Algem

Lending Adapter

Smart Contract Security Assessment

March 07, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of the Algem Lending Adapter protocol.

Algem protocol is a DeFi dApp built on top of Astar Network. Two of the main features they offer are Liquid Staking and Liquid Lending. The former protocol has been audited by Dedaub in the past and the corresponding reports can be found here and here. This time this audit focuses on the contracts related to the Liquid Lending feature.

The Lending Adapter contracts being audited provide an interface for the nASTR token holders to interact with the Sio2 lending protocol through Algem on the Astar network. The only allowed collateral token is the nASTR token which can only be issued by Algem's Liquid Staking protocol. Substantial logic has been added by Algem in order to keep the records of the users in sync with the records of the Sio2 lending protocol. This allows the depositors and the borrowers to earn rewards and to lend with variable or stable interest rates as well.

An effort was made to verify the workflows and interaction between these protocols. However, due to the complex business logic carried out by the Sio2 protocol, along with the synchronisation logic in Algem's adapter, the development of an extensive test suite with high code coverage is highly recommended. This includes testing for scenarios involving multiple users stressing workflows in a sequence, performing multiple deposits, lend, borrow and repay operations so as to ensure the correctness of the adapter's records with respect to the corresponding Sio2 records and the interactions which were performed.

This audit report covers the contracts of the azhlbn/LendingAdapter of the Algem Lending Adapter protocol at commit `5f23dbadbbea078bed9a03b439cdc166c4fa1931`. As part of the audit we also reviewed the fixes of the issues included in the report up to commit `a7b6284ebc1a28f6e92c948b7d66d4a9f4b03bbf`.

Two auditors audited the code base for 3 days.

The audited contracts are the following:

```
src/
├─ Sio2Adapter.sol
├─ Sio2AdapterAssetManager.sol
│
├─ interfaces/
│  ├─ ISio2IncentivesController.sol
│  ├─ ISio2LendingPool.sol
│  └─ ISio2PriceOracle.sol
│
└─ libraries/
   └─ ReserveConfiguration.sol
```

SETTING & CAVEATS

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behaviour. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	addBTokens() of Sio2AdapterAssetManager is public and can be used to DoS the system	**RESOLVED**

The Sio2AdapterAssetManager contract has a public addBTokens() function which can be used by an adversary to completely DoS the protocol, since an attacker can add any number of tokens to the bTokens array.

Sio2AdapterAssetManager::addBTokens()

```
function addBTokens(address _bToken) public {
    // Dedaub: Unrestricted and publicly accessible function that can be
    //         used to DoS the protocol
    bTokens.push(_bToken);
    emit AddBToken(msg.sender, _bToken);
}
```

This directly affects the Sio2Adapter::_harvestRewards() function which fetches the bTokens array and forwards it to Sio2IncentiveController::claimRewards() function.

Sio2Adapter::_harvestRewards()

```
function _harvestRewards(uint256 _pendingRewards) private {
    // Dedaub: Fetching the bTokens array
    address[] memory bTokens = assetManager.getBTokens();
    string[] memory assets = assetManager.getAssetsNames();

    // receiving rewards from incentives controller
    // this rewards consists of collateral and debt rewards
```

```
    // Dedaub: The fetched bTokens array is given to the claimRewards
    uint256 receivedRewards = incentivesController.claimRewards(
        bTokens,
        _pendingRewards,
        address(this)
    );
    ...
}
```

The `Sio2IncentiveController::claimRewards()` function iterates over the array to harvest the rewards.

