# Solid World

Smart Contract Security Assessment

Aug 3, 2023

Solid World

## ABSTRACT

Dedaub was commissioned to perform a security audit of the new Zap component of the Solid World protocol. The core Solid World Protocol, which has been audited by Dedaub before, 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 this audit are the 3 zap contracts that aim to ease the process of collateralization, decollateralization and staking for the protocol users.

The code and accompanying artifacts (e.g., test suite, documentation) have been developed with high professional standards. No security issues/threats that could lead to theft were identified by the audit. No issues leading to loss of funds resulting from the intended use of the system were identified by the audit.

## SETTING & CAVEATS

The scope of the audit includes a number of new zap contracts. The test suite was consulted during the audit but was not part of it. The full list of audited files is:

```
solid-world-dao-contracts
└── zap
    ├── collateralize
    │   ├── BaseSolidZapCollateralize.sol
    │   └── SolidZapCollateralize.sol
    ├── decollateralize
    │   ├── BaseSolidZapDecollateralize.sol
    │   └── SolidZapDecollateralize.sol
    ├── staking
    │   ├── BaseSolidZapStaker.sol
    │   └── SolidZapStaker.sol
    └── BaseZap.sol
```

The audit report covers commit hash 56f2684ce232345ae1fcb9a4e4ff48bb79750ef5 of the public repository solid-world-dao-contracts. Audited suggested fixes were also reviewed up to commit hash f18289f78168ade370523279c5c87d5e625335f2.

Two auditors worked on the codebase for 3 days.

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:<br>• User or system funds can be lost when third-party systems misbehave.<br>• DoS, under specific conditions.<br>• Part of the functionality becomes unusable due to a programming error. |
| LOW | Examples:<br>• Breaking important system invariants but without apparent consequences.<br>• Buggy functionality for trusted users where a workaround exists.<br>• 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 | Simulation functions in SolidZapStaker have side effects and can result in trapped funds if used manually by a user | **WON’T FIX** |

The SolidZapStaker contract has a number of external simulation functions, such as simulateStakeSingleSwap, simulateStakeDoubleSwap and simulateStakeETH. These functions have side effects, going so far as to cause the deployment of liquidity, and only stop short of staking. It is understood that these functions will only be called by off-chain components, but in the event of a user inadvertently calling them, this will cause a loss of funds for the user. This is because the resulting LP tokens will be trapped in the contract instead of being returned to them or staked. It is recommended that these functions be made available only to selected addresses to avoid this situation.

## LOW SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| L1 | In certain cases BaseZap::_sweepTokens should not accept a 0 sweptAmount | **RESOLVED** |

Function `BaseZap::_sweepTokensTo()` does not revert when the `sweptAmount` is equal to `0` even though there are certain scenarios where it should, i.e., when an amount of CRISP tokens is swapped to output tokens, which are eventually swept. There are other cases where the sweptAmount might be `0` and the execution should not revert, i.e., when it refers to CRISP tokens dust which could very well be `0`.

| ID | Description | STATUS |
|---|---|---|
| L2 | SolidZapCollateralize::_sweepETHTo should not accept a 0 sweptAmount | **RESOLVED** |

Function `SolidZapCollateralize::_sweepETHTo()` checks the balance of WETH token and returns normally even if it is zero, even though it is called after a swap whose output token should be WETH.

**SolidZapCollateralize::_sweepETHTo**

```
function _sweepETHTo(address zapRecipient)
    private returns (uint sweptAmount)
{
    sweptAmount = IERC20(weth).balanceOf(address(this));
    if (sweptAmount == 0) {
        return 0;
    }

    IWETH(weth).withdraw(sweptAmount);
    (bool success, ) =
        payable(zapRecipient).call{ value: sweptAmount }("");
    if (!success) {
        revert ETHTransferFailed();
    }
}
```

Requiring that the WETH balance (or its difference) is different from 0 would verify that the swap's calldata are correct and the output token is indeed WETH.

| | | |
|---|---|---|
| L3 | Unsafe use of approve in BaseZap::approveTokenSpendingIfNeeded function | **RESOLVED** |

The `BaseZap::_approveTokenSpendingIfNeeded()` function of the BaseZap contract makes use of an approve call on the function's token parameter. Since this token is arbitrary, the approve call is unsafe. This is due to the fact that certain tokens do not implement the ERC20 interface properly and do not return a success value. Some compilers insert a check for a success value and force the code to revert if there is no return data. Thus this call to approve can fail unexpectedly. Use safeApprove instead which returns a default success value if none is provided.

**BaseZap::_approveTokenSpendingIfNeeded()**

```
function _approveTokenSpendingIfNeeded(address token, address spender) internal {
        // Dedaub - Will silently fail if spender already has some allowance.
        if (IERC20(token).allowance(address(this), spender) == 0) {
        // Dedaub - This call will fail if the token does not return a value.
            IERC20(token).approve(spender, type(uint).max);
        }
    }
```

## 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.)

[NO CENTRALISATION ISSUES]

## 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 | SolidZapStake's _singleSwap and _doubleSwap could revert early if the swap outputs are 0 | **RESOLVED** |
| | Functions `_singleSwap()` and `_doubleSwap()` of the `SolidZapStake` contract do not require that the `swapResults` balances are different from `0`, i.e., it is assumed that the swaps' calldata will be correct. In a different scenario, where the aforementioned assumption does not hold, the execution will revert when the contract will attempt to add (`0`) liquidity to Gamma. We believe it would be better to check the `swapResults` balances and revert early if they are `0`. | |
| A2 | BaseSolidZapCollateralize could restrict the senders of ETH to just the WETH token contract | **RESOLVED** |
| | BaseSolidZapCollateralize `receive()` and `fallback()` could check that the msg.sender is the WETH contract, as it is expected that this should be the only contract that is able to send ETH to the BaseSolidZapCollateralize contract. | |
| A3 | Functions which accept or return ETH still accept arbitrary swaps | **WON'T FIX** |
| | Certain functions such as `SolidZapDecollateralize::zapDecollateralizeETH()` or `SolidZapCollateralize::zapCollateralizeETH()` accept ETH as input or expect it as output. However, they still accept a swap parameter which is controlled by the caller, which can specify a swap which does not involve ETH. It is understood that these functions are meant to be called through the UI, however the possibility of user | |

| | | |
|---|---|---|
| error if called manually remains. It may make sense to construct part of the swap on-chain. | | |
| A4 | Two different interfaces called SWManager are present | **RESOLVED** |
| Two different interfaces called SWManager are defined in SolidZapDecollateralize and in SolidZapCollateralize contracts. The interfaces are different, but have the same name. Consider consolidating these into one interface file to avoid problems in the future. | | |
| A5 | Unclear naming for SolidZapStaker::stakeETH() function | **WON'T FIX** |
| The `SolidZapStaker::stakeETH()` function performs a stakeDoubleSwap but this is not evident from its name. The caller does not know if he is making a singleSwap or doubleSwap from the interface. Consider improving the name for usability purposes. | | |
| A6 | Misleading comments in ISolidZapStaker and ISolidZapCollateralise interfaces. | **RESOLVED** |
| The ISolidZapStaker interface has a number of misleading comments. For instance, both versions of `ISolidZapStaker::stakeSingleSwap()` say that it performs a partial swap but this is not the case. The same problem is present in the comments for `ISolidZapStaker::simulateStakeSingleSwap()`.<br><br>On the other hand the ISolidZapCollateralize interface's comments for both versions of the zapCollateralize function state that the outputToken is 'The token used for obtaining forward credits' when this is in fact the token which is received at the end of the collateralization operation. | | |

## DISCLAIMER

## 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.