HALBORN

Unlockd Finance - Buyout Integration

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **February 14th, 2023 - March 3rd, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	02/20/2023	István Böhm
0.2	Document Updates	03/03/2023	István Böhm
0.3	Draft Review	03/03/2023	Ataberk Yavuzer
0.4	Draft Review	03/03/2023	Piotr Cielas
0.5	Draft Review	03/03/2023	Gabi Urrutia
1.0	Remediation Plan	03/15/2023	István Böhm
1.1	Remediation Plan Review	03/15/2023	Ataberk Yavuzer
1.2	Remediation Plan Review	03/15/2023	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
Ataberk Yavuzer	Halborn	Ataberk.Yavuzer@halborn.com
István Böhm	Halborn	Istvan.Bohm@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Unlockd Finance is a decentralized finance protocol that unlocks democratized NFT liquidity allowing DeFi users to compound their wealth with a NFT-backed, cross-chain, instant loan maintaining 100% of holder perks.

The buyout functionality allows users to directly purchase unhealthy NFTs at the NFT price. Lockey holders will have a discount over the NFT price.

Unlockd Finance engaged Halborn to conduct a security audit on their smart contracts beginning on February 14th, 2023 and ending on March 3rd, 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 two weeks 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 were mostly addressed by the Unlockd Finance team.

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)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

Code repositories:

1. Unlockd Protocol Smart Contracts:

- Repository: UnlockdFinance/unlockd
- Commit ID: e0619d3ea74c8f5ef35addbb0359503e6f3271b3
- Smart contracts in scope:
  - contracts/libraries/logic/LiquidateLogic.sol
    - executeBuyout()
  - contracts/libraries/logic/ValidationLogic.sol
    - validateBuyout()
  - contracts/protocol/LendPool.sol
    - buyOut()
  - contracts/protocol/LendPoolLoan.sol
    - buyoutLoan()
  - contracts/protocol/WETHGateway.sol
    - buyoutETH()
  - contracts/protocol/PunkGateway.sol
    - buyout()
    - buyoutETH()
  - contracts/protocol/LockeyManager.sol
- Fixed commit ID (final): 787a80b84cdc106e26a4d2bdb23df8754d6fabed

Out-of-scope:
- Third-party libraries and dependencies.
- Economic attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	4	7

LIKELIHOOD

IMPACT

(HAL-03)				
(HAL-05) (HAL-06)	(HAL-02) (HAL-04)			
(HAL-07) (HAL-08) (HAL-09) (HAL-10) (HAL-11)		(HAL-01)		

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) WETHGATEWAY REVERTS WHEN USING TRANSPARENTUPGRADEABLEPROXY	Low	RISK ACCEPTED
(HAL-02) UTOKEN IS ONLY COMPATIBLE WITH YEARN WETH VAULT	Low	FUTURE RELEASE
(HAL-03) USING TRANSFER INSTEAD OF SAFETRANSFER	Low	SOLVED - 03/07/2023
(HAL-04) UNINITIALIZED IMPLEMENTATION CONTRACTS	Low	SOLVED - 03/07/2023
(HAL-05) IMPROPER PRICE MANAGER INITIALIZATION	Informational	SOLVED - 03/01/2023
(HAL-06) DISCOUNT PERCENTAGE HAS NO UPPER AND LOWER BOUNDS	Informational	SOLVED - 03/07/2023
(HAL-07) REDUNDANT LOANID VALIDATION	Informational	SOLVED - 03/07/2023
(HAL-08) MISSING EVENTS FOR CONTRACT OPERATIONS	Informational	SOLVED - 03/07/2023
(HAL-09) UNUSED LIBRARIES	Informational	SOLVED - 03/07/2023
(HAL-10) MISLEADING REVERT MESSAGE	Informational	SOLVED - 03/07/2023
(HAL-11) MISTAKENLY SENT ERC-20 TOKENS CANNOT BE RECOVERED FROM THE CONTRACTS	Informational	SOLVED - 03/07/2023

FINDINGS & TECH DETAILS

3.1 (HAL-01) WETHGATEWAY REVERTS WHEN USING TRANSPARENTUPGRADEABLEPROXY - LOW

Description:

It was identified that the WETHGateway contract calls the withdraw function of the WETH token in its withdrawETH, borrowETH, redeemETH and liquidateETH functions.

The WETH token uses the transfer method to send Ether to the caller. This function forces the gas limit of 2300. In case the WETHGateway contract uses a transparent upgradeable proxy pattern, there is an additional cost to delegate the call from the proxy to the receive function of the implementation. Because of this additional cost, the call surpasses the gas limit, which reverts the transaction.

