Hedgey - DelegateTokenClaims / VestingLockups

Date	April 2024

1 Executive Summary

This report presents the results of the engagement with **Hedgey Finance** to review **VestingLockups** and **DelegateTokenClaims**.

The review was conducted over two weeks, from **April 8, 2024** to **April 22, 2024** by **Martin Ortner** and **Arturo Roura**. Due to changes in scope, it was found necessary to extend the engagement from 8x2 to 10x2 person days to better cover risk areas of the two code repositories under review.

Our examination of the codebase highlighted notable strengths such as its structural organization and functional capabilities. However, it also identified areas warranting attention, particularly regarding the accuracy of accounting practices in specific code segments. Notably, there is a reliance on strict outcomes, where the code is structured to either function flawlessly or face operational challenges.

The utilization within this system of any token that deviates from the standard ERC20 implementation requires meticulous consideration. The system is incompatible with tokens that implement fees-on-transfer, exhibit deflationary mechanisms or that modify the standard allowance functionality. Moreover, tokens with potential future fee implementations risk becoming immobilized within the system.

Additionally, the contracts aim to optimize storage usage by cleaning up and burning tokens when they are deemed inactive. While this optimization is generally beneficial, it may inadvertently result in premature zeroing of key mappings/variables or allow for unintended reuse (burning tokens; deleting admins).

During the review process, it was observed that a related contract (`ClaimCampaigns` ; not part of the scope of this review) was exploited (2024-04-19). The vulnerability exploited is also present in the `DelegateTokenClaims` component, and as such tracked with this report.

The fix review was conducted for one week, from **May 6, 2024** to **May 10, 2024** by **Vlad Yaroshuk** and **Arturo Roura**.

2 Scope

Our review focused on the following repositories:

- VestingLockups (06f41767f57b05f77f3c16fc78103aaacb5f557b).
- DelegatedTokenClaims (905fdc2dc89849ba19e718e4147b4b8306f894f2).

Our fix review focused on the following repositories:

- VestingLockups (972c273f9aaf81f88e43a87d60081f041ce9df52).
- DelegatedTokenClaims (4446cb36cc45345f979135051206bbe4fdf36c13). Subsequently, during the review process, commit (4829924d71e458549b125435f0f025c7304aa390) was introduced for comprehensive examination, with a primary focus on implementing additional rectifications relevant to the DelegatedTokenClaims repository.

The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with **Hedgey Finance**, we identified the following priorities for this review:

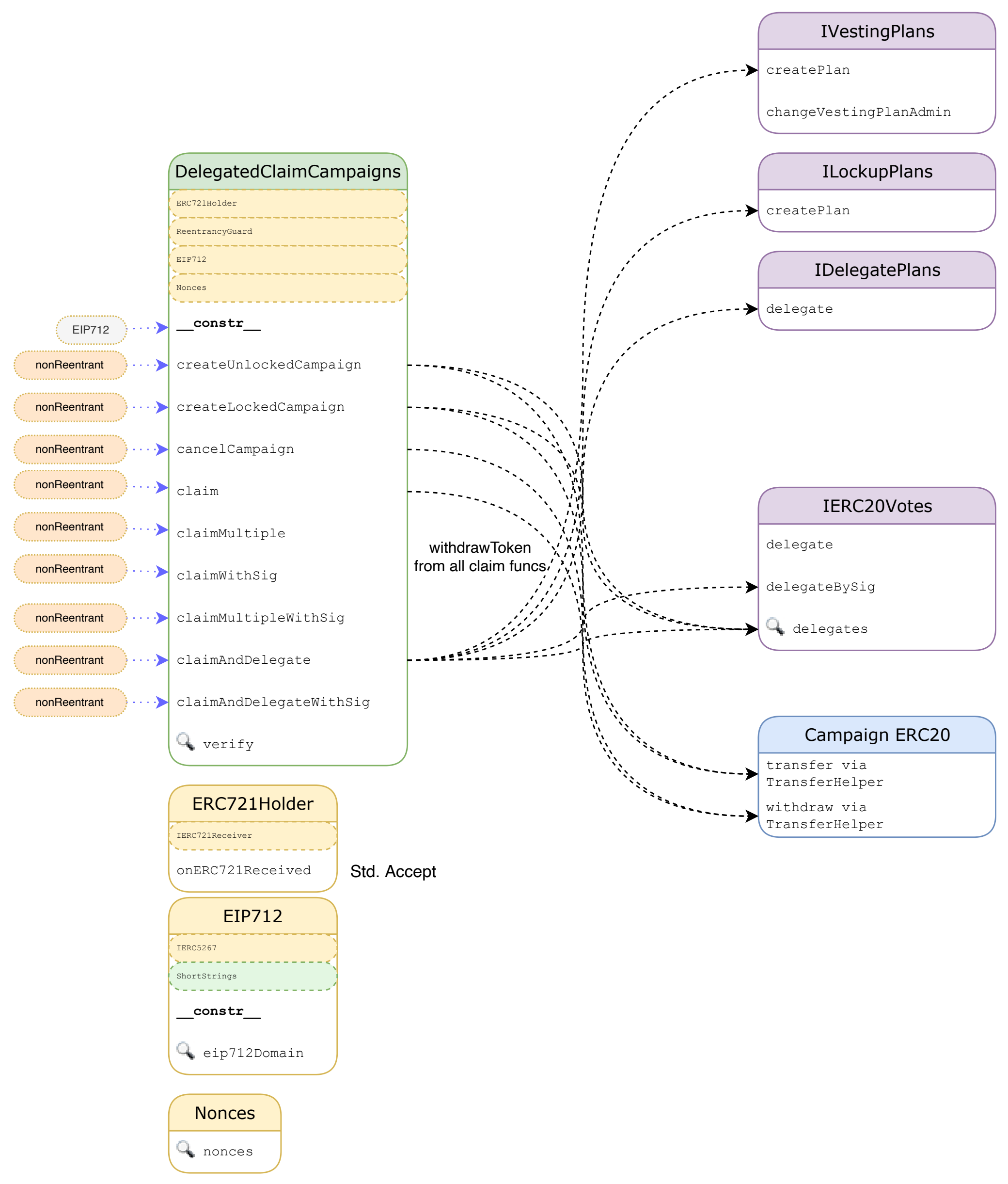
1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 Document Change Log

Version	Date	Description
1.0	2024-04-22	Initial report
2.0	2024-05-09	Resolution boxes added with corresponding fixes
2.1	2024-05-09	Fix review added to executive summary
2.2	2024-05-10	Fix commits adjusted
2.3	2024-05-10	Token considerations added to executive summary

4 System Overview

4.1 DelegateTokenClaims



Deployer

- set EIP712 domain, does not remain in control of the contract.

