Dedaub

Security Technology for Smart Contracts

DECENTER

DeFi Saver Automation V1.0

Smart Contract Security Assessment

Date: Feb. 2, 2021

Abstract

Dedaub was commissioned to perform a security audit on the DeFi Saver automation app, currently deployed to the Ethereum mainnet. Four auditors worked on the task over the course of over two working weeks. We reviewed the code in significant depth, processed it through automated tools, explored actual transactions, and generally attempted to thoroughly validate the security model of the app. We also decompiled the code and analyzed it, using our static analysis (incl. symbolic execution) tools, to detect possible issues.

Setting and Caveats

The code base is significant in size, at a total of 10KLoC, with ~6KLoC being substantial (non-reused, non-interface) functionality. There is architectural repetition, with three major components (Aave, Compound, MCD) having similar overall workflow, yet different specifics.

No audit of a code base this large can be exhaustive. The audit focused on security, establishing the overall security model and its robustness. Since the code is deployed and operational, functional correctness (e.g., that the calculations are correct) was a secondary priority. Functional correctness is anyway best determined by thorough testing, or by a formal-verification kind of audit against full specifications.

Security Model

[*This is the main audit message. Please read carefully.*]
The security model of the app is solid. User funds are secured from attack based on the following general principles:

- Transfers of tokens are done under the user's direct control (i.e., only `transferFrom(msg.sender,...)`, in AllowanceProxy) and are initiated externally.
- The user's funds are held in a DSProxy contract ("smart wallet") owned by the user's account.
- Code "upstream" from the user's DSProxy is carefully guarded: an action affecting user funds can only be initiated by trusted external entities ("bots").
- Code "downstream" from the user's DSProxy is authorized to invoke DSProxy functionality only for the duration of the action (boost or repay). During that time there is no possibility of execution of untrusted code, unless some external service (e.g., Aave or Maker) is

compromised. Code downstream from the DSProxy maintains the excellent property of being stateless (no writing to storage): all effects are limited to the user's smart wallet, through the authorization mechanism.

In future evolution of the codebase, there are two threats that developers should be aware of (these threats are theoretical and unlikely, so we merely document them):

- The temporal authorization of specific code (flash loan callback: e.g., MCDSaverFlashLoan or CompoundSaverFlashLoan) is attackable via reentrancy attacks. What makes the code safe is the invocation of trusted components only. Such components may be generalized in the future, however. E.g., if Maker or Aave ever integrate untrusted tokens, the user may be fooled to interact with one, and a call to a malicious token could result in re-entering the DeFi Saver code (at the point of a flash loan callback) with arbitrary parameters.
- DSProxy does a `delegatecall` into the DeFi saver contracts ("downstream"). This is currently not dangerous, since the downstream code is stateless. In the future, if state is added, a dummy base contract that keeps DSProxy's storage structures separate will be necessary. DSProxy currently has storage structures as follows:

```
uint256 _authority; // STORAGE[0x0] bytes 0 to 19
uint256 _owner; // STORAGE[0x1] bytes 0 to 19
uint256 _cache; // STORAGE[0x2] bytes 0 to 19
```

Trust Model/Centralization Elements

[*This section is included for context, although its contents should already be known to the commissioner of an audit.*]
Users of the service should be aware that (upon subscribing) they are trusting off-chain elements, such as bot code and the owner and admin of the DeFi Saver Automation service. For instance, the service owners can activate bots that boost or repay the user's positions.

In current deployment, DeFi Saver (e.g., owner of MCDMonitorV2) is represented by a Gnosis Multisig wallet with 5 owners.

Full list of other audit findings can be found below.

Critical Severity

No critical severity vulnerabilities were identified

High Severity

No high severity vulnerabilities were identified

Medium Severity

Description	Status
It is possible for an external user to call `subscribe` in SubscriptionsV2/ CompoundSubscriptions/AaveSubscriptions/AaveSubscriptionsV2 without going through `subscribe` in the corresponding subscriptions proxy contract (e.g. SubscriptionsProxyV2). Whether this raises potential for misuse (e.g., a denial-of-service attack, by making the bots waste gas for a non-authorized smart wallet) depends on external code, which we have not seen. But it seems likely that external code reads the subscriber information provided by SubscriptionsV2 directly.	Open
AdminAuth currently isn't working correctly, with the setAdminByOwner function containing the line "`require(admin == address(0))`". This is known. In general, this file does not seem well-motivated: if an admin can change the owner and the owner can change the admin, then there is hardly a difference in privilege between the two.	Open
It is not clear what mechanism is used to keep constant addresses in the code consistent. E.g., `MONITOR_PROXY_ADDRESS = 0x181....` in SubscriptionsProxyV2 is likely inconsistent. Some constant addresses in AAVESubscriptionProxyV2, AAVESubscriptionProxy are incorrect.	Open
Arbitrary users can call `callFunction` in AaveSaverReceiver and `executeOperation` in CompoundSaverFlashLoan/MCDSaverFlashLoan. Although such users will not be authorized to interact with the user's DSProxy, they can still call `transfer` (either of ETH or of tokens) of funds of the contract itself. One instance of AaveSaverReceiver has a small amount of WETH to repay flashloan fees, so maybe this mistake could be repeated. Same issue with AaveSaverReceiverV2.	Open

