# Flat Curve Smart Contracts

Security audit report

Prepared for Tezsure

February 18, 2022

# Document management

## Revision history

| Version | Date | Version details |
|---|---|---|
| 1.0 | January 28, 2022 | Initial version |
| 1.1 | February 18, 2022 | Review after fixes |

# Table of contents

# Project summary

| | |
|---|---|
| **Name** | Flat Curve Smart Contracts |
| **Source** | **Repository**: *https://github.com/Plenty-DeFi/flat-curve*<br>**Revision**: c8a2cadfb910d8f62468a6d7b9f8e6528dd152b7<br>2ee1490e8c5fa83fc81e9d5ef73feaf0bc07665d |
| **Methods** | Code review<br>Behavioral analysis<br>Unit test coverage analysis<br>Manual penetration testing |

# Coverage and scope of work

The audit focused on an in-depth analysis of the implementation of the smart contracts, including:

- TezToCtez.py
- TokenToToken.py

Out of Scope:

- helper-contracts/token.py

We conducted the audit in accordance with the following criteria:

- Behavioral analysis of smart contract source code
- Checks against our database of vulnerabilities and manual attacks against the contract
- Symbolic analysis of potentially vulnerable areas
- Manual code review and code quality evaluation
- Unit test coverage analysis

The audit was performed using manual code analysis. Once potential vulnerabilities were discovered, manual attacks were performed to check if they could be easily exploited.

# Smart contract overview

Flat Curve Smart Contracts were developed using SmartPy language for the Tezos blockchain. The solution consists of two smart contracts that provide swap functionality for Tez to CTez and Token to Token.

Any user can provide liquidity to the pool. The ratio between currencies is determined by the flat curve algorithm. By providing liquidity to the pool users receive LP tokens. LP tokens are used to withdraw liquidity from the contract.

Swap between tokens supports both FA1.2 and FA2 tokens.

# Executive overview

Apriorit conducted a security assessment of Flat Curve Smart Contracts in January 2022 to evaluate its current state and risk posture, evaluate exposure to known security vulnerabilities, determine potential attack vectors, and check if any can be exploited maliciously.

## Summary of strengths

Building upon the strengths of the available implementation can help better secure it by continuing these good practices. In this case, a number of positive security aspects were readily apparent during the assessment:

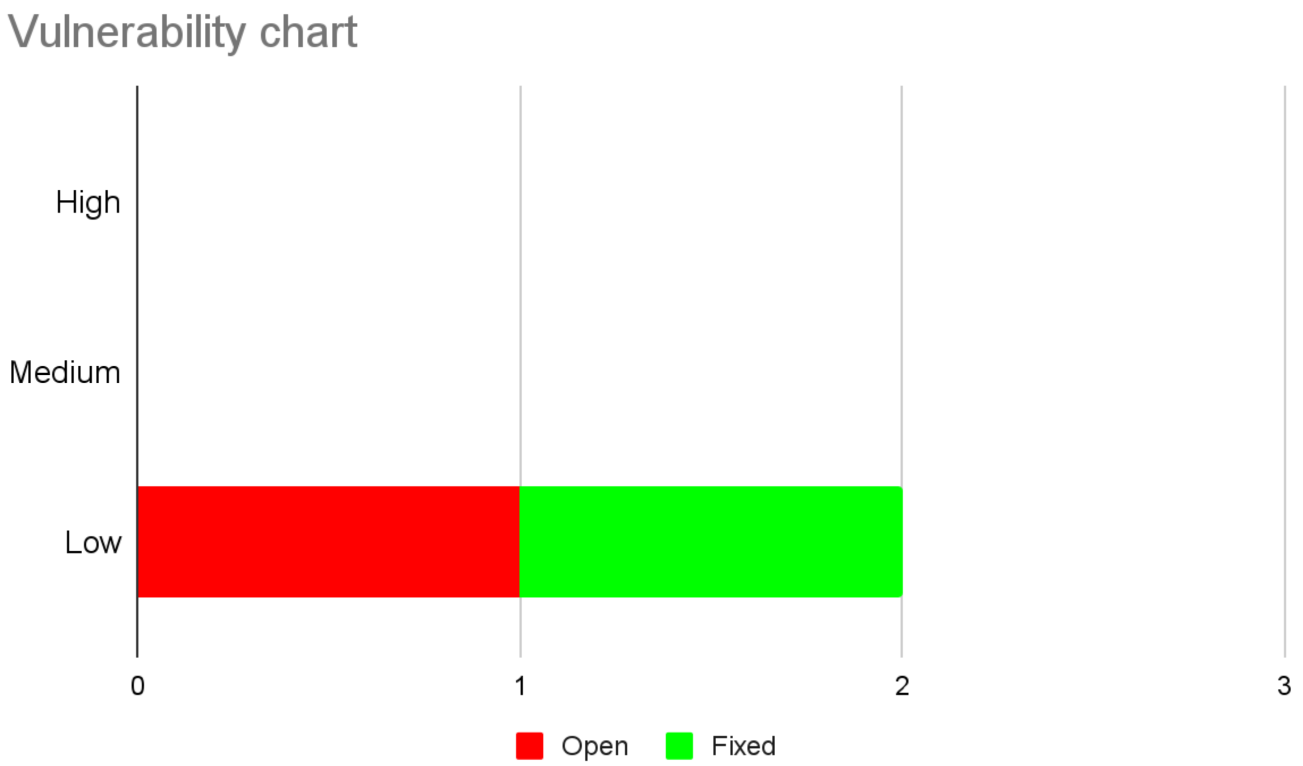
- The code is self-explanatory. The naming policy makes instructions understandable
- The contracts are developed using an up-to-date SmartPy compiler
- Cross-contract interaction is performed securely
- Most verification errors have a custom explanation

