Zellic

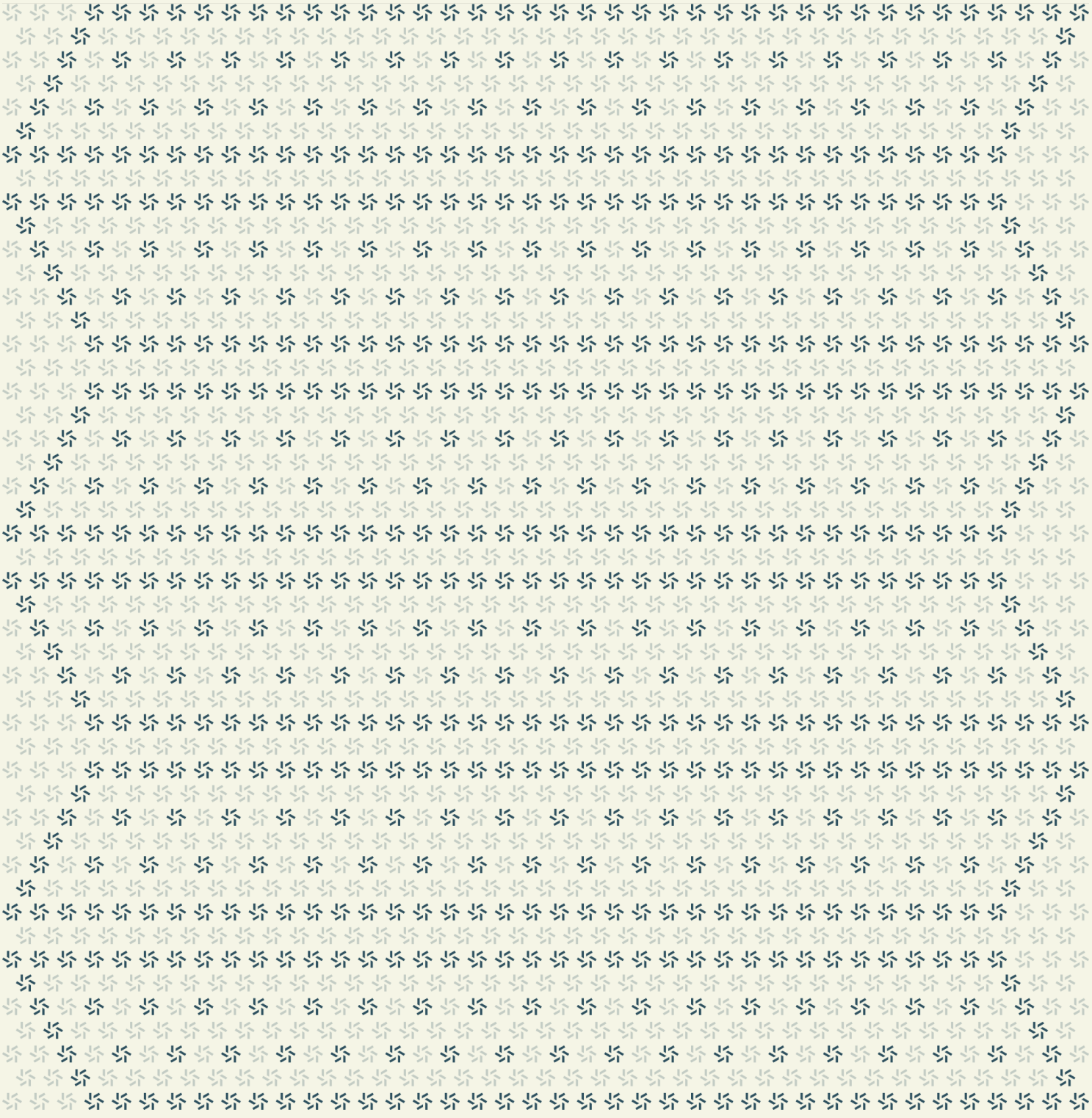
Prepared for
Henryk Sarat
Paxos

Prepared by
Frank Bachman
Junyi Wang
Zellic

October 24, 2024

Mint Controller

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 Paxos from October 14th to October 16th, 2024. During this engagement, Zellic reviewed Mint Controller'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 there proper access control for the minter-controller program?
- Is the warm minting being restricted to the intended minting limits?
- Is the warm minting only being allowed to the whitelisted addresses?

