Dedaub

Security Technology for Smart Contracts

DeFi Saver V3

Smart Contract Security Assessment

Date: Mar. 30, 2021

Abstract

Dedaub was commissioned to perform a security audit on the DeFi Saver V3 architecture, the new version of contracts behind the DeFi Saver dashboard.

The audit was performed on repo https://github.com/DecenterApps/defisaver-v3-contracts at commit `cb29669a`.

Four auditors worked on the task over the course of seven working days. We reviewed the code in significant depth, assessed the economics of the protocol and processed it through automated tools. 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 large, approaching 10KLoC. However several components are out of scope (StrategyExecutor.sol, Subscriptions.sol, SubscriptionProxy.sol, BotAuth.sol, ProxyAuth.sol, and several utility contracts). The audited code base is around 5KLoC with some amount of repetition.

The audit focused on security, establishing the overall security model and its robustness, and also crypto-economic issues. Functional correctness (e.g., that the calculations are correct) was a secondary priority. Functional correctness relative to low-level calculations (including units and scaling) is generally most effectively done through thorough testing rather than human auditing.

Since the contracts are not deployed, we cannot fully comment on centralization elements. In some contracts that have been deployed, we see an EOA as owner. In previous DeFi Saver deployments we encountered multisig accounts in key owner roles, and we expect the same practice to be followed in this deployment. The owner has significant authority over user funds, therefore users should trust the parties playing the owner role. Attack-by-owner vectors have not been considered in this audit, but it is clear that the effects can be dramatic (e.g., the owner can change registered contracts with specific identities that the user is interacting with).

Revisions post-audited-commit were inspected to the extent that they addressed issues identified in this report. The development team should be aware that such revisions are audited with the auditor having

less context than during the original code audit. Therefore post-audited-commit revisions are expected to be made with extreme care, relative to functionality unrelated to the flagged issue that the revision intends to address.

Security Model

The core security model of the protocol is solid. Every user manages their funds through a DSProxy (smart wallet). All actions that the protocol implements are stateless and execute in the storage context of the user's DSProxy. Therefore, the potential for an external attack to the user's funds seems limited.

We have not found a violation of this principle in code that is in-scope for the audit. Greater potential for violation exists in the automation code, which is not yet finalized. Ideally, the code should have a general checker of the consistency of action compositions, once such automation is in place. For instance, there is currently no checking of type-compatibility of one action's output with another's input, even though they may be linked via `paramMapping`.

In code that is in scope of the audit, the attack surface is small. **As a result, our audit report primarily consists of recommendations, not vulnerability reports**. These represent "defensive coding" practices in order to avoid escalating user mistakes (e.g., wrong parameters or misconfiguration) or future vulnerabilities, once automation is introduced. Since the precondition for any reported issue is hypothetical (e.g., "this leak of funds can happen if this parameter is set to X by mistake") we arbitrarily rank recommendations as low-severity issues and advisory issues. However, both categories can contain threats, under the assumption of a mistake in the external parameters to the audited code.

Vulnerabilities and 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: 1) User or system funds can be lost when third party systems misbehave. 2) DoS, under specific conditions. 3) Part of the functionality becomes unusable due to programming error.
Low	Examples: 1) Breaking important system invariants, but without apparent consequences. 2) Buggy functionality for trusted users where a workaround exists. 3) Security issues which may manifest when the system evolves.

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

Critical Severity

[No critical severity issues]

High Severity

Description	Status

Privileged `AAVEv2` flash loan tasks, initiated in `FLAaveV2._flAaveV2` and `FLDyDx._flDyDx` respectively, can reenter into the same flashloan tasks to initiate other tasks to clear all user's funds from the DSProxy. A strict precondition for this attack is for the user to interact with a malicious token/LP during his/her actions, so that the attacker can execute arbitrary code and reenter the (temporarily) privileged flash loan action. (Notably the attack does not work with DyDx flash loans: the `Solo::operate` function is non-reenterable. But it does work with Aave flash loans.)
 	
For instance, in AAVEv2 flash loans, an attacker can achieve this by having some user or recipe interact with a Uniswap LP containing a token engineered to reenter into `FLAaveV2.executeAction(...)`. Reentering within the timeframe in which the flash loan task has permission is the precondition for the attack.
 	
The permitted timeframe of flashloan tasks (in which these can execute actions on the DSProxy) is not automatically limited to a single flash loan if reentrant calls are considered:
`function _parseFLAndExecute(...) {`
`  givePermission(_flActionAddr);  ...`
`  ActionBase(_flActionAddr).executeAction(`
`  _currTask.callData[0],`
`  _currTask.subData[0],`
`  _currTask.paramMapping[0],`
`  _returnValues`
`  ); // Dedaub: reentrant call within this`
`  ...`
`  removePermission(_flActionAddr);`
`}`
 	
Once a rogue token on some Uniswap LP reenters into `FLAaveV2._flAaveV2`, it can invoke a second flash loan (e.g., for zero funds) that execute other tasks offered by the system to drain funds.
	Resolved (reentrancy guard added)

Solution: add reentrancy guards to `FLAaveV2.executeOperation`. For defensiveness it may be a good idea to add them to all flash-loan callbacks. Alternatively, the current architecture of flash loan actions can change. They can be delegatecalled-into, just like regular actions (instead of being regular-called, as they are now). Then the flash loan can take place and the code in `executeOperation` will be: `require(_initiator == proxy, ERR_SAME_CALLER);` Where `proxy` is read from the `_params` of the `executeOperation` (exactly as it is in the current code). In this way, an attacker can initiate another flash loan action, but all they can do is call `execute` on themselves. This is a more elegant approach, much in the spirit of the current security model.	

Medium Severity

[No medium severity issues]

Low Severity

Description	Status
The `TokenUtils::pullTokens` function returns normally instead of reverting for specific parameters (`from == address(0)`, or `from == this`, or `token == ETH_ADDR`, or `amount == 0`). We recommend reverting under some such conditions, especially when the amount to be pulled is greater than the balance. Otherwise clients seem to expect this invariant to hold and it is violated. Specific instances: • if `_from == this`, `DFSBuy::_dfsBuy` has a subtraction that underflows because it expects that the token balance is greater than the pulled amount (which was never transferred, but `pullTokens` silently succeeded) • `McdPayback::_mcdPayback` (and similarly for other actions, e.g., `AavePayback`, `McdSupply`, `UniSupply`, `UniWithdraw`) will give approval for `_amount` tokens after a `pullTokens` of that amount. However, if `from == this`, the amount can be anything and `pullTokens` succeeds.	**Dismissed (Partly resolved: function renamed to make intent clearer, callers will adapt, mcdPayback has no issue due to later logic)**

Such issues are minor on their own, but can be abused by a motivated attacker to turn other weaknesses into much more serious threats, e.g., by finding a vector to get large approvals.	
`TokenUtils::withdrawTokens` has similar issues. They appear somewhat harder to escalate, but, for instance, `UnwrapEth::_unwrapEth`, when called with `_to == 0` will call `withdrawTokens` which will silently succeed while leaving the ETH in the proxy. (Which is better than burning it, but worse than reverting, since this parameter is clearly a mistake.) Relatedly, the comment in SendToken::_sendToken reads `@param _to Where the tokens are sent, can't be the proxy or 0x0` However this "can't" is rather "shouldn't". Nothing prevents these values.	**Dismissed**
`TokenUtils::pullTokens` also adjusts the pulled amount in ways that not all of its callers seem to expect. If `_amount == uint256.max`, "infinite" ERC20 approval may leak in `CompPayback::_payback`, `Unisupply::_uniSupply`, `Uniwithdraw::_uniwithdraw`. (Some of these transactions may revert after the approval when `_amount` is `uint256.max`, some won't.) Refactoring the code, such that the approval is below a check on `_amount` (as actions other than the above three do) would prevent this from happening. Even better, `pullTokens` should communicate what it actually did to its callers, who can react accordingly. Approve leaks can also happen in the following points: • `KyberWrapper::buy`, if `srcAmount > srcAmountAfter` • `ScpWrapper::takeOrder` and `ZeroXWrapper::takeOrder`, if the call to the exchange fails, or not all of `_exData.srcAmount` is expended entirely. • `UniswapWrapperV3::buy`, if the `_srcAddr` spent to buy `_destAmount` tokens is less than `_srcAddr.getBalance(address(this))` (the allowance amount). However, this should not matter if the invariant is maintained that exchange wrappers always transfer out all the holdings at the end of a transaction, as they currently seem to do.	**Resolved**
In `DydxFlashLoanBase::_getMarketIdFromTokenAddress`, if the requested token is not found in `solo`'s markets, then the function will return 0 and the caller would have no idea that an "error" had occurred.	**Resolved**

This might lead to some unexpected behavior, as the caller would continue with the assumption that `solo` market #0 is the one he was searching for. We suggest either reverting, or returning an additional value to indicate the failure of the `DydxFlashLoanBase::_getMarketIdFromTokenAddress` operation.	
The `executeActionDirect` function is `payable`. However, it can accept ETH only for specific actions and parameters. It would be a good practice to reject `msg.value` (i.e., revert) when it makes no sense (e.g., when the token is not WETH, after parsing of parameters). This would be enforced in specific actions, e.g., `CompBorrow`. Actions that should never accept ETH (e.g., `CompClaim`, `DFSBuy`, many more) should reject it in their `executeActionDirect`.	**Dismissed**
In `SumInputs::_sumInputs` the two operands are added with normal addition. This might overflow and cause some unexpected behavior during task execution. Using the SafeMath library can help mitigate this issue.	**Resolved**
In `CompPayback::_payback()` the approval `tokenAddr.approveToken(_cTokenAddr, _amount);` should only be performed if `(tokenAddr != TokenUtils.WETH_ADDR)`. Otherwise it is an approval leak on the CToken.	**Resolved**
In `CompHelper::enterMarket()` and `CompHelper::exitMarket()` returns (indicating success) for calls to `Comptroller::enterMarket()` and `Comptroller::exitMarket()` are ignored. We suggest both of these calls should revert if the returned values are not 0.	**Resolved**
Does `TaskExecutor` really need to pass its entire balance to every action in `TaskExecutor::_executeAction`? It seems easy enough to distinguish between the `msg.value` sent to the most recent public entry point from value that might have been left over. Even if there should not be value left over, `TaskExecutor` is sensitive since it is shared among proxies. Any confusion or misconfiguration can be exploited by other users. A clean ETH-rescue approach in `TaskExecutor` may be desirable.	**Resolved**

`OasisTraderWrapper` (`0x4c9B55f2083629A1F7aDa257ae984E03096eCD25`) is whitelisted in `SaverExchangeRegistry`, however there is no Oasis wrapper implementation in V3 (like `UniswapWrapperV3`, `KyberWrapperV3` etc). If you are sunsetting this integration, then the relevant entry should be removed from `SaverExchangeRegistry`.	**Resolved**

In KyberWrapperV3::buy there is an asymmetry between the handling of ETH and ERC20 tokens. This can be seen in the following snippet:
`if (_srcAddr != KYBER_ETH_ADDRESS) {`
`srcAmount = srcToken.balanceOf(address(this));`
`} else {`
`srcAmount = msg.value;`
`}`
This also has the following side-effect: if `address(this)` already holds some ETH (other than msg.value), then the return value may underflow if `srcAmountAfter` > `srcAmount` (=`msg.value` in our case). The extra ETH seems to be safely transferred out, but an underflow value is not welcome either way.
	Resolved

Other/Advisory Issues

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

Description	Status
The `_subData` and `_returnValues` arrays in `executeAction()` and `_parseParamAddr()` could be declared as calldata arrays. This would prevent them from being copied to memory on each internal transaction corresponding to an action call. Instead their values would only be loaded in `_parseParamAddr()` if needed.	**Optimization**
In `McdWithdraw::_mcdWithdraw()` the block `if (IJoin(_joinAddr).dec() != 18) {frobAmount = convertTo18(_joinAddr, _amount);}` Results in a redundant call to the `.dec()` function if the decimals are less than 18 (the second one in `McdHelper::convertTo18()`). Calling the	**Resolved**

function without a condition would save gas in this case.	
In `SendToken::_sendToken`, the `_amount` check is redundant as it will be repeated in `TokenUtils.withdrawTokens`. Simply returning `_tokenAddr.withdrawTokens(_to, _amount)` would help save some gas. In a similar fashion, the same check in `PullToken::_pullToken` could be removed. However, the refactored version will have a slightly different semantics, as `TokenUtils.pullTokens` has additional logic that takes the minimum of `allowance` and `balance` as the amount to be pulled.	**Resolved**
Spell check: suppy -> supply, repacle -> replace, auhtorized, beacuse, RECIPIE, recieve, retreive, skipes, wraper, ... Does not affect functionality, but a 5min effort can improve impressions.	**Resolved**
The constant `CALLBACK_SELECTOR = 0xd6741b9e` in FLAaveV2 and FLDyDx is cryptic and, thus, error-prone. It could be replaced with a human-readable value, i.e., `keccak256("_executeActionsFromFL((string,bytes[][],bytes[][],bytes32[],uint8[][]),bytes32)")`.	**Dismissed**
In Compound actions it would be a good practice (less error-prone) to name CToken variable addresses "`ctoken*`" instead of just "`token*`". E.g., in `executeActionDirect`.	**Resolved**
In `CompWithdraw::executeActionDirect` the `from` parameter should be named `to`.	**Resolved**
The following code in `McdHelper::getTokenFromJoin` seems extraneous, already covered by the rest of the logic. `if (isEthJoinAddr(_joinAddr)) return TokenUtils.WETH_ADDR;`	**Resolved**

Is the receive/fallback function in `DFSExchangeCore` only used for `DFSExchange`? (Other contracts that inherit from `DFSExchangeCore` are actions, so they don't need one.) If so, could it not move to `DFSExchange`? Generally it is not clear to us how `DFSExchange` is to be deployed. It is stateless but not an action.	**Dismissed (contract will not be used)**
Use of a floating pragma: The floating pragma `pragma solidity ^0.7.0;` is used allowing the contracts to be compiled with any `0.7.x` version of the Solidity compiler. Although the differences between these versions are small, floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment.	**Resolved**
The contracts were compiled with the Solidity compiler `v0.7.6` which, at the time of writing, has one known minor issue. We have reviewed the issue and do not believe it to affect the contracts. More specifically there is 1 known compiler bug associated with the Solidity compiler `v0.7.6`: • Wrong (cached) value for consecutive keccak hashes produced in inline assembly, hashing memory contents starting from the same index but with different lengths.	**Info**

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.