## Summary of discovered vulnerabilities

During the assessment, no high-risk or medium-risk vulnerabilities were discovered, indicating good attention to security in the smart contract implementation.

Overall, two low-risk vulnerabilities were discovered, one of which was fixed during the audit. Found vulnerabilities are related to secondary functionality. Financial operations are performed securely.

The chart below shows the distribution of findings.



## Summary of low-risk vulnerabilities and recommendations

For more detailed information on all of the findings, refer to Appendix A: Detailed Findings.

**Table 1: Low-risk vulnerabilities**

| Risk rating | Finding name | Recommendation | Status |
|---|---|---|---|
| Low | Inauthentic admin address | It is recommended to set a pending administrator who will then take control | OPEN |
| Low | Possible lock of the swap functionality | It is recommended to add the ability to release the lock manually | FIXED |

# Security rating

Apriorit reviewed Tezsure security posture in regards to the Flat Curve Smart Contracts, and Apriorit consultants identified security strengths as well as vulnerabilities that create low levels of risk. Taken together, the combination of asset criticality, threat likelihood, and vulnerability severity have been used to assign a grade for the overall security of the application. An explanation of the grading scale is included in the second table below.

In conclusion, Apriorit recommends that Tezsure continue to follow existing good security practices and further improve their security posture by addressing all of the described findings.

| | High | Medium | Low | Security | Grade |
|---|---|---|---|---|---|
| Flat Curve Smart Contracts | 0 | 0 | 1 | Highly Secure | A |

## Security grading criteria

| Grade | Security | Criteria description |
|---|---|---|
| A | Highly secure | Exceptional attention to security. No high- or medium-risk vulnerabilities and few minor low-risk vulnerabilities. |
| B | Moderately secure | Good attention to security. No high-risk vulnerabilities and only a few medium- or several low-risk vulnerabilities. |
| C | Marginally secure | Some attention to security, but security requires improvement. A few high-risk vulnerabilities that can be exploited. |
| D | Insecure | Significant security gaps exist. A large number of high-risk vulnerabilities. |

# Code review and recommendations

Based on years of software development experience, Apriorit had formed a list of best practices to write clear and understandable code. Following these best practices makes maintenance easier.

During the assessment, smart contracts code was compared against our list of best practices. As a result of the code review, we formed the following recommendations.

1. **FIXED Fix typos**

   Typos are a common occurrence in a code that does not affect its functionality but complicates the reading and understanding. It is recommended to check naming and spelling with the help of IDE.

   Affected code:
   - TezToCtez.py, line 253: swaped -> swapped
   - TezToCtez.py, line 255: ther -> the
   - TokenToToken.py, lines 5, 12, 28, 32, 44, 48, 52: reciever ->receiver
   - TokenToToken.py, line 107: precison ->precision
   - TokenToToken.py, line 107: differnce -> difference

2. **FIXED Remove unnecessary code**

   It is recommended to avoid adding instructions that don't do anything or duplicate already performed operations.