CampaignManager

- can cancel a campaign at any point in time, returning unclaimed token -> manager revokes access to campaign

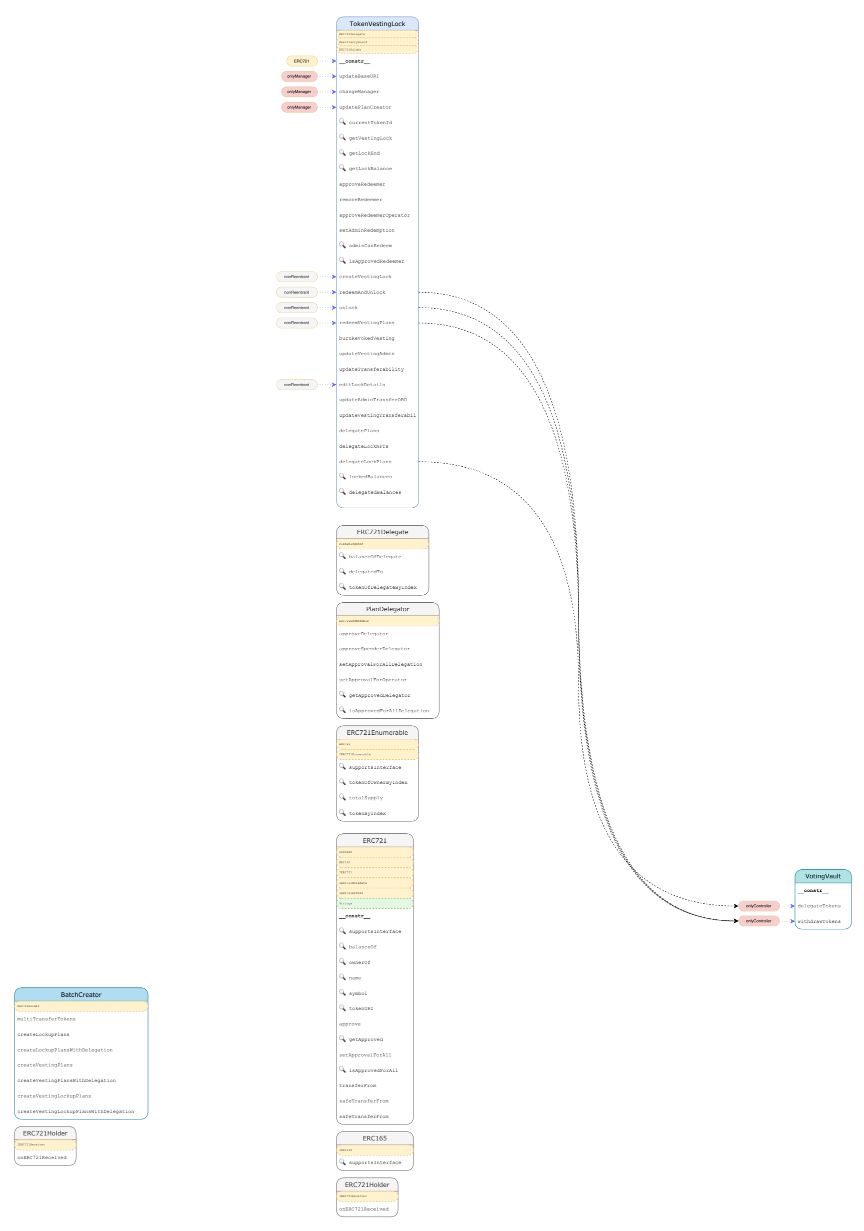
Claimer

- can claim locked/unlocked tokens for single campaignId one time
- can claim locked/unlocked tokens for multiple campaignIds in batch (one time per campaignId)
- can claim and delegate unlocked tokens with `ERC20Votes.delegateBySig` delegation signature and locked tokens directly (to delegator).

Anyone

- can create locked/unlocked campaigns (token approval required) -> becomes `CampaignManager`
- can claim (and delegate) locked/unlocked tokens for `Claimer` with EIP712 signature (checked-nonce, expiry) **PayMaster** role.
  - ⚠️ claim multiple with signature only checks first element which is fine according to the client as it just proves the intention and claiming any other campaigns always benefit the claimer and not the caller.
  - ⚠️ `CLAIM_TYPE` struct (signature structure) is the same for single claim, multi claim, claim and delegate (front-run)
  - ⚠️ allows empty merkle proofs, effectively proving against the merkle root (tree with one element) which technically is ok but in most cases makes no sense. It is suggested to enforce tree depth or leaf id checks for stronger security guarantees.
  - ⚠️ claim and delegate can be front-run setting any delegatee (see findings)

4.2 VestingLockups



Deployer

- deploys & parameterizes contract
- sets vesting plan
- sets batch plan creator
- is set as initial `Manager` for the contract

🧍 Manager

- Can change `ERC721.baseURI` at any point in time
- Can transfer `Manager` role to new `Manager` (1-step! - Consider 2-step transfer, else, risk of `Manager` losing access)
- Can update batch plan creator (`BatchCreator`)

🧍 hedgeyPlanCreator

- `BatchCreator` contract
- In charge of creating vesting plans and a lockupPlans for those newly created vesting plans
- This role can be modified by the `Manager`

🧍 VestingAdmin

There are two different `vestingAdmin` states being maintained: the vesting contract vestingAdmin and the lockup recorded vestingAdmin. Upon lockup plan creation the lockup vestingAdmin will be set to the underlying vesting plan vestingAdmin, however, they can be set independently.

- Within the vesting contract:
  - Can revoke plans, making any non-vested tokens return to the `vestingAdmin` and the tokens that are vested will be delivered to the beneficiaries
  - Can revoke plans in the present or in the future
    - If the admin chooses to remove revoke on a timestamp way in the future, he will receive all tokens that would be unvested after the revoke time
  - Can change the vesting contract vesting admin
  - Can change the lockup contract vesting admin
  - If `adminTransferOBO` is set to `true` for the vesting plan, it is possible for the admin to transfer ownership of the ERC721
- Within the lockup contract:
  - If the ownership is transferred to the lockup contract the vesting admin can create vesting locks for the plan
  - Can redeem and unlock
  - Can change the lockup contract vesting admin
  - Can update the lockup nft transferability
  - Can edit the lockup plan as long as the current timestamps hasn't gone over the first unlockable timestamp.
  - If `adminTransferOBO` is set to `true` for the lockup plan, it is possible for the admin to transfer ownership of the ERC721

🧍 Approved Redeemer

- Redeem in the vesting contract can only be done by the owner, approved redeemers have the authority in the lockup contract(owner of the vesting nft) to redeem on behalf of it
- The owner of the lockup(Beneficiary) and the redeem operators can set the admin as a redeemer

🧍 Approved Operator Delegators

- Can delegate
- Can redeem and unlock on behalf of the beneficiary

🧍 Plan Beneficiaries

- Can redeem their own tokens
- Can unlock their own tokens
- Can update vesting plan admin transferability
- Can update lockup token admin transferability
- Can update lockup token general transferability
- Can delegate

5 Findings: General

Each issue has an assigned severity:

- `Minor` issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- `Medium` issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- `Major` issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- `Critical` issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 Always Use the Best Type Available `Acknowledged`

Description

Declare state variables with the best type available and downcast to `address` if needed. Typecasting inside the corpus of a function is unneeded when the parameter's type is known beforehand. Declare the best type in function arguments, state vars. Always return the best type available instead of falling back to `address` .

Examples

- TransferHelper - argument declarations `address` should be `IERC20`

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/libraries/TransferHelper.sol:L18-L30

```
function transferTokens(
  address token,
  address from,
  address to,
  uint256 amount
) internal {
  uint256 priorBalance = IERC20(token).balanceOf(address(to));
  require(IERC20(token).balanceOf(from) >= amount, 'THL01');
  SafeERC20.safeTransferFrom(IERC20(token), from, to, amount);
  uint256 postBalance = IERC20(token).balanceOf(address(to));
  require(postBalance - priorBalance == amount, 'THL02');
}
```

- TokenVestingLock - declare `IVesting` in arguments

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/TokenVestingLock.sol:L122-L131

```
constructor(
  string memory name,
  string memory symbol,
  address _hedgeyVesting,
  address _hedgeyPlanCreator
) ERC721(name, symbol) {
  hedgeyVesting = IVesting(_hedgeyVesting);
  hedgeyPlanCreator = _hedgeyPlanCreator;
  manager = msg.sender;
}
```

5.2 TransferHelper - Misleading Comment ✓ Fixed

Resolution

Addressed in commit `4829924d71e458549b125435f0f025c7304aa390`.

