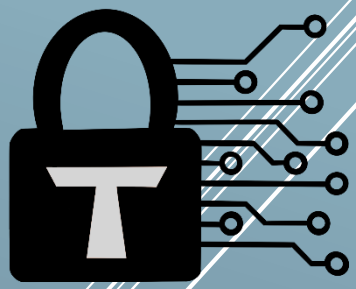
# Trust Security

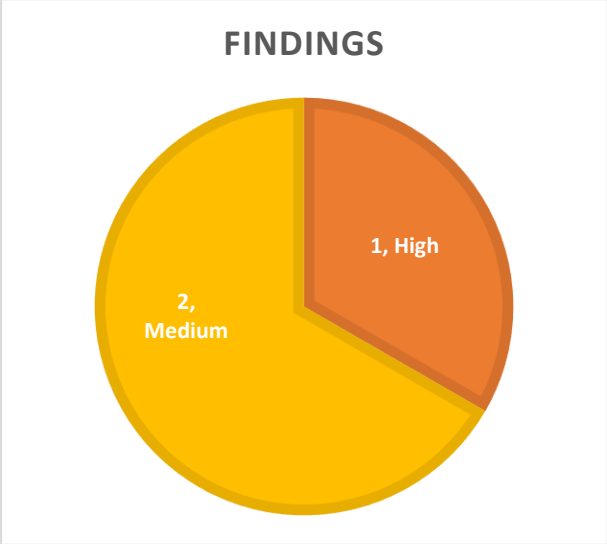


Smart Contract Audit

dHEDGE Buyback Protocol

05/09/2024

# Executive summary

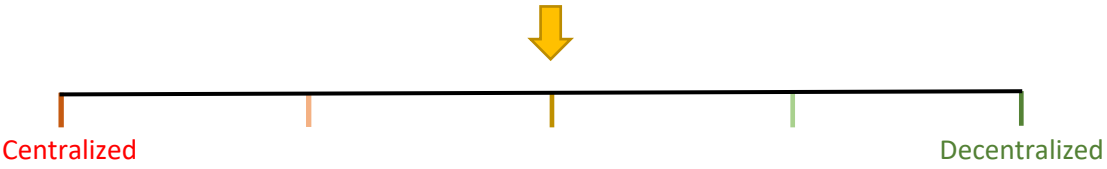



| Category | Launch Platforms |
|---|---|
| Audited file count | 6 |
| Lines of Code | 454 |
| Auditor | MiloTruck SpicyMeatball |
| Time period | 16-26/08/24 |

Findings

| Severity | Total | Open | Fixed | Acknowledged |
|---|---|---|---|---|
| High | 1 | 0 | 1 | 0 |
| Medium | 2 | 0 | 2 | 0 |
| Low | 0 | 0 | 0 | 0 |

Centralization score




Signature

Digitally signed by Trust
Date: 2024.09.05 10:28:25 +03'00'

EXECUTIVE SUMMARY 1

DOCUMENT PROPERTIES 3

**Versioning** **3**

**Contact** **3**

INTRODUCTION 4

**Scope** **4**

**Repository details** **4**

**About Trust Security** **4**

**About the Auditors** **5**

**Disclaimer** **5**

**Methodology** **5**

QUALITATIVE ANALYSIS 6

FINDINGS 7

**High severity findings** **7**

TRST-H-1: Out-of-order execution changes the tokens received on L2 7

**Medium severity findings** **11**

TRST-M-1: Incorrect **lastTokenToBuyPrice** update in *L2ComptrollerV2Base._redeem()* 11

TRST-M-2: *L2ComptrollerV2Base.claim()* cannot be called by contracts on L2 11

**Centralization risks** **15**

TRST-CR-1: Owner risks 15

# Document properties

## Versioning

| Version | Date | Description |
|---|---|---|
| 0.1 | 26/08/24 | Client report |
| 0.2 | 31/08/24 | Mitigation review |
| 0.3 | 05/09/24 | Final Report |

## Contact

**Trust**

trust@trust-security.xyz

# Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

## Scope

Changes to the following files in PR #21:

- src/abstracts/L1ComptrollerV2Base.sol
- src/abstracts/L2ComptrollerV2Base.sol
- src/arb-stack/L1ComptrollerArb.sol
- src/arb-stack/L2ComptrollerArb.sol
- src/op-stack/v2/L1ComptrollerOPV2.sol
- src/op-stack/v2/L2ComptrollerOPV2.sol

## Repository details

- **Repository URL:** https://github.com/dhedge/buyback-contract
- **Commit hash:** 9fed663d5697ffe8c755818a030625473d9571cb
- **Mitigation review commit hash:** 712009e1643fb9f265edb2f2c8b94295961b5edd
- **Final Commit hash:** 532b3d43a98d982fd56b7f9749a812af5e3e7770

## About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

## About the Auditors

MiloTruck is a blockchain security researcher who specializes in smart contract security. Since March 2022, he has competed in over 25 auditing contests on Code4rena and won several of them against the best auditors in the field. He has also found multiple critical bugs in live protocols on Immunefi and is an active judge on Code4rena.

