Grappa Finance - Full Collateral Engine report

Minting options engine

Status: not deployed

Author of Smart Contracts: @antoncoding

Prepared by: Delvir0, Independent Security Researcher

Date of completion: 08-03-2023

About Grappa Finance - Full Collateral Engine

Grappa Finance in it's core is an exchange and settlement platform for cash-settled options and physical assets. It's smart contracts enable sellers and buyers a single point of minting and settlement of options. Handling of these options (for sellers) is done via Engines which connect with the Grappa Finance system. The Full Collateral Engine is one of those engine which ensures that sellers deposit up to the full collateral in order to mint an option which then can be sold.

Summary & Scope

`grappafinance/full-collat-engine/src` , including contracts inherited, under commit 42d52

The following contracts were in scope:

- BaseEngine.sol@8e67b0b
- DebitSpread.sol@8e67b0b
- FullMarginEngine.sol@42d52af
- FullMarginLib.sol@42d52af
- FullMarginMath.sol@42d52af
- errors.sol@42d52af
- types.sol@42d52af

Summary of Findings

Identifier	Title	Severity	Status	Commit to fix
[C-01]	Minter can mint options with incorrect collateral asset, leading the protocol to be drained	Critical	Fixed	b62d98d
[M-02]	Other entities interacting with the mint function can be DOSSed	Medium	Fixed	9f931a6
[M-03]	Changing amount of account access is open to a sandwich attack	Medium	Acknowledged	
[L-04]	Assumption on resetting collateral will fail in cases where tokens do not support 0 transfers	Low	Fixed	e3a27c6
[I-05]	Naming convention of longStrike and shortStrike is subjected to perspective	Informational	Acknowledged	

Detailed Findings

[C-01] Minter can mint options with incorrect collateral asset, leading the protocol to be drained

FullMarginEngine.sol

The Engine has a single point of entry for all the possible actions provided:

```
function execute(address _subAccount, ActionArgs[] calldata actions) public override
nonReentrant {
        _assertCallerHasAccess(_subAccount);
        // update the account state and do external calls on the flight
        for (uint256 i; i < actions.length;) {
            if (actions[i].action == ActionType.AddCollateral) _addCollateral(_subAccount,
actions[i].data);
            else if (actions[i].action == ActionType.RemoveCollateral)
_removeCollateral(_subAccount, actions[i].data);
            else if (actions[i].action == ActionType.MintShort) _mintOption(_subAccount,
actions[i].data);
            else if (actions[i].action == ActionType.BurnShort) _burnOption(_subAccount,
actions[i].data);
            else if (actions[i].action == ActionType.MergeOptionToken) _merge(_subAccount,
actions[i].data);
            else if (actions[i].action == ActionType.SplitOptionToken) _split(_subAccount,
actions[i].data);
            else if (actions[i].action == ActionType.SettleAccount) _settle(_subAccount);
            else revert FM_UnsupportedAction();

            unchecked {
                ++i;
            }
        }
        if (!_isAccountAboveWater(_subAccount)) revert BM_AccountUnderwater();
```

When minting or creating an option, each type (CALL, PUT, CALL_SPREAD, PUT_SPREAD) has each own requirement of underlying collateral. This could be either tokens like WETH, WBTC, ARB, UNI or stables. The collateral is used in order to pay the option holder (which has been sold to by the minter) if the option is worth anything at time of expiry. Collateral health is checked at the end of the `execute()` function: `if (!_isAccountAboveWater(_subAccount)) revert BM_AccountUnderwater();` which eventually leads to `getCollateralRequirement()` .\

It is important to note that when calculating the required collateral, the value (which is in 6 decimals) gets converted to the decimal value according to the stored `_account.collateralDecimals` .

The danger in the single point of entry with only a check at the end is it opens the possebility to break the system before the check is performed. An CALL options requires the underlying collateral to be the same asset representing the option. This is due to the fact that the price could increase Infinitely. The opposite is true when it's a PUT option, the max loss would be the price of the asset. Since the asset could drop to 0 and leave it worthless, the underlying needs to be (underlyingPrice * 1USDC).

collateralId and amount is stored in the following struct:

```
struct FullMarginAccount {
    uint256 tokenId;
    uint64 shortAmount;
    uint8 collateralId;
    uint80 collateralAmount;
}
```

The accounting of adding and removing collateral is done in the following way:

```
    function addCollateral(FullMarginAccount storage account, uint8 collateralId, uint80
amount) internal {
        uint80 cacheId = account.collateralId;
        if (cacheId == 0) {
            account.collateralId = collateralId;
        } else {
            if (cacheId != collateralId) revert FM_WrongCollateralId();
        }
        account.collateralAmount += amount;
    }


    function removeCollateral(FullMarginAccount storage account, uint8 collateralId,
uint80 amount) internal {
        if (account.collateralId != collateralId) revert FM_WrongCollateralId();

        uint80 newAmount = account.collateralAmount - amount;
        account.collateralAmount = newAmount;
        if (newAmount == 0) {
            account.collateralId = 0;
        }
    }
```

The problem is that there is no check in place that links the required collateral type of an option token that has been minted with the current collateral of the seller. Meanining after minting we could change the deposited collateral type. Above would be a problem if we would reach a scenario where the seller has an option which requires collateralA of low decimal value while depositing collateralB of a high decimal value.

Here is one of the flows of how this can be achieved:

- A seller uses the `execute()` function to perform 4 actions (given ETH == 2000USDC):
  - Add USDC as collateral (1e6)
  - Mint a PUT option (with very high strike price like 1 million)
  - Remove USDC collateral
  - Add 1e-6 ETH as collateral. Collateral required calculation will be calculated in USDC term: 1 million * 1e6 = 1e12 (this passes the collateral check. At settlement the option now enables to withdraw "1 million - 2000" USDC from the protocol)

Recommendation

There are different ways to prevent this. I would suggest to add a check in `_isAccountAboveWater()` where it is checked if `optionCollateralRequirement == account.collateralId` which is also the most gass efficient way.

Review

This was fixed by adding an additonal check in `FullMarginLib.removeCollateral` which now also checks if `account.tokenId == 0` in order to reset the collateralId. Commit can be found here.

[M-02] Other entities interacting with the mint function can be DOSSed

FullMarginEngine.sol

Each account can have up to 255 sub accounts, where sub-account(0) is the main account, in order to be able to handle different option tokens with each their collateral requirement (inspired by Euler's sub-Accounts). The property's of these are held in the struct `FullMarginAccount` and can be moved to any (sub)account:

```
     * @notice transfer an account to someone else
     * @dev expected to be call by account owner
     * @param _subAccount the id of subaccount to transfer
     * @param _newSubAccount the id of receiving account
     */
    function transferAccount(address _subAccount, address _newSubAccount) external {
        if (!_isPrimaryAccountFor(msg.sender, _subAccount)) revert FM_NoAccess();

        if (!marginAccounts[_newSubAccount].isEmpty()) revert FM_AccountIsNotEmpty();
        marginAccounts[_newSubAccount] = marginAccounts[_subAccount];

        delete marginAccounts[_subAccount];
    }
```

At point of minting an option token, it is required to either have `FullMarginAccount.collateralId == 0` or to the correct underlying collateralId. If both are incorrect, the transactions reverts. Since it's possible to send the `FullMarginAccount` to anyone and it's possible to calculate the 255 addresses of each account, we could prevent any entity from minting by transfering an account with the incorrect collateralId.

Here is a simple flow of how this can be achieved:

- Attacker monitors mempool and sees mint (CALL) transactions for sub-account(1) of a competing seller
- Attacker sees this and transfers an account with `collateralId == USDC` to that sub account
- Since an CALL option requires e.g. ETH, the transaction will fail

The result is a possebility to damage an entity by making their service fail (mint and sell options) during crucial times.

Recommendation

I would like to recommend rethinking the need for the transfer of sub-accounts. Incase it's needed you could work with whitelists where a user is not able to send the details of a sub-account to sub-accounts of an other head-account unless it's whitelisted to do so.

Review

This has been fixed by entirely removing the `FullMarginEngine.transferAccount` thus removing the attack vector. Commit can be found here.

[M-03] Changing amount of account access is open to a sandwich attack

BaseEngine.sol

The BaseEngine contract has the option to allow any other entity to perform actions on behave. This is recorded with an amount of actions which is inputted:

```
    function setAccountAccess(
        address _account,
        uint256 _allowedExecutions
    ) external {
        uint160 maskedId = uint160(msg.sender) | 0xFF;
        allowedExecutionLeft[maskedId][_account] = _allowedExecutions;

        emit AccountAuthorizationUpdate(maskedId, _account, _allowedExecutions);
    }
```

This leads to the typical allowance vulnerability where allowance is set to x amount and increased to a new amount.

Simple example where this could lead to an issue:

- UserA has set UserB to 50 allowedExecutions
- UserA decides to add another 50 allowedExecutions and calls `setAccountAccess( ,100)`
- UserB now had the opportunity to perform 50 allowedExecutions before above call gets mined and another 100 after. This results in a total allowedExecutions of 150.

Recommendation

Reconstruct the function to an increase/ decrease flow instead of setting a x amount. This can be achieved in the same as the decreaseAllowance/ increaseAllowance flow of OZ:
https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#ERC20-decreaseAllowance-address-uint256-

Review

Acknowledged. Given that both parties are trusted, this will be noted in the protocol documentation to create awareness regarding the potential issue.

[L-04] Assumption on resetting collateral will fail in cases where tokens do not support 0 transfers

FullMarginLib.sol

At settlement, theres a possebility of the amount of debt being equal to collateralAmount. If this happens, it is assumed that `account.collateralId` can be changed to an other id in order to be able to mint other options by calling `removeCollateral(0)`

The problem is that calling `removeCollateral(0)` triggers a 0 amount transfer while not all tokens support that.

```
    function settleAtExpiry(FullMarginAccount storage account, int80 payout) internal {
        // clear all debt
        account.tokenId = 0;
        account.shortAmount = 0;

        int256 collateral = int256(uint256(account.collateralAmount));

        // this line should not underflow because collateral should always be enough
        // but keeping the underflow check to make sure
        account.collateralAmount = (collateral - payout).toUint256().toUint80();

        // do not check ending collateral amount (and reset collateral id) because it is
very
        // unlikely the payout is the exact amount in the account
        // if that is the case (collateralAmount = 0), use can use removeCollateral(0) //
@audit-issue incorrect since some tokens do not support 0 transfers
        // to reset the collateral id
    }
```

In this case the specific account would be stuck to the current `collateralId`.

Recommendation

Given the scenario, the following are to have in mind when considering a fix 1. low probablilty of above happening 2. there are 255 remaining accounts to be used. This can be fixed by checking the remaining collateral and resetting the collateralId if it's 0.

Review

This has been fixed by following the same check pattern of `FullMarginLib.removeCollateral`. Since the Engine requires full collateral, if remaining collateral amount == 0 there is no possible scenario that collateralId is being reset while there are any options remaining. Commit can be found here.

[I-05] Naming convention of longStrike and shortStrike is subjected to perspective

Multiple contracts

When dealing with options, the terms short and long strike are in it's core and is adapted in the contracts. When a minter has an CALL option with a long strike at x and sells it, it would be considered an CALL option of **short strike** at x at the perspective of the minter (seller).

The contracts are written in such a way that `longStrike` and `shortStrike` are written in the perspective of the minter *and in the scenario of the option already sold*. Having the logic of the strikes inverted causes a lot of confusion when reading through the logic of the codebase and does not align with Grappa.sol.

Recommendation

Invert the current naming of the strikes so that they represent the values of what's actually being minted instead of how they would be seen in the perspective of the seller. `longStrike -> shortStrike` `shortStrike -> longStrike`

Review

Acknowledged. Given the sensitivity of the change across the whole codebase this will be done in the future with the required amount of time in order not to rush it.