Security Review Report
NM-0070 Dyve

NETHERMIND

(Mar. 15, 2023)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind of the Dyve protocol. Dyve is an NFT lending platform that allows ERC721 and ERC1155 compatible tokens to be lent out for value speculation (shorting) or access to NFT utility. Borrowers pay a fee and lock an amount of collateral specified by the lender into the contract, which remains until the borrower returns an NFT from the same collection or fails to return an NFT within the lender's specified time frame. At this point, the lender can claim the borrower's collateral. Orders are signed and submitted off-chain and are fulfilled on-chain.

The audited code consists of 399 lines of Solidity with code coverage of 99.06%. This audit was supported by documentation accessible from the Dyve Notion Page, which explains order types, business logic, and a technical overview. The audit was performed using a combination of the following: **a)** Manual analysis of the codebase; **b)** Automated analysis tools; and, **c)** Custom tests.

Along this document, we report 14 points of attention, where one is classified as high, one is classified as medium, and three are classified as low. The other nine points of attention are classified as informational or best practices. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 presents assumptions followed during the audit. Section 4 summarizes the issues. Section 5 presents the system overview. Section 6 discusses the risk rating methodology adopted for this audit. Section 7 details the issues. Section 8 discusses complimentary fuzzing efforts. Section 9 discusses the documentation provided by the client for this audit. Section 10 presents the compilation, coverage, and tests. Section 11 concludes the document.

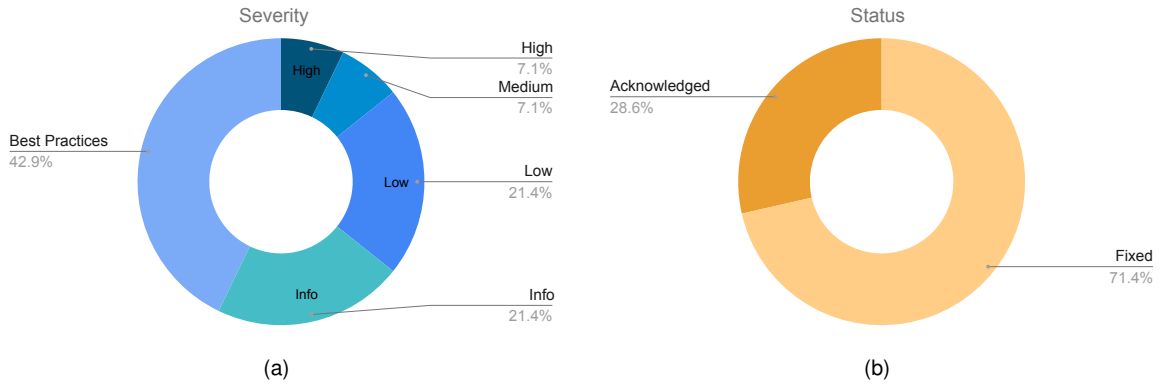

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (1), **Low** (3), **Undetermined** (0), **Informational** (3), **Best Practices** (6). **Distribution of status: Fixed** (10), **Acknowledged** (4), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jan. 27, 2023
Response from Client	Mar. 8, 2023
Final Report	Mar. 15, 2023
Methods	Manual Review, Automated Analysis
Repository	github.com/Dyve-co/Contracts
Commit Hash (Initial Audit)	a0f7d06e1bf994825c808ebd9abee230afe3e746
Commit Hash (Reaudit)	49ed8ae1b8a03a63783213cf15f1c20b07481e86
Documentation	Notion page
Documentation Assessment	Medium
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/Dyve.sol	352	98	27.8%	68	518
2	contracts/libraries/OrderTypes.sol	47	6	12.8%	6	59
	Total	**399**	**104**	**28.6%**	**74**	**577**

3 Assumptions

The prepared security review is based on the following assumptions:

- The NFT tokens used in the protocol meet the `ERC721` or `ERC1155` specification.
- The whitelisted currency `ERC20` tokens do not have behaviors that can readjust balances, such as fee-on-transfer or rebasing tokens.