`Sio2IncentiveController::claimRewards()`

```
function claimRewards(
    address[] calldata assets,
    uint256 amount,
    address to
) external override returns (uint256) {
    if (amount == 0) return 0;
    ...

    DistributionTypes.UserStakeInput[] memory userState =
        new DistributionTypes.UserStakeInput[](assets.length);

    // Deduab: Iteration over all bTokens elements
    for (uint256 i = 0; i < assets.length; i++) {
        userState[i].underlyingAsset = assets[i];
        (userState[i].stakedByUser, userState[i].totalStaked) =
            ISToken(assets[i]).getScaledUserBalanceAndSupply(user);
    }
    ...
}
```

An attacker can add enough tokens to the `bTokens` array to make the `_harvestRewards()` call run out of gas. Note that `_harvestRewards()` is called in the `update` modifier by calling the `_updates()` function. As this modifier is used by all the main functions of the `Sio2Adapter`, the whole protocol can be effectively blocked.

H2	Unbounded loops can be exploited to prevent liquidation	**RESOLVED**

The `User.borrowedAssets` field is used by a number of functions, which iterate over all borrowed assets for a specific account.

`Sio2Adapter::calcEstimateUserDebtUSD()`

```
function calcEstimateUserDebtUSD(
    address _userAddr
) public view returns (uint256 debtUSD) {
    User memory user = userInfo[_userAddr];
    for (uint256 i; i < user.borrowedAssets.length;) {
        ...
    }
}
```

Under adversarial use such loops can be exploited. Imagine a malicious user with a large underwater position risking being liquidated. Assuming that a large number of assets that can be borrowed are available in the system, this user can completely prevent being liquidated as follows:

- He can borrow small amounts of each of the available to-be-borrowed assets, increasing in that way the size of his `borrowedAssets` array.
- Once the `borrowedAssets` is large enough, the `liquidateCall()` will fail for any liquidator trying to liquidate the malicious user's underwater position. This happens because the `liquidateCall()` calls the `calcEstimateUserDebtUSD()` function twice, and this iterates over all of the

user's borrowed assets. This function is called directly in the code of `liquidateCall()` and also indirectly through a call to `getHF()`. Thus, this results in double iteration over an already large array.

Sio2Adapter::liquidationCall()

```
function liquidationCall(
    string memory _debtAsset,
    address _user,
    uint256 _debtToCover
) external {
    ( , , address debtAssetAddr, , , , , , , ) =
        assetManager.assetInfo(_debtAsset);
    address liquidator = msg.sender;

    // check user HF and update state
    require(getHF(_user) < 1e18,
        "User has healthy enough position");

    // get total user debt in usd and a specific asset
    uint256 userTotalDebtInUSD = calcEstimateUserDebtUSD(_user);
    ...
}
```

Sio2Adapter::getHF()

```
function getHF(
    address _user
) public update(_user) returns (uint256 hf) {
    uint256 debtUSD = calcEstimateUserDebtUSD(_user);
    require(debtUSD > 0, "User has no debts");
    ...
}
```

- The user is protected from the DoS since the `repay*()` functions perform only 1 iteration over the `borrowedAssets` array through the `update` modifier which calls the `_updateUserRewards()` (through `_updates()`) which iterates over the array.

Sio2Adapter::repayFull()

```
function repayFull(
    string memory _assetName
) external update(msg.sender) nonReentrant {
    ...
    _repay(_assetName, fullDebtAmount, msg.sender);
}
```

Sio2Adapter::update

```
modifier update(address _user) {
    _updates(_user);
    _;
    _updateLastSTokenBalance();
}
```

Sio2Adapter::_updates()

```
function _updates(address _user) private {
    ...
    if (block.number > lastUpdatedBlock) {
        _updatePools();
        _updateUserRewards(_user);

        lastUpdatedBlock = block.number;
    }

    emit Updates(msg.sender, _user);
}
```

Sio2Adapter::_updateUserRewards()

```
function _updateUserRewards(address _user) private {
    User storage user = userInfo[_user];
    for (uint256 i; i < user.borrowedAssets.length; ) {
        ...
    }
}
```

- In the future the malicious user can repay his debt reducing his debt position.

Note that even if the `liquidationCall()` was iterating only once over the `borrowedAssets` array, it would still be susceptible to the attack since it contains more logic around the call to `_repay` and thus consumes more gas than the `repay*()` functions.