Description

An inline comment in `TransferHelper` mentions the library being used for `WETH` handling, which is not accurate. It is recommended to change the comment expressing that the library is used to enforce that FeeOnTransfer Tokens cannot be used.

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/libraries/TransferHelper.sol:L8-L9

```
/// @notice Library to help safely transfer tokens and handle ETH wrapping and unwrapping of WETH
library TransferHelper {
```

5.3 TransferHelper - Unused `using for` Declaration Acknowledged

Description

`SafeERC20` is declared via the `usingFor` syntax, however, it is not being accessed that way. This might be a misunderstanding on how `usingFor` works or just an oversight. In any way, we would recommend to either remove the declaration or stick to the `usingFor` syntax.

Examples

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/libraries/TransferHelper.sol:L9-L29

```
library TransferHelper {
  using SafeERC20 for IERC20;

  /// @notice Internal function used for standard ERC20 transferFrom method
  /// @notice it contains a pre and post balance check
  /// @notice as well as a check on the msg.senders balance
  /// @param token is the address of the ERC20 being transferred
  /// @param from is the remitting address
  /// @param to is the location where they are being delivered
  function transferTokens(
    address token,
    address from,
    address to,
    uint256 amount
  ) internal {
    uint256 priorBalance = IERC20(token).balanceOf(address(to));
    require(IERC20(token).balanceOf(from) >= amount, 'insufficient balance');
    SafeERC20.safeTransferFrom(IERC20(token), from, to, amount);
    uint256 postBalance = IERC20(token).balanceOf(address(to));
    require(postBalance - priorBalance == amount, 'to_error');
  }
```

5.4 TransferHelper - Variation in Error Messages Between DelegateClaimCampaigns and VestingLockup TransferHelper ✓ Fixed

Resolution

Addressed in commit `9ace701e9cb085ce4a2ab1e890f83f4509dc4339`.

Description

It was noted that the two `TransferHelper` implementations have subtle variations in error messages that may be unintended.

Examples

- DelegateTokenClaims

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/libraries/TransferHelper.sol:L18-L29

```
function transferTokens(
  address token,
  address from,
  address to,
  uint256 amount
) internal {
  uint256 priorBalance = IERC20(token).balanceOf(address(to));
  require(IERC20(token).balanceOf(from) >= amount, 'THL01');
  SafeERC20.safeTransferFrom(IERC20(token), from, to, amount);
  uint256 postBalance = IERC20(token).balanceOf(address(to));
  require(postBalance - priorBalance == amount, 'THL02');
}
```

- VestingLockups

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/libraries/TransferHelper.sol:L18-L29

```
function transferTokens(
  address token,
  address from,
  address to,
  uint256 amount
) internal {
  uint256 priorBalance = IERC20(token).balanceOf(address(to));
  require(IERC20(token).balanceOf(from) >= amount, 'insufficient balance');
  SafeERC20.safeTransferFrom(IERC20(token), from, to, amount);
  uint256 postBalance = IERC20(token).balanceOf(address(to));
  require(postBalance - priorBalance == amount, 'to_error');
}
```

6 Findings: DelegateTokenClaims

6.1 `claimLockup.tokenLocker` Can Set Arbitrary Token Approval by Creating-Canceling Campaign, Stealing Token Locked in the Contract Critical ✓ Fixed

Resolution

Addressed in commit `4829924d71e458549b125435f0f025c7304aa390` the client addressed this issue with several changes to the codebase:

1. Adding a flash loan protection to the creation and canceling of campaigns.
2. Instead of initially increasing the allowance to the token locker for the full campaign amount during campaign creation, a revised approach has been implemented. Now, the allowance is increased during the claiming process by `claimAmount`. Subsequently, a verification step ensures that the allowance reaches zero upon completion of the claiming process. Notably, the cancellation of a campaign requires that the allowance returns to its default state of zero.
3. `createLockedCampaign` requires that the `claimLookup.tokenLocker`, passed in as an argument, be whitelisted based on a list of valid token lockers set in the constructor.

Description

Note: This finding is from an attack on the `ClaimCampaigns` contract that was exploited by a malicious actor while this engagement was ongoing. The exploited vulnerability is also present in the `DelegateTokenClaims` contract.

`createLockedCampaign` calls `SafeERC20.safeIncreaseAllowance` on the `claimLockup.tokenLocker` contract, the plan creator provides. There are no restrictions on this `tokenLocker` and it is chosen by the caller. The token approval is not reset after the transaction nor in the `cancelCampaign` function. Futhermore, flashloans are feasible because campaigns can be created and cancelled in the same block.

An attacker, therefore, can call `createLockedCampaign` setting themselves as `claimLockup.tokenLocker`, receiving the token approval on an external `ERC20`, providing an initial amount of tokens via a flash-loan and repaying it directly by canceling the campaign in the same transaction. The token approval will not be reset, the attacker can spend tokens owned by `DelegatedClaimCampaigns` (other campaigns tokens) on their behalf.

Examples

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L147-L175

```
function createLockedCampaign(
  bytes16 id,
  Campaign memory campaign,
  ClaimLockup memory claimLockup,
  address vestingAdmin,
  uint256 totalClaimers
) external nonReentrant {
  require(!usedIds[id], 'in use');
  usedIds[id] = true;
  require(campaign.token != address(0), '0_address');
  require(campaign.manager != address(0), '0_manager');
  require(campaign.amount > 0, '0_amount');
  require(campaign.end > block.timestamp && campaign.end > campaign.start, 'end error');
  require(campaign.tokenLockup != TokenLockup.Unlocked, '!locked');
  if (campaign.delegating) {
    require(IERC20Votes(campaign.token).delegates(address(this)) == address(0), '!erc20votes');
  }
  if (campaign.tokenLockup == TokenLockup.Vesting) {
    require(vestingAdmin != address(0), '0_admin');
    _vestingAdmins[id] = vestingAdmin;
  }
  require(claimLockup.tokenLocker != address(0), 'invalide locker');
  TransferHelper.transferTokens(campaign.token, msg.sender, address(this), campaign.amount);
  claimLockups[id] = claimLockup;
  SafeERC20.safeIncreaseAllowance(IERC20(campaign.token), claimLockup.tokenLocker, campaign.amount);
  campaigns[id] = campaign;
  emit ClaimLockupCreated(id, claimLockup);
  emit CampaignStarted(id, campaign, totalClaimers);
}
```

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L179-L186

```
function cancelCampaign(bytes16 campaignId) external nonReentrant {
  Campaign memory campaign = campaigns[campaignId];
  require(campaign.manager == msg.sender, '!manager');
  delete campaigns[campaignId];
  delete claimLockups[campaignId];
  TransferHelper.withdrawTokens(campaign.token, msg.sender, campaign.amount);
  emit CampaignCancelled(campaignId);
}
```

Recommendation

- Enforce that the lockup contract pulls in the same amount of tokens `DelegateTokenClaims` pulled in. Cancel token approvals after the lockup contract pulled in the tokens.
- Enforce a minimum waiting time between creating a campaign and canceling it. There is no reason to allow same-block create/cancel events. This will make (multi-block mev) flashloans infeasible.

