PolyTrade

Lender Portal

Smart Contract Audit Report

December 20, 2021

IMMUNE BYTES

Introduction

1. About PolyTrade

Polytrade is a decentralized trade finance platform that aims to transform receivables financing. It will connect buyers, sellers, insurers, and investors for a seamless receivables financing experience and help users avoid existing market challenges using its platform solutions. Polytrade will provide real-world borrowers access to low interest and swift financing to free up critical working capital tapped from crypto lenders.

By onboarding on Polytrade, everybody gains because the platform bridges the gaps in traditional receivables financing by accessing untapped crypto liquidity. Through Polytrade, we want to boost business growth where liquidity is not a hindrance.

Visit https://polytrade.finance/ to know more about.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 105+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

The PolyTrade team has provided the following doc for the purpose of audit:

1. https://github.com/polytrade-finance/lender-portal-contracts/tree/dev/docs
2. https://polytrade.finance/whitepaper.pdf
3. https://polytrade.finance/one-pager.pdf

IMMUNE BYTES

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: PolyTrade
- Contracts Name: LenderPool.sol
- Languages: Solidity(Smart contract), Typescript (Unit Testing)
- Github commits for initial audit: 75497f0a0371e7b274eb06e1a94eaaba9062aca1
- Github commits for final audit: 461a043e7b6c95f6f2aab71bd00d6f7d708513f5
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level References

Every issue in this report was assigned a severity level from the following:

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	-	-	-
Closed	1	3	4

High Severity Issues

1. **_swapExactTokens() function is susceptible to Sandwich transaction attack.**
   The **_swapExactTokens()** function in the **LenderPool** contract is intended to swap **stablecoin** to **trade** tokens. However, while performing that trade the function does not make use of the **amountOutMin** input variable of **swapExactTokensForTokens**() function of the Uniswap Router contract.
   Due to using **0** as **amountOutMin**, the withdraw transaction for a lender is susceptible to Sandwich Attack. In this attack, a user's trade transaction is both front and back runned by an attacker to gain profit from the user's trade.
   More details - https://ethresear.ch/t/improving-front-running-resistance-of-x-y-k-market-makers

   Recommendation:
   Consider taking **amountOutMin** as input in the **withdraw()** function and pass it to Uniswap's swap function.

   Amended (Dec 20th, 2021): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

Medium Severity Issues

1. **withdraw() function works for invalid input values.**
   The **withdraw**() function of the LenderPool contract is intended to be used to claim and transfer the lender's rewards as well as the principal amount. Since the function is access protected it does not include input validation checks.
   For a valid lender address, any **roundId** input value can be passed to the function. The function accepts non-existent as well as already claimed **roundId** values to be passed to it. The **_lenderRounds** mapping always returns a default (0) value for non-existent **roundIds**.
   In the case of invalid inputs, the **Round.amountLent** value comes out to be **0**. As the **amountLent** value comes out to be 0, this unintended execution of the function does not cause any loss of funds. But still, this type of unintended execution should be explicitly prohibited by the LenderPool contract.

   ***Recommendation*:**
   Consider validating input variables for the function. Also, a require statement can be added in the withdraw function similar to:

   require (round.amountLent > 0, "No amount lent");

   Amended (Dec 20th, 2021): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

2. **No function is present to pull out extra reward tokens.**
   It is assumed that the LenderPool contracts always possess a sufficient amount of reward tokens so that they can be distributed to lenders. We assume that the PolyTrade team will be topping up the LenderPool contract with reward tokens periodically.

   However, there could be a scenario in which after distributing all the lending rewards to all lenders, the LenderPool contract may still possess some extra reward tokens. In this case, it won't be easy for the PolyTrade team to pull out those extra tokens which are still held by the LenderPool contract.

   ***Recommendation*:**
   Consider either implementing logic to pull out extra tokens or only transfer the exact amount of reward tokens at a time to LenderPool.

