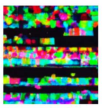

ZKSECURITY

Audit of Aleo ARC-0041

June 10th, 2024

On June 10th, 2024, zkSecurity spent four days reviewing the ARC-0041 proposal, including a full review of `credits.aleo`.

No serious issues were found, and the proposal as well as the code changes were deemed well-documented and organized. We did not find any security vulnerabilities or issues with the implementation of `credits.aleo` either.

In the rest of this document we describe the scope and give an overview of the proposal and logic of `credits.aleo`.

Scope

The audit included the official ARC document, and implementations of the changes:

- The `credits.aleo` program in `synthesizer/program/src/resources/credits.aleo`
- changes related to ARC-31 contained in the finalize logic (`synthesizer/src/vm/finalize.rs`) and some helper methods:
    - rewards, (`synthesizer/src/vm/helpers/rewards.rs`)
    - committee, (`synthesizer/src/vm/helpers/committee.rs`)

The main branch of AleoNet/snarkVM was used, specifically the commit `0f382b6a2b553385eb53f3cb6869f2bc1039feb6`.

Overview of ARC-41

ARC-41 suggests modifying the process used to activate validators, specifically how they can enter and exit the committee.

The change allows external delegators to contribute the 10M credits required to become a validator (instead of a validator doing so themselves), effectively derisking the operation of a validator and decorrelating a validator key from its stake.

The validator now has to contribute a symbolic 100 credits to join or leave the committee.

The ARC-41 proposal comes in line with ARC-37, which previously allowed bonded funds to be withdrawn to a different address when unbonding. Both proposals now completely derisk a validator's private key, only tying it to 100 credits at a minimum.

Overview of `credits.aleo`

The `credits.aleo` file is the main Aleo program (written in Aleo instructions) that encodes how the token and staking system works. This section gives a brief overview of its mechanism.

The state of the `credits.aleo` program is stored in a number of *mappings* which can be seen as key-value stores.

Credits. At its core, the notion of a "credit" is the first primitive object created in the system, which represents Aleo's native token:

```
// The `credits` record is used to store credits privately.
record credits:
    // The address of the owner.
    owner as address.private;
    // The amount of private microcredits that belong to the specified owner.
    microcredits as u64.private;
```

There exists a few functions to manipulate credits privately, which assume (and it is enforced externally) that the caller owns any record being passed as argument:

- `transfer_private(record, recipient, amount) -> (record, record)`: splits a record into two new records: one for the recipient (with the given amount), and one for the original owner of the record (with the change)
- `split(record, amount) -> (record, record)`: splits a record into two new records: one with the given amount, one with the change (minus a fee of 10 millicredits for increasing the size of the record pool)
- `join(record, record) -> record`: consolidates two records
- `fee_private(record, pub amount, pub priority_amount, pub exec_id) -> record`: removes two fees from a record, and exposes these fees

Public credits. Credits can also be stored in a public database of accounts, allowing public operations (like staking):

```
// The `account` mapping is used to store credits publicly.
mapping account:
    // The key represents the address of the owner.
    key as address.public;
```

```
    // The value represents the amount of public microcredits that belong to the
specified owner.
    value as u64.public;
```

a number of functions are defined to operate on these public credits directly:

- `transfer_public(recipient, amount)`: performs a transfer between the sender and the recipient accounts
- `transfer_public_as_signer`: same as `transfer_public` but uses `self.signer` instead of `self.account` (we touches on what this means later in this section)
- `fee_public(amount, priority_amount, exec_id)`: same as `fee_private` but acts on the public balance of the `self.signer` account instead of private records

Transition from private to public. While credits can remain private by staying in the shielded pool, they can also be moved to the public database of accounts (or vice-versa) via the two functions: `transfer_private_to_public(record, recipient, amount) -> record` and `transfer_public_to_private(recipient, amount) -> record`.

Validators. A validator is first and foremost a participant in consensus. Normal addresses can become an active validator by having enough stake delegated to them. We differentiate between an inactive validator and an active validators:

- *inactive validator*: an address that is delegated to by other addresses or the address itself, but is not part of the *committee* of validators
- *active validator*: an address that is part of the committee of validators due to having >=10M credits delegated to it, including at least 100 credits by the address itself.

A validator can be represented by the tuple of keypairs `(address, withdraw_address)` which are enforced to be different, and where different keypairs have different capabilities:

- the `address` keypair, associated to the main address, can self-delegate (via `bond_validator`) without increasing the number of delegators
- the `withdraw_address` keypair (set by the first call to `bond_validator`) can can unbond arbitrary delegators to the `address` (via `unbond_public`)

A validator can join the committee of active validators by waiting for enough credits to be delegated to them, and then calling `bond_validator` with at least 100 credits, so that the delegated credits added to these 100 credits total to 10 million credits.

By doing that, they can also set their *commission*, which is the percentage of any block rewards that the validator can claim for themselves.

Bonding. Anyone can delegate to anyone by *bonding* their funds (via a call to `bond_public`). That being said, you can only bond to a single validator at a time, and to bond to someone else you must first *fully* empty your bond by calling `unbond_public`. You also can't delegate to a validator that is not open to new delegators (unless it's you or you have already bonded to them previously) or that is in the process of unbonding (even if it's you). The minimum amount to bond is 1 credit (via `bond_public`), but you must at least start with 10k credits.

Unbonding. Recovering bonded funds can be done by calling `unbond_public`. This action removes the funds instantly from the delegated funds of a validator, potentially removing them from the committee if they go below the threshold of 10M credits required. Note that a user that unbonds does not recover the funds instantly, as they stay locked for a certain period of time (i.e. 360 blocks) before the user can finally claim them (via the `claim_unbond_public` function).

We summarize the validator flow in the following diagram:

Withdraw address. Added in ARC-37, withdraw addresses decouple the owner of a bond from the address that can withdraw the bond. As such, unbonding is done by the associated withdraw address. In addition, the withdraw address of a validator (as explained above) can unbond arbitrary delegators to the validator.

As such there are three cases to the unbonding flow:

- a staker's associated withdraw address unbonds, potentially removing the delegated validator from the committee
- a validator unbonds all of the delegated funds from one of its delegators, potentially removing themselves from the committee
- a validator unbonds themselves, in which case they get removed from the committee

Note on `self.signer` vs `self.caller`. Some contracts use different features of Aleo to authorize diverse operations. In general, `self.caller` allows an Aleo program (like a wallet as a smart contract) to control an account, whereas `self.signer` forces a specific user to author a command.

For example:

- `self.caller` is used in `transfer_public_to_private`, `bond_public`, `unbond_public`, `set_validator_state`
- `self.signer` is used in `bond_validator`, `fee_public`

List of State Transitions, Events, and Invariants

To facilitate analysis we list all the allowed state transition encoded in the `credits.aleo` program, as well as events that it could emit:

`bond_public(validator, withdraw_address, amount)`:

- emits a `BOND(staker, validator, amount)` event
- increase the number of delegators if `self.caller` was not bonded before

`bond_validator(withdraw_address, amount_commission)`:

- emits a `VALIDATOR_JOINS(validator)` event if the requirements are met and the validator was not already in the committee

`unbond_public(staker_address, amount)`:

- emits a `UNBOND_EVERYTHING(staker)` event if all of the bonded funds under that staker_address are removed
- adds `amount` to the unbonded funds for the `staker_address` associated withdraw address.
- emits a `VALIDATOR_LEAVES(validator)` with the validator delegated to, in case the validator does not meet the requirements to be in the committee anymore.