   For example, in TokenToToken contract burn() function accepts a "params" record and creates a "burnData" record that is the same.

   Affected code:
   - TezToCtez.py, line 62: cash_transfer() - transferData

- TezToCtez.py, line 67: burn() - burnData
- TezToCtez.py, line 72: mint() - mintData
- TokenToToken .py, line 114: burn() - burnData
- TokenToToken .py, line 119: mint() - mintData

3. **Move common code to a separate file**

Duplication of code makes it harder to support requiring modifications in both copies. The combination of code import and inheritance allows moving common code in a separate file where it will exist in one instance.

Affected code: mint(), burn(), util(), newton(), newton_dx_to_dy, ChangeState(), ChangeAdmin()

Details: https://smartpy.io/docs/general/import/

4. **Use python functions instead of duplicating logic**

If the code uses the same operation of condition multiple times, it is recommended to move duplicated logic in a separate function.

Example:

```
def verifyAdmin(self):
    sp.verify(sp.sender == self.data.admin, ErrorMessages.NotAdmin)

@sp.entry_point
def ChangeState(self):
    self.verifyAdmin()
    …
@sp.entry_point
def ChangeBakerAddress(self, newBakerAddress):
    self.verifyAdmin()
    …
```

5. **FIXED Use simple comparisons**

For the conditions that check values to be greater than 0 prefer “value > 0” semantics instead of “value >= 1”.

Affected code:

- TezToCtez.py, line 208: sp.amount>=sp.mutez(1)
- TezToCtez.py, line 258: params.cashSold>=1
- TokenToToken.py, line 255: params.tokenAmountIn >=sp.nat(1)

### 6. FIXED Limit code lines length

It is more comfortable to read the code if it extends only in one dimension. The presence of a horizontal scroll is an indication that the code lines have grown pretty wide.

It is recommended to follow some limit of line width, for example - 180 symbols.

### 7. PARTIALLY FIXED Use named constants instead of magic numbers

A Magic Number is a hard-coded value that may change at a later stage, but that can be therefore hard to update. It is recommended to use named constants.

In the code, the value "1000" is repeated in both TezToCtez and TokenToToken contracts.

### 8. Use custom errors

Custom errors help in tracking issues and provide the ability to add custom handling in dApp. It is recommended to always use custom errors.

Affected code:

- TezToCtez.py, lines 125, 159, 225-228, 276-279
- TokenToToken .py, lines 62, 182-183, 198-200, 231-232

# Test coverage analysis

Unit tests are an essential part of smart contract development. They help to find problems in the code that are missed by the compiler before deploying the contract to the blockchain.

During the audit, the percentage of unit test coverage for each of the contracts was evaluated. The results are presented in the table below.

| Contract | Initial coverage |
|---|---|
| TezToCtez | 76% |
| TokenToToken | 77% |

There are some test case scenarios for the TezToCtez smart contract, that include only positive cases. During the audit, most of the mentioned test cases were covered.

## TezToCtez contract uncovered test cases

| Function | Description | Status |
|---|---|---|
| add_liquidity | lqtTotal is not 0 and calculated tez deposited is 0 | FIXED |
| | lqtTotal is not 0 and calculated cash deposited is 0 | FIXED |
| | lqtTotal is 0 and calculated lqtMinted is less than 0 | FIXED |
| | calculated minted liquidity is 0 | OPEN |
| | calculated cash deposited is greater than max cash deposited | FIXED |
| | calculated minted liquidity is lower than min minted liquidity | FIXED |
| remove_liquidity | calculated tezWithdrawn is less than minTezWithdrawn | FIXED |

