ttUSD

Smart Contract Audit Report
Prepared for TukTuk Finance

Date Issued: Aug 16, 2021
Project ID: AUDIT2021008
Version: v1.0
Confidentiality Level: Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021008
Version	v1.0
Client	TukTuk Finance
Project	ttUSD
Auditor(s)	Weerawat Pawanawiwat Suvicha Buakhom Patipon Suwanbol
Author	Suvicha Buakhom
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Aug 16, 2021	Full report	Suvicha Buakhom

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by TukTuk Finance, Inspex team conducted an audit to verify the security posture of the ttUSD smart contracts between 19 Jul, 2021 and 23 Jul, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of ttUSD smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 4 critical, 1 high, 2 medium, 1 low, 3 very low, and 2 info-severity issues. With the project team's prompt response, 4 critical, 1 high, 2 medium, 1 low and 2 very low-severity issues were resolved in the reassessment, while 1 very low and 2 info-severity issues were acknowledged by the team. Therefore, Inspex trusts that ttUSD smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inpex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

TukTuk Finance is a DeFi platform with multiple products such as yield farming, gachapon, or algorithmic stable coin. It is currently available on Bitkub Chain and Binance Smart Chain.

ttUSD is an fractional-reserve algorithmic stablecoin which is a part of TukTuk Space platform launched on Binance Smart Chain and pegged to the dollar. On ttUSD pool users can mint $ttUSD by using $BUSD and $TUK as collateral. On the other hand, users can bring $ttUSD to redeem $BUSD and $TUK back.

Scope Information:

Project Name	ttUSD
Website	https://space.tuktuk.finance/
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	19 Jul, 2021 - 23 Jul, 2021
Reassessment Date	13 Aug, 2021

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 9e647a26277edc8b0ea30a72d327db4ed52f76f5)

Contract	Location (URL)
CollateralOracle	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/CollateralOracle.sol
CollateralRatioPolicy	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/CollateralRatioPolicy.sol
CollateralReserve	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/CollateralReserve.sol
DollarOracle	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/DollarOracle.sol
MultiPairOracle	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/MultiPairOracle.sol
PcsPairOracle	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/PcsPairOracle.sol
Pool	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/Pool.sol
ProfitFund	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/ProfitFund.sol
ShareOracle	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/ShareOracle.sol
TTUSD	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/TTUSD.sol
Treasury	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/Treasury.sol
TreasuryPolicy	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/TreasuryPolicy.sol
TreasuryVaultAlpaca	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/TreasuryVaultAlpaca.sol
ZapPool	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9e647a2627/contracts/ZapPool.sol

Reassessment: (Commit: 9c1d23645f33ac0f9dc684bf86b67a7f410c2e9a)

Contract	Location (URL)
CollateralOracle	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/CollateralOracle.sol
CollateralRatioPolicy	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/CollateralRatioPolicy.sol
CollateralReserve	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/CollateralReserve.sol
DollarOracle	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/DollarOracle.sol
MultiPairOracle	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/MultiPairOracle.sol
PcsPairOracle	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/PcsPairOracle.sol
Pool	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/Pool.sol
ProfitFund	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/ProfitFund.sol
ShareOracle	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/ShareOracle.sol
TTUSD	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/TTUSD.sol
Treasury	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/Treasury.sol
TreasuryPolicy	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/TreasuryPolicy.sol
TreasuryShield	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/TreasuryShield.sol
TreasuryVaultAlpaca	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/TreasuryVaultAlpaca.sol
TreasuryVaultAlpacaShield	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f/contracts/TreasuryVaultAlpacaShield.sol
ZapPool	https://github.com/TukTukFinance/tuktuk-bsc-ttusd-contract/blob/9c1d23645f

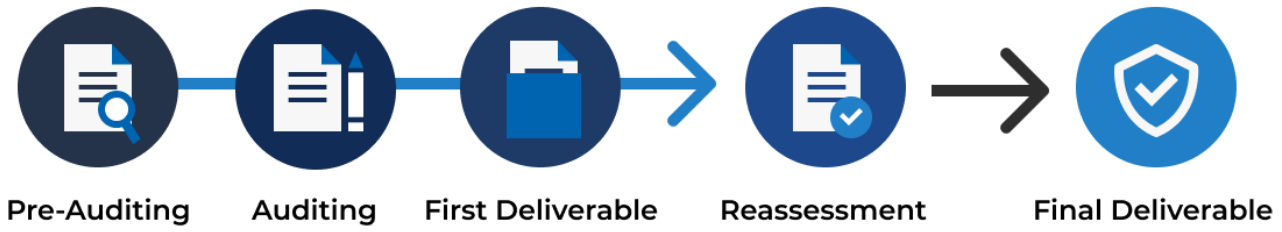
	/contracts/ZapPool.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they are inherited from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Upgradable Without Timelock
Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation

Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

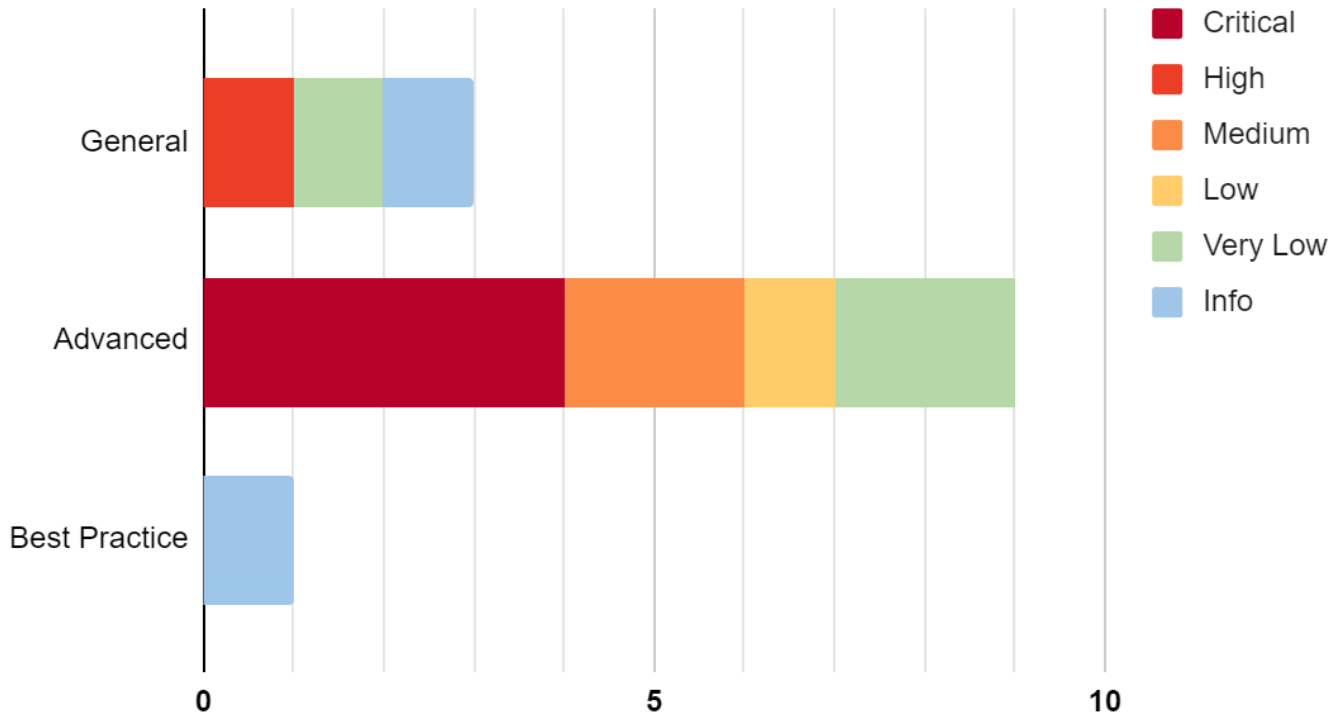
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Impact \ Likelihood	Low	Medium	High
Low	**Very Low**	**Low**	**Medium**
Medium	**Low**	**Medium**	**High**
High	**Medium**	**High**	**Critical**

