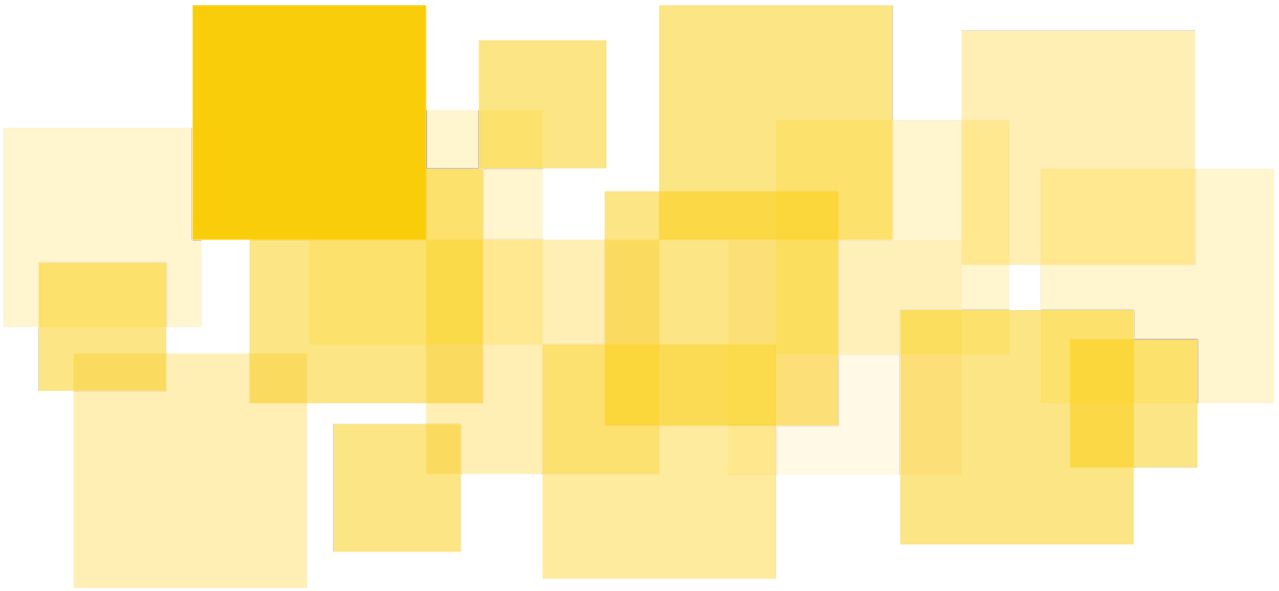
Security Audit Report

Octant v2 Ethereum

Delivered: February 24, 2025



runtime verification

Table of Contents

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Executive Summary

Golem Foundation engaged Runtime Verification Inc. to conduct a security audit of the Octant V2 Core smart contract system. The goal was to identify vulnerabilities that could impact fund security, access control, protocol correctness, and overall system robustness.

The audit was conducted over a period of three calendar weeks between February 3rd, 2025 and February 24th, 2025, during which Runtime Verification performed an in-depth review of the Dragon Strategies and their interaction with the broader Octant V2 system. Special attention was given to deposit logic, asset-to-share ratio mechanisms, and voluntary lockup periods, ensuring these components behaved as intended under various execution scenarios.

Our approach combined manual code review, dynamic analysis, and property-based testing. Where applicable, we formalized key protocol invariants and evaluated them using symbolic execution and test-driven verification within the constraints of the audit timeline. These invariants are described and analyzed in the section "Invariants", with our formal verification efforts described in "Kontrol Formal Verification". The issues uncovered by the audit can be found in the sections "Findings" and "Informative Findings".

This audit reinforces our formal methods-driven approach to security, ensuring that Octant V2 Core operates with strong guarantees of correctness and resilience.

Scope

The scope of this audit includes a review of specific smart contracts within the Octant V2 Core repository, as provided by the client, Golem Foundation. The audit is focused on the Dragon Strategies module, which introduces tokenized vault strategies designed to manage ERC-20 assets efficiently. The primary objective of the audit is to identify potential security vulnerabilities, assess correctness, and ensure the robustness of the contract logic.

The audit covers the following contracts within the repository at commit f65cac2db7ea53c722916efc060d90c4c6748cc8:

`BaseStrategy.sol` – Serves as the foundational contract for strategy implementations. While modified, the client has a high level of confidence in its correctness.
`DragonTokenizedStrategy.sol` – Implements the core strategy for managing deposited assets within the vault.
`TokenizedStrategy.sol` – Handles tokenized vault logic, including deposits and withdrawals. The _deposit function in this contract was flagged as one of the most complex pieces of logic requiring detailed scrutiny.
`YearnPolygonUsdcStrategy.sol` – A strategy designed to interact with Yearn Finance on Polygon for USDC deposits.

External Dependencies and Considerations

The audited contracts interact with various external protocols, libraries, and modules, including yield strategies, governance modules, and multisig frameworks. These strategies are designed to support ERC-20 tokens, including WETH, ETH, and USDC, and are intended for deployment across multiple EVM and non-EVM chains, such as Ethereum, OP Superchains, Arbitrum, and Polygon.

Limitations of Scope

The audit is limited to the Solidity-based smart contracts listed above within the specified commit. Off-chain components, auto-generated code, client-side applications, deployment

scripts, and contracts beyond the provided scope are not included in this review. Any interactions with external vaults or third-party protocols are assessed only to the extent that they affect the security of the audited contracts.

Additionally, commits addressing any findings in this report will be reviewed to verify the resolution of identified issues.

