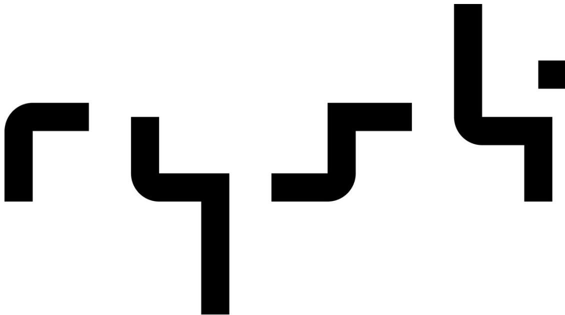
Rysk

Smart Contract Security Assessment

18.06.2022



ABSTRACT

Dedaub was commissioned to audit the Rysk protocol, expected to be deployed on Arbitrum. The Rysk protocol is a decentralized options exchange which aims to achieve sustainable yield using dynamic delta hedging. This audit report covers commit hash `609da7bfdb1289e150a872afeea92e483a0e7982`. Two auditors and a mathematician worked over the codebase over 4.5 weeks. `PerpHedgingReactor.sol` was audited separately and covers commit hash `9d3174fc6f68eec996e71d500ad00a0aa7ab6aca`.

The codebase appears to be well-tested and covers many corner cases. Detailed documentation was provided. The team auditing this protocol was already knowledgeable in decentralized options protocols and to some extent delta hedging of options portfolios. The audit did not include the forked Opyn Gamma protocol, standard libraries and offchain oracles. Although a number of issues were found as part of this audit, many of these issues can be easily resolved by the protocol team.

Security Opinion

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“) was a secondary consideration, however intensive efforts were made to check the correct application of the mathematical formulae in the reviewed code. 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. This scope of the audit also included crypto-economic considerations. A number of checks have been carried out, however the crypto-economic effectiveness of this specific design is novel. Therefore, the financial viability of this protocol in real market conditions cannot be fully established. In terms of protocol completion, the protocol appears to be ready for staging.

In terms of architecture, Dedaub notes that there are several design decisions that ensure the economic security of the protocol. These include multiple delta hedging reactors, and the various incentives in the computation of options prices.

There are two kinds of Rysk users - liquidity providers and options buyers / market makers. Users of the Rysk protocol can become liquidity providers by depositing collateral into the Liquidity Pool. This collateral is then used to collateralize/short the options that the protocol issues and sells. When an option is sold, the premium is distributed to the depositors proportionally to their shares. This yield comes, of course, at a risk. In essence, the liquidity providers are exposed to the option. If the option expires in-the-money they will lose part of their deposit. The purpose of the Rysk protocol is to accomplish a portfolio uncorrelated to the market of the underlying crypto assets.

The correlation of the option price and the underlying asset is measured using $Delta = \partial V / \partial S$, where V is the premium price of the option and S is the price of the underlying asset. A total portfolio Delta (sum of the deltas of all the options for the same underlying asset) equal to 0 means that the portfolio is uncorrelated to the market. There are two issues to consider at this point. First, regarding the exact computation of Delta and, second, how the protocol achieves a delta neutral state. Rysk takes care of both of these issues using advanced mathematical methods and implementing investment strategies used for a long time in TradFi, which we will briefly describe in the following.

Given the strike price, the expiration date and the price of the underlying asset (this last price is retrieved by oracles) the premium V is computed using the Black-Scholes model. However,this is not the option's final price. A utilization premium is added and a delta skew is applied in order to achieve zero delta for the portfolio. The utilization premium equals to $utilizationFactor \cdot V$, where utilizationFactor is computed using the formula:
$utilizationFactor = m_1 \cdot x$, if $x < threshold$

$utilizationFactor = m_2 \cdot x + c$, if $x > threshold$

where m_1, m_2, c and threshold are constants to be decided by governance and x is the Liquidity Pool utilization. Constant m_2 will be significantly larger than m_1, i.e. when the utilization of the Liquidity Pool is high, the protocol prices the options more expensively, to control the sales of options and keep the users safe from liquidation.

