SHERLOCK

Security Review For Allbridge

Public Audit Contest Prepared For:	Allbridge
Lead Security Expert:	EgisSecurity
Date Audited:	July 25 - July 28, 2025
Final Commit:	2b70f36

Introduction

Allbridge Core is a cross-chain bridge specialized on stable coin value transfer. Such transfers are enabled by token pools, where liquidity providers can earn rewards for their deposits. And a new feature, called Allbridge Core Yield, makes the tokens locked in Allbridge Core pools liquid, represented as a yield bearing 100% backed stable coin. This feature starts with Celo blockchain and will expand to other blockchains later.

Scope

Repository: allbridge-public/core-auto-evm-contracts

Audited Commit: d79882a8a7f2793cb3f7fcb21a9b317a7639846a

Final Commit: 2b70f36dbfd6151cdc039791c4e539ff2d585f09

Files:

- contracts/lib/MultiUint.sol
- contracts/lib/PoolUtils.sol
- contracts/MultiToken.sol
- contracts/PortfolioToken.sol
- contracts/VirtualMultiToken.sol

Final Commit Hash

2b70f36dbfd6151cdc039791c4e539ff2d585f09

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
0	2

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0xDemon	LeFy	blockace
0xEkko	Mishkat6451	iam0ti
0xFlare	MysteryAuditor	ivanalexandur
0xloophole	Olugbenga-ayo	magiccentaur
AshishLac	Phaethon	omeiza
BobbyAudit	WillyCode20	s4bot3ur
EgisSecurity	X0sauce	v10g1
Emine	alicrali33	veerendravamshi
HeckerTrieuTien	anonymousjoe	xiaoming90
Hurricane	araj	zxriptor

Issue M-1: Attacker can steal 100% of users deposits

Source:
https://github.com/sherlock-audit/2025-07-allbridge-core-yield-judging/issues/173

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0xEkko, 0xFlare, AshishLac, BobbyAudit, EgisSecurity, HeckerTrieuTien, LeFy, Olugbenga-ayo, Phaethon, X0sauce, alicrali33, anonymousjoe, araj, blockace, iam0ti, ivanalexandur, magiccentaur, s4bot3ur, v10g1, veerendravamshi, xiaoming90, zxriptor

Summary

I want to specify this can happen with any tokens I have just given the example with 1e6.

If an attacker donates a very large amount in the contract, then on the first deposit the contract calls _subDepositRewardsPoolCheck, which will deposit the amount that the address holds of the token into the pool, making totalVirtual inflated.

However the first depositor can also be the attacker, the logic of the contract is that if realTotal is 0 we mint 1:1, if the attacker calls deposit with 1000 wei(because we need to get around the amountSP calculation in pool.deposit which will yield a 1 wei deposit out of the 1000 wei), then the attacker can set realTotal to 1 wei

How the attack will happen:

1.Attacker calls deposit 1st, he sets realTotal = 1 wei, then he sees user1 want to donates 10e6, he front-runs him and transfers 10e6) to the contract, the contract assumes this is a reward and deposits it into the pool, which will make virtualTotal:10e3+ 1, and realTotal:1 wei

2. User2 deposits, he gets to the part where erc20 has to be minted to him:

```
out := div(mul(virtualAmount, realTotal), totalVirtual)
```

Now virtualAmount is 10e3, realTotal is 1 wei, and totalVirtual is 10e3 + 1, which makes (10e3 * 1) / 10e3 = 0

What happens if `out` = zero in `_mintAfterTotalChanged`: `if (realAmount == 0) { return; }`

We return to the deposit but we do NOT revert

Now attacker calls withdraw the formula inside burn is: `realTotal * (amount / totalVirtual)`, so realTotal is 1, amount = 20e6 , totalVirtual is 20e6: (1 * (20e6/20e6) = 1, the attacker withdraws his initial donation + the user transfer.

The attacker can donate on every transfer users make so their deposits mint 0 erc20 and with his 1 erc20 he can steal all of the users deposits

Root Cause

Inside _mintAfterTotalChanged the contract should revert if `realAmount == 0`

Internal Pre-conditions

Attacker needs to be first to call deposit with 1000 wei

External Pre-conditions

-

Attack Path

Described

Impact

Attacker can steal 100% of users deposits, Impact: HIGH

PoC

-

Mitigation

Revert inside `_mintAfterTotalChanged` instead of returning:

```
uint256 realAmount = _fromVirtualAfterTotalChangedForMint(virtualAmount, index);
      if (realAmount == 0) {
          revert;
      }
      return MultiToken._mint(account, realAmount, index);
```

Add a minAmountIn deposit

Implement a new reward system which tracks how much rewards did the contract actually receive from the pool

If you are worried about dust amounts implement a onlyOwner function which sweeps any amounts from contract into the pool, that way the owner can foresee if direct transfers will hurt the system in any way

Issue M-2: No slippage control

Source:
https://github.com/sherlock-audit/2025-07-allbridge-core-yield-judging/issues/241

Found by

0xDemon, 0xloophole, BobbyAudit, EgisSecurity, Emine, Hurricane, Mishkat6451, MysteryAuditor, WillyCode20, X0sauce, alicrali33, omeiza, veerendravamshi, xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

It was observed that the `PortfolioToken.deposit()` function does not have slippage controls.

https://github.com/sherlock-audit/2025-07-allbridge-core-yield/blob/main/core-auto-evm-contracts/contracts/PortfolioToken.sol#L33

```
File: PortfolioToken.sol
28:     /**
29:      * @dev Deposit tokens into the pool.
30:      * @param amount The amount of tokens to deposit.
31:      * @param index The index of the pool to deposit to.
32:      */
33:     function deposit(uint amount, uint index) external {
```

As a result, during deposits, users will be subjected to slippage, leading to them receiving fewer portfolio tokens/real amount than expected.

The pool of the AllBridge protocol uses a formula similar to Curve's StableSwap.

The following Python script is directly translated from the PoolUtils.getD() function. The script will compute the number of pool shares minted based on the current reserve (X and Y, USDT and vUSD) of the pool.

```python
import math

def compute_D(x, y, a):
    xy = x * y
    p1 = a * xy * (x + y)
    p2 = (xy * ((a << 2) - 1)) // 3
    p3 = math.sqrt(p1 * p1 + p2 * p2 * p2)

    if p3 > p1:
        d_ = math.pow(p3 - p1, 1/3)
        d_ = math.pow(p1 + p3, 1/3) - d_
    else:
        d_ = math.pow(p1 + p3, 1/3) + math.pow(p1 - p3, 1/3)

    return d_ * 2

def simulate_deposit(x, y, a, deposit):
    old_D = compute_D(x, y, a)
    total = x + y
    add_x = deposit * x // total
    add_y = deposit * y // total
    new_D = compute_D(x + add_x, y + add_y, a)
    lp_minted = new_D - old_D
    return lp_minted, old_D, new_D


# Test parameters
a = 10
deposit = 100

# Scenario 1: Balanced
x1, y1 = 1000, 1000
lp_minted_1, D0_1, D1_1 = simulate_deposit(x1, y1, a, deposit)

# Scenario 2: Imbalanced
x2, y2 = 1500, 500
lp_minted_2, D0_2, D1_2 = simulate_deposit(x2, y2, a, deposit)

# Scenario 3: More Imbalanced
x3, y3 = 1800, 200
lp_minted_3, D0_3, D1_3 = simulate_deposit(x3, y3, a, deposit)

print("Scenario 1 (Balanced):")
print(f"  LP minted: {lp_minted_1}")
print(f"  D before: {D0_1}")
print(f"  D after: {D1_1}\n")

print("Scenario 2 (Imbalanced):")
print(f"  LP minted: {lp_minted_2}")
print(f"  D before: {D0_2}")
print(f"  D after: {D1_2}\n")

print("Scenario 3 (More Imbalanced):")
print(f"  LP minted: {lp_minted_3}")
print(f"  D before: {D0_3}")
print(f"  D after: {D1_3}")
```

Note that for StableSwap formula or implementation:

- If the pool is balanced, depositing 100 USDT will mint around 100 pool shares (e.g., 99.99999999999818)
- However, when the pool is imbalanced (as shown in the 3rd scenario), depositing 100 USDT will only mint 96 pool shares.

```
Scenario 1 (Balanced):
  LP minted: 99.99999999999818
  D before: 2000.0
  D after: 2099.999999999998

Scenario 2 (Imbalanced):
  LP minted: 99.22424784315808
  D before: 1984.4849568631507
  D after: 2083.709204706309

Scenario 3 (More Imbalanced):
  LP minted: 96.21928342445199
  D before: 1924.3856684890352
  D after: 2020.6049519134872
```

This proves that the amount of pool shares minted to the `PortfolioToken` contract is dependent on the current state/reserve of the pool when the transaction is executed, and is subject to slippage.

Assume that the share price is 1:1 in the `PortfolioToken` contract. One (1) pool share/virtual token will mint one (1) portfolio token/real token.

Alice deposits 100 USDT to the `PortfolioToken` contract, and expects that 99.99999 portfolio tokens/real tokens will be minted to her. However, due to the slippage, she only received 96 portfolio tokens/real tokens in return, resulting in a loss to Alice. If Alice redeems his 96 portfolio, she will receive only around 96 USDT back.

Note that the pool is not always balanced. The pool can become imbalanced for many reasons. For instance, if there is a large cross-chain transfer that swaps a large number of vUSD to USDT OR swaps a large number of USDT to vUSD, the pool will become imbalanced.

Note that it is expected that the imbalance pool will be arbitrage back to a balanced pool. However, an important point is that arbitrage does not occur immediately, as it takes time for the bots to notice the imbalance. If Alice's deposit transaction is executed immediately after a large swap/cross-chain transfer, she will suffer significant slippage due to imbalances in the pool.

Impact

Loss of funds for the victims, as demonstrated in the report.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/allbridge-public/core-auto-evm-contracts/pull/1

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.