Methodology

Runtime Verification takes an approach to security audits informed by our background as experts in formal methods. As such, our methodology involves a combination of the following:

- **Design review:** We start our audit by using our discussions with the client and any documentation provided to develop an understanding of the desired behavior of the protocol. In the process, we identify important properties and invariants that should always be true during the lifetime of the contracts.
- **Code review:** We then inspect the code of the contracts to check for common bugs as well as make sure that the implementation matches the intended behavior that we identified in the design review. During this inspection, we try to make sure that the previously-specified invariants hold in all code paths. In the process, we identify corner cases that were not considered during the design review and, after confirming the intended behavior with the client, refine our invariants to account for those. Any scenarios that we identify that violate the invariants are reported as findings.
- **Pen-and-paper formal analysis:** When a particular property of the protocol design or implementation is too complex to analyze simply by inspecting the code, we may resort to reasoning about it in the form of a mathematical proof. This process is more thorough than a simple code review, helping to cover all possibilities and avoid overlooking corner cases. If a violation is not found in the process, the written-down analysis serves as a proof that the property holds, and can be checked by others to develop confidence in the protocol.
- **Tool-assisted formal verification:** We complement our manual analysis with automated formal verification using Runtime Verification's tool Kontrol. Kontrol is a symbolic execution engine that integrates with the Foundry framework for Solidity property testing. Rather than using fuzzing, Kontrol allows parameterized tests to be executed symbolically, meaning that the input parameters are interpreted as symbolic mathematical variables that can take any value. Using its symbolic reasoning capabilities, Kontrol can verify that a test will pass for every possible input, giving a level of assurance beyond simple fuzzing. As part of the audit, we have

written Kontrol tests for some of the invariants that we identified, and were able to formally verify most of these tests. These tests can furthermore be re-run after code changes to ensure that future versions of the code continues to preserve the invariants.

Invariants

During the design review, we identified the following important invariants of the `DragonTokenizedStrategy` contract:

1. `totalSupply == totalAssets`, unless the strategy suffers a loss greater than can be recouped by burning shares of the `DragonRouter`.
2. For any user, `balances[user] <= totalSupply`.
3. A user who hasn't rage quit cannot withdraw funds if their lockup hasn't expired.
4. A user other than the `target` address cannot have their `unlockTime` changed (increased or decreased) before it expires, except by initiating a rage quit.
5. If a user has rage quit, they can't receive more assets.
6. If the current timestamp is `block.timestamp`, a user who wishes to withdraw all of their assets can do so by `min(unlockTime, block.timestamp + RAGE_QUIT_COOLDOWN_PERIOD)`.
7. If a user has rage quit, their funds are unlocked proportionally to the percentage of their lockup that has elapsed.

Invariants 1-5 have been checked by a manual analysis of the code. Below we present pen-and-paper proofs describing the detailed reasoning that we followed to ensure each of them holds. We have found violations for invariants 6 and 7, reported in the "Findings" section as findings A03 and A04, respectively.

Additionally, we have complemented the manual analysis with automated formal verification using the Kontrol tool. We have written Kontrol tests for invariant 1 as well as other additional properties. See the section "Kontrol Formal Verification" for details.

Invariant Analysis 1: Fixed Exchange Rate

Invariant Statement

Invariant: `totalSupply == totalAssets`, unless the strategy suffers a loss greater than can be recouped by burning shares of the `DragonRouter`.

Auxiliary Result: Conversion between Shares and Assets

Before showing that the invariant holds, we prove the following auxiliary result:

- If `totalSupply == totalAssets`, then `_convertToShares(S, assets, rounding) == assets` and `_convertToAssets(S, shares, rounding) == shares`.

The implementation of `_convertToShares` has three cases:

1. If `totalSupply == 0`, then it returns `assets` (link).
2. Otherwise, if `totalAssets == 0`, then it returns `0` (link). However, this case cannot happen if `totalSupply == totalAssets`, since we would have returned in case (1).
3. Otherwise, it uses the OpenZeppelin library function `mulDiv` to return `(assets * totalSupply) / totalAssets`, rounding up or down depending on the `rounding` parameter (link). In either case, since `totalSupply == totalAssets`, the division is exact and returns `assets` (unless there is an overflow, in which case it would revert).

Therefore, in all valid cases the function returns `assets`.

The implementation of `_convertToAssets` similarly has two cases (link):

1. If `totalSupply == 0`, then it returns `shares`.

2. Otherwise, it uses `mulDiv` to return `(shares * totalAssets) / totalSupply`. Again, since `totalSupply == totalAssets`, the division is exact regardless or rounding and returns `shares` as long as it doesn't revert due to overflow.

Therefore, the function returns `shares` in both cases.

Since the storage data `S` is implied and, as we showed above, `rounding` does not affect the result under the assumption that `totalSupply == totalAssets`, below we use `_convertToShares(assets)` and `_convertToAssets(shares)` as a shorthand.

Invariant Base Case

When the contract is deployed, `totalSupply == 0` and `totalAssets == 0`, therefore the invariant holds.

Invariant Step

We assume that the invariant holds in the current state. Then, if a loss that is greater than the shares of the `DragonRouter` hasn't occurred yet, `totalSupply == totalAssets`. We show that the invariant is maintained after every external call; that is, that `totalSupply' == totalAssets'`, where `X'` denotes the value of the variable `X` after the call returns.

The external functions that might affect this invariant are the following:

- The external functions that update `totalAssets` are `report`, those that call `TokenizedStrategy._deposit` (`deposit`, `depositWithLockup`, `mint`, `mintWithLockup`), and those that call `TokenizedStrategy._withdraw` (`withdraw`, `redeem`).
- The external functions that update `totalSupply` are those that call `TokenizedStrategy._mint` (`report`, `deposit`, `depositWithLockup`, `mint`, `mintWithLockup`) and those that call `TokenizedStrategy._burn` (`report`, `withdraw`, `redeem`).