The last step in the computation of an option's price is the application of the delta skew. The normalized delta is computed (average of the delta before and after the transaction divided by the total value of the portfolio) and $deltaSkewFactor = min(normalizedDelta, maxDiscount)$, where maxDiscount is a constant to be decided by governance. The amount $deltaSkewFactor \cdot utilizationPrice$ is added to or subtracted by the utilization price accordingly if the transaction increases or decreases the value of delta, i.e. the protocol prices higher options which tend to increase delta, therefore incentivizes users to buy options leading towards a delta-neutral state.

The protocol also makes use of hedging reactors to achieve a delta-neutral state. These reactors are contracts which make use of other protocols to open/close positions that will shift the portfolio closer to a delta-neutral state. Collateral from the liquidity pool is used to cover these positions.
Currently two hedging reactors have been implemented, UniswapV3HedgingReactor and PerpHedgingReactor, and the protocol has been designed such that new hedging reactors can easily be added or changed over time.

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 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	`PortfolioValuesFeed::setLiquidityPool` is callable by anyone	**FIXED**

Method setLiquidityPool of the PortfolioValuesFeed contract should be callable only by the governor of the protocol. However, this is not enforced in code as there is no call to the `_onlyGovernor` access control method. To be fully precise, there is a reference to the `_onlyGovernor` method but not a call to it, as can be seen below:

```
function setLiquidityPool(address _liquidityPool) external {
    _onlyGovernor;  // Dedaub (bug): _onlyGovernor();
    liquidityPool = ILiquidityPool(_liquidityPool);
}
```

This was obviously an omission but still one that an attacker could exploit to mess up the protocol's state. Specifically, by pointing the liquidityPool storage variable to a dummy LiquidityPool contract there would be no call to the `resetEphemeralValues` method of the original LiquidityPool when `PortfolioValuesFeed::fulfill` would be called by the oracle, leaving the ephemeral values of the LiquidityPool "dirty".

The protocol can be exploited in the following scenario, given the following preconditions:

1) `ephemeralDelta < 0`
2) The portfolio delta computed by the offchain oracle, passed through fulfill is negative.

By skipping the reset of ephemeralDelta, the value reported by `getPortofolioDelta` (`delta+ephemeralDelta+externalDelta`) is off by `ephemeralDelta`, which is actually beneficial to the attacker as they get the option cheaper.

H2	Incorrect calculation of price differences	FIXED

`OptionsCompute::calculatePercentageDifference` is incorrect. It computes a ratio, and where this ratio is used (`OptionsCompute::validatePortfolioValues`), it expects a value that characterizes the price differences between assets (the higher the difference, the higher the value). As can be seen in the below code snippet of the validatePortfolioValues method, if the priceDelta variable holds the ratio instead of the percentage difference of the asset prices then the computation will revert with a PriceDeltaExceedsThreshold error in scenarios where the actual price difference is small and execute normally when there is significant difference in prices, essentially working the opposite way from what would be expected.

```
function validatePortfolioValues(
    uint256 spotPrice,
    Types.PortfolioValues memory portfolioValues,
    uint256 maxTimeDeviationThreshold,
    uint256 maxPriceDeviationThreshold
) public view {
    uint256 timeDelta = block.timestamp - portfolioValues.timestamp;
    // If too much time has passed we want to prevent a possible oracle attack
    if (timeDelta > maxTimeDeviationThreshold) {
        revert CustomErrors.TimeDeltaExceedsThreshold(timeDelta);
    }
    uint256 priceDelta = calculatePercentageDifference(spotPrice,
        portfolioValues.spotPrice);
    // If price has deviated too much we want to prevent a possible oracle attack
    if (priceDelta > maxPriceDeviationThreshold) {
        revert CustomErrors.PriceDeltaExceedsThreshold(priceDelta);
    }
}
```

As such, the protocol would not be able to operate under normal/expected conditions.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	LINK funds of PortfolioValuesFeed can be depleted	**FIXED**

An attacker can waste all the LINK tokens deposited into the PortfolioValuesFeed contract by repeatedly calling method `requestPortfolioData`, as there is no limit on how often or under which protocol conditions it could be called.