To prevent such attacks we recommend adding a maximum allowed size of `borrowedAssets` (which can be large enough not to affect the regular use of the protocol).

H3	Withdrawing all the collateral before withdrawing rewards leads to loss of rewards	**RESOLVED**

In the `Sio2Adapter` contract, if a user withdraws all of his collateral using the `withdraw()` function, his `User` struct will be deleted from the `userInfo` mapping through a call to the `_removeUser()` function.

Sio2Adapter::withdraw()

```
function withdraw(uint256 _amount) external update(msg.sender) {
    _withdraw(msg.sender, _amount);
}
```

Sio2Adapter::_withdraw()

```
function _withdraw(address _user, uint256 _amount) private nonReentrant {
    ...
    uint256 withdrawnAmount = pool.withdraw(
        address(nastr),
        _amount,
        address(this)
    );

    _updateLastSTokenBalance();

    User storage user = userInfo[_user];

    totalSupply -= withdrawnAmount;
    user.collateralAmount -= withdrawnAmount;
    _updateUserCollateralIncomeDebts(user);

    // send collateral to user or liquidator
    nastr.safeTransfer(msg.sender, withdrawnAmount);

    // remove user if his collateral becomes equal to zero
    if (userInfo[_user].collateralAmount == 0) _removeUser();

    emit Withdraw(msg.sender, _amount);
}
```

Sio2Adapter::_removeUser()

```
function _removeUser() private {
    uint256 lastId = users.length - 1;
    uint256 userId = userInfo[msg.sender].id;
    userInfo[users[lastId]].id = userId;

    // Dedaub: Here the userInfo record of the user gets deleted
    delete userInfo[users[userId]];
```

```
    users[userId] = users[lastId];
    users.pop();
    emit RemoveUser(msg.sender);
}
```

However, removing this struct also deletes the `rewards` field from the struct. Hence, if the user subsequently tries to claim his rewards by calling the `claimRewards()` function, the protocol will not transfer any rewards, because they have been deleted.

Sio2Adapter::claimRewards()

```
function claimRewards() external update(msg.sender) {
    User storage user = userInfo[msg.sender];
    // Dedaub: Here the user.rewards will be 0 since user's records have
    //         already been deleted upon collateral withdrawal
    require(user.rewards > 0, "User has no any rewards");

    uint256 rewardsToClaim = user.rewards;
    user.rewards = 0;
    rewardPool -= rewardsToClaim;
    rewardToken.safeTransfer(msg.sender, rewardsToClaim);

    emit ClaimRewards(msg.sender, rewardsToClaim);
}
```

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	addBTokens() of Sio2AdapterAssetManager can break the synchronization between the contract's data structures	**RESOLVED**

The Sio2AdapterAssetManager contract has a public addBTokens() function which allows anyone to add tokens to the bTokens array.

Sio2AdapterAssetManager::addBTokens()

```
function addBTokens(address _bToken) public {
    bTokens.push(_bToken);
    emit AddBToken(msg.sender, _bToken);
}
```

Arbitrarily adding tokens in the bTokens array can cause problems with the addAsset() and removeAsset() functions, by messing with the records of the system.

These two functions assume that the assets array is in sync with the bTokens array, with the asset.id field used to index the token inside the bTokens array corresponding to the asset.

Sio2AdapterAssetManager::removeAsset()

```
function removeAsset(string memory _assetName) external onlyOwner {
    require(assetInfo[_assetName].addr != address(0),
        "There is no such asset"
    );
    ...
    // remove addr from bTokens
    address lastBAddr = bTokens[bTokens.length - 1];
```

```
    // Dedaub: Here the bTokens array is accessed based on the asset.id
    //         field which can be out-of-sync with its actual index
    bTokens[asset.id] = lastBAddr;
    bTokens.pop();
    ...
}
```

Consider the following scenario:

- Both the `assets` and `bTokens` arrays are empty.
- An item is added to the `bTokens` array at position `0` through `addBTokens`.
- Then, the owner of the contract calls `addAsset()`. This adds the asset to `assets`, but the id is set to `assets.length`, which is `0`.
- The second `bToken` is then added at position `1`.
- If the owner calls `removeAsset()` on the `asset`, the `bToken` at position `0` is deleted, and the `bToken` at position 1 remains, resulting in incorrect records in the system.
- Furthermore, `getBTokens()` function and any other external user of the `bTokens` array will start reporting incorrect results.

