CD SECURITY

AUDIT REPORT

Pear Protocol
March 2025

Prepared by
Inh3l
Pelz

Introduction

A time-boxed security review of the **Pear Protocol** was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Pear Protocol**

The `PearVesting` contract, which was the focus of this audit, manages token vesting schedules, ensuring that recipients receive their allocated tokens gradually over time. It supports customizable vesting plans with parameters such as start time, cliff period, and end time, allowing for precise and controlled token distribution. Recipients can claim vested tokens according to their schedules, while administrators have the ability to create and modify vesting plans. The system enforces vesting restrictions to prevent premature token access, ensuring fair and predictable token releases.

Severity classification

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

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 8e849b80110f027172b473a195f1e8a5de0cb67d**

Scope

The following smart contracts were in scope of the audit:

- `src/vesting/PearVesting.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 0 issues
- Low: 3 issues

Findings Summary

ID	Title	Severity	Status
[L-01]	`changeRecipient()` doesn't update recipient's vesting	Low	Fixed
[L-02]	Edgecase can occur upon `vestingPlan` update which can temporarily DOS claims for users	Low	Fixed
[L-03]	Vesting Plan end time can be set before cliff period	Low	Fixed
[I-01]	Redundant check for pear token transfer success	Informational	Fixed

Detailed Findings

[L-01] `changeRecipient()` doesn't update recipient's vesting

Description

`changeRecipient` doesn't update `userVestingPlans` mapping. As a result, a new recipient's vesting plan is incorrectly mapped to the older recipient when queried.

In `_issueVestingPlan`, the set vesting plan is pushed into the recipient's array of `userVestingPlans`.

```
function _issueVestingPlan(address recipient, uint256 amount, uint32
start, uint32 cliff, uint32 end) internal returns (uint256) {
    if (recipient == address(0)) revert Errors.PearVesting_ZeroAddress();
    if (amount == 0) revert Errors.PearVesting_ZeroAmount();
    if (end <= start) revert Errors.PearVesting_InvalidEndTime();

    uint256 planId = nextVestingPlanId++;

    vestingPlans[planId] = VestingPlan({ amount: amount, claimed: 0,
recipient: recipient, start: start, cliff: cliff, end: end, lastClaim: 0
});

@>  userVestingPlans[recipient].push(planId);
```

```
    if (!pearToken.transferFrom(msg.sender, address(this), amount)) {
      revert Errors.PearVesting_TransferFailed();
    }
    emit Events.VestingPlanCreated(planId, recipient, amount, start,
cliff, end);

    return planId;
  }
```

But in `changeRecipient`, this array is not updated, either for the old recipient or the new recipient.

```
  function changeRecipient(uint256 planId, address newRecipient) external
override onlyAdmin {
    VestingPlan storage plan = vestingPlans[planId];
    if (plan.recipient == address(0)) {
      revert Errors.PearVesting_InvalidPlanId();
    }
    if (newRecipient == address(0)) {
      revert Errors.PearVesting_ZeroAddress();
    }

    plan.recipient = newRecipient;
    emit Events.VestingRecipientChanged(planId, newRecipient);
  }
```

As a result, when `getUserVestingPlans` is called for the new recipient, the plan will not show up, while it shows up for the old recipient.

Recommendations

Push the plan into the new recipient's array while removing it from the old recipient's. This might involve looping through all of old recipient's plan, in which case, care must be take to ensure that the loop doesn't run out gas.

[L-02] Edgecase can occur upon `vestingPlan` update which can temporarily DOS claims for users

Description

The returned value of `_claimableAmount` is dependent on the returned value of `vestedAmount - plan.claimed`. Since the vesting plan's amount and end time can be updated, via `updateVestingPlan`, an edgecase can occur in which, the amount claimed by the user before the plan's is updated exceeds the calculated value of `vestedAmount` after the update. This mostly occurs, when the amount is slightly increased, while the end time enjoys a larger increment.

