Zellic

Prepared for
Javier
Layer N

Prepared by
Frank Bachman
Bryce Casaje
Zellic

April 9, 2025

N1 Bridge

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 Layer N from March 20th, 2025 to March 25th, 2025. During this engagement, Zellic reviewed N1 Bridge's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

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

- Are there any vulnerabilities that could result in the loss of user funds?
- Are access controls implemented correctly to prevent unauthorized operations?
- Are there potential issues with the validation for proposed blocks?
- Does the custom Merkle implementation contain any bugs?

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.

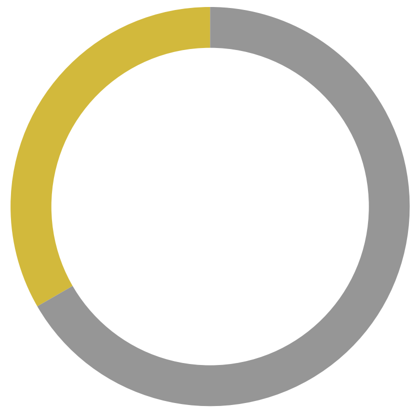
1.4. Results

During our assessment on the scoped N1 Bridge programs, we discovered three findings. No critical issues were found. One finding was of medium impact and the other findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	0
Informational	2



2. Introduction

2.1. About N1 Bridge

Layer N contributed the following description of N1 Bridge:

> A layer 1 blockchain designed for unlimited scale, featuring horizontal scalability, sub-ms latency, and congestion-free throughput.

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:

N1 Bridge Programs

Type	Rust
Platform	Solana

Target	nord
Repository	https://github.com/n1xyz/nord ↗
Version	`ed12f5dadaa555f401e7122085b8fecef90eddec`
Programs	`bridge`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of four person-days. 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 ↗

Bryce Casaje
Engineer
bryce@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 20, 2025	Kick-off call
March 20, 2025	Start of primary review period
March 25, 2025	End of primary review period

3. Detailed Findings

3.1. Insufficient block validation

Target	bridge		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

A block consists of the following fields:

```
pub struct Block {
    pub facts: BlockFacts,
    pub finalized: bool,
    pub slot_proposed: u64,
}

pub struct BlockFacts {
    pub prev_state_facts: StateFacts,
    pub next_state_facts: StateFacts,
    pub da_commitment: [u8; 32],
    pub withdrawal_root: [u8; 32],
}

pub struct StateFacts {
    pub app_state_commitment: [u8; 32],
    pub deposit_root: [u8; 32],
    pub last_deposit_index: u64,
    pub last_action_id: u64,
}
```

The `propose_block` instruction takes a `BlockFacts` struct as an argument to propose a new block to the bridge program, and it can only be run by the account set as the operator.

The operator is meant to be untrusted, so the implementation performs some validation on the block to assert that the block is correct.

It currently validates the following:

1. The block is proposed by the operator.
2. The data-availability (DA) fact is finalized.
3. The last deposit (if supplied) hashes to the provided deposit root in the next state facts.

And when a block is finalized via the `finalize_block` instruction, it performs some additional validation:

1. The previous state facts should match the current state facts.
2. The block has not been previously finalized.
3. The block has existed for enough time (based on `challenge_period_slots`).

However, the `last_deposit` account in the `propose_block` instruction does not have sufficient validation:

```
#[account(
    seeds = [DEPOSIT_SEED,
    &facts.next_state_facts.last_deposit_index.to_le_bytes()],
    bump,
)]
pub last_deposit: Option<Account<'info, Deposit>>,
```

Even if a last deposit does exist, it can still be set to `None` since the account is wrapped in an `Option`.

This messes up the validation for `last_deposit_index` and `deposit_root` for the next state facts:

```
match (
    facts.next_state_facts.last_deposit_index,
    facts.next_state_facts.deposit_root,
    &ctx.accounts.last_deposit,
) {
    (0, root, None) if root == [0; 32] => (),
    (_, root, Some(d)) if root == d.hash() => (),
    _ => return err!(BridgeError::InvalidDepositRoot),
};
```

A block can be proposed with a `last_deposit_index` set to zero and an empty `deposit_root` if `last_deposit` is set to `None`, even if a last deposit does exist.

In addition, there is no validation for the `withdrawal_root` field as well as the `app_state_commitment`, `last_action_id`, and `last_deposit_index` fields in the next state facts.

Impact

Invalid values for `app_state_commitment`, `last_action_id`, `last_deposit_index`, and `deposit_root` can cause errors in off-chain bridge infrastructure that trusts these values to be correct.

