Current logic of SecondSwap_stepVesting.claimable() allows a beneficiary to steal from other beneficiary's allocation

Finding description and impact

The SecondSwap_stepVesting.claimable() is used to calculate claimable amount & claimable steps for a vesting. These are calculates for 2 situations - (i) when numSteps has not reached & (ii) when numSteps has reached. When numSteps has not reached then the claimableAmount is calculated based on the releaseRate, but the problem is that during this calculation it is not checked that at that time whether the beneficiary actually holds the available amount which was calculated using the releaseRate. Now assume a scenario: There are 2 beneficiary - beneficiary & beneficiary_2. A Vesting will be created for 10 days where steps is 10.

```
1. tokenIssuer deployed a vesting plan & he created a vesting for
beneficiary 100e18 amount.
2. After 2 days beneficiary claimed his claimable amount. Now his
totalAmount is 100e18, amountClaimed = 20e18 & stepsClaimed = 2.
3. Now tokenIssuer transferred 50e18 amount of vesting token to
beneficiary_2. Now beneficiary's totalAmount = (100e18 - 50e18) = 50e18,
releaseRate = 50e18/10 = 5e18.
4. After 7 days beneficiary claimed his claimable amount. As numSteps has
not reached yet so claimable amount will be calculated using releaseRate
which is 5e18. As claimableSteps will be (9 - 2) = 7 so claimableAmount
will be = 7 * 5e18 = 35e18. After claiming his balance become 55e18.
5. Now notice thing that at first total vesting was created for 100e18 &
50e18 was transferred for beneficiary_2. So, as per correct estimation the
beneficiary should not have claimed 55e18 amount of vesting token, he
should have claimed 50e18 amount of vesting token. So now the contract has
(100e18 - 55e18) = 45e18 amount of vesting token.
6. Now after 10 days numStep has reached & beneficiary_2 called claim() to
claim his 50e18 amount of vesting token, as he has not claimed yet so his
claimableAmount will be = (vesting.totalAmount - vesting.amountClaimed) =
50e18 - 0 = 50e18. But the tx will revert because the contract does not
have 50e18 amount of token, it has only 45e18 amount of token.
```

POC

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "../../contracts/SecondSwap_VestingManager.sol";
import "../../contracts/SecondSwap_VestingDeployer.sol";
import "../../contracts/SecondSwap_WhitelistDeployer.sol";
import "../../contracts/SecondSwap_MarketplaceSetting.sol";
import "../../contracts/SecondSwap_Marketplace.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "forge-std/Test.sol";
import "forge-std/console.sol";
import "../../contracts/interface/SecondSwap_Vesting.sol";

// Vesting token
contract MockERC20 is ERC20 {
    constructor()ERC20("TK","Token"){}
}

// Currency token, used to purchase vesting token
contract Currency is ERC20 {
    constructor()ERC20("CR","Currency"){}
}

// USDT token
contract MockUSDT is ERC20 {
    constructor()ERC20("USDT","Tether"){}

    function decimals() public view override returns (uint8) {
        return 6;
    }
}
abstract contract InitialDeploy is Test{

    SecondSwap_VestingManager manager;
    SecondSwap_VestingDeployer vestingDeployer;
    address beneficiary;
    address beneficiary_2;
    address s2Admin;
    function deployAndInitializeManager() public {
        manager = new SecondSwap_VestingManager();
        s2Admin = makeAddr("s2aAdmin");
        manager.initialize(s2Admin);
    }

    function deployVestingDeployer() public {
        vestingDeployer = new SecondSwap_VestingDeployer();
        vestingDeployer.initialize(s2Admin,address(manager));
    }
}
contract TestStepVesting is InitialDeploy {
    MockERC20 token;
    address tokenIssuer;
    address feeCollector;
    SecondSwap_WhitelistDeployer whitelistDeployer;
    MockUSDT usdt;
    SecondSwap_MarketplaceSetting marketplaceSetting;
    SecondSwap_Marketplace marketplace;
    Currency currency;
    function setUp() public {
        feeCollector = makeAddr("FEE_COLLECTOR");
        token = new MockERC20();
        beneficiary = makeAddr("BENEFICIARY");
        beneficiary_2 = makeAddr("BENEFICIARY_2");
        deployAndInitializeManager();
        deployVestingDeployer();
        tokenIssuer = makeAddr("TOKEN-ISSUER");
        deal(address(token), address(tokenIssuer), 1000e18);
        vm.startPrank(s2Admin);
        vestingDeployer.setTokenOwner(address(token), tokenIssuer);
        manager.setVestingDeployer(address(vestingDeployer));

        // Marketplace deployment logic
        /// get a feeCollector address
        /// deploy whitelistDeployer
        whitelistDeployer = new SecondSwap_WhitelistDeployer();
        /// deploy currency which will be used to purchase the vesting
token
        currency = new Currency();
        /// deploy usdt
        usdt = new MockUSDT();
        /// deploy MarektplaceSetting contract
        marketplaceSetting = new
SecondSwap_MarketplaceSetting(feeCollector, s2Admin,
address(whitelistDeployer), address(manager), address(usdt));
        /// Deploy marketplace
        marketplace = new SecondSwap_Marketplace();
        /// initializing the marketplace with the currency which users will
pay
        /// to the marketplace when they purchase vesting tokens.
        marketplace.initialize(address(currency),
address(marketplaceSetting));

        manager.setMarketplace(address(marketplace));
        vm.stopPrank();
        vm.warp(0);