We consider each of these external functions and show that they preserve the invariant.

`report`

This function harvests the funds and stores the new value in the variable `newTotalAssets` (link), then updates `totalAssets` to `newTotalAssets` (link):

```
totalAssets' == newTotalAssets
```

As for `totalSupply`, there are two cases:

1. If `newTotalAssets > oldTotalAssets` (where `oldTotalAssets` is the value of `totalAssets` before the update), then it converts `profit = newTotalAssets - oldTotalAssets` to shares and mints that amount of shares to the `DragonRouter` (link), increasing the total supply.
2. If `newTotalAssets <= oldTotalAssets`, then it calculates `loss = oldTotalAssets - newTotalAssets` (link). If `loss == 0`, then `totalSupply` is not updated. Otherwise (link), `loss` is converted to shares (link) and those shares are burned from the `DragonRouter`, as long as there are enough shares to be burned (link).

In case (1), we have that

```
totalSupply' == totalSupply + _convertToShares(profit)
```

Since we assume that `totalSupply == totalAssets` holds before the update, `_convertToShares(profit) == profit`. Therefore,

```
totalSupply' == totalSupply + _convertToShares(profit)
             == totalSupply + profit
             == totalSupply + newTotalAssets - oldTotalAssets
             == totalSupply + newTotalAssets - totalAssets
             == totalAssets + newTotalAssets - totalAssets
             == newTotalAssets
```

Therefore, `totalSupply' == totalAssets'`, and the invariant holds after the call.

In case (2), there are three sub-cases:

1. If `loss == 0`, then `totalSupply' == totalSupply`. But if `loss == 0` this also means `newTotalAssets == oldTotalAssets`, and therefore `totalAssets' == totalAssets`. Therefore, the invariant is preserved.
2. If `loss > 0` and `_convertToShares(loss) > balances[dragonRouter]`, then the invariant is no longer required to hold. Therefore, we can ignore this case.
3. If `loss > 0` and `_convertToShares(loss) <= balances[dragonRouter]`, then `totalSupply` is updated to `totalSupply - _convertToShares(loss)`. Again, since `totalSupply == totalAssets`, we get `_convertToShares(loss) == loss`, and therefore

```
totalSupply' == totalSupply - _convertToShares(loss)
             == totalSupply - loss
             == totalSupply - (oldTotalAssets - newTotalAssets)
             == totalSupply - oldTotalAssets + newTotalAssets
             == totalSupply - totalAssets + newTotalAssets
             == totalAssets - totalAssets + newTotalAssets
             == newTotalAssets
```

Therefore, `totalSupply' == totalAssets'`, and the invariant holds in this case as well.

`deposit` and `depositWithLockup`

These two functions receive a parameter `assets` and update `totalAssets` and `totalSupply` as follows via `TokenizedStrategy._deposit` (link):

```
totalAssets' == totalAssets + assets
totalSupply' == totalSupply + shares
```

where `shares = _convertToShares(assets)` (link). Since `totalSupply == totalAssets`, this means `shares == assets`. Therefore,

```
totalAssets' == totalAssets + assets
totalSupply' == totalAssets + assets
```

Therefore, `totalSupply' == totalAssets'`.

`mint` and `mintWithLockup`

These two functions receive a parameter `shares` and convert it to `assets = _convertToAssets(shares)` (link). They then pass `assets` as input to `DragonTokenizedStrategy._deposit` (link), which makes the same update as `deposit` and `depositWithLockup`. Therefore, the same reasoning above holds, and `totalSupply' == totalAssets'`.

`withdraw` and `redeem`

Both of these functions call `TokenizedStrategy._withdraw` with parameters `shares` and `assets`:

- `withdraw` receives `assets` as an argument and calculates `shares = _convertToShares(assets)` (link).
- `redeem` receives `shares` as an argument and calculates `assets = _convertToAssets(shares)` (link).

In either case, assuming `totalSupply == totalAssets`, we have that `shares == _convertToShares(assets) == assets` and `assets == _convertToAssets(shares) == shares`.

`TokenizedStrategy._withdraw` updates `totalShares` by calling the `TokenizedStrategy._burn` function, which subtracts `shares` from the `totalSupply` (link):

```
totalSupply' == totalSupply - shares
```

It also updates `totalAssets` as follows (link):

```
totalAssets' == totalAssets - (assets' + loss)
```

where `assets'` and `loss` depend on three cases (link):

1. If `idle >= assets`, where `idle` is the amount of idle funds currently sitting in the contract, then `assets' = assets` and `loss = 0`.
2. Otherwise, it calls `BaseStrategy.freeFunds` to try to free the difference from the underlying strategy (link). If it's successful, then again `assets' = assets` and `loss = 0`.

3. If it's not able to free enough funds, then `loss = assets - idle'`, where `idle'` is the new amount of idle funds after freeing (link), and `assets' = idle'` (link).

In cases (1) and (2),

```
totalAssets' == totalAssets - (assets' + loss)
             == totalAssets - (assets + 0)
             == totalAssets - assets
```

In case (3),

```
totalAssets' == totalAssets - (assets' + loss)
             == totalAssets - (idle' + assets - idle')
             == totalAssets - assets
```

In either case, since `totalSupply == totalAssets` and `shares == assets`, we have that `totalSupply' == totalAssets'`.

Invariant Analysis 2: User Balance Correctness

Invariant Statement

Invariant: For any user, `balances[user] <= totalSupply`.

Invariant Base Case

Initially, `balances[user] == 0` and `totalSupply == 0`. Therefore, the invariant holds.

Invariant Step

We show that, for every external function of the `DragonTokenizedStrategy` contract, if `balances[user] <= totalSupply` before the function is called, it will still be the case after it returns.