Let's consider the example below:

Initial setup:

start = X + 100 cliff = 100 end = X + 300 amount = 300

Based on the function as seen below, at `block.timestamp` = X + 230, for instance, `vestedAmount` will be 195, which upon claiming will be set as `plan.claimed`. So, after initial claim, `plan.claimed` == 195.

```
  function _claimableAmount(uint256 planId) internal view returns
(uint256) {
    VestingPlan memory plan = vestingPlans[planId];
    if (block.timestamp < plan.start + plan.cliff) {
      return 0;
    }

    // 50% of token amount is vested on cliff date leaving %50 to be
vested linerly to end date
    uint256 amountToBeVested = plan.amount / 2;
    uint256 vestedAmount = amountToBeVested;
    if (block.timestamp >= plan.end) {
      vestedAmount = plan.amount;
    } else {
      // released = amount * (startTime+duration-currentTime) / duration
@>    vestedAmount += (amountToBeVested * (block.timestamp - (plan.start +
plan.cliff))) / (plan.end - (plan.start + plan.cliff));
    }

@>  return vestedAmount - plan.claimed;
  }
```

Immediately after initial claim, the plan's details updated.

end = X + 600 amount = 350

Before and as at `block.timstamp` = X + 245, `vestedAmount` will be 194, which upon attempting to deduct `plan.claimed` which is 195, will result in an underflow error.

Recommendations

Introduce the possibility of this happening in protocol documentation. A potential fix is to compare `vestedAmount` to `plan.claimed` and if less than or equal to, return 0 instead.

[L-03] Vesting Plan end time can be set before cliff period

Description

Both `_issueVestingPlan` and `updateVestingPlan` functions allow setting an invalid vesting end time that is earlier than the cliff period, contradicting the expected vesting behavior.

Issue in `_issueVestingPlan`:

The function checks that `end` is greater than `start`, but it does not verify that `end` is greater than `start + cliff`. This omission allows creating a vesting plan where the end time falls before the cliff period, meaning tokens that should vest after the cliff will become 100% claimable after cliff period.

Affected Code:

```
function _issueVestingPlan(address recipient, uint256 amount, uint32
start, uint32 cliff, uint32 end) internal returns (uint256) {
    if (recipient == address(0)) revert Errors.PearVesting_ZeroAddress();
    if (amount == 0) revert Errors.PearVesting_ZeroAmount();
    if (end <= start) revert Errors.PearVesting_InvalidEndTime(); //
@audit-issue Should check `end <= start + cliff`
```

Issue in `updateVestingPlan`:

Similarly, the `updateVestingPlan` function allows modifying an existing vesting plan's end time without ensuring that `newEnd` is greater than `plan.start + plan.cliff`. If the end time is reduced below the cliff period, vesting will not behave as expected, leading to incorrect token distribution.

Affected Code:

```
function updateVestingPlan(uint256 planId, uint256 newAmount, uint32
newEnd) external override onlyAdmin {
    VestingPlan storage plan = vestingPlans[planId];
    if (plan.recipient == address(0)) {
      revert Errors.PearVesting_InvalidPlanId();
    }
    if (newEnd <= plan.start) revert Errors.PearVesting_InvalidEndTime();
// @audit-issue should check if `newEnd <= plan.start + plan.cliff`
```

Recommendations

Fix in `_issueVestingPlan`:

Add a validation check to ensure `end` is greater than `start + cliff`:

```
if (end <= start + cliff) revert Errors.PearVesting_InvalidEndTime();
```

Fix in `updateVestingPlan`:

Modify the validation to ensure `newEnd` is greater than `plan.start + plan.cliff`:

```
if (newEnd <= plan.start + plan.cliff) revert
Errors.PearVesting_InvalidEndTime();
```

[I-01] Redundant check for pear token transfer success

Description

`pearToken` is an ERC20 token based on Openzeppelin's ERC20.sol contract.

```
/// @title PearToken
/// @notice ERC20 token for Pear tokens
contract PearToken is ERC20 {
```

In the `transfer`/`transferFrom` functions, a bool of true is returned if and only if the transaction is successful. The functions revert if not successful (if balance is not enough, or sender/recipient is address 0).

```
function transfer(address to, uint256 value) public virtual returns (bool)
{
    address owner = _msgSender();
    _transfer(owner, to, value);
    return true;
}
```

As a result, checking for pearToken's `transfer`/`transferFrom` success is not needed in the `updateVestingPlan`, `_issueVestingPlan` and `claim` functions.

```
    if (!pearToken.transferFrom(msg.sender, address(this), amount)) {
      revert Errors.PearVesting_TransferFailed();
    }
```

```
    bool success = pearToken.transfer(plan.recipient, claimableAmount);
    if (!success) revert Errors.PearVesting_TransferFailed();
```

```
    if (!pearToken.transferFrom(msg.sender, address(this), difference)) {
      revert Errors.PearVesting_TransferFailed();
    }
```

Recommendations

Remove the boolean checks.