Hourglass “Locking” Security Audit Report

PREPARED FOR:

Charlie Pyle

Hourglass Foundation

ARCADIA CONTACT INFO

Email: audits@arcadiamgroup.com

Telegram: https://t.me/thearcadiagroup

Revision history

Date	Reason	Commit
03/01/2024	Initial Audit Scope	#6269a9abfc7b741a3292e4a964557d5732d38b79
4/06/2024	Review Of Remediations	

Table of Contents

Executive Summary

1. Introduction and Audit Scope

Hourglass Foundation engaged Arcadia to perform a security audit of their locking protocol smart contracts; our review of their codebase occurred in the repo pitch-foundation/locking on the commit hash #55cd83e9b048e2a7f08becc10d8f56194ba3dd7a

a. Review Team

Van Cam Pham - Lead Security Engineer

b. Project Background

Hourglass is a protocol that facilitates liquidity for time-locked and semi-fungible assets.

c. Coverage

For this audit, we performed research, test coverage, investigation, and review of Hourglass's locking contracts, followed by issue reporting and mitigation and remediation instructions as outlined in this report. The following code repositories, files, and/or libraries are considered in scope for the review.

Files
src/HourglassTBTFactory.sol
src/depositors/HourglassLockDepositor.sol
src/recipts/HourglassERC20TBT.sol
src/interfaces/IHourglassDepositor.sol
src/interfaces/IHourglassERC20TBT.sol
src/interfaces/IHourglassLockingTBTFactory.sol
src/utils/TwoStepOwnable.sol
src/utils/TwoStepOwnableInterface.sol

2. Audit Summary

a. Audit Methodology

Arcadia completed this security review using various methods, primarily consisting of dynamic and static analysis. This process included a line-by-line analysis of the in-scope contracts, optimization analysis, analysis of key functionalities and limiters, and reference against intended functionality.

The followings are the steps we have performed while auditing the smart contracts:

- Investigating the project and its technical architecture overview through its documentation
- Understanding the overview of the smart contracts, the functions of the contracts, the inheritance, and how the contracts interface with each other thanks to the graph created by Solidity Visual Developer
- Manual smart contract audit:
    - Review the code to find any issue that could be exploited by known attacks listed by Consensys
    - Identifying which existing projects the smart contracts are built upon and what are the known vulnerabilities and remediations to the existing projects
    - Line-by-line manual review of the code to find any algorithmic and arithmetic related vulnerabilities compared to what should be done based on the project's documentation
    - Find any potential code that could be refactored to save gas
    - Run through the unit-tests and test-coverage if exists
- Static Analysis:

- Scanning for vulnerabilities in the smart contracts using Static Code Analysis Software
- Making a static analysis of the smart contracts using Slither

- Additional review: a follow-up review is done when the smart contracts have any new update. The follow-up is done by reviewing all changes compared to the audited commit revision and its impact to the existing source code and found issues.

b. Summary

There were **8** issues found, **0** of which were deemed to be 'critical', and **2** of which were rated as 'high'.

Severity Rating	Number of Original Occurrences	Number of Remaining Occurrences
CRITICAL	0	0
HIGH	2	0
MEDIUM	1	0
LOW	2	0
INFORMATIONAL	3	0
GAS	0	0

Findings in Manual Audit

1. **_deposit** does not take taxed tokens into account

Issue ID

LOCK-1

Status

Acknowledged

Risk Level

Severity: High

Code Segment

```
function _deposit(address user, uint256 amount, bool receiveSplit)
internal {
       // check that deposit cap has not been reached
       if (currentDeposits + amount > depositCap) revert
DepositCapExceeded();
       // increment the current deposits
       currentDeposits += amount;

       // pull the deposit token in from the caller, not the user
       IERC20(underlying).safeTransferFrom(msg.sender,
address(this), amount);

       if (receiveSplit) {
           // mint both the principal and yield tokens
           IHourglassERC20TBT(principalToken).mint(user, amount);
           IHourglassERC20TBT(yieldToken).mint(user, amount);
       } else {
           // otherwise just mint the combined token, which could be
this contract.
           IHourglassERC20TBT(combinedToken).mint(user, amount);
       }

       emit Deposit(user, amount);
   }
```

```
function redeemPrincipal(uint256 amount) external onlyMatured {
        _redemption(amount, principalToken);
    }

    function _redemption(uint256 amount, address subToken) internal {
        // note deposit cap only relevant while not matured, so no
need to decrease the tracker here
        // ensure that the user has enough funds to redeem
        IHourglassERC20TBT(subToken).burn(msg.sender, amount);
        // transfer funds to user
        IERC20(underlying).safeTransfer(msg.sender, amount);

        emit Redeem(msg.sender, amount);
    }
```

Description

The function allows users to deposit the underlying token into the contract and mint corresponding amounts of the **principalToken** and yieldToken tokens or an amount of the **combinedToken** token.

If the underlying token has transfer tax, the actual received token amount will be less than expected. This poses failures when users redeem the **principalToken** token for the underlying token.

Code location

`src/depositors/HourglassLockDepositor.sol, function _deposit, redeemPrincipal, _redemption`

Recommendation

Either ensure that all underlying tokens are tax-free or check the balance difference before and after the deposit to calculate the exact amount of deposited token in the function.

Proof of concept

- Succeed to deposit
- Failed to redeemPrincipal

```
contract MockTaxedERC20 is ERC20 {
   constructor() ERC20("M", "M") {}

   function mint(uint256 amount) public {
       _mint(msg.sender, amount);
   }

   function _update(address from, address to, uint256 value) internal virtual
override {
       super._update(from, to, value * 95 / 100);
       super._update(from, address(0), value * 5 / 100); // burn
   }
}

function testDepositWithdrawTaxedToken() public {
       uint256 depositAmount = 1e3 ether;
       vm.startPrank(address(this));
       MockTaxedERC20 token = new MockTaxedERC20();
       address taxedDepositToken = address(token);

       bytes memory depositorInitData = abi.encodeWithSignature(
           "initialize(address,uint256,string,string,address)", taxedDepositToken,
maturity, "T", "T", address(tbtImpl)
       );
       address _depositorProxy = factory.createMaturity(0, depositorInitData);
       HourglassLockDepositor depositor = HourglassLockDepositor(_depositorProxy);

       vm.startPrank(alice);
       token.mint(1e10 ether);
       MockTaxedERC20(taxedDepositToken).approve(address(depositor),
type(uint256).max);
       depositor.deposit(depositAmount, true);
       assert(taxedDepositToken == depositor.underlying());
       vm.warp(block.timestamp + maturity);
       vm.expectRevert();
       depositor.redeemPrincipal(depositAmount);
```

```
        vm.stopPrank();
    }
```

Remediation

The code has been modified to more clearly state that tax tokens are not supported; additionally, due to the nature of this deployment, only the Client can currently add new assets. If a tax token would be supported later, it would be through the instantiation of a new and modified depositor.

2. Reentrancy could lead to invalid contract data

Issue ID

LOCK-2

Status

Resolved

Risk Level

Severity: High

Code Segment

```
function _deposit(address user, uint256 amount, bool receiveSplit)
internal {
        // check that deposit cap has not been reached
        if (currentDeposits + amount > depositCap) revert
DepositCapExceeded();
        // increment the current deposits
        currentDeposits += amount;

        // pull the deposit token in from the caller, not the user
        IERC20(underlying).safeTransferFrom(msg.sender,
address(this), amount);
```

```
        if (receiveSplit) {
            // mint both the principal and yield tokens
            IHourglassERC20TBT(principalToken).mint(user, amount);
            IHourglassERC20TBT(yieldToken).mint(user, amount);
        } else {
            // otherwise just mint the combined token, which could be
this contract.
            IHourglassERC20TBT(combinedToken).mint(user, amount);
        }

        emit Deposit(user, amount);
    }
```

Description

The function has no re-entrancy guard. The contract state field **currentDeposits** is increased before an actual **transferFrom** of token from the user to the contract. Bad underlying tokens could trigger to call the **deposit** function within **transferFrom** repeatedly, thus updating the **currentDeposits** multiple times for a single deposit. This could update **currentDeposits** to **depositCap** even though the actual deposit is less, thus preventing others from depositing to the contract more.