6.2 `claimMultipleWithSig` - Missing Signature Expiration Check Major ✓ Fixed

Resolution

Addressed in commit 1b3c3da8fd1a1f8a25822c2be2e0ebeaa988179.

The missing expiration check is set in `claimMultipleWithSig`, comparing the `claimSignature.expiry` with the current `block.timestamp`.

Additionally, in commit 6c747bb5254fe9cfaf4f9009370ba13a951fc88f `MULITCLAIM_TYPEHASH` differentiates single claims from batched claims.

Description

`claimMultipleWithSig` does not check for signature expiration. Therefore, an expired signature that would otherwise be rejected by `claimWithSig` can be used with `claimMultipleWithSig`.

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L262-L280

```
function claimMultipleWithSig(
  bytes16[] calldata campaignIds,
  bytes32[][] calldata proofs,
  address claimer,
  uint256[] calldata claimAmounts,
  SignatureParams memory claimSignature
) external nonReentrant {
  address signer = ECDSA.recover(
    _hashTypedDataV4(
      keccak256(
        abi.encode(CLAIM_TYPEHASH, campaignIds[0], claimer, claimAmounts[0], claimSignature.nonce, claimSignature.expiry)
      )
    ),
    claimSignature.v,
    claimSignature.r,
    claimSignature.s
  );
  require(signer == claimer, 'invalid claim signature');
  _useCheckedNonce(claimer, claimSignature.nonce);
```

- For reference, `claimWithSig` checks expiration here

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L233-L244

```
function claimWithSig(
  bytes16 campaignId,
  bytes32[] calldata proof,
  address claimer,
  uint256 claimAmount,
  SignatureParams memory claimSignature
) external nonReentrant {
  require(!claimed[campaignId][claimer], 'already claimed');
  require(!campaigns[campaignId].delegating, 'must delegate');
  require(claimSignature.expiry > block.timestamp, 'claim expired');
  address signer = ECDSA.recover(
    _hashTypedDataV4(
```

Recommendation

Require that the signature is not expired:

```
require(claimSignature.expiry > block.timestamp, 'claim expired');
```

6.3 An Attacker Can Frontrun `claimAndDelegateWithSig` Setting Any `delegatee` Major ✓ Fixed

Resolution

Addressed in commit `f59c1a8e8ffa96f0cb0d6eb54469995973e67b84`.

`delegatee` is now part of the signed data and therefore cannot be set arbitrarily by front running the transaction.

Description

A malicious actor observing a paymasters transaction to claiming and delegating locked tokens with signature `claimAndDelegateWithSig() -> _claimLockedAndDelegate()` can frontrun the transaction, setting an arbitrary `delegatee`. This is possible because `delegatee` is not part of the claim signature nor is `delegationSignature` checked to be signed by `delegatee` in the else-path. Instead, `_claimLockedAndDelegate` delegates directly to the attacker provided address.

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L337-L376

```
function claimAndDelegateWithSig(
  bytes16 campaignId,
  bytes32[] memory proof,
  address claimer,
  uint256 claimAmount,
  SignatureParams memory claimSignature,
  address delegatee,
  SignatureParams memory delegationSignature
) external nonReentrant {
  require(delegatee != address(0), '0_delegatee');
  require(!claimed[campaignId][claimer], 'already claimed');
  require(claimSignature.expiry > block.timestamp, 'claim expired');
  address signer = ECDSA.recover(
    _hashTypedDataV4(
      keccak256(
        abi.encode(CLAIM_TYPEHASH, campaignId, claimer, claimAmount, claimSignature.nonce, claimSignature.expiry)
      )
    ),
    claimSignature.v,
    claimSignature.r,
    claimSignature.s
  );
  require(signer == claimer, 'invalid claim signature');
  _useCheckedNonce(claimer, claimSignature.nonce);
  if (campaigns[campaignId].tokenLockup == TokenLockup.Unlocked) {
    _claimUnlockedAndDelegate(
      campaignId,
      proof,
      claimer,
      claimAmount,
      delegatee,
      delegationSignature.nonce,
      delegationSignature.expiry,
      delegationSignature.v,
      delegationSignature.r,
      delegationSignature.s
    );
  } else {
    _claimLockedAndDelegate(campaignId, proof, claimer, claimAmount, delegatee);
  }
```

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L564-L584

```
      IDelegatePlan(c.tokenLocker).delegate(tokenId, delegatee);
      IERC721(c.tokenLocker).transferFrom(address(this), claimer, tokenId);
    } else {
      tokenId = IVestingPlans(c.tokenLocker).createPlan(
        address(this),
        campaign.token,
        claimAmount,
        start,
        c.cliff,
        rate,
        c.period,
        address(this),
        true
      );
      IDelegatePlan(c.tokenLocker).delegate(tokenId, delegatee);
      IERC721(c.tokenLocker).transferFrom(address(this), claimer, tokenId);
      IVestingPlans(c.tokenLocker).changeVestingPlanAdmin(tokenId, _vestingAdmins[campaignId]);
    }
    emit Claimed(claimer, claimAmount);
    emit LockedTokensClaimed(campaignId, claimer, claimAmount, campaigns[campaignId].amount);
  }
```

Recommendation

It is highly recommended, to make `delegatee` part of the claim signature. This clearly communicates the intention for the claim and invalidates the signature for any other purposes. Additionally, if available and applicable, consider verifying the `delegationSignature` in the else-path, too.

6.4 Batch Claiming With Signature Can Be Frontrun Medium ✓ Fixed

Resolution

Addressed in commit 6c747bb5254fe9cfaf4f9009370ba13a951fc88f. The client performed a partial fix:

- The client implemented specific typehashes for each type of claiming, separating single claims, batched claims and delegation claims.
- The client also added to the signed data the `campaignIds.length`.

Both of these measures greatly decrease the possibility of this issue leaving claims unclaimed. However, a similar issue could still happen if a user has several campaigns to claim and doesn't want to claim them all. Since the current signature includes the length of the `campaignIds` provided to the `claimMultipleWithSig`, a malicious user could still front run a transaction that shows the signature in the mempool and execute `claimMultipleWithSig` with it. The signature doesn't validate other campaignIds in the `campaignIds` array provided as an argument, only the length of this array. This gives the attacker the possibility of including/excluding any campaign that the claimer is eligible to claim as long a the `claimIds.length` is the same as the one in the signature, provided the attacker has access to the proofs of each campaign and the correct `claimAmount`.

The circumstances just described are unusual, and this issue does not impact the claimability of the campaigns. Instead, it leads to a scenario where the signature provided by the user may not accurately reflect their intent. The client has acknowledged this risk.

Description

Batch claiming with signatures requires a valid signature for the first claim. The signature values can be read in the mempool and a malicious actor can therefore frontrun the batch claiming with a single claim using the same signature, claiming only that campaign. This would make the batch claim transaction revert due to the nonce increase and would mean the rest of the batched claims would remain unclaimed.

The `claimWithSig` methods are intended to be used by paymaster. A malicious user frontrunning `claimMultipleWithSig` by calling `claimWithSig` with the same signed data is forcing only one claim to go through while the paymaster was submitting multiple. This might lead to a DoS or griefing attack where the attacker intentionally stalls the claiming performed by paymasters.

