# Digitra Vesting Token

## Smart Contract Audit

coinspect

digitra.com

# Smart Contract Audit

## Summary of Findings

| Id | Title | Risk | Status |
|---|---|---|---|
| **DIG-001** | Vested tokens availability is not guaranteed | High | ✓ |
| **DIG-002** | Vesting schedules will not vest | Medium | ✓ |
| **DIG-003** | Overflow in vesting calculations triggers unclaimable schedules | Low | ⚠ |
| **DIG-004** | Overinflation of the total vesting accumulator | Medium | ✓ |
| **DIG-005** | Tokens could be locked forever due to unclear ownership management | Low | ✓ |
| **DIG-006** | Flat sloped vesting schedules alter the claiming period | Info | ⚠ |
| **DIG-007** | Initial accounts to vest are incorrect | Info | ⚠ |
| **DIG-008** | Use built-in days instead of magic constants | Info | ✓ |
| **DIG-009** | Bad handling of low level calls when transferring and reading token balance | Info | ✓ |

# Executive Summary

In June 2023, Digitra engaged Coinspect to perform a source code review of the Vesting system. The objective of the project was to evaluate the security of the Vesting token along with the main contract that manage the creation and allocation of vested assets.

The following issues were identified during the initial assessment:

| High Risk | Medium Risk | Low Risk |
| --- | --- | --- |
| Open | Open | Open |
| 0 | 0 | 0 |
| Fixed | Fixed | Fixed |
| 1 | 2 | 1 |
| Acknowledged | Acknowledged | Acknowledged |
| 0 | 0 | 1 |
| Reported | Reported | Reported |
| 1 | 2 | 2 |

Coinspect identified one high-risk, two medium-risk and two low-risk issues. The high-risk issue, `DIG-001`, shows how vested tokens availability might not be guaranteed. The first medium-risk issue, `DIG-002` adverts that under the current configurations it is impossible to create new vesting schedules. Lastly, `DIG-004` shows an overinflation scenario of a key global accumulator.

As for the low-risk issues, `DIG-003` shows how an overflow makes unclaimable some type of vesting schedules and `DIG-005` mentions potential risks related to the contract's ownership management plan.

# Assessment and Scope

The audit started on June 12, 2023 and was conducted on the main branch of the git repository located at https://github.com/ZmicerVilenski/Degitra as of commit `ba188f99cb7d2e9d67bfb518134044933faff75b`.

Coinspect reviewed the implemenation of the vesting token along with the main vesting contract. The vesting `Token` is an eight-decimal `mintable/burnable` `ERC20` token that uses the v4.8.3 of Openzeppelin's standard. This token does not expose the `_mint` function and only mints the total supply to the `msg.sender` when it is deployed. It is worth mentioning that the public `burn` and `burnFrom` functions are not used by the `TokenVesting` contract, meaning that users can burn their claimed tokens at anytime without having any particular incentive to do so.

The `TokenVesting` contract enables the creation of vesting schedules where a certain amount of vesting tokens can be allocated to beneficiaries, Coinspect identified that the vested tokens might not be available at all times in the contract leaving some beneficiaries empty-handed as the amounts to transfer by the contract might be greater than the current token balance, `DIG-001`.

It employs a cliff-based system with a linear accumulation of tokens granted on `30 day` periods once the cliff passes. It does not include a delegated claiming system meaning that each beneficiary can only claim the vested tokens for themselves. Vesting schedules can only be created by the owner before the vesting period starts. Coinspect identified that the contract configuration does not allow creating schedules right after its deployment because the `START` constant is outdated, `DIG-002`. Also, the creation of new vesting schedules can trigger overflows when calculating the vested amounts preventing beneficiaries to receive any vested tokens, `DIG-003`.

The vesting contract tracks the total amount to vest in a global accumulator, but also allows the owner to overwrite the vesting schedule of a beneficiary. This process leads to an overinflation of that global accumlator because it is not adjusted considering older values, `DIG-004`. Lastly, the deployment script shows how the token is deployed and allocated to a `NEW_OWNER` account but no ownership is transferred to that account later on, `DIG-005`.

## Detailed Findings

### DIG-001 - Vested tokens availability is not guaranteed

| | |
|---|---|
| Likelihood | **High** |
| Impact | **High** |
| Risk | **High** |
| Resolution | **Fixed** |
| Status | ✓ |
| Location | `TokenVesting.Sol` |

#### Description

Vesting schedules with steeper slopes and amounts leave other schedules empty-handed, preventing them from getting their vested tokens.

