

# CHAINEDTHRIFT SMART CONTRACTS AUDITS REPORT

By Guild Audits



# TABLE OF CONTENTS

Executive Summary	2
Protocol Overview	3
Project Audit Scope and Findings	4
Mode of Audit and Methodologies	6
Types of Issues	7
Functional Testing	8
Report of Findings	10 - 22
Closing Summary	23
Appendix	23
Disclaimer	23



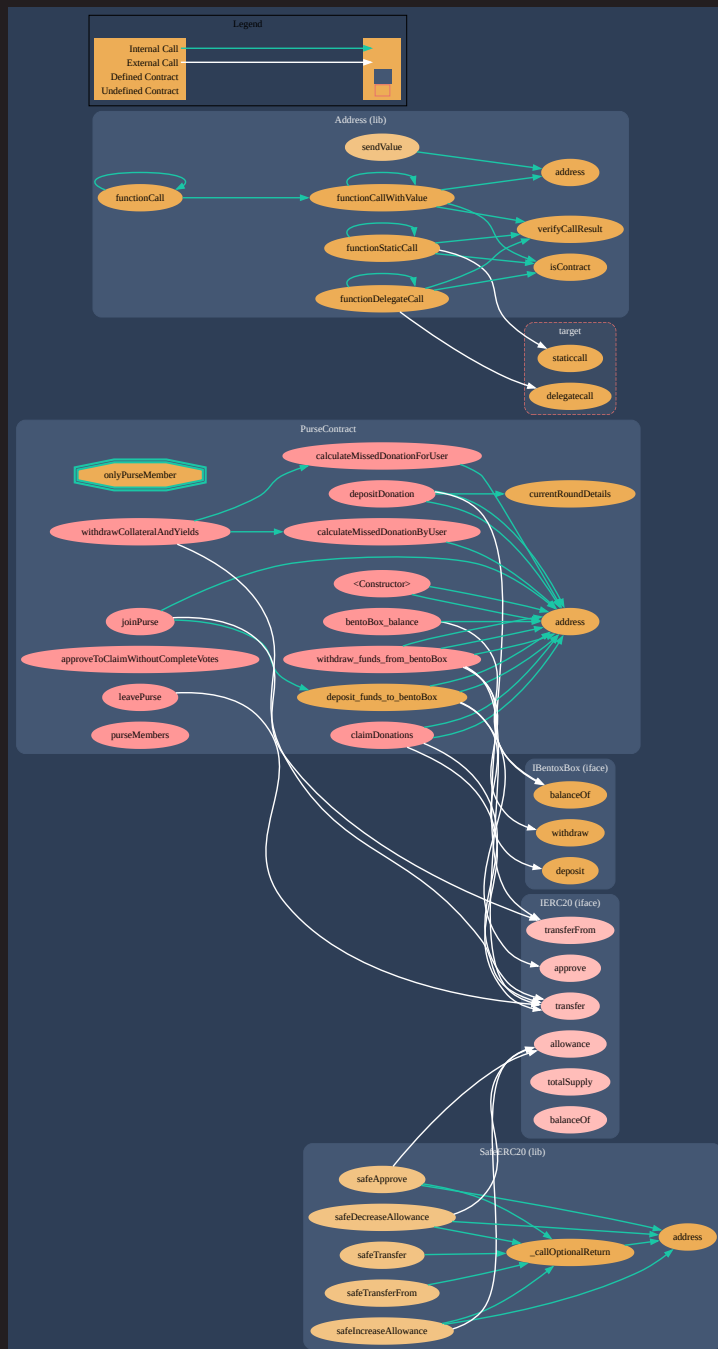
# EXECUTIVE SUMMARY

ChainedThrift is a decentralized finance application that is built on blockchain technology. It is built on the Polygon (Matic) blockchain network. Chained Thrift helps its users achieve their financial goals through its decentralized thrift saving scheme. It also provides autonomous trading options on its decentralized exchange.



# PROTOCOL OVERVIEW

## ChainedThrift



# PROJECT AUDIT SCOPE AND FINDINGS

The motive of this audit is to review the codebase of **ChainedThrift** contracts for the purpose of achieving secured, corrected and quality contracts.