`claim_unbond_public(staker_address)`:

- emits a `CLAIMED_UNBOND_EVERYTHING(staker, address)` event if withdrawaddress is removed / if everything bonded has been emptied out + claimed

`set_validator_state(state)`:

- emits a `VALIDATOR_STATE(validator, state)` event

Given these state transitions and events, one can try to manually or formally verify a number of statements about the program. Here are some examples of properties that could be verified (including properties that were formulated as part of the documentation of `credits.aleo`):

Delegation.

- You should not be able to delegate to a validator that is closed, unless you've already delegated to them.
- You should not be able to delegate to a validator that is unbonding.
- You should not be able to bond to two different validators at the same time (or in other word, there must be an `UNBOND_EVERYTHING(a)` event between two different `BOND(a, v1, _)` and `BOND(a, v2, _)` events).
- You can only change withdraw address once you've unbonded everything.
- The amount delegated to a validator is the sum of the bonds to that validator.
- You should not be able to bond less than 10k credits (except if you're a validator, in which case you can bond 100 credits to yourself)
- A bond is always greater than 100.
- The delegated amount to any validator is always greater or equal to 10,000.

Committee participation.

- A validator that is in the committee cannot be unbonding.
- A validator who is part of the committee has at least 10M credits delegated to them, including 100 credits personally bonded.
- The set of delegated addresses is a superset of the validators in the committee.
- A self-bonded address is always part of the committee.
- A bond of less than 10,000 is a self-bond.
- The total delegated amount for an active validator is at least 10,000,000 credits.

Credits and global values.

- Credits cannot be lost or created out of thin air (conservation of value): the total amount of credits in circulation is the total listed in the accounts map, plus the total listed in the delegated map, plus the total listed in the unbond map(, plus the total amount of private credit records).
- `metadata[NUM_DELEGATORS]` should always represent the number of delegators, which is the length of the bonded map minus the length of the committee map.
- `metadata[COMMITTEE_SIZE]` should always represent the size of the committee, which is the length of the committee map..
- There can never be more than 100,000 delegators at the same time (not including self-bonds).
- The sum of the delegated map is equal to the sum of the bonded map.

Calculation of Rewards

We illustrate how block rewards are calculated for the validator and its delegators:

Potential Issues and Strategic Recommendations

ARC-41 is not up to date. We recommend updating the ARC-41 proposal with the actual changes made in the code since then to more accurately reflect the update.

Finalize logic. We looked at the changes related to the ARC-41 proposal, which were mostly contained in #2453. In `synthesizer/src/vm/finalize.rs` the logic fails to retrieve the proper validator addresses as it collects the first input of the private part of the `bond_validator` function rather than the *finalize* function:

```
fn prepare_for_execution(store: &FinalizeStore<N, C::FinalizeStorage>, execution:
&Execution<N>) -> Result<()> {
  // Construct the program ID.
  let program_id = ProgramID::from_str("credits.aleo")?;
  // Construct the committee mapping name.
  let committee_mapping = Identifier::from_str("committee")?;

  // Check if the execution has any `bond_validator` transitions, and collect
  // the unique validator addresses if so.
  // Note: This does not dedup for existing and new validator addresses.
  let bond_validator_addresses: HashSet<_> = execution
    .transitions()
    .filter_map(|transition| match transition.is_bond_validator() {
      // Check the first input of the transition for the validator address.
      true => match transition.inputs().first() {
        Some(Input::Public(_, Some(Plaintext::Literal(Literal::Address(address),
_)))) => Some(address),
        _ => None,
      },
      false => None,
    })
    .collect();
```

In addition, the code can in theory fail open by returning `None` if there is a further issue with the inputs of the transition.

Complexity of `unbond_public`. Most of the complexity (and thus opportunities for bugs) in `credits.aleo` comes from the `unbond_public` function which aggregates many flows (a validator unbonds a delegator, or themselves, or a delegator unbond themselves) and relies on a number of implied assumptions. While we did not find any issues with the logic, it could be a good idea to split the function into smaller functions to make it easier to reason about.