An invalid Merkle root for `withdrawal_root` would allow an attacker to verify any withdrawal proof and drain funds from the bridge. However, this would require a compromised bridge operator to propose and finalize a malicious block.

Recommendations

Implement validation for fields that can be validated on chain.

For example, the `last_deposit` account should never be `None` if there has been a previous deposit.

Remediation

The issue has been acknowledged by Layer N, and the team intends for most block validation to happen off chain via manual intervention or a challenge mechanism, as mentioned in Discussion point 4.1. ↗.

Some extra checks were added in `716bb92c` ↗, and the following comment was provided:

> A block is allowed to be proposed covering a smaller range of deposits than available at proposal time.
>
> Off-chain bridge infrastructure that should NOT trust these values to be correct as they belong to an unfinalized block.
>
> Furthermore, an invalid Merkle root for withdrawal_root would NOT allow an attacker to verify any withdrawal proof and drain funds from the bridge as that would require the block to be finalized.
>
> While finalization is intended for the operator, there is no trust assumption on the signer selecting valid blocks to finalize. There is however a (planned) monetary incentive on the proposer. The validation of a block prior to finalization is done by the contract through proposal and finalization-time checks AND our fraud-proof challenge mechanism (planned) / manual review during longer challenge-period + manual intervention (interim).

3.2. Associated Token Account check not enforced on withdrawal account

Target	bridge		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

It was found that in `Withdraw`, the `to_account` is not enforced to be an Associated Token Account (ATA).

```
#[derive(Accounts)]
#[instruction(claim: WithdrawalClaim)]
pub struct Withdraw<'info> {
[...]
    #[account(mut, constraint = to_account.owner == claim.user)]
    pub to_account: InterfaceAccount<'info, TokenAccount>,
```

The Solana Token Program allows users to arbitrarily create many token accounts belonging to the same mint. Note that anyone can create these accounts for a specified owner.

It is therefore possible for a caller to create a separate account (still controlled by the owner) to withdraw the funds.

Impact

The caller can withdraw funds to a non-ATA token account. This is not a security issue as the token account would still be controlled and managed by the owner.

Recommendations

Enforce `to_account` in `Withdraw` to be an ATA.

Remediation

Layer N acknowledged the issue, and decided not to apply a remediation with the following explanation:

This is intended behaviour as a flexibility of the api that we get “for free” by not enforcing ATA. The signer of the withdraw transaction is responsible for the destination account and this should be harmless as the owner is validated.

3.3. Usage of `Pubkey` rather than `Mint`

Target	bridge		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

In the `whitelist_asset` and `set_min_deposit` instructions, the asset to configure is supplied by the operator as a `Pubkey` in the instruction arguments:

```
#[instruction(asset: Pubkey, min_deposit: u64)]
pub struct WhitelistAsset<'info> {
    // ...
}
```

```
#[instruction(asset: Pubkey, min_deposit: u64)]
pub struct SetMinDeposit<'info> {
    // ...
}
```

The asset is intended to be a Solana Program Library (SPL) token mint, but this requirement is not set by the implementation.

Impact

The operator can whitelist an asset that is not an SPL token mint.

This does not cause any direct issues since the `deposit_spl` instruction derives the asset from the `mint` of a `TokenAccount`, but this should be fixed to prevent future security regressions.

Recommendations

The asset to configure should not be an instruction argument but instead a `Mint` account from the anchor-spl crate.

Remediation

This issue has been acknowledged by Layer N, and a fix was implemented in commit `ed392f7c` ↗.

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. Block validation

The provided version of the code has no implementation to challenge proposed blocks or manually intervene and stop malicious blocks from being finalized.

However, Layer N intends to implement these features in future editions of the code.

As such, the `propose_block` and `finalize_block` instructions do not need to completely validate all fields of proposed blocks. Nevertheless, the fields that can be validated on chain should be validated to help prevent malicious blocks.

4.2. Unused accounts

The `finalize_block` instruction includes a redundant `payer` account that

1. incurs no fees (since there is no account creation in the instruction), and
2. serves no access-control purpose (since the instruction is permissionless).

As such, the `payer` account can be removed to simplify the interface and reduce potential confusion about its purpose.

This issue has been acknowledged by Layer N, and a fix was implemented in commit `72f1cbed` ↗.

4.3. Test suite

The test suite provided covered most of the instructions implemented by the program.

However, some improvements could still be made, especially in regards to edge cases and negative scenarios, as the test suite mostly focuses on verifying that each instruction works as expected with the correct signers and arguments.