| | | |
|---|---|---|
| | calculated cashWithdrawn is less than minCashWithdrawn | FIXED |
| | lqtBurned is equal to lqtTotal | FIXED |
| | lqtBurned is greater than lqtTotal | FIXED |
| | calculated tezWithdrawn is equal to tezPool | OPEN |
| | calculated tezWithdrawn is greater than tezPool | OPEN |
| | calculated cashWithdrawn is equal to ctezPool | OPEN |
| | calculated cashWithdrawn is greater than ctezPool | OPEN |
| tez_to_ctez | paused contract | FIXED |
| | transferred amount is 0 | FIXED |
| | calculated cashBought is less than MinCash | FIXED |
| | calculated cashBought is equal to ctezPool | OPEN |
| | calculated cashBought is greater than ctezPool | OPEN |
| tez_to_ctez_callback | try to execute directly | FIXED |
| ctez_to_tez | paused contract | FIXED |
| | cashSold is 0 | FIXED |
| | calculated tezBought is less than minAmount | FIXED |
| | calculated tezBought is equal to tezPool | OPEN |
| | calculated tezBought is greater than tezPool | OPEN |
| ctez_to_tez_callback | try to execute directly | FIXED |
| ChangeState | as admin | FIXED |
| | as not admin | FIXED |
| ChangeAdmin | as admin | FIXED |
| | as not admin | FIXED |
| ChangeBakerAddress | as admin | FIXED |
| | as not admin | FIXED |

## TokenToToken contract uncovered test cases

| Function | Description | Status |
|---|---|---|
| add_liquidity | lqtTotal is 0 and invalid ratio | FIXED |
| | lqtTotal is 0 and calculated liquidity is less than 0 | FIXED |
| | lqtTotal is not 0 and calculated token1 amount is 0 | FIXED |
| | lqtTotal is not 0 and calculated token2 amount is 0 | FIXED |
| | lqtTotal is not 0 and ratio token1 to token2 is less than 1 | FIXED |
| | lqtTotal is not 0 and ratio token1 to token2 is greater than 1 | FIXED |
| | lqtTotal is not zero and ratio token1 to token2 is equal to 1 | FIXED |
| | calculated liquidity is 0 | OPEN |
| | token1 amount to transfer is more than maximum | OPEN |
| | token2 amount to transfer is more than maximum | OPEN |
| | transfer FA1.2 token | FIXED |
| | transfer FA2 token | FIXED |
| remove_liquidity | no liquidity in contract | FIXED |
| | insufficient liquidity | FIXED |
| | token1 amount to transfer is less than minimum | FIXED |
| | token2 amount to transfer is less than minimum | FIXED |
| | transfer FA1.2 token | FIXED |
| | transfer FA2 token | FIXED |
| swap | paused | FIXED |
| | zero transfer | OPEN |
| | invalid required pair | FIXED |

| | | |
|---|---|---|
| | required token1 and tokenBought is less than minTokenOut | FIXED |
| | required token1 and tokenBought is equal to token1pool | OPEN |
| | required token1 and tokenBought is more than token1pool | OPEN |
| | required token1 and transfer FA1.2 | FIXED |
| | required token1 and transfer FA2 | FIXED |
| | required token2 and tokenBought is less than minTokenOut | FIXED |
| | required token2 and tokenBought is equal to token2pool | OPEN |
| | required token2 and tokenBought is more than token2pool | OPEN |
| | required token2 and transfer FA1.2 | FIXED |
| | required token2 and transfer FA2 | FIXED |
| ChangeState | as admin | FIXED |
| | as not admin | FIXED |
| ChangeAdmin | as admin | FIXED |
| | as not admin | FIXED |

Also, it is recommended to test all math functions separately, that includes:

- util
- newton
- newton_dx_to_dy
- square_root

# Appendixes

# Appendix A. Detailed findings

## Risk rating

Our risk ratings are based on the same principles as the Common Vulnerability Scoring System. The rating takes into account two parameters: exploitability and impact. Each of these parameters can be rated as high, medium, or low.

**Exploitability** — What knowledge the attacker needs to exploit the system and what preconditions are necessary for the exploit to work:

- **High** — Tools for the exploit are readily available and the exploit requires no specialized system knowledge.
- **Medium** — Tools for the exploit are available but have to be modified. The exploit requires specialized knowledge about the system.
- **Low** — Custom tools must be created for the exploit. In-depth knowledge of the system is required to successfully perform the exploit.

**Impact** — What effect will the vulnerability have on the system if exploited:

- **High** — Administrator-level access and arbitrary code execution or disclosure of sensitive information (private keys, personal information)
- **Medium** — User-level access with no disclosure of sensitive information.
- **Low** — No disclosure of sensitive information. Failure to follow recommended best practices does not result in an immediately visible exploit.

