Unlockd Finance - Reservoir Integration

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **April 10th, 2023 - April 19th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY 4
CONTACTS 5
1 EXECUTIVE OVERVIEW 6
1.1 INTRODUCTION 7
1.2 AUDIT SUMMARY 7
1.3 TEST APPROACH & METHODOLOGY 8
2 RISK METHODOLOGY 9
2.1 Exploitability 10
2.2 Impact 11
2.3 Severity Coefficient 13
2.4 SCOPE 15
3 ASSESSMENT SUMMARY & FINDINGS OVERVIEW 16
4 FINDINGS & TECH DETAILS 17
4.1 (HAL-01) DEBT MARKET LISTING IS NOT CANCELLED ON LIQUIDATION - CRITICAL(10) 19
Description 19
Proof of Concept 20
BVSS 21
Recommendation 21
Remediation Plan 21
4.2 (HAL-02) UPGRADEABLE CONTRACTS LACK RESERVED SPACE FOR FUTURE UPGRADES - LOW(3.3) 22
Description 22
Reference 22
BVSS 22

Recommendation 22
Remediation Plan 23
4.3 (HAL-03) UNINITIALIZED IMPLEMENTATION CONTRACTS - LOW(2.5) 24
Description 24
BVSS 24
Recommendation 24
Remediation Plan 25
4.4 (HAL-04) MISTAKENLY SENT ERC20 and ERC721 TOKENS CANNOT BE RECOVERED FROM THE CONTRACTS - INFORMATIONAL(1.3) 26
Description 26
BVSS 26
Recommendation 26
Remediation Plan 26
4.5 (HAL-05) UNUSED LIBRARIES - INFORMATIONAL(0.0) 27
Description 27
BVSS 27
Recommendation 27
Remediation Plan 27
5 MANUAL TESTING 28
5.1 INTRODUCTION 29
5.2 EXAMPLE SCENARIOS 29
6 AUTOMATED TESTING 32
6.1 STATIC ANALYSIS REPORT 33
Description 33
Results 33
6.2 AUTOMATED SECURITY SCAN 36
Description 36

Results 36

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	04/10/2023	István Böhm
0.2	Document Updates	04/19/2023	István Böhm
0.3	Draft Review	04/20/2023	Grzegorz Trawiski
0.4	Draft Review	04/20/2023	Piotr Cielas
0.5	Draft Review	04/21/2023	Gabi Urrutia
1.0	Remediation Plan	05/02/2023	István Böhm
1.1	Remediation Plan Review	05/02/2023	Ataberk Yavuzer
1.2	Remediation Plan Review	05/03/2023	Piotr Cielas
1.3	Remediation Plan Review	05/04/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com
Grzegorz Trawiski	Halborn	Grzegorz.Trawinski@halborn.com
Ataberk Yavuzer	Halborn	Ataberk.Yavuzer@halborn.com
István Böhm	Halborn	Istvan.Bohm@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Unlockd Finance is a decentralized noncustodial NFT lending protocol where users can participate as depositors or borrowers. Reservoir aggregates and normalizes the whole NFT market into a single unified platform.

Unlockd Finance engaged Halborn to conduct a security audit on their smart contracts beginning on April 10th, 2023 and ending on April 19th, 2023. The security assessment was scoped to the smart contracts provided in the UnlockdFinance/unlockd GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 AUDIT SUMMARY

The team at Halborn was provided 8 days for the engagement and assigned one full-time security engineer to audit the security of the smart contracts in scope. The security engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the audits is to:

- Identify potential security issues within the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which have been successfully addressed by Unlockd Finance. The main one was the following:

- Ensure that all existing debt market listings are canceled prior to liquidating loans.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the audit:

- Research into architecture and purpose
- Smart contract manual code review and walkthrough
- Graphing out functionality and contract logic/connectivity/functions (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes
- Manual testing by custom scripts
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Brownie, Remix IDE)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 Exploitability

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 Impact

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 Severity Coefficient

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

EXECUTIVE OVERVIEW

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

Code repositories:

1. Unlockd Protocol Smart Contracts - Reservoir Integration:

- Repository: UnlockdFinance/unlockd
- Commit ID: d91ea7ad17d40135993a477c06fbd12cf9f2a969
- Smart contracts in scope:
    - contracts/protocol/UToken.sol
        - modifier onlyUTokenManager
        - function updateUTokenManagers
    - contracts/protocol/LendPoolLoan.sol
        - modifier onlyMarketAdapter
        - modifier onlyPoolAdmin
        - function liquidateLoanMarket
        - function updateMarketAdapters
    - contracts/protocol/adapters/abstracts/BaseAdapter.sol
    - contracts/protocol/adapters/ReservoirAdapter.sol
- Fixed commit ID (final): 1785e82f2b7229c9f4d650cee5f3848a2b55482c

Out-of-scope:
- External contracts (e.g., Reservoir modules).
- Third-party libraries and dependencies.
- Economic attacks.

Other limitations:
The audited version of the Unlockd Protocol was only compatible with WETH reserves, and the Reservoir adapter was only compatible with ERC721 tokens. Consequently, any issues arising from the configuration of the protocol with other assets were excluded from the scope of this assessment.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	2	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DEBT MARKET LISTING IS NOT CANCELLED ON LIQUIDATION	Critical (10)	SOLVED - 04/24/2023
UPGRADEABLE CONTRACTS LACK RESERVED SPACE FOR FUTURE UPGRADES	Low (3.3)	SOLVED - 04/24/2023
UNINITIALIZED IMPLEMENTATION CONTRACTS	Low (2.5)	SOLVED - 04/24/2023
MISTAKENLY SENT ERC20 and ERC721 TOKENS CANNOT BE RECOVERED FROM THE CONTRACTS	Informational (1.3)	SOLVED - 04/24/2023
UNUSED LIBRARIES	Informational (0.0)	SOLVED - 04/24/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) DEBT MARKET LISTING IS NOT CANCELLED ON LIQUIDATION - CRITICAL(10)

Description:

It is possible to sell loans at a fixed price or with auctions in the Unlockd protocol using the DebtMarket contract. It was identified that existing listings in the DebtMarket contract are not canceled upon the liquidations of the associated loan via the ReservoirAdapter contract.

This vulnerability enables a malicious user to create a debt market listing before their loan is liquidated, allowing them to immediately repurchase the NFT should another user redeposit it into the protocol. Since the value of a recently deposited NFT used as collateral will be significantly higher than the loan amount, the malicious user can thereby realize a profit by repaying the loan.

Note that in the audited version of the protocol, any user could cancel the listings. However, this issue will be addressed in the latest versions of the protocol, ensuring that only the NFT owners can cancel them. However, this also prevents administrators from clearing non-existent debts from the listings.

Proof of Concept:

The debt market listing is not canceled after the loan liquidation:

```
>>> debtid = contract_debtMarket.getDebtId(nft_asset, 100)
>>> printDictionary(contract_debtMarket.getDebt(debtid))
debtId             : 1
debtor             : 0x3c51e330D1F90DF07CcC07c3fBf2196556BD235E
nftAsset           : 0xBC4CA0EdA7647A8aB7C2061c2E118A18a936f13D
tokenId            : 100
sellType           : 1
state              : 1
sellPrice          : 0
reserveAsset       : 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
scaledAmount       : 10000000000000000000
bidderAddress      : 0x3c51e330D1F90DF07CcC07c3fBf2196556BD235E
bidPrice           : 1000
auctionEndTimestamp: 1681952970
startBiddingPrice  : 1
>>> tx = contract_reservoir_adapter.liquidateReservoir(nft_asset, contract_WETH, data, 15 * 10**18, {'from': pool_liquidator})
Transaction sent: 0xdec405336e9c950c3271cab38d7f4e88d4cc5900f78600327a2d403fb5fc0fd6
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 0
  Transaction confirmed   Block: 17084493   Gas used: 503594 (7.49%)

>>> printDictionary(tx.events['LiquidatedReservoir'])
nftAsset        : 0xBC4CA0EdA7647A8aB7C2061c2E118A18a936f13D
tokenId         : 100
loanId          : 1
borrowAmount    : 12300587779812204497
liquidatedAmount: 15000000000000000000
remainAmount    : 2699412220187795503
extraDebtAmount : 0
>>> debtid = contract_debtMarket.getDebtId(nft_asset, 100)
>>> printDictionary(contract_debtMarket.getDebt(debtid))
debtId             : 1
debtor             : 0x3c51e330D1F90DF07CcC07c3fBf2196556BD235E
nftAsset           : 0xBC4CA0EdA7647A8aB7C2061c2E118A18a936f13D
tokenId            : 100
sellType           : 1
state              : 1
sellPrice          : 0
reserveAsset       : 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
scaledAmount       : 10000000000000000000
bidderAddress      : 0x3c51e330D1F90DF07CcC07c3fBf2196556BD235E
bidPrice           : 1000
auctionEndTimestamp: 1681952970
startBiddingPrice  : 1
```

Example exploitation of the vulnerability:

1. Alice borrows WETH using her NFT as collateral.
2. Alice's loan health factor decreases below the liquidation threshold.
3. Alice creates a very short-term DebtMarket auction and bids on her own auction to win it.
4. Alice wins her auction, but refrains from claiming the NFT.
5. The protocol's administrator liquidates Alice's NFT via the ReservoirAdapter contract.
6. The ReservoirAdapter contract liquidates the loan, but does not cancel the DebtMarket listing.
7. Bob purchases Alice's former NFT and borrows WETH in the Unlockd Protocol, using the NFT as collateral.
8. Alice claims her previous DebtMarket auction and obtains Bob's loan with the NFT.

9. Alice repays the loan and retrieves the NFT.

Note that if the debt market claim or buy functions revert due to mismatched scaledAmount values, the attacker can rebalance them by depositing funds into the protocol to update the properties of the loan.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (10)

Recommendation:

Ensure that all existing debt market listings are canceled prior to liquidating loans.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 1785e82 by cancelling any existing debt market listings in the liquidateReservoir function.

4.2 (HAL-02) UPGRADEABLE CONTRACTS LACK RESERVED SPACE FOR FUTURE UPGRADES - LOW (3.3)

Description:

It was identified that the BaseAdapter abstract contract lacks any storage gaps.

It is considered a best practice in upgradeable contracts to include a state variable named __gap. This __gap state variable will be used as a reserved space for future upgrades. It allows adding new state variables freely in the future without compromising the storage compatibility with existing deployments.

The size of the __gap array is usually calculated so that the amount of storage used by a contract always adds up to the same number (usually 50 storage slots).

Reference:

OpenZeppelin's storage gap

BVSS:

AO:A/AC:L/AX:H/C:N/I:C/A:N/D:N/Y:N/R:N/S:U (3.3)

Recommendation:

It is recommended to add a state variable named __gap as a reserved space for future upgrades in every upgradeable contract.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 1785e82 by adding the __gap state variable to the BaseAdapter abstract as a reserved space for future upgrades.

4.3 (HAL-03) UNINITIALIZED IMPLEMENTATION CONTRACTS - LOW (2.5)

Description:

The BaseAdapter contract uses the Initializable module from OpenZeppelin, and the implementations of this contract are not initialized by the protocol. In the proxy pattern, an uninitialized implementation contract can be initialized by someone else to take over the contract. Even if it does not affect the proxy contracts directly, it is a good practice to initialize them to prevent any unseen vulnerabilities.

In the latest version (4.8.0), this is done by calling the _disableInitializers function in the constructor. However, in the currently used version (4.4.1), this is done by adding an empty constructor with initializer modifier to the upgradable contracts.

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

Consider including a constructor to automatically mark the upgradeable contracts as initialized when they are deployed:

Listing 1: Initialization Example

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() initializer {}
```

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 1785e82 by including a constructor to automatically mark the BaseAdapter contracts as initialized when they are deployed.

4.4 (HAL-04) MISTAKENLY SENT ERC20 and ERC721 TOKENS CANNOT BE RECOVERED FROM THE CONTRACTS - INFORMATIONAL (1.3)

Description:

It was identified that the ReservoirAdapter contract is missing functionality to recover accidental ERC20 and ERC721 transfers. Mistakenly sent tokens are locked in the contracts.

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:H/Y:N/R:F/S:U (1.3)

Recommendation:

It is recommended to add a function to recover accidental token transfers.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 1785e82 by adding the rescue and rescueNFT functions to the ReservoirAdapter contract to recover accidental token transfers.

4.5 (HAL-05) UNUSED LIBRARIES - INFORMATIONAL (0.0)

Description:

An unused library import was identified in the contracts:

It was identified that the OpenZeppelin's IERC721 library import was not used in the ReservoirAdapter contract.

Unused imports decrease the readability of the contracts.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider reviewing the contracts and removing any unnecessary imports from them.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 1785e82 by removing the unnecessary IERC721 library import from the ReservoirAdapter contract.

MANUAL TESTING

5.1 INTRODUCTION

Halborn conducted a comprehensive manual assessment of the smart contracts in scope in a local test environment, examining them for potential logic flaws and vulnerabilities. For the evaluation, a mock Reservoir module smart contract was created and employed during the testing process.

5.2 EXAMPLE SCENARIOS

In the following example, the loan and debt data of an NFT collateral can be seen before and after liquidation:

```
>>> printLoanData(nft_asset, tokenid2)

Loan data:
loanId            : 2
state             : Active
borrower          : Borrower 1
nftAsset          : BAYC
nftTokenId        : 100
reserveAsset      : WETH
scaledAmount      : 9999999340436373093
bidStartTimestamp : 0
bidderAddress     : 0x0000000000000000000000000000000000000000
bidPrice          : 0
bidBorrowAmount   : 0
firstBidderAddress: 0x0000000000000000000000000000000000000000
>>>
>>> printDebtData(nft_asset, tokenid2)

NFT debt data:
loanId          : 2
reserveAsset    : WETH
totalCollateral : 13000000000000000000
totalDebt       : 12397881725433585266
availableBorrows: 0
healthFactor    : 943709599680893989
>>>
>>> tx = contract_reservoir_adapter.liquidateReservoir(nft_asset, contract_WETH, data, 15 * 10**18, {'from': pool_liquidator})
Transaction sent: 0xa8a0a9e40e8d99c16d029f0193d0ec78cac0a46824c47888c29393e8407435c8
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 0
  Transaction confirmed   Block: 17088758   Gas used: 620076 (9.22%)

>>> printLoanDataByLoanId(2)

Loan data:
loanId            : 2
state             : Defaulted
borrower          : Borrower 1
nftAsset          : BAYC
nftTokenId        : 100
reserveAsset      : WETH
scaledAmount      : 9999999340436373093
bidStartTimestamp : 0
bidderAddress     : 0x0000000000000000000000000000000000000000
bidPrice          : 0
bidBorrowAmount   : 0
firstBidderAddress: 0x0000000000000000000000000000000000000000
>>>
>>> printDebtData(nft_asset,tokenid2)

NFT debt data:
loanId          : 0
reserveAsset    : 0x0000000000000000000000000000000000000000
totalCollateral : 0
totalDebt       : 0
availableBorrows: 0
healthFactor    : 0
>>>
```

In the following example, it was verified that the liquidateReservoir function reverts if the mock Reservoir module returns an amount less than the value specified in its expectedLiquidateAmount parameter:

```
>>> # configuring the mock reservoir module to send back 15 WETH to the reservoir adapter
>>> contract_mock_reservoir.setAmount(15 * 10**18)
Transaction sent: 0xa0abe11010a55dc2489006d6b099d612485bd556aa4f125ff0dbccc600af02d5
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 84
  MockReservoir.setAmount confirmed   Block: 17088757   Gas used: 26476 (0.39%)

<Transaction '0xa0abe11010a55dc2489006d6b099d612485bd556aa4f125ff0dbccc600af02d5'>
>>> # creating a snapshot of the state
>>> chain.snapshot()
>>> # setting the minimum return amount to 15 WETH
>>> tx = contract_reservoir_adapter.liquidateReservoir(nft_asset, contract_WETH, data, 15 * 10**18, {'from': pool_liquidator})
Transaction sent: 0xa8a0a9e40e8d99c16d029f0193d0ec78cac0a46824c47888c29393e8407435c8
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 0
  Transaction confirmed   Block: 17088758   Gas used: 620076 (9.22%)

>>> # reverting to the previous state
>>> chain.revert()
17088757
>>> # setting the minimum return amount to 16 WETH
>>> tx = contract_reservoir_adapter.liquidateReservoir(nft_asset, contract_WETH, data, 16 * 10**18, {'from': pool_liquidator})
Transaction sent: 0xf37ce4ce6818238335639abf7ecb1909b8d3066d9d92205c7662d0d5a0df592d
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 0
  Transaction confirmed (reverted)   Block: 17088758   Gas used: 760439 (11.31%)
```

The following is a non-exhaustive list of test cases executed during the manual review:

- Verifying that only authorized users can execute the liquidateReservoir, updateModules, updateLiquidators, updateUTokenManagers, liquidateLoanMarket and updateMarketAdapters functions.
- Verifying that the decoding and data validation functions in the ReservoirAdapter contract are working as described in the functions' documentation.
- Verifying that the NFT collateral is transferred from the uNFT contract to the module address specified in the data parameter when calling the liquidateReservoir function.
- Verifying that it is only possible to liquidate unhealthy active loans.
- Verifying that the reserve state is updated before the borrow amount of calculation.
- Verifying that the liquidateReservoir function reverts if fewer tokens are received from the Reservoir module than the number specified in the expectedLiquidateAmount parameter.
- Verifying that it is possible to cover any potential extra debt from the treasury if there are enough funds available.
- Verifying that liquidateReservoir function reverts if the borrowed

amount cannot be covered by the sale and the treasury.

- Verifying that the correct amount of debt tokens burned from the borrower after successful liquidation.
- Verifying that the uNFT of the collateral is burned after successful liquidation.
- Verifying that after repaying the loan, the borrower receives the remaining amount from the liquidation.
- Verifying that repaid debt is deposited into Yearn vault after successful liquidation.

AUTOMATED TESTING

6.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Results:

contracts/protocol/adapters/abstracts/BaseAdapter.sol

```
BaseAdapter._revert(bytes4) (contracts/protocol/adapters/abstracts/BaseAdapter.sol#232-238) uses assembly
	- INLINE ASM (contracts/protocol/adapters/abstracts/BaseAdapter.sol#234-237)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

BaseAdapter.__BaseAdapter_init(ILendPoolAddressesProvider) (contracts/protocol/adapters/abstracts/BaseAdapter.sol#94-100) is never used a
nd should be removed
BaseAdapter._performLoanChecks(address,uint256) (contracts/protocol/adapters/abstracts/BaseAdapter.sol#110-147) is never used and should
be removed
BaseAdapter._revert(bytes4) (contracts/protocol/adapters/abstracts/BaseAdapter.sol#232-238) is never used and should be removed
BaseAdapter._settleLiquidation(address,address,address,uint256,uint256,uint256) (contracts/protocol/adapters/abstracts/BaseAdapter.sol#20
2-227) is never used and should be removed
BaseAdapter._updateLoanStateAndTransferUnderlying(uint256,address,uint256) (contracts/protocol/adapters/abstracts/BaseAdapter.sol#181-191
) is never used and should be removed
BaseAdapter._updateReserveInterestRates(address) (contracts/protocol/adapters/abstracts/BaseAdapter.sol#159-161) is never used and should
 be removed
BaseAdapter._updateReserveState(address) (contracts/protocol/adapters/abstracts/BaseAdapter.sol#152-154) is never used and should be remo
ved
BaseAdapter._validateLoanHealthFactor(address,uint256) (contracts/protocol/adapters/abstracts/BaseAdapter.sol#168-173) is never used and
should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
```

AUTOMATED TESTING

contracts/protocol/adapters/ReservoirAdapter.sol

```
ReservoirAdapter.liquidateReservoir(address,address,bytes,uint256).settlementData (contracts/protocol/adapters/ReservoirAdapter.sol#113)
is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables

ReservoirAdapter.liquidateReservoir(address,address,bytes,uint256).nftAsset (contracts/protocol/adapters/ReservoirAdapter.sol#70) lacks a
 zero-check on :
                - (success) = nftAsset.call(data) (contracts/protocol/adapters/ReservoirAdapter.sol#125)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

ReservoirAdapter._decodeSafeTransferFromData(bytes) (contracts/protocol/adapters/ReservoirAdapter.sol#220-231) uses assembly
        - INLINE ASM (contracts/protocol/adapters/ReservoirAdapter.sol#225-227)
ReservoirAdapter._decodeReservoirRouterExecuteData(bytes) (contracts/protocol/adapters/ReservoirAdapter.sol#237-247) uses assembly
        - INLINE ASM (contracts/protocol/adapters/ReservoirAdapter.sol#241-243)
BaseAdapter._revert(bytes4) (contracts/protocol/adapters/abstracts/BaseAdapter.sol#232-238) uses assembly
        - INLINE ASM (contracts/protocol/adapters/abstracts/BaseAdapter.sol#234-237)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

Low level call in ReservoirAdapter.liquidateReservoir(address,address,bytes,uint256) (contracts/protocol/adapters/ReservoirAdapter.sol#69
-172):
        - (success) = nftAsset.call(data) (contracts/protocol/adapters/ReservoirAdapter.sol#125)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
```

contracts/protocol/LendPoolLoan.sol

```
Reentrancy in LendPoolLoan.createLoan(address,address,address,uint256,address,address,uint256,uint256) (contracts/protocol/LendPoolLoan.s
ol#87-129):
        External calls:
        - IERC721Upgradeable(nftAsset).safeTransferFrom(_msgSender(),address(this),nftTokenId) (contracts/protocol/LendPoolLoan.sol#109)
        - IUNFT(uNftAddress).mint(onBehalfOf,nftTokenId) (contracts/protocol/LendPoolLoan.sol#111)
        State variables written after the call(s):
        - loanData.loanId = loanId (contracts/protocol/LendPoolLoan.sol#115)
        - loanData.state = DataTypes.LoanState.Active (contracts/protocol/LendPoolLoan.sol#116)
        - loanData.borrower = onBehalfOf (contracts/protocol/LendPoolLoan.sol#117)
        - loanData.nftAsset = nftAsset (contracts/protocol/LendPoolLoan.sol#118)
        - loanData.nftTokenId = nftTokenId (contracts/protocol/LendPoolLoan.sol#119)
        - loanData.reserveAsset = reserveAsset (contracts/protocol/LendPoolLoan.sol#120)
        - loanData.scaledAmount = amountScaled (contracts/protocol/LendPoolLoan.sol#121)
        - _nftTotalCollateral[nftAsset] += 1 (contracts/protocol/LendPoolLoan.sol#124)
        - _userNftCollateral[onBehalfOf][nftAsset] += 1 (contracts/protocol/LendPoolLoan.sol#123)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2

Reentrancy in LendPoolLoan.buyoutLoan(address,uint256,address,uint256,uint256,uint256) (contracts/protocol/LendPoolLoan.sol#262-300):
        External calls:
        - IUNFT(uNftAddress).burn(loan.nftTokenId) (contracts/protocol/LendPoolLoan.sol#287)
        - IERC721Upgradeable(loan.nftAsset).safeTransferFrom(address(this),_msgSender(),loan.nftTokenId) (contracts/protocol/LendPoolLoan
.sol#289)
        Event emitted after the call(s):
        - LoanBoughtOut(initiator,loanId,loan.nftAsset,loan.nftTokenId,loan.bidBorrowAmount,borrowIndex,buyoutAmount) (contracts/protocol
/LendPoolLoan.sol#291-299)
Reentrancy in LendPoolLoan.createLoan(address,address,address,uint256,address,address,uint256,uint256) (contracts/protocol/LendPoolLoan.s
ol#87-129):
        External calls:
        - IERC721Upgradeable(nftAsset).safeTransferFrom(_msgSender(),address(this),nftTokenId) (contracts/protocol/LendPoolLoan.sol#109)
        - IUNFT(uNftAddress).mint(onBehalfOf,nftTokenId) (contracts/protocol/LendPoolLoan.sol#111)
        Event emitted after the call(s):
        - LoanCreated(initiator,onBehalfOf,loanId,nftAsset,nftTokenId,reserveAsset,amount,borrowIndex) (contracts/protocol/LendPoolLoan.s
ol#126)
Reentrancy in LendPoolLoan.liquidateLoan(address,uint256,address,uint256,uint256) (contracts/protocol/LendPoolLoan.sol#336-376):
        External calls:
        - IUNFT(uNftAddress).burn(loan.nftTokenId) (contracts/protocol/LendPoolLoan.sol#363)
        - IERC721Upgradeable(loan.nftAsset).safeTransferFrom(address(this),_msgSender(),loan.nftTokenId) (contracts/protocol/LendPoolLoan
.sol#365)
        Event emitted after the call(s):
        - LoanLiquidated(initiator,loanId,loan.nftAsset,loan.nftTokenId,loan.reserveAsset,borrowAmount,borrowIndex) (contracts/protocol/L
endPoolLoan.sol#367-375)
Reentrancy in LendPoolLoan.liquidateLoanMarket(uint256,address,uint256,uint256) (contracts/protocol/LendPoolLoan.sol#489-517):
        External calls:
        - IUNFT(uNftAddress).burn(loan.nftTokenId) (contracts/protocol/LendPoolLoan.sol#511)
        - IERC721Upgradeable(loan.nftAsset).safeTransferFrom(address(this),_msgSender(),loan.nftTokenId) (contracts/protocol/LendPoolLoan
.sol#514)
        Event emitted after the call(s):
        - LoanLiquidatedMarket(loanId,loan.nftAsset,loan.nftTokenId,loan.reserveAsset,borrowAmount,borrowIndex) (contracts/protocol/LendP
oolLoan.sol#516)
Reentrancy in LendPoolLoan.liquidateLoanNFTX(uint256,address,uint256,uint256,uint256) (contracts/protocol/LendPoolLoan.sol#381-429):
        External calls:
        - IUNFT(uNftAddress).burn(loan.nftTokenId) (contracts/protocol/LendPoolLoan.sol#407)
        - sellPrice = NFTXSeller.sellNFTX(_addressesProvider,loan.nftAsset,loan.nftTokenId,loan.reserveAsset,amountOutMin) (contracts/pro
tocol/LendPoolLoan.sol#412-418)
        Event emitted after the call(s):
        - LoanLiquidatedNFTX(loanId,loan.nftAsset,loan.nftTokenId,loan.reserveAsset,borrowAmount,borrowIndex,sellPrice) (contracts/protoc
ol/LendPoolLoan.sol#420-428)
```

```
Reentrancy in LendPoolLoan.liquidateLoanSudoSwap(uint256,address,uint256,uint256,DataTypes.SudoSwapParams) (contracts/protocol/LendPoolLoan.sol#434-484):
	External calls:
	- IUNFT(uNftAddress).burn(loan.nftTokenId) (contracts/protocol/LendPoolLoan.sol#461)
	- sellPrice = SudoSwapSeller.sellSudoSwap(_addressesProvider,loan.nftAsset,loan.nftTokenId,sudoswapParams.LSSVMPair,sudoswapParams.amountOutMinSudoswap) (contracts/protocol/LendPoolLoan.sol#466-472)
	Event emitted after the call(s):
	- LoanLiquidatedSudoSwap(loanId,loan.nftAsset,loan.nftTokenId,loan.reserveAsset,borrowAmount,borrowIndex,sellPrice,sudoswapParams.LSSVMPair) (contracts/protocol/LendPoolLoan.sol#474-483)
Reentrancy in LendPoolLoan.repayLoan(address,uint256,address,uint256,uint256) (contracts/protocol/LendPoolLoan.sol#179-210):
	External calls:
	- IUNFT(uNftAddress).burn(loan.nftTokenId) (contracts/protocol/LendPoolLoan.sol#205)
	- IERC721Upgradeable(loan.nftAsset).safeTransferFrom(address(this),_msgSender(),loan.nftTokenId) (contracts/protocol/LendPoolLoan.sol#207)
	Event emitted after the call(s):
	- LoanRepaid(initiator,loanId,loan.nftAsset,loan.nftTokenId,loan.reserveAsset,amount,borrowIndex) (contracts/protocol/LendPoolLoan.sol#209)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3

LendPoolLoan.auctionLoan(address,uint256,address,uint256,uint256,uint256) (contracts/protocol/LendPoolLoan.sol#215-257) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(loan.state == DataTypes.LoanState.Active,Errors.LPL_INVALID_LOAN_STATE) (contracts/protocol/LendPoolLoan.sol#230)
	- require(bool,string)(loan.state == DataTypes.LoanState.Auction,Errors.LPL_INVALID_LOAN_STATE) (contracts/protocol/LendPoolLoan.sol#236)
	- require(bool,string)(bidPrice > loan.bidPrice,Errors.LPL_BID_PRICE_LESS_THAN_HIGHEST_PRICE) (contracts/protocol/LendPoolLoan.sol#238)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
```

contracts/protocol/UToken.sol

```
UToken.initialize(ILendPoolAddressesProvider,address,address,uint8,string,string).treasury (contracts/protocol/UToken.sol#57) lacks a zero-check on :
		- _treasury = treasury (contracts/protocol/UToken.sol#65)
UToken.initialize(ILendPoolAddressesProvider,address,address,uint8,string,string).underlyingAsset (contracts/protocol/UToken.sol#58) lacks a zero-check on :
		- _underlyingAsset = underlyingAsset (contracts/protocol/UToken.sol#66)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

