DeFi Saver v3.1

Add Safe Wallet Support (delta)

Smart Contract Security Assessment

Feb 22, 2024

ABSTRACT

Dedaub was commissioned to perform a security audit of the latest upgrade to the DeFi Saver protocol. 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 or issues leading to loss of funds resulting from the intended use of the system were identified by the audit.

Defi Saver is an advanced management dashboard for DeFi. It is built on top of popular DeFi protocols and enables advanced actions and combinations of them, which can be executed manually or in an automated way. Users interact with Defi Saver through smart wallets, namely the DSProxy wallet, thus the protocol does not hold any user funds directly, only users authorizations for strategy automations.

The latest version of the protocol (v3.1) enables compatibility with Safe smart wallets and implements several refactorings/simplifications to the codebase. The whole code delta is considered, while the main audit task to ascertain is to ensure the correct and secure enabling of Safe smart wallets.

SETTING & CAVEATS

This audit report mainly covers the contracts of the defisaver-v3-contracts repository of the DeFi Saver Protocol at commit ca39459cbc46c2829315c5d3ee1e0d409c190b27.

As part of the audit, we also reviewed the fixes for the issues included in the report. The fixes were delivered as part of PR #345 and we can attest that they have been implemented correctly.

Two auditors worked on the codebase for 10 days on the following contracts:

```
contracts/
├── actions/
│   ├── ActionBase.sol
│       ├── aaveV3/*
│       ├── checkers/
│       │   └── AaveV3RatioCheck.sol
│       ├── exchange/
│       │   └── DFSSell.sol
│       ├── fee/*
│       └── flashloan/*
├── auth/*
├── core/*
├── exchangeV3/*
├── triggers/
│   └── AaveV3RatioTrigger.sol
└── utils/
    ├── BotRefills.sol
    ├── CheckWalletType.sol
    ├── DFSProxyRegistry.sol
    ├── DFSProxyRegistryController.sol
    ├── DefisaverLogger.sol
    ├── Discount.sol
    ├── FLFeeFaucet.sol
    ├── FeeReceiver.sol
    ├── FeeRecipient.sol
    ├── SafeERC20.sol
    ├── TokenPriceHelper.sol
    ├── TokenPriceHelperL2.sol
    ├── TokenUtils.sol
    └── TransientStorage.sol
```

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	No check for sequencer uptime can lead to strategies/actions being executed at wrong prices	**ACKNOWLEDGED**

The protocol is deployed on several L2 blockchains, namely Arbitrum, Base and Optimism, which use a centralized sequencer that executes and rolls up the L2 transactions by batching multiple transactions into a single transaction. Thus, if the sequencer is down, messages cannot be transmitted from L1 to L2 and no L2 transactions are executed. In that case chainlink price oracles might have stale prices leading to strategies triggering untimely or being executed at the wrong prices. Chainlink recommends using their L2 sequencer uptime feeds to be able to diagnose downtimes and act accordingly, e.g., revert if the sequencer is down (message/tx sent from L1 to L2 and queued until the sequencer is back up) or even if it is up for a small period of time.

Comments:

According to the protocol team, the presented concerns are valid but the suggestion of a fixed `GRACE_PERIOD` might do more harm than good for the end users in certain scenarios. For instance, if the sequencer recovers and the price starts falling rapidly, waiting for the `GRACE_PERIOD` to end before executing the strategies might mean that *users are liquidated in the meantime. The risk of not having it and executing a strategy with a stale price, is deemed lower. The protocol team intends to handle this case in their backend system as only their bots have execution authorization, thus they can make sure to not execute strategies under stale prices, while avoiding the limitation of the `GRACE_PERIOD` in cases where prices are falling rapidly.*

M2	Missing oracle staleness checks	**ACKNOWLEDGED**

In the `TokenPriceHelper` contract, there are several functions that fetch token prices from different price feeds including Chainlink and Aave. However, the prices returned are not validated nor checked for staleness. Even though the impact of using stale values may be restricted, it is recommended to add such checks to ensure that all the consumed prices are fresh.

As far as the Chainlink oracles are concerned, the following checks could be used to detect stale or invalid values:

```
require(updatedAt != 0);
require(updatedAt <= block.timestamp);
require(block.timestamp - updatedAt <= STALE_PRICE_DELAY);
```

Regarding the `STALE_PRICE_DELAY` initialization, the oracles perform updates when their *deviation threshold* is crossed or when their *heartbeat* time expires. As a result, the `STALE_PRICE_DELAY` could be set to a value relative to the predefined *heartbeat* to ensure that if no updates have occurred until then, the returned value is considered stale. Since different feeds have different *heartbeats*, a mapping from feed to *heartbeat* should be used instead of a single general and thus inaccurate *heartbeat* for all the feeds.

Comments:
According to the protocol team, it would be hard to maintain custom `STALE_PRICE_DELAY` values for each feed and network while the impact would be low.

Also, as mentioned in the comments of issue M1, it would be more appropriate for the backend system to implement any checks for stale oracle -reported prices.

LOW SEVERITY:

ID	Description	STATUS
L1	Token allowances are not handled optimally	**DISMISSED**

The function `TokenUtils::approveToken` is responsible for handling token approvals.

`TokenUtils::approveToken():14`

```
function approveToken(
    address _tokenAddr,
    address _to,
    uint256 _amount
) internal {
    if (_tokenAddr == ETH_ADDR) return;

    if (IERC20(_tokenAddr).allowance(address(this), _to) < _amount) {
        // Dedaub: safeIncreaseAllowance could be used here
        IERC20(_tokenAddr).safeApprove(_to, _amount);
    }

    // Dedaub: if the new allowance (_amount) is less than the old,
              safeDecreaseAllowance could be used
}
```

The aforementioned function allows only approval increases. It is understood that it is expected to be used in such a context and that there are other functions that support approval decreases. Nevertheless, we would suggest also supporting approval decreases via this function for completeness reasons. Regarding approval increases, it is recommended to perform them using a `safeIncreaseAllowance` implementation instead of a plain safeApprove, even though it is understood that approvals will be issued in a context where the spender is not expected to front-run the user.

L2	Bundled subscriptions could be blocked from executing all their strategies	**DISMISSED**

In the StrategyExecutor contract, the executeStrategy function is used by the authorized bots to execute single and bundled strategies. For the bundled strategies case, there exists a scenario where some of the bundled strategies might get blocked/fail to execute.

Strategies can be defined as continuous, which would allow an action to be repeated multiple times. The executeRecipeFromStrategy function of the RecipeExecutor contract, however, deactivates the subscription ID (_subID) related to a bundle of strategies if one of them is executed and is not continuous.

RecipeExecutor::executeRecipeFromStrategy():159

```
function executeRecipeFromStrategy(
    uint256 _subId,
    ...
) public payable {
    ...
    // Dedaub: When the executed strategy is not continuous
    //         the _subId is being deactivated
    if (!strategy.continuous) {
        SubStorage(SUB_STORAGE_ADDR).deactivateSub(_subId);
    }
    ...
}
```

As a result, a bot that is meant to execute the strategies of a bundle could be blocked in case one of them has been declared as non-continuous. Consider the following steps for example:

- `StrategyStorage::createStrategy` is used to create the desired strategies
  - One of them is initialized as non-continuous
- `BundleStorage::createBundle` is used to create a new bundle out of these strategies
- A bot calls the `StrategyExecutor::executeStrategy` providing the `_subId` of the bundle and the index of the strategy which should be executed
  - Assume that the index points to the strategy which was created as non-continuous
- The call delegates to the `RecipeExecutor::executeRecipeFromStrategy` function which executes the strategy
- During this call, however, the check:

  ```
  if (!strategy.continuous) {
      SubStorage(SUB_STORAGE_ADDR).deactivateSub(_subId);
  }
  ```

  will deactivate the used `_subId`, which would be the bundle ID
- Then, when the bot will try to execute another strategy from the bundle this would fail due to the following check it will fail since the bundle would have been deactivated:
  `StrategyExecutor::executeStrategy:60`

  ```
  StoredSubData memory storedSubData =
    SubStorage(SUB_STORAGE_ADDR).getSub(_subId);

  if (!storedSubData.isEnabled) {
      revert SubNotEnabled(_subId);
  }
  ```

Even though this does not seem to pose any direct threat or DoS in strategy execution, it would make sense to ensure that allowing bundling continuous with non-continuous strategies is the desired behavior and would not introduce any issues in future changes.

A possible use case in allowing non-continuous strategies to be bundled with continuous ones would be to add the non-continuous strategy as the last strategy of the bundle to deactivate the `_subId` after executing all of them to prevent re-execution by the bot.

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	Obsolete ds-proxy references	**RESOLVED**
The variable `userProxy` of the `StrategyModel::StoredSubData` struct should be renamed, possibly to `wallet`. Obsolete references to the ds-proxy can also be found in the `SubStorage` contract, namely in the `Subscribe` event and in the `@dev` comment of the `SubStorage::updateSubData` function.		
A2	Unused struct variable	**INFO**
The variable `user` of the `DFSExchangeData::ExchangeData` struct is never used.		
A3	No-op addition	**RESOLVED**
In `CurveUsdSwapper::takeSwapFee`, the amount `_sellAmount / _dfsFeeDivider` can be assigned to `feeAmount` instead of being added to it, as `feeAmount` is `0`.		

A4	Remaining TODOs in contracts with hardcoded addresses	**RESOLVED**
In some of the contracts that contain the hardcoded addresses there are some `TODO`s or `NOT LIVE ADDRESS` left in comments. Moreover, some of the addresses used seem to be placeholders. For example, in `ArbitrumCoreAddress` the `0xEeee...EEEeE` is used for the `MODULE_AUTH_ADDR`. We raise this here for visibility and to ensure that all the addresses point or have been updated to the correct ones for all the modules.		
A5	Unreversed `constant internal` variables	**RESOLVED**
In several of the contracts that contain the hardcoded addresses, some of the addresses are declared as `constant internal` when the majority of them are declared as address `internal constant`. We are mentioning this with uniformity in mind, since we found similar changes among the commits under scope.		
A6	Compiler bugs	**INFO**
The code is compiled with Solidity `0.8.10`. Version `0.8.10`, in particular, has some known bugs, which we do not believe affect the correctness of the contracts.		

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.