SpicyMeatball is a member of the Code4rena Pro League and has reported over 100 bugs in various DeFi protocols.

## Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

## Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

# Qualitative analysis

| Metric | Rating | Comments |
|---|---|---|
| Code complexity | Good | Project kept code as simple as possible, reducing attack risks. |
| Documentation | Good | Project is adequately documented. |
| Best practices | Excellent | Project consistently adheres to best practices. |
| Centralization risks | Moderate | Project has some centralization risks. |

# Findings

## High severity findings

### TRST-H-1: Out-of-order execution changes the tokens received on L2

- **Category:** Logical flaws
- **Source:** L2ComptrollerV2Base.sol
- **Status:** Fixed

**Description**

In *L2ComptrollerV2Base.redeemFromL1()*, **burnTokenAmount** represents the remaining amount of tokens that a user can claim, based on how much tokens was burnt on L1:

```
// The difference of both these variables tell us the claimable token amount in
`tokenToBurn` denomination.
uint256 burnTokenAmount = totalAmountBurntOnL1 - totalAmountClaimed;
```

**burnTokenAmount** is passed to *_redeem()*, which means the function attempts to claim the entire remaining amount:

```
// The reason we are using try-catch block is that we want to store the
`totalAmountBurntOnL1`
// regardless of the failure of the `_redeem` function. This allows for the depositor
// to claim their share on L2 later.
try this._redeem(tokenBurned, IPoolLogic(tokenToBuy), burnTokenAmount, receiver)
returns (
    uint256 buyTokenAmount
) {
```

However, if a user makes multiple calls to *L1ComptrollerV1Base.redeem()* with a different **tokenToBuy, burnTokenAmount** or **receiver**, this could cause the tokens received on L2 to be different from what the user intended.

Assume a user calls *L1ComptrollerV1Base.redeem()* twice with the same **tokenToBurn**:

1. User calls *L2ComptrollerV2Base.redeemFromL1()* to send 1000 BY1 tokens to Alice:
    a. **tokenToBuy** as BY1, **burnTokenAmount** as 1000, **receiver** as Alice
2. User calls *L2ComptrollerV2Base.redeemFromL1()* to send 1000 BY2 tokens to Bob:
    a. **tokenToBuy** as BY2, **burnTokenAmount** as 1000, **receiver** as Bob

If both L1 to L2 transactions are received in order on L2:

- In the first call to *L2ComptrollerV2Base.redeemFromL1():*
    - **totalAmountBurntOnL1 = 1000**
    - **burnTokenAmount = 1000 – 0 = 1000**
    - Alice receives 1000 BY1
- In the second call to *L2ComptrollerV2Base.redeemFromL1():*

    - **totalAmountBurntOnL1 = 2000**
    - **burnTokenAmount = 2000 – 1000 = 1000**
    - Bob receives 1000 BY2
- As expected, Alice receives 1000 BY1 and Bob receives 1000 BY2

However, assume the first L1 to L2 transaction fails and is not executed. When the second transaction is executed:

- **totalAmountBurntOnL1 = 2000**
- **burnTokenAmount = 2000 – 0 = 0**
- Bob receives 2000 BY2

Unexpectedly, Alice received nothing and Bob receives twice the amount of BY2 than intended.

Therefore, when users make multiple calls to *L2ComptrollerV2Base.redeemFromL1()*, there is no guarantee that they will receive the specified amount of tokens on L2.

Note that the same outcome occurs if the first L1 to L2 transaction was executed, but *_redeem()* reverted.

**Recommended mitigation**

Consider passing **burnTokenAmount** from L1 to L2 and redeeming it on L2, instead of the total remaining amount to be claimed.