4 Summary of Issues

	Finding	Severity	Update
1	Reservoir-flagged NFTs can be offered by a lender	High	Fixed
2	Third party oracle dependency	Medium	Fixed
3	`Resevoir` oracle message replay attack	Low	Acknowledged
4	Protocol can receive no fees due to integer division	Low	Acknowledged
5	`premiumCollection` and `premiumTokenId` are not part of the signed order	Low	Acknowledged
6	Lender can borrow from themselves	Info	Acknowledged
7	Missing input validation for order duration	Info	Fixed
8	Non-existent orders have a status of `BORROWED`	Info	Fixed
9	Hardcoded values for the Interface ID	Best Practices	Fixed
10	Mapping entry can be saved to memory	Best Practices	Fixed
11	The `nonReentrant` modifier should be added to all order management functions	Best Practices	Fixed
12	Use of arbitrary number	Best Practices	Fixed
13	`_transferERC20` and `_transferETH` may transfer with no amount	Best Practices	Fixed
14	Code does not match the specification	Best Practices	Fixed

5 System Overview

The Dyve protocol allows for NFT lending, where either a lender can offer an NFT or a borrower can request an NFT at a specified collateral and fee price. These orders are created and signed off-chain. Orders are fulfilled on-chain, where the signed order is passed as an argument. When an order is fulfilled, the collateral and fees are paid by the borrower, and the borrower will receive the lender's NFT. The borrower is expected to return an NFT from the same collection within some specified timeframe, at which point their collateral would be returned. If a borrower does not return an NFT, the lender can claim the borrower's collateral. When a borrower returns an NFTs to the protocol, a Reservoir API is used to prevent returning any stolen NFTs. Fig. 2 below describes the borrow and lend flow for the Dyve protocol.

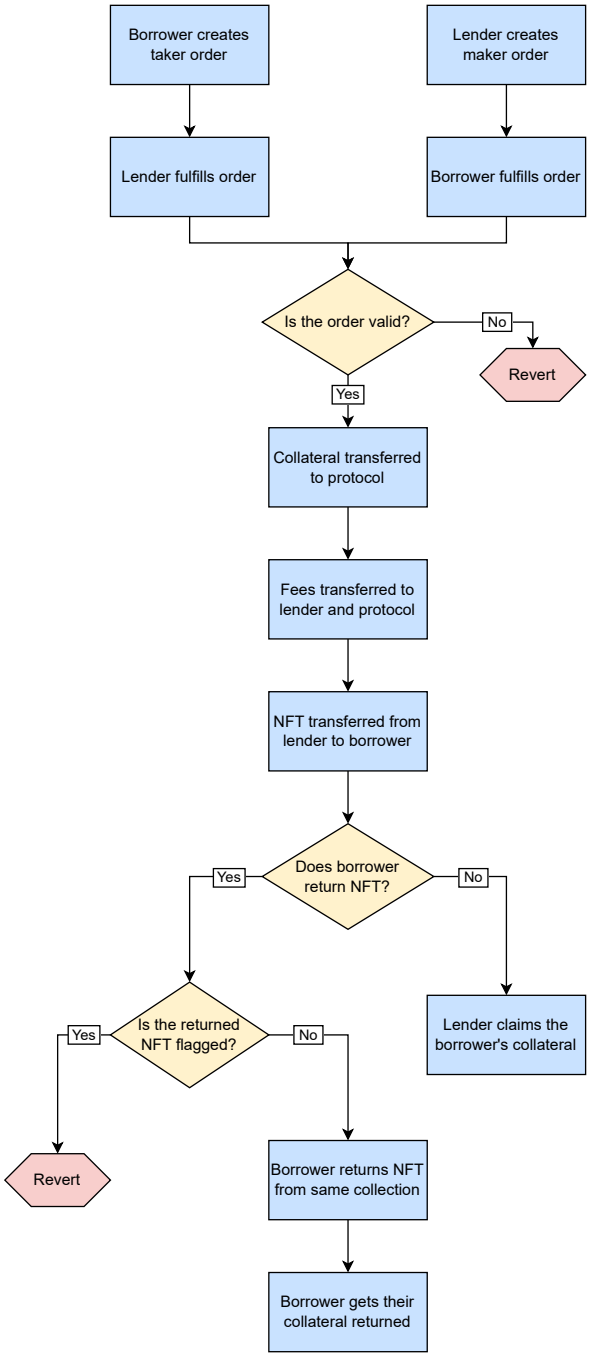

Fig. 2: Borrow and lend protocol flow

6 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to: Data/state integrity, loss of availability, financial loss, reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

7 Issues

7.1 [High] Reservoir-flagged NFTs can be offered by a lender

File(s): contracts/Dyve.sol

Description: When closing a position, a borrower must ensure that Reservoir does not flag the NFT being returned. This check ensures that stolen NFTs cannot be used to close the position. Although the borrower has their NFT checked on return, the same check is not done for a lender when fulfilling the position. If a lender gets their flagged NFT order fulfilled by a borrower, the lender can now guarantee that they will either receive a non-flagged NFT of the same collection or the order collateral (effectively laundering the stolen NFT). Furthermore, Reservoir will flag an NFT that does not support the ERC721 standard. An example of an NFT that does not support the ERC721 standard is Cryptopunks. A lender could take advantage of this property on some NFT collections to create an order that cannot be closed by the borrower, guaranteeing that the lender will get the order collateral. This would mean that even if a borrower successfully shorted an NFT and wanted to close the position to secure profits, they would lose their collateral.

Recommendation(s): Consider adding a Reservoir NFT flag check when fulfilling an order.

Status: Fixed.

Update from the client: Fixed in commit 49ed8ae1b8a03a63783213cf15f1c20b07481e86 .

7.2 [Medium] Third party oracle dependency

File(s): contracts/Dyve.sol

Description: The Dyve protocol makes use of the Reservoir API to determine whether an NFT was stolen before accepting it when closing a position. In the function `closePosition`, the `message` argument (which comes from the API) must match the `RESERVOIR_ORACLE_ADDRESS`. This is shown below:

```
if (!ReservoirOracle._verifyMessage(messageId, maxMessageAge, message)) {
    revert ReservoirOracle.InvalidMessage();
}
```

It is impossible to change the `RESERVOIR_ORACLE_ADDRESS` once it has been set. Suppose the API is discontinued or experiences downtime. In that case, it will be impossible to close orders as there will be no way to generate a valid message that can pass the message verification in `closePosition`. This would effectively act as a DOS against normally closing a position, leading to the borrower always losing their collateral.

Recommendation(s): Consider adding a function that allows the `RESERVOIR_ORACLE_ADDRESS` variable to be changed. This way, if the API has issues, it will still be possible to change the oracle address so orders can still be closed.

Status: Fixed.

Update from the client: Fixed in commit 49ed8ae1b8a03a63783213cf15f1c20b07481e86 .

7.3 [Low] `Reservoir` oracle message replay attack

File(s): contracts/Dyve.sol

Description: When a user closes a position with `closePosition` they must provide an argument `message`, which contains information indicating whether an NFT is flagged. There is a time window of 5 minutes from when the message was created when it is considered valid. It is possible for a position to be closed, then the same NFT immediately be put up for lending again. A second borrower could occur, and the borrower could potentially return the NFT using the previous message even if the NFT had been flagged during the second borrower's possession. This could allow a borrower to return an NFT flagged back to the lender.

Recommendation(s): Consider having a unique `message` between orders.

Status: Acknowledged.

7.4 [Low] Protocol can receive no fees due to integer division

File(s): contracts/Dyve.sol

Description: The functions `_transferETH(...)` and `_transferERC20(...)` determine a `protocolFee`, which is the number of tokens that should be sent to the `protocolFeeRecipient` address. In both functions, `protocolFee` is calculated as follows:

```
uint256 protocolFee = (fee * protocolFeeManager.determineProtocolFeeRate(premiumCollection, premiumTokenId, to)) /
↪  10000;
```

The left-hand side of the integer division can be a number smaller than `10000`. In this case, the integer division would result in the `protocolFee` being zero. An example would be `9999 / 10000 == 0`. The potentially lost fee is insignificant for Ether transfers and other 18 decimal ERC20 tokens. However, for some tokens with a smaller number of decimals, the fee loss may be higher. An example of such a token is WBTC (8 decimals) or GUSD (2 decimals). A lender aiming to take advantage of this to avoid protocol fees would have to set their `fee` to a value that multiplies with their fee rate so that it remains under 10000. This constraint forces the lender to charge very little for fees and isn't economically viable to take advantage of.

