Solid World

Smart Contract Security Assessment

March 1, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of the Solid World protocol. The core Solid World Protocol is a set of mechanisms (e.g., tokenization, pooling, and ecosystem reward distribution to stakeholders/liquidity providers) that aim to create and incentivize a liquid market for forward carbon offset agreements.

The most critical subjects covered by the audit are the collateralization and decollateralization mechanisms, the reactive time appreciation calculations, the interactions with the staking contract, the accounting and distribution of rewards to stakers and access control.

The code and accompanying artifacts (e.g., test suite, documentation) have been developed with high professional standards. No security issues/threats were identified by the audit.

The main findings of the audit are (1) a potential out of gas situation in the RewardDistributor and DecollateralisationManager contracts and (2) a low possibility of the system ending up in a blocked state that requires the intervention of the protocol's admins. As these situations described in (1) would require states extremely out of the bounds of what Solid World identified as "business as usual", no code-level contingencies were deemed necessary, but were noted as possible scenarios. As for (2) the protocol team is in a position to monitor and actively act to avoid this from occurring, while the potential fixes would add unnecessary complexity to the system and its function. There were a few advisory issues raised, the majority of which were addressed.

SETTING & CAVEATS

The scope of the audit includes the core contracts of the Solid World protocol. The test suite was consulted during the audit but was not part of it. The full list of audited files can be found below:

```
solid-world-dao-contracts
└── contracts
    ├── CollateralizedBasketTokenDeployer.sol
    ├── CollateralizedBasketToken.sol
    ├── ForwardContractBatchToken.sol
    ├── libraries
    │   ├── ABDKMath64x64.sol
    │   ├── DomainDataTypes.sol
    │   ├── GPv2SafeERC20.sol
    │   ├── manager
    │   │   ├── CarbonDomainRepository.sol
    │   │   ├── CategoryRebalancer.sol
    │   │   ├── CollateralizationManager.sol
    │   │   ├── DecollateralizationManager.sol
    │   │   └── WeeklyCarbonRewards.sol
    │   ├── ReactiveTimeAppreciationMath.sol
    │   ├── RewardsDataTypes.sol
    │   └── SolidMath.sol
    ├── Pausable.sol
    ├── PostConstruct.sol
    ├── rewards
    │   ├── EmissionManager.sol
    │   ├── RewardsController.sol
    │   ├── RewardsDistributor.sol
    │   └── UniswapEACAggregatorProxyAdapter.sol
    ├── SolidStaking.sol
    ├── SolidWorldManager.sol
```

```
└── SolidWorldManagerStorage.sol
```

The audit report covers commit hash b955651893d36b1f946a7fabce2dd28284de002c of the at the time private repository solid-world-dao-contracts. Audited suggested fixes were also reviewed up to commit hash b3e79c2456ecca913be0ff165fd49992eba8e6e1.

Two auditors worked on the codebase for 2 weeks.

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 behavior. 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:

[NO HIGH SEVERITY ISSUES]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Potential out of gas situation in RewardDistributor and DecollateralisationManager contracts	**DISMISSED**

The RewardsDistributor::`getAllUnclaimedRewardAmountsForUserAndAsset()` function performs a nested loop that iterates over all possible rewards for all amounts staked by a given user. Since both of these amounts are potentially unbounded, an out of gas error may eventually occur.

```
//RewardsDistributor.sol::getAllUnclaimedRewardAmountsForUserAndAsset
function getAllUnclaimedRewardAmountsForUserAndAssets(
  address[] calldata assets, address user
)
  external
  view
  override
  returns (address[] memory rewardsList, uint[] memory unclaimedAmounts)
{
  RewardsDataTypes.AssetStakedAmounts[] memory assetStakedAmounts =
    _getAssetStakedAmounts(assets,user);
  rewardsList = new address[](_rewardsList.length);
  unclaimedAmounts = new uint[](rewardsList.length);
```

```
  for (uint i; i < assetStakedAmounts.length; i++) {
    for (uint r; r < rewardsList.length; r++) {
      rewardsList[r] = _rewardsList[r];
      unclaimedAmounts[r] += _assetData[assetStakedAmounts[i].asset]
        .rewardDistribution[rewardsList[r]]
        .userReward[user]
        .accrued;

      if (assetStakedAmounts[i].userStake == 0) {
        continue;
      }
      unclaimedAmounts[r] += _computePendingRewardAmountForUser(
        user,
        rewardsList[r],
        assetStakedAmounts[i]
      );
    }
  }
  return (rewardsList, unclaimedAmounts);
}
```

Similarly, the function `getBatchesDecollateralisationInfo()` of the contract `DecollateralisationManager` loops over all batchIds, the number of which could be unbounded. As already mentioned, this might eventually lead to an out of gas failure.

```
//DecollateralisationManger.sol::getBatchesDecollateralisationInfo()
function getBatchesDecollateralizationInfo(
  SolidWorldManagerStorage.Storage storage _storage,
  uint projectId,
  uint vintage
)
  external
  view
  returns (DomainDataTypes.TokenDecollateralizationInfo[] memory result)
{
```

```
  DomainDataTypes.TokenDecollateralizationInfo[] memory allInfos =
    new DomainDataTypes.TokenDecollateralizationInfo[](
      _storage.batchIds.length
    );
  uint infoCount;

  for (uint i; i < _storage.batchIds.length; i++) {
    uint batchId = _storage.batchIds[i];
    if (
      _storage.batches[batchId].vintage != vintage ||
      _storage.batches[batchId].projectId != projectId
    ) {
      continue;
    }

    (uint amountOut, uint minAmountIn, uint minCbtDaoCut) =
      _simulateDecollateralization(
        _storage,
        batchId,
        DECOLLATERALIZATION_SIMULATION_INPUT
      );

    // Dedaub: part of the code is omitted for brevity

    infoCount = infoCount + 1;
  }

  result = new DomainDataTypes.TokenDecollateralizationInfo[](infoCount);
  for (uint i; i < infoCount; i++) {
    result[i] = allInfos[i];
  }
}
```

This issue was discussed with the Solid World team, who estimated that the protocol will not use enough reward tokens, stakes or batchIds to cause it to run out of gas.

LOW SEVERITY:

ID	Description	STATUS
L1	Failure to call function updateCarbonRewardDistribution of the RewardsDistributor contract regularly can disrupt the normal function of the protocol	**ACKNOWLEDGED**

The RewardsDistributor contract has a public `UpdateCarbonRewardDistribution()` function that needs to be called by a keeper (or any user of the protocol) once per week in order to mint weekly rewards. A failure to do so leads to two issues.

Firstly, any call to the `updateCarbonRewardDistribution()` function will fail. This will happen because of the `canUpdateCarbonRewardDistribution` function, which will return false and cause a revert if `block.timestamp > nextDistributionEnd`. Failure to call `updateCarbonRewardDistribution()` for a week will make the aforementioned condition evaluate to true. In case this happens, the `distributionEnd` parameter will need to be manually overridden through a call to `RewardsController::configureAssets()` by specifying a new config with a new distributionEnd.

```
//RewardsDistributor.sol::updateCarbonRewardDistribution()
function updateCarbonRewardDistribution(
  address[] calldata assets,
  address[] calldata rewards,
  uint[] calldata rewardAmounts
) external override onlyEmissionManager {
  if (
    assets.length != rewards.length ||
    rewards.length != rewardAmounts.length
  ) {
```

```
    revert InvalidInput();
  }

  for (uint i; i < assets.length; i++) {
    if (!_canUpdateCarbonRewardDistribution(assets[i], rewards[i])) {
      revert UpdateDistributionNotApplicable(assets[i], rewards[i]);
    }
  }
  // Dedaub: part of the code is omitted for brevity
}
```

```
//RewardDistributor.sol::canUpdateCarbonRewardDistribution()
function _canUpdateCarbonRewardDistribution(address asset, address reward)
  internal view returns (bool)
{
  uint32 currentDistributionEnd =
    _assetData[asset].rewardDistribution[reward].distributionEnd;
  uint32 nextDistributionEnd =
    _computeNewCarbonRewardDistributionEnd(asset, reward);

  bool isInitializedDistribution = currentDistributionEnd != 0;
  bool isBetweenDistributions =
    block.timestamp >= currentDistributionEnd &&
    block.timestamp < nextDistributionEnd;

  return isInitializedDistribution && isBetweenDistributions;
}
```

Secondly, a failure to call `updateCarbonRewardDistribution()` until after what would have been the next distribution end, will cause the rewards of the period in between not to be minted. This is because the `SolidMath.computeWeeklyBatchReward()` function, which decides how many

rewards to mint, is stateless and does not depend on any parameter which keeps a history of the rewards which should have been issued.

```
//SolidMath.sol::computeWeeklyBatchReward()
function computeWeeklyBatchReward(
  uint certificationDate,
  uint availableCredits,
  uint timeAppreciation,
  uint rewardsFee,
  uint decimals
) internal view returns (uint netRewardAmount, uint feeAmount)
```

The Solid World team has been aware of the issue and the steps that are required to "unblock" the system in case it ends up in such a state.

