Zellic

**Prepared for**
Gabe Rodriguez & Jon Cinque
Anza

**Prepared by**
Frank Bachman
Nathanial Lattimer
Zellic

**May 16, 2025**

# SPL Token Wrap Program

## Solana Application 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 Anza from May 5th to May 8th, 2025. During this engagement, Zellic reviewed SPL Token Wrap Program'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:

- How does the Token Wrap program ensure that wrapped tokens are backed by their originating assets?
- Could an on-chain attacker freeze wrapped assets owned by other users?
- Can the Token Wrap program be used to mint unlimited wrapped tokens?

## 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.

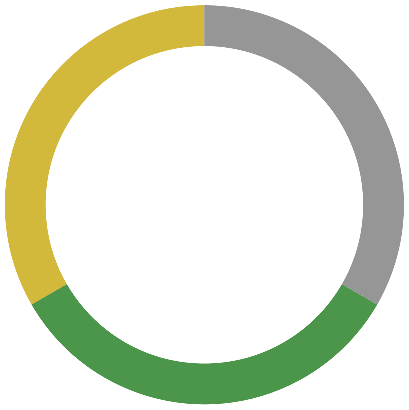
## 1.4. Results

During our assessment on the scoped SPL Token Wrap Program programs, we discovered three findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining finding was informational in nature.

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

## Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| 🟥 Critical | 0 |
| 🟧 High | 0 |
| 🟨 Medium | 1 |
| 🟩 Low | 1 |
| ⬜ Informational | 1 |



# 2. Introduction

## 2.1. About SPL Token Wrap Program

Anza contributed the following description of SPL Token Wrap Program:

> The SPL Token Wrap Program enables the creation of "wrapped" versions of existing SPL tokens, facilitating interoperability between different token standards. App developers that are find themselves wishing they could take advantage of some of the latest features of a specific token program, this program helps with that.

## 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 programs.

**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 programs 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:

### SPL Token Wrap Program Programs

| | |
|---|---|
| **Type** | Rust |
| **Platform** | SVM-compatible |

| | |
|---|---|
| **Target** | token-wrap |
| **Repository** | https://github.com/solana-program/token-wrap ↗ |
| **Version** | `75c5529d5a191f12bd58b6b92ca0104ce3464763` |
| **Programs** | `token-wrap` |

## 2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.2 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

## 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 ↗

**Nathanial Lattimer**
Engineer
d0nut@zellic.io ↗

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **May 5, 2025** | Start of primary review period |
| **May 8, 2025** | End of primary review period |

## 3. Detailed Findings

### 3.1. Escrow accounts are not bound to caller, resulting in denial of service

| | | | |
|---|---|---|---|
| **Target** | process_unwrap | | |
| **Category** | Coding Mistakes | **Severity** | Medium |
| **Likelihood** | Medium | **Impact** | Medium |

#### Description

The escrow account is an important component of the token wrapping process. It is used to hold the assets that back the wrapped tokens on chain. The main constraint applied to the escrow account is that the wrapped token authority is the authority of this account. It does not enforce that the escrow account is bound to the caller that wrapped the tokens to begin with.

This presents a potential problem. This design flaw means that other entities with wrapped tokens can use another entity's escrow account to unwrap their tokens to the source token. This forces the original caller to find another escrow with sufficient funds to perform the transfer, or they will be unable to unwrap their token at all. A caller may be forced to split up an unwrap call into multiple smaller transactions to satisfy the demand.

#### Impact

Callers may have to find escrow accounts other than the original escrow they created and intended for the wrapped mint in order to satisfy future unwrap calls. Programs expecting the escrow account to be the same for wrapping/unwrapping might experience denial of service on withdrawals.

#### Recommendations

Create a single escrow account via an ATA associated with the wrapped mint.

#### Remediation

This issue has been acknowledged by Anza, and a fix was implemented in commit `6c29a885` ↗.