The only functions that modify `balances[user]` are `TokenizedStrategy._mint` and `TokenizedStrategy._burn`, which add or subtract an `amount` to it, respectively. Both functions add or subtract the same `amount` to `totalSupply`. Therefore, if `balances[user] <= totalSupply` before execution, it will still be the case after.

Since there are no other functions that modify `totalSupply`, the invariant is always preserved.

Invariant Analysis 3: Lockup

Invariant Statement

Invariant: A user who hasn't rage quit cannot withdraw funds if their lockup hasn't expired.

Proof

The only external functions that allow a user to withdraw funds are `withdraw` and `redeem`, which revert on the condition `assets > _maxWithdraw(S, _owner)` (link) and `shares > _maxRedeem(S, _owner)` (link), respectively.

If a user has not rage quit and their lockup hasn't expired (`block.timestamp < unlockTime`), then `_userUnlockedShares` returns `0` (link). The output of `_maxRedeem` is bounded by `_userUnlockedShares` (link). Similarly, the output of `_maxWithdraw` is bounded by `_userUnlockedShares` converted to assets, which will likewise be `0` (link). As a result, both `_maxRedeem` and `_maxWithdraw` will return `0` in this case. Therefore, `withdraw` or `redeem` will revert if this user tries to withdraw any value above 0.

Note that even if the user tries to withdraw 0 shares or assets, `withdraw` (link) and `redeem` (link) will still revert.

Invariant Analysis 4: `unlockTime` Safety

Invariant Statement

Invariant: A user other than the `target` address cannot have their `unlockTime` changed (increased or decreased) before it expires, except by initiating a rage quit.

Proof

Aside from `initiateRageQuit`, the only function that modifies `unlockTime` is the internal function `_setOrExtendLockup`. `_setOrExtendLockup` is only called by `DragonTokenizedStrategy._deposit`, which reverts on the condition (link)

```
_balanceOf(S, receiver) > 0 && IBaseStrategy(address(this)).target() != address(receiver)
```

We show that if `unlockTime` hasn't expired (`unlockTime > block.timestamp`), then `balances[receiver] > 0`, and therefore `DragonTokenizedStrategy._deposit` must revert before getting to `_setOrExtendLockup` as long as `receiver != target`.

Note first that if `isRageQuit == true`, then both `initiateRageQuit` (link) and `DragonTokenizedStrategy._deposit` (link) revert. Therefore, we only need to consider the case where `isRageQuit == false`. Since `isRageQuit` can never be set back to `false` after being set to `true`, we can assume that `initiateRageQuit` was never called by this user.

If `unlockTime > block.timestamp` and the user has not rage quit, `unlockTime` must have been previously set by `_setOrExtendLockup`. This can only have happened after executing `_processDeposit` (link), which calls `TokenizedStrategy._deposit` (link), which calls `TokenizedStrategy._mint` (link), which increases `balances[receiver]` by `shares` (link). Since `DragonTokenizedStrategy._deposit` reverts if `shares == 0` (link), it must be the case that `shares > 0`, and therefore `balances[receiver] > 0`.

All that's left, therefore, is proving that `balances[receiver]` cannot have decreased before the next call of `_setOrExtendLockup`. The only way to decrease

`balances[receiver]` is via `_burn` (link), which, unless `receiver` is the `DragonRouter`, is only called by `_withdraw`, which itself can only be called by `withdraw` / `redeem`. However, both of these functions revert (link) if

```
block.timestamp < lockup.unlockTime && !lockup.isRageQuit
```

which is the case for us. Therefore, `withdraw` / `redeem` cannot have been executed since the last time `balances[receiver]` was set by `_setOrExtendLockup`, and therefore `balances[receiver]` must still be greater than 0. As a result, `DragonTokenizedStrategy._deposit` must revert without changing the user's `unlockTime`.

Note that if the `receiver` *is* the `DragonRouter`, `DragonTokenizedStrategy._deposit` also reverts (link), therefore the `unlockTime` can never be set.

After Fixing Revert Condition for Finding A07

Note that to address finding A07, the revert condition in `DragonTokenizedStrategy._deposit` will need to include the condition `lockupDuration > 0`. Therefore, `_setOrExtendLockup` will still be able to be executed in the case `lockupDuration == 0`, even if `balances[receiver] > 0`. However, both branches of `_setOrExtendLockup` return without updating `unlockTime` if `lockupDuration == 0`. Therefore, the invariant is still maintained.

Invariant Analysis 5: Rage Quit Safety

Invariant Statement

Invariant: If a user has rage quit, they can't receive more assets.

Proof

If a user calls `initiateRageQuit`, `isRageQuit` is set to `true`. Since no function sets `isRageQuit` back to `false`, it remains `true`.

The only external functions that allow a user other than the `DragonRouter` to receive assets are the ones that call `DragonTokenizedStrategy._deposit` (namely, `deposit`, `depositWithLockup`, `mint` and `mintWithLockup`). Since this function reverts when `isRageQuit` is `true` for the receiver (link), it's impossible for a user who has rage quit to receive more assets.

Kontrol Formal Verification

At Runtime Verification, we specialize in building rigorous, mathematically grounded proofs to ensure that smart contracts behave as intended under all possible inputs and scenarios. Our tool, Kontrol, is designed to integrate seamlessly with Solidity-based projects, enabling developers to write property-based tests in Solidity and leverage symbolic execution to verify them.