Add an **amountBurntOnL1** parameter to *L2ComptrollerV2Base.redeemFromL1()*:

```
  function redeemFromL1(
      address tokenBurned,
      address tokenToBuy,
+     uint256 amountBurntOnL1
      uint256 totalAmountBurntOnL1,
      address l1Depositor,
      address receiver
  ) external whenNotPaused {
```

When calling *_redeem()*, pass **amountBurntOnL1** instead of **burnTokenAmount**:

```
- try this._redeem(tokenBurned, IPoolLogic(tokenToBuy), burnTokenAmount, receiver)
returns (
+ try this._redeem(tokenBurned, IPoolLogic(tokenToBuy), amountBurntOnL1, receiver)
returns (
      uint256 buyTokenAmount
  ) {
```

In *L1ComptrollerV1Base.redeem()*, pass **burnTokenAmount** as the amount of tokens burnt for this call:

```
  // Send a cross chain message to `l2Comptroller` for releasing the buy tokens.
  _sendMessage(
      abi.encodeCall(
          L2ComptrollerV2Base.redeemFromL1,
-         (tokenToBurn, tokenToBuy, totalBurntAmount, msg.sender, receiver)
+         (tokenToBuy, tokenToBuy, burnTokenAmount, totalBurntAmount, msg.sender,
receiver)
      ),
      additionalData
  );
```

Note that with this change, if *_redeem()* or an L1 to L2 transaction fails, users must call *L2ComptrollerV2Base.claim()* to receive their funds.

**Team response**

Fixed in PR #28.

The purpose for providing a **receiver** field in the redeem function in the **L1ComptrollerV2** contract was simply to allow smart contract wallet users or similar as the same address may not be available on the destination chain. While not clearly written in the docs or in the contract comments, it's assumed that the **receiver** address is an address in control of the token burner (the caller of the redeem function).

An easier approach is to remove **l1depositor** dependency altogether. Given that the **L2Comptroller** doesn't really care if the tokens have been burnt by the **receiver** or the **l1depositor** as long as the amount passed by the **L1Comptroller** is correct, this approach works. This means instead of storing the burnt amount of the **l1depositor** (which actually burnt the tokens), we store the amount for the receiver.

**Mitigation review**

We did consider this when thinking of potential mitigations, but with this approach, anyone can call *L1ComptrollerV2Base.redeem()* to redeem tokens on another's behalf.

For example:

- User calls *L1ComptrollerV1Base.redeem()* with **tokenToBuy** as BY1, **receiver** as Alice.
- Assume the L1 -> L2 transaction fails, or is delayed.
- Attacker calls *L1ComptrollerV1Base.redeem()* with **tokenToBuy** as BY2, **receiver** as Alice, and **burnTokenAmount** as 0.
- This causes Alice to receive BY2 instead of BY1.

I would recommend changing both **L1ComptrollerV2Base** and **L2ComptrollerV2Base** to store a mapping of **depositor => receiver => tokenToBurn => amount**.

This ensures that a user can only trigger *L2ComptrollerV2Base.redeemFromL1()* for a receiver with tokens that he had burned. *claim()* can still called by the receiver address, but he has to pass the **l1Depositor** address.

**Team response**

Amended in PR #29.

**Mitigation review**

Verified, the recommended fix was implemented.

## Medium severity findings

### TRST-M-1: Incorrect lastTokenToBuyPrice update in *L2ComptrollerV2Base._redeem()*

- **Category:** Logical flaws
- **Source:** L2ComptrollerV2Base.sol
- **Status:** Fixed

**Description**

In *L2ComptrollerV2Base._redeem()*, when the new token price is larger than the previous token price, **lastTokenToBuyPrice** is updated as such:

```
// Updating the buy token price for future checks.
if (lastTokenToBuyPrice < tokenToBuyPrice) {
    lastTokenToBuyPrice = tokenToBuyPrice;

    emit BuyTokenPriceUpdated(tokenToBuy, tokenToBuyPrice);
}
```

However, since **lastTokenToBuyPrice** is a local variable, this statement does not actually update the contract's state.

As a result, the price check does not work as the token's price will never increase.

**Recommended mitigation**

```
  // Updating the buy token price for future checks.
  if (lastTokenToBuyPrice < tokenToBuyPrice) {
-     lastTokenToBuyPrice = tokenToBuyPrice;
+     buyTokenDetails[tokenToBuy].lastTokenToBuyPrice = tokenToBuyPrice;

      emit BuyTokenPriceUpdated(tokenToBuy, tokenToBuyPrice);
  }
```

**Team response**

Fixed in PR #27.

**Mitigation review**

Verified, the recommended fix was implemented.

### TRST-M-2: *L2ComptrollerV2Base.claim()* cannot be called by contracts on L2

- **Category:** Logical flaws
- **Source:** L2ComptrollerV2Base.sol
- **Status:** Fixed

**Description**

When *L2ComptrollerV2Base.redeemFromL1()* is called, the **l1Depositor** parameter contains the address that called *L1ComptrollerV2Base.redeem()*. This address is used as the **depositor** in the **burnAndClaimDetails** mapping:

```
// Store the new total amount of tokens burnt on L1 and claimed against on L2.
burnAndClaimDetails[l1Depositor][tokenBurned].totalAmountBurned =
totalAmountBurntOnL1;
```

If *_redeem()* or a previous L1 to L2 transaction failed, users can claim their tokens by calling *L2ComptrollerV2Base.claim()*.

*claim()* takes **msg.sender** as the **depositor** address:

```
// `totalAmountClaimed` is of the `tokenToBurn` denomination.
uint256 totalAmountClaimed =
burnAndClaimDetails[msg.sender][tokenBurned].totalAmountClaimed;
uint256 totalAmountBurntOnL1 =
burnAndClaimDetails[msg.sender][tokenBurned].totalAmountBurned;
```

When EOAs call *L1ComptrollerV2Base.redeem()*, there is no issue as they have the same address on L1 and L2.

However, if a contract on L1 calls *L1ComptrollerV2Base.redeem()* and *_redeem()* fails, it is impossible for them to call *claim()* on L2. This is because contracts usually do not have the same address across different chains.

As a result, contracts that bridge tokens using the protocol will be unable to directly claim their funds on L2.

**Recommended mitigation**

If contracts are not meant to bridge tokens using the protocol, consider reverting in *L1ComptrollerV2Base.redeem()* if the caller is a contract.

Otherwise, consider allowing users to specify an address to call *claim()* on their behalf on L2.

In *L1ComptrollerV2Base.redeem()*, add a **claimer** parameter that is sent to L2:

```
    function redeem(
        address tokenToBurn,
        address tokenToBuy,
        uint256 burnTokenAmount,
        address receiver,
+       address claimer,
        bytes memory additionalData
    ) public payable whenNotPaused whenL2ComptrollerSet {
        // ...

        // Send a cross chain message to `l2Comptroller` for releasing the buy tokens.
        _sendMessage(
            abi.encodeCall(
                L2ComptrollerV2Base.redeemFromL1,
-               (tokenToBurn, tokenToBuy, totalBurntAmount, msg.sender, receiver)
+               (tokenToBurn, tokenToBuy, totalBurntAmount, msg.sender, receiver,
claimer)
            ),
            additionalData
        );
```

When *L2ComptrollerV2Base.redeemFromL1()* is called, store the **claimer** address in the **burnAndClaimDetails** mapping:

```
   struct BurnAndClaimDetails {
      uint256 totalAmountBurned;
      uint256 totalAmountClaimed;
+     address claimer;
  }
```

```
  // Store the new total amount of tokens burnt on L1 and claimed against on L2.
  burnAndClaimDetails[l1Depositor][tokenBurned].totalAmountBurned =
totalAmountBurntOnL1;
+ burnAndClaimDetails[l1Depositor][tokenBurned].claimer = claimer;
```

In *claim()*, allow the caller to be either the **l1Depositor** or **claimer** address:

```
  function claim(
      address tokenBurned,
      IPoolLogic tokenToBuy,
      uint256 burnTokenAmount,
+     address l1Depositor,
      address receiver
  ) public whenNotPaused {
+     address claimer = burnAndClaimDetails[l1Depositor][tokenBurned].claimer;
+     if (msg.sender != l1Depositor && msg.sender != claimer) {
+         revert NotL1DepositorOrClaimer();
+     }
```

**Team response**

Fixed in PR #29.

**Mitigation review**

Verified, this issue has been fixed as the **receiver** address can now call *claim()*.

## Centralization risks

### TRST-CR-1: Owner risks

Due to the existence of an owner, the protocol should be considered fully centralized. The owner can cause a user to lose funds in numerous ways, for example, he can:

- Misconfigure the price of burnt tokens on L2.
- Not transfer tokens to the **L2Comptroller** contract, causing it to have insufficient liquidity.
- Prevent users from receiving funds on L2 by:
    - Removing all tokens that can be bought from **buyTokenDetails**.
    - Changing the **l1Comptroller** address.
    - Changing chain-specific configuration values, such as **crossDomainMessenger** for Optimism.

If the owner address is compromised and becomes malicious, it should be assumed that the protocol can be exploited.