## Number of Contracts in Scope:

- purse (540 SLOC)
- purse Factory (83 SLOC)

## Link to Project codebase:

Codebase: [GitHub](#)

Commit: [initial-audit](#)

[c926c4819d7bd5669a93cf47f439386a6579d383](#)

## Duration for Audit:

October 03, 2022, to February 2023

## Audit Methodology:

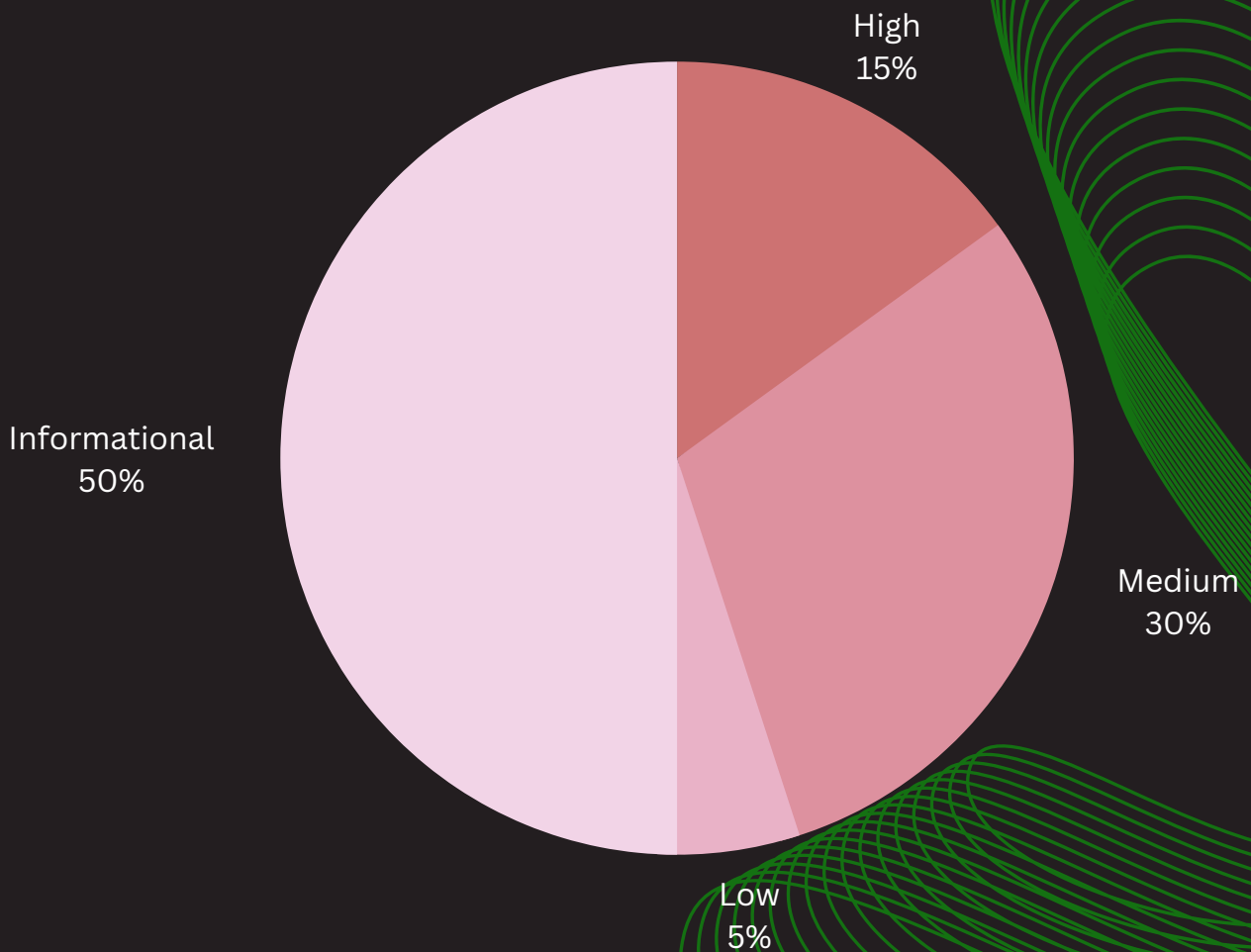
- Automated Testing
- Functional Testing
- Manual Review



### Issues found:

- Total issues: 19
- Total High: 3
- Total Medium: 6
- Total Low: 1
- Total Informational: 10

### Chart Illustration







# MODE OF AUDIT AND METHODOLOGIES

The mode of audit carried out in this audit process is as follow:

- **Manual Review:** This is the first and principal step carried out to understand the business logic behind a project. At this point, it involves research, discussion and establishment of familiarity with contracts. Manual review is critical to understand the nitty-gritty of the contracts.
- **Automated Testing:** This is the running of tests with audit tools to further cement discoveries from the manual review. After a manual review of a contract, the audit tools which could be Slither, Echidna, or Mythril are run on the contract to find out issues.
- **Functional Testing:** Functional testing involves manually running unit, static, and dynamic testing. This is done to find out possible exploit scenarios that could be used to steal money from the contracts. This helps understand the functionality of the contracts and find out lapses in the reverts check in contract.

The methodologies for establishing severity issues:

- High Level Severity Issues 
- Medium Level Severity Issues 
- Low Level Severity Issues 
- Informational Level Severity Issues 





# TYPES OF ISSUES

**Open:** Security vulnerabilities identified that must be resolved and are currently unresolved.

**Resolved:** These are the issues identified in the initial audit and have been successfully fixed.

**Acknowledged:** Vulnerabilities that have been acknowledged but are yet to be resolved.

**Partially Resolved:** Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.





# FUNCTIONAL TESTING

Some functional testing were carried out to help ascertain code security:

- ✓ Incorrect code Implementation
- ✓ Incorrect code Implementation
- ✓ Array length not validated
- ✓ Members can withdraw collateral before last collector
- ✓ Members can withdraw collateral before purse start
- ✓ Inability to get back collateral when the required number of members to start a purse is not met
- ✓ Purse contract locks Ether
- ✓ Looping storage variables
- ✓ Incorrect modifier use
- ✓ Inefficient storage layout
- ✓ Unused struct member
- ✓ Unused mapping declarations
- ✓ Unused declared data-structure



# FUNCTIONAL TESTING

- ✓ Long require statement reason string
- ✓ Incorrect spelling
- ✓ State variable that could be declared constant
- ✓ Write after write
- ✓ Function visibility can be changed to external
- ✓ Comparison with Boolean literal
- ✓ purse member who chose position 0 get their contribution locked forever.





**Recommendation:**

A local variable of uint256 num, should be created, the count will only be increased when the check is passed. See image below

```
uint256 num;
for (uint256 i = 0; i < members.length; i++) {
    if (
        members[i] != _memberAdress &&
        has_donated_for_member[_memberAdress
            [members[i]] == false
    ) {
        members_who_member_didnt_donate_for[i] =
            (members[i]);
        num++;
    }
}
return (
    members_who_member_didnt_donate_for, num *
    purse.deposit_amount
);
```

Status: **Resolved**

- **Issue: Incorrect code Implementation (Context: purse.sol L420)**
  - The function **calculateMissedDonationForUser** was supposed to return an array of members that do not donate for '\_memberAdress' but because the array of members was loop using a fixed array in memory, this will initially populate the resulting array with zero address before pushing the member array. This cause wrong calculation of funds on



```
members_who_didnt_donate_for_user.length*  
purse.deposit_amount
```

- The resulting amount for this user will be multiplied by the original member that didn't donate \* the remaining zero address in the array.
- **Impact of the vulnerability :** Assuming no donation was missed for a particular user, the returned array will be filled with zero address in length of the total user in the thrift multiplied by the deposit amount. On withdrawing collateral and yield, the thrift will pay the user the deposit amount \* all the total number of thrift members.

#### Recommendation:

A local variable of uint256 num, should be created, the count will only be increased when the check is passed. See image below.

```
uint256 num;  
for (uint256 i = 0; i < members.length; i++) {  
    if (   
        members[i] != _memberAdress &&  
        has_donated_for_member[_memberAdress  
        [members[i]] == false  
    ) {  
        members_who_member_didnt_donate_for[i] =  
            (members[i]);  
        num++;  
    }  
}  
return (   
    members_who_member_didnt_donate_for,num *  
    purse.deposit_amount  
);
```

Status: **Resolved**



- **Issue: purse member who chose position 0 get their contribution locked forever. (Context: purse.sol)**

- Members are allowed to pick a position less than or equal to the length of members in the purse which means members can pick position 0 and position 0 is not factored in when paying out contribution to each round which can cause stuck of funds inside the contract forever.

**Recommendation:**

We recommend a check to prevent member from picking 0 as a position to prevent the sanity of it's member.

**Status: Resolved**



## MEDIUM SEVERITY ISSUES ^

- **Issue: Array length not validated( Context: `purse.sol` [L455](#) [L427](#) )**
  - The length of the memory array used to store the array of member who did not donate, will not catch the total members in the purse because the logic ("`i < array.length-1`") does not catch the full length of the members of the array of the purse because it is reduced by 1.
  - **Impact of the vulnerability :** The impact of the error will prompt the function to miss out from catching the last member of the array with an error message "Index out of bound" therefore the last member of the array won't be able to get their collateral and interest.

**Recommendation:**

We recommend that the "-1" in the for loop should be removed so as to be able to loop through all the members of the purse. See image below.

```
address[] memory members_who_didnt_donate_for_user
=newaddress[](
    members.length
);
```

Status: **Resolved**





- **Issue: Members can withdraw collateral before last collector(**

**Context: `purse.sol` [L474](#))**

- Doing an active purse, a purse member can call the withdraw function and successfully withdraw all their collateral, provided that the funds have not be sent to the **bentoBox** contract, they will continue to receive donations from other members and can decide to run after receiving their donations.
- **Impact of the vulnerability** : A member can defraud other purse members without having a collateral.

**Recommendation:**

We recommend that the **`deposit_funds_to_bentoBox`** function should be immediately called by the last person that joined the purse, these will stop any opportunity of member withdrawing their collateral before funds are sent to **bentoBox** contract

**Status: Resolved**

- **Issue: Members can withdraw collateral before purse start (**

**Context: `purse.sol` [L474](#))**

- A user can join a purse and withdraw their collateral before purse start, they will still be able to receive contributions from members in the purse after the start of the purse.
- **Impact of the vulnerability** : If the above happens, this will cause a loss of funds to the other members of the purse because the said user might decide to run away without paying donations to other members of the purse.



**Recommendation:**

We recommend adding a require statement to stop member from withdrawing their collateral before purse start.

**Status:** Resolved

- **Issue: Inability to get back collateral when the required number of members to start a purse is not met. ( Context: [purse.sol L171](#))**
  - In the contract, there is no provision for the purse members to be claim their money back when the duration of the purse last for a long time without the required amount of users joining the purse before it kicks off.
  - **Impact of the vulnerability :** The money will be trapped in the contract as far as the required numbers of members has not joined the purse.

**Recommendation:**

Members should be able to have options of getting the funds back when it takes a very long time to kick off. We recommend setting a time limit to start a purse.

**Status:** Resolved



- **Issue: Purse contract locks Ether ( Context: purse.sol [L171](#))**

- The purse.sol constructor is make as payable which means ether can be sent into the contract, but there is no function handling withdrawal capacity for ether
- **Impact of the vulnerability :** Every Ether sent into purse contract will be lost.

**Recommendation:**

Remove the payable attribute in the constructor as the purse is only using erc20 for transactions.

**Status: Resolved**

- **Issue: Incorrect modifier use ( Context: purse.sol [L160](#))**

- The modifier param \_address was not used in the modifier require statement, this expose the contract to possible vulnerabilities as the pass in address will not be checked.

**Recommendation:**

We recommend replacing the msg.sender in the require statement to the modifier param. See image below

```
modifieronlyPurseMember(address _address) {  
    require(isPurseMember[_address] ==true, "only  
    purse members please");  
    _;  
}
```

**Status: Resolved**



## LOW SEVERITY ISSUES

- **Issue: Looping storage variables (Context: `purse.sol` [L229](#))**
  - Looping variable in the storage cost gas consumption because every read to the storage consumes a high gas cost and looping should be avoided if possible to avoid users paying high gas cost for performing a transaction.

### **Recommendation:**

If looping will be used in a function, it is recommended to loop through a memory variable which will cost less gas: A memory variable should be declared and assign the state variable because looping through the memory variable cost less gas.

**Status:** **Partially Resolved**



## INFORMATIONAL SEVERITY ISSUES

- **Issue: Inefficient storage layout ( Context: `purse.sol` [L58](#))**
  - The **struct** `purse`' S declaration is inefficient and causes a stack too deep error as the struct's properties can be break into another struct.

### Recommendation:

We recommend to remove **uint256 purseld** has the variable is currently unused and taking a whole slot in storage.

Status: **Acknowledged**

- **Issue: Unused struct member ( Context: `purse.sol` [L68](#))**
  - The struct member `purseld` in the struct **purse** is unused and might cause some unnecessary gas usage and more bytecode during contract deployment.

### Recommendation:

We advise to remove **uint256 purseld** from the codebase to increase legibility.

Status: **Acknowledged**

- **Issue: Unused mapping declarations ( Context: `purse.sol` [L91](#), [L93](#), [L95](#))**
  - The mapping declarations on the aforementioned lines is unused or was forget to be implemented in the codebase.

### Recommendation:

We recommend removing the mapping declarations on the aforementioned lines to increase the legibility of the codebase.

Status: **Acknowledged**



- **Issue: Unused declared data-structure ( Context: `purse.sol` [L108](#))**

- The struct member `ToVotePurseState` declared on the aforementioned line is never used in the code.

**Recommendation:**

We recommend to remove the struct `memberToVotePurseState` on the aforementioned line to increase the legibility of the codebase as the struct is never used in the code.

**Status:** **Resolved**

- **Issue: long require statement reason string ( Context: `purse.sol` [L353](#), [L383](#))**

- The require call on the aforementioned lines have a lengthy reason string, the gas cost of using strings are expensive.

**Recommendation:**

We recommend a gas efficient method by using custom error with the require call.

**Status:** **Acknowledged**

- **Issue: Incorrect spelling ( Context: `purse.sol` [L212](#))**

- The comment on the aforementioned line has incorrect spelling for the word **donating**.

**Recommendation:**

We recommend to correct the word spellings on the aforementioned line.

**Status:** **Acknowledged**



- **Issue: State variable that could be declared constant ( Context: purse.sol L123)**

- The bentoBox\_address state variables is assigned to only once during the contract level declaration.

**Recommendation:**

We advise that the constant keyword is introduced in the variable declaration to greatly optimize the gas cost involved in utilizing the variable.

**Status:** **Resolved**

- **Issue: write after write ( Context: purse.sol L64)**

- `purse_count = purse_count++;`
- Here `purse_count` variable was written but never read and written again

**Recommendation:**

We recommend to change the line of code to `purse_count++`

**Status:** **Acknowledged**

- **Issue: unction visibility can be changed to external ( Context: purse.sol L218, L257, L290, L372, L381, L474, L531)**

- The functions on the aforementioned lines can have their visibility
- changed from **public** to **external** as they are never called from within the contract.

**Recommendation:**

We recommend to correct the word spellings on the aforementioned line.

**Status:** **Acknowledged**





- **Issue: Comparison with Boolean literal ( Context: `purse.sol` L224, L268, L292, L305, L482)**
  - The aforementioned lines perform comparison with Boolean literal. Boolean constant can be used directly and do not need to be compared to true or false.

**Recommendation:**

We advise to directly use the Boolean expression on the aforementioned lines instead of performing comparison with Boolean literal.

**Status:** **Acknowledged**



## CLOSING SUMMARY

There were discoveries of some high, medium, low and informational issues after the audit. The audit team thereafter suggested some remediation to help remedy the issues found in the contract.

## APPENDIX

- *Audit: The review and testing of contracts to find bugs.*
- *Events: In smart contracts, several pieces of information are important not to be missed. For this reason, an event is a solidity feature that helps us track some parameters when they are emitted.*
- *Issues: These are possible bugs that could lead exploits or help redefine the contracts better.*
- *Slither: A tool used to automatically find bugs in a contract.*
- *Severity: This explains the status of a bug.*

## DISCLAIMER

While the audit report is aimed at achieving a quality codebase with assured security and correctness, it should not be interpreted as a guide or recommendation for people to invest in **ChainedThrift** contracts.

With smart contract audit being a multifaceted process, we admonish the **ChainedThrift** team to carry out further audit from other audit firms or provide a bug bounty program to ensure that more critical audit is done to the contract.

## GUILD AUDITS

Guild Audits is geared towards providing blockchain and smart contract security in the fuming web3 world. The firm is passionate about remedying the constant hacks and exploits that deters the web3 motive.

