Zellic

Prepared for

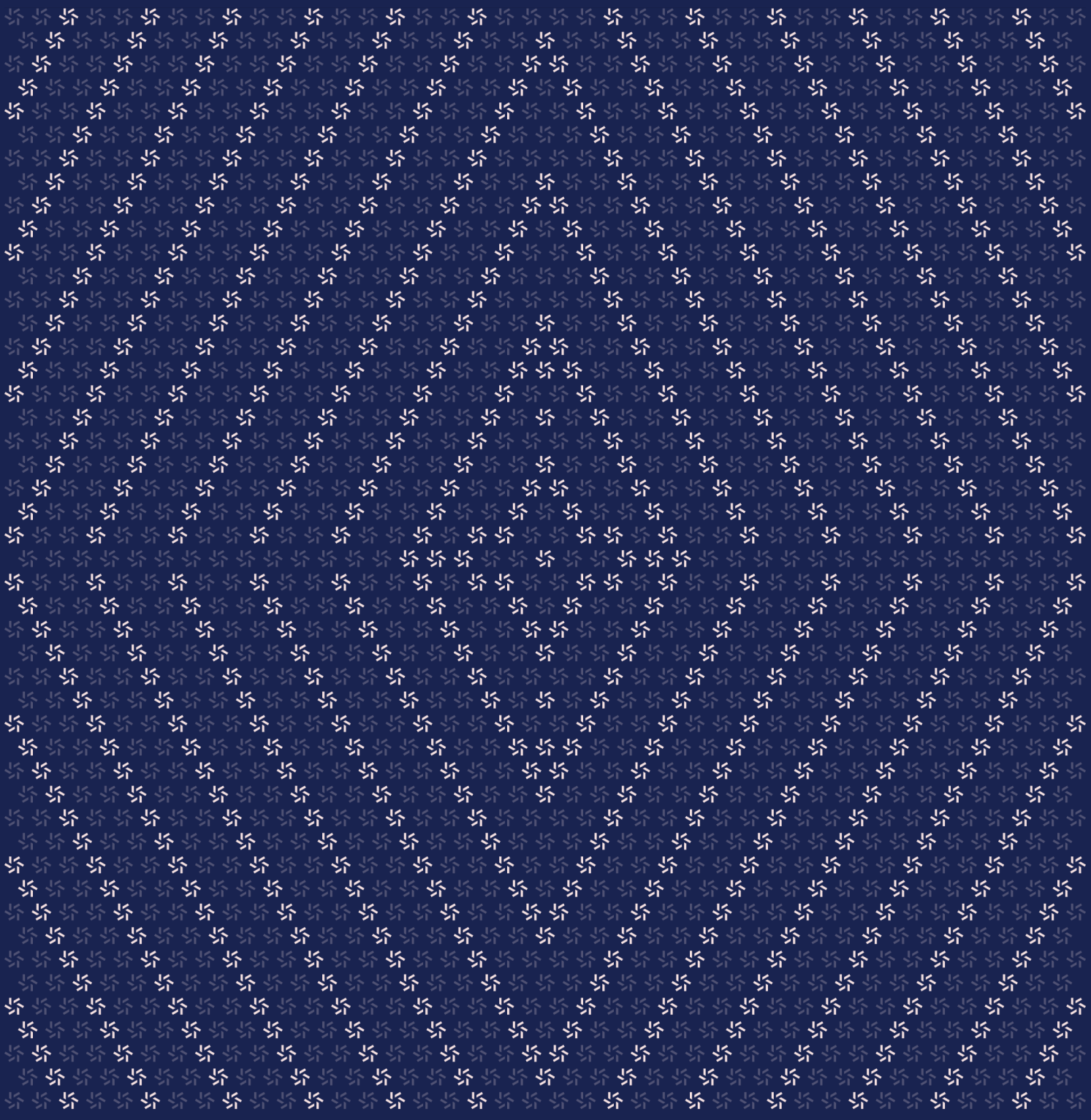
Prosper on behalf of
its DAO and community

Prepared by
Jinheon Lee
Juchang Lee
Sylvain Pelissier
Zellic

January 21, 2025

Prosper Omnichain Fungible Token

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 Prosper from January 14th to January 15th, 2025. During this engagement, Zellic reviewed Prosper's upgraded Omnichain Fungible Token'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 the cross-chain–messaging functionality correctly implemented?
- Are the different roles correctly handled?
- Could an on-chain attacker drain the tokens?
- Could a malicious message trigger a lockup of user funds/tokens?

1.3. Non-goals and Limitations

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

- LayerZero OFT internals
- The TrancheManager contract
- 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.

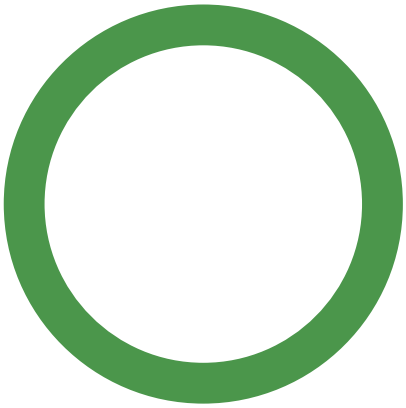
1.4. Results

During our assessment on the scoped Prosper Omnichain Fungible Token contracts, we discovered two findings, both of which were low impact.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	2
Informational	0



2. Introduction

2.1. About Prosper Omnichain Fungible Token

Prosper contributed the following description of Prosper Omnichain Fungible Token:

> Prosper is building the largest RWA protocol for Bitcoin hashrate and treasury.

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.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known 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. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential 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, upgradability weaknesses, centralization risks, and so on.

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 contracts 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:

Prosper Omnichain Fungible Token Contracts

Type	Solidity
Platform	EVM-compatible
Target	prosper-hash-token
Repository	https://github.com/hyplabs/prosper-hash-token ↗
Version	`e43be0e1802669b7716c406806f0b8f47b5f3587`

Programs	`Token.sol`
	`storage/TokenStorage.sol`
	`interfaces/IToken.sol`

2.4. Project Overview

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

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:

Jinheon Lee
Engineer
jinheon@zellic.io ↗

Juchang Lee
Engineer
lee@zellic.io ↗

Sylvain Pelissier
Engineer
sylvain@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 14, 2025	Kick-off call
January 14, 2025	Start of primary review period
January 15, 2025	End of primary review period

3. Detailed Findings

3.1. Burn request emitted on identical amounts

Target	Token		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

A user can request to burn some of their tokens by calling the `requestBurn` function. The function creates or updates the `burnRequests` in storage with the new amount:

```
function requestBurn(uint256 amount) external {
    Storage.Layout storage $ = Storage.layout();

    if (amount == 0) {
        revert RequestBurn__AmountIsZero();
    }

    (, uint256 currentBurnAmount) = $.burnRequests.tryGet(_msgSender());

    $.burnRequests.set(_msgSender(), amount);

    if (amount < currentBurnAmount) {
        _transfer(address(this), _msgSender(), currentBurnAmount - amount);
    } else if (amount > currentBurnAmount) {
        _transfer(_msgSender(), address(this), amount - currentBurnAmount);
    }

    emit BurnRequested(_msgSender(), amount);
}
```

Finally, it emits a `BurnRequested` event. However, if a user calls this function twice with the same amount, two identical events are emitted, even if no change happens regarding the contract.

Impact

Depending on the usage of this function, if a service or the front end uses those events, they may make misinformed decisions based on those events. For example, if the price of the token is updated and displayed from those events, it would lead to incorrectly displayed prices.

Recommendations

We recommend to revert if the amount is identical to the previous request, since the update is unnecessary.

Remediation

This issue has been acknowledged by Prosper.

3.2. Denial of service on pending requests

Target	Token		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `getPendingBurnRequests` function is used to retrieve all burn requests previously requested by users:

```
function getPendingBurnRequests()
    external
    view
    returns (BurnRequest[] memory requests)
{
    Storage.Layout storage $ = Storage.layout();
    uint256 length = $.burnRequests.length();

    requests = new BurnRequest[](length);

    for (uint256 i; i < length; ++i) {
        (address account, uint256 amount) = $.burnRequests.at(i);
        requests[i] = BurnRequest({ account: account, amount: amount });
    }
}
```

An attacker could maliciously request many small burns, increasing the number of requests until the call fails, running out of gas.

Impact

Even if the function is declared as `view`, depending on its usage, it may create a denial of service, preventing other parts of the system using this function from working properly or even completely blocking some features.

Recommendations

We recommend updating the function to access a single request based on the index of the request. This would prevent denial of service in the case of a large number of requests. It would leave the

other parts of the system to implement a proper way of handling the size of the requests together with the `getPendingBurnRequestsCount` function.

Remediation

This issue has been acknowledged by Prosper.

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. Centralization risk

There are four types of privileged accounts for the Token contract:

1. The minter
2. The burner
3. The upgrader
4. The owner

The minter can mint any number of new PROS tokens to any account.

The burner can burn tokens that were requested to be burned. However, user funds that are requested to be burned are not fully locked since a new burn request with a lower amount can be made to recover the difference. Since a burn request with a zero amount is not allowed, at least one token would be locked in the contract.

The upgrader can upgrade the contract to a new implementation.

The owner has the `DEFAULT_ADMIN_ROLE` role and thus can grant or revoke any of the previous roles. In addition, the owner can withdraw the former PROS tokens that have been swapped and transfer them to any receiver. Those tokens can be swapped again against new PROS tokens. The owner is also able to pause and unpause the contract.

The above introduces centralization risks that users should be aware of, as they grant a single point of control over the system.

We recommend that these centralization risks be clearly documented for users so that they are aware of the extent of the owner's control over the contract. This can help users make informed decisions about their participation in the project. Additionally, clear communication about the circumstances in which the owner may exercise these powers can help build trust and transparency with users. Therefore, it is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for owner access.

4.2. Missing upgrade test suite

The test suite is well-written and covers most of the edge cases. However, the upgradability of the Token contract is not covered by tests. We recommend building a rigorous test suite for this feature

as well to ensure that the system operates securely and as intended.

4.3. Possible gas optimization

The `getBurnRequest` function returns a burn-request amount associated with a given account. If the request does not exist, it returns zero. However, the internal behavior of `tryGet` is identical, making the last conditional assignment unnecessary. Here is a possible optimization:

```
function getBurnRequest(
    address account
) external view returns (uint256 amount) {
   (bool exists, uint256 value) = Storage.layout().burnRequests.tryGet(
   (, amount) = Storage.layout().burnRequests.tryGet(
        account
    );
   amount = exists ? value : 0;
}
```

Since this is a view function, it does not directly save gas. However, it may if the function is called in another transaction.

5. 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.

5.1. Module: Token.sol

Function: `batchExecuteBurn(address[] calldata accounts)`

The function burns a batch of tokens that were previously requested to be burned through the `requestBurn` function. The function can be executed only by the burner role.

Inputs

- `accounts`
  - **Control**: The `burnRequests` mapping is checked to contain a burning request for this account. The function reverts if the length of `accounts` is zero.
  - **Impact**: The contract cannot burn more or less tokens than requested to be burned.

Branches and code coverage (including function calls)

Intended branches

- Tokens are burned according the previous burn requests, and the requests are successfully removed.
  - ☑ Test coverage
- Tokens are burned according the previous burn requests, and the account with no requests are skipped.
  - ☑ Test coverage

Negative behavior

- The function reverts when the function is called by a non-burner role.
  - ☑ Negative test
- The function reverts when the array is empty.
  - ☑ Negative test

Function call analysis

- `executeBurn -> _burn(address(this), totalBurnAmount)`

- **External/Internal?** External.
- **Argument control?** `totalBurnAmount`.
- **Impact**: Burns the total amount of tokens requested.

Function: `batchMint(address[] accounts, uint256[] amounts)`

The function mints tokens in a single batch to each address in `accounts`, according to the corresponding amount in `amounts`.

Inputs

- `accounts`
  - **Control**: Fully controlled by the caller.
  - **Impact**: Addresses for minting.
- `amounts`
  - **Control**: Fully controlled by the caller.
  - **Impact**: Amounts for minting.

Branches and code coverage

Intended branches

- Check if `accounts` and `amounts`'s lengths are the same.
  - ☑ Test coverage
- Mint a batch to each address according to the corresponding amount in `amounts`.
  - ☑ Test coverage

Negative behavior

- If `accounts` and `amounts`'s lengths are different, revert with `BatchMint__InputArrayMismatch`.
  - ☑ Negative test

Function: `executeBurn(address account)`

The function burns the tokens that were previously requested to be burned by the `account` address through the `requestBurn` function. The function can be executed only by the burner role.

Inputs

- `account`
  - **Control**: The `burnRequests` mapping is checked to contain a burning request for this account.

- **Impact**: The contract cannot burn tokens that were not requested to be burned.

Branches and code coverage (including function calls)

Intended branches

- Tokens are burned according the previous burn request, and the request is successfully removed.
  - ☑ Test coverage

Negative behavior

- The function reverts when the request was not previously performed.
  - ☑ Negative test
- The function reverts when the function is called by a non-burner role.
  - ☑ Negative test

Function call analysis

- `executeBurn -> _burn(address(this), amount)`
  - **External/Internal?** External.
  - **Argument control?** `amount`.
  - **Impact**: Burn the amount of tokens requested.

Function: `requestBurn(uint256 amount)`

The function can be called by a user who wants to burn tokens or reduce the amount of tokens to be burned from a previous request. The difference of amount is sent either to the contract or to the user.

Inputs

- `amount`
  - **Control**: The amount is checked to be nonzero.
  - **Impact**: Prevents a user from completely canceling a previous burn request or sending a nonexisting request with zero amount.

Branches and code coverage (including function calls)

Intended branches

- Tokens are correctly transferred to the contract when a request is created.

- ☑ Test coverage

- Tokens are successfully transferred to the contract when the requested amount is increased.
  - ☑ Test coverage
- Tokens are successfully transferred to the user when the requested amount is decreased.
  - ☑ Test coverage

Negative behavior

- The function reverts when the amount exceeds user balance.
  - ☑ Negative test
- The function reverts when the amount is zero.
  - ☑ Negative test

Function call analysis

- `requestBurn -> _transfer(address(this), _msgSender(), currentBurnAmount - amount)`
  - **External/Internal?** Internal.
  - **Argument control?** `amount`.
  - **Impact**: Transfers tokens' difference to the contract.
- `_transfer(_msgSender(), address(this), amount - currentBurnAmount)`
  - **External/Internal?** Internal.
  - **Argument control?** `amount`.
  - **Impact**: Transfers tokens' difference to the user.

Function: `send(SendParam sendParam, MessagingFee fee, address refundAddress)`

This function debits tokens from the caller, builds and sends a cross-chain message with the provided fee, refunds leftovers to `refundAddress`, and returns messaging and OFT receipts.

This function is identical to the `send` function from LayerZero's `OFTCoreUpgradeable.sol` contract except that it adds additional zero checks on addresses, it uses `whenNotPaused` modifier and it uses `_msgSender()` instead of `msg.sender`.

Inputs

- `sendParam`
  - **Control**: Fully controlled by the caller.
  - **Impact**: Structure for sending includes the receiver address and amounts.
- `fee`
  - **Control**: Fully controlled by the caller.

    - **Impact**: Fee data for LayerZero sending.
- `refundAddress`
    - **Control**: Fully controlled by the caller.
    - **Impact**: Address for refunding with LayerZero sending.

Branches and code coverage

Intended branches

- Check if the receiver address and `refundAddress` are not zero.
    - ☑ Test coverage
- Check if the send amount is zero.
    - ☑ Test coverage

Negative behavior

- If the receiver address or `refundAddress` is zero, revert transaction with `Send__ZeroAddress`.
    - ☑ Negative test
- If the send amount is zero, revert transaction with `Send__ZeroAmount`.
    - ☑ Negative test

Function: `swapToken(uint256 amount)`

This function receives `SWAPPABLE_TOKEN` and mints this contract token to the caller.

Inputs

- `amount`
    - **Control**: Fully controlled by the caller.
    - **Impact**: Amount of swap.

Branches and code coverage

Intended branches

- Check if `SWAPPABLE_TOKEN` is not zero address.
    - ☑ Test coverage
- Receive an amount of `SWAPPABLE_TOKEN`.
    - ☑ Test coverage
- Mint an amount of the contract token to the caller.
    - ☑ Test coverage

Negative behavior

- If `SWAPPABLE_TOKEN` is zero address, revert transaction with `SwapToken__TokenDoesNotExist`.
  - ☑ Negative test

Function: `withdrawSwappables(address receiver)`

The function allows the owner of the contract to withdraw the total balance of former PROS tokens that were previously swapped by the users with the `swapToken` function. The function reverts if `SWAPPABLE_TOKEN` is set to the zero address. Note that those tokens can then by swapped again to obtain new PROS tokens.

Inputs

- `receiver`
  - **Control**: Implicit validation that the receiver is not the zero address by the `safeTransfer` function.
  - **Impact**: Withdraws the former PROS tokens to any receiver address.

Branches and code coverage (including function calls)

Intended branches

- Tokens are successfully withdrawn to the receiver by the owner.
  - ☑ Test coverage

Negative behavior

- The function reverts when the function is called by an address other than the owner.
  - ☑ Negative test
- The function reverts when the `SWAPPABLE_TOKEN` is set to zero.
  - ☑ Negative test
- The function reverts when the receiver address is set to zero.
  - ☐ Negative test

Function call analysis

- `withdrawSwappables -> safeTransfer`
  - **External/Internal?** External.
  - **Argument control?** `receiver`.
  - **Impact**: The tokens are transferred to the `receiver` address.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to both Binance Smart Chain ↗ and Ethereum ↗ Mainnets, but the contracts were paused.

During our assessment on the scoped Prosper Omnichain Fungible Token contracts, we discovered two findings, both of which were low impact. Prosper carefully reviewed and acknowledged the low-impact issues.

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.