Plenty DeFi Smart Contracts

Security audit report

Prepared for Tezsure
June 11, 2021

Table of contents

Project summary

<table>
<tr><td>Name</td><td>Plenty DeFi Smart Contracts</td></tr>
<tr><td>Source</td><td><table><tr><th>Repository</th><th>Revision</th></tr><tr><td><em>https://github.com/Tezsure/Plenty-Contracts</em></td><td>branch/Audit - bc50a76
branch/Audit-Review - afd30af</td></tr></table></td></tr>
<tr><td>Methods</td><td>Code review,
Behavioural analysis,
Unit test coverage analysis,
Manual penetration testing</td></tr>
</table>

Coverage and scope of work

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

- PlentyToken - the reward token contract
- Staking - core contract with farm logic
- PlentyAdmin - governance contract

We have conducted the audit under the following criteria:

- Behavioural 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 evaluation of code quality
- Unit test coverage analysis

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

Executive overview

Apriorit conducted a security assessment of Plenty DeFi Smart Contracts to evaluate its current state and risk posture.

This security assessment was conducted in May-June 2021 to evaluate the exposure to known security vulnerabilities, to determine potential attack vectors and to check if any of them can be exploited maliciously.

Summary of strengths

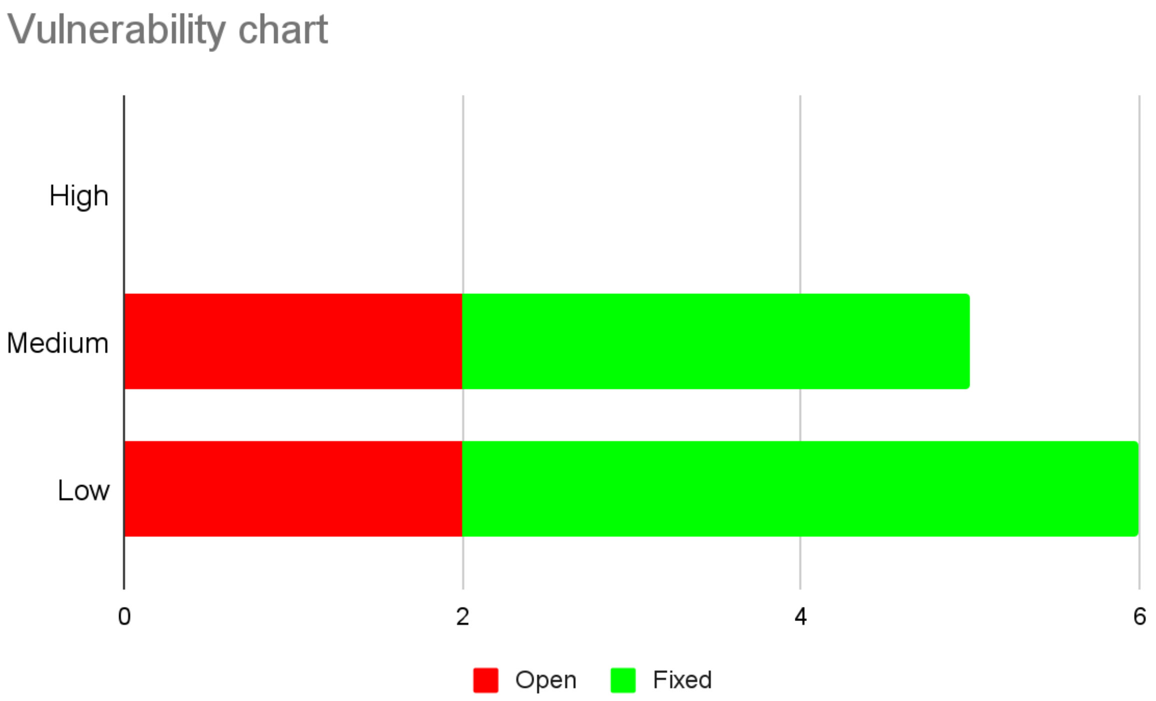
Building upon the strengths of the available implementation can help better secure it by continuing these good practices. In this case, a number of positive security aspects were readily apparent during the assessment:

- The code is self-explanatory. Naming policy makes instructions understandable
- Verification errors has custom explanation
- The contracts perform only the declared functionality
- The main execution scenario has been covered with unit tests
- The contracts are developed using up to date SmartPy compiler
- The contracts are using decentralized governance

Summary of discovered vulnerabilities

During the assessment 5 medium and 6 low vulnerabilities were discovered. 7 (3 medium and 4 low) out of 11 vulnerabilities were fixed during the audit. Medium vulnerabilities related to the excessive admin rights and possible to cause unexpected behaviour. Low vulnerabilities have little to no ability to break the execution flow.

The chart below shows the distribution of findings discovered during the assessment.



Summary of Medium Risk Vulnerabilities and Recommendations

For more detailed information on all of the findings discovered, refer to the detailed findings section (Appendix A: Detailed Findings) of the report.

Risk Rating	Finding Name	Recommendation	Status
Medium risk	Admin has full control over distributed tokens	The “approvals” of the “from” account is not checked if transfer is initiated by the admin. It is recommended to limit admin access to the users balances	OPEN
Medium risk	Voting does not verify parameters of the operation	Admin can set any parameter after voting is done. It is recommended to implement pending parameters	OPEN
Medium risk	Admin can transfer any token through RecoverExcessToken entry point	It is recommended to limit admin access to the staking token	FIXED
Medium risk	Mint can be skipped for some duration	The ModifyParameters() function changes the level of the last update without token minting. It is recommended to perform mint() inside ModifyParameters()	FIXED
Medium risk	Admin can remove account at any time	Operation protection by voting would not work if admin has the ability to freely remove any account. It is recommended to add voting protection for AdminOperation()	FIXED

Summary of Low Risk Vulnerabilities and Recommendations

For more detailed information on all of the findings discovered, refer to the detailed findings section (Appendix A: Detailed Findings) of the report.

Risk Rating	Finding Name	Recommendation	Status
Low risk	Wrong verification of duration	The function sp.as_nat() throws an error only if the input is negative. To confirm that result is not equal to zero, consider adding sp.verify() or sp.if check	OPEN
Low risk	Variable "Difference" is always zero	When GetReward() is called it executes the UpdateReward() function. Therefore, the values of lastTimeReward and lastUpdateTime would be the same. It is recommended to remove excessive logic	OPEN
Low risk	Mint and burn are enabled for the paused contract	It is recommended to review whether this operations should be available in the paused contract	FIXED
Low risk	UpdateReward may operate different balances	UpdateReward function accepts the address parameter to access the account balance, but one call uses sp.sender instead. It is recommended to always use address variable to access account balance	FIXED
Low risk	Verification of MapKey is excessive	On unstake() call, the verification "MapKey >= 0" is performed. MapKey is a sp.TNat variable that cannot be less than zero. It is recommended to remove excessive logic	FIXED

Low risk	GetReward, AddReward, and unstake are active for the paused contract	It is recommended to review whether this operations should be available in the paused contract	FIXED

Security rating

Apriorit reviewed Tezsure security posture in regards to the Plenty DeFi Smart Contracts, and Apriorit consultants identified security practices that are strengths as well as vulnerabilities that create medium and low risks. Taken together, the combination of asset criticality, threat likelihood and vulnerability severity have been combined to assign an assessment 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 continues to follow good security practices that are already established and further improves security posture by addressing all of the described findings.

	High	Medium	Low	Security	Grade
Plenty DeFi Smart Contracts	0	2	2	Moderately Secure	B

Security grading criteria

Grade	Security	Criteria Description
A	Highly Secure	Exceptional attention to security. No high or medium risk vulnerabilities with a few minor low risk vulnerabilities.
B	Moderately Secure	Good attention to security. No high risk vulnerabilities with 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 were identified and can be exploited.
D	Insecure	Significant gaps in security exist. A large number of high risk vulnerabilities were identified during the assessment.

Code review and recommendations

Through the years of software development, we have formed a list of best practices to write a clear and understandable code. Following these recommendations make maintenance easier.