The control flow in this case will be:

- A bad user deposits 100 token by calling deposit, in turns calling internal function **_deposit**
- **currentDeposits** is increased 100
- **transferFrom** calls deposit again
- **currentDeposits** is increased 100 again, and so on

Code location

```
src/depositors/HourglassLockDepositor.sol, function _deposit
```

Recommendation

Ensure that all underlying tokens are trusted

Add reentrancy guard to **deposit** and **depositFor** functions

Proof of concept

ERC20 token contract example that calls deposit function repeatedly again to invalidly update the depositor contract data

```
contract MockBadERC20 is ERC20 {
   address private depositor;
   constructor(address _depositor) ERC20("B", "B") {
       depositor = _depositor;
   }

   function mint(uint256 amount) public {
       _mint(msg.sender, amount);
   }

   function _update(address from, address to, uint256 value) internal virtual
override {
       if (from == depositor) {
           IHourglassDepositor(depositor).deposit(value, true);
       } else {
           super._update(from, to, value);
       }
   }
}
```

Remediation

The client has reorganized the flow so that the transaction transfers first.

3. Lack of check for index out of range

Issue ID

LOCK-3

Status

Resolved

Risk Level

Severity: Medium

Code Segment

```
function createMaturity(uint256 index, bytes calldata depositorData)
        external
        onlyRole(DEPLOYER)
        returns (address newDepositor)
    {
        if (!versions[index].isActive) revert InactiveVersion();
        if (depositorData.length == 0) revert CannotBeZero();

        // create the new depositor, which creates it's own tokens
        newDepositor = address(new
BeaconProxy(versions[index].depositorBeacon, ""));
        // register this as a system addres so it can call & create
clones as needed
        isSystemAddress[newDepositor] = true;

        // initialize the depositor
        (bool success,) = newDepositor.call(depositorData);
        if (!success) revert InitializationFailed();

        emit NewMaturityCreated(deployments[index].length,
newDepositor);

        // add it to the deployments
        deployments[index].push(Deployment({depositor:
newDepositor}));

        return (newDepositor);
    }
```

Description

The input index variable is used as the index for reading a data item from the state **versions** array. This could result in an out-of-range error, which is hard to track in production.

Code location

```
src/HourglassTBTFactory.sol
```

Recommendation

Check whether the index variable is out of range for accessing the array and revert with a clear error code.

4. Lack of events for tracking state changes

Issue ID

LOCK-4

Status

Acknowledged

Risk Level

Severity: Low, likelihood: Low

Code Segment

```
function grantReceiptImplementation(address implementation) external
onlyRole(REGISTRY_MANAGER) {
       if (implementation == address(0)) revert CannotBeZero();
       allowableReceiptImplementations[implementation] = true;
   }
```

```
function cloneReceipt(address implementation) external returns
(address deployment) {
       if (!isSystemAddress[msg.sender]) revert
UnauthorizedAddress();
       if (!allowableReceiptImplementations[implementation]) revert
UnauthorizedCloneImplementation();
       // Create a non-upgradeable clone referencing the desired
implementation (the tokens)
       // note that this creates an uninitialized, non-upgradeable
clone
       // the depositor must immediately initialize it with the
relevant values.
       if (implementation.code.length == 0) revert
InvalidImplementation();

       /// @solidity memory-safe-assembly
       assembly {
           // Cleans the upper 96 bits of the `implementation` word,
then packs the first 3 bytes
           // of the `implementation` address with the bytecode
before the address.
           mstore(0x00, or(shr(0xe8, shl(0x60, implementation)),
0x3d602d80600a3d3981f3363d3d373d3d3d363d73000000))
           // Packs the remaining 17 bytes of `implementation` with
the bytecode after the address.
           mstore(0x20, or(shl(0x78, implementation),
0x5af43d82803e903d91602b57fd5bf3))
           deployment := create(0, 0x09, 0x37)
       }

       if (deployment == address(0)) {
           revert ERC1167FailedCreateClone();
       }

       isSystemAddress[deployment] = true;

       return deployment;
   }
```

Description

The function updates the **allowableReceiptImplementations** mapping that controls the set of allowable token implementations.

As it is nontrivial to query all keys in a Solidity mapping, the lack of event emission can be problematic if external parties or users need to be notified of state changes. Events are crucial for transparency and interoperability, allowing external systems to react to changes in the contract state. It's generally advisable to emit events whenever relevant to ensure proper communication of state changes.

