

# Smart Contract Audit

Prepared for Otoco • June 2021

v210623

## 1. Executive Summary

## 2. Assessment

LaunchCurveExponential

LaunchToken

PoolFactory

LaunchPool

## 3. Summary of Findings

## 4. Detailed Findings

OGO-001 - Insufficient access controls

OGO-002 - Launch pool creation DoS

OGO-003 - Insufficient testing

OGO-004 - Reentrancy can produce malformed launch pools

OGO-005 - Unsafe use of arbitrary ERC20 token contracts

OGO-006 - Unchecked transfer in function distributeSharesChunk

OGO-007 - Unverified transfer amount in function stake

OGO-008 - Sponsor has opportunities for cheating

## 5. Disclaimer

# 1. Executive Summary

In **June 2021**, **OtoCo** engaged **Coinspect** to perform a source code review of the OtoGo smart contracts. The objective of the project was to evaluate the security of the smart contracts.

The following issues were identified during the assessment:

High Risk	Medium Risk	Low Risk
1	5	2

As for high risk issues, it has been found that lack of access controls in an initialization function allows anyone to change the pool factory variables and can be used to steal funds (OGO-001).

Regarding medium risk issues, it was found that it is possible to perform a DoS attack to prevent the creation of launch pools (OGO-002). Also, lack of consideration of unusual ERC20 token contracts puts the launch pools at risk (OGO-005, OGO-006, OGO-007). And it must also be noted that the sponsor of a launch pool has opportunities for cheating, partly by design (OGO-008).

The two low risk issues include one for insufficient testing (OGO-003) and another one for a non-critical reentrancy problem (OGO-004).

The Assessment section includes more observations and several additional recommendations.

## Update June 23, 2021

Verification of fixes was conducted on the main branch of the git repository at <https://github.com/otoco-io/OtoGo-SmartContracts> as of commit 56837b31a022ade235e19c301c34a76ecaebbd88 of **June 19, 2021**.

The final version subject to revision was limited to the following Solidity source files, shown here with their sha256sum hash:

```
6736f7e060f543ddf6d8a568e288f656e48377d6b706f3e708ad6caa0c418449 launchcurve.sol
ee6779e2d1b578b7a03742bd646741d5391b5bdb36a886a69c2b854848e6a059 launchpool.sol
c01b86a895d5f0adca9b2c6835e388cc97d56fcc5207cbcd78987f556c951239 launchtoken.sol
4fd6092bdfa8b42f19d535c5ac69c4323b0b894717c699e58d5552eeabd04cd4 Migrations.sol
bd81d0ef4031b570182276d5210d85bb7a21c4d01f4f5aa3d1351df022409893 poolfactory.sol
```

It was found that all findings have been addressed and fixed, except the low-risk issue OGO-003 regarding lack of test coverage reports. OtoCo communicated to Coinspect that coverage reports will be considered for the future.



```
at runMicrotasks (<anonymous>)
at processTicksAndRejections (internal/process/task_queues.js:97:5)
```

The tests do not have very good coverage. It is recommended to extend the tests to cover more cases, in particular for all the issues that have been found during the assessment. Also, all the events emitted by the contracts should be tested, including their arguments. It is also recommended to create test coverage reports, and integrate them in the development cycle to make sure tests provide full coverage (See OGO-003.)

## LaunchCurveExponential

The `LaunchCurveExponential` contract implements the curve that is used to compute the number of shares to be distributed to each investor (depending on the total supply of the launch pool, the current balance of the pool, the stake of each investor and a reducer factor that smoothes the exponentiality of the curve).

## LaunchToken

The `LaunchToken` contract is a copy of OpenZeppelin's ERC20 implementation with the following changes:

- Decimals and total supply are constructor parameters, and the total supply of tokens is minted to `msg.sender` in the constructor;
- Internal functions `_mint` and `_burn` have been removed.

In addition to functions `_mint` and `_burn`, the internal function `_setupDecimals` is also never used and it can also be removed. However, as a coding best practice, it is suggested to **refactor the `LaunchToken` contract to inherit from OpenZeppelin's ERC20 base contract instead of just copying its code.**

## PoolFactory

The `PoolFactory` contract implements a factory that deploys clones of a given "master" `LaunchPool` contract, with parameters specified by the pool *sponsor* (the caller to the function `createLaunchPool` that deploys the pool).