   Amended (Dec 20th, 2021): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

3. **Incorrect use of Pausable contract.**
   The LenderPool contract inherits the **Pausable** contract to include the pausability feature. LenderPool also implements a **whenNotPaused** modifier to the **newRound**() function.
   However, there is no mechanism present inside the LenderPool contract to pause and unpause the contract. The internal **_pause**() and **_unpause**() functions of the **Pausable** smart contract have to be explicitly called by inheriting the LenderPool contract to access the pausability feature.
   Also, as the **newRound**() function is only callable by the owner of the contract, it is unnecessary to have the pause and unpause feature. Removing the Pausable contract will also reduce the bytecode size of the LenderPool contract.

   ***Recommendation*:**
   Consider not inheriting the Pausable contract as it is unnecessary.

   Amended (Dec 20th, 2021): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

Low Severity Issues

1. **Repeated ERC20.approve() external calls.**
   The **_claimRewards**() function in LenderPool contract performs a **stableInstance.approve()** call for router contract with an infinite amount.

```
function _claimRewards(address lender, uint roundId) private {
    Round memory round = _lenderRounds[lender][roundId];
    stableInstance.approve(address(router), ~uint(0));

    /// ... more code here
}
```

   Since the LenderPool contract approves the router with an infinite (Max uint256) amount, there is no need to give this approval on every claim transaction. Removing this extra call will result in gas cost reduction for the claim transaction.

   Recommendation:
   Consider approving only once in the **constructor**() of the LenderPool contract.

   Amended (Dec 20th, 2021): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

2. **mappings can be combined into one struct.**
   The **_amountLent** and **_roundCount** mappings in the LenderPool contract are used to store lender-specific data. Since both of these are always used together, they can be combined into s struct. This change will make the contract's storage more optimized.

   Amended (Dec 20th, 2021): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

3. **Smaller-sized uint variables can be used.**
   The **uint256 startPeriod** and **endPeriod** variables in the **Round** struct datatype are intended to store timestamps. Storing a timestamp does not need a huge storage slot.
   Hence these timestamps can be safely stored in a **uint32** or a **uint64** datatype instead of a **uint256** datatype. This change will make the contract's storage more optimized.

Amended (Dec 20th, 2021): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

4. **No function is present to read the _stableAPY variable.**
   The **_stableAPY uint** variable in the LenderPool contract is declared as a **private** variable. So this variable is not accessible from outside the LenderPool smart contract.
   The contract also doesn't contain any **view** function to read the **_stableAPY** value. It will be difficult for external entities (like frontends) to read the **_stableAPY** value directly from the contract.

   Recommendation:
   Consider implementing a function to read the **_stableAPY** value from the smart contract.

   Amended (Dec 20th, 2021): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

Recommendation / Informational

1. **No Events emitted after imperative State Variable modification**
   Line no -57-59

   Functions that update an imperative arithmetic state variable contract should emit an event after the update.
   The **setMinimumDeposit()** function in the contract updates a crucial state variable, i.e., **minimumDeposit** but doesn't emit any event on its modification.

   The absence of event emission for important state variables update also makes it difficult to track them off-chain as well.

   Recommendation:
   As per the best practices in smart contract development, an event should be fired after changing crucial arithmetic state variables.

   Amended (Dec 20th, 2021): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

2. **_getFinishedRounds() function could be optimized and redesigned effectively**

   The _getFinishedRounds() function in the contract includes repetitive code as it involves a similar IF statement at two different instances within the function body.

```
339     function _getFinishedRounds(address lender)
340         private
341         view
342         returns (uint[] memory)
343     {
344         uint length = _roundCount[lender];
345         uint j = 0;
346         for (uint i = 0; i < length; i++) {
347             if (
348                 block.timestamp >= _lenderRounds[lender][i].endPeriod &&
349                 _lenderRounds[lender][i].amountLent > 0
350             ) {
351                 j++;
352             }
353         }
```

While this affects the readability of the function, it also makes it comparatively less gas efficient.

