GMX Synthetics

Smart Contract Security Assessment

November 20, 2022

ABSTRACT

Dedaub was commissioned to perform a security audit of the GMX Synthetics Protocol.

GMX is a decentralized spot and perpetual exchange trading platform. The audit covers an upcoming version of the protocol. The major addition to this version of the protocol is a two-step execution method. Users first submit an order for execution, whose details are stored on-chain. Then, keepers, who listen for such transactions, submit token prices for the block containing the order, and a second transaction is sent that executes the order.

This audit report covers the contracts of the repository [gmx-io/gmx-synthetics](), at commit hash `3b38da007ab0e4407cb38c06afb9e35e4cf4cb2c`.

The full audited contract list is the following:

```
contracts
├── bank
│   ├── Bank.sol
│   └── StrictBank.sol
├── data
│   ├── DataStore.sol
│   └── Keys.sol
├── deposit
│   ├── Deposit.sol
│   ├── DepositStore.sol
│   └── DepositUtils.sol
├── eth
│   ├── EthUtils.sol
│   └── IWETH.sol
├── events
│   └── EventEmitter.sol
├── exchange
```

```
│   ├── DepositHandler.sol
│   ├── OrderHandler.sol
│   └── WithdrawalHandler.sol
├── feature
│   └── FeatureUtils.sol
├── fee
│   ├── FeeReceiver.sol
│   └── FeeUtils.sol
├── gas
│   └── GasUtils.sol
├── gov
│   └── Governable.sol
├── market
│   ├── MarketFactory.sol
│   ├── Market.sol
│   ├── MarketStore.sol
│   ├── MarketToken.sol
│   └── MarketUtils.sol
├── nonce
│   └── NonceUtils.sol
├── oracle
│   ├── IPriceFeed.sol
│   ├── OracleModule.sol
│   ├── Oracle.sol
│   ├── OracleStore.sol
│   └── OracleUtils.sol
├── order
│   ├── DecreaseOrderUtils.sol
│   ├── IncreaseOrderUtils.sol
│   ├── Order.sol
│   ├── OrderStore.sol
│   ├── OrderUtils.sol
│   └── SwapOrderUtils.sol
├── position
```

```
│   ├── DecreasePositionUtils.sol
│   ├── IncreasePositionUtils.sol
│   ├── Position.sol
│   ├── PositionStore.sol
│   └── PositionUtils.sol
├── pricing
│   ├── PositionPricingUtils.sol
│   ├── PricingUtils.sol
│   └── SwapPricingUtils.sol
├── reader
│   └── Reader.sol
├── role
│   ├── RoleModule.sol
│   ├── Role.sol
│   └── RoleStore.sol
├── router
│   ├── ExchangeRouter.sol
│   └── Router.sol
├── swap
│   └── SwapUtils.sol
├── utils
│   ├── Array.sol
│   ├── Bits.sol
│   ├── Calc.sol
│   ├── EnumerableValues.sol
│   ├── Null.sol
│   └── Precision.sol
└── withdrawal
      ├── Withdrawal.sol
      ├── WithdrawalStore.sol
      └── WithdrawalUtils.sol
```

SETTING & CAVEATS

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behaviour. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. In order to simplify the consumption of the document, Dedaub 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 a programming error.
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” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

ID	Description	STATUS
C1	Reentrancy vulnerability in cancelOrder	**OPEN**

`OrderHandler::cancelOrder`, which is an external function, is not protected by a reentrancy guard. Moreover, `OrderUtils::cancelOrder` (which performs the actual operation) transfers funds to the user (`orderStore.transferOut`) before updating its state. As a consequence, a malicious adversary could re-enter in `cancelOrder`, and execute an arbitrary number of transfers, effectively draining the contract’s full balance in the corresponding token.

```
function cancelOrder(
    DataStore dataStore,
    EventEmitter eventEmitter,
    OrderStore orderStore,
    bytes32 key,
    address keeper,
```

```
    uint256 startingGas
) internal {
    Order.Props memory order = orderStore.get(key);
    validateNonEmptyOrder(order);

    if (isIncreaseOrder(order.orderType()) || isSwapOrder(order.orderType())) {
        if (order.initialCollateralDeltaAmount() > 0) {
              orderStore.transferOut(
                    EthUtils.weth(dataStore),
                    order.initialCollateralToken(),
                    order.initialCollateralDeltaAmount(),
                    order.account(),
                    order.shouldConvertETH()
              );
        }
    }

    // Dedaub: state changed after the transfer, also idempotent
    orderStore.remove(key, order.account());
```

Note that the main re-entrancy method, namely the `receive` hook of an ETH transfer, is in fact protected by using `payable(receiver).transfer` which limits the gas available to the adversary's `receive` hook. Nevertheless, an ERC20 token transfer is an external contract call and should be assumed to potentially pass the execution to the adversary. For instance, an ERC777 token (which is ERC20 compatible) would implement transfer hooks that could easily be used to perform a reentrancy attack.

Note also that the state update (`orderStore.remove(key, order.account())`) is idempotent, so it can be executed multiple times during a reentrancy attack without causing an error.

To protect against reentrancy we recommend

1. Adding reentrancy guards, and
2. Execute all state updates before external contract calls (such as transfers).

Note that (2) by itself is sufficient, so reentrancy guards could be avoided if gas is an issue. In such a case, however, comments should be added to the code to clearly state

that updates should be executed before external calls, to avoid a vulnerability being reintroduced in future restructuring of the code.

HIGH SEVERITY:

ID	Description	STATUS
H1	Conditional execution of orders using `shouldConvertETH`	**OPEN**

For all order types that involve ETH, a user can set the option `shouldConvertETH =true` to indicate that he wishes to receive ETH instead of WETH. Although convenient, this gives an adversary the opportunity to execute "conditional orders" in a very easy way. The adversary can simply use a smart contract as the receiver of the order, and set a receive function as follows:

```
contract Adversary {
    bool allow_execution = false;
    receive() {
        require(allow_execution);
    }
}
```

Then, in the time period between the order creation and its execution, the adversary can decide whether he wishes the order to succeed or not, and set the allow_execution variable accordingly. If unset, the receive function will revert and the protocol will cancel the order.

The possibility of conditional executions could be exploited in a variety of different scenarios, a concrete example is given at the end of this item.

Note that the use of `payable(receiver).transfer` (in `Bank::_transferOutEth`) does not protect against this attack. The 2300 gas sent by `transfer` are enough for a simple check like the one above. Note also that, although the case of ETH is the simplest to exploit, any tokens that use hooks to allow the receiver to reject transfers (eg ERC777) would enable the same attack. Note also that, if needed, the time period between creation and execution could be increased by simultaneously submitting a large number of orders for tiny amounts (see L2 below).

One way to protect against conditional execution is to employ some manual procedure for recovering the funds in case of a failed execution (for instance, keeping the funds in an escrow account), instead of simply canceling the order. Since a failed execution should not happen under normal conditions, this would not affect the protocol's normal operation.

Concrete example of exploiting conditional executions:
The adversary wants to take advantage of the volatility of ETH at a particular moment, but without any trading risk. Assume that the current price of ETH is 1000 USD, he proceeds as follows:

- He creates a market swap order A to buy ETH at the current price. In this order, he sets `shouldConvertETH = true` and the receive function above that conditionally allows the execution.
- He also creates a limit order B to sell ETH at 1010 USD.

He then monitors the price of ETH before the order's execution:

- If ETH goes down, he does nothing. `allow_execution` is false so order A will fail, and order B will also fail since the price target is not met.
- If ETH goes up, he sets allow_execution = true, which leads to both orders succeeding for a profit of 10 USD / ETH.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Incorrect handling of rebalancing tokens	**OPEN**

`StrictBank::recordTokenIn` computes the number of received tokens by comparing the contract's current balance with the balance at the previous execution. However, this approach could lead to incorrect results in the case of ERC20 tokens with non-standard behavior, for instance:

- Tokens in which balances automatically increase (without any transfers) to include interest.
- Tokens that allow interacting from multiple contract addresses

To be more robust with respect to such types of tokens, we recommend comparing the balance before and after the current incoming transfer, and not between different transactions.

M2	Self-close bad debt attack	**OPEN**

The protocol is susceptible to an attack involving opening a delta neutral position using Sybil accounts, with maximum leverage, on a volatile asset.

Scenario:
1. Attacker controls Alice and Bob
2. Alice opens a large long position with maximum leverage
3. Bob opens a large short position with maximum leverage
4. Market moves
5. Alice liquidates their underwater position, causing bad debt
6. Bob closes their other position, profiting on the slippage

The reason why this attack is possible is that using the current design, it takes multiple blocks for a liquidator to react and by that time their order is executed it is possible that one of the positions is underwater. Secondly, when liquidating, Alice does not suffer a bad price from the slippage incurred in the liquidation but Bob benefits from the

slippage, when closing their position just after. Another factor that contributes towards this attack is that the liquidation penalty is linear, while the price impact advantage is higher-order, making the attack increasingly profitable the larger the positions.

To deter this, the protocol could support (i) partial liquidations for large positions and therefore force the positions to be closed gradually, making the attack non-viable, and, (ii) slippage open-interest calculations used to determine the price of the liquidation.

LOW SEVERITY:

L1	Missing receive function	**OPEN**

OrderStore does not contain a receive function so it cannot receive ETH. However, this is needed by `Bank::_transferOutEth`, which withdraws ETH before sending it to the receiver.

```
function _transferOutEth(address token, uint256 amount, address receiver)
internal {
      require(receiver != address(this), "Bank: invalid receiver");

      IWETH(token).withdraw(amount);
      payable(receiver).transfer(amount);

      _afterTransferOut(token);
}
```

WIthout a receive function any transaction with `shouldConvertETH = true` would fail.

L2	No lower bounds for swaps and positions	**OPEN**

Since there are no lower bounds for the size of a position, someone could in principle create a large number of tiny orders. Such a strategy would cost the adversary only the gas fees and the total amount of the requested positions, which can be as small as he wishes. In this 2-step procedure used in this version of the protocol, such a behavior could potentially create a problem, because the order keepers would have to execute a huge number of orders.
We suggest setting a minimum size for positions and swaps.

L3	Inconsistency in calculating liquidations	**OPEN**

The comment in `PositionUtils::isPositionLiquidatable` indicates that price impact is not used when computing whether a position is liquidatable. However, the price impact is in fact used in the code:

```
// price impact is not factored into the liquidation calculation
// if the user is able to close the position gradually, the impact
// may not be as much as closing the position in one transaction
function isPositionLiquidatable(
      DataStore dataStore,
      Position.Props memory position,
      Market.Props memory market,
      MarketUtils.MarketPrices memory prices
) internal view returns (bool) {
      ...

      int256 priceImpactUsd = PositionPricingUtils.getPriceImpactUsd(...)
      …

      int256 remainingCollateralUsd = collateralUsd.toInt256() +
positionPnlUsd + priceImpactUsd + fees.totalNetCostAmount;
```

On the other hand, when the liquidation is executed, the price impact is not used. The comment in `DecreasePositionUtils::processCollateral` indicates that this is intentional:

```
// the outputAmount does not factor in price impact
// for example, if the market is ETH / USD and if a user uses USDC to long ETH
// if the position is closed in profit or loss, USDC would be sent out from or
// added to the pool without a price impact
// this may unbalance the pool and the user could earn the positive price impact
// through a subsequent action to rebalance the pool
// price impact can be factored in if this is not desirable
```

If this inconsistency is intentional, it should be properly documented in the comments.

Note that, when deciding on a liquidation strategy, you should have in mind the possibility of *cascading* liquidations, namely the possibility that executing a liquidation causes other positions to become liquidatable.

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	
N1	Order keepers can cause DoS in all operations	**OPEN**

Due to the two step execution system, any operation requires an order keeper to execute it. Trust is needed in that order keepers will timely execute all pending orders. If the order keeper does not submit execution transactions, all operations will cease to function, including closing positions and withdrawing funds.

It would be beneficial to implement fallback mechanisms that guarantee that users can at least withdraw their funds in case order keepers cease to function for any reason. For instance, the protocol could allow users to execute orders by providing oracle prices themselves, but only if an order is stale (a certain time has passed since its creation).		
N2	Order keepers can frontrun/reorder transactions	
There is nothing in the current system that prevents an order keeper from front-running or reordering transactions, for instance to exploit changes in the price impact. The protocol could include mechanisms that limit this possibility: for instance, the order keeper could be forced to execute orders in the same order they were created.		

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	Possible erroneous computation of price impact	**INFO**
Price impact is calculated as *(initial imbalance) ^ (price impact exponent) * (price impact factor) - (next imbalance) ^ (price impact exponent) * (price impact factor)* The values of exponent (e) and impact factor (f) are set by the protocol for each market. If the impact factor is simply a percentage, then the price impact will have units USD^(price impact exponent). But this seems erroneous, since price impact is treated as a USD amount which is finally added to the amount requested by the user. A problem arises in case that these two quantities are selected independently of each other but also of the pool's deposits and status.		

For example, consider a pool with tokens A and B of total USD value x and y respectively. Consider that x < y. Then the imbalance equals d = y - x. If a user swaps A tokens of worth d/2, then prior to the price impact he will get B tokens of the same value. The new deposits of the pool will now be x'=y'=(x+y)/2 and the pool will become balanced. The price impact for this transaction is f*dˆe, which could be (if the parameters are not chosen carefully) larger than d/2, which is the requested swap amount. Also, this fact could lead to a pool which is even more imbalanced than the previous state.
We suggest that (total_deposits)ˆ(e-1)*f always be less than 1 to avoid the above mentioned undesirable behavior.

A2	Inconsistency in the submission time of different tokens	**INFO**

Price keepers are responsible for submitting prices for most of the protocol's tokens. The submitted price should be the price retrieved from exchange markets at the time of each order's creation (the median of all these prices is finally used). However, for some tokens Chainlink oracles are used. In this case, the price at the time of the order execution is used.

A3	A user can liquidate its own position	**INFO**

`ExchangeRouter::createLiquidation` can be called only by liquidation keepers. However, there is nothing preventing a user from calling `createOrder` with `orderType=Liquidation`, effectively creating a liquidation order for their own position. Although this is not necessarily an issue, it is unclear whether this functionality is intentional or not.

A4	Inefficient price impacts on large markets	**INFO**

The price impact calculation is a function with an exponential factor of the **absolute** differences between open long and short interests before and after a trade. This works well for the average market, but on a large market with large open interests, it is not more efficient to open large positions. Consider that in other AMM designs, it is possible to open large positions with minimal price impact if the market is large (e.g., Uniswap ETH-USDC).

A5	Known compiler bugs	**INFO**
The code can be compiled with Solidity 0.8.0 or higher. For deployment, we recommend no floating pragmas, but a specific version, to be confident about the baseline guarantees offered by the compiler. Version 0.8.0, in particular, has some known bugs, which we do not believe affect the correctness of the contracts.		

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, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.