Low Severity

Description	Status
There is some replicated functionality with some inconsistency. [Marked here when it may be more important than a mere suggestion (e.g., affecting authorization code), other instances in the "lowest" category.] ● In admin/authorization mechanisms ○ E.g., DSGuard/DSAuth but also local permission mechanisms AdminAuth (authorize for an Owner and an Admin), Auth (the above plus a mapping of authorized addresses--seems unused), and even: ■ ad hoc "allowed" mapping in CompoundMonitorProxy ■ similar ad hoc mapping in MCDPriceVerifier ● The MCD code uses a DSGuard/DSAuth directly, instead of ProxyPermission, as in the Compound and Aave code. ● In the Compound code, the `onlyAllowed` functionality has no counterpart in Maker code. Seems to currently be unused (`addAllowed` is only called externally, not from the reviewed contracts) so that it reverts to just an owner check. ● Inconsistency in structure and naming of the 3 different components (Compound/Aave/Maker). The contracts under the 3 "saver" subdirectories could be named and structured consistently, to make each one's role apparent. We recommend maintaining Auth and AdminAuth (possibly renaming, so that the difference is evident) and reusing uniformly. Similarly, having a uniform way of treating the DSProxy permissions (i.e., not inheriting ProxyPermission in some cases but doing `guard.permit` directly in MCDSubscriptionsProxyV2).	Open
Use of Solidity's `transfer()` primitive: The use of Solidity's `send()` and `transfer()` primitives for transferring ETH with low (2300) gas should be avoided if possible because they assume that the gas costs of the EVM instructions don't change. However these costs can change and have changed in the past, causing the `receive()` functionality of contracts to fail with 2300 gas. Most of the `transfer()` calls are to EOAs or known addresses with minimal/empty `receive()` functions (unlikely to ever fail). However in some cases funds are transferred to the owner of the DSProxy (e.g. AaveSaverProxy.sol: line 50) which can be a contract. A failed such transfer can cause the transaction to fail, and the bot to waste gas.	Open

(*This possibility of a DoS is the reason we rank this "low" instead of "lowest".*)	

Lowest/Style/Info/Suggestions

Description	Status
There is replication of similar code elements ● Several safe math libraries: SafeMath (OZ), DSMath (DappSys), CarefulMath (Compound) ● Low-level function signatures that are used in multiple places, such as `bytes4(keccak256("execute(address,bytes)")` (used twice in auth/ProxyPermission.sol) should be extracted into a constant field, to prevent programmer errors (e.g. typo in the signature). ● `getDecimals` and `sliceUint` are implemented in both exchangeV3/DFSPrices.sol and exchangeV3/DFSExchangeCore.sol We suggest that the code is refactored such that a single implementation of each method is maintained and used.	Open
SafeERC20 library: old version used, there is the issue with approves over non-zero current allowances. Shouldn't affect the code since all approvals are to trusted entities.	Open
ILendingPool is used to capture the interface of several different contracts. Since they are trusted, this should be fine. However it is a questionable practice and one can certainly imagine scenarios where this violation of type information is error prone. E.g., the author trusts one provider because they have inspected certain functions they call. But they cast the provider's address to an abstract contract type that also allows calling more functions. The extra calls may be performed by accident (or they could be forced through an attack scenario).	Open
Use of a floating pragma: The floating pragma `pragma solidity ^0.6.0;` is used allowing the contracts to be compiled with any `0.6.x` version of the Solidity compiler. Floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment.	Open
The contracts were compiled with the Solidity compiler `v0.6.12` which has some known minor issues (but relatively few, compared to earlier versions). We have reviewed the issues and do not believe them to affect the contract. More specifically,	Open

at the time of writing, there are 3 known compiler bugs associated with the Solidity compiler `v0.6.12:` ● Copying an empty `bytes` or `string` array from memory to storage can cause data corruption. (We couldn't find bytes arrays in storage.) ● Direct assignments of storage arrays with an element size <= 16 bytes (more than one values fit in one 32 byte word) are not correctly cleared if the length of the newly assigned value is smaller than the length of the previous one. (No such array is ever stored.) ● Redefinitions of free functions do not result in an error. (No free functions found in the code.)	

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 status 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.