It is possible to create a schedule with a steep slope that uses up all the available tokens of the vesting contract, consequently denying other users the possibility to claim their vested tokens. An unaware owner can create multiple vesting schedules exceeding the total amount of available tokens.

This can be achieved because there is no token availability check when creating a new vesting schedule and all tokens to vest are sent when the contract is deployed:

```
    function createVestingSchedule(
        address _beneficiary,
        uint16 _durationDays,
        uint8 _cliffDays,
        uint112 _amountTotal,
        uint112 _amountAfterCliff
    ) external onlyOwner {
        require(
            START > uint32(block.timestamp), "TokenVesting: forbidden
to create a schedule after the start of vesting"
        );
        require(_durationDays > 0, "TokenVesting: duration must be >
0");
        require(_amountTotal > 0, "TokenVesting: amount must be > 0");
        require(_durationDays >= uint16(_cliffDays), "TokenVesting:
duration must be >= cliff");
        // require(
        //     _amountTotal >= _amountAfterCliff,
        //     "TokenVesting: total amount must be >= amount after
cliff"
```

```
        // );
        vestingSchedules[_beneficiary] = VestingSchedule(_cliffDays,
_durationDays, _amountTotal, 0, _amountAfterCliff);
        vestingSchedulesTotalAmount += (_amountTotal +
_amountAfterCliff);
        emit ScheduleCreated(_beneficiary, _durationDays,
_amountTotal);
    }
```

The following proof of concept shows an adversarial scenario where a steep-sloped proposal is created, compromising the availability of the whole vesting process. This scenario assumes that the `NEW_OWNER` sends also his tokens to the vesting contract so all the `totalSupply` is available. Even under those preconditions, vested tokens are made unavailable:

```
    function test_can_compromise_token_availability() public {
        uint256 initialVestingBalance =
vToken.balanceOf(address(vesting));
        uint256 totalvTokenSupply = vToken.totalSupply();
        console2.log("Vesting balance = %s", initialVestingBalance);
        console2.log("Token Total Supply = %s", totalvTokenSupply);
        // The vesting contract can have at most, the totalSupply of
vesting tokens.
        // Assuming that the NEW_OWNER sends all the tokens to the
vesting contract:
        vm.startPrank(NEW_OWNER);
        vToken.transfer(address(vesting), vToken.balanceOf(NEW_OWNER));
        assertEq(
            vToken.balanceOf(address(vesting)),
            totalvTokenSupply,
            "Vesting does not have the total supply"
        );
        vm.stopPrank();
        // The owner creates the schedules
        // durationDays = 100, cliffDays = 10, total = 100e8,
afterCliff = 0
        vm.prank(vesting.owner());
        vesting.createVestingSchedule(SOME_BENEFICIARY, 100, 10,
uint112(10_000e8), 0);
        // After creating all the schedules, submits a draining
schedule
        // durationDays = 10, cliffDays = 10, total = 2 *
totalvTokenSupply, afterCliff = 0
        vm.prank(vesting.owner());
        vesting.createVestingSchedule(
            OTHER_BENEFICIARY, 10, 10, uint112(totalvTokenSupply -
100e8), 0
        );
        // Advance time enabling claims
        vm.warp(1_718_236_800 + 10 days + 30 days + 1); // START + 10
days + 30 days (slice)
        // The draining schedule can be called at any time, denying
other users their vested tokens
        uint256 beforeClaimBalance =
vToken.balanceOf(OTHER_BENEFICIARY);
        vm.prank(OTHER_BENEFICIARY);
        vesting.claim();
```

```
        uint256 afterClaimBalance =
vToken.balanceOf(OTHER_BENEFICIARY);
        console2.log("Tokens claimed by the OWNER = %s",
afterClaimBalance - beforeClaimBalance);
        // Other users are not able to claim, even if they have vested
tokens
        assertGt(vesting.computeReleasableAmount(SOME_BENEFICIARY), 0,
"No vested amount yet");
        vm.prank(SOME_BENEFICIARY);
        vm.expectRevert(bytes("TRANSFER_FAILED"));
        vesting.claim();
    }
```

It is worth mentioning that this scenario could happen with lower vested amounts but larger amount of users. Last users to claim their vested amount could potentially face this issue.