ID	Description	STATUS
M2	LINK might not be retrievable from `PortfolioValuesFeed`	**FIXED**

The constructor of PortfolioValuesFeed is provided the address of the LINK token contract via the `_link` parameter.

```
constructor(
    address _oracle,
    bytes32 _jobId,
    uint256 _fee,
    address _link,
    address _authority
) AccessControl(IAuthority(_authority)) {
    if (_link == address(0)) {
        setPublicChainlinkToken();
    } else {
        setChainlinkToken(_link);
    }
    oracle = _oracle;
    jobId = _jobId;
    fee = _fee;
    link = _link;
}
```

When `_link` is set to the 0 address, the setPublicChainlinkToken method is called to set the respective storage variable of the parent contract (ChainlinkClient). Nevertheless, the `link` storage variable of the PortfolioValuesFeed contract is set to 0,

while there is no method to update it later. In such a scenario (`link` is set to 0), it becomes impossible to retrieve any LINK deposited into the contract via the withdrawLink method.

M3	`PortfolioValuesFeed::fulfill` can be sandwiched	**DISMISSED**

`PortfolioValues::fulfill` can be sandwiched using MEV techniques or simply by predicting the request/fulfill delay and replicating the oracle's algorithm offline.

Scenario 1 (decentralized chains):

1) LiquidityPool has relatively high EphemeralDelta and stale portfolio greeks
2) Attacker detects calls to fulfill frontruns by buying a call or put option (depending on option pricing after fulfill).
3) Attacker immediately sells back the option after fulfill is executed.

Scenario 2 is the same as scenario 1, but in this case the attacker estimates the delay between calling `LiquidityPool::requestPortfolioData` and the oracle calling `LiquidityPool::fulfill`. In addition the attacker has to replicate the options pricing algorithm offline (the details on public whitepapers are probably enough to approximately replicate this) and perform the attack over multiple transactions.

M4	Liquidity providers may experience a “bank run” situation	**ACKNOWLEDGED**

According to Rysk:

> *Liquidity Providers are users who will provide liquidity to the vaults in the interest of generating yield.*

When users buy options from Rysk part of the total liquidity gets locked to back the written/sold options up to their expiration. At the same time, extra liquidity is used as a buffer for margin requirements of the options vault. If a liquidity provider decides to withdraw their funds, they can do so as long as the amount is less than the total provided liquidity minus the liquidity backing the options and the buffer.

Bank run situations in a high liquidity utilization scenario might leave some users unable to retrieve their deposited liquidity in whole, as the protocol needs to maintain certain liquidity to collateralize the written/sold options as described above.

[The issue was acknowledged and addressed in a newer version of the protocol by the introduction of a partition system that ensures a user's withdrawal request will be honored when the associated epoch is processed. The specific version of the protocol was not part of this audit.]

M5	Delta skew calculation is gameable	DISMISSED

The method with which delta skew is calculated is exploitable. Delta skew is a function that incentivizes or disincentivizes the price of an option based on whether the transaction causes the pool to converge to neutrality. Unfortunately the skew does not take into consideration the net asset value (NAV) of the pool after the transaction succeeds. The following scenario demonstrates a typical attack.

Scenario 1, selling options back to the system. Preconditions:

1. Alice wants to sell her options back to the pool.
2. The liabilities of her options **are a significant part of the NAV** of the pool.
3. Alice is a whitelisted seller
4. The transaction will cause the pool to increase its absolute delta

In this scenario, the normalized delta is computed by using the NAV in the denominator, which is higher than after the transaction succeeds. Therefore, Alice is receiving an unfair discount when increasing the portfolio delta.

Scenario 2, buying options. Preconditions:

1. Alice wants to buy an option

2. The increase in liabilities after minting the options would be **a significant part of the NAV** of the pool.
3. The transaction will cause the pool to decrease its absolute delta

In this scenario, Alice receives a discount when buying the option. However, since this is calculated on the current NAV, it will be higher than the actual contribution of delta shifting considering the NAV of the pool after the transaction.

These issues can be confirmed by replicating each scenario using multiple smaller transactions, vs. larger transactions, and observing the delta skew.