Recommendation(s): This finding is only concerned with fee loss for high-value, low decimal `ERC20` tokens. For the vast majority of tokens, this will not be an issue as the potential losses are virtually non-existent, and there is no incentive for a lender to act in this way. When whitelisting a currency for use within the protocol, consider the value per token and decimal amount.

Status: Acknowledged.

Update from the client: We agree and will consider this for future token whitelisting.

7.5 [Low] `premiumCollection` and `premiumTokenId` are not part of the signed order

File(s): `libraries/OrderTypes.sol`

Description: When fulfilling an order, the fields `premiumCollection` and `premiumTokenId` from the struct `OrderTypes.Order` are used to calculate the portion of order fees that the Dyve protocol should take. Since these fields are not included in the order signature, the borrower is free to change these fields to any value. This can allow a malicious borrower to make a lender give a larger fee cut to the Dyve protocol by tampering with the order's premium collection and token fields. Note that the order types `ERC721_TO_ERC20` and `ERC1155_TO_ERC20` are not affected by this as the lender is the one fulfilling the order, and they would not act in a way that negatively impacts their fees. The code snippet below shows the function used for computing the hash of an order. Notice that `premiumCollection` and `premiumTokenId` are not used to compute the hash.

```
// keccak256("Order(uint256 orderType,address signer,address collection,uint256 tokenId,uint256 amount,uint256
↪  duration,uint256 collateral,uint256 fee,address currency,uint256 nonce,uint256 endTime)")
bytes32 internal constant ORDER_HASH = 0xaad599fc66ff6b968ccb16010214cc3102e0a7e009000f61cab3f208682c3088;
...
function hash(Order memory order) internal pure returns (bytes32) {
    return keccak256(
        abi.encode(
            ORDER_HASH,
            order.orderType,
            order.signer,
            order.collection,
            order.tokenId,
            order.amount,
            order.duration,
            order.collateral,
            order.fee,
            order.currency,
            order.nonce,
            order.endTime
        )
    );
}
```

Recommendation(s): Ensure that the `premiumCollection` and `premiumTokenId` fields are signed when fulfilling an order to prevent fee tampering.

Status: Acknowledged.

Update from the client: The recommended solution by Nethermind is not implemented because it would cause a new issue, reducing the flexibility for a lender to get reduced fees if they own multiple premium NFTs. As borrowers are not incentivized to adjust directed platform fees, we do not anticipate any material redirects. As a control, Dyve can review any instances retrospectively and redirect necessary funds to the lenders as needed.

Update from Nethermind: The issue introduced by the recommendation refers to a scenario where the lender transfers the token used as a premium token. The decision of whether the lender receives the premium feature or not would be made by the lender. Therefore, we consider the recommendation a valid solution. However, by not introducing this change, the Dyve team aims for more freedom of action for its lenders. They rely on the lack of clear incentives for borrowers not to direct fees to the lenders.

7.6 [Info] Lender can borrow from themselves

File(s): `contracts/Dyve.sol`

Description: A lender can borrow from their order. This can lead to unexpected behavior as it is possible for a user to create and then interact with an order within the same transaction. Functions that rely on `block.timestamp` may assume that some time has passed between function calls which may not be the case.

Recommendation(s): A lender borrowing from their order is not an expected user action and may lead to unexpected behavior within the protocol. Consider adding a check to prevent a lender and borrower from being at the same address.

Status: Acknowledged.

7.7 [Info] Missing input validation for order duration

File(s): `contracts/Dyve.sol`

Description: A lender can create a valid order where the duration is zero. Although it does not make sense for a borrower to fulfill this order, as mentioned in a previous finding, a lender can borrow from their order allowing for positions to be opened and then collateral claimed within the same transaction, which may lead to unexpected behavior.

Recommendation(s): Consider adding a check to ensure that `OrderTypes.Order.duration` cannot be zero when fulfilling an order.

Status: Fixed.

Update from the client: Fixed in commit 49ed8ae1b8a03a63783213cf15f1c20b07481e86 .

7.8 [Info] Non-existent orders have a status of `BORROWED`

File(s): `contracts/Dyve.sol`

Description: Every order has a `status` field with the type `enum OrderStatus`. The first value declared in an enum is equivalent to `0`, the second is equivalent to `1`, and so on. The first enum value in `OrderStatus` is `BORROWED`. Since the unwritten state defaults to zero, queries to a non-existent order will have a `status` of `BORROWED`. This could cause issues when interacting with other smart contracts and/or off-chain monitoring. The declaration of the `OrderStatus` enum is shown below:

```
// The NFT's listing status
enum OrderStatus {
    BORROWED,
    EXPIRED,
    CLOSED
}
```

Recommendation(s): Change the `OrderStatus` enum so that its first (default) value indicates a non-existent order.

Status: Fixed.

Update from the client: Fixed in commit 49ed8ae1b8a03a63783213cf15f1c20b07481e86 .

7.9 [Best Practices] Hardcoded values for the Interface ID

File(s): `contracts/Dyve.sol`

Description: Writing magic values manually is always error-prone. The Dyve contract has the `interfaceId` values for `EIP-721` and `EIP-1155` manually written, as can be seen below. A typo in these values could cause integration issues to the protocol.

```
bytes4 public constant INTERFACE_ID_ERC721 = 0x80ac58cd;
bytes4 public constant INTERFACE_ID_ERC1155 = 0xd9b67a26;
```

Recommendation(s): Consider using the syntax `type(Type).interfaceId` instead of manually writing the values.

```
- bytes4 public constant INTERFACE_ID_ERC721 = 0x80ac58cd;
- bytes4 public constant INTERFACE_ID_ERC1155 = 0xd9b67a26;
+ bytes4 public constant INTERFACE_ID_ERC721 = type(IERC721).interfaceId;
+ bytes4 public constant INTERFACE_ID_ERC1155 = type(IERC1155).interfaceId;
```

Status: Fixed.

Update from the client: Fixed in commit 49ed8ae1b8a03a63783213cf15f1c20b07481e86 .

7.10 [Best Practices] Mapping entry can be saved to memory

File(s): `contracts/Dyve.sol`

Description: In the functions `closePosition` and `claimCollateral` the order data is loaded as follows:

```
Order storage order = orders[orderHash];
```

Since the mapping is loaded to a `storage` variable rather than a `memory` variable, accessing any elements inside the struct will incur a `SLOAD` operation, which can be expensive. The variable `order` can be changed to `memory` instead, reducing gas costs when accessing elements. It should be noted that changing the order status of a memory variable would not cause a change in storage. Also, when emitting the event using the memory variable, the `order` field will contain the unchanged `BORROWED` value, so the event emission would have to be changed as well. An example is shown below for the function `claimCollateral`.

```
function claimCollateral(bytes32 orderHash) external {
- Order storage order = orders[orderHash];
+ Order memory order = orders[orderHash];

  // ...

- order.status = OrderStatus.EXPIRED;
+ orders[orderHash].status = OrderStatus.EXPIRED;

  // ...

  emit Claim(
    order.orderHash,
    order.orderType,
    order.borrower,
    order.lender,
    order.collection,
    order.tokenId,
    order.amount,
    order.collateral,
    order.currency,
-   order.status
+   OrderStatus.EXPIRED
  );
}
```

Recommendation(s): Change the variable `order` to `memory` to reduce storage queries to save gas.

Status: Fixed.

Update from the client: Fixed in commit 49ed8ae1b8a03a63783213cf15f1c20b07481e86 .

7.11 [Best Practices] The `nonReentrant` modifier should be added to all order management functions

File(s): `contracts/Dyve.sol`

Description: The function `fulfillOrder(...)` has the `nonReentrant` modifier, however the functions `closePosition(...)` and `claimCollateral(...)` do not. Under normal use cases, the user would not be expected to reenter these functions, so adding `nonReentrant` helps to prevent undefined behavior that could potentially lead to unexpected results.

Recommendation(s): Add the `nonReentrant` modifier to the functions `closePosition(...)` and `claimCollateral(...)`.

Status: Fixed.

Update from the client: Fixed in commit 49ed8ae1b8a03a63783213cf15f1c20b07481e86 .

7.12 [Best Practices] Use of arbitrary number

File(s): `contracts/Dyve.sol`

Description: In the function `cancelAllOrdersForSender(...)`, the number `500000` is hardcoded. It is considered a best practice to define arbitrary values as constants.

Recommendation(s): Consider using a constant variable instead of arbitrary numbers.

Status: Fixed.

Update from the client: Fixed in commit 49ed8ae1b8a03a63783213cf15f1c20b07481e86

7.13 [Best Practices] `_transferERC20` and `_transferETH` may transfer with no amount

File(s): `contracts/Dyve.sol`

Description: In the functions `_transferERC20(...)` and `_transferETH(...)` it is possible to have a `protocolFee` of zero if the lender owns an NFT from a "premium collection". Even if the `protocolFee` is zero, a transfer to `protocolFeeRecipient` is still attempted. In `_transferEth(...)` the call `CALL(...)` operation is still executed, costing unnecessary gas. In `_transferERC20(...)`, on top of costing unnecessary gas, this can potentially lead to unexpected behavior as some ERC20 tokens may revert on zero transfers which would prevent an order from being fulfilled.

Recommendation(s): Consider checking if the fee amount to be sent to the protocol is zero, and if so, skip the transfer.

Status: Fixed.

Update from the client: Fixed in commit 49ed8ae1b8a03a63783213cf15f1c20b07481e86 .

7.14 [Best Practices] Code does not match the specification

File(s): `contracts/Dyve.sol`

Description: In the definition of the struct `Dyve.Order` there is an inline documentation (shown below) stating that the `Order.amount` field should always be set to `1` in the case of an ERC721 order. However, there are no checks in the code to ensure this condition.

```
struct Order {
    ...
    uint256 tokenId; // id of the token
    uint256 amount; // amount of the token (only applicable for ERC1155, always set to 1 for ERC721)
    uint256 duration; // duration of the borrow
    ...
}
```

A similar situation occurs for the `currency` field. The documentation provided by the Dyve team describes the `currency` field as follows: "The type of ERC20 token to be used (zero address if native ETH is to be used)". However, this behavior does not exist in the code.

Recommendation(s): Ensure consistency between code and documentation.

Status: Fixed.

Update from the client: Fixed in commit 49ed8ae1b8a03a63783213cf15f1c20b07481e86 .

8 Complementary Validations Performed by Nethermind

During the audit, Nethermind developed fuzz tests for the Dyve contract to ensure correct behavior across many different interactions. Fuzz testing for Solidity smart contracts involves providing random or unexpected inputs to a smart contract to find vulnerabilities or bugs in its code. This technique can help detect unhandled exceptions, security flaws, and unexpected behaviors.

8.1 Contract changes to support fuzz testing

To reduce the test complexity and development time, some small changes were made to the Dyve contract. The changes are listed below:

- The `closePosition` function no longer checks if the NFT is flagged.
- The `_validateOrder` function no longer ensures the `signature` is valid.

8.2 Functions and properties tested

The following is a list of functions that were tested. Each function contains a list of properties that were checked.

1. `closePosition`
   - `fullfilOrder` function should fail if the lent NFT did not belong to the lender
   - `fullfilOrder` function should fail if the lender does not have enough amount of ERC-1155 tokens
   - `fullfilOrder` function should fail if the borrower does not have enough amount of ERC-20 tokens to pay
   - `fullfilOrder` function should fail if the borrower does not have enough ETH to pay
   - `fullfilOrder` function should fail if the order has already expired
   - `fullfilOrder` function should fail if the order's fee or order's collateral is set to zero
   - `fullfilOrder` function should fail if order's `nonce` is invalid
   - After a successful call, the lent NFT should belong to the borrower
   - After a successful call, the amount of ERC-1155 tokens belonging to the lender should have decreased by the lent `amount`
   - After a successful call, the amount of ERC-1155 tokens belonging to the borrower should have increased by the lent `amount`
   - After a successful call, the `msg.sender`'s balance (the funds are taken from it) should have decreased by the `collateral + fee` amount
   - After a successful call, the ETH balance of the lender should have increased by the `fee` amount
   - After a successful call, the ETH balance of the Dyve contract should have increased by the `collateral` amount
   - After a successful call, the lender's ERC-20 balance should have increased by the `fee` amount
   - After a successful call, the borrower's ERC-20 balance should have decreased by the `fee + collateral` amount
   - After a successful call, the Dyve contract's ERC-20 balance should have increased by the `collateral` amount
   - function `fulfillOrder` should succeed if the order is valid and the users involved own the correct tokens
2. `closePosition`
   - `closePosition` should fail if the Order was already closed
   - `closePosition` should fail if the Order has already expired
   - After a successful call, the lent NFT should have returned to the lender
   - After a successful call, the lender should have received back its ERC-1155 tokens
   - After a successful call, the borrower should have returned the lent ERC-1155 tokens
   - After a successful call, the borrower should have received back the `collateral`
   - After a successful call, the Dyve contract should have transferred the `collateral` of the order
   - After a successful call, the borrower should have received back the `collateral`
   - After a successful call, the Dyve contract should have transferred the `collateral` of the order
   - function `closePosition` should succeed if the order has not expired or has not been closed

3. `claimCollateral`
   - `claimCollateral` should fail if the Order was already closed
   - `claimCollateral` should fail if the Order has not expired yet
   - After a successful call, the lent NFT should remain with the borrower
   - After a successful call, the lender should have the same amount of ERC-1155 tokens
   - After a successful call, the borrower should have the same amount of ERC-1155 tokens
   - After a successful call, the lender should have received the Order's `collateral`
   - After a successful call, the Dyve contract should have transferred the Order's `collateral`
   - function `claimCollateral` should succeed if the order is open and expired already
4. `cancelMultipleMakerOrders`
   - function `cancelMultipleMakerOrders` should revert if the list of nonces is empty
   - function `cancelMultipleMakerOrders` should revert if the caller tries to cancel an order with a `nonce` lower than the minimum nonce set for the caller
   - function `cancelMultipleMakerOrders` should succeed if the caller tries to cancel at least one valid `nonce`
5. `cancelAllOrdersForSender`
   - function `cancelAllOrdersForSender` should revert if called with a `nonce` lower or equal to the minimum nonce valid for the caller
   - function `cancelAllOrdersForSender` should revert if called with a `nonce` greater or equal to the minimum nonce valid for the caller plus 500000
   - function `cancelAllOrdersForSender` should succeed if the `nonce` used is in the correct range

8.3 Fuzz testing summary

These properties shown above were tested through 50,000 runs. All tests passed with no errors or unusual behavior. The test suite and environment has been delivered as a żip file along with the report.

9 Documentation Evaluation

Technical documentation is created to explain what the software product does. This way, developers and stakeholders can easily follow the purpose and the underlying functionality of each file/function/line. Documentation can come not only in the form of a README.md but also using code as documentation (to write clear code), diagrams, websites, research papers, videos, and external documentation. Besides being a good programming practice, proper technical documentation improves the efficiency of audits. Less time can be spent understanding the protocol and more time can be put towards auditing which improves the efficiency and overall output of the audit.

The audited files follow the NatSpec documentation, which is the recommended way to document inline comments. There is only one function that does not have a NatSpec summary, which is `hash()` in `OrderTypes.sol`. The files have a comment ratio of 12.8% and 27.8%, respectively, which is satisfactory. Aside from the NatSpec documentation, the code contains some inline comments explaining specific workings, which are very helpful to understand the flow of the protocol and the inner workings of functions.

The `README.md` contains a short description of the protocol but does not contain any information to guide developers on how to install, compile and test the code. The image linked in the document appears to be a broken link, pointing to a different organization and repository. Dyve also provided external documentation to help with the audit process. The documentation covers three main sections:

- Order Types
- Technical Overview
- Business Overview

All three sections contain detailed information about the system and how it is intended to work. **We recommend the following improvements to be made to the current state of documentation:**

- Create user-facing documentation.
- Use existing external documentation to create developer documentation.
- Improve the `README.md` by adding instructions for initial setup, compiling and testing.

10 Test Suite Evaluation

10.1 Contracts Compilation Output

```
$ npx hardhat compile
Compiled 36 Solidity files successfully
```

10.2 Tests Output

```
$ npx hardhat test

Dyve
    Initial checks
       checks initial properties were set correctly (45ms)
    Fulfill order functionality
       consumes maker ask (listing ERC721) with taker bid using ETH (234ms)
       consumes maker ask (listing ERC1155) with taker bid using ETH (126ms)
       consumes maker ask (listing ERC721) with taker bid using USDC (150ms)
       consumes maker ask (listing ERC1155) with taker bid using USDC (131ms)
       consumes maker bid (offer ERC721) with taker ask using USDC (98ms)
       consumes maker bid (offer ERC1155) with taker ask using USDC (112ms)
       checks validation for fulfillOrder (114ms)
    Premium collections functionality
       consumes Maker Bid Listing (using ETH) then the lender claims the collateral (80ms)
       consumes maker ask (listing ERC721) with taker bid using USDC and the maker owns an NFT from a premium collection
      ↪  with zero fees (84ms)
       consumes maker ask (listing ERC721) with taker bid using USDC and the maker owns an NFT from a premium collection
      ↪  with non-zero fees (102ms)
       consumes maker ask (listing ERC721) with taker bid using USDC and the maker uses a non premium collection in the
      ↪  premium collection maker field (84ms)
       consumes maker ask (listing ERC721) with taker bid using USDC and the maker uses a premium collection, but does
      ↪  not own the specified token in the maker order (92ms)
    Closing position functionality
       consumes Maker Bid ERC721 (with ETH) Listing then closes the position (96ms)
       consumes Maker Bid ERC1155 (with ETH) Listing then closes the position (288ms)
       consumes Maker Bid ERC721 (with USDC) Listing then closes the position (125ms)
       consumes Maker Bid ERC1155 (with USDC) Listing then closes the position (157ms)
       checks validation for closePosition (345ms)
    Claim collateral functionality
       consumes Maker Bid Listing (using USDC) then the lender claims the collateral (93ms)
       checks validation for claimCollateral (96ms)
    Cancel order functionality
       cancels all orders for user then fails to list order with old nonce
       checks validation for cancelAllOrdersForSender
       cancels an order and then fails to list the same order
       checks validation for cancelMultipleMakerOrders
    Protocol Fee Manager Contract functionality
       adds, adjusts and removes mockERC721 as a premium collection
       sets mockERC721 to a non-zero rate
       updates the protocol fee (60ms)
       updates the ProtocolFeeManager in the Dyve contract
    Whitelisted Currencies Contract functionality
       adds and removes USDC as a whitelisted currency
       updates the WhitelistedCurrencies Contract functionality


  30 passing (17s)
```

10.3 Code Coverage

```
npx hardhat coverage
```

The relevant output is presented below.

```
-----------------------------|----------|----------|----------|----------|----------------|
File                         |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-----------------------------|----------|----------|----------|----------|----------------|
 contracts/                  |    99.05 |    84.82 |      100 |    98.54 |                |
  Dyve.sol                   |      100 |     87.5 |      100 |      100 |                |
  ProtocolFeeManager.sol     |      100 |       70 |      100 |      100 |                |
  ReservoirOracle.sol        |    90.91 |       90 |      100 |     87.5 |          62,63 |
  WhitelistedCurrencies.sol  |      100 |       50 |      100 |      100 |                |
 contracts/libraries/        |      100 |      100 |      100 |      100 |                |
  OrderTypes.sol             |      100 |      100 |      100 |      100 |                |
-----------------------------|----------|----------|----------|----------|----------------|
All files                    |    99.06 |    84.82 |      100 |    98.55 |                |
-----------------------------|----------|----------|----------|----------|----------------|
```

11 About Nethermind

Founded in 2017 by a small team of world-class technologists, Nethermind builds Ethereum solutions for developers and enterprises. Boosted by a grant from the Ethereum Foundation in August 2018, our team has worked tirelessly to deliver the fastest Ethereum client in the market. Our flagship Ethereum client is all about performance and flexibility. Built on .NET core, a widespread, enterprise-friendly platform, Nethermind makes integration with existing infrastructures simple, without losing sight of stability, reliability, data integrity, and security. **Nethermind is made up of several engineering teams across various disciplines, all collaborating to realize the Ethereum roadmap, by conducting research and building high-quality tools**. Teams focus on specific areas of the Ethereum problem space. Each consists of specialists and experienced developers working alongside interns, learning the ropes in the Nethermind Internship Program. **Our mission is to gather passionate talent from around the world, and to tackle some of the blockchain's most complex problems**. Nethermind provides software solutions and services for developers and enterprises building the Ethereum ecosystem. We offer security reviews to projects built on EVM compatible chains and StarkNet. We have expertise in multiple areas of the Ethereum ecosystem, including protocol design, smart contracts (written in Solidity and Cairo), MEV, etc. We develop some of the most used tools on Starknet and one of the most used Ethereum clients. Learn more about us at https://nethermind.io.