Struct Finance

Struct-Core Contracts

Smart Contract Security Assessment

May 30, 2022

ABSTRACT

Dedaub was commissioned to perform an audit on the struct-core contracts on branch audit/dedaub-may-2022 and commit hash c2277351bab3d1629879d4eeacd61d4700e184a7. The audited contracts list is the following:

products/FEYProduct.sol
products/FEYProductFactory.sol
common/GACManaged.sol
common/GlobalAccessControl.sol
common/StructPriceOracle.sol
types/DataTypes.sol
logic/Validation.sol
libraries/helpers/Helpers.sol
libraries/helpers/Constants.sol
tokenization/StructERC1155.sol
tokenization/StructSPToken.sol
adapters/TraderJoeLpAdapter

Some external contracts in close interaction with the system were also inspected, in essence Trader Joe's Masterchef and Router contracts and Chainlink's oracle contracts on Avalanche.

Two auditors worked on the codebase over two weeks.

Setting & Caveats

The audited codebase is of moderate size of ~3.5KLoC,

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. 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.

Architecture and High-Level Recommendations

Struct Finance aims to provide structured products in the Decentralized Finance ecosystem, in essence “products that combine baskets of different derivatives, such as interest rate products and options, to create a vehicle that can be tailored to a wide array of different risk profiles”.

The audited codebase includes two main contracts, FEYProductFactory and FEYProduct. Contract FEYProduct (Fixed and Enhanced Yield Product) includes all the logic for swapping the yield and risk of *two sets of investors* over a *specific interest rate product*. The two sets of investors are called tranches and are those of Fixed and Enhanced Yield, which can also be expressed as Low and High Risk. The interest rate product is that of liquidity providing (LP) in a Trader Joe’s AMM pool. Once a FEYProduct is created – meaning that a new contract is deployed and is configured regarding the chosen AMM pool, the fixed yield rate, the duration of the investment period, etc. – users can deposit amounts in any of the two pool’s tokens. Each pool’s token represents a tranche in the Struct investment environment, so by depositing an amount of a token the investors automatically choose their risk position.
After the deposit period is over, the invested amounts are ready to be deposited into the pool. The pool requires equal value of the two tokens to be deposited as liquidity, so in case of unbalanced tranches some amount is first swapped from the token in excess to the one in shortage. Swap transactions are required by the protocol in some other cases too, for example when harvesting the LP token staking-rewards and redepositing them as liquidity to the pool. Though the design has taken measures against sandwich attacks, we have outlined some thoughts regarding such attacks in case of large swapped amounts in issue L2.

Contract FEYProductFactory allows whitelisted users to configure and create their own FEY products. Two kinds of fees are applied: performance and management fees. Performance fees are applied only in case of a profitable investment period and is a rate on the total profits. The protocol keeps a percentage of the total fees accrued, while the rest of it is transferred to the FEY product creator.

A number of sensitive parameters of the protocol remain configurable - only by governance - after the project's deployment and initialization. Such parameters are, for example, the maximum allowed capacity deviation between the two tranches, performance fee rate and management fee rate. At the time of writing, the Struct Finance team is focused on developing the core functionality of the system and the governance entity is currently a simple multisignature. The team plans to add timelock functionality to the current scheme and, furthermore, upgrade the governance procedures to a more decentralized solution in a later stage of the project.

The codebase is generally well-written and is accompanied by a large test suite.

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: -User or system funds can be lost when third party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming errors.
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”, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Typographic errors could cause severe faulty functionality	**RESOLVED**
In FEYProductFactory::_deployProduct the following lines are dead code:		

```
_configTrancheSr.spTokenId == latestSpTokenId + 1;
_configTrancheJr.spTokenId == latestSpTokenId + 2;

_configTrancheSr.decimals == _configTrancheSr.tokenAddress.decimals();
_configTrancheJr.decimals == _configTrancheJr.tokenAddress.decimals();
```

They seem to be typographic errors - the above dead statements should be assignments, meaning "==" should be "=", so that the system assigns the spTokenId values of the tranches. The tests did not recognize these errors because, within the tests, these 4 tranches' configuration parameters are manually assigned to the correct values.
This could completely block the system's functionality, since tranche's token id could be assigned to wrong values, given as parameters by an attacker, affecting not only the problematically initialized tranche itself but the whole system.