        /// tokenIssuer need to set the maxSellPercent, admin need to set
buyerFee, sellerFee
    }

        function test_secondBeneficiaryCantWithdrawHisFullAmount() public {
        vm.startPrank(tokenIssuer);
        address newVesting = vestingDeployer.deployVesting(address(token),
block.timestamp, block.timestamp + 10 days, 10, "1");
        token.approve(address(newVesting), 1000e18);
        vestingDeployer.createVesting(beneficiary, 100e18, newVesting);
        uint currentTime = vm.getBlockTimestamp();
        vm.stopPrank();
        // Just after 2 days the beneficiary claimed his claimable amount
        skip(2 days);
        vm.prank(beneficiary);
        SecondSwap_Vesting(newVesting).claim();
        // Now beneficiary's totalAmount = 100e18, amountClaimed = 20e18,
stepsClaimed = 2

        // Now tokenIssuer transferred 50e18 amount vesting to another
```

```
beneficiary, named beneficiary_2
        vm.prank(tokenIssuer);
        SecondSwap_Vesting(newVesting).transferVesting(beneficiary,
beneficiary_2,50e18);
        // Now beneficiary's totalAmount = (100e18 - 50e18) = 50e18,
releaseRate = 50e18/10 = 5e18

        // @note If the first beneficiary waits till the endTime of vesting
then he will get the remaining of his claimable
        // amount, but if he claims one day earlier then he will receive
more amount. Because before the endTime reach the
        // claimable amount is calculated by releaseRate.
        // skip(8 days);
        // (uint claimableAmount, uint claimableSteps) =
SecondSwap_Vesting(newVesting).claimable(beneficiary);
        // assertEq(claimableAmount, 30e18);
        // So, he claimed after 7 days, i.e after 9 days in total i.e 1 day
before the vesting ends
         skip(7 days);
         (uint claimableAmount, uint claimableSteps) =
SecondSwap_Vesting(newVesting).claimable(beneficiary);
         assertEq(claimableAmount, 35e18);

        // Then he claimed
        vm.prank(beneficiary);
        SecondSwap_Vesting(newVesting).claim();

        // @note after claiming his balance is 55e18
        assertEq(token.balanceOf(beneficiary), 55e18);

        // Now contract has 45e18 amount of token
        assertEq(token.balanceOf(newVesting), 45e18);

        // Now endTime of the vesting has reached
        skip(1 days);
        assertEq(vm.getBlockTimestamp(), currentTime + 10 days);

        // Now claimableAmount for second beneficiary should be 50e18
because he did not claim anything yet
        (uint claimableAmount_2, uint claimableSteps_2) =
SecondSwap_Vesting(newVesting).claimable(beneficiary_2);
        assertEq(claimableAmount_2, 50e18);
        assertEq(claimableSteps_2, 8);

        // But now his claim will revert because the contract does not have
enough token
        // to pay the beneficiary. Because token balance of the contract is
45e18 but
        // claimable amount is 50e18.
        vm.prank(beneficiary_2);
        vm.expectRevert();
        SecondSwap_Vesting(newVesting).claim();
    }
```

```
}
```

Recommended mitigation steps

Use the SecondSwap_stepVesting.available() to check whether the claimableAmount is matching with the available amount before executing the claim.

Purchase cannot possible if discount is 100%

Finding description and impact

The discount percentage of a listing is completely depends on the seller/beneficiary. If he want then he can choose to give 100% discount. But the marketplace contract will DoS when discount percentage is 100% i.e 10000. Assume a scenario where seller wants to airdrop a very little amount of vesting token:

1. Seller listed 10e18 amount of vesting token for 100% discount percentage & DiscountType is FIX.
2. A user attempted to buy those token. Here the problem starts when discountPrice is calculated in _getDiscountPrice(), as discountType is FIX so this code will execute inside that function:

```
} else if (listing.discountType == DiscountType.FIX) {
            discountedPrice = (discountedPrice * (BASE - 
listing.discountPct)) / BASE;
        }
```

As listing.discountPct is 10000 so dicountPrice will be 0. 3. Now inside _handleTransfers() baseAmount is calculated like this:

```
uint256 baseAmount = (_amount * discountedPrice) /
            uint256(
                10 **
                    (
                        IERC20Extended(
                            address(
                                
IVestingManager(IMarketplaceSetting(marketplaceSetting).vestingManager())
                                    
.getVestingTokenAddress(listing.vestingPlan)
                            )
                        ).decimals()
                    )
            );
```

As discountPrice is 0 so baseAmount will also be 0 & the tx will revert due to this check after the baseAmount calculation:

```
require(baseAmount > 0, "SS_Marketplace: Amount too little");
```

Recommended mitigation steps

Either do not allow 100% discount or put a check for discount percentage for the listing after calculating baseAmount inside _handleTransfers(), then put the above mentioned require statement.

Incorrect require statement check inside SecondSwap_MarketPlace.listVesting() allows to set 0 as _minPurchaseAmt for PARTIAL listings.

Finding description and impact

A listing is SINGLE type means it is required to purchase the all listed amount at once whereas a listing type is PARTIAL means the listing can be partially purchased. There is a require statement in SecondSwap_Marketplace.listVesting() which checks that whether the to be listed vesting has correctly configured with listing type & amount or not. The check looks like this:

```
require(
            _listingType != ListingType.SINGLE || (_minPurchaseAmt > 0 &&
_minPurchaseAmt <= _amount),
            "SS_Marketplace: Minimum Purchase Amount cannot be more than
listing amount"
        );
```

In this statement if _listingType != ListingType.SINGLE satisfies then the second condition is not checked. Now assume someone want to list a PARTIAL type listing with 0 _minPurchaseAmt which is restricted, what will happen is first it will check the first condition, as the listingType is not equal to SINGLE so the check will satisfy and the amount check will not be done. By this one can list a PARTIAL listing with 0 _minPurchaseAmt.

Recommended mitigation steps

```
function listVesting(
        address _vestingPlan,
        uint256 _amount,
        uint256 _price,
        uint256 _discountPct,
        ListingType _listingType,
        DiscountType _discountType,
        uint256 _maxWhitelist,
        address _currency,
        uint256 _minPurchaseAmt,
        bool _isPrivate
```

```
    ) external isFreeze {
        require(
-            _listingType != ListingType.SINGLE || (_minPurchaseAmt > 0 &&
_minPurchaseAmt <= _amount),
+            _listingType != ListingType.PARTIAL || (_minPurchaseAmt > 0 &&
_minPurchaseAmt <= _amount),
            "SS_Marketplace: Minimum Purchase Amount cannot be more than
listing amount"
        );
        // rest of codes...
}
```

Seller/beneficiary can list more than maxSellPercent which breaks the invariant which says it is not possible to list more than maximum sell percentage.

Finding description and impact

maxSellPercent is a crucial invariant which says that a seller can't list more than maxSellPercent of his total vesting amount. But this can be broken. There is a function called spotPurchase() in SecondSwap_VestingMarketplace contract which is used to purchase vesting token from beneficiaries, also there is a local variable called sellLimit in SecondSwap_VestingManager.listVesting(), seller can use these 2 to list more than the max sell percent. Assume this scenario:

```
1.  tokenIssuer created a vesting for beneficiary of 1000e18 &
maxSellPercent is 2000 i.e 20%. i.e it is not possible to list more than
200e18 for the beneficiary.
2. The beneficiary listed 200e18 amount of vesting token.
3. Now the spotPurchase() does not have any restriction on buyer, so seller
bought 50e18 amount of vesting token from his own listing. So now for
seller Allocation.bought = 50e18, Allocation.sold = 200e18 & totalAmount =
850e18. Because after listing 200e18 from 1000e18 he bought 50e18. We can
see below calculation to know how much amount the beneficiary list again:
```

Inside listVesting():

```
        sellLimit = 50e18
        currentAlloc = 850e18
        sold = 200e18
        currentAlloc + sold > bought because (850e18 + 200e18) > 50e18
        sellLimit = (850e18 + 200e18 - 50e18) * 2000/10000 + 50e18
                  = (1000e18 * 2000/10000) + 50e18
                  = 200e18 + 50e18 = 250e18
                  Means we can list more 50e18 vesting token. It
                  clearly breaking the invariant of maxSellPercent.
```

So now sellLimit is 250e18 & the beneficiary has sold 200e18, means he can list more 50e18 amount. But limit was 200e18. From here we can clearly see that the seller can list more than the maxSellPercent.

Recommended mitigation steps

Do not allow a seller to purchase from its own listing.

Incorrect handling of USDT in penaltyFee

Finding description and impact

As mentioned in SecondSwap_MarketplaceSetting contract penaltyFee is charged in wei.

```
    /**
     * @notice Fee charged for early unlisting (in wei)
     */
    uint256 public penaltyFee;
```

But this will create issue because penaltyFee is charged in USDT which has 6 decimal. So if penaltyFee is set in 1000 wei then 1000 USDT will be transferred from the seller's account. See the below mentioned snippet from unlistVesting().

```
(IMarketplaceSetting(marketplaceSetting).usdt()).safeTransferFrom(
                    msg.sender,
                    IMarketplaceSetting(marketplaceSetting).feeCollector(),
// 3.7. Value difference caused by the same penalty fee
                    IMarketplaceSetting(marketplaceSetting).penaltyFee()
                );
```

Proof of Concept

https://github.com/code-423n4/2024-12-secondswap/blob/214849c3517eb26b31fe194bceae65cb0f52d2c0/contracts/SecondSwap_MarketplaceSetting.sol#L26-L29

https://github.com/code-423n4/2024-12-secondswap/blob/214849c3517eb26b31fe194bceae65cb0f52d2c0/contracts/SecondSwap_Marketplace.sol#L355-L359

Recommended mitigation steps

Handle the wei & USDT correctly.