The first step in formally verifying a protocol is identifying its main invariants, which must hold under all circumstances. In the case of Octant V2 Core the main invariant is: **"Separate yield from principal: Users deposit assets and receive 1:1 vault tokens and the asset-to-share ratio should remain constant (1:1)."**

Proving with Kontrol

After identifying the protocol's main invariant, the next step was to translate it into Solidity and write Kontrol proofs that ensure the invariant holds in every possible state. The asset-to-share ratio remains constant (1:1) if the total supply of the protocol remains equal to the total assets. This was translated into Solidity as follows:

```
function principalPreservationInvariant(Mode mode) internal view {
    uint256 totalSupply = strategy.totalSupply();
    uint256 totalAssets = strategy.totalAssets();

    if (mode == Mode.Assume) {
        vm.assume(totalSupply == totalAssets);
    } else {
       assert(totalSupply == totalAssets);
    }
}
```

To formally prove that some invariant holds it is necessary to prove that:

1. The invariant holds in the initial state

This is trivially true because when contracts are deployed, `strategy.totalSupply() == 0` and `strategy.totalAssets() == 0`; therefore, `totalSupply == totalAssets` is true.

2. For every pre-state, if the invariant holds in that state, then after a state transition, the invariant must still hold in the post-state

To prove this, we must prove that for every state-changing function, if the invariant holds before execution, it still hold afterward. Therefore, for every (external or public) function that can potentially change the state, there is a test with the following structure:

```
function testStrategyFunction(args) public {
    principalPreservationInvariant(Mode.Assume);

    contract.publicOrExternalFunction(params);

    principalPreservationInvariant(Mode.Assert);
}
```

Symbolic Execution

When running the above test with Kontrol, the `args` are symbolic, which means that we prove the invariant holds for every possible input the function can take, whereas, with Foundry, we only prove that the invariant holds for the specific concrete `args` the test runs with.
Note that we still need to define what "For every possible state" means. Foundry allows us to define an initial pre-state (with the `setUp` function) to run the tests. However, that initial pre-state is concrete, which means that we would be proving the above invariant to a particular pre-state. In Kontrol, we have the `symbolicStorage` cheatcode, which makes the storage of a given contract fully symbolic. Therefore, when setting up the pre-state state to run the proofs, after deploying the strategy contract, we call `kevm.symbolicStorage(address(strategy))`, making the initial pre-state as abstract as possible.

After the symbolic `setUp`, if the specified proofs pass with Kontrol, we prove that the invariant holds in every possible state of the protocol.

Kontrol Proofs

To prove the main invariant, we defined the following tests with the above structure:

- `testDeposit`
- `testDepositWithLockup`
- `testMint`
- `testMintWithLockup`
- `testWithdraw`
- `testRedeem`
- `testTend`

With respect to the `report` function we can only prove that the invariant holds if we can assume there is no loss in the protocol, i.e., the dragon Router shares are enough to cover the loss of assets. Therefore, we wrote two proofs:

- `testReport` - proves the invariant holds when assuming there is no loss in the protocol
- `testReportWithLoss` - if there is a loss in the protocol, then `strategy.totalSupply() > strategy.totalAssets()`

We also proved that a user cannot withdraw funds if their lockup hasn't expired:

- `testWithdrawRevert`
- `testRedeemRevert`

Additionally, we specified tests for access control functions:

- `testSetPendingManagement`
- `testSetKeeper`
- `testSetEmergencyAdmin`

Finally, for the `deposit` and `mint` functions, we also asserted some state-expected changes after the function execution.

Reproducing the Proofs

To reproduce the results of this verification locally, follow the steps below:

1. Install Kontrol

```
bash <(curl https://kframework.org/install)
kup install kontrol
```

2. Run the proofs

```
export FOUNDRY_PROFILE=kprove
kontrol build
kontrol prove
```

The `kontrol.toml` file contains a set of flags to run Kontrol with. Refer to the Kontrol documentation if you want to learn more about Kontrol options.

Identifying issues

When running the above tests with Kontrol, `testWithdraw` failed, and we could identify finding A02. We defined `testWithdrawWithLossReverts` to show that if there is a loss, then the function is always expected to revert. If the recommendation for finding A02 is implemented, then `testWithdraw` passes and `testWithdrawWithLossReverts` shour fail.
When trying to prove `testRedeem`, we identified finding A06. We then defined `testMaxRedeemAlwaysReverts`, which shows that in case there is a loss, and `strategy.totalSupply() > strategy.totalAssets()`, then the function `_maxRedeem` always reverts. The implication is that if there is a loss, the function `redeem` will always revert since it calls the internal function `redeem`. Once the `_maxRedeem` function is fixed, the proof `testRedeem` should pass, and the proof `testMaxRedeemAlwaysReverts` should fail.

Kontrol Report

When running `kontrol prove`, the optional flag `xml-test-report` can be typed, which will generate an XML report with relevant information about the proof execution - status, execution times, failure information, etc.
Alternatively, it is possible to set up our KaaS tool to run the tests and analyse the generated results. This option is very useful for integration with CI because it allows specifying a commit hash to run the tests alongside run options. Here is the link for the document with the proof status after implementing the suggestions for fixing the issues, which can also be found attached to this report in the Appendix.

Findings

The following are findings that contradict the desired behavior of the protocol, violate invariants, impact user experience or may lead to security vulnerabilities.

A01: A malicious `target` can `deposit` or `mint` shares to a receiver without transferring any `assets` to the strategy contract

Severity: High | Difficulty: High | Recommended Action: Fix Code | Not addressed by client

The `_deposit` function in the `TokenizedStrategy.sol` calls the `execTransactionFromModule` function on the `target`.

runtimeverification/_audits_golemfoundation_octant-v2-core/src/dragons/vaults/TokenizedStrategy.sol

Line 752 to 761 in f65cac2

```
752         } else {
753             if (
754                 IAvatar(target).execTransactionFromModule(
755                     address(_asset),
756                     0,
757                     abi.encodeWithSignature("transfer(address,uint256)", address(this), assets),
758                     Enum.Operation.Call
759                 ) == false
760             ) revert TokenizedStrategy__TransferFailed();
761         }
```

A safe `target` would execute this function by making a call on the address given as a parameter with the call data given as a parameter. In this case it would call `_asset.transfer(strategyContractAddress, assets)` resulting in a transfer of `assets` amount from the `target` to the strategy contract (see implementation of a Safe owner here).

Scenario

A malicious `target` can have its own implementation of `execTransactionFromModule` as for example:

```
function execTransactionFromModule(
        address to,
        uint256 value,
        bytes memory data,
        Enum.Operation operation
) public virtual returns (bool success) {
        return true;
}
```

With this implementation, the `_deposit` function in the `TokenizedStrategy.sol` would proceed, minting the `shares` to the receiver without having received any `assets`.

Recommendation

1. Store the balance of the strategy contract prior to calling `execTransactionFromModule`.
2. Ensure that after this function call, the balance of the strategy contract has increased by the `assets` amount.

```
            } else {
+               uint256 previousBalance = _asset.balanceOf(address(this));
                require(
                    IAvatar(S.owner).execTransactionFromModule(
                        address(_asset),
                        0,
                        abi.encodeWithSignature("transfer(address,uint256)",
address(this), assets),
                        Enum.Operation.Call
                    ) == true,
                    "ERC4626: deposit more than max"
                );
+               require(_asset.balanceOf(address(this)) == previousBalance + assets);
```

```
}
```

A02: If the strategy has not enough funds to cover a withdraw, the `withdraw` function reverts even if the user specifies a valid `maxLoss`

Severity: Medium | Difficulty: High | Recommended Action: Fix Code | Not addressed by client

The `_withdraw` function in the `TokenizedStrategy.sol` contract first checks if the strategy has enough `idle` (with idle being the `_asset` balance of the strategy) to cover the withdrawal request. If not, the function will free funds from the `YIELD_SOURCE` contract.

runtimeverification/_audits_golemfoundation_octant-v2-core/src/dragons/vaults/TokenizedStrategy.sol

Line 806 to 814 in f65cac2

```
806         uint256 idle = address(_asset) == ETH ? address(this).balance :
_asset.balanceOf(address(this));
807         uint256 loss;
808         // Check if we need to withdraw funds.
809         if (idle < assets) {
810             // Tell Strategy to free what we need.
811             unchecked {
812                 IBaseStrategy(address(this)).freeFunds(assets - idle);
813             }
814
```

The implementation of `freeFunds` just calls the internal function `_freeFunds` which is implemented as follows:

runtimeverification/_audits_golemfoundation_octant-v2-core/src/dragons/modules/YearnPolygonUsdcStrategy.sol

Line 67 to 69 in f65cac2

```
67      function _freeFunds(uint256 _amount) internal override {
68          IStrategy(YIELD_SOURCE).withdraw(_amount, address(this), address(this));
69      }
```

If the funds in the `YIELD_SOURCE` contract are not enough to cover the withdrawal, then the function reverts (See implementation of `YIELD_SOURCE.withdraw` in polygonscan, which requires that the amount to be withdrawn is less than the maximum withdrawable amount for the specific owner).
Therefore, even if the user has provided a valid `maxLoss` that it is willing to support, it does not receive the amount of assets within the `maxLoss` because the computation of the remaining assets to transfer is never executed:

runtimeverification/_audits_golemfoundation_octant-v2-core/src/dragons/vaults/TokenizedStrategy.sol

Line 811 to 831 in f65cac2

```
811             unchecked {
812                 IBaseStrategy(address(this)).freeFunds(assets - idle);
813             }
814
815             // Return the actual amount withdrawn. Adjust for potential under
withdraws.
816             idle = address(_asset) == ETH ? address(this).balance :
_asset.balanceOf(address(this));
817
818             // If we didn't get enough out then we have a loss.
819             if (idle < assets) {
820                 unchecked {
821                     loss = assets - idle;
822                 }
823                 // If a non-default max loss parameter was set.
824                 if (maxLoss < MAX_BPS) {
825                     // Make sure we are within the acceptable range.
826                     if (loss > (assets * maxLoss) / MAX_BPS) revert
TokenizedStrategy__TooMuchLoss();
827                 }
828                 // Lower the amount to be withdrawn.
829                 assets = idle;
830             }
831         }
```

Recommendation

In the internal `_freeFunds` function, check the maximum withdrawable amount in the `YIELD_SOURCE` and only withdraw the minimum between the amount to cover the user withdrawal and the maximum withdrawable amount:

```
function _freeFunds(uint256 _amount) internal override {
        uint256 _withdrawAmount = Math.min(_amount,
IStrategy(YIELD_SOURCE).maxWithdraw(address(this)));
        IStrategy(YIELD_SOURCE).withdraw(_withdrawAmount, address(this), address(this));
}
```

A03: Rage quit may increase user's `unlockTime`

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Not addressed by client

In `DragonTokenizedStrategy.initiateRageQuit`, the lockup is reset to `RAGE_QUIT_COOLDOWN_PERIOD` regardless of the remaining time in the user's original lockup.:

runtimeverification/_audits_golemfoundation_octant-v2-core/src/dragons/vaults/DragonTokenizedStrategy.sol

Line 197 in f65cac2

```
197          lockup.unlockTime = block.timestamp +
_strategyStorage().RAGE_QUIT_COOLDOWN_PERIOD;
```

This means that if the user had less than `RAGE_QUIT_COOLDOWN_PERIOD` left in their lockup, the time they would have to wait to fully withdraw their funds would be increased with the rage quit.

Recommendation

Update the code to keep the previous `unlockTime` if it is less than `block.timestamp + RAGE_QUIT_COOLDOWN_PERIOD`.

A04: Calculation of unlocked shares during a rage quit breaks if user withdraws fewer shares than the maximum

Severity: Low | Difficulty: Low | Recommended Action: Fix Design | Not addressed by client

During a rage quit, a user should be able to withdraw their assets proportionally to the amount of time that has passed since the rage quit started. For example, if half of the rage quit period has passed, half of the user's assets should be unlocked for withdrawal.

However, in practice, this calculation measures the proportion of time elapsed since the last withdrawal, rather than since the start of the rage quit period. This is fine if the user always withdraws the maximum amount that has been unlocked so far, but if at any point the user withdraws *less* than this amount, the calculation will start to underestimate how many shares have been unlocked from this point on.

For example, a user starting with 100 shares would be expected to have 50 of them unlocked after half of the rage quit period has passed. However, if they withdraw 10 shares at the 25% mark, they will only be able to withdraw 30 additional shares at the 50%, rather than 40. See the example scenario below for details.

Scenario

1. At timestamp `T_0`, a user with 100 shares calls `initiateRageQuit`. The lockup information is set as follows:

```
lockupTime = T_0
unlockTime = T_0 + RAGE_QUIT_COOLDOWN_PERIOD
lockedShares = 100
isRageQuit = true
```

2. At timestamp `T_1 = T_0 + (RAGE_QUIT_COOLDOWN_PERIOD / 4)` (that is, a quarter of the way through the rage quit period), the user calls `redeem`, redeeming 10 shares.

This is possible since 10 is less than the 25 shares currently unlocked, as per the calculation in `_userUnlockedShares`. `redeem` updates the lockup as follows:

```
lockedShares = 90
lockupTime = T_1
```

3. At timestamp `T_2 = T_0 + (RAGE_QUIT_COOLDOWN_PERIOD / 2)`, the user calls `redeem` again to redeem 40 shares. This should be possible since half of the rage quit period has passed, and therefore the user should have unlocked 50 shares (10 redeemed at `T_1` + 40 redeemed at `T_2`). However, the calculation in `_userUnlockedShares` gives us

```
timeElapsed
= block.timestamp - lockupTime
= T_2 - T_1
= (T_0 + (RAGE_QUIT_COOLDOWN_PERIOD / 2)) - (T_0 + (RAGE_QUIT_COOLDOWN_PERIOD /
4))
= (RAGE_QUIT_COOLDOWN_PERIOD / 2) - (RAGE_QUIT_COOLDOWN_PERIOD / 4)
= RAGE_QUIT_COOLDOWN_PERIOD / 4

unlockTime - lockupTime
= (T_0 + RAGE_QUIT_COOLDOWN_PERIOD) - T_1
= (T_0 + RAGE_QUIT_COOLDOWN_PERIOD) - (T_0 + (RAGE_QUIT_COOLDOWN_PERIOD / 4))
= RAGE_QUIT_COOLDOWN_PERIOD - (RAGE_QUIT_COOLDOWN_PERIOD / 4)
= 3 * RAGE_QUIT_COOLDOWN_PERIOD / 4

unlockedPortion
= (timeElapsed * balance) / (unlockTime - lockupTime)
= ((RAGE_QUIT_COOLDOWN_PERIOD / 4) * 90) / (3 * RAGE_QUIT_COOLDOWN_PERIOD / 4)
= 90 / 3
= 30
```

As a result the user is only able to unlock 30 shares, rather than the expected 40.

Recommendation

Adjust how the unlocked shares are calculated to account for this possibility. For example, if the values of `lockedShares` and `lockupTime` remain the same throughout

the rage quit instead of being updated with each withdrawal, they can be used to calculate the proportion of shares that have been unlocked so far:

```
timeElapsed = block.timestamp - lockupTime
unlockedPortion = (timeElapsed * lockedShares) / (unlockTime - lockupTime)
```

We can then subtract the shares that have already been withdrawn to calculate the maximum withdrawal amount:

```
sharesPreviouslyWithdrawn = lockedShares - balances[user]
maxWithdrawalAmount = unlockedPortion - sharesPreviouslyWithdrawn
```

This remains correct as long as the user’s balance changes only through withdrawals.

A05: `MINIMUM_LOCKUP_DURATION` is not accepted as a valid lockup duration

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Not addressed by client

The name `MINIMUM_LOCKUP_DURATION` seems to suggest that a lockup that is greater or equal to it should be valid, but `_setOrExtendLockup` doesn't accept `MINIMUM_LOCKUP_DURATION` as a lockup:

runtimeverification/_audits_golemfoundation_octant-v2-core/src/dragons/vaults/DragonTokenizedStrategy.sol

Line 69 to 71 in f65cac2

```
69              if (lockupDuration <= _strategyStorage().MINIMUM_LOCKUP_DURATION) {
70                  revert DragonTokenizedStrategy__InsufficientLockupDuration();
71              }
```

Recommendation

Replace `<=` with `<` in the condition above.

A06: Internal function `_maxRedeem` in `DragonTokenizedStrategy` always reverts if `S.totalSupply > S.totalAssets`

Severity: Low | Difficulty: High | Recommended Action: Fix Code | Not addressed by client

If the protocol suffers a loss and the DragonRouter shares are not enough to cover it, then the protocol gets to a state S where `S.totalSupply > S.totalAssets` (see this Kontrol proof).
If the protocol gets to a state where `S.totalSupply > S.totalAssets`, then the function `_maxRedeem` in `DragonTokenizedStrategy` always reverts because the `mulDiv` function reverts with a `Math.MathOverflowedMulDiv` error. The following Kontrol proof can ensure that:

runtimeverification/_audits_golemfoundation_octant-v2-core/test/kontrol/YearnPolygonUsdcStrategyTest.k.sol

Line 383 to 391 in 223d665

```
383     function testMaxRedeemAlwaysReverts(address _owner) public {
384         UserInfo memory user = setupSymbolicUser(_owner);
385
386         vm.assume(strategy.totalSupply() > strategy.totalAssets());
387         vm.assume(strategy.totalAssets() > 0);
388
389         vm.expectRevert(abi.encodeWithSelector(Math.MathOverflowedMulDiv.selector));
390         strategy.maxRedeem(_owner);
391     }
```

Therefore, any call to the `redeem` function would revert since this function calls the internal `_maxRedeem` function.

Recommendation

Change the implementation of `_maxRedeem` in `DragonTokenizedStrategy.sol` to the following:

```
function _maxRedeem(StrategyData storage S, address _owner) internal view override
returns (uint256 maxRedeem_) {
    // Get the max the owner could withdraw currently.
    maxRedeem_ = IBaseStrategy(address(this)).availableWithdrawLimit(_owner);
    if (maxRedeem_ == type(uint256).max) {
        maxRedeem_ = _userUnlockedShares(S, _owner);
    }
    else {
        maxRedeem_ = Math.min(
            // Can't redeem more than the balance.
            _convertToShares(S, maxRedeem_, Math.Rounding.Floor),
            _userUnlockedShares(S, _owner)
        );
    }
}
```

A07: `_deposit` reverts when the user already has assets, even if the lockup would not get extended

Severity: Medium | Difficulty: Low | Recommended Action: Fix Code | Not addressed by client

The internal function `_deposit` in `DragonTokenizedStrategy` reverts if the receiver already has funds deposited in their name (unless they are the `target` contract):

runtimeverification/_audits_golemfoundation_octant-v2-core/src/dragons/vaults/DragonTokenizedStrategy.sol

Line 356 to 358 in f65cac2

```
356         if (_balanceOf(S, receiver) > 0 && IBaseStrategy(address(this)).target() != address(receiver)) {
357             revert DragonTokenizedStrategy__ReceiverHasExistingShares();
358         }
```

This is intended to prevent a receiver's lockup from being extended by a later deposit against their will. However, `_deposit` is called not only by `depositWithLockup` and `mintWithLockup`, but also `deposit` and `mint`, the alternate version of these functions with `lockupDuration = 0`. This means that `deposit` and `mint` will revert if the user already has assets, even though they wouldn't cause the lockup to get extended. As a consequence, a user who already has assets cannot receive additional deposits, even when their lockup duration wouldn't be affected.

Recommendation

Add the condition `lockupDuration > 0` to the check above, to prevent reverting on the case when the lockup would not get extended.

A08: `lockedShares` is not updated if `lockupDuration` is 0 and there is no previous lockup

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Not addressed by client

In `_setOrExtendLockup`, if the user has no current lockup and the new `lockupDuration` is 0, the `lockedShares` variable is not updated. This doesn't significantly impact the operation of the protocol since `lockedShares` is not used until the user rage quits (and in this case it will anyway be recalculated by `initiateRageQuit`), but it might provide users with incorrect information (when `getUserLockupInfo` is called, for example).

Recommendation

Add the following update to this branch, as in the other non-reverting branches:

```
lockup.lockedShares = totalSharesLocked;
```

Informative Findings

The following are findings that relate to documentation and best coding practices.

B01: Distinct `_deposit` and `_mint` functions in `TokenizedStrategy` and `DragonTokenizedStrategy` sharing the same name

Severity: Informative | Recommended Action: Fix Code | Not addressed by client

The `_deposit` and `_mint` internal functions in `DragonTokenizedStrategy` share the same name as internal functions in the `TokenizedStrategy` contract it inherits from. This may cause confusion since their behaviors are logically distinct, and even though they take different parameters, it may not be immediately clear for someone inspecting the code which one is being called.

For example, `DragonTokenizedStrategy._mint` calls `DragonTokenizedStrategy._deposit`, but `DragonTokenizedStrategy._processDeposit` calls `TokenizedStrategy._deposit` (which itself calls `TokenizedStrategy._mint`).

B02: Broken link for Base Strategy Flow Diagram

Severity: Informative | Recommended Action: Document Prominently | Not addressed by client

The link to the Base Strategy Flow Diagram in `DragonBaseStrategy.md` leads to a nonexistent file.

B03: Return value of `_setOrExtendLockup` is ignored

Severity: Informative | Recommended Action: Fix Code | Not addressed by client

The internal function `_setOrExtendLockup` in `DragonTokenizedStrategy` returns a `uint256`, but this return value is not used in the `_deposit` function, which is the only place where it is called. Additionally, this return value is inconsistent: in most cases it will be the user's `lockedShares` after the update, but if the user has no lockup and the new lockup duration is 0, it returns 0 regardless of the value of `lockedShares`.

B04: `TokenizedStrategy` makes assumptions about how functions will be implemented in derived contracts

Severity: Informative | Recommended Action: Document Prominently | Not addressed by client

Some functions in `TokenizedStrategy` are implemented in a way that assumes that other `virtual` functions that don't have a provided implementation and must instead be implemented in derived contracts will behave in a certain way:

- `previewDeposit`, `previewMint`, `previewWithdraw` and `previewRedeem` assume the direction of rounding of the corresponding functions when converting from shares to assets or vice-versa.
- `shutdownStrategy` assumes that `_maxDeposit` and `_maxMint` will be called by `deposit` and `mint` in the derived contracts, so that the `shutdown` flag will be checked.

These assumptions should be documented so that anyone inheriting from `TokenizedStrategy` will know to implement these functions in a way that matches the expected behavior.

Appendix: KaaS Formal Verification Report

Below you will find the results for the tests described in the "Kontrol Formal Verification" section, executed on Runtime Verification's online platform KaaS. This report can also be accessed at https://kaas.runtimeverification.com/job/702129d9-58ed-4710-9bd5-73401c2a7d16/xml-report.