In addition, none of the tests in the test suite simulate that the program functions correctly with more than one block.

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.

Discriminant checks, ownership checks, and rent checks are not discussed for each individual account; unless otherwise stated, the program uses Anchor types, which perform the necessary checks automatically.

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: bridge

Instruction: `deposit_spl`

This instruction allows the depositor to deposit SPL tokens to the bridge.

Input parameters

- `amount: u64`: The amount of SPL tokens to deposit.

Accounts

- **`depositor`**: The depositor account that is the authority for the deposited tokens — pays for the new initialized accounts.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have sufficient lamports to cover the account-creation cost.
- **`deposit`**: The account to hold the new deposit data.
    - Signer: No.
    - Init: Yes.
    - PDA: Yes (derived from `DEPOSIT_SEED` and `contract_storage.last_deposit_index + 1` with bump).

- Mutable: Yes.
- Constraints: Must be a new account initialized with space `8 + 104` (discriminator + `Deposit` size).

- **`prev_deposit`**: The previous deposit data.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `DEPOSIT_SEED` and `contract_storage.last_deposit_index` with bump).
  - Mutable: No.
  - Optional: Yes.
  - Constraints: Must be of type `Deposit`.
- **`asset_config`**: The configuration data for this asset.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `ASSET_CONFIG_SEED` and `from_account.mint` with bump).
  - Mutable: No.
  - Constraints: Must be an `AssetConfig` where `min_deposit` is less than or equal to `amount`.
- **`contract_storage`**: The global contract storage.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `CONTRACT_STORAGE_SEED` with bump).
  - Mutable: Yes.
  - Constraints: Must be a `ContractStorage` account. If `last_deposit_index` is not `0`, a `prev_deposit` must be provided.
- **`from_account`**: The SPL token account where tokens are taken from.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must be a valid SPL token account.
- **`to_account`**: The ATA where tokens will be sent.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must be an ATA for the authority account, with the same mint as `from_account`.

Additional checks and behavior

- Transfers `amount` of the SPL token from `from_account` to `to_account` with the authority `depositor`.
- Creates and stores a `Deposit` where `transfer` holds the deposit information and `prev_deposit_root` is the hash of the previous deposit, if supplied.
- Increments `contract_storage.last_deposit_index`.
- Logs a `Deposit` event.

CPI

- `transfer`: Transfers the specified tokens from `from_account` to `to_account`, signed by `depositor`.

Instruction: `finalize_block`

This instruction finalizes a proposed block.

Input parameters

- `block_id: u64`: The ID for the block to finalize.

Accounts

- **`payer`**: The signer for this transaction.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: None.
- **`block`**: The block to be finalized.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `BLOCK_STORAGE_SEED` and `block_id` with bump).
  - Mutable: Yes.
  - Constraints: The block's `prev_state_facts` must match `contract_storage.fina_state_facts`, the block must not be finalized, and the current time slot should be greater than `block.slot_proposed + contract_storage.challenge_period_slots`.

- **`contract_storage`**: The global contract storage.
    - Signer: No.
    - Init: No.
    - PDA: Yes (derived from `CONTRACT_STORAGE_SEED` with bump).
    - Mutable: Yes.
    - Constraints: Must be a `ContractStorage` account.

Additional checks and behavior

- Sets `block.finalized` to `true`.
- Sets `contract_storage.fina_block_id` to `block_id`.
- Sets `contract_storage.fina_state_facts` to `block.facts.next_state_facts`.
- Logs a `FinalizeBlock` event.

CPI

N/A.

Instruction: `deposit_spl`

This instruction finalizes a DA fact.

Input parameters

- `fact: [u8; 32]`: The DA fact to finalize.

Accounts

- **`payer`**: The signer for this transaction who pays account-creation fees.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have sufficient lamports to cover the account-creation cost.
- **`fact_state_storage`**: The account to store the DA-fact finalization.
    - Signer: No.
    - Init: Yes.

    - PDA: Yes (derived from `DA_FACT_STORAGE_SEED` and `fact` with bump).
    - Mutable: Yes.
    - Constraints: Must be a new account initialized with space `8 + 1` (discriminator + `FactStateStorage` size).

Additional checks and behavior

- Sets `fact_state_storage` to a `FactStateStorage` struct with the state `FactState::Finalized`.
- Logs a `FinalizeDaFact` event.

CPI

N/A.

Instruction: `initialize`

This instruction initializes the global contract storage.