### Recommendation

Instead of sending all tokens upon deployment, add a `transferFrom` call transferring the amounts-to-vest from the owner to the contract when creating new schedules.

### Status

Fixed on commit 7f0385d53ffa9e3e81b71063dce0e8ef1e1fcd33.

The owner now funds each schedule with a `transferFrom` call.

## DIG-002 - Vesting schedules will not vest

| | |
|---|---|
| Likelihood | **High** |
| Impact | **Low** |
| Risk | **Medium** |
| Resolution | **Fixed** |
| Status | ✓ |
| Location | `TokenVesting.Sol` |

### Description

It is impossible to create new vesting schedules because the vesting start date is outdated.

As the tokens to vest are sent upon deployment to the contract and the constant `START` date is in the past, the following check makes it impossible to create new vesting schedules:

```
    uint32 constant START = 1685232000; // Sun May 28 2023 00:00:00
GMT+0000. start time of the vesting period
```

```
    function createVestingSchedule(
        address _beneficiary,
        uint16 _durationDays,
        uint8 _cliffDays,
        uint112 _amountTotal,
        uint112 _amountAfterCliff
    ) external onlyOwner {
        require(
            START > uint32(block.timestamp),
            "TokenVesting: forbidden to create a schedule after the
start of vesting"
        );
        {...}
    }
```

The first `require` statement will always revert for contracts deployed after `Sun May 28 2023 00:00:00 GMT+0000` and current timestamp by the time of this audit exceeded that value (`1686583000`).

### Recommendation

Update the vesting `START`. Alternatively, it could be replaced by an `immutable` variable defined upon deployment.

**Status**

Fixed on commit 7f0385d53ffa9e3e81b71063dce0e8ef1e1fcd33.

The `START` constant was replaced by an immutable variable.

## DIG-003 - Overflow in vesting calculations triggers unclaimable schedules

| | |
|---|---|
| Likelihood | **Low** |
| Impact | **Medium** |
| Risk | **Low** |
| Resolution | **Acknowledged** |
| Status | ⚠ |
| Location | `TokenVesting.Sol` |

### Description

It is possible to create vesting schedules that always revert while claiming due to an overflow in the vesting calculations.

When creating a vesting schedule, the type of the `_durationDays` variable allows passing values up to `type(uint16).max`. Later, this value is used inside `_computeReleasableAmount()` when `claim()` or `computeReleasableAmount()` are called to estimate the vested amount:

```
    function createVestingSchedule(
        address _beneficiary,
        uint16 _durationDays,
        uint8 _cliffDays,
        uint112 _amountTotal,
        uint112 _amountAfterCliff
    ) external onlyOwner;
```

```
    function claim() external nonReentrant {
        VestingSchedule storage vestingSchedule =
vestingSchedules[msg.sender];
        require(vestingSchedule.amountTotal > 0, "TokenVesting: only
investors can claim");
        uint256 vestedAmount =
_computeReleasableAmount(vestingSchedule);
        {...}
    }
```

However, most calculations inside `_computeReleasableAmount()` are made using `uint32` as type:

```
    if (uint32(block.timestamp) < START + cliffDuration) {
        return 0;
    }
    else if (uint32(block.timestamp) >= START +
(uint32(vestingSchedule.durationDays) * 86400)) {
```

```
        return uint256(vestingSchedule.amountTotal +
vestingSchedule.amountAfterCliff - vestingSchedule.released);
    }
```

The condition shown above will be checked only if the cliff time passed, meaning that we are currently in a period where tokens should vest. However, a vesting schedule that triggers an overflow when calculating the second condition could be created denying users the right to claim their vested tokens:

```
    // START was changed to 1718236800 to enable schedule creation
    function test_can_create_unclaimable_schedule_with_overflow()
public {
        // First we show that a benign schedule could be created for a
user
        vm.prank(vesting.owner());
        vesting.createVestingSchedule(SOME_BENEFICIARY, 10, 1, 100e8,
1e8);
        // Create a vesting schedule
        vm.prank(vesting.owner());
        vesting.createVestingSchedule(OTHER_BENEFICIARY, 49_711, 1,
100e8, 1e8); // [uint32(49_711) * 86400] will overflow
        // Advance until claiming is open
        vm.warp(1_718_236_800 + 1 days + 1); // START + 1 days + 1
        // The first user claims
        vm.prank(SOME_BENEFICIARY);
        vesting.claim();
        assertGt(
            vToken.balanceOf(SOME_BENEFICIARY), 0, "Some Beneficiary
did not receive any tokens"
        );
        // The second user claims
        vm.prank(OTHER_BENEFICIARY);
        vm.expectRevert(); // Will revert due to Arithmetic
over/underflow
        vesting.claim();
    }
```

### Recommendation

Ensure that no vesting schedule config parameter can trigger an arithmetic over/underflow in vesting calculations.

### Status

Acknowledged. Behavior mentioned on commit 79223e26e1be5a21e662da7321ab4fb75b641761.

This is considered unlikely because the overflow duration is over 140 years. As no countermeasures will be done, it will still be possible for the owner to create schedules that revert when claiming.

## DIG-004 - Overinflation of the total vesting accumulator

| | |
|---|---|
| Likelihood | **Low** |
| Impact | **High** |
| Risk | **Medium** |
| Resolution | **Fixed** |
| Status | ✓ |
| Location | `TokenVesting.Sol` |

### Description

Creating multiple vesting schedules for the same beneficiary overinflates the `vestingSchedulesTotalAmount` accumulator.

This happens because `createVestingSchedule()` allows passing the same beneficiary multiple times, overwriting the beneficiary's schedule and always increases the `vestingSchedulesTotalAmount`, regardless if a previous schedule was created for that beneficiary:

```
    function createVestingSchedule(
        address _beneficiary,
        uint16 _durationDays,
        uint8 _cliffDays,
        uint112 _amountTotal,
        uint112 _amountAfterCliff
    ) external onlyOwner {
        {...}
        vestingSchedules[_beneficiary] = VestingSchedule(_cliffDays,
_durationDays, _amountTotal, 0, _amountAfterCliff);
        vestingSchedulesTotalAmount += (_amountTotal +
_amountAfterCliff);
        emit ScheduleCreated(_beneficiary, _durationDays,
_amountTotal);
    }
```

This could be abused to trigger a revert or to generate fake return values in `getWithdrawableAmount()` as it uses the `vestingSchedulesTotalAmount` in the right hand of a subtraction:

```
    function getWithdrawableAmount() external view returns (uint256) {
        return _balanceOf(tokenAddress, address(this)) -
vestingSchedulesTotalAmount;
    }
```

The following proof of concept shows how the `vestingSchedulesTotalAmount` could be overinflated:

```
    function test_inflating_vestingScheduleTotalAmount() public {
        vm.prank(vesting.owner());
        vesting.createVestingSchedule(SOME_BENEFICIARY, 120, 0, 100e8,
0);
        // This has not overflown because the balance is greater than
vestingScheduleTotalAmount
        uint256 withdrawableAmount = vesting.getWithdrawableAmount();
        console2.log("Withdrawable amount: %s", withdrawableAmount);
        // Another schedule is created for the same beneficiary
        vm.prank(vesting.owner());
        vesting.createVestingSchedule(SOME_BENEFICIARY, 120, 0,
type(uint112).max, 0);
        // The following call reverts
        vm.expectRevert();
        withdrawableAmount = vesting.getWithdrawableAmount();
        // And the scheduled amounts were overwritten:
        assertEq(
            vesting.getVestingSchedule(SOME_BENEFICIARY).amountTotal,
            type(uint112).max,
            "Vesting amount mismatch"
        );
        // But the vestingSchedulesTotalAmount considers both
totalAmounts
        assertEq(
            vesting.vestingSchedulesTotalAmount(),
            uint256(type(uint112).max) + uint256(100e8),
            "vestingSchedulesTotalAmount mismatch"
        );
    }
```

#### Recommendation

Adjust the `vestingSchedulesTotalAmount` accumulator accordingly when creating a schedule for the same beneficiary multiple times.

#### Status

Fixed on commit 7f0385d53ffa9e3e81b71063dce0e8ef1e1fcd33.

The global accumulator was removed along with the external method.

## DIG-005 - Tokens could be locked forever due to unclear ownership management

| | |
|---|---|
| Likelihood | **Low** |
| Impact | **Low** |
| Risk | **Low** |
| Resolution | **Fixed** |
| Status | ✓ |
| Location | `Deploy.Js` |

### Description

A mistake when managing the Vesting contract's ownership could make impossible to create new vesting schedules, leaving all the tokens locked.

The `NEW_OWNER` receives a considerable amount of tokens however this account is not assigned as the actual owner of the vesting contract.

Upon deployment, a portion of the minted tokens are transferred to a new account:

```
  // TRANSFER_TOKENS
  const newOwner = process.env.NEW_OWNER;
  const amountForOwner = '450000000000000';
  await token.transfer(newOwner, amountForOwner);
  console.log(`Send: ${amountForOwner} tokens to: ${newOwner}`);
```

Also, the remaining minted tokens are sent to the Vesting contract right after its deployment:

```
  // DEPLOY VESTING
  const Vesting = await hre.ethers.getContractFactory("TokenVesting");
  const vesting = await Vesting.deploy(token.address);
  await vesting.deployed();
  console.log(`Vesting deployed to ${vesting.address}`);
  // TRANSFER_TOKENS
  const amountForVesting = '2955000000000000';
  await token.transfer(vesting.address, amountForVesting);
  console.log(`Send: ${amountForVesting} tokens to:
${vesting.address}`);
```

However, there are no further steps in the deployment script transferring ownership of the vesting contract to that `NEW_OWNER` account. This means that if the private key used on the scripts to deploy and manage the contracts is lost, compromised or leaked, the non-vested tokens could remain locked inside the contract forever.

#### Recommendation

Design and execute a clear contract ownership managing plan.

#### Status

Fixed on commit 79223e26e1be5a21e662da7321ab4fb75b641761.

## DIG-006 - Flat sloped vesting schedules alter the claiming period

| | |
|---|---|
| Likelihood | _ |
| Impact | **Recommendation** |
| Risk | **Info** |
| Resolution | **Acknowledged** |
| Status | ⚠ |
| Location | `TokenVesting.Sol` |

### Description

It is possible to create flat sloped vesting schedules for long vesting periods with low amounts.

When the current timestamp is between the cliff time and the schedule duration, a linear model is used to calculate the amount of vested tokens. There are some border scenarios where the slope calculation outputs zero because solidity rounds down when dividing:

```
    function _computeReleasableAmount(VestingSchedule memory
vestingSchedule) internal view returns (uint256) {
        {...}
        else {
            {...}
            uint256 vestedAmount = (vestingSchedule.amountTotal *
uint256(vestedSeconds))
                / ((uint256(vestingSchedule.durationDays) -
uint256(vestingSchedule.cliffDays)) * 86400); // Compute the amount of
tokens that are vested.
            {...}
        }
    }
```

This division could lead to some periods of time between the `cliffDays` and the total `duration` (claiming period) with zero sloped calculations. For example, the following configuration leads to 180 days with zero tokens claimed besides of being in a vesting period:

```
VestingSchedule(_cliffDays = 0, _durationDays = 1440 /* 4 years */ ,
_amountTotal = 7, 0, _amountAfterCliff = 0);
```

With this configuration, the `vestedAmount` will be calculated as follows if `claim()` is called if less than 180 days passed since the vesting started. Assuming that the current timestamp leads to a `vestedSeconds = 180 days` (which would happen if 209 days passed):

```
vestedAmount = (7 * 180 days) / (1440 days) = 7 * 0.125 = 0.875 < 1
vestedAmount = 0 // As Solidity rounds down

vestedAmount = (7 * 210 days) / (1440 days) = 7 * 0.145 = 1.02 > 1
vestedAmount = 1
```

The following proof of concept shows the scenario from before illustrating how a flat vesting happens between the days `0` and `209`:

```
    function test_creates_flat_vesting_schedules() public {
        // The owner creates the schedules
        // durationDays = 4yrs = 1440, cliffDays = 0, total = 100e8,
afterCliff = 0
        vm.prank(vesting.owner());
        vesting.createVestingSchedule(SOME_BENEFICIARY, 1440, 0,
uint112(7), 0);
        // Tries to claim after 209 days
        vm.warp(1_718_236_800 + 209 days + 1);
        vm.prank(SOME_BENEFICIARY);
        vm.expectRevert(bytes("TokenVesting: nothing to claim"));
        vesting.claim();
        // After 210 days, the first token is vested
        vm.warp(1_718_236_800 + 210 days + 1);
        vm.prank(SOME_BENEFICIARY);
        vesting.claim();
    }
```

### Recommendation

Clearly document this behavior about the creation of schedules with long durations with low amounts. Alternatively, don't allow creating schedules with long durations.

### Status

Acknowledged. Behavior mentioned on commit 79223e26e1be5a21e662da7321ab4fb75b641761.

## DIG-007 - Initial accounts to vest are incorrect

| | |
|---|---|
| Likelihood | _ |
| Impact | **Recommendation** |
| Risk | **Info** |
| Resolution | **Acknowledged** |
| Status | ⚠ |
| Location | `Vesting.Yml` |

### Description

The accounts that will be granted with vested tokens in `vesting.yml` have incorrect addresses. This means that if the `fillVesting.js` script is run before modifying the config file, the vested tokens will be locked forever and won't be recoverable.

For example:

```
  - round_name: Private Sale (Seed)
    beneficiary: "0x0000000000000000000000000000000000000002"
    durationDays: 690
    cliffDays: 0
    amountTotal: "6428571" #(7,142,857 - 714,286)
    amountAfterCliff: "714286"
```

### Recommendation

Ensure that the `vesting.yml` is configured with the right parameters before runnning the `fillVesting` script.

### Status

Acknowledged.

## DIG-008 - Use built-in days instead of magic constants

| | |
|---|---|
| Likelihood | – |
| Impact | **Recommendation** |
| Risk | **Info** |
| Resolution | **Fixed** |
| Status | ✓ |
| Location | `TokenVesting.Sol` |

### Description

Several parts of the code use `86400` as a time-based constant when calculating the releaseable amount, for example:

```
uint32 cliffDuration = (uint32(vestingSchedule.cliffDays) * 86400);
```

This constant represent the amount of seconds in a day. Using this constant in different contexts could cause confusion.

### Recommendation

Define constants using the built-in `days` keyword to represent time-based constants.

### Status

Fixed on commit 7f0385d53ffa9e3e81b71063dce0e8ef1e1fcd33.

Built-in time based constants are now used.

## DIG-009 - Bad handling of low level calls when transferring and reading token balance

| | |
|---|---|
| Likelihood | _ |
| Impact | **Recommendation** |
| Risk | **Info** |
| Resolution | **Fixed** |
| Status | ✓ |
| Location | `TokenVesting.Sol` |

### Description

The `_safeTransfer` and `_balanceOf` methods succeed if they call a non-contract account. If the token contract is mistakenly set upon deployment, vesting tokens will be sent to the contract and users will be able to mark their positions as claimed freely without getting anything in exchange.

```
    function _safeTransfer(address _token, address _to, uint256 _value)
internal {
        // Transfer selector
`bytes4(keccak256(bytes('transfer(address,uint256)')))` should be equal
to 0xa9059cbb
        (bool success, bytes memory data) =
_token.call(abi.encodeWithSelector(0xa9059cbb, _to, _value));
        require(success && (data.length == 0 || abi.decode(data,
(bool))), "TRANSFER_FAILED");
    }
```

If the low level call targets a non-contract account, the call will succeed and the returned data length will be zero, thus passing the `require` statement. This scenario is considerably unlikely as the target of both `_safeTransfer` and `_balanceOf` is set upon deployment and never changed.

The following test uses a deployment of the vesting contract where the token address was mistakenly set and does not revert when claiming the vested tokens:

```
    function test_schedules_for_inexistent_tokens_succeed() public {
        assertTrue(vesting.tokenAddress() != address(vToken));
        vm.prank(vesting.owner());
        vesting.createVestingSchedule(SOME_BENEFICIARY, 120, 0,
uint112(10_000e8), 0);
        vm.warp(1_718_236_800 + 30 days + 1);
        vm.prank(SOME_BENEFICIARY);
        vesting.claim();
    }
```

#### Recommendation

Rely on the `data.length` condition only if the target is a deployed contract. Alternatively, use OpenZeppelin's SafeERC20 library.

#### Status

Fixed on commit 7f0385d53ffa9e3e81b71063dce0e8ef1e1fcd33.

The methods performing the low level calls were removed. Token interactions are now made directly through its interface.

## Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any off-chain systems or frontends that communicate with the contracts, nor the general operational security of the organization that developed the code.

# Appendix

## File hashes

```
40a3ed2eae0cdcd183d14f6e8e23dfbc993604760df743ef6f20fe5728a8a4ee  ./contracts/Token.sol
cc44aff23f1c52aa084e5af8315386c9feda92d20c1b5596def57aa73672c2d8  ./contracts/TokenVesting.sol
```