[The issue has been **dismissed** by the developers of the protocol as the delta skew calculation will not be employed in the Rysk Alpha launch codebase, i.e., the respective code has been removed according to the specification of the new protocol version.]

M6	`amountOutMinimum` always set to 0 when performing swaps	FIXED

In `UniswapV3HedgingReactor::hedgeDelta`, `amountOutMinimum` is always set to 0. This value is used in `UniswapV3HedgingReactor::_swapExactInputSingle` when selling wETH in exchange for USDC.

If the protocol was deployed in Ethereum, an attacker could manipulate the price on the Uniswap pool prior to this call to steal all of the collateral being hedged by sandwiching the reactor's swap since there is currently no lower limit for the amount of USDC returned to the reactor. However, Arbitrum, the L2 solution where the protocol is going to be deployed, uses a FIFO sequencer instead of a public mempool, which drastically reduces the chances of such attacks. At the same time, Arbitrum is working with Chainlink on creating a decentralized transaction ordering solution that will minimize MEV. Nevertheless, as Arbitrum is still evolving and no solution is set in stone, it is suggested to use a price value for ETH given by the user (LiquidityPool), which could also be computed using an Oracle.

LOW SEVERITY:

ID	Description	STATUS
L1	Removing a non-empty hedging reactor	**FIXED**

`LiqiudityPool::removeHedgingReactorAddress` does not check whether a hedging reactor contains any funds before removing it. Once a hedging reactor is removed from the liquidity pool, any remaining funds allocated to it will not be accessible, as `UniswapV3HedgingReactor::withdraw` can only be called by the parent LiquidityPool.

Currently, the only way to access these funds is to add the hedging reactor back to the LiquidityPool, withdraw all the funds from it, and then remove it again.
A better solution would be to close any positions held by the reactor, and then withdraw all funds as part of `LiqiudityPool::removeHedgingReactorAddress`.

ID	Description	STATUS
L2	Inaccurate margin requirements risk calculation	**DISMISSED**

This issue details small issues related to the appropriate pricing of collateral utilization due to a combination of (1) Vault health checks when buying options, (2) Dynamic naked margin requirements in Opyn, and (3) the "impedance mismatch" between these.

Example 1: Not pricing margin requirement "upper bound" transition time
In Opyn Gamma, the margin requirements change when the time to expiry crosses a predefined threshold.

For instance, if an option has 2 weeks until expiry, for each day that passes the margin requirements on Opyn in this example are 5% less each day (working below).

Margin upper bound value (2 week expiration): 2.7e26
Margin upper bound value (1 week expiration): 1.9e26
Number of days in a week: 7

Note that: $2.7e26 * 0.95^7 \approx 1.9e26$

However, the pricing of an option only considers the fundamental pricing, collateral utilization, and delta skew. If the time to the next transition is also included in the pricing (the collateral utilization adjusted slightly), the protocol can manage the risk of vault liquidations slightly better.

Example 2: Preventing the portfolio from achieving Delta neutrality
Whenever a vault's naked margin requirements are below a certain threshold, the protocol cannot issue options in this vault. This may have the effect of preventing the portfolio from achieving Delta neutrality. As explained in Example 1, a major component of the vault's health is derived from Opyn's time to expiry threshold, that's typically a step function over time, each step being 1 week. The health check can take into consideration the time until the next step.

L3	Inconsistent computation of utilization premium	ACKNOWLEDGED

The utilization premium, which is previously described under "Security Opinion", does not fairly calculate the utilization premium for options that have a relatively high price.

Although we cannot think of a way this might be exploited, as an important component in pricing an option this could be improved. It can potentially result in pricing options higher than the market price, thus reducing the efficiency of the protocol.

The scenario arises since the current collateral does not factor the additional collateral that is added as a result of the option pricing (see `collateralAsset.balanceOf(this)`, which computes the utilization after) in the following code.

```
function addUtilizationPremium(...) {
...
  quoteState.utilizationBefore = collateralAllocated_.div(
    collateralAllocated_ +
ERC20(collateralAsset).balanceOf(address(this))
  );
  ...
  quoteState.utilizationAfter = (
    quoteState.collateralToAllocate + collateralAllocated_).div(
      collateralAllocated_ +
ERC20(collateralAsset).balanceOf(address(this))
  );
  ....
}
```

Note that the option price is only levied at the end of the transaction that mints the option, for instance in `OptionHandler::executeOrder`.