Recommendation:
The function can be redesigned to avoid code repetition as well as optimize gas usage.

Notes/Important Pointers

1. **The owner can deposit any amount from any lender's wallet.**
   The **newRound()** function in the LenderPool contract is used for depositing stablecoins from the user's wallet to the LenderPool contract. The owner has the right to deposit any amount from a user's wallet. The deposit amount is capped by the user's stablecoin balance or user's approval limit to the LenderPool contract, whichever is lesser.

   The users of the LenderPool smart contract must not approve more than the amount that they want to deposit to the LenderPool contract. Also, due to the kind of implementation of the **newRound**() function, the Frontend interacting with the LenderPool contract must not ask for an *Infinite Amount* approval from its users.

   Acknowledged (Dec 20th, 2021): The issue has been acknowledged by the Polytrade Finance team.

Unit Tests

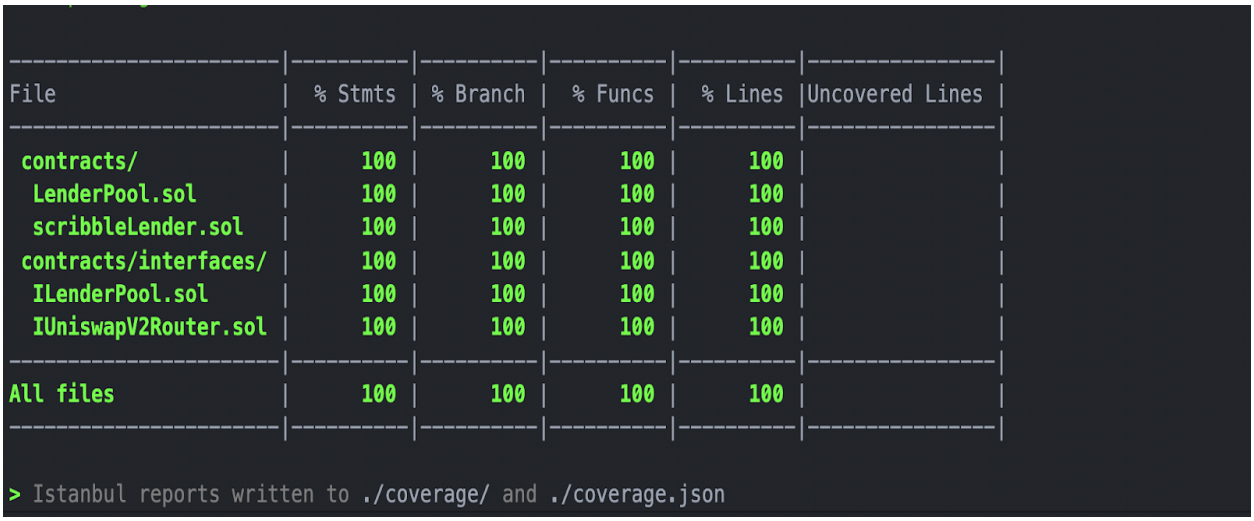
```
  LenderPool - Multiple Rounds
    ✓ Should return the Trade Token (633ms)
    ✓ Should return the USDC Token (610ms)
15900.80889
31776.32742
47626.615909
63451.734485
79251.743083
95026.70145
110776.669142
126501.705528
142201.869787
157877.220912
    ✓ Should buy USDC on quickswap (10972ms)
    ✓ Should distribute USDC to 10 different addresses (12423ms)
    LenderPool - 1 - StableAPY: 10%, USDC, minDeposit: 100 USDC
      ✓ Should return the LenderPool once it's deployed (1785ms)
      ✓ Should set the minimum deposit to 100 USDC
      ✓ Should approve MAX UINT for user1 (609ms)
      ✓ Should fail running a new round (0) with lower amount
      ✓ Should run new round (0)
      ✓ Should return the number of rounds after new round (0)
      ✓ Should return the total amount lent (0)
      ✓ Should run new round (1)
      ✓ Should return the number of rounds after new round (1)
      ✓ Should return the total amount lent (1)
      ✓ Should run new round (2)
      ✓ Should return the number of rounds after new round (2)
      ✓ Should return the total amount lent (2)
      ✓ Should run new round (3)
      ✓ Should return the number of rounds after new round (3)
      ✓ Should return the total amount lent (3)
      ✓ Should run new round (4)
      ✓ Should return the number of rounds after new round (4)
      ✓ Should return the total amount lent (4)
      ✓ Should returns rewards after 10 days (round0) (327ms)
      ✓ Should fail to withdraw if before the endPeriod
      ✓ Should returns all finished rounds with no finished rounds
      ✓ Should returns all finished rounds after 10 + 20 days (30days passed)
      ✓ Should returns all finished rounds after 30 more days (60days passed)
      ✓ Should returns all finished rounds after 30 more days (90days passed)
      ✓ Should returns all finished rounds after 30 more days (120days passed)
      ✓ Should returns all finished rounds after 30 more days (150days passed)