4. Summary of Findings

From the assessments, Inspex has found 13 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Arbitrary Function Execution	Advanced	Critical	Resolved
IDX-002	Token Draining by Owner	Advanced	Critical	Resolved
IDX-003	Arbitrary Token Minting by Owner	Advanced	Critical	Resolved
IDX-004	Price Oracle Manipulation by Owner	Advanced	Critical	Resolved
IDX-005	Centralized Control of State Variable	General	High	Resolved
IDX-006	Improper Deduction of User's Share	Advanced	Medium	Resolved *
IDX-007	Short MINIMUM_DELAY time in Timelock	Advanced	Medium	Resolved
IDX-008	ZapPool Price Impact Calculation	Advanced	Low	Resolved
IDX-009	Improper Account Type Checking	Advanced	Very Low	Resolved
IDX-010	Improper Collateral Reserve Amount	Advanced	Very Low	Resolved *
IDX-011	Outdated Compiler Version	General	Very Low	Acknowledged
IDX-012	Oracle Denial of Service	General	Info	No Security Impact
IDX-013	Improper Function Visibility	Best Practice	Info	No Security Impact

* The mitigations or clarifications by TukTuk Finance can be found in Chapter 5.

5. Detailed Findings Information

5.1. Arbitrary Function Execution

ID	IDX-001
Target	Treasury TreasuryVaultAlpaca
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: Critical** **Impact: High** The contract owner can arbitrarily execute any function as the callee contract (`Treasury` or `TreasuryVaultAlpaca`) and drain funds. **Likelihood: High** The only restriction deterring the owner from performing this attack is the `Timelock` contract with a short minimum delay of 6 hours.
Status	Resolved TukTuk Finance team has resolved this issue by implementing `TreasuryShield` for `Treasury` and `TreasuryVaultAlpacaShield` for `TreasuryVaultAlpaca` as suggested. `TreasuryShield` contract: https://bscscan.com/address/0x1Ac1FbC797182e76C2FaCc0069dAD563740621fc#code `TreasuryVaultAlpacaShield` contract: https://bscscan.com/address/0x903275FcdC44613351daa2dEe8A2DeeE7Cf17864#code Ownership transfer of `Treasury` to `TreasuryShield` contract: https://bscscan.com/tx/0x611c108f484a6e853253d6fcbf33d4b566d46c197fbee02acb832a9db1d2780c#eventlog Ownership transfer of `TreasuryVaultAlpaca` to `TreasuryVaultAlpacaShield` contract: https://bscscan.com/tx/0xccc1c5ee3bf1dfdae0b5fb63eb50c00b67d9f0db7202c842d511223978e750f8#eventlog

5.1.1. Description

In the `Treasury` contract, the `executeTransaction()` function can be used by the owner to execute arbitrary functions as the `Treasury` contract.

Treasury.sol

```
1394 function executeTransaction(
1395     address target,
```

```
1396      uint256 value,
1397      string memory signature,
1398      bytes memory data
1399  ) public onlyOwner returns (bytes memory) {
1400      bytes memory callData;
1401
1402      if (bytes(signature).length == 0) {
1403          callData = data;
1404      } else {
1405          callData = abi.encodePacked(bytes4(keccak256(bytes(signature))), data);
1406      }
1407      // solium-disable-next-line security/no-call-value
1408      (bool success, bytes memory returnData) = target.call{value:
      value}(callData);
1409      require(success, string("TreasuryVaultAave::executeTransaction: Transaction
      execution reverted."));
1410      return returnData;
1411  }
```

This allows the owner to execute critical functions that can be executed by `Treasury` contract only, such as `CollateralReserve.transferTo()` function, allowing the owner to drain all funds in the reserve.

CollateralReserve.sol

```
606  function transferTo(
607      address _token,
608      address _receiver,
609      uint256 _amount
610  ) public override onlyTreasury {
611      require(_receiver != address(0), "Invalid address");
612      require(_amount > 0, "Cannot transfer zero amount");
613      IERC20(_token).safeTransfer(_receiver, _amount);
614  }
```

Likewise, there is an `executeTransaction()` function in `TreasuryVaultAlpaca`, allowing the owner to execute arbitrary functions as the `TreasuryVaultAlpaca` contract.

For example, instead of using the `withdraw()` function, the owner can call the `withdrawAll()` function of the fair launch contract manually, as called in line 744, and transfer the token out to another wallet to steal the funds.

CollateralReserve.sol

```
732  function withdraw() external onlyTreasury {
733      IERC20 _asset = IERC20(asset);
734      IERC20 _ibAsset = IERC20(ibAsset);
735
```

```
736     if (ibVaultBalance == 0) {
737         // when pauseVault = true, transfer all balance to treasury
738         _asset.safeTransfer(treasury, vaultBalance);
739         vaultBalance = _asset.balanceOf(address(this));
740         return;
741     }
742 
743     //step 1. withdraw all from fairLaunch
744     IFairLaunch(fairLaunch).withdrawAll(address(this), pid);
745     uint256 ibBalance = _ibAsset.balanceOf(address(this));
746 
747     //step 2. withdraw from ibBUSD to BUSD
748     IVault(ibAsset).withdraw(ibBalance); // withdraw to BUSD
749     uint256 newBalance = _asset.balanceOf(address(this)); // withdraw
    everything in vault
750     uint256 profit = 0;
751     if (newBalance > vaultBalance) {
752         profit = newBalance - vaultBalance;
753     }
754 
755     //step 3. transfer BUSD to treasury, vaultProfitFund
756     _asset.safeTransfer(treasury, vaultBalance);
757     _asset.safeTransfer(vaultProfitFund, profit);
758     vaultBalance = _asset.balanceOf(address(this));
759     ibVaultBalance = _ibAsset.balanceOf(address(this));
760 
761     emit Withdrawn(newBalance);
762     emit Profited(profit);
763 }
```

5.1.2. Remediation

Inspex suggests removing the `executeTransaction()` function in both `Treasury` and `TreasuryVaultAlpaca` contracts.

However, if the `Treasury` and `TreasuryVaultAlpaca` contracts cannot be modified and redeployed, Inspex suggests implementing a shield contract that forwards all `onlyOwner` functions except the `executeTransaction()` function.

The following example is the example shield contract of the Treasury contract. The `TreasuryShield` contract will forward all `onlyOwner` functions to `Treasury` contract except `executeTransaction()` function:

TreasuryShield.sol

```
1 pragma solidity 0.8.6;
2 
```

```
3  import "interfaces/ITreasury.sol";
4  import "@openzeppelin/contracts/access/Ownable.sol@v4.1.0";
5
6  contract TreasuryShield is Ownable {
7      ITreasury public treasury;
8
9      constructor(address _owner, ITreasury _treasury) public {
10         transferOwnership(_owner);
11         treasury = _treasury;
12     }
13
14     function addPool(address pool_address) public onlyOwner {
15         ITreasury(treasury).addPool(pool_address);
16     }
17
18     function removePool(address pool_address) public onlyOwner {
19         ITreasury(treasury).removePool(pool_address);
20     }
21
22     function setTreasuryPolicy(address _treasuryPolicy) public onlyOwner {
23         ITreasury(treasury).setTreasuryPolicy(_treasuryPolicy);
24     }
25
26     function setCollateralRatioPolicy(address _collateralRatioPolicy) public
   onlyOwner {
27         ITreasury(treasury).setCollateralRatioPolicy(_collateralRatioPolicy);
28     }
29
30     function setOracleDollar(address _oracleDollar) external onlyOwner {
31         ITreasury(treasury).setOracleDollar(_oracleDollar);
32     }
33
34     function setOracleShare(address _oracleShare) external onlyOwner {
35         ITreasury(treasury).setOracleShare(_oracleShare);
36     }
37
38     function setOracleCollateral(address _oracleCollateral) external onlyOwner
   {
39         ITreasury(treasury).setOracleCollateral(_oracleCollateral);
40     }
41
42     function setCollateralAddress(address _collateral) public onlyOwner {
43         ITreasury(treasury).setCollateralAddress(_collateral);
44     }
45
46     function setCollateralReserve(address _collateralReserve) public onlyOwner
   {
```

```
47          ITreasury(treasury).setCollateralReserve(_collateralReserve);
48      }
49
50      function setProfitSharingFund(address _profitSharingFund) public onlyOwner
    {
51          ITreasury(treasury).setProfitSharingFund(_profitSharingFund);
52      }
53
54      function setController(address _controller) public onlyOwner {
55          ITreasury(treasury).setController(_controller);
56      }
57  }
```

The owner of the `Treasury` and `TreasuryVaultAlpaca` contracts should then be set to its shield contract using the `transferOwnership()` function.

Please note that the fixes for other issues are not yet applied in the examples above.

5.2. Token Draining by Owner

ID	IDX-002
Target	CollateralReserve Treasury
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: Critical** **Impact: High** The owner can drain all tokens from the collateral reserve contract. **Likelihood: High** The only restriction deterring the owner from performing this attack is the `Timelock` contract with a short minimum delay of 6 hours.
Status	Resolved TukTuk Finance team has resolved this issue by renouncing ownership from `CollateralReserve`. The owner has renounced the ownership of the `CollateralReserve` contract in the following transaction: https://bscscan.com/tx/0x4ece9064ecfb3dab32a3d10c841a7deed7a934a4689d900506c30adc281c412b#eventlog

5.2.1. Description

There are 2 scenarios that the owner can drain the token from `CollateralReserve` contract as follows:

Scenario 1: Draining tokens by using addPool() function

To attack with this scenario, the attacker must be the owner of `Treasury` contract and the following attack steps must be performed:

- The attacker adds the specific wallet to the `pools` state by using `addPool()` function.
- The attacker drains all tokens in the `CollateralReserve` contract by executing the `requestTransfer()` function.

The `requestTransfer()` function can be used to transfer any token (`_token`) in `CollateralReserve` contract to any address (`_receiver`) with unrestricted amount (`_amount`) as shown in the following source code:

Treasury.sol

```
1286 function requestTransfer(
```

```
1287        address _token,
1288        address _receiver,
1289        uint256 _amount
1290    ) external override onlyPools {
1291        ICollateralReserve(collateralReserve).transferTo(_token, _receiver,
        _amount);
1292    }
```

With `onlyPools` modifier, to execute the `requestTransfer()` function, the caller must be the pool as shown below:

Treasury.sol

```
1084    modifier onlyPools {
1085        require(pools[msg.sender], "Only pools can use this function");
1086        _;
1087    }
```

However, the contract owner can execute `addPool()` in the `Treasury` contract to add his wallet address to the `pools` state and call the `requestTransfer()` function to drain tokens.

Treasury.sol

```
1314    function addPool(address pool_address) public onlyOwner {
1315        require(pools[pool_address] == false, "poolExisted");
1316        pools[pool_address] = true;
1317        pools_array.push(pool_address);
1318        emit PoolAdded(pool_address);
1319    }
```

Scenario 2: Draining tokens by using setTreasury() function

To attack with this scenario, the attacker must be the owner of `CollateralReserve` contract and the following attack steps must be performed:

- The attacker sets an address owned by the attacker as a whitelisted address through the `setTreasury()` function.
- The attacker executes the `transferTo()` function using the added address to transfer tokens in `CollateralReserve` contract out to the attacker's address.

The `transferTo()` function allows any address that is qualified in `onlyTreasury` modifier to transfer any token (`_token`) to any address (`_receiver`) with an unrestricted amount (`_amount`).

CollateralReserve.sol

```
606     function transferTo(address _token, address _receiver, uint256 _amount) public
        override onlyTreasury {
607         require(_receiver != address(0), "Invalid address");
```

```
608     require(_amount > 0, "Cannot transfer zero amount");
609     IERC20(_token).safeTransfer(_receiver, _amount);
610 }
```

The `setTreasury()` function allows the owner to set any address as the treasury address. This means any address can be qualified in `onlyTreasury()` modifier, allowing that address to execute the `transferTo()` function freely.

CollateralReserve.sol

```
617 function setTreasury(address _treasury) public onlyOwner {
618     require(_treasury != address(0), "Invalid address");
619     treasury = _treasury;
620     emit TreasuryChanged(treasury);
621 }
```

5.2.2. Remediation

Due to the fact that the `addPool()` and `setTreasury()` functions can cause very high impact to the platform, Inspex suggests limiting the usage of them via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a Timelock contract to delay the changes for a significant amount of time e.g., 3 days

However, for the `setTreasury()` function, it is highly recommended to allow only one calling of the function.

Ideally, this critical state variable should not be modifiable to represent transparency to users. Inspex suggests editing the `CollateralReserve` contract to allow setting the treasury address only once by removing the `setTreasury()` function from the contract.

5.3. Arbitrary Token Minting by Owner

ID	IDX-003
Target	Treasury TTUSD
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: Critical** **Impact: High** The contract owner can mint unlimited $ttUSD. **Likelihood: High** The only restriction deterring the owner from performing this attack is the Timelock contract with the very short delay (6 hours).
Status	Resolved TukTuk Finance team has resolved this issue by renouncing ownership from `TTUSD`. The owner has renounced the ownership of the `TTUSD` contract in the following transaction: https://bscscan.com/tx/0x5f67e3a6cf860764b30253be43a2c06f338d40c52132c209e1bcdc5c648d295c#eventlog

5.3.1. Description

There are 2 scenarios that the owner can mint unlimited $ttUSD as follows:

Scenario 1: Minting $ttUSD by using addPool() function

To attack with this scenario, the attacker must be the owner of `Treasury` contract and the following attack steps must be performed:

- The attacker adds the specific wallet to the `pools` state by using the `addPool()` function of the `Treasury` contract.
- The attacker mint unlimited $ttUSD by executing `poolMint()` function of `TTUSD` contract.

In the `TTUSD` contract, the `poolMint()` function allows only specific addresses to mint $ttUSD.

TTUSD.sol

```
1211  function poolMint(address _address, uint256 _amount) external override
      onlyPools {
1212      super._mint(_address, _amount);
1213      emit DollarMinted(msg.sender, _address, _amount);
```

```
1214 }
```

The restriction is `onlyPools()` modifier only, which verify that `msg.sender` is in `pools` by using `hasPool()` in `Treasury` contract

TTUSD.sol

```
1190 modifier onlyPools() {
1191     require(ITreasury(treasury).hasPool(msg.sender), "!pools");
1192     _;
1193 }
```

Treasury.sol

```
1132 function hasPool(address _address) external view override returns (bool) {
1133     return pools[_address] == true;
1134 }
```

The `addPool()` function of the `Treasury` contract registers the received address to `pools` state. This allows that address to execute the `poolMint()` function in `TTUSD` contract freely, resulting in unlimited minting of $ttUSD.

Treasury.sol

```
1314 function addPool(address pool_address) public onlyOwner {
1315     require(pools[pool_address] == false, "poolExisted");
1316     pools[pool_address] = true;
1317     pools_array.push(pool_address);
1318     emit PoolAdded(pool_address);
1319 }
```

Scenario 2: Minting $ttUSD by using setTreasuryAddress() function

To attack with this scenario, the attacker must be the owner of `TTUSD` contract and the following attack steps must be performed:

- The attacker deploys the malicious contract containing `hasPool()` function which always returns `true`.
- The attacker sets the deployed contract as a whitelisted address through the `setTreasuryAddress()` function of `TTUSD` contract.
- The attacker executes the `poolMint()` function of `TTUSD` contract using the added address to mint unlimited $ttUSD to the attacker address.

The `poolMint()` function allows the address that is qualified in `onlyPools` modifier to mint the $ttUSD to any address (`_address`) with unrestricted amount (`_amount`).

TTUSD.sol

```
1211 function poolMint(address _address, uint256 _amount) external override
     onlyPools {
1212     super._mint(_address, _amount);
1213     emit DollarMinted(msg.sender, _address, _amount);
1214 }
```

The `onlyPools()` modifier calls the `hasPool()` function from the `treasury` contract in order to validate the `msg.sender`.

TTUSD.sol

```
1190 modifier onlyPools() {
1191     require(ITreasury(treasury).hasPool(msg.sender), "!pools");
1192     _;
1193 }
```

Unfortunately, the `treasury` state can be set by the contract owner with `setTreasuryAddress()` function as shown below:

TTUSD.sol

```
1218 function setTreasuryAddress(address _treasury) public onlyOwner {
1219     require(_treasury != address(0), "Invalid address");
1220     treasury = _treasury;
1221     emit TreasuryChanged(_treasury);
1222 }
```

Therefore, the contract owner can deploy the malicious contract containing `hasPool()` function which always returns as shown in the following example:

FakeTreasury.sol

```
1 contract FakeTreasury {
2     function hasPool(address _address) external view override returns (bool) {
3         return true;
4     }
5 }
```

Finally, the contract owner can set the `FakeTreasury` contract to the `treasury` state with the `setTreasuryAddress()` function and the `onlyPools()` modifier will always be bypassed.

5.3.2. Remediation

Due to the fact that the `addPool()` and `setTreasuryAddress()` functions can cause very high impact to the platform, Inspex suggests limiting the usage of them via the following options:

- Implementing a community-run governance to control the usage of these functions
- Using a `Timelock` contract to delay the changes for a significant amount of time e.g., 3 days

However, for the `setTreasuryAddress()` function, it is highly recommended to allow only one calling of the function.

5.4. Price Oracle Manipulation by Owner

ID	IDX-004
Target	CollateralOracle CollateralRatioPolicy DollarOracle MultiPairOracle PcsPairOracle Pool ShareOracle Treasury ZapPool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: Critical** **Impact: High** The owner can control the price of the assets used in the calculation, allowing the minting or redeeming of $ttUSD for high profit. **Likelihood: High** The only restriction deterring the owner from performing this attack is the Timelock contract with the very short delay (6 hours).
Status	**Resolved** TukTuk Finance team has solved this issue by implementing a timelock mechanism with 3 days delay. `Timelock` contract address: https://bscscan.com/address/0xc16f05324a94e9964f4be619207ada26f1964bc6 Ownership transfer of `CollateralOracle` to `Timelock` contract: https://bscscan.com/tx/0x40359915d28e99db2b03b7365969565c0e84412af7d1b25956634f83ffb093db#eventlog Ownership transfer of `CollateralRatioPolicy` to `Timelock` contract: https://bscscan.com/tx/0x4f7126fe40298e20199acbad46b8cd461ea14aa9bcb733cfe22553b84a6e1e20#eventlog Ownership transfer of `DollarOracle` to `Timelock` contract: https://bscscan.com/tx/0x0a80c386abda9e2b52c6b11dca24334a297828722010a43283901ff67eca092f#eventlog Ownership transfer of `MultiPairOracle` to `Timelock` contract: https://bscscan.com/tx/0x9e390119058f645e7172b4c640ac14203ea1a663fdb3598fb9ae24

	2ba60daffb#eventlog Ownership transfer of `PcsPairOracle` to `Timelock` contract: https://bscscan.com/tx/0x036e1ba736c341de2280ca7bd4e60e6bacb0e77e4e0b85043dc6fbdb58fd2755#eventlog Ownership transfer of `Pool`to `Timelock` contract: https://bscscan.com/tx/0xdfc67759da5c961370a2546421958d24f788be7ef9598629f31a34817d191f04#eventlog Ownership transfer of `PoolV2` to `Timelock` contract: https://bscscan.com/tx/0x986f249557537875befea0f749da57b031cb36cf4f8a385ec2134d94d854af4f#eventlog Ownership transfer of `ShareOracle` to `Timelock` contract: https://bscscan.com/tx/0x7ce96dab8e89f0e5b02a5c8bb7d4bc6355a91cfd43ac9ab866a0529acebe9582#eventlog Ownership transfer of `Treasury` to `TreasuryShield` contract which is owned by `Timelock` transaction: https://bscscan.com/tx/0x611c108f484a6e853253d6fcbf33d4b566d46c197fbee02acb832a9db1d2780c#eventlog Ownership transfer of `ZapPool` to `Timelock` contract: https://bscscan.com/tx/0x677eb140508573c7ea98dbf1281a2578ce840dfd7eeaf1d37850b34135bb0ba3#eventlog

5.4.1. Description

In the minting and redeeming of $ttUSD, reference prices of the collateral ($BUSD), share ($TUK), and stable ($ttUSD) tokens are fetched from the price oracles and used in the calculations.

For example, in the `mint()` function, the price of the collateral is fetched from the `getCollateralPrice()` function in line 1163 and saved in the `_price_collateral` variable.

This variable is used to calculate the value of the collateral transferred into the contract in line 1168, and the total value of $ttUSD to be minted in line 1170. If this value is manipulated to be excessively high, $ttUSD can be excessively minted.

Pool.sol

```
1151 function mint(
1152     uint256 _collateral_amount,
1153     uint256 _share_amount,
1154     uint256 _dollar_out_min
1155 ) external {
1156     require(mint_paused == false, "Minting is paused");
1157     if (mint_update_oracle) {
1158         ITreasury(treasury).updateOracleShare();
```

```
1159        }
1160        (, uint256 _share_price, , uint256 _tcr, , , uint256 _minting_fee, ) =
1161            ITreasury(treasury).info();
1162        require(_share_price > 0, "Invalid share price");
1163        uint256 _price_collateral = getCollateralPrice();
1164        uint256 _total_dollar_value = 0;
1165        uint256 _required_share_amount = 0;
1166        if (_tcr > 0) {
1167            uint256 _collateral_value =
1168                (_collateral_amount * _price_collateral) /
1169                    PRICE_PRECISION;
1170            _total_dollar_value = (_collateral_value * COLLATERAL_RATIO_PRECISION)
    / _tcr;
1171            if (_tcr < COLLATERAL_RATIO_MAX) {
1172                _required_share_amount =
1173                    ((_total_dollar_value - _collateral_value) * PRICE_PRECISION) /
1174                    _share_price;
1175            }
1176        } else {
1177            _total_dollar_value = (_share_amount * _share_price) / PRICE_PRECISION;
1178            _required_share_amount = _share_amount;
1179        }
1180        uint256 _actual_dollar_amount =
            _total_dollar_value - ((_total_dollar_value * _minting_fee) /
1181  PRICE_PRECISION);
1182        require(_dollar_out_min <= _actual_dollar_amount, "slippage");
1183
1184        if (_required_share_amount > 0) {
1185            require(_required_share_amount <= _share_amount, "Not enough SHARE
    input");
1186            _transferShareToReserve(msg.sender, _required_share_amount);
1187        }
1188        if (_collateral_amount > 0) {
1189            _transferCollateralToReserve(msg.sender, _collateral_amount);
1190        }
1191        IDollar(dollar).poolMint(msg.sender, _actual_dollar_amount);
1192        ITreasury(treasury).updateCollateralMintProfit(_collateral_amount);
1193    }
```

The `getCollateralPrice()` function calls the `consult()` function from `oracle` to get the price.

Pool.sol

```
1145  function getCollateralPrice() public view override returns (uint256) {
1146      return IOracle(oracle).consult();
1147  }
```

However, the `oracle` can be set by the owner using the `setOracle()` function. Therefore, if the owner sets the `oracle` to a malicious contract, the owner can freely manipulate the source of price.

Pool.sol

```
1362 function setOracle(address _oracle) external onlyOwner {
1363     require(_oracle != address(0), "Invalid address");
1364     oracle = _oracle;
1365 }
```

The following functions can be used to control the source of price feeds.

Target	Function
CollateralOracle (L:166)	setChainlinkCollateralUsd()
CollateralRatioPolicy (L:1133)	setPriceTarget()
CollateralRatioPolicy (L:1151)	setDollar()
CollateralRatioPolicy (L:1171)	setOracleDollar()
DollarOracle (L:192)	setOracleCollateralUsd()
DollarOracle (L:196)	setOracleDollarCollateral()
MultiPairOracle (L:339)	addPair()
MultiPairOracle (L:347)	removePair()
PcsPairOracle (L:527)	setPeriod()
Pool (L:1362)	setOracle()
ShareOracle (L:409)	setChainlinkBnbUsd()
ShareOracle (L:420)	setOracleShareBnb()
Treasury (L:1348)	setOracleDollar()
Treasury (L:1353)	setOracleShare()
Treasury (L:1358)	setOracleCollateral()
Treasury (L:1363)	setCollateralAddress()
ZapPool (L:1435)	setOracle()

5.4.2. Remediation

Due to the fact that the functions that can control source of price feeds, will be able to cause very high impact to the platform, Inspex suggests limiting the use of them via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a Timelock contract to delay the changes for a significant amount of time e.g., 3 days

5.5. Centralized Control of State Variable

ID	IDX-005
Target	CollateralOracle CollateralRatioPolicy DollarOracle MultiPairOracle PcsPairOracle Pool ProfitFund ShareOracle TreasuryPolicy TTUSD ZapPool
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity: High** **Impact: Medium** The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood: High** There is nothing to restrict the changes from being done; however, the changes are limited by fixed values in the smart contracts.
Status	**Resolved** TukTuk Finance team has solved this issue by implementing the timelock mechanism with 3 days delay. `Timelock` contract address: https://bscscan.com/address/0xc16f05324a94e9964f4be619207ada26f1964bc6 Ownership transfer of `CollateralOracle` to `Timelock` contract: https://bscscan.com/tx/0x40359915d28e99db2b03b7365969565c0e84412af7d1b25956634f83ffb093db#eventlog Ownership transfer of `CollateralRatioPolicy` to `Timelock` contract: https://bscscan.com/tx/0x4f7126fe40298e20199acbad46b8cd461ea14aa9bcb733cfe22553b84a6e1e20#eventlog Ownership transfer of `DollarOracle` to `Timelock` contract: https://bscscan.com/tx/0x0a80c386abda9e2b52c6b11dca24334a297828722010a43283901ff67eca092f#eventlog

	Ownership transfer of `MultiPairOracle` to `Timelock` contract: https://bscscan.com/tx/0x9e390119058f645e7172b4c640ac14203ea1a663fdb3598fb9ae242ba60daffb#eventlog Ownership transfer of `PcsPairOracle` to `Timelock` contract: https://bscscan.com/tx/0x036e1ba736c341de2280ca7bd4e60e6bacb0e77e4e0b85043dc6fbdb58fd2755#eventlog Ownership transfer of `Pool` to `Timelock` contract: https://bscscan.com/tx/0xdfc67759da5c961370a2546421958d24f788be7ef9598629f31a34817d191f04#eventlog Ownership transfer of `PoolV2` to `Timelock` contract: https://bscscan.com/tx/0x986f249557537875befea0f749da57b031cb36cf4f8a385ec2134d94d854af4f#eventlog Ownership transfer of `ProfitFund` to `Timelock` contract: https://bscscan.com/tx/0x3753123fe621c8dcc16c73580915932ee157ad447d7d02553e7549db66596140#eventlog Ownership transfer of `ShareOracle` to `Timelock` contract: https://bscscan.com/tx/0x7ce96dab8e89f0e5b02a5c8bb7d4bc6355a91cfd43ac9ab866a0529acebe9582#eventlog Ownership transfer of `TreasuryPolicy` to `Timelock` contract: https://bscscan.com/tx/0x61dc3f4bb60870ba272b2d99b3790ba8995034adbe974a40b1d00b18d321b48d#eventlog The owner has renounced the ownership of the `TTUSD` contract in the following transaction: https://bscscan.com/tx/0x5f67e3a6cf860764b30253be43a2c06f338d40c52132c209e1bcdc5c648d295c#eventlog Ownership transfer of `ZapPool` to `Timelock` contract: https://bscscan.com/tx/0x677eb140508573c7ea98dbf1281a2578ce840dfd7eeaf1d37850b34135bb0ba3#eventlog

5.5.1. Description

Critical state variables can be updated any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

Target	Function	Modifier

CollateralOracle (L:166)	setChainlinkCollateralUsd()	onlyOwner
CollateralRatioPolicy (L:1129)	setRatioStep()	onlyOwner
CollateralRatioPolicy (L:1133)	setPriceTarget()	onlyOwner
CollateralRatioPolicy (L:1137)	setRefreshCooldown()	onlyOwner
CollateralRatioPolicy (L:1141)	setPriceBand()	onlyOwner
CollateralRatioPolicy (L:1145)	setTreasury()	onlyOwner
CollateralRatioPolicy (L:1151)	setDollar()	onlyOwner
CollateralRatioPolicy (L:1157)	reset()	onlyOwner
CollateralRatioPolicy (L:1163)	toggleCollateralRatio()	onlyOwner
CollateralRatioPolicy (L:1167)	toggleEffectiveCollateralRatio()	onlyOwner
CollateralRatioPolicy (L:1171)	setOracleDollar()	onlyOwner
DollarOracle (L:192)	setOracleCollateralUsd()	onlyOwner
DollarOracle (L:196)	setOracleDollarCollateral()	onlyOwner
MultiPairOracle (L:339)	addPair()	onlyOwner
MultiPairOracle (L:347)	removePair()	onlyOwner
PcsPairOracle (L:527)	setPeriod()	onlyOwner
Pool (L:1346)	toggleMinting()	onlyOwner
Pool (L:1350)	toggleRedeeming()	onlyOwner
Pool (L:1354)	setMintUpdateOracle()	onlyOwner
Pool (L:1358)	setRedeemUpdateOracle()	onlyOwner
Pool (L:1362)	setOracle()	onlyOwner
Pool (L:1367)	setRedemptionDelay()	onlyOwner
Pool (L:1371)	setTreasury()	onlyOwner
ProfitFund (L:1107)	addPool()	onlyOwner
ProfitFund (L:1115)	removePool()	onlyOwner
ProfitFund (L:1122)	rescueFund()	onlyOwner

ShareOracle (L:416)	setChainlinkBnbUsd()	onlyOwner
ShareOracle (L:420)	setOracleShareBnb()	onlyOwner
TreasuryPolicy (L:214)	setTreasury()	onlyOwner
TreasuryPolicy (L:220)	setRedemptionFee()	onlyOwner
TreasuryPolicy (L:225)	setMintingFee()	onlyOwner
TreasuryPolicy (L:230)	setExcessCollateralSafetyMargin()	onlyOwner
TreasuryPolicy (L:235)	setIdleCollateralUtilizationRatio()	onlyOwner
TreasuryPolicy (L:240)	setReservedCollateralThreshold()	onlyOwner
TTUSD (L:1218)	setTreasuryAddress()	onlyOwner
ZapPool (L:1406)	setMintUpdateOracle()	onlyOwner
ZapPool (L:1410)	addWhitelistContract()	onlyOwner
ZapPool (L:1416)	removeWhitelistContract()	onlyOwner
ZapPool (L:1421)	toggleMinting()	onlyOwner
ZapPool (L:1425)	setSlippage()	onlyOwner
ZapPool (L:1430)	setTreasury()	onlyOwner
ZapPool (L:1435)	setOracle()	onlyOwner
ZapPool (L:1440)	setRouter()	onlyOwner
Ownable	renounceOwnership()	onlyOwner
Ownable	transferOwnership()	onlyOwner

Please note that the `Ownable` contract is inherited from OpenZeppelin library and `contracts/libs/Ownable.sol`.

5.5.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a Timelock contract to delay the changes for a reasonable amount of time

5.6. Improper Deduction of User's Share

ID	IDX-006
Target	Pool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The user will lose the pending share token that should be redeemable. **Likelihood:** Medium This function is not used in the normal situation; however, it is possible that this function is needed when the reserve of $TUK inside the collateral reserve is not enough for the user to redeem.
Status	Resolved * TukTuk Finance Team team has clarified that the `collectRedemptionWithoutShare()` function is working correctly as designed, and the users can make the decision whether they want to collect $BUSD only or not.

5.6.1. Description

The `collectRedemptionWithoutShare()` function in `Pool` contract can be used to claim the collateral token ($BUSD) without claiming the share token ($TUK). However, the amount of the user's redeemable share is set to 0 even when that amount is not collected.

Pool.sol

```
1293 function collectRedemptionWithoutShare() external {
1294     require(
1295         (last_redeemed[msg.sender] + redemption_delay) <= block.number,
1296         "<redemption_delay"
1297     );
1298
1299     bool _send_collateral = false;
1300     uint256 _share_amount;
1301     uint256 _collateral_amount;
1302
1303     // Use Checks-Effects-Interactions pattern
1304     if (redeem_share_balances[msg.sender] > 0) {
1305         _share_amount = redeem_share_balances[msg.sender];
1306         redeem_share_balances[msg.sender] = 0;
1307         unclaimed_pool_share = unclaimed_pool_share - _share_amount;
1308     }
```

```
1309
1310        if (redeem_collateral_balances[msg.sender] > 0) {
1311            _collateral_amount = redeem_collateral_balances[msg.sender];
1312            redeem_collateral_balances[msg.sender] = 0;
1313            unclaimed_pool_collateral = unclaimed_pool_collateral -
    _collateral_amount;
1314            _send_collateral = true;
1315        }
1316
1317        if (_send_collateral) {
1318            _requestTransferCollateral(msg.sender, _collateral_amount);
1319        }
1320    }
```

5.6.2. Remediation

Inspex suggests removing code responsible for the share deduction from the `collectRedemptionWithoutShare()` function, for example:

Pool.sol

```
1293    function collectRedemptionWithoutShare() external {
1294        require(
1295            (last_redeemed[msg.sender] + redemption_delay) <= block.number,
1296            "<redemption_delay"
1297        );
1298
1299        bool _send_collateral = false;
1300        uint256 _collateral_amount;
1301
1302        // Use Checks-Effects-Interactions pattern
1303        if (redeem_collateral_balances[msg.sender] > 0) {
1304            _collateral_amount = redeem_collateral_balances[msg.sender];
1305            redeem_collateral_balances[msg.sender] = 0;
1306            unclaimed_pool_collateral = unclaimed_pool_collateral -
    _collateral_amount;
1307            _send_collateral = true;
1308        }
1309
1310        if (_send_collateral) {
1311            _requestTransferCollateral(msg.sender, _collateral_amount);
1312        }
1313    }
```

5.7. Short MINIMUM_DELAY Time in Timelock

ID	IDX-007
Target	CollateralReserve Treasury TreasuryVaultAlpaca
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium The only restriction deterring the owner from performing this attack is the Timelock contract with a short minimum delay of 6 hours.
Status	Resolved TukTuk Finance team has resolved this issue by implementing new `Timelock` with 24 hours delay as recommended in the following contract: https://bscscan.com/address/0xc16F05324A94E9964F4bE619207ADa26f1964bC6#code

5.7.1. Description

At the time of the audit, contacts `CollateralReserve`, `Treasury`, and `TreasuryVaultAlpaca` has been deployed on the Binance Smart Chain with `Timelock` mechanism at address `0xee25b3b338487dd29a2c4a6a0dcfda31802e03b1`. The timelock mechanism sets a time delay before any change can be made to the smart contract source code, and will provide time for users to monitor the changes and take action safely before any potentially unwanted change has taken effect.

Anywise, the `MINIMUM_DELAY` in the `Timelock` contact has been set to 21600 seconds (6 hours) which is too short. The 6 hours delay of changes that can be made to the smart contract is too short for the user to take action before those changes occur, allowing the owner to potentially execute the following functions for profits:

The controllable privileged state update functions are as follows:

Target	Function	Modifier
CollateralReserve (L:616)	setTreasury()	onlyOwner
Treasury (L:1314)	addPool()	onlyOwner

Treasury (L:1322)	removePool()	onlyOwner
Treasury (L:1336)	setTreasuryPolicy()	onlyOwner
Treasury (L:1342)	setCollateralRatioPolicy()	onlyOwner
Treasury (L:1348)	setOracleDollar()	onlyOwner
Treasury (L:1353)	setOracleShare()	onlyOwner
Treasury (L:1358)	setOracleCollateral()	onlyOwner
Treasury (L:1363)	setCollateralAddress()	onlyOwner
Treasury (L:1368)	setCollateralReserve()	onlyOwner
Treasury (L:1374)	setProfitSharingFund()	onlyOwner
Treasury (L:1386)	setController()	onlyOwner
Treasury (L:136)	setVault()	onlyController
Treasury (L:1394)	executeTransaction()	onlyOwner
TreasuryVaultAlpaca (L:765)	claimIncetiveReward()	onlyOwner
TreasuryVaultAlpaca (L:778)	setTreasury()	onlyOwner
TreasuryVaultAlpaca (L:784)	setVaultProfitFund()	onlyOwner
TreasuryVaultAlpaca (L:790)	setStakingPool()	onlyOwner
TreasuryVaultAlpaca (L:797)	setPauseVault()	onlyOwner
TreasuryVaultAlpaca (L:802)	setIbAsset()	onlyOwner
TreasuryVaultAlpaca (L:808)	setReward()	onlyOwner
TreasuryVaultAlpaca (L:816)	executeTransaction()	onlyOwner

5.7.2. Remediation

Inspex recommends setting `MINIMUM_DELAY` in the `Timelock` contract to at least 24 hours.

5.8. ZapPool Price Impact Calculation

ID	IDX-008
Target	ZapPool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Low The platform will lose a part of the share token to the price impact every time the `zapMint()` function is executed. **Likelihood:** Medium It is likely that users will use the `zapMint()` function when they want to swap $BUSD to $TUK with high price impact.
Status	Resolved TukTuk Finance team has resolved this issue by deducting $ttUSD amount from swapping slippage in commit `9c1d23645f33ac0f9dc684bf86b67a7f410c2e9a` and deploying new `ZapPool`. New `ZapPool` contract: https://bscscan.com/address/0x293b0A500b9ea286108dD0Ecb9FCBaB03831c397

5.8.1. Description

In order to mint $ttUSD, users need to collateralize their $BUSD and $TUK to the platform via `Pool` contract. Users can also collateralize $BUSD only via `ZapPool` contract, in which a portion of $BUSD will be swapped to $TUK through the `router.swapExactTokensForTokens()` function in the `zapMint()` function.

ZapPool.sol

```
1354 function zapMint(uint256 _collateral_amount, uint256 _dollar_out_min) external
     onlyUserOrWhitelistedContracts nonReentrant {
1355     require(mint_paused == false, "Minting is paused");
1356     if (mint_update_oracle) {
1357         ITreasury(treasury).updateOracleShare();
1358     }
1359     (, uint256 _share_price, , uint256 _tcr, , , uint256 _minting_fee, ) =
     ITreasury(treasury).info();
1360     require(_share_price > 0, "Invalid share price");
1361     uint256 _price_collateral = getCollateralPrice();
1362
1363     uint256 _collateral_value = (_collateral_amount * _price_collateral) /
     PRICE_PRECISION;
```

```
1364        uint256 _actual_dollar_amount = _collateral_value - ((_collateral_value *
    _minting_fee) / PRICE_PRECISION);
1365        require(_actual_dollar_amount >= _dollar_out_min, "slippage");
1366
1367        collateral.safeTransferFrom(msg.sender, address(this), _collateral_amount);
1368        if (_tcr < COLLATERAL_RATIO_MAX) {
1369            uint256 _share_value = (_collateral_value * (RATIO_PRECISION - _tcr)) /
    RATIO_PRECISION;
1370            uint256 _min_share_amount = (_share_value * PRICE_PRECISION *
    (RATIO_PRECISION - slippage)) / _share_price / RATIO_PRECISION;
1371            uint256 _swap_collateral_amount = (_collateral_amount *
    (RATIO_PRECISION - _tcr)) / RATIO_PRECISION;
1372            collateral.safeApprove(address(router), 0);
1373            collateral.safeApprove(address(router), _swap_collateral_amount);
1374            uint256[] memory _received_amounts =
    router.swapExactTokensForTokens(_swap_collateral_amount, _min_share_amount,
    router_path, address(this), block.timestamp + LIMIT_SWAP_TIME);
1375            emit ZapSwapped(_swap_collateral_amount,
    _received_amounts[_received_amounts.length - 1]);
1376        }
1377
1378        uint256 _balanceShare = ERC20(address(share)).balanceOf(address(this));
1379        uint256 _balanceCollateral = collateral.balanceOf(address(this));
1380        if (_balanceShare > 0) {
1381            _transferShareToReserve(_balanceShare);
1382        }
1383        if (_balanceCollateral > 0) {
1384            _transferCollateralToReserve(_balanceCollateral); // transfer all
    collateral to reserve no matter what;
1385        }
1386        dollar.poolMint(msg.sender, _actual_dollar_amount);
1387        ITreasury(treasury).updateCollateralMintProfit(_balanceCollateral);
1388    }
```

When swapping $BUSD to $TUK, the price impact and slippage is limited using `_min_share_amount`, but it does not affect the amount of minted $ttUSD. This is because `_actual_dollar_amount` depends on `_price_collateral` and `_minting_fee` only.

Therefore, the platform will lose a portion of **share** token, $TUK, for the $ttUSD minters when they redeem $ttUSD back to $BUSD and $TUK.

For example:

Assuming that:

- Price of 1 $TUK is 1 $BUSD
- Price impact is 5%

- $BUSD price from the oracle is 1
- Minting fee is 0

Users can swap 10,000 $BUSD to $TUK with 5% of the price impact in the AMM, they will get 9,500 $TUK.

Alternatively, users can mint 100,000 $ttUSD via `zapMint()` with 100,000 $BUSD. `ZapPool` contract swaps 10,000 $BUSD to 9,500 $TUK. Therefore, `ZapPool` has 90,000 $BUSD and 9,500 $TUK, in which 500 $TUK is missing due to the 5% price impact.

However, when users `redeem()` 100,000 $ttUSD back, they receive 90,000 $BUSD and 10,000 $TUK.

As a result, the users can swap $BUSD to $TUK with zero price impact and the platform takes this loss instead.

5.8.2. Remediation

Inspex suggests that the platform should take the price impact into consideration for the amount of $ttUSD minted.

There are multiple solutions to this problem, for example, assuming that the `tcr` is 90% and `slippage` is 5%:

- Add extra 0.5% slippage collateral when the user calls the `zapMint()` function. This extra collateral will be used as an additional input amount (`_swap_collateral_amount` * 1.005) to swap $BUSD to $TUK, which will result in a compensation of swap slippage loss. The collateral left after swapping should be returned to the minter.
- Mint the real amount of $ttUSD after deducting from swap slippage. Deducting 0.5% slippage from `_collateral_amount` first and swap the rest. This will result in less $ttUSD amount for users, but no extra charges. The collateral left after swapping should be returned to the minter.

5.9. Improper Account Type Checking

ID	IDX-009
Target	ZapPool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Very Low **Impact:** Low Functions with `onlyUserOrWhitelistedContracts()` modifier cannot restrict calling from smart contracts as it is intended for. **Likelihood:** Low There is no significant profit to be gained, resulting in low motivation for the attack.
Status	Resolved TukTuk Finance team has resolved this issue as suggested in commit `9c1d23645f33ac0f9dc684bf86b67a7f410c2e9a` and deploying new `ZapPool`. New `ZapPool` contract: https://bscscan.com/address/0x293b0A500b9ea286108dD0Ecb9FCBaB03831c397

5.9.1. Description

The `!msg.sender.isContract()` condition in line 1320 of `onlyUserOrWhitelistedContracts()` modifier is used to validate whether `msg.sender` is a smart contract or not as shown below:

ZapPool.sol

```
1319 modifier onlyUserOrWhitelistedContracts() {
1320     require(!msg.sender.isContract() || whitelistContracts[msg.sender], "Allow
     non-contract only");
1321     _;
1322 }
```

The `isContract()` function of `Address.sol` library checks `EXTCODESIZE` opcode which returns the size of the bytecode on an address. If the size is larger than zero, the address is a contract.

Address.sol

```
1 function isContract(address account) internal view returns (bool) {
2     // This method relies on extcodesize, which returns 0 for contracts in
3     // construction, since the code is only stored at the end of the
4     // constructor execution.
5
```

```
    uint256 size;
    // solhint-disable-next-line no-inline-assembly
    assembly {
        size := extcodesize(account)
    }
    return size > 0;
}
```

However, the bytecode is stored at the end of the constructor function call. Therefore, calling the affected functions from within the constructor will cause the `EXTCODESIZE` to return 0. As a result, the `isContract()` can return false when called from the constructor function of a newly deployed contract.

The following code is an example of contract that can bypass the condition check in the `onlyUserOrWhitelistedContracts()` modifier of `ZapPool.zapMint()` function:

BypassOnlyUser.sol

```
contract BypassOnlyUser {

    IZapPool public zappool;

    constructor(IZapPool _zappool) {
        zappool = _zappool;
        zappool.zapMint();
    }
}
```

5.9.2. Remediation

Inspex suggests checking that the caller is the smart contract or not by comparing `msg.sender` with `tx.origin`. The `tx.origin` returns the transaction creation address. If the `tx.origin` is not equal to `msg.sender`, the caller will not be an externally-owned account (EOA).

ZapPool.sol

```
modifier onlyUserOrWhitelistedContracts() {
    require(msg.sender == tx.origin || whitelistContracts[msg.sender], "Allow
non-contract only");
    _;
}
```

5.10. Improper Collateral Reserve Amount

ID	IDX-010
Target	TTUSD
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low The total amount of collateral stored in the reserve pool is less than what it should be. Thus, not all $ttUSD can be redeemed. **Likelihood:** Low It is unlikely that the reserve pool will not have enough collateral token for redeeming.
Status	Resolved * TukTuk Finance team has clarified that this genesis supply minting is known to the users, and has included this information in the documentation here: https://tuktukfinance.gitbook.io/tuktuk-finance/tuktuk-space/ttusd-stable-coin

5.10.1. Description

The `initialize()` function was called when the $ttUSD token had been deployed. In this function, the 5,000 $ttUSD (`genesis_supply`) were minted without collateralization.

TTUSD.sol

```
1195 function initialize(
1196     string memory _name,
1197     string memory _symbol,
1198     address _treasury
1199 ) external initializer onlyOwner {
1200     name = _name;
1201     symbol = _symbol;
1202     treasury = _treasury;
1203     _mint(msg.sender, genesis_supply);
1204 }
```

Therefore, the total amount of collateral stored in the reserve pool is less than it should be. Thus, the $ttUSD minted from `initialize()` function cannot be redeemed.

5.10.2. Remediation

Inspex suggests removing the mint capability from the `initialze()` function. It is recommended to directly mint the genesis supply $ttUSD from the `Pool` contract.

TTUSD.sol

```
1195 function initialize(
1196     string memory _name,
1197     string memory _symbol,
1198     address _treasury
1199 ) external initializer onlyOwner {
1200     name = _name;
1201     symbol = _symbol;
1202     treasury = _treasury;
1203 }
```

5.11. Outdated Compiler Version

ID	IDX-011
Target	CollateralOracle CollateralRatioPolicy CollateralReserve DollarOracle MultiPairOracle PcsPairOracle Pool ProfitFund ShareOracle Treasury TreasuryPolicy TreasuryVaultAlpaca TTUSD ZapPool
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Very Low **Impact:** Low From the list of known Solidity bugs, direct impact cannot be caused from those bugs themselves. **Likelihood:** Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	**Acknowledged** TukTuk Finance team has acknowledged this issue. The fix is not yet deployed on the mainnet; however, this issue was resolved as suggested in commit `9c1d23645f33ac0f9dc684bf86b67a7f410c2e9a`. In the future deployment, code in this commit will not face this issue.

5.11.1. Description

The Solidity compiler versions specified in the smart contracts were outdated as seen in the example below. It may potentially be used to cause damage to the smart contracts or the users of the smart contract.

Pool.sol

```
1062 pragma solidity 0.8.4;
1063
```

```
1064 contract Pool is Ownable, ReentrancyGuard, Initializable, IPool {
1065     using SafeERC20 for ERC20;
```

The following table contains all targets which the compiler version is outdated.

Target	Version
CollateralOracle	0.8.4
CollateralRatioPolicy	0.8.4
CollateralReserve	0.8.4
DollarOracle	0.8.4
MultiPairOracle	0.8.4
PcsPairOracle	0.8.4
Pool	0.8.4
ProfitFund	0.8.4
ShareOracle	0.8.4
Treasury	0.8.4
TreasuryPolicy	0.8.4
TreasuryVaultAlpaca	0.8.4
TTUSD	0.8.4
ZapPool	0.8.4

5.11.2. Remediation

Inspex suggests upgrading the Solidity compiler to the latest stable version[2].

During the audit activity, the latest stable versions of Solidity compiler in major 0.8 is 0.8.6, For example:

Pool.sol

```
1062 pragma solidity 0.8.6;
1063
1064 contract Pool is Ownable, ReentrancyGuard, Initializable, IPool {
1065     using SafeERC20 for ERC20;
```

5.12. Oracle Denial of Service

ID	IDX-012
Target	PcsPairOracle
Category	General Smart Contract Vulnerability
CWE	CWE-755: Improper Handling of Exceptional Conditions
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	No Security Impact TukTuk Finance team has acknowledged this issue. The fix is not yet deployed on the mainnet; however, this issue was resolved as suggested in commit `9c1d23645f33ac0f9dc684bf86b67a7f410c2e9a`. In the future deployment, code in this commit will not face this issue.

5.12.1. Description

The `PcsPairOracle` contract is designed to allow integer overflow and underflow behaviors. However, the solidity compiler version 0.8 has the built-in integer overflow and underflow protections. The transaction will be rejected when integer overflow or underflow behaviors occur as shown in the following example source code in line 603, 606, and 608:

PcsPairOracle.sol

```
584 function currentCumulativePrices(address _pair)
585     internal
586     view
587     returns (
588         uint256 price0Cumulative,
589         uint256 price1Cumulative,
590         uint32 blockTimestamp
591     )
592 {
593     blockTimestamp = currentBlockTimestamp();
594     IUniswapLP uniswapPair = IUniswapLP(_pair);
595     price0Cumulative = uniswapPair.price0CumulativeLast();
596     price1Cumulative = uniswapPair.price1CumulativeLast();
597 
598     // if time has elapsed since the last update on the pair, mock the
    accumulated price values
599     (uint112 reserve0, uint112 reserve1, uint32 _blockTimestampLast) =
600         uniswapPair.getReserves();
```

```
601     if (_blockTimestampLast != blockTimestamp) {
602         // subtraction overflow is desired
603         uint32 timeElapsed = blockTimestamp - _blockTimestampLast;
604         // addition overflow is desired
605         // counterfactual
606         price0Cumulative += uint256(FixedPoint.fraction(reserve1, reserve0)._x)
    * timeElapsed;
607         // counterfactual
608         price1Cumulative += uint256(FixedPoint.fraction(reserve0, reserve1)._x)
    * timeElapsed;
609     }
610 }
```

The following lines of code in `PcsPairOracle` contract are affected:

Function	Line
isCanUpdate	533
update	543
update	554
update	557
currentCumulativePrices	603
currentCumulativePrices	606
currentCumulativePrices	608

5.12.2. Remediation

Inspex suggests covering code area that desires integer overflow and underflow behaviors with the **unchecked** block as in the following example:

PcsPairOracle.sol

```
584 function currentCumulativePrices(address _pair)
585     internal
586     view
587     returns (
588         uint256 price0Cumulative,
589         uint256 price1Cumulative,
590         uint32 blockTimestamp
591     )
592 {
593     blockTimestamp = currentBlockTimestamp();
594     IUniswapLP uniswapPair = IUniswapLP(_pair);
```

```
595     price0Cumulative = uniswapPair.price0CumulativeLast();
596     price1Cumulative = uniswapPair.price1CumulativeLast();
597
598     // if time has elapsed since the last update on the pair, mock the
    accumulated price values
599     (uint112 reserve0, uint112 reserve1, uint32 _blockTimestampLast) =
600         uniswapPair.getReserves();
601     if (_blockTimestampLast != blockTimestamp) {
602         unchecked {
603             // subtraction overflow is desired
604             uint32 timeElapsed = blockTimestamp - _blockTimestampLast;
605             // addition overflow is desired
606             // counterfactual
607             price0Cumulative += uint256(FixedPoint.fraction(reserve1,
    reserve0)._x) * timeElapsed;
608             // counterfactual
609             price1Cumulative += uint256(FixedPoint.fraction(reserve0,
    reserve1)._x) * timeElapsed;
610         }
611     }
612 }
```

5.13. Improper Function Visibility

ID	IDX-013
Target	CollateralRatioPolicy CollateralReserve ProfitFund Treasury TreasuryVaultAlpaca TTUSD ZapPool
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact TukTuk Finance team has acknowledged this issue. The fix is not yet deployed on the mainnet; however, this issue was resolved as suggested in commit `9c1d23645f33ac0f9dc684bf86b67a7f410c2e9a`. In the future deployment, code in this commit will not face this issue.

5.13.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

The following source code shows that the `setRatioStep()` function of the `CollateralRatioPolicy` is set to public and it is never called from any internal function.

CollateralRatioPolicy.sol

```
1129 function setRatioStep(uint256 _ratio_step) public onlyOwner {
1130     ratio_step = _ratio_step;
1131 }
```

The following table contains all functions that have `public` visibility and are never called from any internal function.

Target	Function
CollateralRatioPolicy.sol (L: 1091)	refreshCollateralRatio()

CollateralRatioPolicy.sol (L: 1129)	setRatioStep()
CollateralRatioPolicy.sol (L: 1133)	setPriceTarget()
CollateralRatioPolicy.sol (L: 1137)	setRefreshCooldown()
CollateralRatioPolicy.sol (L: 1163)	toggleCollateralRatio()
CollateralRatioPolicy.sol (L: 1167)	toggleEffectiveCollateralRatio()
CollateralRatioPolicy.sol (L: 1171)	setOracleDollar()
CollateralReserve.sol (L: 600)	fundBalance()
CollateralReserve.sol (L: 606)	transferTo()
CollateralReserve.sol (L: 616)	setTreasury()
ProfitFund.sol (L: 1088)	balance()
ProfitFund.sol (L: 1094)	transferTo()
ProfitFund.sol (L: 1107)	addPool()
ProfitFund.sol (L: 1115)	removePool()
ProfitFund.sol (L: 1122)	rescueFund()
Treasury.sol (L: 1214)	updateOracleDollar()
Treasury.sol (L: 1218)	updateOracleShare()
Treasury.sol (L: 1222)	recallFromVault()
Treasury.sol (L: 1226)	enterVault()
Treasury.sol (L: 1314)	addPool()
Treasury.sol (L: 1322)	removePool()
Treasury.sol (L: 1394)	executeTransaction()
TreasuryVaultAlpaca.sol (L: 816)	executeTransaction()
TTUSD.sol (L: 1218)	setTreasuryAddress()
ZapPool.sol (L: 1350)	unclaimed_pool_collateral()

5.13.2. Remediation

Inspex suggests changing all functions' visibility to `external` if they are not called from any `internal` function as shown in the following example:

CollateralRatioPolicy.sol

```
1129 function setRatioStep(uint256 _ratio_step) external onlyOwner {
1130     ratio_step = _ratio_step;
1131 }
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] “OWASP Risk Rating Methodology.” [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

[2] “Releases — Ethereum Solidity Releases” [Online]. Available: https://github.com/ethereum/solidity/releases. [Accessed: 20-July-2021]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE