DefiSaver V3 Strategies

Smart Contract Security Assessment

Dec. 22, 2021

ABSTRACT

Dedaub was commissioned to perform a security audit on Decenter's DefiSaver V3 strategy smart contracts at commit hash `7b7b64d3a76ef9bdc3f6659a6721c96a5c65182f`. The code can be found in the DeFi Saver v3 contracts Github repository. The scope of the audit, as defined by the client, includes the directories/files:

- core
- auth
- triggers, in terms of their base model and not detail
- ActionBase.sol
- actions/checkers
- actions/fee
- utils/TempStorage.sol.

We have audited earlier versions of the DeFi Saver contracts, but not the full Strategy/Recipe automation that is defined in the above code. We avoid repeating the main system architecture elements and caveats from previous audits, focusing instead on the audited subset.

SETTING & CAVEATS

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than 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, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

The scope of this audit includes the contracts that handle Strategies: composite operations consisting of a number of triggers (preconditions), a number of modular actions, as well as a parameter mapping array that is used to assign action parameters from subscription data or the return value of a previous call. Related/alternative strategies can be combined into Bundles. Users can create Strategies and Bundles using

the corresponding {Strategy|Bundle}Storage contract, which can be configured to either be open to anyone or limited to a single address. A user can subscribe to a Strategy or Bundle using the SubStorage contract by providing a StrategySub which includes the Strategy/Bundle id, triggerData, and subData (provided to actions), which is hashed and stored as part of the StoredSubData identifying the Subscription instance. To interface with the above mentioned *Storage contracts, the StrategyProxy and SubProxy contracts are provided for users to delegate-call into using their DSProxies.

Upon subscribing to a Strategy, users give the ProxyAuth contract permission to call the `execute` method of their DSProxy. The ProxyAuth contract can only do that when called by the StrategyExecutor contract which, in turn, only allows whitelisted bots to call its protected functionality.

The definition of a Strategy determines how much "freedom" a bot has when initiating the execution of a subscription instance/Recipe. For the Strategy's triggers it can be deduced by inspecting their source code, with the current implementations favoring subscription data to bot-provided calldata (only the McdRatioTrigger makes use of the calldata). For the Strategy's actions their code needs to be inspected together with the parameter mapping that is part of the Strategy definition.

The stateful contracts mentioned throughout this report are registered on the DFSRegistry contract which serves as the project's central registry, allowing the project's owner to add new entries/contracts and update them after a waiting period. References to intra-protocol contracts in most of the codebase are resolved using the DFSRegistry, making it a centralization element of the protocol. It should be noted that the waiting period is defined per-entry and it doesn't have a hard-coded minimum value so it can be as small as the owner sets it to be.

In addition, most contracts audited are subcontracts of the AdminAuth, allowing the project's administrators to easily retrieve stuck funds or destroy the contracts if such a need arises, without affecting their users in a negative way.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect 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: 01) User or system funds can be lost when third party systems misbehave. 02) DoS, under specific conditions. 03) Part of the functionality becomes unusable due to programming error.
LOW	Examples: 01) Breaking important system invariants, but without apparent consequences. 02) Buggy functionality for trusted users where a workaround exists. 03) Security issues which may manifest when the system evolves. 04) Significant inefficiencies.

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors.

CRITICAL SEVERITY

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	The SubStorage::isValidId modifier has an off-by-one error	**CLOSED**

The modifier SubStorage::isValidId is defined as:

```
modifier isValidId(uint256 _id, bool _isBundle) {
 if (_isBundle) {
  if (_id > BundleStorage(registry.getAddr(BUNDLE_STORAGE_ID)).getBundleCount()) {
    revert SubIdOutOfRange(_id, _isBundle);
  }
 } else {
  if (_id > StrategyStorage(registry.getAddr(STRATEGY_STORAGE_ID)).
                                                    getStrategyCount()) {
    revert SubIdOutOfRange(_id, _isBundle);
  }
 }
 _;
}
```

However, both `getBundleCount` and `getStrategyCount` return a length. Valid elements are up to length-1, not up to length. It is not clear if this bug can have an escalating impact, since the strategy/bundle fields will be empty anyway. However, it should be fixed to prevent any potential for serious issues.

L2	Inconsistent ratio returned by the RatioHelper contracts for different protocols	**CLOSED**

The scope of this audit includes ratio helper contracts for Compound, Liquity, Maker, and Reflexer, which are used by the corresponding ratio trigger contracts. These helpers do not consistently handle the case where the total debt amount is equal to 0. The compound helper returns a ratio of uint256.max, while the Liquity and Maker helpers return 0. In addition, the Reflexer helper does not consider such a case, which will lead to an Error if it arises.

L3	LiquityRatioHelper can read a stale PriceFeed oracle price	**CLOSED**

The getRatio() method of the LiquityRatioHelper contract uses Liquity's PriceFeed to get the ETH/USD price. However it uses the stored 'lastGoodPrice' value instead of the fetchPrice() method offered by the PriceFeed that recomputes a fresh price value by querying other oracles. This can lead to the method returning an inaccurate ratio.

L4	TempStorage pattern can be unsafe if malicious actors get the control-flow	**CLOSED**

The TempStorage contract serves as an intra-transaction (Recipe execution) way to pass values around (from one trigger/action to another) when this cannot be done via call/return-data. Its current implementation however allows any caller to overwrite any value, which can lead to an exploit if an attacker manages to gain the control-flow (via CALL) during the execution of a Recipe.

```
function set(string memory _id, bytes32 _value) public {
   storageSlot[_id] = _value;
}

function get(string memory _id) public view returns (bytes32) {
   return storageSlot[_id];
}
```

OTHER/ ADVISORY ISSUES:

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

ID	Description	STATUS
A1	The Exponential/CarefulMath libraries no longer return error codes	**DISMISSED**

The Exponential and CarefulMath libraries are being used in the code. However, they are compiled with Solidity 0.8, whereas the code clearly expects earlier versions of Solidity and detects overflows/underflows. In some cases, the check is performed before the operation, and the library works as before: it returns an error code. In others, the operation is performed before the check and the transaction will revert in case of an overflow, based on the Solidity 0.8 checking. This is the case, for instance, for multiplication:

```
function mulUInt(uint a, uint b) internal pure returns (MathError, uint) {
        if (a == 0) {
                return (MathError.NO_ERROR, 0);
        }

        uint c = a * b;

        if (c / a != b) {
                return (MathError.INTEGER_OVERFLOW, 0);
        } else {
                return (MathError.NO_ERROR, c);
        }
}
```

The developers should consider whether reverting is equally acceptable. The code we have inspected does not check the return/success values of Exponential operations, therefore we expect that the developers already account for reverting behavior.

A2	Constants are being redefined in Strategy-based contracts	**CLOSED**

The following constants are repeatedly defined in various contracts that all inherit from StrategyModel:

```
DFSRegistry public constant registry = DFSRegistry(REGISTRY_ADDR);
bytes4 constant STRATEGY_STORAGE_ID = bytes4(keccak256("StrategyStorage"));
bytes4 constant SUB_STORAGE_ID = bytes4(keccak256("SubStorage"));
bytes4 constant BUNDLE_STORAGE_ID = bytes4(keccak256("BundleStorage"));
```

E.g., `STRATEGY_STORAGE_ID` is defined (identically, as aboce) in RecipeExecutor, BundleStorage, StrategyProxy, SubStorage, StrategyTriggerView.

These constant definitions could move to StrategyModel and be transparently inherited, avoiding all potential for inconsistencies and shortening the code.

A3	View functions BundleStorage::getPaginatedBundles and StrategyStorage::getPaginatedStrategies return wrong-length array for the final page of contents.	**DISMISSED**

These functions return longer arrays for the final page of contents, with the array contents being zero. It is possible that the current off-chain caller of these functions can tolerate the inconsistency, but it seems unlikely that this behavior is ideal and future uses might not.