      ✓ Should returns all finished rounds after 30 more days (180days passed)
      ✓ Should withdraw all finished Rounds (10129ms)
      ✓ Should returns all finished rounds after withdrawal


  34 passing (42s)
```

```
LenderPool - Advanced
  ✓ Should return the Trade Token (531ms)
  ✓ Should return the USDT Token (466ms)
  ✓ Should return the DAI Token (462ms)
  ✓ Should buy USDT on quickswap (9797ms)
  ✓ Should buy DAI on quickswap (6484ms)
  ✓ Should distribute USDT to 10 different addresses (6961ms)
  ✓ Should distribute DAI to 10 different addresses (5569ms)
  LenderPool - 1 - StableAPY: 5%, USDT, minDeposit: 100 USDT
    ✓ Should return the LenderPool once it's deployed (1209ms)
    ✓ Should set the minimum deposit to 100 USDT
    User0 - Round0, amount: 1000 USDT, bonusAPY: 10%, Tenure: 30, TradeBonus: true
      ✓ Should run new Rounds for user0 (742ms)
      ✓ Should get round 0 from user0
      ✓ Should get all rounds for user0
      ✓ Should return stable rewards for user0 for round0 at the endPeriod
      ✓ Should return bonus rewards for user0 for round0 at the endPeriod
      ✓ Should withdraw for user0 for round0 (7745ms)
    User1 - Round0, amount: 100 USDT, bonusAPY: 8%, Tenure: 30, TradeBonus: true
      ✓ Should run new Round for user1 (542ms)
      ✓ Should return stable rewards for user1 for round0 at the endPeriod
      ✓ Should return bonus rewards for user1 for round0 at the endPeriod
      ✓ Should withdraw for user1 for round0 (711ms)
    User2 - Round0, amount: 1000 USDT, bonusAPY: 10%, Tenure: 30, TradeBonus: true
      ✓ Should run new Round for user2 (628ms)
      ✓ Should return stable rewards for user2 for round0 at the endPeriod
      ✓ Should return bonus rewards for user2 for round0 at the endPeriod
      ✓ Should withdraw for user2 for round0 (516ms)
  LenderPool - 2 - StableAPY: 8%, USDT, minDeposit: 1000 USDT
    ✓ Should return the LenderPool once it's deployed (981ms)
    ✓ Should set the minimum deposit to 1000 USDT
    User0 - Round0, amount: 5000 USDT, bonusAPY: 7%, Tenure: 60, TradeBonus: false
      ✓ Should run new Rounds for user0 (546ms)
      ✓ Should return stable rewards for user0 for round0 at the endPeriod
      ✓ Should return bonus rewards for user0 for round0 at the endPeriod
      ✓ Should withdraw for user0 for round0 (576ms)
    User1 - Round0, amount: 500 USDT, bonusAPY: 9%, Tenure: 60, TradeBonus: true
      ✓ Should fail run new Round if amount less than minimum deposit (528ms)
  LenderPool - 3 - StableAPY: 6%, DAI, minDeposit: 100 DAI
    ✓ Should return the LenderPool once it's deployed (1895ms)
    ✓ Should set the minimum deposit to 100 DAI
    ✓ Should run new Rounds for user0 (519ms)
    ✓ Should return stable rewards for user0 for round0 at the endPeriod
    ✓ Should return bonus rewards for user0 for round0 at the endPeriod
    ✓ Should withdraw for user0 for round0 (549ms)


36 passing (52s)
```