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	Centralized minting of ForwardContractBatchTokens	**OPEN**

The owner of the SolidWorldManager contract is able to mint ForwardContractBatchToken tokens by creating a new batch using the `addBatch()` function, which calls the equivalent function from CarbonDomainRepository.

```
//CarbonDomainRepository.sol::addBatch()
function addBatch(
  SolidWorldManagerStorage.Storage storage _storage,
  DomainDataTypes.Batch calldata batch,
  uint mintableAmount
) external {
  // Dedaub: part of the code is omitted for brevity

  _storage._forwardContractBatch.mint(
    batch.supplier, batch.id, mintableAmount, ""
  );

  emit BatchCreated(batch.id);
}
```

It is understood that this process corresponds to an off-chain process which tokenises the promise of future carbon credits. In addition once a batch has been created, no more ForwardContractBatchToken tokens with the same id can be created, which serves as an additional mitigation.

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	Redundant use of the override modifier in several contracts	**RESOLVED**

In several contracts (most of which have been forked from Aave), many functions are marked with the override modifier when no such function is actually inherited by the parent contract. These are probably leftovers from the time (prior to Solidity 0.8.8) when the `override` keyword was mandatory when a contract was implementing a function from a parent interface

EmissionManager

- configureAssets
- setRewardOracle
- setDistributionEnd
- setEmissionPerSecond
- updateCarbonRewardDistribution
- setClaimer
- setRewardsVault
- setEmissionManager
- setSolidStaking
- setEmissionAdmin
- setCarbonRewardsManager
- getRewardsController
- getEmissionAdmin
- getCarbonRewardsManager

RewardsController

- getRewardsVault

- getClaimer
- getRewardOracle
- configureAssets
- setRewardOracle
- setClaimer
- setRewardsVault
- setSolidStaking
- handleUserStakeChange
- claimAllRewards
- claimAllRewardsOnBehalf
- claimAllRewardsToSelf

RewardsDistributor

- getRewardDistributor
- getDistributionEnd
- getRewardsByAsset
- getAllRewards
- getUserIndex
- getAccruedRewardAmountForUser
- getUnclaimedRewardAmountForUserAndAssets
- setDistributionEnd
- setEmissionPerSecond
- updateCarbonRewardDistribution

SolidStaking

- addToken
- stake
- withdraw
- withdrawStakeAndClaimRewards
- balanceOf
- totalStaked
- getTokensDistributor::getAllUnclaimedReward

Resolved in commit 1ad958b6f0d74507c038bd49da281a572e170907.

A2	Creation events could incorporate additional information	INFO

Creation events, `CategoryCreated`, `ProjectCreated`, `BatchCreated`, could include more information related to the category, project or batch associated with them.

A3	Storage related gas optimization	RESOLVED

The fields of DomainDataTypes::Category struct can be reordered to be tighter packed in 4 instead of 5 storage slots.

```
// DomainDataTypes.sol::Category
struct Category {
  uint volumeCoefficient;
  uint40 decayPerSecond;
  uint16 maxDepreciation;
  uint24 averageTA;
  uint totalCollateralized;
  uint32 lastCollateralizationTimestamp;
  uint lastCollateralizationMomentum;
}

// Dedaub: tighter packed version
struct Category {
  uint volumeCoefficient;
  uint40 decayPerSecond;
  uint16 maxDepreciation;
  uint24 averageTA;
  uint32 lastCollateralizationTimestamp;
  uint totalCollateralized;
  uint lastCollateralizationMomentum;
}
```

We measured that in certain test cases the use of less SLOAD and STORE instructions reduced the gas consumption by around 1.5-2% and did not cause any regression in

terms of gas consumption (and of course correctness).

Resolved in commit b3e79c2456ecca913be0ff165fd49992eba8e6e1.

A4	Compiler version and possible bugs	**RESOLVED**

The code is compiled with the floating pragma ^0.8.16. It is recommended that the pragma is fixed to a specific version. Versions ^0.8.16 of Solidity in particular, have some known bugs, which we do not believe affect the correctness of the contracts.

Resolved in commit d68cfaf512d5eb8da646780350713d6c98ad7da2.

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.