MONO CEROS ALPHA

PROJECT

Elrond Launchpad

CLIENT

Elrond

DATE

October 2021

REVIEWERS

Daniel Luca

@cleanunicorn

Andrei Simion

@andreiashu

Table of Contents

Details

- **Client** Elrond
- **Date** October 2021
- **Lead reviewer** Daniel Luca (@cleanunicorn)
- **Reviewers** Daniel Luca (@cleanunicorn), Andrei Simion (@andreiashu)
- **Repository**: Elrond Launchpad
- **Commit hash** `f0bf99cafcee468a0e4c3aaa74f22df04f1c82b3`
- **Technologies**
    - Rust

Issues Summary

SEVERITY	OPEN	CLOSED
Informational	2	0
Minor	3	0
Medium	1	1
Major	0	1

Executive summary

This report represents the results of the engagement with **Elrond** to review **Elrond Launchpad**.

The review was conducted over the course of **2 weeks** from **October 4th to October 15th, 2021**. A total of **20 person-days** were spent reviewing the code.

Week 1

During the first week, we familiarized ourselves with the code and the project. Then, we reviewed the code from the beginning to the end of the week.

We set up a few meetings throughout the week to discuss the code and learn how to navigate the codebase. We also discussed the project goals and the project scope.

Week 2

We continued to keep communication open with the development team while navigating the code and trying out different attack vectors.

We started to focus more on how the `Stage` selection is determined, paying particular attention to how the `get_launch_stage` function can be made to perform an invalid state transition.

We discovered critical and medium severity issues which were fixed in a further pull request provided by the Elrond Team.

We spent the rest of the week focusing on reviewing the above-mentioned pull request changes to ensure no further issues were introduced.

Scope

The initial review focused on the Elrond Launchpad repository, identified by the commit hash `f0bf99cafcee468a0e4c3aaa74f22df04f1c82b3`.

We merged fixes from branch `fixes-after-audit` at commit hash `17810ee9957bf95d42fade8ac7e73267fa7490b1`.

Includes:

- code/launchpad/src/random.rs
- code/launchpad/src/ticket_status.rs
- code/launchpad/src/ongoing_operation.rs
- code/launchpad/src/launchpad.rs
- code/launchpad/src/launch_stage.rs
- code/launchpad/src/setup.rs

Issues

The owner can claim all ticket payments without giving away any rewards

Status Fixed Severity Major

Description

The owner has full control over the user's funds and the prizes. The current design does not provide security for the participating users to receive the tickets or be refunded.

The owner can wait for all the users to claim tickets, sending the payment to the contract.

code/launchpad/src/launchpad.rs#L170-L175

```
    #[payable("*")]
    #[endpoint(confirmTickets)]
    fn confirm_tickets(
        &self,
        #[payment_token] payment_token: TokenIdentifier,
        #[payment_amount] payment_amount: Self::BigUint,
```

code/launchpad/src/launchpad.rs#L186-L194

```
        let ticket_payment_token = self.ticket_payment_token().get();
        let ticket_price = self.ticket_price().get();
        let total_ticket_price = Self::BigUint::from(nr_tickets_to_confirm) * ticket_price;

        require!(
            payment_token == ticket_payment_token,
            "Wrong payment token used"
```

```
        );
        require!(payment_amount == total_ticket_price, "Wrong amount sent");
```

After the tickets were confirmed, the funds are in the contract.

Next, the owner can withdraw all funds from the contract.

code/launchpad/src/launchpad.rs#L29-L42

```
    #[only_owner]
    #[endpoint(claimTicketPayment)]
    fn claim_ticket_payment(&self) -> SCResult<()> {
        let ticket_payment_token = self.ticket_payment_token().get();
        let sc_balance = self.blockchain().get_sc_balance(&ticket_payment_token, 0);
        let owner = self.blockchain().get_caller();

        if sc_balance > 0 {
            self.send()
                .direct(&owner, &ticket_payment_token, 0, &sc_balance, &[]);
        }

        Ok(())
    }
```

The owner can blacklist the winners.

code/launchpad/src/launchpad.rs#L53-L59

```
    #[only_owner]
    #[endpoint(addAddressToBlacklist)]
    fn add_address_to_blacklist(&self, address: Address) -> SCResult<()> {
        self.blacklist().insert(address);

        Ok(())
    }
```

Which blocks them from receiving any won rewards.

code/launchpad/src/launchpad.rs#L292-L300

```
    #[endpoint(claimLaunchpadTokens)]
    fn claim_launchpad_tokens(&self) -> SCResult<()> {
        self.require_stage(LaunchStage::Claim)?;

        let caller = self.blockchain().get_caller();
        require!(
            !self.blacklist().contains(&caller),
            "You have been put into the blacklist and may not claim tokens"
        );
```

Thus, the owner retrieved all ticket prices and also blocks the users from receiving their rewards.

Recommendation

Before we discussed and agreed on a recommendation, a good fix was already published.

Contract owner can inadvertently move the stage from `Claim` back to `SelectNewWinners`

Status Fixed Severity Medium

Description

In `get_launch_stage` function, the state is implicitly determined based on a set of variables, some that are dynamic.

In one edge case, when the *Stage* of the contract is at `LaunchStage::Claim`, the contract owner, inadvertently, can move the stage back to `LaunchStage::SelectNewWinners` by calling `refundConfirmedTickets` function to refund a ticket.

Example scenario:

1. `total_confirmed_tickets == total_winning_tickets` and we're past the epoch when the `claim_start_epoch` was set. Therefore `get_launch_stage` will return `LaunchStage::Claim`:

code/launchpad/src/launchpad.rs#L385-L389

```
        if total_confirmed_tickets >= total_winning_tickets {
            let claim_start_epoch = self.claim_start_epoch().get();
            if current_epoch >= claim_start_epoch {
                return LaunchStage::Claim;
            } else {
```

2. The owner calls the `refundConfirmedTickets` function to refund a ticket of an already blacklisted address. This will decrement the `total_confirmed_tickets` value by the number of refunded tickets:

code/launchpad/src/launchpad.rs#L91-L92

```
        self.total_confirmed_tickets()
            .update(|confirmed| *confirmed -= nr_refunded_tickets);
```

3. Now `get_launch_stage` will return `LaunchStage::SelectNewWinners` since the condition `total_confirmed_tickets <= total_winning_tickets` evaluates to `false` and all the other conditions in the function are not met:

code/launchpad/src/launchpad.rs#L413-L416

```
        }

        LaunchStage::SelectNewWinners
    }
```

This is a problem because once the stage reaches `Claim` it should not change back to another stage. Additionally, for the owner of the contract, this transition is not at all obvious and causes issues for the users trying to claim their tickets.

The randomness algorithm implements Sattolo's algorithm instead of Fisher-Yates

Status Open Severity Medium

Description

The randomness function is called once to shuffle the tickets.

code/launchpad/src/launchpad.rs#L123-L125

```
    #[endpoint(selectWinners)]
    fn select_winners(&self) -> SCResult<BoxedBytes> {
        self.require_stage(LaunchStage::SelectWinners)?;
```

This method needs to loop over each position, select a random ticket and put the ticket at that position. The method `shuffle_single_ticket` is called to pick a random ticket and place it in the current position.

code/launchpad/src/launchpad.rs#L136-L143

```
        let run_result = self.run_while_it_has_gas(|| {
            let is_winning_ticket = ticket_position <= nr_winning_tickets;
            self.shuffle_single_ticket(
                &mut rng,
                ticket_position,
                last_ticket_position,
                is_winning_ticket,
            );
```

After each swap, the position is incremented by 1.

code/launchpad/src/launchpad.rs#L136-L143

```
        let run_result = self.run_while_it_has_gas(|| {
            let is_winning_ticket = ticket_position <= nr_winning_tickets;
            self.shuffle_single_ticket(
                &mut rng,
                ticket_position,
                last_ticket_position,
                is_winning_ticket,
            );
```

The method `shuffle_single_ticket` picks a random ticket and places it in the current position (`current_ticket_position`).

code/launchpad/src/launchpad.rs#L438-L446

```
    /// Fisher-Yates algorithm,
    /// each position is swapped with a random one that's after it.
    fn shuffle_single_ticket(
        &self,
        rng: &mut Random<Self::CryptoApi>,
        current_ticket_position: usize,
        last_ticket_position: usize,
        is_winning_ticket: bool,
    ) {
```

This part selects a number between `current_ticket_position + 1` and `last_ticket_position` inclusive.

code/launchpad/src/launchpad.rs#L447-L448

```
        let rand_index =
            rng.next_usize_in_range(current_ticket_position + 1, last_ticket_position + 1);
```

The Fisher-Yates algorithm states that you should pick a number between the current position and the last position. The current position should be included in the selection range.

Wikipedia describes a variation of this algorithm, named Sattolo's algorithm.

> A very similar algorithm was published in 1986 by Sandra Sattolo for generating uniformly distributed cycles of (maximal) length n.[6][7] The only difference between Durstenfeld's and Sattolo's algorithms is that in the latter, in step 2 above, the random number j is chosen from the range between 1 and i–1 (rather than between 1 and i) inclusive. This simple change modifies the algorithm so that the resulting permutation always consists of a single cycle.

The current code actually implements the Sattolo's algorithm because it selects items to swap from the `current_ticket_position + 1` instead of `current_ticket_position`.

The Wikipedia page warns this can be easy to accidentally implement.

> In fact, as described below, it is quite easy to accidentally implement Sattolo's algorithm when the ordinary Fisher–Yates shuffle is intended. This will bias the results by causing the permutations to be picked from the smaller set of (n−1)! cycles of length N, instead of from the full set of all n! possible permutations.

Technically the current code version implements Sattolo's algorithm instead of Fisher-Yates which is mentioned in the comments.

code/launchpad/src/launchpad.rs#L438-L440

```
    /// Fisher-Yates algorithm,
    /// each position is swapped with a random one that's after it.
    fn shuffle_single_ticket(
```

Recommendation

Change the selection interval to include the `current_ticket_position` when selecting a random number.

This code should use `current_ticket_position` instead of `current_ticket_position + 1`.

code/launchpad/src/launchpad.rs#L447-L448

```
        let rand_index =
            rng.next_usize_in_range(current_ticket_position + 1, last_ticket_position + 1);
```

References

- Fisher-Yates shuffle
- Sattolo's algorithm

`force_claim_period_start` can put the contract in an invalid state

Status Open Severity Minor

Description

`force_claim_period_start` is an owner_only operated function that allows the owner of the contract to move the stage into `Claim`:

code/launchpad/src/launchpad.rs#L44-L50

```
#[only_owner]
#[endpoint(forceClaimPeriodStart)]
fn force_claim_period_start(&self) -> SCResult<()> {
    let total_winning_tickets = self.nr_winning_tickets().get();
    self.total_confirmed_tickets().set(&total_winning_tickets);

    Ok(())
```

The issue is that this function should fail when the owner, mistakenly, calls it and creates a stage transition that is erroneous.

For example, when in any stage before `ConfirmTickets`, this function should fail: moving to `Claim` stage without allowing users to confirm their tickets does not make sense.

Recommendation

The function should check if the stage transition is a valid one before allowing an owner to perform it.

Method `TicketStatus.is_confirmed` can be simplified

Status Open Severity Minor

Description

The `TicketStatus` implements method `is_confirmed`.

code/launchpad/src/ticket_status.rs#L22-L34

```
// Pass Option::None to ignore generation
pub fn is_confirmed(&self, opt_current_generation: Option<u8>) -> bool {
    match *self {
        TicketStatus::Confirmed { generation } => {
            if let Some(current_generation) = opt_current_generation {
                generation == current_generation
            } else {
                true
            }
        }
        _ => false,
    }
}
```

This method is used in 2 cases:

code/launchpad/src/launchpad.rs#L82

```
            if !ticket_status.is_confirmed(None) {
```

code/launchpad/src/launchpad.rs#L307

```
            if !ticket_status.is_confirmed(None) {
```

In both cases the argument is `None`. The additional complexity in the method is never used, thus the method can be simplified.

Recommendation

Simplify the method if the additional complexity is never used.

Unused method `Ticket.is_winning`

Status Open Severity Minor

Description

The `TicketStatus` implements the method

code/launchpad/src/ticket_status.rs#L12-L20

```
    pub fn is_winning(&self, current_generation: u8) -> bool {
        if let TicketStatus::Winning { generation } = *self {
            if generation == current_generation {
                return true;
            }
        }

        false
    }
```

However, this method is not used anywhere.

Recommendation

Remove unused code.

Code typo `confiration_period_end_epoch`

Status Open Severity Informational

Description

code/launchpad/src/launchpad.rs#L425-L430

```
        let confiration_period_end_epoch =
            confirmation_period_start_epoch + confirmation_period_in_epochs;
        if current_epoch < confirmation_period_start_epoch {
            return LaunchStage::WaitBeforeConfirmation;
        }
        if current_epoch < confiration_period_end_epoch {
```

Implicit vs. Explicit state resolution

Status Open Severity Informational

Description

The Elrond Launchpad contract *LaunchStage* selection relies on the `get_launch_stage` function which returns the current stage:

code/launchpad/src/launchpad.rs#L379-L380

```
    #[view(getLaunchStage)]
    fn get_launch_stage(&self) -> LaunchStage {
```

This function is used to restrict actions only to specific stages:

code/launchpad/src/launchpad.rs#L292-L294

```
    #[endpoint(claimLaunchpadTokens)]
    fn claim_launchpad_tokens(&self) -> SCResult<()> {
        self.require_stage(LaunchStage::Claim)?;
```

The logic determines the current stage dynamically, based on the value of multiple state variables:

- `self.nr_winning_tickets()`
- `self.total_confirmed_tickets()`
- `self.claim_start_epoch()`
- `self.winner_selection_start_epoch()`
- `self.current_generation()`
- `self.confirmation_period_start_epoch()`
- `self.confirmation_period_in_epochs()`

Some of these variables can be changed by the owner at will (with some limitations).