Based on the combination of parameters, an overall risk rating is assigned to a vulnerability.

# Vulnerabilities discovered in the smart contract

## Open issues

Low risk

**Inauthentic admin address**

**Description:**

The administrator performs key functions of the contract. Changing the administrator should be done with special care.

**Affected code:**

- TezToCtez.py, line 306: ChangeAdmin()
- TokenToToken.py, line 287: ChangeAdmin()

**Recommendation:**

It is recommended to set a pending administrator who will then take control. Example:

```
@sp.entry_point
def setPendingAdmin(self, params):
    sp.verify(sp.sender == self.data.admin, ErrorMessages.NotAdmin)
    self.data.pendingAdminAddress = params.adminAddress

@sp.entry_point
def acceptAdmin(self, params):
    sp.verify(sp.sender == self.data.pendingAdminAddress, “INVALID_ACCESS”)
    self.data.admin = self.data.pendingAdminAddress
```

## Closed issues

Low risk

### Possible lock of the swap functionality

**Description:**

To perform a swap between Tez and Ctez the contract interacts with the Ctez admin smart contract to calculate the displacement value which will be returned through the callback. To prevent intermediate function calls, the Locked variable is used. It is set in the way “self.data.Locked = ~ self.data.Locked” to both set and release the lock.

In case, if execution finishes successfully without the Ctez admin sending a callback, the Locked will not allow finishing new swaps.

**Affected code:**

- TezToCtez.py, “Locked” variable

**Recommendation:**

It is recommended to add the ability to release the lock manually. Example:

```
@sp.entry_point
def releaseLock(self, params):
    sp.verify(sp.sender == self.data.admin, ErrorMessages.NotAdmin)
    self.data.Locked = False
```

# Appendix B. Description of methodologies

## Smart contract security checks

Apriorit uses a comprehensive and methodical approach to assess the security of blockchain smart contracts. We take the following steps to find vulnerabilities, expose weaknesses, and identify deviations from accepted best practices in assessed applications. Notes and results from these testing steps are included in the corresponding section of the report.

Our security audit includes the following stages:

1. **Discovery**. The first step is to perform reconnaissance and information gathering to decide how resources can be used in the most secure way. It is important to obtain a thorough understanding of the smart economics, the logic of smart contracts, and the environment they operate within so tests can be targeted appropriately. Within this stage, the following tasks are done:

    a. Identifies technologies

    b. Analyzes the specification, whitepaper, and smart contract source base

    c. Creates a map of relations among smart contracts

    d. Researches the structure of smart contract storage

    e. Researches and analyzes standard implementations for functionality

2. **Configuration Management.** The configuration of the smart contracts is analyzed.

3. **User management and user permissions.** The majority of smart contracts have to manage individual users and their permissions. Most smart contracts split permissions between the contract owner, administrator, etc. Within this stage, the following are done:

   a. Determines whether all functions can be called only by the expected role

   b. Reviews user management functions and role assignment

   c. Reviews permissions for each role

4. **Data validation**. Inputs to a smart contract from users or other smart contracts are its operational life-blood but are also the source of most high-risk attacks. These steps ensure that data provided to the application is treated and checked. All cases of invalid or unexpected data should be handled appropriately.

5. **Efficiency check**. Each function uses some amount of GAS during the call. In the case of a GAS shortage or overlimit, the smart contract function call will fail. Chipper smart contracts will be more interesting for users because no one wants to waste money, so all functions should be optimized in terms of GAS use.

6. **High-quality software development standards**. The standard requires teams to follow the best practices of coding standards. This will help to avoid or mitigate the most common mistakes during development that lead to smart contract security vulnerabilities. It will also help with traceability and root cause analysis. This stage includes:

   a. Manual code review and evaluation of code quality

   b. Unit test coverage analysis

7. **Internal function protection.** Contract vulnerabilities are often introduced due to the semantic gap between the assumptions that contract developers make about the underlying execution semantics and the actual semantics of smart contracts. The internal function should not be callable from the outside because it can lead to inconsistency or unacceptable changes in the storage. Within this stage, the following known vulnerabilities are checked:

   a. The unauthorized contract function call by view callback

   b. The unauthorized intermediate function call.