Collateral stemming from this fee is added after the utilization premium computation takes place, for instance, in executeOrder:

```
function executeOrder(...) … {
  ...
  // addUtilizationPremium called through quotePriceWithUtilizationGreeks
  ...
  SafeTransferLib.safeTransferFrom(
                collateralAsset_,
                msg.sender,
                address(liquidityPool),
```

```
                    convertedPrem
            );
    ...
}
```

The option price may be significant, in cases when the option is in the money. It should be noted that under normal market conditions, if the pool is backing ITM options, it would likely be due to a long-term shift in price of the underlying and it's also likely that the pool is not delta neutral.

L4	`PriceFeed::getNormalizedRate` should check the oracle's price	**FIXED (partially)**

`PriceFeed::getNormalizedRate` returns the price of an asset after consulting a Chainlink data feed. According to Chainlink data feeds docs the data feed aggregator defines `minAnswer` and `maxAnswer` values, which define the range of acceptable and thus reported values. Nevertheless, not all applications share the same definition for the range of acceptable values. Also, an application should not depend on the data feed aggregator for such sensitive protocol values. An application should be able to handle times when the price reported by Chainlink remains stale due to being outside of the range of acceptable values. Also, it should respond accordingly to extreme price volatility or even handle the extreme scenario in which the data feed reports a totally inaccurate value.

Additional checks and price sources could be used to make the application more robust. More specifically, the two dominant solutions employed by an array of well established protocols are (1) maintaining a price TWAP and (2) querying a protocol that maintains its own price feed/TWAP in case the Chainlink reported price deviation exceeds a predefined threshold.

[The issue has been **partially addressed** by adding staleness and anomalous value checks for the values returned by the Chainlink oracle. The protocol developers have not opted for a fallback (TWAP) oracle due to the lack of trustworthy alternatives to Chainlink oracles on the Arbitrum network at the moment.]

L5	PerpHedgingReactor::_changePosition does not check actual amount swapped	FIXED

When PerpHedgingReactor::_changePosition is called, it is passed an _amount parameter, which represents the amount of position to open/close. The function then returns this same _amount parameter as is, without performing any checks to ensure that a position of exactly _amount size has been created on the Rage Trade protocol.

Internally Rage Trade uses Uniswap V3 pools to create its vAMMs. PerpHedgingReactor::_changePosition calls ClearingHouse::swapToken of Rage Trade that calls the swap method of a slightly modified UniswapV3Pool. After studying the pool's implementation and how Uniswap and Rage Trade fees are calculated we concluded that the created position will be of exactly _amount size, which is inline with the understanding of the Rysk developers. It is crucial to ensure that, as otherwise, the accounting of the delta of the LiquidityPool and the internalDelta of the PerpHedgingReactor would be incorrect. A defensive solution would be to have PerpHedgingReactor::_changePosition return the actual size of the position created (or in other words the amount swapped by the pool and returned by ClearingHouse::swapToken), vTokenAmountOut.

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	Modifiers should be preferred to functions for access control	**INFO**
	The developers of the protocol have opted for using functions instead of modifiers for access control purposes. We would advise against that, as in our opinion modifiers are easier to read, understand and reason about. At the same time, issues like C1 cannot arise.	
A2	`LiquidityPool::settleVault` could be made external	**FIXED**
	`LiquidityPool::settleVault` is defined as public while it is not called by any other method of the LiquidityPool contract and thus could be made external.	
A3	`LiquidityPool::quotePriceWithUtilizationGreeks` could be made external	**FIXED**
	`LiquidityPool::quotePriceWithUtilizationGreeks` is defined as public while it is not called by any other method of the `LiquidityPool` contract and thus could be made external.	
A4	Inconsistent function naming practice	**FIXED**
	Most of the methods defined as internal have their name beginning with an underscore. However, this is not true for every internal method. Examples, which are not following this convention, are: • `LiquidityPool::getVolatilityFeed()` • `LiquidityPool::getPortfolioValuesFeed()` • `LiquidityPool::getOptionRegistry()` • `LiquidityPool::getUnderlyingPrice(address, address)`	