The issue is that the logic that determines the current stage makes use of dynamic variables that can be and are changed during the lifecycle of this contract:

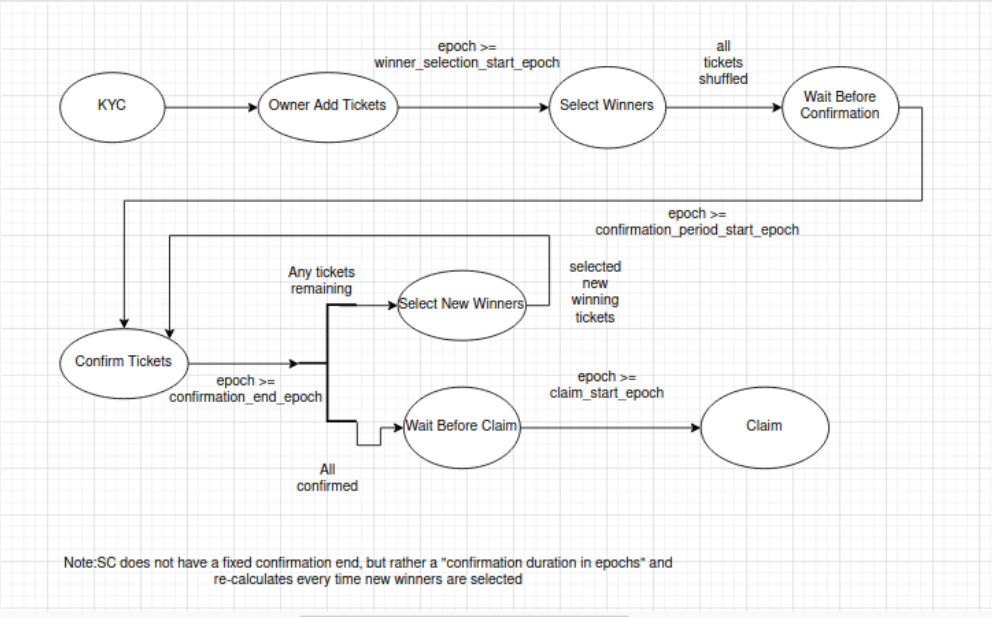
code/launchpad/src/launchpad.rs#L383-L397

```
        let total_winning_tickets = self.nr_winning_tickets().get();
        let total_confirmed_tickets = self.get_total_confirmed_tickets();
        if total_confirmed_tickets >= total_winning_tickets {
            let claim_start_epoch = self.claim_start_epoch().get();
            if current_epoch >= claim_start_epoch {
                return LaunchStage::Claim;
            } else {
                return LaunchStage::WaitBeforeClaim;
            }
        }

        let winner_selection_start_epoch = self.winner_selection_start_epoch().get();
        if current_epoch < winner_selection_start_epoch {
            return LaunchStage::AddTickets;
        }
```

Compared to an explicit logic whereby the owner can manage the stage transitions manually (and the contract code validates that each transition is valid), the current version of the code is more prone to errors. An invalid state transition (see #5) was already found but was difficult to spot because of the way the `get_launch_stage` function uses dynamic variables that can be changed in other parts of the contract.

The documentation features a state diagram that outlines the transition flow of stages:



Below is an example of how this flow chart does not necessarily represent what the code can do:

1. Current stage is `ConfirmTickets` (ie. `confirmation_period_start_epoch < current_epoch < confiration_period_end_epoch`)
2. Owner updates the following state variables so that `current_epoch < winner_selection_start_epoch`: `winner_selection_start_epoch`, `confirmation_period_start_epoch`

3. Current stage is `AddTickets` - the code allowed the owner to perform a transition from ConfirmTickets to AddTickets which is not reflected in the flowchart
4. Further on, this invalid transition can pose a risk since adding more tickets in its current state might expose other issues

Recommendation

We prefer a more explicit way of managing the state. If resources allow we recommend rewriting the `get_launch_stage` function to allow the owner to set the current stage and only do sanity checks to ensure that the state transition is valid.

License