Protocol Audit Report

Version 1.0

Akshat

September 21, 2024

Protocol Audit Report

Akshat

August 26 , 2024

Prepared by: Akshat

Table of Contents

Protocol Summary

This repository is the Staking contract for the Fjord ecosystem. Users who gets some ERC20 emitted by Fjord Foundry can stake them to get rewards.

Disclaimer

Akshat makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Scope

All Contracts in `src` are in scope.

```
src/
#-- FjordAuction.sol
#-- FjordAuctionFactory.sol
#-- FjordPoints.sol
#-- FjordStaking.sol
#-- FjordToken.sol
#-- interfaces
    #-- IFjordPoints.sol
```

Roles

- **AuthorizedSender**: Address of the owner whose cancellable Sablier streams will be accepted.
- **Buyer**: User who aquire some ERC20 FJO token.
- **Vested Buyer**: User who get some ERC721 vested FJO on Sablier created by Fjord.

- **FJO-Staker**: Buyer who staked his FJO token on the Fjord Staking contract.
- **vFJO-Staker**: Vested Buyer who staked his vested FJO on Sablier created by Fjord, on the Fjord Staking contract.
- **Penalised Staker**: a Staker that claim rewards before 3 epochs or 21 days.
- **Rewarded Staker**: Any kind of Stakers who got rewarded with Fjord's reward or with ERC20 BJB.
- **Auction Creator**: Only the owner of the AuctionFactory contract can create an auction and offer a valid project token earn by a "Fjord LBP event" as an auctionToken to bid on.
- **Bidder**: Any Rewarded Staker that bid his BJB token inside a Fjord's auctions contract.

Issues found

2 High Severity Bugs

Findings

[H-1] `FjordAuction::auctionEnd` function has a erraneous calculation , causing it to revert when `FjordAuction::totalTokens` is a large value , making users unable to claim their rewards , and tokens are forever stuck in the contract

Description `auctionEnd` function has the following line:

```
    function auctionEnd() external {
        if (block.timestamp < auctionEndTime) {
            revert AuctionNotYetEnded();
        }
        if (ended) {
            revert AuctionEndAlreadyCalled();
        }

        ended = true;
        emit AuctionEnded(totalBids, totalTokens);

        if (totalBids == 0) {
            auctionToken.transfer(owner, totalTokens);
            return;
        }

=>      multiplier = totalTokens.mul(PRECISION_18).div(totalBids);

        // Burn the FjordPoints held by the contract
        uint256 pointsToBurn = fjordPoints.balanceOf(address(this));
        fjordPoints.burn(pointsToBurn);
    }
```

Even though this line uses the `SafeMath` library of open-zeppelin , if `totalTokens.mul(PRECISION_18)` overflows , this will revert . This will revert only if `totalTokens` is set to a very large value . That value can be calculated as follows

```
    uint a = type(uint256).max;
    uint b = 1e18;
    uint c = a/b;
```

Any value greater than `c` will cause the overflow.

If it reverts , `ended` flag cannot be set true , and hence `FjordAuction::claimTokens` can never be called due to the following check

```
    function claimTokens() external {
=>      if (!ended) {
            revert AuctionNotYetEnded();
        }

        uint256 userBids = bids[msg.sender];
        if (userBids == 0) {
            revert NoTokensToClaim();
        }

        uint256 claimable = userBids.mul(multiplier).div(PRECISION_18);
        bids[msg.sender] = 0;

        auctionToken.transfer(msg.sender, claimable);
        emit TokensClaimed(msg.sender, claimable);
    }
```

Now , this `totalTokens` value is set in the constructor by whoever is deploying the contract. There is a very low chance the the deployer would be willing to put up so many tokens to be distributed (precisely greater than `c` as shown above) , but if they do , then the auction can never be completed AND money in the form of 2 tokens , `FjordAuction::fjordPoints` and `FjordAuction::auctionToken` , will be stuck in the contract forever.

Impact Bidders cannot claim their rewards , and both `FjordAuction::fjordPoints` and `FjordAuction::auctionToken` tokens are forever stuck in the contract

Proof of Concepts 1. A bidder bids in the auction 2. The auction ends 3. Somebody calls the `auctionEnd` function , which reverts

PoC

In your `auction.t.sol` , change your `totalTokens` value to the following

```
    uint a = type(uint256).max;
    uint b = 1e18; // equal to PRECISION_18
    uint c = a/b;
    uint256 public totalTokens = c+1 ;
```

And remember to comment out the following line

```
    uint256 public totalTokens = 1000 ether;
```

And , place the following test into `auction.t.sol` test suite

```
    function test_auctionEnd_HasMathThatBreaks() public{
        address bidder = address(0x2);
        uint256 bidAmount = 100 ether;

        deal(address(fjordPoints), bidder, bidAmount);

        vm.startPrank(bidder);
        fjordPoints.approve(address(auction), bidAmount);
        auction.bid(bidAmount);
        vm.stopPrank();

        skip(biddingTime);

        vm.expectRevert(); // panic error for arithmetic overflow will be triggered
        auction.auctionEnd();
    }
```

You will also notice that if you change your `totalTokens` variable to `c+1` , 3 of your pre-written tests ALSO FAIL .

Recommended mitigation Best mitigation is to check beforehand whether `totalTokens.mul(PRECISION_18)` will overflow , and if it will , carry out the division before the multiplication , as shown in the following code

```
    function auctionEnd() external {
        if (block.timestamp < auctionEndTime) {
            revert AuctionNotYetEnded(); // e this function can only be called after the 'deadline'
        }
        if (ended) {
            revert AuctionEndAlreadyCalled();
        }

        ended = true;
        emit AuctionEnded(totalBids, totalTokens);

        if (totalBids == 0) {
            auctionToken.transfer(owner, totalTokens);
            return;
        }

```

```
-       multiplier = totalTokens.mul(PRECISION_18).div(totalBids);

+       if (totalTokens > type(uint256).max.div(PRECISION_18)) {
+           multiplier = totalTokens.div(totalBids).mul(PRECISION_18);
+       } else {
+           multiplier = totalTokens.mul(PRECISION_18).div(totalBids);
+       }

        // Burn the FjordPoints held by the contract
        uint256 pointsToBurn = fjordPoints.balanceOf(address(this));
        fjordPoints.burn(pointsToBurn);
    }
```

By making this change , you will see that all of your tests in `auction.t.sol` will pass even with very lage values of `totalPoints`. Only one of the tests , `auction.t.sol::testAuctionEnd` will not pass as it has the same erraneous line , fix it , and then all your tests will pass.

[H-2] `FjordAuction::claimTokens` has a erraneous calcualation for `claimable` , causing it to overflow when `userBids.mul(multiplier)` overflows.

Description `claimTokens` function contains the following erraneous line

```
    function claimTokens() external {
        if (!ended) {
            revert AuctionNotYetEnded();
        }

        uint256 userBids = bids[msg.sender];
        if (userBids == 0) {
            revert NoTokensToClaim();
        }

=>      uint256 claimable = userBids.mul(multiplier).div(PRECISION_18);
        bids[msg.sender] = 0;

        auctionToken.transfer(msg.sender, claimable);
        emit TokensClaimed(msg.sender, claimable);
    }
```

This is similar to the previous finding I submitted , where I proved that the following line will cause oveflow error , am pasting the line again for reference :

```
    function auctionEnd() external {
        if (block.timestamp < auctionEndTime) {
            revert AuctionNotYetEnded();
        }
        if (ended) {
            revert AuctionEndAlreadyCalled();
        }

        ended = true;
        emit AuctionEnded(totalBids, totalTokens);

        if (totalBids == 0) {
            auctionToken.transfer(owner, totalTokens);
            return;
        }

=>      multiplier = totalTokens.mul(PRECISION_18).div(totalBids);

        // Burn the FjordPoints held by the contract
        uint256 pointsToBurn = fjordPoints.balanceOf(address(this));
        fjordPoints.burn(pointsToBurn);
    }
```

As I have already proved , under certain circumstances `totalTokens.mul(PRECISION_18)` will overflow . The specific case where it will overflow is re-iterated as follows:

```
    uint a = type(uint256).max;
    uint b = 1e18;
    uint c = a/b;
```

Any value greater than `c` will cause the overflow.

Now that we have established that `totalTokens.mul(PRECISION_18)` can overflow in some cases , we can also see that `totalTokens.mul(PRECISION_18)` is actually equal to `totalBids.mul(multiplier)` . Now consider the case where only 1 person bid in the auction(just taking this case for simplicity) , then this will actually equal `userBids.mul(multiplier)` where `userBids` is the bid of that user.

So , whenever `totalTokens` will be greater than `c` , then `totalTokens.mul(PRECISION_18)` overflows AND HENCE `userBids.mul(multiplier)` overflows.

Now look again at the problematic line in `claimTokens` function

```
    uint256 claimable = userBids.mul(multiplier).div(PRECISION_18);
```

Clearly `claimable` may overflow in cases described above. If it overflows , this will revert as we are using `.mul` method of `SafeMath` , which reverts in case of overflow. Hence the user will not be able to claim their rewards

Impact User will never be able to collect their rewards in some cases.