• LiquidityPool::getNormalizedBalance(address)		
A5	OptionHandler::pauseContract could be renamed to pause	**FIXED**
Method pauseContract of the OptionHandler contract could be renamed to pause as its symmetric method is named unpaused.		
A6	Incorrect comment in OptionRegistry::adjustCollateralCaller	**FIXED**
The comment in OptionRegistry::adjustCollateralCaller regarding making sure that the balance change is recorded in the LiquidityPool is incorrect, as the collateral is coming from the caller of the method, not from the LiquidityPool.		
A7	OptionRegistry::redeem performs multiple calls to OptionSeries::balanceOf while the result can be cached	**FIXED**
In OptionRegistry::redeem the call ERC20(_series).balanceOf(msg.sender) is performed thrice. The three calls could be cut down to one and the result could be cached and reused, as there are no changes to the actual balance of the msg.sender in between the three calls.		
A8	Method PriceFeed::getRate could be made external	**FIXED**
PriceFeed::getRate is defined as public while it is not called by any other method of the PriceFeed contract and thus could be made external. Otherwise, getRate could be called by getNormalizedRate, leading to less code duplication.		
A9	Protocol storage variables can be made immutable	**FIXED**
Storage variables optionRegistry and priceFeed of the Protocol contract can be made immutable, as they cannot be modified after the contract’s construction.		
A10	Unused error defined in AccessControl.sol	**FIXED**

The error AUTHORITY_INITIALIZED defined in AccessControl.sol is not used anywhere.

A11	VolatilityFeed::_isKeeper could use custom error in revert	FIXED

VolatilityFeed::_isKeeper reverts when the msg.sender is not a keeper, manager or governor. Nevertheless, the revert does not use the custom error NotKeeper defined in CustomErrors

A12	LiquidityPool::_isTradingPaused should be renamed to _isNotTradingPaused	FIXED

LiquidityPool::_isTradingPaused is used to stop and revert execution when trading has been paused. Thus, the function should be renamed to _isNotTradingPaused to reflect the fact that not reverting, i.e., returning, means that trading has not been paused.

A13	aboveThresholdYIntercept can be calculated on-chain	FIXED

LiquidityPool::setUtilizationSkewParams sets 4 values which represent the utilization skew graph. These are:

- belowThresholdGradient
- aboveThresholdGradient
- aboveThresholdYIntercept
- utilizationFunctionThreshold

aboveThresholdYIntercept can be calculated using the other parameters rather than given by the caller. Doing this would ensure that the 2 lines represented by the function must always intersect at the utilizationFunctionThreshold, as intended.

```
aboveThresholdYIntercept = _utilizationFunctionThreshold *
                  (_belowThresholdGradient - _aboveThresholdGradient)
```

A14	Mistakes in Comments / Docs	FIXED

In `LiquidityPool::setBufferPercentage`

`_bufferPercentage` comment is incorrect.

In `OptionHandler::createStrangle`

Order of return values does not match function comment.

In `OracleFeeds.md`

Tracks events from the request and earlier (not later)

A15	PerpHedgingReactor storage variables can be made immutable	FIXED

Storage variables `accountId`, `collateralId` and `poolId` of the PerpHedgingReactor contract can be made immutable, as they cannot be modified after the contract's construction.

A16	Compiler bugs	INFO

The contracts were compiled with the Solidity compiler v0.8.9 which, at the time of writing, has some known bugs. We inspected the bugs listed for this version and concluded that the subject code is unaffected.

CENTRALIZATION ASPECTS

As is common in many new protocols, the owner of the smart contracts yields considerable power over the protocol, including changing the contracts holding the user's funds, setting important price parameters, collateralization, etc.

In particular, whoever controls the governor role can:

- Change the options pricing parameters passed to the Black Scholes equations, minting themselves free Options.

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.