The following functions all operate on the same signature. By observing a call to either of the function someone can fron-trun and re-use the data to call another.

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L233-L251

```
function claimWithSig(
  bytes16 campaignId,
  bytes32[] calldata proof,
  address claimer,
  uint256 claimAmount,
  SignatureParams memory claimSignature
) external nonReentrant {
  require(!claimed[campaignId][claimer], 'already claimed');
  require(!campaigns[campaignId].delegating, 'must delegate');
  require(claimSignature.expiry > block.timestamp, 'claim expired');
  address signer = ECDSA.recover(
    _hashTypedDataV4(
      keccak256(
        abi.encode(CLAIM_TYPEHASH, campaignId, claimer, claimAmount, claimSignature.nonce, claimSignature.expiry)
      )
    ),
    claimSignature.v,
    claimSignature.r,
    claimSignature.s
```

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L262-L278

```
function claimMultipleWithSig(
  bytes16[] calldata campaignIds,
  bytes32[][] calldata proofs,
  address claimer,
  uint256[] calldata claimAmounts,
  SignatureParams memory claimSignature
) external nonReentrant {
  address signer = ECDSA.recover(
    _hashTypedDataV4(
      keccak256(
        abi.encode(CLAIM_TYPEHASH, campaignIds[0], claimer, claimAmounts[0], claimSignature.nonce, claimSignature.expiry)
      )
    ),
    claimSignature.v,
    claimSignature.r,
    claimSignature.s
  );
```

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L337-L357

```
function claimAndDelegateWithSig(
  bytes16 campaignId,
  bytes32[] memory proof,
  address claimer,
  uint256 claimAmount,
  SignatureParams memory claimSignature,
  address delegatee,
  SignatureParams memory delegationSignature
) external nonReentrant {
  require(delegatee != address(0), '0_delegatee');
  require(!claimed[campaignId][claimer], 'already claimed');
  require(claimSignature.expiry > block.timestamp, 'claim expired');
  address signer = ECDSA.recover(
    _hashTypedDataV4(
      keccak256(
        abi.encode(CLAIM_TYPEHASH, campaignId, claimer, claimAmount, claimSignature.nonce, claimSignature.expiry)
      )
    ),
    claimSignature.v,
    claimSignature.r,
    claimSignature.s
```

Recommendation

Claim signatures should be bound to the function they're intended to be used on. For example, signatures could have a unique identifier added to the hashed elements (i.e. or different TypeHash name) to distinguish batch claiming transactions from single claiming transactions.

6.5 `createCampaign` - `id` Can Be Front-Run Medium Acknowledged

Resolution

The client acknowledged this issue but their overall system requires Id's to be arbitrary. They have proceeded to keep this functionality as it is.

Description

The `id` parameter is not directly bound to any transaction related data. According to inline comments it is "… uuid or CID of the file that stores the merkle tree ". It is unclear how important it is that this uuid only being used for the correct merkle hash.

Since this uuid is not directly bound to the merkle tree (it is not calculated in the contract), an attacker could potentially front-run a call to `create*Campaign`, using the same `id` to deliberately disrupt targeted transactions. This could lead to inconvenience and potential transaction failures.

Examples

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L140-L175

```
/// @notice primary function for creating an locked or vesting claims campaign. This function will pull the amount of tokens in t
/// additionally it will check that the lockup details are valid, and perform an allowance increase to the contract for when toke
/// @dev the merkle tree needs to be pre-generated, so that you can upload the root and the uuid for the function
/// @param id is the uuid or CID of the file that stores the merkle tree
/// @param campaign is the struct of the campaign info, including the total amount tokens to be distributed via claims, and the r
/// @param claimLockup is the struct that defines the characteristics of the lockup for each token claimed.
/// @param vestingAdmin is the address of the vesting admin, which is used for the vesting plans, and is typically the msg.sender
function createLockedCampaign(
  bytes16 id,
  Campaign memory campaign,
  ClaimLockup memory claimLockup,
  address vestingAdmin,
  uint256 totalClaimers
) external nonReentrant {
  require(!usedIds[id], 'in use');
  usedIds[id] = true;
  require(campaign.token != address(0), '0_address');
  require(campaign.manager != address(0), '0_manager');
  require(campaign.amount > 0, '0_amount');
  require(campaign.end > block.timestamp && campaign.end > campaign.start, 'end error');
  require(campaign.tokenLockup != TokenLockup.Unlocked, '!locked');
  if (campaign.delegating) {
    require(IERC20Votes(campaign.token).delegates(address(this)) == address(0), '!erc20votes');
  }
  if (campaign.tokenLockup == TokenLockup.Vesting) {
    require(vestingAdmin != address(0), '0_admin');
    _vestingAdmins[id] = vestingAdmin;
  }
  require(claimLockup.tokenLocker != address(0), 'invalide locker');
  TransferHelper.transferTokens(campaign.token, msg.sender, address(this), campaign.amount);
  claimLockups[id] = claimLockup;
  SafeERC20.safeIncreaseAllowance(IERC20(campaign.token), claimLockup.tokenLocker, campaign.amount);
  campaigns[id] = campaign;
  emit ClaimLockupCreated(id, claimLockup);
  emit CampaignStarted(id, campaign, totalClaimers);
}
```

Recommendation

Bind the `id` to a merkle tree directly by calculating the `id` in the contract or requiring the merkle root to be submitted.

6.6 `createLockedCampaign` Might Create Unclaimable Campaigns Due to Unvalidated `ClaimLockup` (`TimelockLibrary.validateEnd`, `DIV/0`) Minor Partially Addressed

Resolution

Partially fixed in commit `4829924d71e458549b125435f0f025c7304aa390`.

While this commit includes checks for `periods` and `periods`, an unclaimable campaign can still be created. This may happen during the claiming of a locked campaign if the `tokenLocker.createPlan()` is passed incorrect vesting plan arguments and reverts. This would imply that the campaign creator would have to cancel the campaign and create another a new one with valid vesting plan arguments. The client acknowledges this issue.

Description

Not validating `ClaimLockup memory claimLockup` in `createLockedCampaign` might lead to locked campaigns being unclaimable because `ILockupPlans(c.tokenLocker).createPlan(ClaimLockup)` may revert on invalid parameterization. This means, an unclaimable campaign can be created and users will only find out on claim that the `ClaimLockup` is incompatible with `ILockupPlans`.

To recover from this, the campaign manager can cancel the campaign and re-create it with valid parameters.