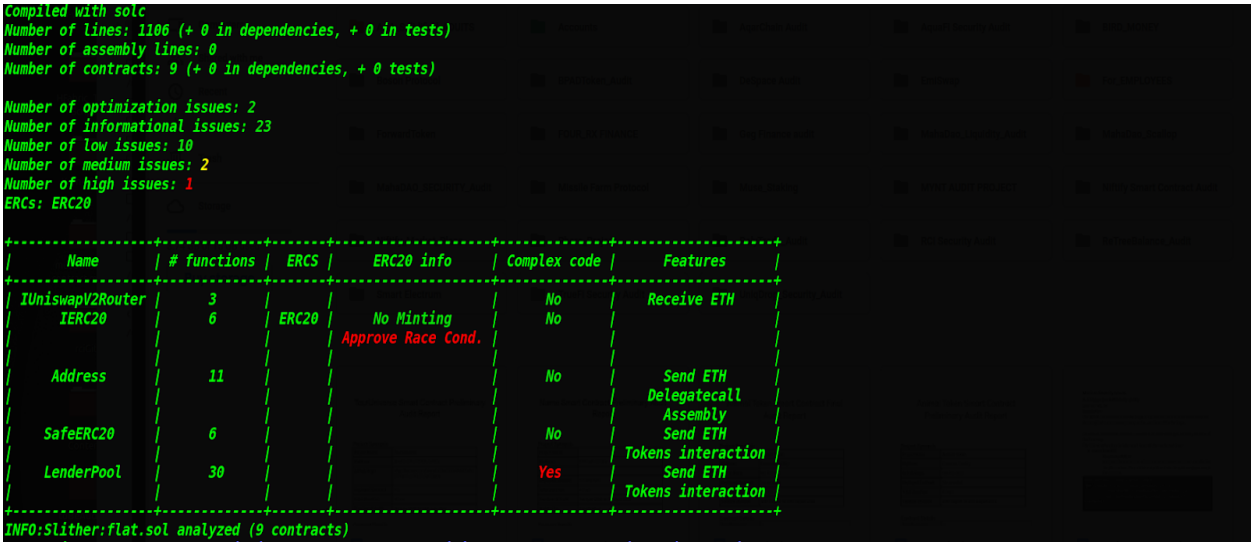
Test Coverage Report



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	100	100	100	
LenderPool.sol	100	100	100	100	
scribbleLender.sol	100	100	100	100	
contracts/interfaces/	100	100	100	100	
ILenderPool.sol	100	100	100	100	
IUniswapV2Router.sol	100	100	100	100	
All files	100	100	100	100	

> Istanbul reports written to ./coverage/ and ./coverage.json

Automated Audit Result



```
Compiled with solc
Number of lines: 1106 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 9 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 2
Number of informational issues: 23
Number of low issues: 10
Number of medium issues: 2
Number of high issues: 1
ERCs: ERC20

+----------------+------------+-------+-------------------+-------------+--------------------+
|      Name      | # functions | ERCS |     ERC20 info    | Complex code |      Features      |
+----------------+------------+-------+-------------------+-------------+--------------------+
| IUniswapV2Router |     3      |       |                   |      No     |    Receive ETH     |
|      IERC20    |     6      | ERC20 |     No Minting    |      No     |                    |
|                |            |       | Approve Race Cond. |             |                    |
|                |            |       |                   |             |                    |
|     Address    |     11     |       |                   |      No     |     Send ETH       |
|                |            |       |                   |             |   Delegatecall     |
|                |            |       |                   |             |     Assembly       |
|    SafeERC20   |     6      |       |                   |      No     |     Send ETH       |
|                |            |       |                   |             | Tokens interaction |
|    LenderPool  |     30     |       |                   |     Yes     |     Send ETH       |
|                |            |       |                   |             | Tokens interaction |
+----------------+------------+-------+-------------------+-------------+--------------------+
INFO:Slither:flat.sol analyzed (9 contracts)
```

