apriorit

Plenty Swap Smart Contracts

Security audit report

Prepared for Tezsure

August 17, 2021

Table of contents

Project summary

Name	Plenty Swap Smart Contracts
Source	<table><tr><th>Repository</th><th>Revision</th></tr><tr><td>*https://github.com/Plenty-DeFi/PlentySwap*</td><td>*1bad1f1af8ebb8933a423bb681b2a0e40e5fb299* *a57b919336143b81b5b078971b7dd0d7fceb63b9*</td></tr></table>
Methods	Code review, Behavioural analysis, Unit test coverage analysis, Manual penetration testing

Coverage and scope of work

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

- SwapV3
- MultiSig
- Registry
- xPlenty

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
- Gas usage 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 Swap Smart Contracts to evaluate its current state and risk posture.

This security assessment was conducted in July-August 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:

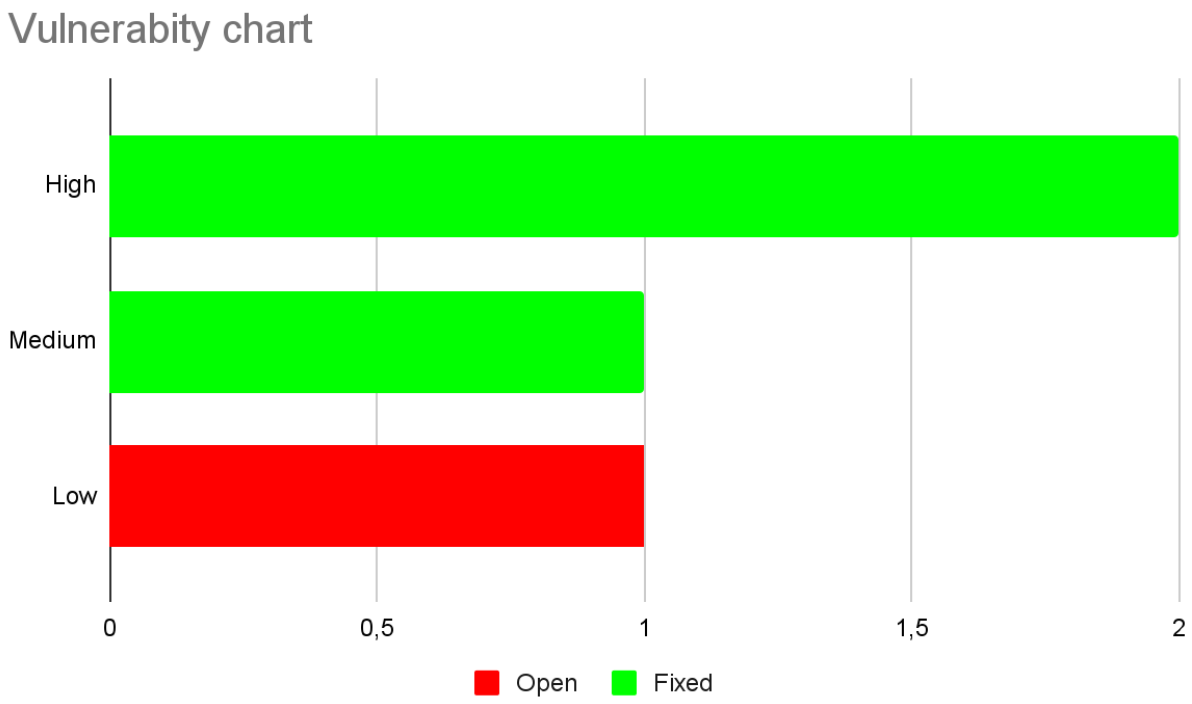
- The code and project files are well structured, which makes it easy to read and maintain. The code is self-explanatory. Naming policy makes instructions understandable
- Verification errors have custom explanation
- The expected behavior of the uniswapV2 protocol has been met
- The main execution scenario has been covered with unit tests
- The contracts are developed using up to date SmartPy compiler
- A non-standard approach to multisig logic is used which is more reliable and credible because it significantly decentralizes governance.

Summary of discovered vulnerabilities

During the assessment 2 high, 1 medium and 1 low vulnerabilities were discovered. 3 (2 high and 1 medium) out of 4 vulnerabilities were fixed during the audit.

High vulnerability significantly changes the behavior of the contract bypassing the norms of the system. 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 High Risk Vulnerabilities and Recommendations

The recommendations below should be implemented as soon as possible since they provide effective and efficient mitigation for many of the high risk issues identified. For more detailed information on all of the findings discovered, refer to the detailed findings section (Appendix A: Detailed Findings) of the technical report.

Risk Rating	Finding Name	Recommendation	Status
High risk	Removal of a fraud voter can be blocked	It is recommended to change the architecture of voting. Please note that this change has a large impact on the solution architecture	FIXED
High risk	There is a possibility to cancel voting during a repeated calling of “RemoveAccount()” function	It is recommended to change the architecture of voting. Please note that this change has a large impact on the solution architecture	FIXED

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	Inauthentic admin address	It is recommended that a new admin must confirm his role using the “acceptGovernance” entrypoint	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	Centralized distribution of rewards in xPlenty.py	It is recommended to change the approach of rewarding from floating to fixed percentage that does not depend on the administrator, or remove reward for liquidity providers	OPEN

Security rating

Apriorit reviewed security posture of Plenty Swap smart contracts, and Apriorit consultants identified security practices that are strengths as well as vulnerabilities that create high, 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 Swap 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 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, the following recommendations were formed:

FIXED 1. Typos in code

There were some typos in the names of variables, comments, etc. All of them were fixed during the audit. The links to the place in code where they can be found are listed below:

simulations/multisigTewsting.py:
39 Treausty -> Treasury

simulations/README.md:
53 Reseting -> Resetting

simulations/registry.py:
3 Registy -> Registry

simulations/Swap.py:
136 Exceding -> Exceeding

multisig.py:
28 specfig -> specific
63 vaulContract -> vaultContract
70 liquidty -> liquidity
145, 146, 301 recieve -> receive

SwapV3.py:

49,54,65,105,121,125 reciever -> receiver

55,57,87,111 transfered -> transferred

105, 121,125 reciver -> receiver

171,172 acumulated -> accumulated

173,487 liquidty -> liquidity

204,286,376 recipent -> recipient

204 recieve -> receive

xPlenty.py:

41,47,57 reviever -> receiver

48,50,80,106 transfered -> transferred

98,114,118 reciver -> receiver

135 recipent -> recipient

137,197 deposting depositing

163,261 revieve -> receive

164,263,356 recieved -> received

182 Funtion -> Function

275,304,306,308 Accured -> Accrued

360 recieving -> receiving

2. Names of some files do not correspond to the contract name

The name of the following files does not correspond to the name of the contract, which causes dissonance:

- xPlenty.py SwapContract SwapContract
- SwapV3.py AMM Exchange

3. 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:

```
TPair = sp.TRecord(token1 =sp.Address, token1Id=sp.TNat, ...)

@sp.entry_point
def AddPair(self, params)
        sp.set_type(params, TPair )

...
@sp.entry_point
def RemovePair(self, params)
        sp.set_type(params, TPair )
```

4. Consider Tezos gas restrictions

Unfortunately, not all errors can be caught at the unit testing stage, even with 100% coverage. A fairly common problem of smart contracts is excessive gas consumption. At the moment, this error cannot be detected by means of SmartPy - it can be tracked after the deployment of the contract and the subsequent invoking of the "heavy" entry point.

The Tezos entrypoints make the Plenty Swap smart contracts more granular for their users, but internally each smart contract is still one inextricable unit. If a smart contract has n-entrypoints of approximately equal size, Tezos will load up to n-times more code than it will be interpreted. That might be quite inefficient for large number of calls.