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L168-L174

```
require(claimLockup.tokenLocker != address(0), 'invalide locker');
TransferHelper.transferTokens(campaign.token, msg.sender, address(this), campaign.amount);
claimLockups[id] = claimLockup;
SafeERC20.safeIncreaseAllowance(IERC20(campaign.token), claimLockup.tokenLocker, campaign.amount);
campaigns[id] = campaign;
emit ClaimLockupCreated(id, claimLockup);
emit CampaignStarted(id, campaign, totalClaimers);
```

- _claimLockedTokens

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L485-L494

```
if (campaign.tokenLockup == TokenLockup.Locked) {
  tokenId = ILockupPlans(c.tokenLocker).createPlan(
    claimer,
    campaign.token,
    claimAmount,
    start,
    c.cliff,
    rate,
    c.period
  );
```

- Example: `TokenVestingPlans.create()`

see TokenVestingPlans

```
function createPlan(
    address recipient,
    address token,
    uint256 amount,
    uint256 start,
    uint256 cliff,
    uint256 rate,
    uint256 period,
    address vestingAdmin,
    bool adminTransferOBO
  ) external nonReentrant returns (uint256 newPlanId) {
    require(recipient != address(0), '0_recipient');
    require(token != address(0), '0_token');
    (uint256 end, bool valid) = TimelockLibrary.validateEnd(start, cliff, amount, rate, period);
    require(valid);
    _planIds.increment();
```

see TokenVestingPlans

```
function validateEnd(
    uint256 start,
    uint256 cliff,
    uint256 amount,
    uint256 rate,
    uint256 period
  ) internal pure returns (uint256 end, bool valid) {
    require(amount > 0, '0_amount');
    require(rate > 0, '0_rate');
    require(rate <= amount, 'rate > amount');
    require(period > 0, '0_period');
    end = (amount % rate == 0) ? (amount / rate) * period + start : ((amount / rate) * period) + period + start;
    require(cliff <= end, 'cliff > end');
    valid = true;
  }
```

Note: An unvalidated `c.periods` may lead to a `DIV/0` assertion.

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L547-L551

```
if (claimAmount % c.periods == 0) {
  rate = claimAmount / c.periods;
} else {
  rate = claimAmount / c.periods + 1;
}
```

Recommendation

Consider validating `ClaimLockup` using `TimelockLibrary` on campaign creating to fail early if the provided parameters are invalid.

6.7 Partial Deletion of the Campaings Minor Partially Addressed

Resolution

The problem has been partially fixed in the `4829924d71e458549b125435f0f025c7304aa390` commit by deleting the `claimLockups` when the campaign is finished. The client acknowledges the remaining `_vestingAdmins` is not deleted.

Description

In the `_claimLockedTokens` and `_claimUnlockedTokens` functions of the `DelegatedClaimCampaigns` contract, when the claimable amount of the campaign is zero, the `campaignId` is deleted from the `campaigns` public mapping, however, the `campaignId` is still present in the `claimLockups` mapping, returning non-zero parameters of the already non-existing `campaignId`. This can be misleading for the integrators, and affect other contracts in the system, leading to unexpected behaviour.

Examples

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L408-L410

```
if (campaigns[campaignId].amount == 0) {
  delete campaigns[campaignId];
}
```

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L473-L475

```
if (campaigns[campaignId].amount == 0) {
  delete campaigns[campaignId];
}
```

Recommendation

We recommend to clean `claimLockups` mapping when the campaign is finished together with the `campaigns` mapping, as well as `_vestingAdmins` mapping.

6.8 `verify()` - Unnecessary Return Value When Verifying Merkle Tree Inclusion Acknowledged

Description

`verify()` returns static `bool true` and reverts on errors. The method cannot return `false`. Therefore, it is unnecesary, to return `true` and wrap calls to `verify()` in. `require()` because the method will never return `false`.

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L586-L596

```
/// @dev the internal verify function from the open zepellin library.
/// this function inputs the root, proof, wallet address of the claimer, and amount of tokens, and then computes the validity of
/// @param root is the root of the merkle tree
/// @param proof is the proof for the specific leaf
/// @param claimer is the address of the claimer used in making the leaf
/// @param amount is the amount of tokens to be claimed, the other piece of data in the leaf
function verify(bytes32 root, bytes32[] memory proof, address claimer, uint256 amount) public pure returns (bool) {
  bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(claimer, amount))));
  require(MerkleProof.verify(proof, root, leaf), 'Invalid proof');
  return true;
}
```

- unnecessary `require()`

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L403

```
require(verify(campaign.root, proof, claimer, claimAmount), '!eligible');
```

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L469

```
require(verify(campaign.root, proof, claimer, claimAmount), '!eligible');
```

../DelegateTokenClaims-905fdc2dc89849ba19e718e4147b4b8306f894f2/contracts/DelegatedClaimCampaigns.sol:L538

```
require(verify(campaign.root, proof, claimer, claimAmount), '!eligible');
```

Recommendation

Remove the `bool` return value from the functions declaration. Revert on error.

7 Findings: VestingLockups

7.1 VotingVault - Revoking Access Automatically When Contract Token Balance Falls to Zero May Leave Tokens Stuck Major ✓ Fixed

Resolution

The problem has been fixed in the `6508b5865654c948b16d8ccd4172bd4ef9570b0a` commit by removing the deletion of the `token` and `controller` addresses.

Description

Note: This finding was also raised by the client in a call during the engagement. We share their view on this pattern being problematic and list the finding for transparency.

When the `VotingVault` token balance falls to zero after `withdrawTokens()` is called, `token` and `controller` are reset to zero, which basically renounces the contracts ability to handle the configured token. Any future tokens will be locked in the contract.

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/periphery/VotingVault.sol:L40-L46

```
function withdrawTokens(address to, uint256 amount) external onlyController {
  TransferHelper.withdrawTokens(token, to, amount);
  if (IERC20(token).balanceOf(address(this)) == 0) {
    delete token;
    delete controller;
  }
}
```

Recommendation

There does not seem to be a need for deleting `token` and `controller`. Controller should still have the ability to withdraw any future tokens even if the contracts balance was zero for an amount of time.

7.2 Tokens Can Get Locked Upon Vesting Plan Transfer Major ✓ Fixed

Resolution

Addressed in commit 972c273f9aaf81f88e43a87d60081f041ce9df52.

If an approved redeemer tries to redeem and the ownership of the vesting NFT is changed to another contract that isn't the lockup, it will set the `totalAmount` to the `availableAmount`. During the `balanceAtTime` function `totalAmount` will therefore be lower than `rate` and it will skip the clause that was causing this issue.

Description

The unlock function calls `balanceAtTime()` to see if there are any unlockable tokens to be transferred to the beneficiary.

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/TokenVestingLock.sol:L417-L433

```
function _unlock(uint256 lockId) internal returns (uint256 unlockedBalance) {
  require(isApprovedRedeemer(lockId, msg.sender), '!approved');
  VestingLock memory lock = _vestingLocks[lockId];
  uint256 lockedBalance;
  uint256 unlockTime;
  (unlockedBalance, lockedBalance, unlockTime) = UnlockLibrary.balanceAtTime(
    lock.start,
    lock.cliff,
    lock.totalAmount,
    lock.availableAmount,
    lock.rate,
    lock.period,
    block.timestamp
  );
  if (unlockedBalance == 0) {
    return 0;
  }
```

To do so it will check if the `availableTokens`, previously redeemed from the vesting contract, are higher than the `rate`. If there are less than `rate` it will not unlock any tokens to the beneficiary. The lockup contract always unlocks tokens in multiples of `rate`, except when the lockup has finished, where it will transfer whatever is available.