Similarly, an event is needed to track for new receipt deployment in function **cloneReceipt**, which updates the state mapping **isSystemAddress**

Code location

```
src/HourglassTBTFactory.sol, lines 73-76, lines 162-186
```

Recommendation

Define an event and emit an instance when there is an update to the mapping. The following is an example of a code recommendation

```
event AllowableReceiptImplementationAdded(address implementation);
   function grantReceiptImplementation(address implementation)
external onlyRole(REGISTRY_MANAGER) {
       if (implementation == address(0)) revert CannotBeZero();
       allowableReceiptImplementations[implementation] = true;
       emit AllowableReceiptImplementationAdded(implementation);
   }
```

Remediation

Hourglass is utilizing the existing events emitted by the depositor. It is a better-positioned location for events to be emitted as it is where the token addresses are tracked. This links it directly to the depositor address

5. Lack of check whether the input version is active or not

Issue ID

LOCK-5

Status

Resolved

Risk Level

Severity: Low

Code Segment

```
function upgradeVersion(uint256 version, address newImplementation)
external onlyRole(DEFAULT_ADMIN_ROLE) {
       if (newImplementation == address(0)) revert CannotBeZero();
       // upgrade the beacon to the new implementation
       address beacon = versions[version].depositorBeacon;
       emit VersionUpgraded(version,
IBeacon(beacon).implementation(), newImplementation);
       UpgradeableBeacon(beacon).upgradeTo(newImplementation);
   }
```

Description

The **upgradeVersion** function upgrades the beacon to the new implementation, however, this function does not check whether the version to upgrade is active or not. Upgrading an inactive version could lead to waste of gas fees or lead to potential upgrade issues.

Code location

```
src/HourglassTBTFactory.sol
```

Recommendation

Add a check whether the input version is active or not

6. Function **cancelOwnershipTransfer**: should revert if there is no potential owner update

Issue ID

LOCK-6

Status

Acknowledged

Risk Level

Severity: Informational

Code Segment

```
function cancelOwnershipTransfer() external override onlyOwner {
       // Emit an event indicating that the potential owner has been
cleared.
       emit PotentialOwnerUpdated(address(0));

       // Clear the current new potential owner.
       delete _potentialOwner;
   }
```

Description

The function always succeeds even if there is no pending ownership transfer request to be made. Logically, however, if there is no pending ownership transfer request, the function

should always revert. Reverting ensures that the function is only called under appropriate conditions.

Code location

```
src/utils/TwoStepOwnable.sol
```

Recommendation

Revert if the contract state variable **_potentialOwner** is **address(0)**

7. Code readability

Issue ID

LOCK-7

Status

Acknowledged

Risk Level

Severity: Informational

Code Segment

```
// location: HourglassTBTFactory.sol: L66-70
// emit the addition of the new depositor version
       emit NewDepositorAdded(_depositorImplementation, depBeacon,
versions.length);

       // store the version info
       versions.push(Version({depositorBeacon: depBeacon, isActive:
true}));

// location: HourglassTBTFactory.sol: L99-102
      emit NewMaturityCreated(deployments[index].length,
newDepositor);
```

```
        // add it to the deployments
        deployments[index].push(Deployment({depositor:
newDepositor}));

// location: HourglassTBTFactory.sol: L122-123
        emit VersionUpgraded(version,
IBeacon(beacon).implementation(), newImplementation);
        UpgradeableBeacon(beacon).upgradeTo(newImplementation);

// location: TwoStepOwnable.sol: L139-142
        emit OwnershipTransferred(_owner, newOwner);
        // Set the new owner.
        _owner = newOwner;
```

Description

Events should be emitted after applying the related state changes

Code location

```
src/HourglassTBTFactory.sol
src/utils/TwoStepOwnable.sol
```

Recommendation

We recommend emitting events after applying the related state changes

8. Postfix operators consume more gas than prefix operators

Issue ID

LOCK-8

Status

Resolved

Risk Level

Severity: Informational

Code Segment

```
// location: HourglassTBTFactory.sol: L31
for (uint256 i; i < depositors.length; i++) {
```

```
// location: HourglassTBTFactory.sol: L50
for (uint256 i; i < numDepositors; i++) {
```

Description

The use of postfix operators i++ consumes more gas than prefix operators ++i

Code location

```
src/HourglassTBTFactory.sol
```

Recommendation

Use prefix operators instead of postfix

Disclaimer

While best efforts and precautions have been taken in preparing this document, Arcadia and the Authors assume no responsibility for errors, omissions, or damages resulting from the use of the provided information. Additionally, Arcadia would like to emphasize that the use of Arcadia's services does not guarantee the security of a smart contract or set of smart contracts and does not guarantee against attacks. One audit on its own is not enough for a project to be considered secure; that categorization can only be earned through extensive peer review and battle testing over an extended period.