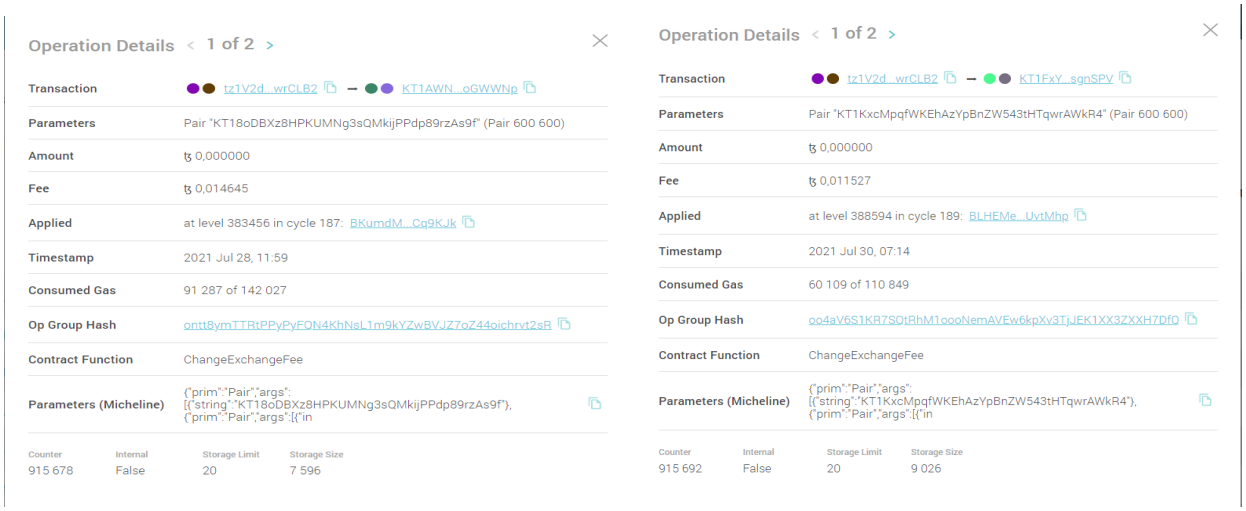
In this case, one of the ways to decrease gas consumption is to split the smart

contract into smaller parts and load only the parts necessary for interpretation. The idea is to put the smart contract code into big_maps as lambdas. SmartPy high-level language provides such a mechanism in a simplified form without sacrificing the readability of the code. SmartPy can be requested to "lazify" an entry point:

```
@sp.entry_point(lazify=True)
```

It is not necessary to put all entry points into the storage. This approach can be applied only to some parts of the smart contract. The bigger an entrypoint is, the more gas is saved by putting it into lazy storage. If there is a tiny entrypoint, it may be useless to put it into lazy storage due to the additional cost of accessing big_map.

According to our observations, “lazification” reduces gas consumption by 2-1.5 times on average. This difference is demonstrated on two screenshots below: the first one is a current version of the Multisig contract, and the second one is the version with lazification.



Entrypoints which can be lazyfied are mentioned below:

multisig.py:

- changeExchangeState

- changeExchangeFee
- changeExchangeMaxSwap
- changeExchangeAdmin
- changeVaultAddress
- ChangeAdmin
- AddAccount
- RemoveAccount
- WithdrawExchangeFee
- ResetVoting

Registry.py

- AddPair

SwapV3.py

- Swap
- AddLiquidity
- RemoveLiquidity
- WithdrawSystemFee

xPlenty.py

- buy
- buy_callback
- sell
- sell_callback

5. xPlenty smart contract can mint and burn tokens

Description:

Mint and burn functions in token.py have logic to validate the caller of the functions. These functions can be called only by the exchanges. It means that the exchange can manipulate the total amount of token issuing. Every user can receive

an unlimited amount of tokens or lose all tokens just because the exchange can do anything. Such tokens have less trust from the community.

Recommendation:
We do not find any vulnerabilities related to this logic but to have more control it is recommended to use a fixed amount of xPlenty tokens and remove mint and burn functions in the token.py. Also, it is recommended to implement pools of xPlenty tokens.

Test coverage analysis

Unit tests are the 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 below:

Contract	Initial coverage	Final coverage
PlentySwapAdmin	81 %	100 %
AMM	40 %	100 %
Registry	83 %	100 %
SwapContract	33 %	66 %

The main flow of execution was fairly covered, but it is recommended to cover exceptional scenarios. It is also recommended to add a check FA2 in unit tests for SwapV3.py and xPlenty.py.
The uncovered functions are listed below.