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

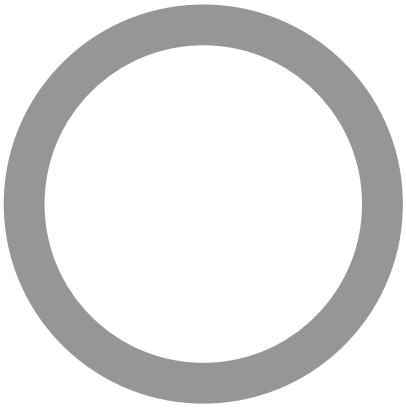
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Mint Controller programs, we discovered three findings, all of which were informational in nature.

Breakdown of Finding Impacts

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



2. Introduction

2.1. About Mint Controller

Paxos contributed the following description of Mint Controller:

> The minter_controller program is being created to support Paxos warm minting capabilities. This program adds transaction controls to Solana mint authorities for the token-2022 program. The transaction controls include rate limitings and whitelisting. The program requires the token mint authority to be a spl token multisig.

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.

2.3. Scope

The engagement involved a review of the following targets:

Mint Controller Programs

Type	rust
Platform	Solana
Target	solana-programs
Repository	https://github.com/paxosglobal/solana-programs ↗
Version	`7fafbe6d7b11084f048b51da86e0e3d7fc590949`
Programs	`mint-controller`

2.4. Project Overview

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

Frank Bachman
Engineer
frank@zellic.io ↗

Junyi Wang
Engineer
junyi@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 14, 2024	Start of primary review period
October 16, 2024	End of primary review period

3. Detailed Findings

3.1. Account sizes are hardcoded

Target	mint-controller		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the `AddMinter` and `AddWhitelistedAddress` instructions, the space required to be reserved for program accounts is hardcoded.

```
#[account(
    init,
    payer = payer,
    space = 8 + 32 + 32 + 32 + 1, ////8 discriminator + 32 minter_authority +
    32 mint account + 32 to address + 1 bump
    seeds = [b"mint-whitelist", minter_authority.key().as_ref(),
    mint_account.key().as_ref(), to_address.key().as_ref()],
    bump
)]
pub whitelisted_address: Account<'info, WhitelistedAddress>,
```

Impact

The size reserved for the accounts is correct; however, hardcoding these values in the program could leave room for error, especially if any changes are made to the account structure.

Recommendations

Anchor provides a convenient way to automatically compute the required size with the `InitSpace` macro. More information can be found in the Anchor documentation ↗.

Remediation

This issue has been acknowledged by Paxos, and a fix was implemented in commit `df763051` ↗.

3.2. The `AddMinter` admin is set through instruction parameters

Target	mint-controller		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `AddMinter` sets the minter admin directly through the instruction parameters. The admin is not required to be a signer.

```
pub fn add_minter(ctx: Context<AddMinter>, capacity: u64, refill_per_second:
    u64, admin: Pubkey) -> Result<()> {
    ctx.accounts.minter.minter_authority
    = ctx.accounts.minter_authority.key();
    ctx.accounts.minter.mint_account = ctx.accounts.mint_account.key();
    ctx.accounts.minter.bump = ctx.bumps.minter;
    ctx.accounts.minter.admin = admin;
    ctx.accounts.minter.pending_admin = None;
```

Impact

Setting the admin directly through the instruction parameters can leave room for error. If the admin is set to an incorrect address, this cannot be updated later.

Recommendations

To fix this issue, the admin could be set to `mint_authority` by default and updated later through the admin-transfer process. Alternatively, a signature could be required from the admin account when adding the minter.

Remediation

This issue has been acknowledged by Paxos, and a fix was implemented in commit `df763051` ↗.