This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

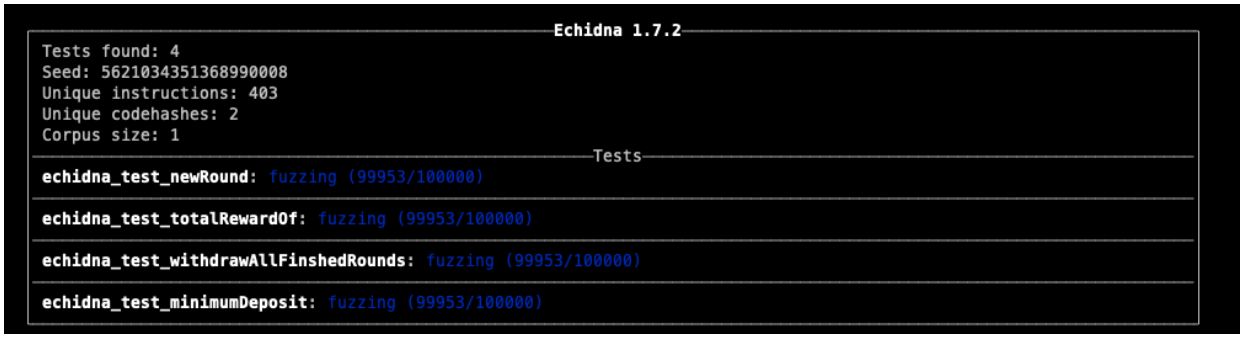
Fuzz Testing

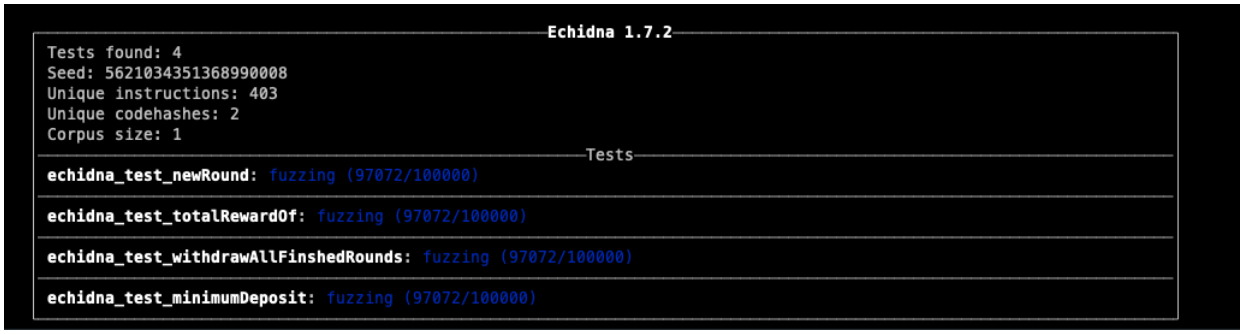
- New Round making - **PASS**
- setMinimum deposit - **PASS**
- Invalid Time withdrawAll - **PASS**
- totalReward - **PASS**
- validTime withdrawAll - **PASS**
- transferOwnership - **PASS**
- renounceOwnership - **PASS**

