Dedaub

Security Technology for Smart Contracts

Maple Finance - Maple Core

Smart Contract Security Assessment

Date: Mar. 12, 2021

Abstract

Dedaub was commissioned to perform a security audit on Maple Finance's core smart contracts (`maple-core`). The audit was performed on commit `05ef95f` with revisions subsequently checked against the issues identified.

Four auditors worked on the task over the course of 8 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 medium-size, at around 3.5KLoC (non-test, non-interface files). However, the economic mechanisms behind the Maple Core protocol are complicated. We also account for protocol composability issues and the economic risks these bring in this audit.

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 (e.g., units) is generally most effectively done through thorough testing.

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.

Trust Model/Centralization Elements

[*This section is included for context, although its contents should already be known to the commissioner of an audit.*]
There are significant centralization elements, but these are in accordance with a service that offers under-collateralized loans. Much of the trust in the protocol is derived in the off-chain world. E.g., users need to trust the protocol owner account and the vetting of loans performed by the pool delegate. Borrowers are trusted because of real-world reputation and potential consequences. A notable centralization element, however, is that, in this version of the code, there is significant trust in the pool delegates. Pool delegates can subvert many of the checks and balances if they act maliciously. For instance, there are at least two different flash-loan-based attacks outlined below (present in the final, revised version of the code) that pool delegates can perform.

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 user's funds.
High	Third party attackers may block the system or cause the system or users to lose funds.
Medium	Examples: 1) User or system funds can be lost when third party systems misbehave. 2) DoS, under specific conditions.
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

Description	Status
An attacker can cause the pool to burn BPTs at highly unfavorable rates, upon handling a default. Upon a loan default, a pool will claim the available loan funds--function `claim` in Pool. Anyone can call this function and it will eventually invoke `handleDefault` in PoolLib. This function burns BPT on Balancer using `exitswapExternAmountOut` and passing it *infinity* as the BPT number to burn (i.e., as many as needed to recover a certain amount of `liquidityAsset`). But Balancer is an AMM, so its pricing can be manipulated inside a single transaction via flash loan. Manipulation of the pool can yield profit for the attacker, up to the value of BPTs being burned.	**Resolved: Functionality limited to pool delegates, in revised code. Pool delegates considered trusted actors.**

High Severity

Description	Status
Loan liquidation can be forced to suffer max slippage (10%). An attacker can make the liquidation of collateral for a defaulting loan to be done at prices 10% lower, via flash loan manipulation of the Uniswap pool. Specifically, either anyone (after the extended grace period) or an LP provider (after the grace period but during the extended grace period) can call `triggerDefault` on a loan. This will eventually call `triggerDefault` in LoanLib, which will swap the collateral for the loan asset on Uniswap. The attacker can manipulate the Uniswap pool beforehand (e.g., via flash loan) so that it is tilted and offers 10% lower prices for the collateral asset. 10% slippage is accepted by the code (since the difference in exchange rates between Uniswap and Chainlink oracles can be significant). However, the scenario described is a predictable, repeatable attack.	**Resolved: Functionality limited to LPs with a large percentage of loan FDTs, in revised code. (Parameter needs to be set with care.)**
A pool delegate can subvert the check that a pool is well-staked (performed in `Pool.finalize`). The pool delegate can take a flash loan and perform a Balancer manipulation attack, so that the check `getInitialStakeRequirements` in `finalize`, which calls the Balancer pool, succeeds. The value of the BPTs is checked against real monetary values, so the caller can make BPTs temporarily appear very expensive.	**Dismissed: Pool delegates considered trusted actors in current version of the protocol.**

Medium Severity

[No medium severity issues]

Low Severity

Description	Status
Some ERC20 operations (`transfer`, `transferFrom` and `approve`) will fail, if the underlying token is not fully compliant with the ERC20 standard, as it is possible that they do not return a boolean value for the aforementioned operations to indicate the success of the call (most notable exception is USDT). It is recommended that the OpenZeppelin SafeERC20 wrappers be used for these operations, to ensure compatibility with such tokens.	**Resolved**
`MapleGlobals.isValidCalc` should be defined as a view function.	**Resolved**
No need for public `withdrawFundsOnBehalf` in FDT (really, Loan).	**Resolved**
Several contract fields are only set during the contracts' creation. They could use the immutable modifier to reduce the gas costs of their uses. • **`PremiumCalc:`** `premiumBips` • **`LateFeeCalc:`** `feeBips` • **`FDT:`** `fundsToken` • **`MapleTreasury:`** `mpl, fundsToken, uniswapRouter` • **`MapleGlobals:`** `mpl` • **`Pool:`** `stakingFee, delegateFee` • **`Loan:`** `apr, termDays, paymentIntervalSeconds, collateralRatio, fundingPeriodSeconds, createdAt` • **`ChainlinkOracle:`** `assetAddress`	**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
`StakeLocker.canUnstake()` relies on low-level constants of Pool. We suggest refactoring the code to hide these, in order to improve readability.	**Resolved**
The `depositDate` calculation has mildly surprising consequences, especially apparent in later stages of the protocol, when `lockupPeriod < penaltyDelay`. The linear re-balancing of `depositDate` when funds are added loses information and may serve as a counter-incentive to deposits. Example: • `lockupPeriod` is 0, `penaltyDelay` is 1yr • user deposited 10K USDC, a year ago. User's `depositDate` is *now - 1yr* • user deposits another 10K USDC. User's `depositDate` becomes *now - 6months* • user finds themselves in need of some cash. Withdraws 5K USDC and pays penalty as if all 20K were deposited 6 months ago, although the user had 10K (i.e., more than what they are trying to withdraw) deposited a year ago.	**Dismissed**
There are a few instances of local variables that are not being explicitly initialized (assumed to be zero when first used): • `penalty` in `PoolLib.calcWithdrawPenalty` • Return value in `BasicFDT._updateFundsTokenBalance` • `losses` in `ExtendedFTD.recognizeLosses` • Return value in `ExtendedFDT._updateLossesBalance`	**Dismissed**

The contracts were compiled with the Solidity compiler `v0.6.11` 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, at the time of writing, there are 2 known compiler bugs associated with the Solidity compiler `v0.6.11`: • Copying an empty `bytes` or `string` array from memory to storage can cause data corruption: ○ This could affect the name and symbol fields of several FDT tokens (without leading to any exploits). However the fact that these contracts are created using the appropriate factories with generated names eliminates this issue. • 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.)	**Resolved**
In `LoanFactory.isValidGovernor`, `LoanFactory.isValidGovernorOrAdmin`, and `LoanFactory._whenProtocolNotPaused` the revert messages refer to `PoolFactory`.	**Info**
In `PoolFactory._whenProtocolNotPaused` and `LoanFactory._whenProtocolNotPaused` the natspec comments are wrong.	**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.