Input parameters

- `operator: Pubkey`: The public key for the bridge operator role.
- `challenge_period_slots: u64`: The number of slots where a block can be challenged before it is finalized.
- `app_state_commitment: [u8; 32]`: The initial app state commitment value.

Accounts

- **`payer`**: The signer for this transaction who pays account-creation fees.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have sufficient lamports to cover the account-creation cost.
- **`program`**: The bridge program.
    - Signer: Yes.
    - Init: No.
    - PDA: No.

- Mutable: No.
- Constraints: Must be an account with the address `crate::ID` (which should be the bridge program).

- **`contract_storage`**: The global contract storage.
  - Signer: No.
  - Init: Yes.
  - PDA: Yes (derived from `CONTRACT_STORAGE_SEED` with bump).
  - Mutable: Yes.
  - Constraints: Must be a new account initialized with space `8 + 144` (discriminator + `ContractStorage` size).

Additional checks and behavior

- Checks that `crate::ID` is not `[0; 32]`.
- Creates and stores a `ContractStorage` account with the values from the input parameters.

CPI

N/A.

Instruction: `propose_block`

This instruction allows the operator to propose a new block to the bridge.

Input parameters

- `facts: BlockFacts`: Information about the new block to be proposed.

Accounts

- **`operator`**: The operator account who pays for the new initialized accounts.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have the same public key as `contract_storage.operator` and have sufficient lamports to cover the account-creation cost.
- **`block`**: The account to hold the new proposed block data.

- Signer: No.
- Init: Yes.
- PDA: Yes (derived from `BLOCK_STORAGE_SEED` and `contract_storage.last_block_id + 1` with bump).
- Mutable: Yes.
- Constraints: Must be a new account initialized with space `8 + 233` (discriminator + `Block` size)

- **`last_deposit`**: The previous deposit data.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `DEPOSIT_SEED` and `facts.next_state_facts.last_deposit_index` with bump).
  - Mutable: No.
  - Optional: Yes.
  - Constraints: Must be of type `Deposit`.
- **`da_fact_state`**: The DA commitment account data.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `DA_FACT_STORAGE_SEED` and `facts.da_commitment` with bump).
  - Mutable: No.
  - Constraints: Must be `FactStateStorage` where `da_fact_state.state` is `FactState::Finalized`.
- **`contract_storage`**: The global contract storage.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `CONTRACT_STORAGE_SEED` with bump).
  - Mutable: Yes.
  - Constraints: Must be a `ContractStorage` account.

Additional checks and behavior

- Checks that `facts.next_state_facts.deposit_root` is valid by ensuring it is either all zeros when `facts.next_state_facts.last_deposit_index` is `0` and `last_deposit` is `None` or matches the hash of `last_deposit` when a deposit exists.
- Creates and stores a `Block` with the facts from the input parameter.
- Increments `contract_storage.last_block_id`.
- Logs a `ProposeBlock` event.

CPI

N/A.

Instruction: `set_min_deposit`

This instruction allows the operator to set the minimum deposit for an asset.

Input parameters

- `asset: Pubkey`: The public key for the asset to configure.
- `min_deposit: u64`: The minimum deposit required for this asset.

Accounts

- **`operator`**: The operator account that is setting the minimum deposit.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must have the same public key as `contract_storage.operator`.
- **`contract_storage`**: The global contract storage.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `CONTRACT_STORAGE_SEED` with bump).
  - Mutable: No.
  - Constraints: Must be a `ContractStorage` account.
- **`asset_config`**: The asset-configuration account matching the public key.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `ASSET_CONFIG_SEED` and `asset` with bump).
  - Mutable: Yes.
  - Constraints: The `min_deposit` input parameter must be greater than zero.

Additional checks and behavior

- Sets `asset_config.min_deposit` to the `min_deposit` from the input parameters.
- Logs a `SetMinDeposit` event.

CPI

N/A.

Instruction: `whitelist_asset`

This instruction allows the operator to whitelist an asset for depositing and withdrawing.

Input parameters

- `asset: Pubkey`: The public key for the asset to configure.
- `min_deposit: u64`: The minimum deposit required for this asset.

Accounts

- **`operator`**: The operator account that is whitelisting the asset.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must have the same public key as `contract_storage.operator` and enough lamports to pay for account-creation costs.
- **`contract_storage`**: The global contract storage.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `CONTRACT_STORAGE_SEED` with bump).
  - Mutable: No.
  - Constraints: Must be a `ContractStorage` account.
- **`asset_config`**: The asset-configuration account matching the public key.
  - Signer: No.
  - Init: Yes.
  - PDA: Yes (derived from `ASSET_CONFIG_SEED` and `asset` with bump).
  - Mutable: Yes.
  - Constraints: Must be a new account initialized with space `8 + 8` (discriminator + `AssetConfig` size), and the `min_deposit` input parameter must be greater than zero.

Additional checks and behavior

- Creates and stores a new `AssetConfig` with the `min_deposit` from the input parameters.
- Logs a `WhitelistAsset` event.

CPI

N/A.

Instruction: `withdraw`

This instruction allows a user to withdraw tokens from the bridge.

Input parameters

```
pub struct WithdrawalClaim {
    pub user: Pubkey,           // The user that withdraws.
    pub amount: u64,            // The amount of tokens to withdraw.
    pub block_id: u64,          // The block ID of the withdrawal.
    pub proof: Vec<[u8; 32]>,   // The Merkle proof of the withdraw event.
    pub leaf_index: LeafIdx,    // The index of the withdrawal in the Merkle
    tree.
    pub leaves_count: LeafIdx,  // The number of leaves in the Merkle tree.
}
```

Accounts

- **`payer`**: The account that will pay for the creation of the nullifier.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have enough lamports to pay for account-creation costs.
- **`state_update`**: The finalized block that contains the withdrawal event.
  - Signer: No.
  - Init: No.
  - PDA: Yes (derived from `BLOCK_STORAGE_SEED` and `claim.block_id` with bump).

    - Mutable: No.
    - Constraints: Must be a `Block` that is finalized.
- **`withdrawal_nullifier`**: The account to store the nullifier to ensure that a withdrawal cannot happen twice.
    - Signer: No.
    - Init: Yes.
    - PDA: Yes (derived from `WITHDRAWAL_NULLIFIER_SEED`, `claim.block_id`, and `claim.leaf_index` with bump).
    - Mutable: Yes.
    - Constraints: Must be a new account initialized with space `8` (discriminator).
- **`from_account`**: The bridge SPL token account where tokens are taken from.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must be an ATA for the authority account, with the same mint as `to_account`.
- **`to_account`**: The user SPL token account where tokens are sent to.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must be a valid SPL token account owned by `claim.user`.
- **`authority`**: The authority account that owns tokens held by the bridge.
    - Signer: No.
    - Init: No.
    - PDA: Yes (derived from `AUTHORITY_SEED`).
    - Mutable: No.
    - Constraints: None.

Additional checks and behavior

- Takes the `withdrawal_root` from the `state_update` block provided and verifies that the user's withdrawal exists in the root.
- Transfers `claim.amount` tokens from `from_account` to `to_account`.
- Logs a `Withdraw` event.

CPI

- `transfer`: Transfers the specified tokens from `from_account` to `to_account`, signed by `authority`.

Module: merkle

This module contains functions for constructing Merkle roots and verifying Merkle paths. It is a custom implementation based on IETF RFC 6962. It creates a skewed Merkle tree. The tree is constructed by first selecting the largest subset of leaves that can form a complete binary Merkle tree and obtaining their root. To compute the overall Merkle root, the root of this subset is treated as a left sibling. The right sibling is computed by repeating this process recursively on the remaining leaves. It is implemented with in-place transforms.

While hashing leaves and intermediate nodes, the preimage is prefixed with `LEAF_TAG` and `PAIR_TAG` respectively to prevent second preimage attacks.

Function: `proof_path`

This function computes a minimal Merkle proof of inclusion of a leaf at index `leaf_idx`.

Inputs

- `leaf_idx`: The index of the leaf for which the path is to be derived.
- `leaves`: The list of all leaves in the Merkle tree.
- `digest_cb`: The function encapsulating the logic for hashing a leaf struct.

Function: `root_from_proof`

This function computes the Merkle root from a proof of inclusion of a leaf at index `leaf_idx`.

Inputs

- `leaf`: The leaf for which the proof of inclusion is provided.
- `leaf_idx`: The index of the proven leaf.
- `leaves_len`: The total number of leaves — used to calculate parity for all steps in Merkle proof verification.
- `proof`: The Merkle proof of inclusion of the leaf.
- `digest_cb`: The function encapsulating the logic for hashing a leaf struct.

Function: `root_from_leaves`

This function computes the Merkle root for a given list of leaves.

Inputs

- `leaves`: The list of all leaves in the Merkle tree.
- `digest_cb`: The function encapsulating the logic for hashing a leaf struct.

6. Assessment Results

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

During our assessment on the scoped N1 Bridge programs, we discovered three findings. No critical issues were found. One finding was of medium impact and the other findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.