During the assessment we compared the code against our list of best practices. As a result of the code review, 8 recommendations were formed.

1. Provide in-code description for storage and external entry points

The readability of the code consists of two parts: self-explanatory code and documentation. While the first part was performed noticeably high, we found the lack of human readable documentation. The existence of documentation has a positive effect on long-term project support. It can exist in the form of a separate document or as in-code commentary. Here is an example of how in-code commentaries may look like:

```
“””
Transfers the given amount of StakeToken from sender address and add them to the
   staking balance of the sender
dev: updates sender’s reward before modifications
params: TRecord
         amount: TNat - the amount of StakeToken to stake
“””
@sp.entry_point
def stake(self, params):
...
```

2. Use scientific notation for large numbers

Scientific notation allows to express large numbers in more readable form and also prevents common mistakes with missing or excessive zeros.

We recommend to use scientific notation for large numbers:

```
DECIMALS = int(1e18)
```

Instead of:

```
DECIMALS = 1000000000000000000 # 18 decimals
```

3. Import code from another file instead of duplication

Duplication of code makes it harder to support requiring modifications in both copies. SmartPy allows to import the code from another file or external resource with the following command:

```
AnotherContract = sp.io.import_script_from_url(“file:AnotherContract.py”)
```

We recommend keeping the contracts separated and to follow the logic one file - one contract.

Also, try to prevent constans duplication between multiple files. Consider allocating such constants in one place and importing them in contract files.

4. Use inheritance to provide utils functions to different contracts

It is common practice that some general functions appear in multiple contracts. With the usage of SmartPy inheritance mechanism we can prevent duplication of such code. Example:

```
class TransferTokens(sp.Contract):
    def TransferFATokens(self, sender, ...):
        ...
    def TransferFATwoTokens(self, sender, ...):
        ...
class Staking(TransferTokens):
    ...
class PlentyAdmin(TransferTokens):
    ...
```

5. Move unit tests to separate files

Clean code practices state that unit tests should not be coupled with the source code structure. Consider separating tests and contracts code into separate files and place those files into the "tests" folder.

6. Reuse the code as python function instead of duplication

The usage of python functions in SmartPy contracts makes them more readable without altering the logic. Also it simplifies the modification of the code.

For example, the PlentyAdmin contract repeatedly uses the same code to validate that operation is allowed, and it can be moved to a separate function:

```
def validateOperationAllowed(self, op):
    sp.verify(sp.sender == self.data.admin)
    sp.verify(self.data.Operation == op)
    sp.verify(sp.len(self.data.Accounts.elements()) == sp.len(self.data.Votes.keys()))
def FarmRecoverTokens(self, params):
    self.validateOperationAllowed(sp.nat(5))
    ...
def FarmDepositFeeChange(self, params):
    self.validateOperationAllowed(sp.nat(6))
    ...
```

7. Declare constants for operations instead of raw numbers

Named constants improves readability of the code and makes it easier to maintain. Example:

```
class Operation():
    ChangeAdmin = sp.nat(7)
    ChangeState = sp.nat(8)
...
def FarmChangeState(self, FarmAddress):
    verify(self.data.Operation == Operation.ChangeState)
```

8. Use alias for repeatedly used complex types

The syntax of SmartPy entry point requires to specify input parameters type in the function itself and also when creating the handle to this entry point. The usage of aliases allows to simplify this process. Without aliases the modification of parameters in one place would cause error only in runtime (through execution on blockchain or with unit tests), while with them it would be a compile error. Example:

```
TSomeVals = sp.TRecord(val1=sp.TNat, val2=sp.TNat, ...)
...
@sp.entry_point
def someEntryPoint(self, params)
        sp.set_type(params, TSomeVals)
...
@sp.entry_point
def enotherEntryPoint(self, params)
        ...
        fnHandle = sp.contract(TSomeVals, contrAddress, “someEntryPoint”)
```

Test coverage analysis

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

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

Contract	Initial coverage	Final coverage
PlentyToken	60%	85%
Staking	50%	84%
PlentyAdmin	55%	55%

The main flow of execution was fairly covered, but it is recommended to also cover exceptional scenarios.

PlentyToken contract uncovered test cases

Function	Description	Status
mint	Check balance after mint test	FIXED
mint	Try to mint not as administrator	FIXED
mint	Exceed max supply	OPEN
mint	Check that block level updated after successful mint	FIXED
ModifyParameters	Call mint() and check balance after second call	OPEN

	to ModifyParameters skipping few block levels	
burn	Test successful burn and check account balance and total supply	FIXED
burn	Try with insufficient balance	FIXED
transfer	Try to transfer from self address with insufficient balance	OPEN
transfer	Check admin tries to transfer without approval	OPEN
setAdministrator	Try as admin	FIXED
setAdministrator	Try as not admin	FIXED

Staking contract uncovered test cases

Function	Description	Status
GetReward	Try GetReward() again after it was received	FIXED
GetReward	Test behaviour when called before periodFinish	OPEN
stake	Test when there is no reward	FIXED
unstake	Test behaviour when called before periodFinish	OPEN
unstake	Test with different depositFee depending on the cycle	OPEN
unstake	Withdraw part of the stake	OPEN
AddReward	Add reward while previous reward is active	FIXED
changeAdmin	Try as admin	FIXED
changeAdmin	Try as not admin	FIXED

changeState	Try as admin	FIXED
changeState	Try as not admin	FIXED
WithdrawFee	Try as not admin	FIXED

PlentyAdmin contract uncovered test cases

Function	Description	Status
deposit	Add test with expected behaviour	OPEN
FarmDeposit	Try with insufficient balance	OPEN
AdminOperation	Try add the same account again	OPEN
AdminOperation	Try to remove existent account	OPEN
AdminOperation	Try to remove non existent account	OPEN
AdminOperation	Remove account during voting who didn’t vote	OPEN
AdminOperation	Remove account during voting who did vote	OPEN
WithdrawFunds	Try with insufficient balance	OPEN
WithdrawFunds	Check that balances is correct after withdrawing	OPEN
deposit_callback	Try to call manually	OPEN
VoteOperation	Try as non existent account	OPEN
VoteOperation	Try to vote for incorrect operation	OPEN
SetOperation	Try as non existent account	OPEN
SetOperation	Try to set operation when voting is in progress	OPEN
AdminChange	Try with insufficient votes count	OPEN

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 pre-conditions are necessary for the exploit to work:

- **High** - Tools for the exploit are readily available and the exploit requires no specialized knowledge about the system.
- **Medium** - Tools for the exploit available but have to be modified. The exploit requires some 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 the vulnerability will 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 that does not result in an immediately visible exploit.

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

Smart contracts discovered vulnerabilities

Open issues

Medium risk Admin has full control over distributed tokens

Description:

During the transfer() the contract performs complex verification whether this operation is valid. The corresponding sp.verify() check is constructed in such a way that it will not check “approvals” if the transfer() is called by the admin.

Affected code:

PlentyToken.py: FA12_core::transfer()

```
sp.verify(self.is_administrator(sp.sender) |
          (~self.is_paused() &
             ((params.from_ == sp.sender) |
              (self.data.balances[params.from_].approvals[sp.sender] >= params.value))),
                 FA12_Error.NotAllowed)
```

Recommendation:

The “approvals” of the “from” account isn't checked if transfer is initiated by the admin. It is recommended to limit admin access to the users balances

Medium risk Voting does not verify parameters of the operation

Description:

Admin can set any parameter after voting is done. With pending parameters, voters would see the upcoming changes.

Affected code:

AdminContract.py: PlentyAdmin::SetOperation()

Recommendation:

It is recommended to implement pending parameters

Low risk Wrong verification of duration

Description:

The function sp.as_nat() throws an error only if the input is negative. The value of lastUpdate variable cannot logically be bigger than current block level.

This mistake does not invalidate further operations, as the result will remain unchanged.

By adding sp.if check, the excessive calculation can be skipped. If sp.verify() would be added, the repeated call to the mint() function in the same block would fail the transaction.

Affected code:

PlentyToken.py: FA12_mint_burn::mint()

```
duration = sp.local('duration',sp.as_nat(sp.level - self.data.lastUpdate, message =
                    "Contract Call in the same block height"))
```

Recommendation:

To confirm that the result is not equal to zero consider adding sp.verify() or sp.if check.

Low risk Variable "Difference" is always zero

Description:

When GetReward() is called it executes the UpdateReward() function. Therefore the values of lastTimeReward and lastUpdateTime would be the same.

Affected code:

StakingV4.py: Staking::GetReward()

```
Difference = sp.local('Difference', sp.as_nat(lastTimeReward.value
                                             - self.data.lastUpdateTime))
```

Recommendation:

It is recommended to remove excessive logic.

Issues fixed during audit

Fixed Medium risk Admin can transfer any token through RecoverExcessToken entry point

Description:

RecoverExcessToken() function allows to transfer any token from the contract balance, including those that are staked by users. User balances should be secured and not accessible without their permission.

Affected code:

StakingV4.py: Staking::RecoverExcessToken()

Recommendation:

It is recommended to limit admin access to the staking token.

Fixed Medium risk Mint can be skipped for some duration

Description:

The ModifyParameters() function changes the level of the last update without token minting. Last update level is used for determining how many tokens should be minted.

It is better to avoid the dependence from off-chain logic.

Affected code:

PlentyToken.py: FA12_mint_burn::ModifyParameters()

Recommendation:

It is recommended to perform mint() inside ModifyParameters()

Fixed Medium risk Admin can remove account at any time

Description:

The idea of decentralized governance would fail if one person can change the list of participants at any time.

Operation protection by voting would not work if admin has the ability to freely remove any account.

Affected code:

AdminContract.py: PlentyAdmin::AdminOperation()

Recommendation:

It is recommended to add voting protection for AdminOperation()

Fixed Low risk Mint and burn are enabled for the paused contract

Description:

Usually, paused contracts are not accessible for user interaction.

Affected code:

PlentyToken.py: FA12_mint_burn::mint(), FA12_mint_burn::burn()

Recommendation:

It is recommended to review whether these operations should be available in the paused contract.

Fixed Low risk UpdateReward may operate different balances

Description:

UpdateReward function accepts the address parameter to access the account balance, but one call uses sp.sender instead.

In the current codebase, the scenarios when address variable and sp.sender are not the same is protected by additional sp.if check.

Affected code:

StakingV4.py: Staking::UpdateReward()

```
self.data.balances[address].rewards += (self.data.balances[address].balance
            * sp.as_nat( self.data.rewardPerTokenStored
            - self.data.balances[sp.sender].userRewardPerTokenPaid) )
```

```
              / abs(DECIMAL)
```

Recommendation:

It is recommended to always use the address variable to access account balance.

Fixed Low risk Verification of MapKey is excessive

Description:

On unstake() call, the verification "MapKey >= 0" is performed. MapKey is a sp.TNat variable that cannot be less than zero.

Also, the error message is incorrect.

Affected code:

StakingV4.py: Staking::unstake()

```
sp.verify(params.MapKey >= 0 , message = "Cannot Stake Amount Less than 1")
```

Recommendation:

It is recommended to remove excessive logic

Fixed Low risk GetReward, AddReward, and unstake are active for the paused contract

Description:

Usually, paused contracts are not accessible for user interaction.

Affected code:

StakingV4.py: Staking::GetReward(), Staking::AddReward(), Staking::unstake()

Recommendation:

It is recommended to review whether these operations should be available in the paused contract.

Appendix B. Methodologies description

Smart contracts security checks

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.

Our security checklist for smart contract includes:

- Integer Overflow
- Reentrancy
- Race Conditions
- Unchecked External Call
- Unprotected Function
- Short Address Attack
- Multiple sends in a single transaction
- Delegatecall or callcode to untrusted contract
- Timestamp dependence
- DoS with (Unexpected) revert
- Storage Allocation Exploits
- DoS with Block Gas Limit
- Custom ABI-encoded arrays as input
- Underflow Storage Manipulation
- Combinations of vulnerabilities
- GAS usage analysis