3.3. The test to mint from the legacy token program is incorrect

Target	mint-controller		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the test below, `it` tries to use the legacy token program to mint tokens through the `mint-controller`. The expected result is to fail as token-2022 is not used. The transaction does fail; however, this is due to the mint amount exceeding the capacity set within `mint-controller`.

```
it('Cannot mint using token program with TOKEN-2022 mint account', async ()
    => {
  // Derive the associated token address account for the mint and payer.
  const associatedTokenAccountAddress = getAssociatedTokenAddressSync(mintPDA,
    payer.publicKey, true, TOKEN_PROGRAM_ID);

  try {
    const mintTokenSignature = await minterControllerProgram.methods
    .mintToken(capacity)
    .accounts({
      payer: minterAuthorityKeypair.publicKey,
      minterAuthority: minterAuthorityKeypair.publicKey,
      toAddress: payer.publicKey,
      associatedTokenAccount: associatedTokenAccountAddress,
      mintAccount: mintPDA,
      mintMultisig: mintMultisigAddr,
      tokenProgram: TOKEN_PROGRAM_ID,
    })
    .signers([minterAuthorityKeypair])
    .rpc();
    assert.fail('Should fail when using TOKEN_PROGRAM_ID')
  } catch (err) {
    assert.isTrue(err.toString().includes('incorrect program id for
    instruction'))
  }
});
```

The `check_spl_token_program_account` function verifies that the token program address is implemented as follows:

```
pub fn check_spl_token_program_account(spl_token_program_id: &Pubkey) ->
    ProgramResult {
    if spl_token_program_id != &id()
    && spl_token_program_id != &spl_token::id() {
        return Err(ProgramError::IncorrectProgramId);            /// ^^^
    TOKEN_PROGRAM_ID
    }  // spl_token_program_id = TOKEN_2022_PROGRAM_ID
    Ok(())
}
```

This allows for the token program address to be either the legacy token program or the token-2022 program. We wrote another test to verify that minting is possible through the legacy token program:

```
it('Audit::Cannot mint using token program with TOKEN-2022 mint account',
    async () => {
  // Derive the associated token address account for the mint and payer.
  const associatedTokenAccountAddress = getAssociatedTokenAddressSync(mintPDA,
    payer.publicKey, true, TOKEN_PROGRAM_ID);

  try {
    const mintTokenSignature = await minterControllerProgram.methods
    .mintToken(amount)
    .accounts({
      payer: minterAuthorityKeypair.publicKey,
      minterAuthority: minterAuthorityKeypair.publicKey,
      toAddress: payer.publicKey,
      associatedTokenAccount: associatedTokenAccountAddress,
      mintAccount: mintPDA,
      mintMultisig: mintMultisigAddr,
      tokenProgram: TOKEN_PROGRAM_ID,
    })
    .signers([minterAuthorityKeypair])
    .rpc();
    let tokenAmount = await
    provider.connection.getTokenAccountBalance(associatedTokenAccountAddress);
    assert.equal(amount.toString(), tokenAmount.value.uiAmountString)
  } catch (err) {
    assert.isTrue(err.toString().includes('incorrect program id for
    instruction'))
  }
});
```

This test runs successfully and is able to mint through the legacy token program.

Impact

There is no security impact, as it is safe to mint through the legacy token program.

Recommendations

If required, a constraint can be added to the `MintToken` instruction to restrict `token_program` to be the token-2022 program address.

Remediation

This issue has been acknowledged by Paxos.

4. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the programs 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. Program: minter-controller

Function: `accept_admin_transfer(ctx)`

This is called by the new admin to accept an admin transfer.

Inputs

- `payer`
  - **Validation**: Must be signer.
  - **Impact**: Unused.
- `pending_admin`
  - **Validation**: Must be signer and must match the currently registered pending admin in the `Minter`.
  - **Impact**: The new admin will be equal to this address.
- `minter_authority`
  - **Validation**: Must be used in the seed of the minter.
  - **Impact**: Used to derive the `Minter` PDA.
- `mint_account`
  - **Validation**: Must be SPL `Mint`.
  - **Impact**: Used to derive the `Minter` PDA.
- `minter`
  - **Validation**: Must be a PDA derived from `minter_authority` and `mint_account`.
  - **Impact**: The `Minter` whose admin is modified.

Branches and code coverage (including function calls)

Intended branches

- Successfully updates the admin to the `pending_admin`.
  - ☑ Test coverage

Negative behavior

- Reverts if the `pending_admin` is not set.

- ☐ Negative test
- Reverts if the new `pending_admin` does not match the `pending_admin` of the `minter`.
  - ☑ Negative test
- Reverts if the `pending_admin` did not sign the transaction.
  - ☑ Negative test
- Reverts if the `minter_authority` does not match the `minter`.
  - ☐ Negative test
- Reverts if the `mint_account` does not match the `minter`.
  - ☐ Negative test
- Reverts if the passed-in `mint_account` is not an SPL `Mint`.
  - ☐ Negative test
- Reverts if the `minter` is not a PDA of the current program.
  - ☐ Negative test

Function: `add_minter(ctx, capacity, refill_per_second, admin)`

This creates a new `Minter` object, which manages the passed-in `Mint` with a rate limit.

Inputs

- `payer`
  - **Validation**: Must be a signer.
  - **Impact**: Used to pay for gas costs only.
- `minter_authority`
  - **Validation**: Must be a signer.
  - **Impact**: The account must sign on any mints by this `Minter`.
- `mint_account`
  - **Validation**: Must be an SPL `Mint` account.
  - **Impact**: This is the `Mint` that is used to mint tokens by this `Minter`.
- `minter`
  - **Validation**: Must be a valid PDA address of the current program derived using the `minter_authority` and `mint_account` given.
  - **Impact**: The new `Minter` is initialized here.
- `system_program`
  - **Validation**: Must be the system program.
  - **Impact**: Used internally by Anchor.
- `capacity`
  - **Validation**: None.
  - **Impact**: Max capacity of the rate limit.
- `refill_per_second`
  - **Validation**: None.
  - **Impact**: Rate-limit quota that is regenerated every second.

- `admin`
  - **Validation**: None.
  - **Impact**: This account must be a signer when the rate limit is modified.

Branches and code coverage (including function calls)

Intended branches

- Creates a `Minter` with the given parameters.
  - ☑ Test coverage

Negative behavior

- Reverts if `minter_authority` is not a signer.
  - ☑ Negative test
- Reverts if `mint_account` is not an SPL `Mint` account.
  - ☐ Negative test
- Reverts if `minter` is not a PDA address of the current program derived in the aforementioned manner.
  - ☑ Negative test
- Reverts if `minter` is already initialized.
  - ☑ Negative test

Function: `add_whitelisted_address(ctx)`

This adds a whitelisted address account that can be used when minting.

Inputs

- `payer`
  - **Validation**: Must be a signer.
  - **Impact**: Used to pay for the creation of the new whitelist account.
- `admin`
  - **Validation**: Must be a signer.
  - **Impact**: Must match the admin stored in the `Minter`.
- `minter_authority`
  - **Validation**: Must match the `minter_authority` stored in the `Minter`.
  - **Impact**: Stored in the whitelist account to derive the `Minter` address.
- `mint_account`
  - **Validation**: Must match the `mint_account` stored in the `Minter`.
  - **Impact**: Stored in the whitelist account to derive the `Minter` address.
- `minter`
  - **Validation**: Must be a PDA of the current program derived from the

`minter_authority` and `mint_account`.
    - **Impact**: The address whitelisted can now have tokens minted to it from this `Minter`.
- `to_address`
    - **Validation**: None. This is intentional, since any account could be the target of a mint.
    - **Impact**: This address is whitelisted, which means mints can now target this account.
- `whitelisted_address`
    - **Validation**: Must not be initialized already and must be be a PDA derived from `minter_authority`, `mint_account`, and `to_address`.
    - **Impact**: This is the address that stores the whitelist entry itself.
- `system_program`
    - **Validation**: Must be the system program.
    - **Impact**: Used internally by Anchor.

Branches and code coverage (including function calls)

Intended branches

- Correctly initializes the whitelist entry account.
    - ☑ Test coverage

Negative behavior

- Reverts if the `admin` is not a signer.
    - ☑ Negative test
- Reverts if the `minter_authority` does not match the `minter`.
    - ☐ Negative test
- Reverts if the `mint_account` does not match the `minter`.
    - ☐ Negative test
- Reverts if the `admin` does not match the `minter`.
    - ☐ Negative test
- Reverts if the `whitelisted_address` is not a PDA derived from the aforementioned items.
    - ☐ Negative test
- Reverts if the `whitelisted_address` is already initialized.
    - ☑ Negative test

Function: `mint_token(ctx, amount)`

This mints the token in the amount given to a whitelisted address if the rate limit allows it.

Inputs

- `payer`
  - **Validation**: Must be a signer.
  - **Impact**: Pays for gas needed to create the Associated Token Account (ATA) if it needs to be created.
- `minter_authority`
  - **Validation**: Must be a signer and match the `minter`.
  - **Impact**: Not used, except to check that the caller is authorized to mint tokens.
- `mint_multisig`
  - **Validation**: None but verified by the SPL token when it is used to check the multi-sig signature.
  - **Impact**: This is used by the SPL token to verify the signatures required to mint tokens.
- `mint_account`
  - **Validation**: Must be an SPL `Mint`.
  - **Impact**: Used by the SPL token when minting.
- `to_address`
  - **Validation**: Must match the whitelist entry passed in.
  - **Impact**: The new tokens are minted for this owner.
- `whitelist`
  - **Validation**: Must be a PDA of the current program derived from the `minter_authority`, `mint_account`, and `to_address`.
  - **Impact**: This is used to check if the `to_address` can be minted to from the current mint.
- `associated_token_account`
  - **Validation**: Must be an ATA matching the `mint_account`, `to_address`, and `token_program`.
  - **Impact**: This is used to hold the actual tokens minted.
- `token_program`
  - **Validation**: Must be an SPL token program.
  - **Impact**: Used in the call to `mint_to`.
- `associated_token_program`
  - **Validation**: This account' is unused.
  - **Impact**: N/A.
- `system_program`
  - **Validation**: Must be the system program.
  - **Impact**: Used internally by Anchor.

Branches and code coverage (including function calls)

Intended branches

- Mints the requested tokens to the given ATA when the parameters are correct.

- ☑ Test coverage

Negative behavior

- Reverts if the `minter_authority` is not a signer.
  - ☑ Negative test
- Reverts if the `minter_authority` does not match the `minter`.
  - ☑ Negative test
- Reverts if the `mint_account` does not match the `minter`.
  - ☐ Negative test
- Reverts if the `to_address` does not match the whitelist.
  - ☐ Negative test
- Reverts if the `whitelist` is not a PDA as described above.
  - ☑ Negative test
- Reverts if the `associated_token_account` does not match the other parameters as described above.
  - ☐ Negative test
- Reverts if the rate limit has been hit.
  - ☑ Negative test

Function: `remove_whitelisted_address(ctx)`

This deletes a whitelisted address entry account, rendering the whitelisted address no longer able to be minted to.

Inputs

- `payer`
  - **Validation**: Must be a signer.
  - **Impact**: The rent from the deleted account is refunded to this address.
- `admin`
  - **Validation**: Must be a signer and must match the `minter`.
  - **Impact**: This is only used to check that the caller is authorized.
- `minter_authority`
  - **Validation**: Must match the `minter`.
  - **Impact**: Used to derive the PDA address of the `minter`.
- `mint_account`
  - **Validation**: Must be a SPL `Mint` that matches the `minter`.
  - **Impact**: Used to derive the PDA address of the `minter`.
- `minter`
  - **Validation**: Must be PDA owned by the current program that contains a `Minter`.
  - **Impact**: Used to check the `minter_authority` and `mint_account`.

- `to_address`
  - **Validation**: Used in the derivation for the whitelist entry account.
  - **Impact**: The address whose whitelist entry is being deleted.
- `whitelisted_address`
  - **Validation**: Must be a PDA of the current program derived form the `minter_authority`, `mint_account`, and `to_address`.
  - **Impact**: The whitelisted address entry account that is being deleted.

Branches and code coverage (including function calls)

Intended branches

- Deletes the whitelist address entry given.
  - ☑ Test coverage

Negative behavior

- Reverts if the `admin` is not a signer.
  - ☑ Negative test
- Reverts if the `payer` is not a signer.
  - ☐ Negative test
- Reverts if the `admin` does not match the `minter`.
  - ☐ Negative test
- Reverts if the `minter_authority` does not match the minter.
  - ☐ Negative test
- Reverts if the `mint_account` does not match the minter.
  - ☐ Negative test
- Reverts if the `whitelisted_address` account does not match one of the other given parameters.
  - ☑ Negative test

Function: `start_admin_transfer(ctx, pending_admin)`

This starts the admin transfer by assigning a pending admin to the minter.

Inputs

- `payer`
  - **Validation**: Must be a signer.
  - **Impact**: Unused.
- `admin`
  - **Validation**: Must be a signer and match the `minter`.
  - **Impact**: Used to check if the caller is authorized to call this function.

- `minter_authority`
  - **Validation**: Must match the `minter`.
  - **Impact**: Used to derive the PDA address of the `minter`.
- `mint_account`
  - **Validation**: Must be an SPL `Mint`.
  - **Impact**: Used to derive the PDA address of the `minter`.
- `minter`
  - **Validation**: Must be a PDA of the current program matching the `admin`, `minter_authority`, and `mint_account`.
  - **Impact**: This is the `minter` whose admin is being transferred.
- `pending_admin`
  - **Validation**: None.
  - **Impact**: This is the new pending admin who can now claim admin by calling `accept_admin_transfer`.

Branches and code coverage (including function calls)

Intended branches

- Sets the pending admin to the given value.
  - ☑ Test coverage

Negative behavior

- Reverts if the `admin` is not a signer.
  - ☑ Negative test
- Reverts if the `admin` does not match the `minter`.
  - ☑ Negative test
- Reverts if the `minter_authority` does not match the `minter`.
  - ☐ Negative test
- Reverts if the `mint_account` does not match the `minter`.
  - ☐ Negative test

Function: `update_rate_limit(ctx, capacity, refill_per_second)`

This updates the rate limit to the new parameters. Must be signed by the admin key.

Inputs

- `payer`
  - **Validation**: Must be a signer.
  - **Impact**: Unused.
- `admin`

    - **Validation**: Must be a signer and match the `minter`.
    - **Impact**: Used to check if the caller is authorized to call this function.
- `minter_authority`
    - **Validation**: Must match the `minter`.
    - **Impact**: Used to derive the PDA address of the `minter`.
- `mint_account`
    - **Validation**: Must be an SPL `Mint`.
    - **Impact**: Used to derive the PDA address of the `minter`.
- `minter`
    - **Validation**: Must be a PDA of the current program matching the `admin`, `minter_authority`, and `mint_account`.
    - **Impact**: This is the `minter` whose rate limit parameters are updated.
- `capacity`
    - **Validation**: None.
    - **Impact**: The new rate-limit quota capacity.
- `refill_per_second`
    - **Validation**: None.
    - **Impact**: The new rate-limit quota recovery rate.

Branches and code coverage (including function calls)

Intended branches

- Sets the `capacity` and `refill_per_second` to their intended values.
    - ☑ Test coverage

Negative behavior

- Reverts if the `admin` is not a signer.
    - ☑ Negative test
- Reverts if the `admin` does not match the `minter`.
    - ☑ Negative test
- Reverts if the `minter_authority` does not match the `minter`.
    - ☐ Negative test
- Reverts if the `mint_account` does not match the `minter`.
    - ☐ Negative test

5. Assessment Results

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

During our assessment on the scoped Mint Controller programs, we discovered three findings, all of which were 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.