PlentySwapAdmin contract uncovered functions (multisig.py)

Function	Description	Status
WithdrawExchangeFee	Function to withdraw Fee from a particular AMM	FIXED
ResetVoting	Function to reset voting operation	FIXED

AMM contract uncovered functions (SwapV3.py)

Function	Description	Status
ModifyFee	Admin function to modify the LP and System Fees	FIXED
ChangeState	Admin function to toggle contract state	FIXED
ChangeAdmin	Admin function to Update Admin Address	FIXED
ModifyMaxSwapAmount	Admin function to modify the max swap limit	FIXED
getReserveBalance	View function to get the current AMM Liquidity reserve	FIXED
getExchangeFee	View function to get the Fee Percentage for Liquidity Providers and System Fee	FIXED

Registry contract uncovered functions (Registry.py)

Function	Description	Status
UpdateAdminAddress	Change governance function	FIXED

SwapContract contract uncovered functions (xPlenty.py)

Function	Description	Status
buy_callback	Callback function from Plenty Token Contract for buying xPlenty by depositing plenty tokens	OPEN
sell_callback	Function for Selling xPlenty Token to get back Plenty Tokens	OPEN
ChangeState	Admin Function to Change the State of the Contract	FIXED
RecoverExcessToken	Admin function to Recover any other Token received by the Contract	FIXED

This contract implies 2 different types of tokens: one has a modified interface (with exchangeAddress), and the other one has a standard interface. There are no unit tests to cover different scenarios of exchange of these tokens, so it is recommended to add it.

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

Low risk Centralized distribution of rewards in xPlenty.py

Description:

xPlenty.py allows exchanging tokens, but the reward that the user can get directly depends on the administrator's action. This logic is implemented inside the sell_callback entrypoint:

```
plentyAccured.value = ( self.data.senderAmount.open_some() * PlentyBalance ) / self.data.totalSupply
```

PlentyBalance is the total balance of the contract's plenty tokens which can be increased in case someone calls "buy" entrypoint or if someone transfers tokens to SwapContract address (for example, an administrator).

When users sell their xPlenty tokens, they not only get their initial plenty tokens back, but also have to get the percentage of fee depending on their share of totalSupply. But this fee depends more on the administrator's action, rather than on percentage of their share. So, the reward will be received only when the administrator transfers the plenty tokens to the SwapContract address. Then, the reward will be received mainly by the first user who knows about this action. This approach casts doubt on the fairness of the rewards distribution.

Affected code:

xPlenty.py: def sell_callback(self,PlentyBalance)

Recommendation:

It is recommended to change the approach of rewarding from floating to fixed percentage that does not depend on the administrator, or remove reward for liquidity providers.

Issues fixed during audit

FIXED High risk Removal of fraud voter can be blocked

Description:

Multisig.py file contains the logic of adding and removing voters. Voters can vote for changing exchange fee, exchange admin, adding and removing voters and many other operations. Any voter can block almost any operation, even the operation of removing him from the voters list.

To remove a voter, ⅔ of all voters should vote for it. But any voter can call the ResetVoting function to remove the current voting state. It means that the frauder can block all attempts to remove him from the voters list.

Affected code:

Multisig.py: def RemoveAccount(self,address)

Recommendation:

It is recommended to change the architecture of voting. Please note that this change has a large impact on the solution architecture.

FIXED High risk There is a possibility to cancel voting during a repeated calling of “RemoveAccount()” function

Description:

In Multisig.py, it is possible to repeatedly call the RemoveAccount function and pass an address which differs from the initial one. This will cause a reset of the voting.

For example, Alice (initiator) wants to remove Bob (potential frauder) from the voters list.

1. Alice calls RemoveAccount (Bob address) and expects ⅔ of the votes.
2. Any other account (including Bob) can call the RemoveAccount (Alice address).

In this case, ResetParams () will be called in the code:

```
@ sp.entry_point
def RemoveAccount (self, address):
    ...
  sp.set_type (address, sp.TAddress)
  self.ValidateAccount ()
  sp.if self.data.Operation == Operation.NoOperation:
    self.data.Operation = Operation.RemoveAccount
    self.data.tempAddress = sp.some (address)
    self.data.Votes [sp.sender] = True
  sp.else:
    self.VotingState (Operation.RemoveAccount)
    sp.if ~ self.data.Votes.contains (sp.sender):
      sp.if address == self.data.tempAddress.open_some ():
        self.data.Votes [sp.sender] = True
      sp.else:
        self.ResetParams ()   <<----------
```

As a result, the voting is interrupted and Bob stays in the voters list.

Affected code:

Multisig.py: def RemoveAccount(self,address)

Recommendation:

It is recommended to change the architecture of voting. Please note that this change has a large impact on the solution architecture.

FIXED Medium risk Inauthentic admin address

Description:

If the current admin address is changed to an inauthentic address, the control over the smart contract will be lost. In the current implementation this situation can be avoided only if the multisig smart contract is used.

Affected code:

SwapV3.py: def ChangeAdmin(self,adminAddress)

Recommendation:

To change the admin, it is recommended to set a new admin address with a pending status (by calling setPendingGovernance). To finalize the operation, the new admin should accept a new role (by calling acceptGovernance).

A possible way to resolve this issue is to implement the code similar to the Governance class (see below) and inherit SwapContract from it. The code for it looks as follows:

```
class Errors:
  def make(s): return ("Governance_" + s)

  GOV_NOT_SET_PENDING_ADMIN      = make("GOV_NOT_SET_PENDING_ADMIN")
  GOV_NOT_PENDING_ADMIN          = make("GOV_NOT_PENDING_ADMIN")

class Governance(sp.Contract):
  """
    Sets a new pending governance for the governor

    dev: Governance function to set a new governance

    params: TAddress - The address of the new pending governance
  """
  @sp.entry_point
  def setPendingGovernance(self, pendingAdminAddress):
    sp.set_type(pendingAdminAddress, sp.TAddress)
    self.verifyAdministrator()
    self.data.pendingAdministrator = sp.some(pendingAdminAddress)


  """
    Accept a new governance for the Governor

    params: TUnit
  """
  @sp.entry_point
  def acceptGovernance(self, unusedArg):
    sp.set_type(unusedArg, sp.TUnit)
```

```
    sp.verify(sp.sender ==
self.data.pendingAdministrator.open_some(Errors.GOV_NOT_SET_PENDING_ADMIN),
Errors.GOV_NOT_PENDING_ADMIN)
    self.data.administrator = self.data.pendingAdministrator.open_some()
    self.data.pendingAdministrator = sp.none
```

Appendix B. Methodologies description

Smart contracts security checks

Apriorit uses a comprehensive and methodical approach to assess the security of the blockchain smart contracts. The following steps are taken to find vulnerabilities, expose weaknesses and identify deviations from the accepted best practices in the 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, so as to decide how resources can be used in the most secure way. It is important to obtain a thorough understanding of the smart economics, 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. Identify technologies
    b. Analyze the specification, whitepaper, source base of smart contracts
    c. Create a map of relations among the smart contracts.
    d. Research structure of the smart contracts storage.
    e. Research and analyze standard implementations for the 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 of the smart contracts split users permissions for contracts owner, administrator etc. Within this stage, the following tasks are done:
    a. Determine whether all functions can be called only by expected role.

b. Review the user management functions and role assignment
c. Review permissions for each role

4. **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. Internal function should not be available for call from the outside because it can lead to the inconsistency or inacceptable changes of 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

5. **Data Validation**. Inputs to the smart contracts from users or other smart contracts are its operational life-blood, but are also the source of the 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.

6. **Efficiency check**. Each function uses some amount of gas during the call. In case of gas shortage or overlimit, the smart contract function call will be failed. Chipper smart contracts will be more interesting for users because no one wants to waste money. So all the functions should be optimized regarding the gas usage.

7. **High-quality software development standard support**. The standard requires the teams to follow the best practices of coding standards. This will help to avoid or mitigate against the most common mistakes during development that will lead to security vulnerabilities in the smart contracts. 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.