PolyTrade

Client Portal

Smart Contract Audit Report

January 12, 2021

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

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: Token.sol, PricingTable.sol, Offer.sol, PriceFeeds.sol
- Languages: Solidity(Smart contract), Typescript (Unit Testing)
- Github commits for initial audit: da168eb601cca2851ed56f213e742cd6e46c6bbc
- Github commits for final audit: 8d399f242aca69dd12048e61a1f15bd44a520068
- Testnet deployment:
    - Token: 0x8f256e58d0309Fcfb75506E4EE2beD32bcf997f9
    - PriceFeeds: 0x4E87257F423F10603EE150E3A1d9918988d4df4F
    - PricingTable: 0xc260793891953Cd3D4c7E60bB1f2Dc6a6587bde6
    - Offers: 0x04E16934A5B6c7eE82cE187AB56a22b81bd47Cb4
- 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	-	-	4

High Severity Issues

No issues were found.

Medium Severity Issues

No issues were found.

Low Severity Issues

1. **The contract should verify the validity of Chainlink's oracle data.**
   The **getPrice**() function of **PriceFeeds** contract calls the **latestRoundData**() function of **AggregatorV3Interface** contract to fetch an asset's price. However, it does not check the validity and freshness of the chainlink oracle's price.
   If there is a problem with chainlink starting a new round and finding consensus on the new value for the oracle (e.g. chainlink nodes abandon the oracle, chain congestion, vulnerability/attacks on the chainlink system) the **PriceFeeds** contract may continue using outdated stale data (if oracles are unable to submit no new round is started).

   Recommendation:
   Consider performing validation checks on the returned data of **AggregatorV3Interface.latestRoundData()** function.

   Amended (Jan 12th, 2022): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

2. **Null address checks are not present at multiple places.**
   The **setStableAggregator**() function of **PriceFeeds** contract and **setPricingTableAddress**(), **setPriceFeedAddress**() function of **Offer** smart contract accept any address value for updating the respective contract states. But these functions do not perform validation check **null** address value (0x000…). Smart contracts must explicitly verify all the input variables for their functions.

   Recommendation:
   Consider adding a validation check for **zero** address in the above-mentioned functions.

   Amended (Jan 12th, 2022): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

3. **_checkParams() function is missing validity check for pricingId.**
   The **_checkParams**() function in **Offer** smart contract is used to validate multiple input variables of **createOffer**() function. However, it misses the validity check for the **pricingId** variable. Since the function is intended to be used as a sanity check function it should explicitly verify and validate every input parameter.

   Recommendation:
   Consider validating the **pricingId** parameter using the **IPricingTable.isPricingItemValid()** function.

   Amended (Jan 12th, 2022): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

4. **The createOffer() function accepts any address as the stableAddress variable.**
   In the **createOffer()** (and **reserveRefund()**) function of **Offer** smart contract can accept any ethereum address as the **OfferItem.OfferParams.stableAddress** variable. These functions also make external calls to these smart contracts. Making external calls to untrusted contracts can be risky and should be done cautiously.

   Recommendation:
   Consider implementing a whitelist for acceptable **stableAddress** values or any other form of validation mechanism for **stableAddress** variables.

   Amended (Jan 12th, 2022): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

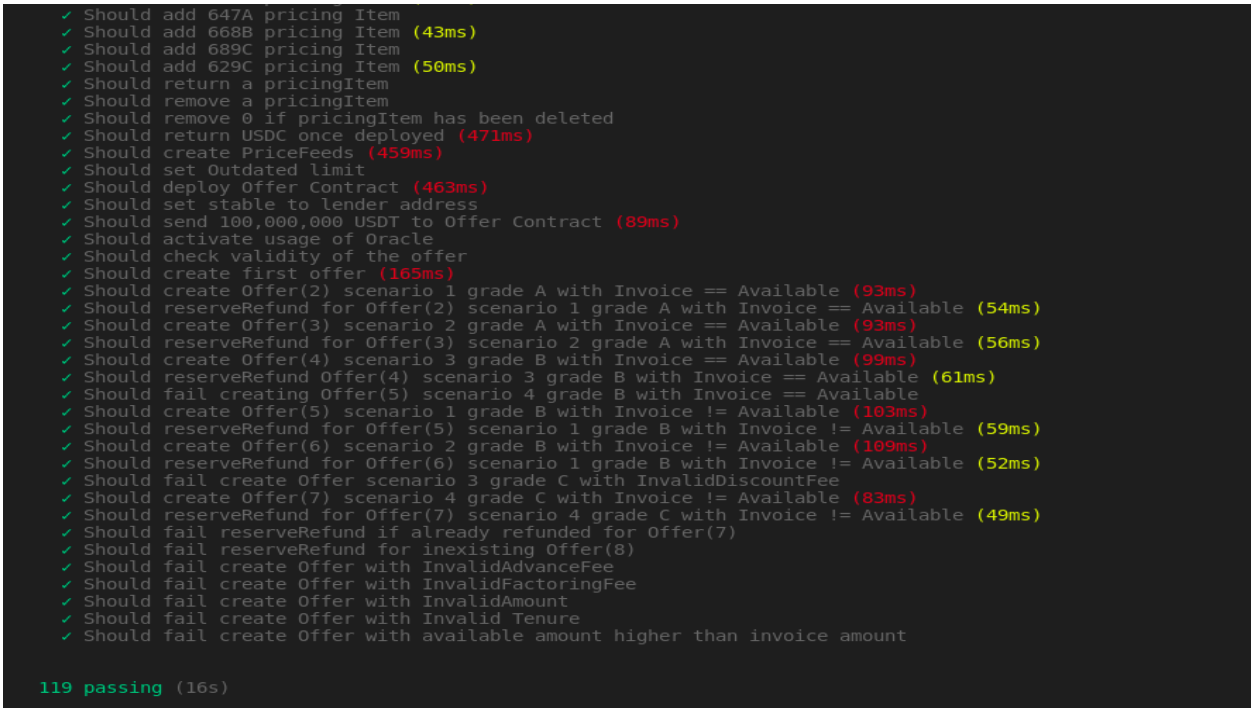
Recommendation / Informational

1. **activateOracle() and deactivateOracle() functions can be combined together.**
   The **activateOracle()** and **deactivateOracle()** functions of **Offer** smart contract can be combined together to reduce the contract size.
   Like this:

```
    /**
     * @dev Emitted when Oracle usage is toggled
     */
    event ToggledOracle(bool state);
    /**
     * @dev Toggle usage of the Oracle
     */
    function toggleOracleSwitch() external onlyOwner {
        toggleOracle = !toggleOracle;
        emit ToggledOracle(toggleOracle);
}
```

   Amended (Jan 12th, 2022): The issue has been fixed by the Polytrade Finance team and is no longer present in the smart contract.

Unit Tests



Test Coverage Report

91.75% Statements 89/97 **96.43%** Branches 27/28 **87.5%** Functions 21/24 **92.38%** Lines 97/105

File	Statements		Branches		Functions		Lines	
Chainlink/	100%	4/4	100%	0/0	100%	3/3	100%	4/4
Offer/	87.5%	56/64	95.83%	23/24	78.57%	11/14	88.73%	63/71
PricingTable/	100%	26/26	100%	4/4	100%	5/5	100%	27/27
Token/	100%	3/3	100%	0/0	100%	2/2	100%	3/3

100% Statements 107/107 **100%** Branches 38/38 **100%** Functions 24/24 **100%** Lines 115/115

File	Statements		Branches		Functions		Lines	
Chainlink/	100%	6/6	100%	2/2	100%	4/4	100%	6/6
Offer/	100%	75/75	100%	32/32	100%	15/15	100%	82/82
PricingTable/	100%	26/26	100%	4/4	100%	5/5	100%	27/27

Automated Audit Result

Solhint Linting Violations

Solhint is an open-source project for linting solidity code, providing both security and style guide validations. It integrates seamlessly into most mainstream IDEs. We used Solhint as a plugin within our VScode for this analysis. No Linting violations were detected by Solhint.

Contract Library

Contract-library contains the most complete, high-level decompiled representation of all Ethereum smart contracts, with security analysis applied to them in real-time. We performed analysis using contract Library on the Kovan address of the SporeToken, SporeStake and LiquidityFarming contracts used during manual testing:

- Token: 0x8f256e58d0309Fcfb75506E4EE2beD32bcf997f9
- PriceFeeds: 0x4E87257F423F10603EE150E3A1d9918988d4df4F
- PricingTable: 0xc260793891953Cd3D4c7E60bB1f2Dc6a6587bde6
- Offers: 0x04E16934A5B6c7eE82cE187AB56a22b81bd47Cb4

It raises no major concern for the contracts.

Slither

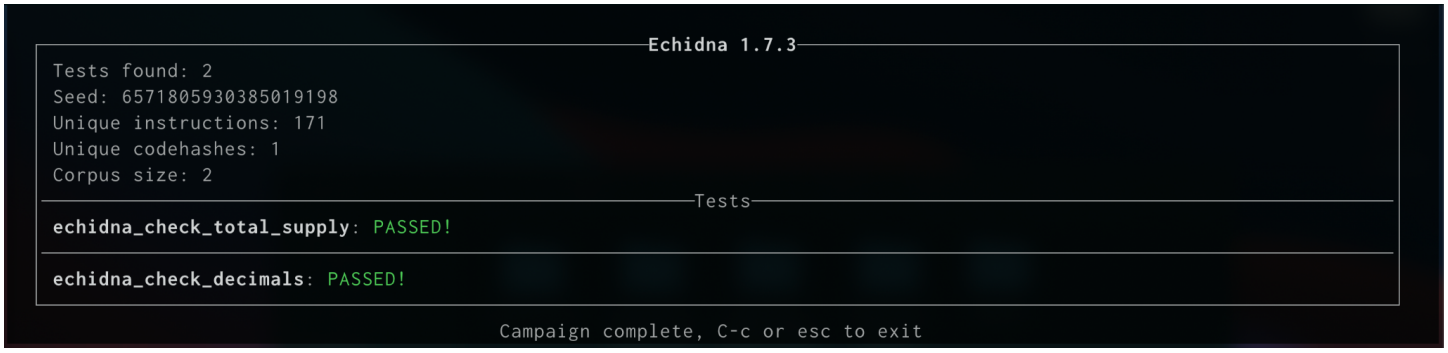
Slither, an open-source static analysis framework. This tool provides rich information about Ethereum smart contracts and has critical properties. While Slither is built as a security-oriented
static analysis framework, it is also used to enhance the user's understanding of smart contracts, assist in code reviews, and detect missing optimizations.

The concerns slither raises have already been covered in the manual audit section.

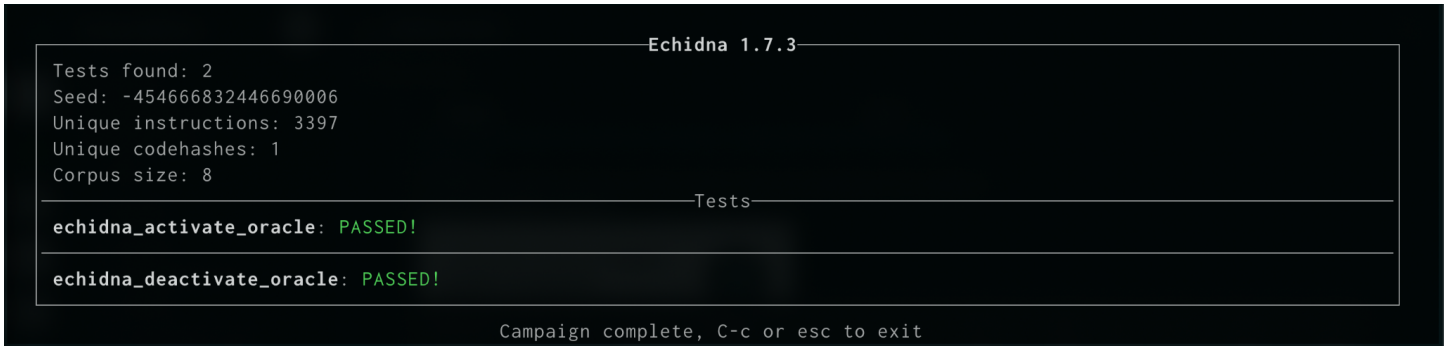
```
Offers._calculateDiscountAmount(uint256,uint16,uint8) (Offer-flat.sol#1135-1141) performs a multiplication on the result of a division:
        -(((advancedAmount * discountFee) / 365) * tenure) / _precision (Offer-flat.sol#1140)
Offers._calculateLateAmount(uint256,uint16,uint24) (Offer-flat.sol#1151-1157) performs a multiplication on the result of a division:
        -(((lateFee * advancedAmount) / 365) * lateDays) / _precision (Offer-flat.sol#1156)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply

Offers.createOffer(bytes2,IOffer.OfferParams) (Offer-flat.sol#927-980) uses a dangerous strict equality:
        - require(bool,string)((offer.advancedAmount + offer.reserve) == params.invoiceAmount,advanced + reserve != invoice) (Offer-flat.sol#954-957)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities

Offers.reserveRefund(uint256,uint64,uint16).refunded (Offer-flat.sol#1003) is a local variable never initialized
Offers.createOffer(bytes2,IOffer.OfferParams).offer (Offer-flat.sol#945) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables

Reentrancy in Offers.createOffer(bytes2,IOffer.OfferParams) (Offer-flat.sol#927-980):
        External calls:
        - stable.safeTransfer(offer.params.treasuryAddress,amountToTransfer) (Offer-flat.sol#973)
        - stable.safeTransfer(offer.params.treasuryAddress,amount) (Offer-flat.sol#975)
        Event emitted after the call(s):
        - OfferCreated(_countId,pricingId) (Offer-flat.sol#978)
Reentrancy in Offers.reserveRefund(uint256,uint64,uint16) (Offer-flat.sol#991-1050):
        External calls:
        - stable.safeTransfer(offer.params.treasuryAddress,amount) (Offer-flat.sol#1047)
        Event emitted after the call(s):
        - ReserveRefunded(offerId,amount) (Offer-flat.sol#1049)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3

Offers.createOffer(bytes2,IOffer.OfferParams) (Offer-flat.sol#927-980) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)((offer.advancedAmount + offer.reserve) == params.invoiceAmount,advanced + reserve != invoice) (Offer-flat.sol#954-957)
Offers.reserveRefund(uint256,uint64,uint16) (Offer-flat.sol#991-1050) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(offers[offerId].refunded.netAmount == 0,Offer already refunded) (Offer-flat.sol#997-1000)
        - block.timestamp > (dueDate + offer.params.gracePeriod) && block.timestamp - dueDate > offer.params.gracePeriod (Offer-flat.sol#1010-1011)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Address.isContract(address) (Offer-flat.sol#120-130) uses assembly
        - INLINE ASM (Offer-flat.sol#126-128)
Address.verifyCallResult(bool,bytes,string) (Offer-flat.sol#289-309) uses assembly
        - INLINE ASM (Offer-flat.sol#301-304)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
```

```
Offers.constructor(address,address,address).treasuryAddress (Offer-flat.sol#862) lacks a zero-check on :
                - treasury = treasuryAddress (Offer-flat.sol#866)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

Reentrancy in Offers.createOffer(uint16,IOffer.OfferParams) (Offer-flat.sol#967-1029):
        External calls:
        - stable.safeTransferFrom(stableToPool[address(stable)],treasury,amountToTransfer) (Offer-flat.sol#1013-1017)
        - stable.safeTransferFrom(stableToPool[address(stable)],treasury,amount) (Offer-flat.sol#1020-1024)
        Event emitted after the call(s):
        - OfferCreated(_countId,pricingId) (Offer-flat.sol#1027)
Reentrancy in Offers.reserveRefund(uint256,uint64,uint16) (Offer-flat.sol#1040-1104):
        External calls:
        - stable.safeTransferFrom(stableToPool[address(stable)],treasury,amount) (Offer-flat.sol#1097-1101)
        Event emitted after the call(s):
        - ReserveRefunded(offerId,amount) (Offer-flat.sol#1103)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3

Offers.reserveRefund(uint256,uint64,uint16) (Offer-flat.sol#1040-1104) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(_offerToPricingId[offerId] != 0 && offers[offerId].refunded.netAmount == 0,Invalid Offer)
        - block.timestamp > (dueDate + offer.params.gracePeriod) && block.timestamp - dueDate > offer.params.gracePeriod
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Address.isContract(address) (Offer-flat.sol#120-130) uses assembly
        - INLINE ASM (Offer-flat.sol#126-128)
Address.verifyCallResult(bool,bytes,string) (Offer-flat.sol#289-309) uses assembly
        - INLINE ASM (Offer-flat.sol#301-304)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
```

Fuzz Testing




```
client-portal-contracts >>> echidna-test ./contracts/echidna/Token/TToken.sol --contract TestToken
Analyzing contract: /Users/pushpitbhardwaj/Downloads/Docs and Certificates/Internships/Ongoing/ImmuneBytes/work/client-portal
-contracts/contracts/echidna/Token/TToken.sol:TestToken
echidna_check_total_supply: passed! 🎉
echidna_check_decimals: passed! 🎉

Unique instructions: 171
Unique codehashes: 1
Corpus size: 2
Seed: 6571805930385019198
```




```
client-portal-contracts >>> echidna-test ./contracts/echidna/Offer/TOffer.sol --contract TestOffer --config ./contracts/echid
na/Offer/config.yml
Warning: unused option: mutation
Warning: unused option: dashboard
Loaded total of 6 transactions from contracts/echidna/Offer/corpus/coverage
Analyzing contract: /Users/pushpitbhardwaj/Downloads/Docs and Certificates/Internships/Ongoing/ImmuneBytes/work/client-portal
-contracts/contracts/echidna/Offer/TOffer.sol:TestOffer
echidna_activate_oracle: passed! 🎉
echidna_deactivate_oracle: passed! 🎉

Unique instructions: 3397
Unique codehashes: 1
Corpus size: 8
Seed: -454666832446690006
```

Concluding Remarks

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

Our auditors suggest that 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