It is important to mention that the **`PoolFactory` contract is not truly decentralized or autonomous**. The contract inherits from `OwnableUpgradeable`, which means that it has an *owner*, and the owner can freely change its code and storage.

The interface `PoolInterface` in the source file `poolfactory.sol` has a function `transferOwnership`, but the actual `LaunchPool` contract does not have this function. It is recommended to remove the `transferOwnership` function from `PoolInterface`.

The event `PoolCreated` that is emitted by the function `createLaunchPool` is never tested. It is recommended to extend the tests to make sure the event `PoolCreated` is emitted when expected and with correct parameters (See OGO-003).

The name of the function `updateTokenContract` is confusing and misleading, because it actually changes the address of the `LaunchPool` contract that is cloned to create new pools (`_poolSource`), not a token contract. It is recommended to rename the function `updateTokenContract` to a self-explanatory name (for example `updatePoolSource`).

The function `initialize` can be called by anyone. This allows any attacker to change `_poolSource` (the `LaunchPool` contract that is cloned to create new pools) and add new curves, bypassing the access restrictions specified in functions `updateTokenContract` and `addCurveSource` (both are `onlyOwner`). It is recommended to use **OpenZeppelin's Initializable base contract and add the initializer modifier to the initialize function** (See OGO-001).

The function `createLaunchPool` uses `cloneDeterministic` from OpenZeppelin's `ClonesUpgradeables` library to make a clone of `_poolSource`:

```
function createLaunchPool (
    address[] memory _allowedTokens,
    uint256[] memory _uintArgs,
    string memory _metadata,
    address _shares,
    uint16 _curve
) external returns (address pool){
    pool = ClonesUpgradeable.cloneDeterministic(_poolSource,
    (keccak256(abi.encodePacked(_metadata))));
    PoolInterface(pool).initialize(_allowedTokens, _uintArgs, _metadata, msg.sender,
    _shares, _curveSources[_curve]);
    emit PoolCreated(_metadata, pool);
}
```

Using the `cloneDeterministic` function with the hash of the metadata as salt guarantees that no two pools will be created with the same metadata. However, the function `updateMetadata` in contract `LaunchPool` allows the sponsor to change the metadata at any time after the pool has been created. It is recommended to either remove the `updateMetadata` function in `LaunchPool`, or use function `clone` instead of `cloneDeterministic`. It must be reconsidered whether the functionality provided by `cloneDeterministic` is needed, i.e. if it is desirable to allow the prediction of pool addresses before they are created.

Also, note that `cloneDeterministic` will fail if it is called twice with the same master address and salt, because it is not possible to deploy two contracts at the same address. This creates a DoS for the function `createLaunchPool`, since any attacker with front-running capabilities could call `createLaunchPool` first and make the legitimate call fail (See OGO-002).

## LaunchPool

The function `extendEndTimestamp` contains a typo, instead of requiring the extension to be less than 1 year it requires it to be less than 356 days:

```
function extendEndTimestamp(uint256 extension)
    external
    onlySponsor
    isStaking
{
    // Prevent extension to be bigger than 1 year, to not allow overflows
    require(extension < 356 days, "Extensions must be small than 1 year");
    _endTimestamp += extension;
}
```

None of the events defined by the contract are tested (`Staked`, `Unstaked`, `Distributed`, `MetadataUpdated`). It is recommended to **extend the tests to make sure events are emitted when expected and with correct parameters**. (See OGO-003). Besides, the event `MetadataUpdated` is never emitted, it should be emitted by the function `updateMetadata` (which is actually recommended to be removed because it conflicts with the use of `cloneDeterministic` in the contract `PoolFactory`).

The word “sponsor” is used ambiguously sometimes to mean “investor” (an address with stakes in the pool), and this makes the code harder to read. It is recommended

to make a stricter use of these words in the code in order to improve readability and in general as a good coding practice.

The function `initialize` makes external calls to arbitrary user-controllable addresses (`_sharesAddress` and `allowedTokens`), allowing reentrancy attacks. This makes possible the creation of malformed pools that do not fulfill all the requirements, for example a pool with more than 3 allowed tokens. It is recommended to **use OpenZeppelin's Initializable base contract and add the `initializer` modifier to the `initialize` function.**

The `LaunchPool` contract makes calls to functions `transfer` and `transferFrom` in arbitrary ERC20 contracts (the “allowed tokens”, specified by the sponsor when the pool is created). But, not all “ERC20” token contracts adhere strictly to the specification, and there are some ambiguities in the specification too. For example, not all “ERC20” contracts revert on transfer failure, some of them just return `false`; and to make things worse, some of them do not return anything at all (only revert on failure). It is recommended to **avoid directly calling `transfer` and `transferFrom` and instead use `safeTransfer` and `safeTransferFrom` from the `SafeERC20` library**(See OGO-005).

The function `distributeSharesChunk` does not check if the call to `transferFrom` was successful:

```
_stake = _stakes[_stakesDistributed];
if (_stake.amount > 0) {
    token.transferFrom(_sponsor, _stake.investor, _stake.shares);
    // Zero amount and shares to not be distribute again same stake
    emit Distributed(
        _stakesDistributed,
        _stake.investor,
        _stake.amount,
        _stake.shares
    );
}
```

If the token contract returns `false` on failure but does not revert, the failure would go unnoticed and investors would lose their shares. Although this is not a problem if the token is a `LaunchToken`, the shares token of a given launch pool can actually be any address (it is freely specified by the sponsor on pool creation). It is recommended to **use the `SafeERC20` library and call `safeTransferFrom` instead of `transferFrom`** (See OGO-006).



The function `stake` does not check if the call to `transferFrom` actually transferred the specified amount. Some “ERC20” contracts such as USDT can discount a fee on every transfer, and the final amount transferred to the destination address can be less than the amount specified in the call to `transferFrom`. This problem can result in some investors losing funds, since calling `unstake` would transfer back to the investor the amount originally specified in the call to `transferFrom` and not the actual amount received by the `LaunchPool` contract. It is recommended to **compute the actual amount transferred as the difference between balances before and after the transfer**. And also, use `SafeERC20’s safeTransferFrom` instead of calling `transferFrom` directly (See OGO-007).


The sponsor can pause and unpause the `LaunchPool` contract. While the `LaunchPool` contract is paused, staking and unstaking are not allowed. This means that a malicious sponsor could call the function `pause` while still in the staking stage (`Initialized`) to prevent investors from unstaking their tokens. It is recommended to **allow unstaking not only in `Initialized` and `Aborted` stages, but also in a `Paused` stage**.

Finally, it must be noted that by design the `LaunchPool` contract does not guarantee that the investors will receive their share. The sponsor has an advantage, because ultimately **the sponsor can decide whether to distribute the promised shares in exchange of the staked tokens, or instead abort the pool, or even leave the staked tokens locked in the pool forever and keep the promised shares for himself**.

### 3. Summary of Findings

ID	Description	Risk	Fixed
OGO-001	Insufficient access controls	High	✓
OGO-002	Launch pool creation DoS	Medium	✓
OGO-003	Insufficient testing	Low	✗
OGO-004	Reentrancy can produce malformed launch pools	Low	✓
OGO-005	Unsafe use of arbitrary ERC20 token contracts	Medium	✓
OGO-006	Unchecked transfer in function distributeSharesChunk	Medium	✓
OGO-007	Unverified transfer amount in function stake	Medium	✓
OGO-008	Sponsor has opportunities for cheating	Medium	✓

## 4. Detailed Findings

OGO-001 Insufficient access controls		
Total Risk <b>High</b>	Impact High	Location poolfactory.sol
Fixed 	Likelihood High	

### Description

The function `initialize` can be called by anyone. This allows any attacker to change `_poolSource` (the `LaunchPool` contract that is cloned to create new pools) and add new curves, bypassing the access restrictions specified in the functions `updateTokenContract` and `addCurveSource` (both are `onlyOwner`).

This allows a variety of attacks. For example, an attacker can change `_poolSource` to a malicious contract that steals all the investors' stakes as well as their shares.


### Recommendation

Use OpenZeppelin's `Initializable` base contract and add the `initializer` modifier to the `initialize` function. Also, reconsider whether to make the contract `OwnableUpgradeable`, or to make it immutable and truly decentralized.

### Status

Fixed. The function `initialize` was modified to call `__Ownable_init` first, preventing double initialization. A test was added to verify that double initialization is not possible.

## OGO-002 Launch pool creation DoS

Total Risk <b>Medium</b>	Impact Medium	Location poolfactory.sol
Fixed 	Likelihood Low	

### Description

The function `createLaunchPool` uses `cloneDeterministic` from OpenZeppelin's `ClonesUpgradeables` library to make a clone of `_poolSource`:

```
function createLaunchPool (
    address[] memory _allowedTokens,
    uint256[] memory _uintArgs,
    string memory _metadata,
    address _shares,
    uint16 _curve
) external returns (address pool){
    pool = ClonesUpgradeable.cloneDeterministic(_poolSource,
    (keccak256(abi.encodePacked(_metadata))));
    PoolInterface(pool).initialize(_allowedTokens, _uintArgs, _metadata, msg.sender,
    _shares, _curveSources[_curve]);
    emit PoolCreated(_metadata, pool);
}
```

Note that `cloneDeterministic` will fail if called twice with the same master address and salt, because it is not possible to deploy two contracts at the same address. This creates a DoS for function `createLaunchPool`, since any attacker with front-running capabilities could call `createLaunchPool` first and make a legitimate call fail.

### Recommendation

It must be reconsidered whether the functionality provided by `cloneDeterministic` is needed, i.e. if it is desired to allow the prediction of pool addresses before they are created. If address prediction is not needed, then the function `clone` could be used instead.

If it is decided to keep using `cloneDeterministic`, the DoS can be avoided by including the address of the sponsor (`msg.sender`) in the salt, and in that case the deployment address would depend on the sponsor address too.

## Status

Fixed by using `clone` instead of `cloneDeterministic`.

## OGO-003 Insufficient testing

Total Risk <b>Low</b>	Impact Low	Location test/
Fixed <b>X</b>	Likelihood Low	

### Description

Based on a review of the test cases, the tests included do not seem to have very good coverage, however coverage reports with exact coverage were not available during this audit.

The events emitted by the contracts are not tested. This includes `Staked`, `Unstaked`, `Distributed`, and `MetadataUpdated` in `LaunchPool`, and `PoolCreated` in `PoolFactory`.

### Recommendation

Implement test coverage reports and integrate them in the development cycle. Make sure to add new tests as needed to keep full coverage.


Extend the tests to make sure all events are emitted when expected and with correct parameters.

### Status

Tests have been extended to include the events previously not tested, and coverage was improved by adding some more tests: 11 tests have been added, to a total of 61 tests, all passing.

However, OtoCo communicated to Coinspect that test coverage reports will not be implemented at the time being due to lack of time and will be considered for the future.

## OGO-004 Reentrancy can produce malformed launch pools

Total Risk <b>Low</b>	Impact Low	Location launchpool.sol
Fixed 	Likelihood Low	

### Description

The function `initialize` makes external calls to arbitrary user-controllable addresses (`_sharesAddress` and `allowedTokens`), and this allows reentrancy attacks. This makes possible the creation of malformed pools that do not fulfill all requirements, for example a pool with more than 3 allowed tokens.

### Recommendation

It is recommended to use OpenZeppelin's `Initializable` base contract and add the `initializer` modifier to the `initialize` function.

### Status

Fixed as per Coinspect recommendation. The contract now uses OpenZeppelin's `Initializable` base contract and the `initializer` modifier was added to the `initialize` function.

## OGO-005 Unsafe use of arbitrary ERC20 token contracts

Total Risk	Impact	Location
<b>Medium</b>	High	launchpool.sol
Fixed	Likelihood	
✓	Low	

### Description

The LaunchPool1 contract makes calls to functions `transfer` and `transferFrom` in arbitrary ERC20 contracts (the shares token and the “allowed tokens” specified by the sponsor on creation of the pool). But not all “ERC20” token contracts adhere strictly to the specification, and there are some ambiguities in the specification too. For example, not all “ERC20” contracts revert on transfer failure, some of them just return `false`; and to make things worse, some of them do not return anything at all (only revert on failure).

