already enabled.
  - ☑ Negative test

Message: `ExecuteMsg::Pause`

This allows the admin to pause the contract, preventing user operations until it is resumed.

Inputs

- `info.sender`
  - **Validation**: The `pause` function verifies that the `info.sender` is the contract owner.
  - **Impact**: N/A.

Branches and code coverage (including function calls)

Intended branches

- If the contract is not already paused, it is marked as paused in the contract state.
  - ☑ Test coverage

Negative behavior

- Fail if the contract is already paused.
  - ☑ Negative test

Message: `ExecuteMsg::SetPendingOwner`

This allows the current owner to set a pending owner. The pending owner has no privileges until explicitly accepted.

Inputs

- `info.sender`
  - **Validation**: The `set_pending_owner` function verifies that the `info.sender` is the current owner.
  - **Impact**: N/A.
- `new_owner`
  - **Validation**: The provided `new_owner` address is validated to ensure it is in a proper address format.
  - **Impact**: The value that `new_owner` is set to.

Branches and code coverage (including function calls)

Intended branches

- If `new_owner` is valid, it is saved as the pending owner in the state.
  - ☑ Test coverage

Negative behavior

- Fail if `new_owner` is not a valid address.
  - ☑ Negative test

Message: `ExecuteMsg::RemoveAgent`

This removes an agent from the whitelist.

Inputs

- `info.sender`
  - **Validation**: The `remove_agent` function verifies that the `info.sender` is an existing agent.
  - **Impact**: Ensures only authorized agents can initiate the removal of other agents from the whitelist.
- `agent_to_remove`

- **Validation**: The `agent_to_remove` address is validated using `deps.api.addr_validate`. Additionally, the address must not belong to the owner and must already exist in the whitelist.
- **Impact**: Specifies the agent to be removed, affecting the whitelist state.

Branches and code coverage (including function calls)

Intended branches

- If the `agent_to_remove` address is valid, not the owner, and exists in the whitelist, it is removed.
  - ☑ Test coverage

Negative behavior

- Fail if the `info.sender` is not an agent.
  - ☑ Negative test
- Fail if the `agent_to_remove` address is invalid.
  - ☑ Negative test
- Fail if the `agent_to_remove` address belongs to the owner.
  - ☑ Negative test
- Fail if the `agent_to_remove` address is not in the whitelist.
  - ☑ Negative test

Message: `ExecuteMsg::SetDefaultValidator`

This allows the admin to set a given validator as the new default validator.

Inputs

- `info.sender`
  - **Validation**: The `set_default_validator` function verifies that the `info.sender` is the owner.
  - **Impact**: N/A.
- `new_default_validator_addr`
  - **Validation**: The address is verified to ensure it belongs to a valid validator and that it is enabled.
  - **Impact**: Default validator gets updated to this address.

Branches and code coverage (including function calls)

Intended branches

- If `new_default_validator_addr` is valid, it is updated as the new default validator.

- ☑ Test coverage

Negative behavior

- Fail if `new_default_validator_addr` is not a valid validator address.
  - ☑ Negative test

Message: `ExecuteMsg::SetFee`

This allows the admin to set the treasury fee charged on rewards.

Inputs

- `info.sender`
  - **Validation**: The `set_fee` function verifies that the `info.sender` is the owner.
  - **Impact**: N/A.
- `new_fee`
  - **Validation**: The fee is verified to be lower than 100%.
  - **Impact**: The fee is set to `new_fee`.

Branches and code coverage (including function calls)

Intended branches

- If the fee is lower than 100%, it is updated in the state.
  - ☑ Test coverage

Negative behavior

- Fail if the fee is larger than 100%.
  - ☑ Negative test

Message: `ExecuteMsg::SetMinimumDeposit`

This allows the admin to set the minimum INJ amount a user can deposit.

Inputs

- `info.sender`
  - **Validation**: The `set_min_deposit` function verifies that the `info.sender` is the owner.
  - **Impact**: N/A.
- `new_min_deposit`
  - **Validation**: The amount is verified to be greater than 1 INJ.

- **Impact**: The minimum deposit is set to this value.

Branches and code coverage (including function calls)

Intended branches

- If the amount is larger than 1 INJ, it is updated in the state.
  - ☑ Test coverage

Negative behavior

- Fail if the amount is lower than 1 INJ.
  - ☑ Negative test

Message: `ExecuteMsg::SetTreasury`

This allows the admin to set the treasury address.

Inputs

- `info.sender`
  - **Validation**: The `set_treasury` function verifies that the `info.sender` is the owner.
  - **Impact**: N/A.
- `new_treasury`
  - **Validation**: The provided `new_treasury` address is validated to ensure it is a proper address format.
  - **Impact**: The treasury address is set to this value.

Branches and code coverage (including function calls)

Intended branches

- If the `new_treasury` address is valid, it is updated in the state.
  - ☑ Test coverage

Negative behavior

- Fail if `new_treasury` is not a valid address.
  - ☑ Negative test

Message: `ExecuteMsg::Stake`

This allows whitelisted users to stake their INJ on the default validator.

Inputs

- `info.sender`
  - **Validation**: The stake function verifies that the `info.sender` is a whitelisted address.
  - **Impact**: This is the address that recieves TruINJ.
- `info.funds`
  - **Validation**: The `internal_stake` function verifies that exactly one coin (INJ) was sent.
  - **Impact**: The amount of assets to stake.

Branches and code coverage (including function calls)

Intended branches

- Calculates the exchange rate as per the total INJ staked, calculates the rewards and the contract rewards available, and mints shares to the user based on that exchange rate.
  - ☑ Test coverage
- The rewards from the validator are increased in the `CONTRACT_REWARDS` storage to be used later.
  - ☑ Test coverage
- The treasury is minted some fee (TruINJ) as a percentage of the validator rewards.
  - ☑ Test coverage

Negative behavior

- The transaction should revert if the caller is not whitelisted.
  - ☑ Negative test
- The transaction should revert if the contract is paused.
  - ☑ Negative test
- The transaction should revert if the amount of INJ staked is less than the `min_deposit`.
  - ☑ Negative test

Message: `ExecuteMsg::StakeToSpecificValidator`

This allows whitelisted users to stake their INJ on the provided validator.

Inputs

- `info.sender`
  - **Validation**: The stake function verifies that the `info.sender` is a whitelisted address.
  - **Impact**: This is the address that receives TruINJ.
- `info.funds`

  - **Validation**: The `internal_stake` function verifies that exactly one coin (INJ) was sent.
  - **Impact**: The amount of assets to stake.
- `validator_addr`
  - **Validation**: The `internal_stake` function verifies that the validator is whitelisted.
  - **Impact**: The validator on which the user wants to stake.

Branches and code coverage (including function calls)

Intended branches

- Calculates the exchange rate as per the total INJ staked, calculates the rewards and the contract rewards available, and mints shares to the user based on that exchange rate.
  - ☑ Test coverage
- The rewards from the validator are increased in the `CONTRACT_REWARDS` storage to be used later.
  - ☑ Test coverage
- The treasury is minted some fee (TruINJ) as a percentage of the validator rewards.
  - ☑ Test coverage

Negative behavior

- The transaction should revert if the caller is not whitelisted.
  - ☑ Negative test
- The transaction should revert if the contract is paused.
  - ☑ Negative test
- The transaction should revert if the validator is not whitelisted.
  - ☑ Negative test
- The transaction should revert if the amount of INJ staked is less than the `min_deposit`.
  - ☑ Negative test

Message: `ExecuteMsg::Unpause`

This allows the admin to unpause the contract, resuming normal user operations.

Inputs

- `info.sender`
  - **Validation**: The `unpause` function verifies that the `info.sender` is the contract owner.
  - **Impact**: N/A.

Branches and code coverage (including function calls)

Intended branches

- If the contract is paused, it is marked as unpaused in the contract state.
  - ☑ Test coverage

Negative behavior

- Fail if the contract is not currently paused.
  - ☑ Negative test

Message: `ExecuteMsg::Unstake`

This allows whitelisted users to unstake from the default validator.

Inputs

- `info.sender`
  - **Validation**: The stake function verifies that the `info.sender` is a whitelisted address.
  - **Impact**: This is the address that unstakes the TruINJ and could claim the INJ.
- `amount`
  - **Validation**: The `internal_unstake` function verifies that the value is greater than zero and less than the maximum assets that the user could withdraw.
  - **Impact**: The amount of assets to unstake.

Branches and code coverage (including function calls)

Intended branches

- Calculates the exchange rate as per the total INJ staked, calculates the rewards and the contract rewards available, and creates a claim based on that exchange rate.
  - ☑ Test coverage
- If the assets to unstake are greater than the validator's total staked assets, then the excess assets are taken from the `CONTRACT_REWARDS`, which is then updated.
  - ☑ Test coverage
- The treasury is minted some fee (TruINJ) as a percentage of the validator rewards.
  - ☑ Test coverage

Negative behavior

- The transaction should revert if the caller is not whitelisted.
  - ☑ Negative test
- The transaction should revert if the contract is paused.
  - ☑ Negative test

- The transaction should revert if the amount value is greater than the maximum value of assets that the user could withdraw, or it should revert if the shares to burn for that user are zero.
  - ☑ Negative test

Message: `ExecuteMsg::UnstakeFromSpecificValidator`

This allows whitelisted users to unstake from a specified validator.

Inputs

- `info.sender`
  - **Validation**: The stake function verifies that the `info.sender` is a whitelisted address.
  - **Impact**: This is the address that unstakes the TruINJ and could claim the INJ.
- `amount`
  - **Validation**: The `internal_unstake` function verifies that the value is greater than zero and less than the maximum assets that the user could withdraw.
  - **Impact**: The amount of assets to unstake.
- `validator_addr`
  - **Validation**: The `internal_unstake` function verifies that the validator is whitelisted.
  - **Impact**: The validator from which the user wants to unstake.

Branches and code coverage (including function calls)

Intended branches

- Calculates the exchange rate as per the total INJ staked, calculates the rewards and the contract rewards available, and creates a claim based on that exchange rate.
  - ☑ Test coverage
- If the assets to unstake are greater than the validator's total staked assets, then the excess assets are taken from the `CONTRACT_REWARDS`, which is then updated.
  - ☑ Test coverage
- The treasury is minted some fee (TruINJ) as a percentage of the validator rewards.
  - ☑ Test coverage

Negative behavior

- The transaction should revert if the caller is not whitelisted.
  - ☑ Negative test
- The transaction should revert if the contract is paused.
  - ☑ Negative test
- The transaction should revert if the validator is not whitelisted.

- ☑ Negative test

- The transaction should revert if the amount value is greater than the maximum value of assets that the user could withdraw, or it should revert if the shares to burn for that user are zero.
  - ☑ Negative test

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Mainnet.

During our assessment on the scoped TruFin Injective Staker contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature.

5.1. 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.