Note that this problem repeats itself any time the `addBToken()` function is used, and can happen at different indices.

L2	`addAsset()` of `Sio2AdapterAssetManager` can lead to repeated tokens in the `bTokens` array	**RESOLVED**

The `Sio2AdapterAssetManager` contract has a function called `addAsset()` which adds a token to the `bTokens` array each time an asset is added to the protocol.

Sio2AdapterAssetManager::addAsset()

```
function addAsset(
    string memory _assetName,
    address _assetAddress,
    address _bToken,
    uint256 _rewardsWeight
) external onlyOwner {
    ...
    assets.push(asset.name);
    assetsAddresses.push(asset.addr);
    assetInfo[_assetName] = asset;
    bTokens.push(_bToken);

    emit AddAsset(msg.sender, _assetName, _assetAddress);
}
```

However, if an already registered bToken gets accidentally reused for a different asset, the function will successfully register it in the bTokens array again without checking whether it has already been inserted in the system or not.

This may cause code to be executed multiple times for the same token. This issue is being flagged here as it may lead to potential problems as the code evolves.

L3	`__ReentrancyGuard_init()` is not called	**RESOLVED**

The function `__ReentrancyGuard_init()` of `ReentrancyGuardUpgradeable.sol` is not called by the `initialize()` function of both the `Sio2Adapter.sol` and the `Sio2AdapterAssetManager.sol` contracts that inherit from it.

This function contains the logic of initializing the reentrancy guards for the contracts and should be called along with any other initializer of the inherited contracts.

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Commented constructor in `Sio2AdapterAssetManager`	**RESOLVED**
The `Sio2AdapterAssetManager` has a constructor which calls the `_disableInitializers()` function for preventing any future initialization of the implementation contract. However, this constructor is commented out.		
A2	Duplicate Asset struct declaration in different contracts	**RESOLVED**
The `Asset` struct is declared twice, once in the `Sio2Adapter` and once in the `Sio2AdapterAssetManager`. This may lead to potential maintainability and security issues if one struct is changed and the other is not by mistake.		
A3	`calcEstimateUserDebtUSD()` is called twice spending unnecessary gas	**RESOLVED**

The `calcEstimateUserDebtUSD()` function of the `Sio2Adapter` contract is called twice when `liquidationCall()` is called. Once directly through the function's code and once indirectly by calling the `getHF()` function, which calls again `calcEstimateUserDebtUSD()`.
 	
`Sio2Adapter::liquidationCall()`
`function liquidationCall(`
`  string memory _debtAsset,`
`  address _user,`		

```
    uint256 _debtToCover
) external {
    ( , , address debtAssetAddr, , , , , , , ) =
        assetManager.assetInfo(_debtAsset);
    address liquidator = msg.sender;

    // check user HF and update state
    require(getHF(_user) < 1e18,
        "User has healthy enough position");

    // get total user debt in usd and a specific asset
    uint256 userTotalDebtInUSD = calcEstimateUserDebtUSD(_user);
    ...
}
```

Sio2Adapter::getHF()

```
function getHF(
    address _user
) public update(_user) returns (uint256 hf) {
    uint256 debtUSD = calcEstimateUserDebtUSD(_user);
    require(debtUSD > 0, "User has no debts");
    ...
}
```

This function performs an iteration over all user's borrowed assets for estimating his debt in USD.

Sio2Adapter::calcEstimateUserDebtUSD()

```
function calcEstimateUserDebtUSD(
    address _userAddr
) public view returns (uint256 debtUSD) {
    User memory user = userInfo[_userAddr];
    for (uint256 i; i < user.borrowedAssets.length;) {
```

```
        ...
    }
}
```

Thus, you could calculate the user's debt once and use this instead of calling the function twice, saving a considerable amount of gas. One solution could be to make `getHF()` return the calculated value for the `liquidationCall()` and use this instead.

A4	No check whether the `revenuePool` has sufficient assets to withdraw	**RESOLVED**

In the `Sio2Adapter` contract, the `withdrawRevenue()` function allows the `owner` to withdraw the accumulated funds from the `revenuePool`. However, there is no check on whether the pool's funds are sufficient for withdrawing the requested amount, causing the function to revert due to underflow when trying to subtract the given amount from the pool.

Sio2Adapter::withdrawRevenue()

```
function withdrawRevenue(uint256 _amount) external onlyOwner {
    require(_amount > 0, "Should be greater than zero");
    require(rewardToken.balanceOf(address(this)) >= _amount,
        "Not enough SIO2 revenue tokens"
    );

    // Dedaub: There is no check whether the revenuePool has enough funds
    //         to cover the requested _amount
    revenuePool -= _amount;
    rewardToken.safeTransfer(msg.sender, _amount);

    emit WithdrawRevenue(msg.sender, _amount);
}
```

A5	Updating variables that are not used anywhere	RESOLVED

In the `Sio2AdapterAssetManager` contract, the `assetAddresses` array gets updated when adding or removing an asset from the system, but it is not used anywhere else in the protocol.

Sio2AdapterAssetManager::addAsset()

```
function addAsset(
    string memory _assetName,
    address _assetAddress,
    address _bToken,
    uint256 _rewardsWeight
) external onlyOwner {
    ...
    assets.push(asset.name);
    assetsAddresses.push(asset.addr);
    assetInfo[_assetName] = asset;
    bTokens.push(_bToken);
    emit AddAsset(msg.sender, _assetName, _assetAddress);
}
```

Sio2AdapterAssetManager::removeAsset()

```
function removeAsset(
    string memory _assetName
) external onlyOwner {
    ...
    // remove from assetsAddresses
    address lastAddr = assetsAddresses[assetsAddresses.length - 1];
    assetsAddresses[asset.id] = lastAddr;
    assetsAddresses.pop();
    ...
}
```

A6	More indicative naming of `getInfo()` function in `Sio2AdapterAssetManager`	**RESOLVED**

The `Sio2AdapterAssetManager` contract has a function called `getInfo()` which returns the `rewardsWeight` of an asset. A more indicative name such as `getRewardsWeight()` could be chosen to reflect the actual purpose of this function.

A7	Possible redundant event is emitted in `Sio2Adapter _updateLastSTokenBalance()` function	**RESOLVED**

The `Sio2Adapter` has a function called `_updateLastSTokenBalance()` which updates `lastSTokenBalance` if the corresponding balance of the `snastrToken` has changed. However, an `UpdateLastSTokenBalance` event is emitted irrespectively of whether the variable was updated.

A8	Unused variables and events	**RESOLVED**

The following variables and events are not used anywhere in the protocol and can be removed:

- **Sio2Adapter.sol**
  - `DOT_ADDR` (*variable*)
  - `SetupCollateralParams` (*event*)

- **Sio2AdapterAssetManager.sol**
  - `COLLATERAL_REWARDS_WEIGHT` (*variable*)

A9	Functions could be made external	**RESOLVED**

The following functions could be made `external` instead of `public`, as they are not called by any of the contract's functions:

- **Sio2Adapter.sol**
  - estimateHF()
  - getUser()

- **Sio2AdapterAssetManager.sol**
  - initialize()
  - setAdapter()
  - getBTokens()
  - getAssetsNames()
  - getInfo()

A10	Compiler bugs	**INFO**

The code can be compiled with Solidity 0.8.4. Version 0.8.4, in particular, has some known bugs, which we do not believe affect the correctness of the contracts.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.