Failing to account for these corner cases could break the logic and result in loss of funds, for example if failed transfers pass unnoticed.

### Recommendation

Avoid directly calling `transfer` and `transferFrom` on untrusted contracts, and instead use `safeTransfer` and `safeTransferFrom` from the SafeERC20 library.

### Status

Fixed as per Coinspect recommendation, using `safeTransfer` and `safeTransferFrom` instead of `transfer` and `transferFrom`.



## OGO-006 Unchecked transfer in function distributeSharesChunk

Total Risk	Impact	Location
<b>Medium</b>	High	launchpool.sol
Fixed ✓	Likelihood	
	Low	

### Description

The function `distributeSharesChunk` does not check if the call to `transferFrom` was successful:

```
_stake = _stakes[_stakesDistributed];
if (_stake.amount > 0) {
    token.transferFrom(_sponsor, _stake.investor, _stake.shares);
    // Zero amount and shares to not be distribute again same stake
    emit Distributed(
        _stakesDistributed,
        _stake.investor,
        _stake.amount,
        _stake.shares
    );
}
```

If the token contract returns `false` on failure but does not revert, the failure would go unnoticed and investors would lose their shares.

Although this is not a problem if the token is `LaunchToken`, the shares token of a given launch pool can actually be any address (it is freely specified by the sponsor on pool creation).

### Recommendation

Use the `SafeERC20` library and call `safeTransferFrom` instead of `transferFrom`.

### Status

Fixed following Coinspect recommendations, using `safeTransferFrom` instead of `transferFrom`.

## OGO-007 Unverified transfer amount in function stake

Total Risk	Impact	Location
Medium	High	launchpool.sol
Fixed	Likelihood	
✓	Low	

### Description

The function `stake` does not check if the call to `transferFrom` actually transferred the specified amount. Some “ERC20” contracts such as USDT can discount a fee on every transfer, and the final amount transferred to the destination address can be less than the amount specified in the call to `transferFrom`. This problem can result in some investors losing funds, since calling `unstake` would transfer back to the investor the amount originally specified in the call to `transferFrom` and not the actual amount received by the `LaunchPool` contract.

### Recommendation


Compute the actual amount transferred as the difference between balances before and after the transfer. And also, use `SafeERC20`’s `safeTransferFrom` instead of calling `transferFrom` directly.

### Status

Fixed as per Coinspect recommendation. The function `stake` now uses `safeTransferFrom` instead of `transferFrom`, and it computes the actual amount transferred as the difference of balance before and after the transfer.

There’s a caveat: function `stake` requires the staked amount to be greater or equal to `_stakeAmountMin`, but the verification is made against the original amount received as argument, and not the actual amount computed from the balance. This means that the sponsor has to take into account any fees discounted by the token contract and add them to `_stakeAmountMin`, otherwise the staked amount could be smaller than the amount desired by the sponsor.

## OGO-008 Sponsor has opportunities for cheating

Total Risk <b>Medium</b>	Impact High	Location launchpool.sol
Fixed 	Likelihood Low	

### Description

The sponsor can pause and unpause the LaunchPool contract. While the LaunchPool contract is paused, staking and unstaking are not allowed. This means that a malicious sponsor could call the function pause while still in the staking stage (Initialized) to prevent investors from unstaking their tokens.

Also, by design the LaunchPool contract does not guarantee that the investors will receive their share. The sponsor has an advantage, because ultimately the sponsor can decide whether to distribute the promised shares in exchange of the staked tokens, or instead abort the pool, or even leave the staked tokens locked in the pool forever and keep the promised shares for himself.

### Recommendation

Allow unstaking not only in Initialized and Aborted stages, but also in Paused stage.

Consider alternatives to make the contract less dependent on the sponsor benevolence, or make clear that the contract assumes the investors trust the sponsor.

### Status

Unstaking is now also allowed in the Paused stage.

There is still room for the sponsor to cheat, but OtoCo communicated to Coinspect that the contract is designed assuming the investors trust the sponsor, in which case the possibility of cheating is not a problem.

## 5. 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.