## 3.2. Missing ownership check on unwrapped mint account

| | | | |
|---|---|---|---|
| **Target** | process_create_mint | | |
| **Category** | Business Logic | **Severity** | Low |
| **Likelihood** | Medium | **Impact** | Low |

### Description

The `process_create_mint` instruction takes in a variety of accounts as arguments to construct a wrapped token mint. One of the accounts, the `unwrapped_mint_account`, corresponds to the account that the wrapped mint should be based on. The idea is that the wrapped mint should be based on a current, legitimate mint, as suggested by the implementation in that the function deserializes the account, copying relevant information, to craft a new mint that wraps the source mint's tokens.

This function is missing an important constraint. The `unwrapped_mint_account` *is* deserialized, confirming that the account's data matches that of a mint account, but the ownership is never checked. This means that any account can be supplied, as long as it deserializes as expected. Thankfully, given the escrow mechanism of the wrapping, this new pseudo mint should not be usable but may break third parties not anticipating the deviant mint.

### Impact

Wrapped token mints can be created that do not correspond to a legitimate source mint.

### Recommendations

Add an ownership check on the mint to confirm that it belongs to either the SPL Token or SPL Token 2022 program to ensure the source mint is legitimate.

### Remediation

This issue has been acknowledged by Anza, and a fix was implemented in commit `5cc59df2` ↗.

## 3.3. Mint-creation error message uses excessive rent

| Target | process_create_mint | | |
|---|---|---|---|
| Category | Coding Mistakes | Severity | Informational |
| Likelihood | N/A | Impact | Informational |

### Description

In `process_create_mint`, the backpointer PDA account is created to associate the newly created wrapped token mint with the original mint. As the account must be created, rent is required to be supplied for the account. Rent requirement is enforced by determining the space required to store the backpointer and then checking the `minimum_balance(..)` required via the `Rent` sysvar.

If the rent requirement is not met, an error message is logged with the amount of rent required.

```
let backpointer_rent_required = rent.minimum_balance(space);
if wrapped_backpointer_account.lamports() <
    rent.minimum_balance(backpointer_space) {
    msg!(
        "Error: wrapped_backpointer_account requires pre-funding of {}
    lamports",
        backpointer_rent_required
    );
    Err(ProgramError::AccountNotRentExempt)?
}
```

In this instance, the `backpointer_rent_required` is incorrectly computed as it uses the `space` variable defined earlier via `spl_token_2022::state::Mint::get_packed_len()` rather than the correct `backpointer_space` variable. However, the `backpointer_rent_required` amount is only used in the error log. The check itself uses the correct amount with `backpointer_space`.

### Impact

The required rent amount present in the error log is higher than actually required by the backpointer account.

### Recommendations

Compute the rent for `backpointer_rent_required` with `backpointer_space`. Also, this can be reused for the check instead of calling `rent.minimum_balance()` again.

### Remediation

This issue has been acknowledged by Anza, and a fix was implemented in commit `84e30ae8` ↗.

## 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. Wrapped-mint freeze authority

Token mints have an optional freeze authority, which allows the designated authority to freeze or unfreeze token accounts. This enables or disables the ability to transfer tokens from those accounts.

When the token wrap program initializes a wrapped mint, it assigns the same freeze authority as the base mint. However, if the freeze authority of the base mint is later updated using the `SetAuthority` instruction, the wrapped mint's freeze authority does not automatically update. This can result in the base mint and the wrapped mint's freeze authorities falling out of sync.

Note that it is still possible for the original freeze authority to update the authority in the wrapped mint through the `SetAuthority` instruction.

## 5. Threat Model

As time permitted, we analyzed each instruction in the program and created a written threat model for the most critical instructions. A threat model documents the high-level functionality of a given instruction, the inputs it receives, and the accounts it operates on as well as the main checks performed on them; it gives an overview of the attack surface of the programs and of the level of control an attacker has over the inputs of critical instructions.

For brevity, system accounts and well-known program accounts have not been included in the list of accounts received by an instruction; the instructions that receive these accounts make use of Anchor types, which automatically ensure that the public key of the account is correct.

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

### 5.1. Program: Token Wrap

#### Instruction: `CreateMint`

This instruction creates a wrapped token mint, enabling the wrapping of tokens between the SPL Token and the Token 2022 programs. It establishes a wrapped mint and a backpointer account to link the wrapped mint to its unwrapped counterpart. The caller must prefund the wrapped mint and backpointer accounts with sufficient lamports to cover rent exemption.

**Input parameters**

- `idempotent: bool`: Determines the instruction's behavior if the wrapped mint or backpointer accounts already exist. If `true`, the instruction is idempotent and will not fail if the accounts are already initialized. If `false`, it fails if either account is already initialized.

**Accounts**

- **`wrapped_mint_account`**: The unallocated account that will become the wrapped mint.
    - Signer: No.
    - Init: Yes, the instruction initializes this account as a mint.
    - PDA: Yes, derived using `get_wrapped_mint_address(unwrapped_mint_address, wrapped_token_program_id)`.
    - Mutable: Yes, the account is written to during initialization.
    - Constraints: Must be unallocated initially, address must match the PDA derivation, and must be prefunded with enough lamports to be rent-exempt.
- **`wrapped_backpointer_account`**: The unallocated account that will store the backpointer to the unwrapped mint.
    - Signer: No.

    - Init: Yes, the instruction initializes this account.
    - PDA: Yes, derived using `get_wrapped_mint_backpointer_address(wrapped_mint_address)`.
    - Mutable: Yes, the account is written to during initialization.
    - Constraints: Must be unallocated initially, address must match the PDA derivation, and must be prefunded with enough lamports to be rent-exempt.
- **`unwrapped_mint_account`**: The existing mint account being wrapped.
    - Signer: No.
    - Init: No, it must already exist.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be an initialized mint account.
- **`system_program_account`**: The Solana system program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program ID (`solana_system_interface::program::id()`).
- **`wrapped_token_program_account`**: The SPL Token program that will manage the wrapped mint (e.g., Token 2022 or the legacy SPL Token program).
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be a valid SPL Token program account.

**Additional checks and behavior**

- **PDA derivation validation**: It verifies that the `wrapped_mint_account` and `wrapped_backpointer_account` addresses match their expected PDA derivations based on the `unwrapped_mint_address` and `wrapped_token_program_id`, or `wrapped_mint_address`, respectively.
- **Idempotency**: If `idempotent` is `true`, the instruction skips initialization if the accounts are already initialized and owned correctly (wrapped mint by the token program, backpointer by the program ID). If `false`, it fails if either account is initialized.
- **Rent checks**: It ensures both `wrapped_mint_account` and `wrapped_backpointer_account` have sufficient lamports to meet the rent-exemption threshold for their respective sizes (`Mint` size and `Backpointer` size).
- **Mint initialization**: The wrapped mint is initialized with the same decimals and freeze

authority as the unwrapped mint. The mint authority is set to a PDA derived from the wrapped mint address (`get_wrapped_mint_authority`).
- **Backpointer initialization**: The backpointer account is set to store the `unwrapped_mint_address`, enabling reverse lookup from wrapped to unwrapped mint.
- **Ownership checks**: If idempotent, it verifies that the initialized `wrapped_mint_account` is owned by the `wrapped_token_program_account` and the `wrapped_backpointer_account` is owned by the program ID.

**CPI**

- **Allocate and assign**: It calls `allocate` and `assign` via CPI to allocate space and assign ownership of the `wrapped_mint_account` to the `wrapped_token_program_account` and the `wrapped_backpointer_account` to the program ID. These are signed with the PDA seeds.
- **Initialize mint**: It calls `initialize_mint2` via CPI to initialize the `wrapped_mint_account` as a mint with the specified decimals, mint authority, and freeze authority.

### Instruction: `Wrap`

This instruction wraps tokens by transferring a specified amount of unwrapped tokens from a user's account to an escrow account, then minting an equivalent amount of wrapped tokens to a recipient account. It facilitates the transition of tokens from one program (e.g., legacy SPL Token) to another (e.g., Token 2022).

**Input parameters**

- `amount: u64`: The number of tokens to wrap. Must be greater than zero.

**Accounts**

- **`recipient_wrapped_token_account`**: The token account that will receive the newly minted wrapped tokens.
    - Signer: No.
    - Init: No, must be preinitialized.
    - PDA: No.
    - Mutable: Yes, wrapped tokens are minted to this account.
    - Constraints: Must be associated with the `wrapped_mint`.
- **`wrapped_mint`**: The mint account for the wrapped tokens.
    - Signer: No.
    - Init: No, must be preinitialized.

  - PDA: Yes, derived using `get_wrapped_mint_address(unwrapped_mint_address, wrapped_token_program_id)`.
  - Mutable: Yes, its supply increases as tokens are minted.
  - Constraints: Must be initialized and match the PDA derivation.
- **`wrapped_mint_authority`**: The authority that controls minting for the wrapped mint.
  - Signer: No.
  - Init: No.
  - PDA: Yes, derived using `get_wrapped_mint_authority(wrapped_mint_address)`.
  - Mutable: No.
  - Constraints: Must match the PDA derivation.
- **`unwrapped_token_program`**: The SPL Token program managing the unwrapped tokens (e.g., legacy SPL Token or Token 2022).
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the token program ID for the unwrapped tokens.
- **`wrapped_token_program`**: The SPL Token program managing the wrapped tokens.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the token program ID for the wrapped tokens.
- **`unwrapped_token_account`**: The user's token account containing the unwrapped tokens to wrap.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes, tokens are transferred out of this account.
  - Constraints: Must be associated with the `unwrapped_mint`.
- **`unwrapped_mint`**: The mint account for the unwrapped tokens.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the mint for the unwrapped tokens.
- **`unwrapped_escrow`**: The escrow account that holds unwrapped tokens after wrapping.

    - Signer: No.
    - Init: No, must be preinitialized.
    - PDA: No.
    - Mutable: Yes, tokens are transferred into this account.
    - Constraints: Must be owned by the `wrapped_mint_authority`.
- **`transfer_authority`**: The authority that can transfer tokens from the `unwrapped_token_account`.
    - Signer: Yes, unless it is a multi-sig (then optional).
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must have authority to transfer tokens from the `unwrapped_token_account`.
- **`multisig_signer_pubkeys`**: Optional additional signers for a multi-sig transfer authority.
    - Signer: Yes, if present.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Required if `transfer_authority` is a multi-sig — the number of signers must match the multi-sig requirements.

**Additional checks and behavior**

- **Amount validation**: It ensures `amount` is greater than zero to prevent invalid wraps.
- **Account validation**: It verifies that `wrapped_mint` matches the PDA derived from `unwrapped_mint` and `wrapped_token_program`. It confirms `wrapped_mint_authority` matches the PDA derived from `wrapped_mint`.
- **Escrow ownership**: It checks that `unwrapped_escrow` is owned by the `wrapped_mint_authority`.
- **Token transfer**: It transfers `amount` of unwrapped tokens from `unwrapped_token_account` to `unwrapped_escrow`.
- **Token minting**: It mints `amount` of wrapped tokens to `recipient_wrapped_token_account` using the `wrapped_mint_authority`.

**CPI**

- **Transfer checked**: It calls `invoke_transfer_checked` to transfer `amount` of unwrapped tokens from `unwrapped_token_account` to `unwrapped_escrow`, using the `transfer_authority` and any multi-sig signers.

- **Mint to**: It calls `mint_to` via CPI, signed with the `wrapped_mint_authority` PDA seeds, to mint `amount` of wrapped tokens to `recipient_wrapped_token_account`.

## Instruction: `Unwrap`

This instruction unwraps tokens by burning a specified amount of wrapped tokens from a user's account and transferring an equivalent amount of unwrapped tokens from an escrow account to a recipient account. It reverses the wrapping process, returning tokens to their original form.

**Input parameters**

- `amount: u64`: The number of tokens to unwrap. Must be greater than zero.

**Accounts**

- **`unwrapped_escrow`**: The escrow account holding unwrapped tokens.
    - Signer: No.
    - Init: No, must be preinitialized.
    - PDA: No.
    - Mutable: Yes, tokens are transferred out of this account.
    - Constraints: Must be owned by the `wrapped_mint_authority`.
- **`recipient_unwrapped_token`**: The token account that will receive the unwrapped tokens.
    - Signer: No.
    - Init: No, must be preinitialized.
    - PDA: No.
    - Mutable: Yes, tokens are transferred into this account.
    - Constraints: Must be associated with the `unwrapped_mint`.
- **`wrapped_mint_authority`**: The authority that controls the wrapped mint.
    - Signer: No.
    - Init: No.
    - PDA: Yes, derived using `get_wrapped_mint_authority(wrapped_mint_address)`.
    - Mutable: No.
    - Constraints: Must match the PDA derivation.
- **`unwrapped_mint`**: The mint account for the unwrapped tokens.
    - Signer: No.
    - Init: No.
    - PDA: No.

    - Mutable: No.
    - Constraints: Must be the mint for the unwrapped tokens.
- **`wrapped_token_program`**: The SPL Token program managing the wrapped tokens.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the token program ID for the wrapped tokens.
- **`unwrapped_token_program`**: The SPL Token program managing the unwrapped tokens.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the token program ID for the unwrapped tokens.
- **`wrapped_token_account`**: The user's token account containing the wrapped tokens to unwrap.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes, tokens are burned from this account.
    - Constraints: Must be associated with the `wrapped_mint`.
- **`wrapped_mint`**: The mint account for the wrapped tokens.
    - Signer: No.
    - Init: No, must be preinitialized.
    - PDA: Yes, derived using `get_wrapped_mint_address(unwrapped_mint_address, wrapped_token_program_id)`.
    - Mutable: Yes, its supply decreases as tokens are burned.
    - Constraints: Must be initialized and match the PDA derivation.
- **`transfer_authority`**: The authority that can burn tokens from the `wrapped_token_account`.
    - Signer: Yes, unless it is a multi-sig (then optional).
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must have authority to burn tokens from the `wrapped_token_account`.
- **`multisig_signer_pubkeys`**: Optional additional signers for a multi-sig transfer authority.

- Signer: Yes, if present.
- Init: No.
- PDA: No.
- Mutable: No.
- Constraints: Required if `transfer_authority` is a multi-sig — the number of signers must match the multi-sig requirements.

**Additional checks and behavior**

- **Amount validation**: It ensures `amount` is greater than zero to prevent invalid unwraps.
- **Account validation**: It verifies that `wrapped_mint` matches the PDA derived from `unwrapped_mint` and `wrapped_token_program`. It confirms `wrapped_mint_authority` matches the PDA derived from `wrapped_mint`.
- **Token burning**: It burns `amount` of wrapped tokens from `wrapped_token_account` using the `transfer_authority`.
- **Token transfer**: It transfers `amount` of unwrapped tokens from `unwrapped_escrow` to `recipient_unwrapped_token` using the `wrapped_mint_authority`.

**CPI**

- **Burn**: It calls `burn` via CPI to burn `amount` of wrapped tokens from `wrapped_token_account`, authorized by `transfer_authority` and any multi-sig signers.
- **Transfer checked**: It calls `invoke_transfer_checked` to transfer `amount` of unwrapped tokens from `unwrapped_escrow` to `recipient_unwrapped_token`, signed with the `wrapped_mint_authority` PDA seeds.

## 6. Assessment Results

During our assessment on the scoped SPL Token Wrap Program programs, we discovered three findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining finding was 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.