The Struct Finance team immediately confirmed and resolved this issue.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Legacy code may cancel part of the rewards	**RESOLVED**

In `Helpers::_swapAndAddLiquidity` the accrued reward amounts are supposed to be swapped against senior/junior pool tokens and added into the pool as liquidity. Amounts `_reward1ToSwap` and `_reward2ToSwap`, which are passed as parameters, do not hold the total contract's balance of the corresponding reward token but the pure value of reward amounts accrued.
However, code is written like these variables hold the whole contract's balance, resulting in excluding part of the rewards to be swapped and added as liquidity into the pool:

```
function _swapAndAddLiquidity(
        address _tokenSr,
        address _tokenJr,
        uint256 _srTokenExcess,
        uint256 _jrTokenExcess,
        DataTypes.Addresses calldata _addresses,
        DataTypes.SwapPath calldata _path,
        bool _isDualReward,
        uint256 _reward1ToSwap,
        uint256 _reward2ToSwap
    ) public {
    // ...
    if (address(_addresses.reward1) == _tokenSr) {
        // Dedaub: _reward1ToSwap holds the pure reward1 accrued amount
        if (_reward1ToSwap > _srTokenExcess) _reward1ToSwap -= _srTokenExcess;
        // ...
    } else if (address(_addresses.reward1) == _tokenJr) {
        if (_reward1ToSwap > _jrTokenExcess) _reward1ToSwap -= _jrTokenExcess;
        // ...
    } else { // ...
        }
    }

    if (_isDualReward) {
        // Dedaub: _reward1ToSwap holds the pure reward2 accrued amount
        if (address(_addresses.reward2) == _tokenSr) {
            if (_reward2ToSwap > _srTokenExcess) _reward2ToSwap -= _srTokenExcess;
            // ...
        } else if (address(_addresses.reward2) == _tokenJr) {
            if (_reward2ToSwap > _jrTokenExcess) _reward2ToSwap -= _jrTokenExcess;
        }
    }
}
```

The Struct Finance team immediately confirmed and resolved this issue by removing the legacy lines of code.

LOW SEVERITY:

ID	Description	STATUS
L1	Use of safeIncreaseAllowance instead of safeApprove	**OPEN**

In many cases the system needs to approve amounts to external contracts (i.e., contracts of Joe Trader's system). This is implemented using the increaseAllowance functionality of ERC20 tokens, for example:

```
function _increaseAllowanceAndAddLiquidty(
  IJoeRouter router,
  address _tokenSr,
  address _tokenJr,
  uint256 _srTokenExcess,
  uint256 _jrTokenExcess
) private {
  // ...
  IERC20Metadata(_tokenSr).safeIncreaseAllowance(address(router), _srToAdd);
  IERC20Metadata(_tokenJr).safeIncreaseAllowance(address(router), _jrToAdd);
  // ...
}
```

We don't see any reason to use `safeIncreaseAllowance` instead of the simpler `safeApprove`. On the contrary, it would be a good practice to use `safeApprove` in order to ensure no remaining approvals are aggregated in the external contracts, which would mean linking the approved amounts to any potential security risk of those contracts.

ID	Description	STATUS
L2	Swapping investment amounts could be susceptible to sandwich attacks	**OPEN**

There are several of cases where a swap transaction needs to take place within an investment period:
1. After the protocol's deposit phase is over, the tranches' investment amounts are adjusted and added as liquidity in the corresponding AMM pool. Adjustment is needed

in case of value deviation between the deposits of the two tranches, since liquidity providing requires an equal value of the two pool's assets to be deposited. This is handled by swapping part of one tranche's excessive investments into the other tranche's asset, so as to calibrate the value held in the two tranches. This adjustment step is expected to take place in most investment periods, since users can freely deposit any amount in a tranche without restrictions (entry point is `FEYProduct::invest`).
2. When an investment period is over, the product's investments are withdrawn from the LP and the funds are properly allocated to the senior and junior tranches (entry point is `FEYProduct::removeFundsFromLP`).
3. Any time during the INVESTED state of an investment period, rewards can be claimed and are swapped against the pool's assets and added as liquidity (entry point is `FEYProduct::harvestAndRecompound`).

The following precautions have been taken to protect these swaps from sandwich attacks:
a. onlyEOAOrRole modifier, where Role denotes a whitelisted entity. The check for EOA account is not effective and can be worked around be an attacker. Consider, for example, an attacker who submits a bundle of transactions to be mined to MEV nodes.
b. The AMM's price is checked against the price calculated using Chainlink's oracles and the deviation is required to lie within a specific range.

Precaution a. cannot effectively protect against sandwich attacks, since any adversary can still work around the onlyEOA restriction.
Precaution b. does offer a level of protection, by bounding an attack's risk and consequences. The current implementation suggests that if the AMM's price does not deviate *much* from the oracle's price, then it is safe for the swap to be performed and the slippage protection is applied in respect to the AMM's price (via setting the `minAmountOut` variable). However, if the swapped amount is fairly large then it could still be profitable for an adversary to perform the attack: she can tilt the pool as much as possible, up to the point that the AMM's price stays just within the accepted deviation range, let the victim swap transaction be executed and then untilt the pool to take the profit from the bad swap.

For this attack to be profitable the pool should be of moderate or small size, so as to be tiltable, while the swapped amount should be fairly large so as to recover for: the transaction fees, the swap fees and any other external fees needed (e.g. MEV bribes).
In swap cases 1&2 the swapped amounts concern investment amounts and could be expected to be large, even though the two tranches' deposits are not allowed to deviate arbitrarily much (maximum deviation is set to 5%). A stricter precaution for these cases would be to apply the slippage protection directly on the oracle's price, instead of that of the AMM's. Also in case of extremely large amounts (also in comparison to the pool's size), another solution would be to incrementally perform a swap in a number of swap transactions executed in separate blocks, so as to lower the swapped amounts and make the attack unprofitable.
In swap case 3 the swap amount concerns accrued rewards and are not expected to be significantly large if the `harvestAndRecompound` function is called at a regular basis. For this reason, we suggest having a keeper contract taking care that this function is called often enough.

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	Gas optimizations	**OPEN**

1. There are several places where a function's local variable reads from storage and is declared storage instead of memory, though it is only read and never written. Similarly for some cases of function parameters.

```
/// Helpers::getInvestedAndExcess
uint256[] storage prefixSums_ = investor.depositSums;
```

```
/// Helpers::chargeFees
function chargeFees(
        uint256 _tokensDepositedSr,
        uint256 _tokensAtMaturitySr,
        uint256 _tokensDepositedJr,
        uint256 _tokensAtMaturityJr,
        // Dedaub: declare the next two as memory instead of storage
        DataTypes.TrancheConfig storage _trancheConfigSr,
        DataTypes.TrancheConfig storage _trancheConfigJr,
        uint256 _creatorFee,
        // Dedaub: declare as memory instead of storage
        DataTypes.ProductConfig storage _productConfig
    ) external returns (uint256, uint256) { ... }

/// FEYProduct.sol
function removeFundsFromLP()
    external
    override
    nonReentrant
    onlyEOAOrRoles(_msgSender(), gac.keeperWhitelistedRoles())
{
    // ...

    // Dedaub: declare as memory instead of storage
    DataTypes.TrancheConfig storage _trancheConfigSr =
trancheConfig[DataTypes.Tranche.Senior];
    DataTypes.TrancheConfig storage _trancheConfigJr =
trancheConfig[DataTypes.Tranche.Junior];
    // ...
}

function _calulateUserShareAndTransfer(DataTypes.Tranche _tranche) private {
    // Dedaub: declare as memory instead of storage
    DataTypes.TrancheInfo storage _trancheInfo = trancheInfo[_tranche];
    DataTypes.TrancheConfig storage _trancheConfig = trancheConfig[_tranche];
   // ...
}

function _chargeFee(
    // Dedaub: declare as memory instead of storage
```

```
    DataTypes.TrancheConfig storage _trancheConfigSr,
    DataTypes.TrancheConfig storage _trancheConfigJr
) private { /// ... }
```

A2	Code simplification	OPEN

1. In `Helpers::getInvestedAndExcess` code can be simplified by calculating excess value only once at the end of the function's body:

```
function getInvestedAndExcess(DataTypes.Investor storage investor, uint256
invested)
    external
    view
    returns (uint256 userInvested, uint256 excess)
{
    // ...

    uint256 prefixSum = prefixSums_[leastUpperBound];
    if (prefixSum == invested) {
        // Not all deposits got in, but there are no partial deposits
        userInvested = investor.userSums[leastUpperBound];
        // Dedaub: same calculation of 'excess' as in the rest of the cases
        excess = investor.userSums[length - 1] - userInvested;
    } else {
        userInvested = leastUpperBound > 0 ? investor.userSums[leastUpperBound -
1] : 0;
        uint256 depositAmount = investor.userSums[leastUpperBound] - userInvested;
        if (prefixSum - depositAmount < invested) {
            userInvested += (depositAmount + invested - prefixSum);
            // Dedaub: same calculation of 'excess' as in the rest of the cases
            excess = investor.userSums[length - 1] - userInvested;
        } else {
            // Dedaub: same as above
            excess = investor.userSums[length - 1] - userInvested;
        }
    }
```

```
}
```

could become:

```
function getInvestedAndExcess(DataTypes.Investor storage investor, uint256
invested)
    external
    view
    returns (uint256 userInvested, uint256 excess)
{
    // ...

    uint256 prefixSum = prefixSums_[leastUpperBound];
    if (prefixSum == invested) {
        // Not all deposits got in, but there are no partial deposits
        userInvested = investor.userSums[leastUpperBound];
    } else {
        userInvested = leastUpperBound > 0 ? investor.userSums[leastUpperBound -
1] : 0;
        uint256 depositAmount = investor.userSums[leastUpperBound] - userInvested;
        if (prefixSum - depositAmount < invested) {
            userInvested += (depositAmount + invested - prefixSum);
        }
    }
    excess = investor.userSums[length - 1] - userInvested;
}
```

2. In Helpers::getTrancheTokenRate getAssetPrice is redundantly called twice for each tranche's token:

```
function getTrancheTokenRate(
    IStructPriceOracle _structPriceOracle,
    address _asset1,
    address _asset2,
    address[] storage _path,
    IJoeRouter _router
)
```

```
    external
    view
    returns (bool, uint256, uint256, uint256)
{
    uint256 _priceAsset1 = getAssetPrice(_structPriceOracle, _asset1);
    uint256 _priceAsset2 = getAssetPrice(_structPriceOracle, _asset2);

    uint256 _chainlinkRate = (
        // Dedaub: already calculated above as _priceAsset1,_priceAsset2
        ((getAssetPrice(_structPriceOracle, _asset1) * 10**18) /
            getAssetPrice(_structPriceOracle, _asset2))
    );
    // ...
}
```

We suggest simplifying the code for improved readability and gas savings.

A3	Unused variables	OPEN

1. In TraderJoeLpAdapter.sol state variable

```
address public product;
```

is unused.

2. In FEYProduct.sol state variable

```
address private productFactory;
```

is assigned during the contract's initialization but never used.

A4	Dead code	OPEN

1. In StructPriceOracle::getAssetPrice there is a redundant sanity check on chainlink's return variables:

```
function getAssetPrice(address asset) public view override returns (uint256) {
    (uint80 roundId, int256 price, , , uint80 answeredInRound) =
assetsSources[asset]
        .latestRoundData();

    // Dedaub: chainlink's 'EAC' version is used, the following check is a no-op
    require(roundId == answeredInRound, "OUTDATED");
    // ...
}
```

The above check is recommended when using Chainlink's 'Flux' aggregators contracts, but on Avalanche a later version (called 'EAC') of aggregator contracts is deployed, which maintains the same signature of function latestRoundData for backwards compatibility but values roundId and answeredId are always identical.
For reference: AggregatorProxy contract, OffchainAggregator contract

2. Data structure `TrancheInfo` contains the field `tokensInvested` which is assigned a value for both tranche types in FEYProduct::invest:

```
trancheInfo[DataTypes.Tranche.Senior].tokensInvested = Helpers.tokenDecimalsToWei(
    trancheConfig[DataTypes.Tranche.Senior].decimals,
    _investedSr
);

trancheInfo[DataTypes.Tranche.Junior].tokensInvested = Helpers.tokenDecimalsToWei(
    trancheConfig[DataTypes.Tranche.Junior].decimals,
    _investedJr
);
```

However, this struct's field is never used in the codebase.

A5	Constant variables	**OPEN**

There are a few constant values that remain hard-coded as follows:
1. In FEYProduct.sol

```
slippage = 30 * 10**3; //3%
```

2. In FEYProductFactory.sol

```
require(_productConfig.fixedRate < 750000, "INVALID_RATE");
```

We suggest assigning these values to constant variables for readability and maintenance.

A6	StructToken rewards are divided equally when tranches are unbalanced.	**OPEN**

In the FEYProduct contract, the number of structToken rewards available for distribution to investors of a boosted product is calculated when the removeFundsFromLP() function is called.

```
if (isBoosted) {

            /// Check structToken balance before
            uint256 _balanceBefore = IERC20Metadata(structToken)
                     .balanceOf(address(this));

            /// Call productBooster contract to indicate that rewards
            /// should stop
            structProductBooster.fundsRemoved(address(this));

            /// Check structToken balance after to record the amount of
            /// struct tokens rewards allocated - structRewards
            /// Rewards are divided by two since there are two tranches
            structRewards =
                (IERC20Metadata(structToken)
                 .balanceOf(address(this)) - _balanceBefore) / 2;
        }
```

This amount of available StructToken rewards is divided by two and stored in the structRewards state variable. This represents the amount of tokens to be distributed to investors in each of the Senior and Junior tranches. Hence investors in the Junior and Senior tranches obtain exactly the same amount of rewards even if the assets invested in the Junior and Senior tranches are unbalanced (because one of the tranches is leveraged). This in turn can lead to an uneven distribution of structToken rewards relative to assets invested.

This issue was discussed with the Struct Finance team who confirmed that this is the intended functioning of the protocol.

We advise disclosure of this protocol behavior to potential investors in the public facing documentation.

A7	Function tokenDecimalsToWei() truncates decimal places	**OPEN**

The FEYProduct contract keeps track of deposited assets by scaling down the assets so that they fit into 18 decimal places. This is achieved through the use of the Helpers::tokenDecimalsToWei() function.

```
function tokenDecimalsToWei(uint256 _decimals, uint256 _amount) public pure
returns (uint256) {
        return (_amount * Constants.WEI) / 10**_decimals;
}
```

When the original number of decimals needs to be recovered in order to send funds to an AMM or return them to a user, the assets are scaled up again using the function Helpers::weiToTokenDecimals().

Now if the asset in question has n > 18 decimals, the function Helpers::tokenDecimalsToWei() becomes lossy and truncates the least significant n - 18 decimal places. This means that upon this conversion, the remaining dust remains trapped in the FEYProduct contract and is never invested or returned to the user.

While the amounts in question may not be significant, we advise disclosure of this behaviour to potential investors in the public facing documentation.

A8	Misleading comment	**OPEN**

There is a misleading comment in the FEYProductFactory contract:

```
/// @dev Address of the Struct price oracle
    IDistributionManager public distributionManager;
```

A9	Compiler known issues	**INFO**

Solidity compiler version v0.8.11 has, at the time of writing, some known bugs. We inspected the code and found that it is not affected by these bugs.

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.