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/libraries/UnlockLibrary.sol:L66-L93

```
} else if (availableAmount < rate && totalAmount > rate) {
  // if the available amount is less than the rate, and the total amount is greater than the rate,
  // then it is still mid vesting or unlock stream, and so we cant unlock anything because we need to wait for the available amou
  lockedBalance = availableAmount;
  unlockTime = start;
  unlockedBalance = 0;
} else {
  /// need to make sure clock is set correctly
  uint256 periodsElapsed = (redemptionTime - start) / period;
  uint256 calculatedBalance = periodsElapsed * rate;
  if (totalAmount <= calculatedBalance && availableAmount <= calculatedBalance) {
    /// if the total and the available are less than the calculated amount, then we can redeem the entire available balance
    lockedBalance = 0;
    unlockTime = redemptionTime;
    unlockedBalance = availableAmount;
  } else if (availableAmount < calculatedBalance) {
    // else if the available is less than calculated but total is still more than calculated amount - we are still in the middle
    // so we need to determine the total number of periods we can actually unlock, which is the available amount divided by the r
    uint256 availablePeriods = availableAmount / rate;
    unlockedBalance = availablePeriods * rate;
    lockedBalance = availableAmount - unlockedBalance;
    unlockTime = start + (period * availablePeriods);
  } else {
    // the calculated amount is less than available and total, so we just unlock the calculated amount
    unlockedBalance = calculatedBalance;
    lockedBalance = availableAmount - unlockedBalance;
    unlockTime = start + (period * periodsElapsed);
  }
```

The vestingAdmin from the underlying vesting plan can change the ownership of the vesting NFT, which for a correct lockup set up should be the lockup contract address. If the ownership of the vesting NFT changes, tokens are no longer redeemable, which means `availableBalance` cannot be increased.

This creates an issue where any resulting amount from `availableAmount % rate` will get stuck in the contract, since `balanceAtTime` will always calculate the tokens to unlock as a multiple of `rate`.

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/libraries/UnlockLibrary.sol:L74-L75

```
uint256 periodsElapsed = (redemptionTime - start) / period;
uint256 calculatedBalance = periodsElapsed * rate;
```

Since the `availableAmount` will be lower than the `rate`, and `totalAmount` will be higher than the `rate`, it will permanently trigger the clause previously mentioned where no tokens get unlocked, rendering the remaining `availableAmount` stuck in the contract.

Recommendation

A state where the vesting lockup is no longer redeemable should be considered, to be able to withdraw whatever `availableTokens` there are and therefore burn the `lockId`.

7.3 Selective Out-of-Gas (OOG) May Allow `lockedVesting.owner` to Bypass vestingToken Ownership Check Medium Acknowledged

Description

When the owner of a locked vesting plan on `TokenVestingLock` calls `burnRevokedVesting`, `TokenVestingLock` checks if the

- `hedgeyVesting` vestingPlanId is valid, and the vestingPlanId is still managed by `TokenVestingLock`, in which case it proceeds with burning and deleting traces of it in `TokenVestingLock`
- `hedgeyVesting` vestingPlanId is invalid, indicated by `hedgeyVesting.ownerOf(lock.vestingTokenId)` reverting with `ERC721NonexistentToken(tokenId)`, in which case it proceeds with burning and deleting traces of it in `TokenVestingLock`
- `hedgeyVesting` vestingPlanId is valid, but the vestingPlanId is **not** managed by `TokenVestingLock`, it should revert.

However, there is an unhandled edge-case that can be controlled by the caller, where the tryCatch's error handler fires if the tried call to `hedgeyVesting.ownerOf(lock.vestingTokenId)` runs out of gas. By deliberately limiting gas in a way that the call to `hedgeyVesting.ownerOf(lock.vestingTokenId)` fails, the caller can force the `catch()` block to be executed, bypassing the token ownership check, burning and deleting traces of the token (given, enough gas is left to execute `_burn()`) instead of reverting.

When `hedgeyVesting.ownerOf(lock.vestingTokenId)` reverts, it is not immediately clear if this is because of `ERC721NonexistentToken` or some other condition (OOG). The `catch()` block does not explicitly check for the expected revert reason.

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/TokenVestingLock.sol:L388-L402

```
function burnRevokedVesting(uint256 lockId) external {
  require(msg.sender == ownerOf(lockId), '!owner');
  VestingLock memory lock = _vestingLocks[lockId];
  require(lock.availableAmount == 0);
  try hedgeyVesting.ownerOf(lock.vestingTokenId) {
    require(hedgeyVesting.ownerOf(lock.vestingTokenId) != address(this), '!revoked');
    _burn(lockId);
    delete _vestingLocks[lockId];
    delete _allocatedVestingTokenIds[lock.vestingTokenId];
  } catch {
    _burn(lockId);
    delete _vestingLocks[lockId];
    delete _allocatedVestingTokenIds[lock.vestingTokenId];
  }
}
```

../contracts/token/ERC721/ERC721.sol:L448-L453

```
function _requireOwned(uint256 tokenId) internal view returns (address) {
    address owner = _ownerOf(tokenId);
    if (owner == address(0)) {
        revert ERC721NonexistentToken(tokenId);
    }
    return owner;
```

```
function _requireOwned(uint256 tokenId) internal view returns (address) {
      address owner = _ownerOf(tokenId);
      if (owner == address(0)) {
          revert ERC721NonexistentToken(tokenId);
      }
      return owner;
 }
```

Recommendation

At least, in the `catch` block, check for the expected revert reason `ERC721NonexistentToken`, else revert. In general we strongly advocate against trying to convey business-logic information through errors as this can be problematic and a potentially extremely dangerous anti-pattern in Solidity.

```
 } catch (bytes memory reason /*lowLevelData*/) {
        require(bytes4(reason)==ERC721.ERC721NonexistentToken.selector)
        ...
 }
```

7.4 Lockup Can Get Extended by a Period Medium ✓ Fixed

Resolution

Addressed in commit 5332cc8a7f40c9a94c7283ad33f9290a2dc87a63.

The `unlockTime` is now set to the timestamp relative to the `availablePeriods`.

Description

The lockup can be extended under certain circumstances based on the current `_redeem` and `_unlock` functions. If a user redeems at any point but doesn't unlock, and then after the final period timestamp executes unlock instead of redeemAndUnlock it will extend the lockup for another period.

This is due to a clause in the `unlockLibrary.balanceAtTime()`

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/libraries/UnlockLibrary.sol:L74-L80

```
uint256 periodsElapsed = (redemptionTime - start) / period;
uint256 calculatedBalance = periodsElapsed * rate;
if (totalAmount <= calculatedBalance && availableAmount <= calculatedBalance) {
  /// if the total and the available are less than the calculated amount, then we can redeem the entire available balance
  lockedBalance = 0;
  unlockTime = redemptionTime;
  unlockedBalance = availableAmount;
```

`calculatedBalance` is calculated based on the periods elapsed between the current timestamp and the start, which is the last timestamp the lockup was unlocked. The `calculatedBalance` should only be equal or higher to the `totalAmount` once the lockup has finished. The `start` is set to the `redemptionTime`, which is the `block.timestamp`.

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/TokenVestingLock.sol:L443-L447

```
} else {
  _vestingLocks[lockId].availableAmount = lockedBalance;
  _vestingLocks[lockId].start = unlockTime;
  _vestingLocks[lockId].totalAmount = remainingTotal;
}
```

This causes the issue where if there are still tokens to redeem, the `availableTokens` are equal or higher to `rate` due to a prior redeeming, and `unlock` gets executed after the lockup has finished, the `start` will be set to the current timestamp. This means that for the user to unlock the remaining vestedLocked tokens he would have to redeem them and wait another period.