Reentrancy in UToken._transfer(address,address,uint256,bool) (contracts/protocol/UToken.sol#338-354):
	External calls:
	- super._transfer(from,to,amount.rayDiv(index)) (contracts/protocol/UToken.sol#347)
		- _getIncentivesController().handleAction(sender,currentTotalSupply,oldSenderBalance) (contracts/protocol/IncentivizedERC20.sol#49)
		- _getIncentivesController().handleAction(recipient,currentTotalSupply,oldRecipientBalance) (contracts/protocol/IncentivizedERC20.sol#51)
	Event emitted after the call(s):
	- BalanceTransfer(from,to,amount,index) (contracts/protocol/UToken.sol#353)
Reentrancy in UToken.burn(address,address,uint256,uint256) (contracts/protocol/UToken.sol#86-100):
	External calls:
	- _burn(user,amountScaled) (contracts/protocol/UToken.sol#95)
		- _getIncentivesController().handleAction(account,oldTotalSupply,oldAccountBalance) (contracts/protocol/IncentivizedERC20.sol#74)
	- IERC20Upgradeable(_underlyingAsset).safeTransfer(receiverOfUnderlying,amount) (contracts/protocol/UToken.sol#97)
	Event emitted after the call(s):
	- Burn(user,receiverOfUnderlying,amount,index) (contracts/protocol/UToken.sol#99)
Reentrancy in UToken.mint(address,uint256,uint256) (contracts/protocol/UToken.sol#110-122):
	External calls:
	- _mint(user,amountScaled) (contracts/protocol/UToken.sol#117)
		- _getIncentivesController().handleAction(account,oldTotalSupply,oldAccountBalance) (contracts/protocol/IncentivizedERC20.sol#63)
	Event emitted after the call(s):
	- Mint(user,amount,index) (contracts/protocol/UToken.sol#119)
Reentrancy in UToken.mintToTreasury(uint256,uint256) (contracts/protocol/UToken.sol#169-184):
	External calls:
	- _mint(treasury,amount.rayDiv(index)) (contracts/protocol/UToken.sol#180)
		- _getIncentivesController().handleAction(account,oldTotalSupply,oldAccountBalance) (contracts/protocol/IncentivizedERC20.sol#63)
	Event emitted after the call(s):
	- Mint(treasury,amount,index) (contracts/protocol/UToken.sol#183)
	- Transfer(address(0),treasury,amount) (contracts/protocol/UToken.sol#182)
Reentrancy in UToken.sweepUToken() (contracts/protocol/UToken.sol#150-161):
	External calls:
	- LendingLogic.executeDepositYearn(_addressProvider,DataTypes.ExecuteYearnParams(_underlyingAsset,amount)) (contracts/protocol/UToken.sol#155-158)
	Event emitted after the call(s):
	- UTokenSwept(address(this),address(underlyingAsset),amount) (contracts/protocol/UToken.sol#160)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3

UToken._getLendPoolConfigurator() (contracts/protocol/UToken.sol#326-328) is never used and should be removed
UToken._getUnderlyingAssetAddress() (contracts/protocol/UToken.sol#287-289) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

IncentivizedERC20.__gap (contracts/protocol/IncentivizedERC20.sol#78) is never used in UToken (contracts/protocol/UToken.sol#26-365)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-state-variable
```

The results were reviewed by Halborn, the vulnerabilities were determined to be false positives or out of scope and these results were not included in the report.

6.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

contracts/protocol/adapters/abstracts/BaseAdapter.sol

Report for contracts/protocol/adapters/abstracts/BaseAdapter.sol
https://dashboard.mythx.io/#/console/analyses/81213be6-73be-4b51-895a-94959280dd72

Line	SWC Title	Severity	Short Description
97	(SWC-123) Requirement Violation	Low	Requirement violation.

contracts/protocol/adapters/ReservoirAdapter.sol

Report for contracts/protocol/adapters/ReservoirAdapter.sol
https://dashboard.mythx.io/#/console/analyses/81213be6-73be-4b51-895a-94959280dd72

Line	SWC Title	Severity	Short Description
15	(SWC-123) Requirement Violation	Low	Requirement violation.

contracts/protocol/LendPoolLoan.sol

Report for contracts/protocol/LendPoolLoan.sol
https://dashboard.mythx.io/#/console/analyses/abe134d4-a48b-4434-994a-610f521319e8

Line	SWC Title	Severity	Short Description
23	(SWC-123) Requirement Violation	Low	Requirement violation.
81	(SWC-107) Reentrancy	Low	A call to a user-supplied address is executed.
81	(SWC-113) DoS with Failed Call	Low	Multiple calls are executed in the same transaction.
611	(SWC-123) Requirement Violation	Low	Requirement violation.

contracts/protocol/UToken.sol

Report for contracts/protocol/UToken.sol
https://dashboard.mythx.io/#/console/analyses/1a6dff0c-8f18-4c37-afca-11764b5d3568

Line	SWC Title	Severity	Short Description
193	(SWC-113) DoS with Failed Call	Low	Multiple calls are executed in the same transaction.

The results were examined by Halborn, and the findings were not included in the report because they were false positive or out of scope.

THANK YOU FOR CHOOSING

HALBORN