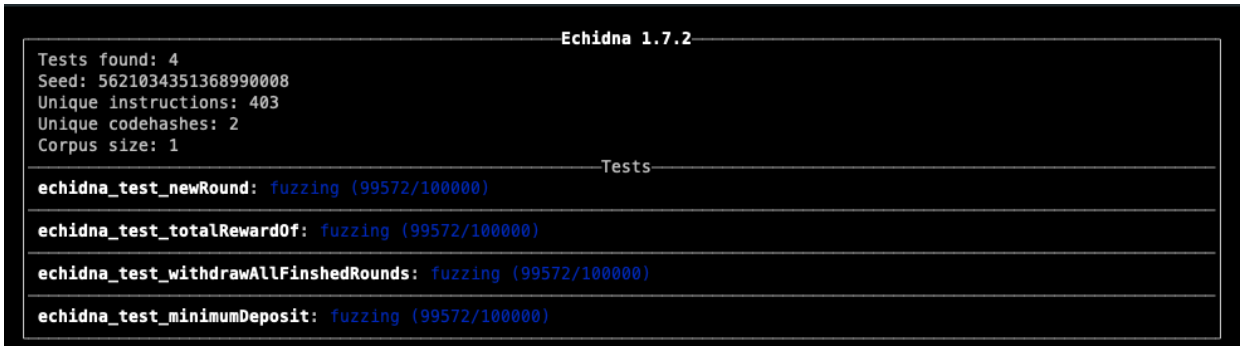
Screenshots:

Part 1 Started:

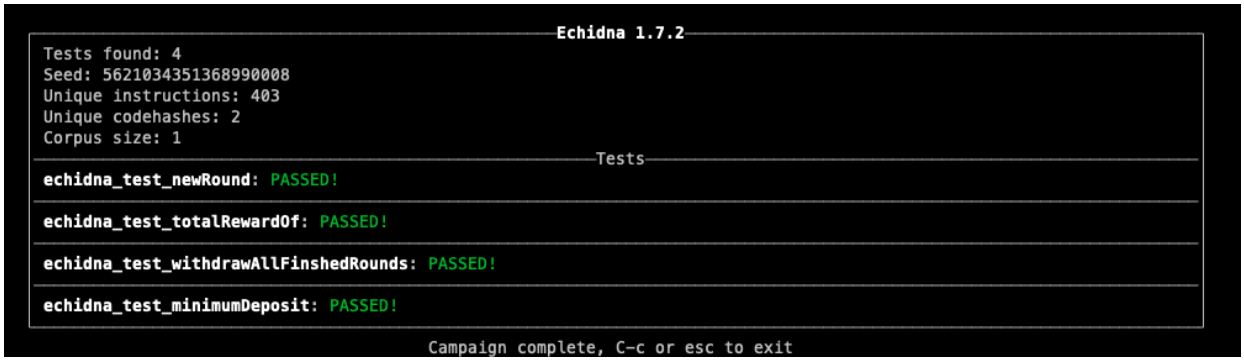
Part 1 In progress:







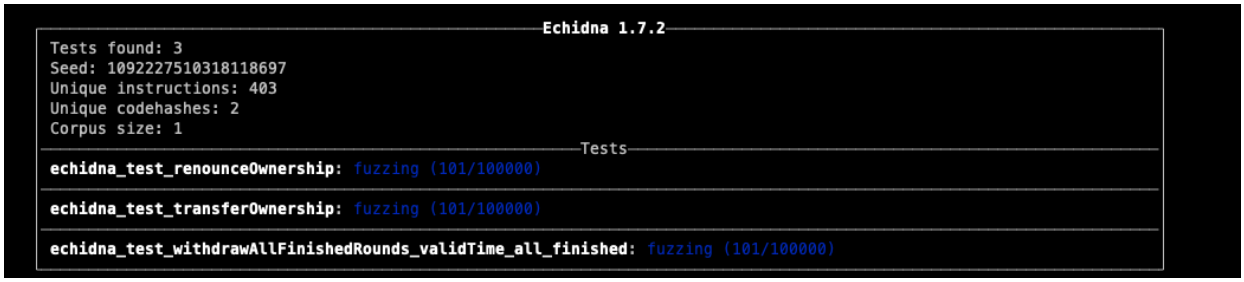
Part 1 Output:



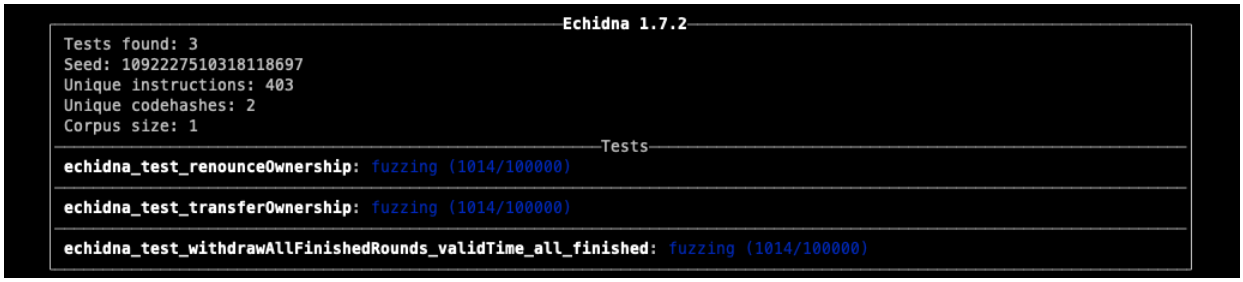

```
ethsec@b88b8f92d449:/code/Ayush/Audits/lender-portal-contracts/contracts/echidna$ echidna-test TLenderPool.sol --contract TestPool --config TLenderPool.ym
Warning: unused option: mutation
Warning: unused option: dashboard
Loaded total of 1 transactions from corpus/coverage
Analyzing contract: /code/Ayush/Audits/lender-portal-contracts/contracts/echidna/TLenderPool.sol:TestPool
echidna_test_newRound: passed! 🎉
echidna_test_totalRewardOf: passed! 🎉
echidna_test_withdrawAllFinshedRounds: passed! 🎉
echidna_test_minimumDeposit: passed! 🎉

Unique instructions: 403
Unique codehashes: 2
Corpus size: 1
Seed: 5621034351368990008
ethsec@b88b8f92d449:/code/Ayush/Audits/lender-portal-contracts/contracts/echidna$
```

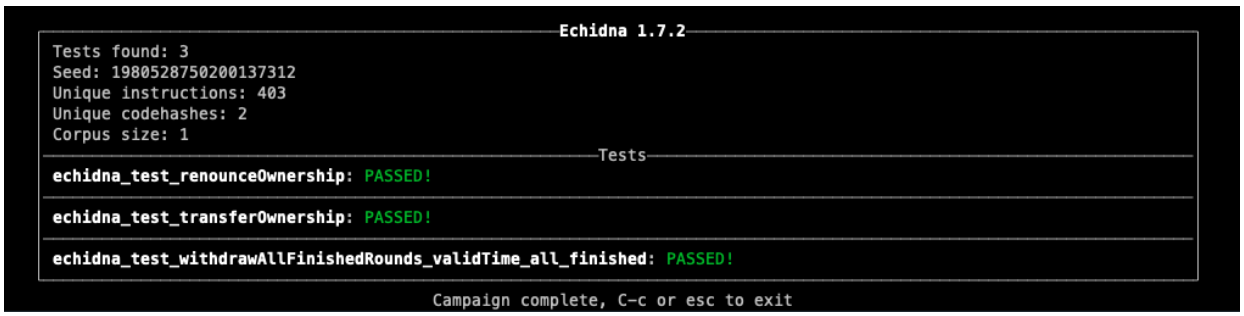
Part 2 Started:



Part 2 In progress:



Part 2 completed:



Concluding Remarks

While conducting the audits of the PolyTrade Finance smart contract, it was observed that the contracts contain High, Medium, and Low severity issues.

Our auditors suggest that High, Medium, and Low severity issues should be resolved by the developers. The recommendations given will improve the operations of the smart contract.

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the PolyTrade Finance platform or its product nor this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes