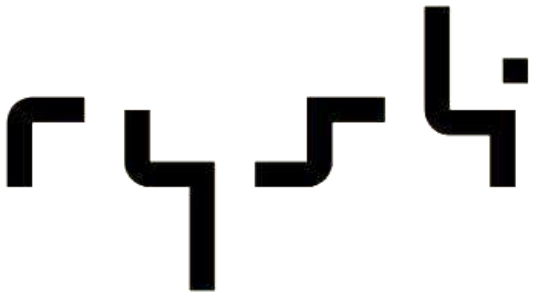
Rysk

Smart Contract Security Assessment

February 10, 2023



ABSTRACT

Dedaub was commissioned to perform a security audit of part of the Rysk protocol. The audit was based on commit 7613d8f of the dynamic-hedging repository.

Thee auditors worked on the codebase for 3 weeks on the following contracts:

```
dynamic-hedging
├── packages/
    ├── contracts/
        ├── contracts/
            ├── libraries/
            │   ├── CombinedActions.sol
                ├── RyskActions.sol
                ├── SABR.sol
            ├── BeyondPricer.sol
            ├── OptionCatalogue.sol
            ├── OptionExchange.sol
            ├── OptionRegistry.sol
            ├── VolatilityFeed.sol
```

OVERVIEW OF THE PROTOCOL

The Rysk protocol allows liquidity providers to deposit liquidity into a liquidity pool. This liquidity is used to provide the collateral required to write put or call options, which can be sold to users of the protocol. In return the pool obtains a premium, which is owned by the liquidity providers, and the protocol obtains a fee for the transaction.

Users of the protocol can also sell the options which have been underwritten back to the liquidity pool. In addition options minted on Opyn which have been whitelisted by Rysk may also be sold to the pool.

The pool tries to offer the right incentives to users to buy its options, or to sell options to it, in order to control its risk exposure. In addition, it is able to hedge its current portfolio through the use of perpetual forward contracts or spot marked operations.

Users of the protocol should be aware of the fact that the hedging operations mentioned above are orchestrated and carried out by a number of off-chain bots controlled by the protocol's quant team, while the hedging strategy used is a proprietary one.

DESCRIPTION OF CONTRACTS IN SCOPE

The BeyondPricer contract is responsible for pricing options contracts based on an adjusted Black-Scholes model which obtains its implied volatility from a SABR model (the parameters of this model are provided by bots controlled by the Rysk team). The OptionsCatalogue contract is responsible for whitelisting the options allowed to interact with the Rysk protocol. The OptionExchange contract is the entrypoint for users wishing to interact with the protocol, while the OptionRegistry contract is used by the OptionExchange to control all the interactions required with a variant of the Opyn protocol deployed by Rysk.

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 behavior. 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. 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 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:

[NO CRITICAL SEVERITY ISSUES]

HIGH SEVERITY:

ID	Description	STATUS
H1	OptionExchange::redeem() is susceptible to front-running	**RESOLVED**

The OptionExchange contract's `redeem()` function calls `_swapExactInputSingle()` with minimum output set to 0, making it susceptible to a front-running/sandwich attack when collateral is being liquidated. It is recommended that a minimum representing an acceptable loss on the swap is used instead.

```
// OptionExchange::redeem
function redeem(address[] memory _series) external {
  _onlyManager();
  uint256 adLength = _series.length;
  for (uint256 i; i < adLength; i++) {
    // ... Dedaub: Code omitted for brevity.

    if (otokenCollateralAsset == collateralAsset) {
      // ... Dedaub: Code omitted for brevity.
    } else {
      // Dedaub: Minimum output set to 0. Susceptible to sandwich attacks.
      uint256 redeemableCollateral =
        _swapExactInputSingle(redeemAmount, 0, otokenCollateralAsset);
      SafeTransferLib.safeTransfer(
        ERC20(collateralAsset),address(liquidityPool),redeemableCollateral
      );
      emit RedemptionSent(
        redeemableCollateral, collateralAsset, address(liquidityPool)
      );
```

```
    }
  }
}
```

H2	VolatilityFeed updates are susceptible to front-running	**DISMISSED**

The VolatilityFeed contract uses the SABR model to compute the implied volatility of an option series. This model uses a number of parameters which are regularly updated by a keeper through the `updateSabrParameters()` function. It is possible for an attacker to front-run this update, transact with the LiquidityPool at the old price and then transact back with the LiquidityPool at the new price (computed in advance) if the difference is profitable.

The Rysk team has indicated that trading will be paused for a few blocks to allow for parameter updates to happen and to effectively prevent this situation.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	No staleness check on the volatility feed	**ACKNOWLEDGED**

The function `quoteOptionPrice` of the BeyondPricer contract retrieves the implied volatility from the function `VolatilityFeed::getImpliedVolatility()`. However, the returned value is not accompanied by a timestamp that can be used by the `quoteOptionPrice()` function to determine whether the value is stale or not. Since the implied volatility returned is affected by a keeper, which is responsible for updating the parameters of the underlying SABR model, it is recommended that staleness checks are implemented in order to avoid providing wrong implied volatility values.

LOW SEVERITY:

ID	Description	STATUS
L1	Inconsistent use of price feeds for the price of the underlying	**DISMISSED**

The BeyondPrice contract gets the price of the underlying token via the function `_getUnderlyingPrice()`, which consults a Chainlink price feed for the price.

```
// BeyondPrice::_getUnderlyingPrice
function _getUnderlyingPrice(address underlying, address _strikeAsset)
  internal view returns (uint256)
{
  return PriceFeed(protocol.priceFeed()).
    getNormalizedRate(underlying, _strikeAsset);
}
```

However, when trying to obtain the same price in the function `_getCollateralRequirements()`, the addressBook is used to get the price feed from an Oracle implementing the IOracle interface.

```
// BeyondPrice::_getCollateralRequirements
function getCollateralRequirements(
  Types.OptionSeries memory _optionSeries, uint256 _amount
) internal view returns (uint256) {
  IMarginCalculator marginCalc =
    IMarginCalculator(addressBook.getMarginCalculator());

  return
    marginCalc.getNakedMarginRequired(
```

```
      _optionSeries.underlying,
      _optionSeries.strikeAsset,
      _optionSeries.collateral,
      _amount / SCALE_FROM,
      _optionSeries.strike / SCALE_FROM, // assumes in e18
      IOracle(addressBook.getOracle()).getPrice(_optionSeries.underlying),
      _optionSeries.expiration,
      18, // always have the value return in e18
      _optionSeries.isPut
    );
}
```

The same addressBook technique is used in the `getCollateral()` function of the OptionRegistry contract and in the `checkVaultHealth()` function of the Option registry contract.

It is recommended that this is refactored to use the Chainlink feed in order to avoid a situation where different prices for the underlying are obtained by different parts of the code.

The Rysk team intends to keep the price close to what the Opyn system would quote, thus using the Opyn chainlink oracle is actually correct as it represents the actual situation that would occur for these given quotes

L2	Multiple uses of div before mul in OptionExchange's _handleDHVBuyback() function	**RESOLVED**

In the OptionExchange contract's `_handleDHVBuyback()` function, a division is used before a multiplication operation at lines 925 and 932. It is recommended to use multiplication prior to division operations to avoid a possible loss of precision in the calculation. Alternatively, the `mulDiv` function of the PRBMath library could be used.

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	STATUS
N1	Centralized Implied Volatility Updates	**ACKNOWLEDGED**

The implied volatility used by the BeyondPricer contract to price options is determined by the SABR model. However, the SABR model is a function of several parameters set by bots controlled by the Rysk team. This means that the Rysk team has the ability to affect option prices through the control of these parameters.

The Rysk team has acknowledged the issue and has stated that the decentralization of the implied volatility computation is not currently feasible but will be part of their progressive decentralization efforts.

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 reentrancy in OptionRegistry::redeem()	**ACKNOWLEDGED**

The OptionRegistry's `redeem()` function is not access controlled and calls the OpynInteractions library contract's `redeem()` function, which interacts with the GammaController and the option and collateral tokens. Dedaub's static analysis tools warned about a potential reentrancy risk. Our manual inspection identified no such immediate risk, but as the tokens supported are not strictly defined and a future version of the code could potentially make such an attack possible, it is advisable to add a reentrancy guard around OptionRegistry's `redeem()` function.

ID	Description	STATUS
A2	Minor optimisation in OptionRegistry's *open()* function	**ACKNOWLEDGED**

The `OptionRegistry::open()` function performs the assignment vaultIds[series] = vaultId_ on line 271. But this can be moved into the if block starting at line 255, since the vaultId_ only changes value if this if block is executed.

```
// OpenRegistry::open
function open(
  address _series,
  uint256 amount,
  uint256 collateralAmount
) external returns (bool, uint256) {
  _isLiquidityPool();
  // make sure the options are ok to open
  Types.OptionSeries memory series = seriesInfo[_series];
  // assumes strike in e8
  if (series.expiration <= block.timestamp) {
```

```
    revert AlreadyExpired();
  }
  // ... Dedaub: Code omitted for brevity.
  if (vaultId_ == 0) {
    vaultId_ = (controller.getAccountVaultCounter(address(this))) + 1;
    vaultCount++;
  }
  // ... Dedaub: Code omitted for brevity.
  // Dedaub: Below assignment can be moved inside the above block.
  vaultIds[_series] = vaultId_;
  // returns in collateral decimals
  return (true, collateralAmount);
}
```

A3	Misleading comment in OptionExchange's `_swapExactInputSingle()` function	**RESOLVED**

The OptionExchange's `_swapExactInputSingle()` function definition is annotated with several misleading comments. For instance, it mentions that `_amountIn` has to be in WETH when it can support any collateral token. It also mentions that `_assetIn` is the stablecoin that is bought, when it is in fact the collateral that is swapped. The description of the function, which reads "function to sell exact amount of WETH to decrease delta" is incorrect.

```
// OptionExchange::_swapExactInputSingle
/** @notice function to sell exact amount of wETH to decrease delta
 *  @param _amountIn the exact amount of wETH to sell
 *  @param _amountOutMinimum the min amount of stablecoin willing to
          receive. Slippage limit.
 *  @param _assetIn the stablecoin to buy
 *  @return the amount of usdc received
 */
function _swapExactInputSingle(
```

```
  uint256 _amountIn,
  uint256 _amountOutMinimum,
  address _assetIn) internal returns (uint256) {
  // ... Dedaub: Code omitted for brevity.
}
```

A4	Misleading comment in BeyondPricer's `_getSlippageMultiplier()` function	**RESOLVED**

The division of the `_amount` by 2, mentioned in the code comment, does not appear in the code. It appears that this comment corresponds to a previous version of the codebase and it should be removed.

```
//BeyondPricer::_getSlippageMultiplier
function _getSlippageMultiplier(
  uint256 _amount,
  int256 _optionDelta,
  int256 _netDhvExposure,
  bool _isSell
) internal view returns (uint256 slippageMultiplier) {
  // divide _amount by 2 to obtain the average exposure throughout the tx.
  // Dedaub: The above comment is not relevant any more.
  // ... Dedaub: Code omitted for brevity.
}
```

A5	SABR library's `lognormalVol()` can in principle return negative values	**ACKNOWLEDGED**

The formula of the SABR model that is responsible for computing the implied volatility (https://web.math.ku.dk/~rolf/SABR.pdf formula (2.17a)) is an approximate one. It is not clear to us if this value will always be non-negative as it should be. For example,

for absolute values of ρ close to 1 and large values of v, the last term of this formula, and probably the whole value of the implied volatility will be negative.
The execution of `VolatilityFeed::getImpliedVolatility` will revert if the value returned by `lognormalVol()` is non-negative, to protect the protocol from using this absurd value. Nevertheless, if this keeps happening for a while, the protocol will be unable to price the options and therefore will be unable to work.

This issue could be avoided either by a careful choice of the SABR parameters by the protocol's keepers or by using an alternative volatility feed in case this happens.

A6	Missing check in BeyondPricer's `quoteOptionprice()`	**RESOLVED**

In `BeyondPricer::quoteOptionPrice()` a check that `_optionseries.expiration >= block.timestamp` is missing. If the function is called to price an option series with a past expiration date, it will return an absurd result. We suggest adding a check that would revert the execution with an appropriate message in case the condition is not satisfied.

A7	OptionExchange::_checkHash is defined as public even though its name suggests otherwise	**RESOLVED**

Function `OptionExchange::_checkHash`, which returns if an option series is approved or not, is defined as public. However, the starting underscore in "`_checkHash`" implies that this functionality should not be exposed externally (via the public modifier) creating an inconsistency, even though it is probably useful/necessary to the users of the protocol.

A8	OptionExchange::_buyOption returns an incorrect value	**RESOLVED**

Whenever a user wants to buy an amount of options, first it is checked if the long exposure of the protocol to this option series is positive. If this is the case, then the protocol first sells the options it holds, to decrease its long exposure, and if they are not

enough, then the Liquidity pool writes extra options to reach the amount requested by the user. The problem is that the `_buyOption` function, in the case the Liquidity pool is called to write these extra options, returns only this extra amount, and not the total amount sold to the user.

A9	Consistency of compiler versions	**RESOLVED**

The code of the BeyondPricer, OptionExchange and OptionCatalogue contracts is compiled with the floating pragma >=0.8.0, and the OptionRegistry contract is compiled with the floating pragma >=0.8.9. It is recommended that the compiler version is fixed to a specific version and that this is kept consistent amongst source files.

A10	Compiler bugs	**ACKNOWLEDGED**

The code of the BeyondPricer, OptionExchange and OptionCatalogue contracts is compiled with the floating pragma >=0.8.0, and the OptionRegistry contract is compiled with the floating pragma >=0.8.9. Versions 0.8.0 and 0.8.9 in particular, have 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.