The reason why a prior redemption is necessary for this attack to happen is due to the following clause requiring the `availableBalance`, which is increased upon redemption, to be higher or equal to `rate`, or else it would simply not change any state:

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/libraries/UnlockLibrary.sol:L66-L71

```
} else if (availableAmount < rate && totalAmount > rate) {
  // if the available amount is less than the rate, and the total amount is greater than the rate,
  // then it is still mid vesting or unlock stream, and so we cant unlock anything because we need to wait for the available amou
  lockedBalance = availableAmount;
  unlockTime = start;
  unlockedBalance = 0;
```

7.5 Use the Calls Return Value in TryCatch Instead of Performing Duplicate Calls Minor ✓ Fixed

Resolution

The problem has been fixed in the `c4ecc967daef440a5347822f7a7d0734905aed82` commit.

Description

TryCatch can make the calls return values available in the try block. This should be preferred instead of performing duplicate calls (atomicity of call and return values)

Examples

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/TokenVestingLock.sol:L462-L474

```
function _redeemVesting(uint256 lockId) internal returns (uint256 balance, uint256 remainder) {
  require(isApprovedRedeemer(lockId, msg.sender), '!approved');
  uint256 vestingId = _vestingLocks[lockId].vestingTokenId;
  require(_allocatedVestingTokenIds[vestingId], 'not allocated');
  try hedgeyVesting.ownerOf(vestingId) {
    require(hedgeyVesting.ownerOf(vestingId) == address(this), '!ownerOfNFT');
  } catch {
    return (0, 0);
  }
  (balance, remainder, ) = hedgeyVesting.planBalanceOf(vestingId, block.timestamp, block.timestamp);
  if (balance == 0) {
    return (0, 0);
  }
```

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/TokenVestingLock.sol:L392-L397

```
try hedgeyVesting.ownerOf(lock.vestingTokenId) {
  require(hedgeyVesting.ownerOf(lock.vestingTokenId) != address(this), '!revoked');
  _burn(lockId);
  delete _vestingLocks[lockId];
  delete _allocatedVestingTokenIds[lock.vestingTokenId];
} catch {
```

Recommendation

Use TryCatch with `returns` keyword when expecting return values.

7.6 BatchCrator Can Be Manipulated to Emit Wrong Event ✓ Fixed

Resolution

The problem has been fixed by creating a whitelist of the `lockupContract` in the `f9c349cdaf40f7c68e26bfbdca850c8ac7370c52` commit.

Description

Both the `createVestingLockupPlans` and the `createVestingLockupPlansWithDelegation` can be manipulated to emit their events while the vestingPlans have been created but the lockup plans haven't.

To do so a malicious actor can create a contract and input it as the `lockupContract` address, where the contract upon `address vestingContract = IVestingLockup(lockupContract).hedgeyVesting()` call will return a valid vesting address. The following execution would create a vesting plan with the vesting contract address, making the malicious contract the beneficiary, but wouldn't do anything in the `createVestingLock` call.

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/periphery/BatchCreator.sol:L329-L345

```
function createVestingLockupPlansWithDelegation(
  address lockupContract,
  address token,
  uint256 totalAmount,
  IVestingLockup.Recipient[] calldata recipients,
  address[] calldata delegatees,
  Plan[] calldata vestingPlans,
  address vestingAdmin,
  bool adminTransferOBO,
  Plan[] calldata locks,
  bool transferablelocks,
  uint8 mintType
) external returns (uint256[] memory, uint256[] memory) {
  require(vestingPlans.length == recipients.length, 'lenError');
  require(vestingPlans.length == locks.length, 'lenError');
  require(totalAmount > 0, '0_totalAmount');
  address vestingContract = IVestingLockup(lockupContract).hedgeyVesting();
```

This would result in the same execution as `createVestingPlans` but emitting a `VestingLockupBatchCreated` event.

7.7 `hedgeyVesting` Should Be Declared as Immutable ✓ Fixed

Resolution

The problem has been fixed in the `f5b9a720ce84173bff41b4aee71e6e221f59a946` commit

Description

`hedgeyVesting` cannot be updated and should therefore be declared `immutable`.

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/TokenVestingLock.sol:L23-L25

```
contract TokenVestingLock is ERC721Delegate, ReentrancyGuard, ERC721Holder {
  /// @dev this is the implementation stored of the specific Hedgey Vesting contract this particular lockup contract is tied to
  IVesting public hedgeyVesting;
```

7.8 Consider Performing a 2-Step Manager Role Transfer Acknowledged

Description

It should be noted that transferring the `manager` role does not require the new manager to accept it. The current manager will lose access as soon as `changeManager` succeeds. This can be risky because if something goes wrong with this call (i.e. human error, wrong address, ...) manager will not be able to perform their duties anymore.

../VestingLockups-06f41767f57b05f77f3c16fc78103aaacb5f557b/contracts/TokenVestingLock.sol:L150-L156

```
/// @notice function to change the admin address
/// @param newManager is the new address for the admin
function changeManager(address newManager) external onlyManager {
  manager = newManager;
  emit ManagerChanged(newManager);
}
```

According to the documentation manager is a MultiSig:

> The multisignature wallet used to manage which BatchCreator contract is currently deployed and linked to the TokenVestingLock contract.

It is assumed that manager is indeed a multi-sig and all participants verify the correctness of the role transfer, however, we would nevertheless recommend implementing a 2-step transfer.

Appendix 1 - Files in Scope

4.2 VestingLockups

06f41767f57b05f77f3c16fc78103aaacb5f557b

File	SHA-1 hash
contracts/DelegatedClaimCampaigns.sol	`e0478dfa311d4850171be3314d921201aaa664d7`
contracts/interfaces/IDelegatePlan.sol	`c5ec93c4cb46a7edda49de7517df5416337623fc`
contracts/interfaces/IERC20Votes.sol	`4ad911b820416d0ea66d19a804b1f4d64efd0414`
contracts/interfaces/ILockupPlans.sol	`ac0c55d92d2f4077b94ecae43e7930b3b40dffd5`
contracts/interfaces/IVestingPlans.sol	`cb8ac9b7fbae2c1e56f04fc1cc501b956fc4945c`
contracts/libraries/TransferHelper.sol	`c1119d0a05dd1d533836fb74943dc5318f882d62`

A.1.2 DelegatedTokenClaims

905fdc2dc89849ba19e718e4147b4b8306f894f2.

File	SHA-1 hash
contracts/ERC721Delegate/ERC721Delegate.sol	`19157685413a4b59b0435c79c03a948f6e003527`
contracts/ERC721Delegate/IERC721Delegate.sol	`29192c077ab71265c95160b2657de33e6b2336e7`
contracts/ERC721Delegate/PlanDelegator.sol	`929396d89511747f576e0e65e0ae7c143326aa7a`
contracts/TokenVestingLock.sol	`1156cef585280e955c09ceb5d23e4c2938d30af9`
contracts/interfaces/IVesting.sol	`8daf8ca80362b6f2282bdac2a43886e84778ebdc`
contracts/interfaces/IVestingLockup.sol	`cd9e5b17f5bf71e65934bc98fa58eb580418f1e2`
contracts/libraries/TransferHelper.sol	`29ca155e5d08edd37f66fdb9ea0bf7828f3a24fd`
contracts/libraries/UnlockLibrary.sol	`4bbe558e3564d3484d0ea507c07666be647dde2e`
contracts/periphery/BatchCreator.sol	`2c5cf28b9186af055b08c30ff25d443ed154f28a`
contracts/periphery/VotingVault.sol	`d626eff22c483eaeaf6d0f219cd543bbccfd037b`

Appendix 2 - Disclosure

Consensys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.