For instance, the definition of `StrategyStorage::getPaginatedStrategies` is:

```
function getPaginatedStrategies(uint _page, uint _perPage) public view
returns (Strategy[] memory) {
       Strategy[] memory strategiesPerPage = new Strategy[](_perPage);

       uint start = _page * _perPage;
       uint end = start + _perPage;

       end = (end > strategies.length) ? strategies.length : end;

       uint count = 0;
       for (uint i = start; i < end; i++) {
              strategiesPerPage[count] = strategies[i];
              count++;
       }

       return strategiesPerPage;
}
```

Therefore, the array is always of length `_perPage`. Extra elements remain blank.

A4	ChainLinkPriceTrigger does not check the freshness of the data fetched	**DISMISSED**

The ChainLinkPriceTrigger uses the latestRoundData() method of the Chainlink FeedRegistry without checking when that value was stored. It can be enhanced to optionally also check that for its trigger logic.

A5	Variable written to TempStorage can be overwritten in some executions	**CLOSED**

In the isTriggered() method of the McdRationTrigger contract, the "MCD_RATIO" value is written to the TempStorage contract to later be read by the corresponding checker. However its semantics do not seem clear because under one condition (`RatioCheck(triggerCallData.ratioCheck) == RatioCheck.BOTH_RATIOS`) the variable written to the TempStorage can be overwritten once before being stored:

```
if    (RatioCheck(triggerCallData.ratioCheck)    ==    RatioCheck.CURR_RATIO    ||
RatioCheck(triggerCallData.ratioCheck) == RatioCheck.BOTH_RATIOS){
   checkedRatio = getRatio(triggerSubData.vaultId, 0);
   shouldTriggerCurr = ...
}

if    (RatioCheck(triggerCallData.ratioCheck)    ==    RatioCheck.NEXT_RATIO    ||
RatioCheck(triggerCallData.ratioCheck) == RatioCheck.BOTH_RATIOS){
   checkedRatio = getRatio(triggerSubData.vaultId, triggerCallData.nextPrice);
   shouldTriggerNext = ...
}

TempStorage(tempStorageAddr).set("MCD_RATIO", bytes32(checkedRatio));

return shouldTriggerCurr || shouldTriggerNext;
```

A6	DFSRegistry: Semantics of previousAddresses mapping unclear	**CLOSED**

The DFSRegistry contains a previousAddresses mapping to support instant rollbacks to the previous address if a problem is detected with an updated implementation.

```
function addNewContract(
   bytes4 _id,
   address _contractAddr,
```

```
    uint256 _waitPeriod
) public onlyOwner {
    if (entries[_id].exists){
        revert EntryAlreadyExistsError(_id);
    }

    entries[_id] = Entry({
        contractAddr: _contractAddr,
        ...
    });

    // Remember tha address so we can revert back to old addr if needed
    previousAddresses[_id] = _contractAddr;
    emit AddNewContract(msg.sender, _id, _contractAddr, _waitPeriod);
}

function revertToPreviousAddress(bytes4 _id) public onlyOwner {
    if (!(entries[_id].exists)){
      revert EntryNonExistentError(_id);
    }
    if (previousAddresses[_id] == address(0)){
      revert EmptyPrevAddrError(_id);
    }

    address currentAddr = entries[_id].contractAddr;
    entries[_id].contractAddr = previousAddresses[_id];

    emit RevertToPreviousAddress(msg.sender, _id, currentAddr,
previousAddresses[_id]);
}
```

However its semantics are currently not clear for some edge cases:

1. When a new Entry/Contract is added the previousAddress for it is the same as its current address. This means that the call to the revertToPreviousAddress() method can succeed without actually performing a rollback.
2. The 0 address is never passed as a value to the previousAddresses mapping, making the check that results in throwing an EmptyPrevAddrError() obsolete.

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, as well as a public bug bounty program.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.