This problem was also brought to the attention by the OpenZeppelin team after the Istanbul hardfork:
OpenZeppelin upgradeable contracts affected by Istanbul hardfork

Code Location:

The withdraw function of the WETH contract is used multiple times in the WETHGateway contract. For example, in the withdrawETH function:

Listing 1: contracts/protocol/WETHGateway.sol (Line 160)

```
158     uWETH.transferFrom(msg.sender, address(this), amountToWithdraw
 ↳ );
159     cachedPool.withdraw(address(WETH), amountToWithdraw, address(
 ↳ this));
160     WETH.withdraw(amountToWithdraw);
161     _safeTransferETH(to, amountToWithdraw);
162   }
```

The withdraw function of the WETH contract uses the transfer method to send Ether to the caller:

Listing 2: WETH9.sol (Line 45)

```
42     function withdraw(uint wad) public {
43         require(balanceOf[msg.sender] >= wad);
44         balanceOf[msg.sender] -= wad;
45         msg.sender.transfer(wad);
46         Withdrawal(msg.sender, wad);
47     }
```

Risk Level:

Likelihood - 3
Impact - 1

Recommendation:

One of the possible solutions is to use EIP-2930: Optional access lists to pre-specify and pre-pay for the accounts and storage slots that the transaction plans to access. After the operation, the SLOAD and EXT* opcodes would only cost 100 gas, low enough not to revert the following executions.

Other possible solutions involve adding a receive function to the proxy contract so that it does not delegate the calls to the implementation contract, or using a middleman contract to receive the Ether from the WETH contract.

Remediation Plan:

RISK ACCEPTED: The Unlockd team accepted the risk of this finding. The Unlockd team will test the WETHGateway contract in production environments and mitigate the issue with EIP-2930 if necessary.

3.2 (HAL-02) UTOKEN IS ONLY COMPATIBLE WITH YEARN WETH VAULT - LOW

Description:

It was identified that it is only possible to borrow WETH because the `getAvailableLiquidity` function of the `UToken` contract returns the available liquidity from the configured Yearn WETH vault, regardless of what token the user wants to borrow.

Code Location:

The borrow feature is implemented in the `_borrow()` function of the `BorrowLogic` contract. The function checks whether the amount to borrow is lesser than the available liquidity from the reserve:

Listing 3: contracts/libraries/logic/BorrowLogic.sol

```
157     require(IUToken(uToken).getAvailableLiquidity() >= params.
 ↳ amount, Errors.LP_RESERVES_WITHOUT_ENOUGH_LIQUIDITY);
```

However, the `getAvailableLiquidity` function of the `UToken` contract returns the available liquidity from the WETH Yearn vault:

Listing 4: contracts/protocol/UToken.sol

```
215   function getAvailableLiquidity() public view override returns (
 ↳ uint256) {
216     return LendingLogic.calculateYearnAvailableLiquidityInReserve(
 ↳ _addressProvider);
217   }
```

Listing 5: contracts/libraries/logic/LendingLogic.sol (Line 92)

```
89   function calculateYearnAvailableLiquidityInReserve(
90     ILendPoolAddressesProvider addressesProvider
91   ) internal view returns (uint256 availableLiquidityInReserve) {
92     address yVaultWETH = addressesProvider.getAddress(keccak256("
↳ YVAULT_WETH"));
93
94     uint256 availableLiquidityInShares = IERC20Upgradeable(
↳ yVaultWETH).balanceOf(address(this));
95
96     uint256 pricePerShare = IYVault(yVaultWETH).pricePerShare();
97
98     availableLiquidityInReserve = availableLiquidityInShares.
↳ wadMul(pricePerShare);
99   }
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

The contracts should be updated to enable users to borrow other assets outside WETH.

Remediation Plan:

PENDING: The Unlockd team accepted the risk of this finding. The contracts will be updated to be compatible with other reserves in a later release.

3.3 (HAL-03) USING TRANSFER INSTEAD OF SAFETRANSFER - LOW

Description:

It was identified that the borrow, _repay, auction, redeem, liquidate and buyout functions in the PunkGateway contract use the transfer and transferFrom functions.

It is good practice to use OpenZeppelin's SafeERC20Upgradeable wrapper and the safeTransfer and safeTransferFrom functions unless one is sure the given token reverts in case of failure.

Note that the likelihood of the risk is lowered because the Unlockd team only plans to add WETH reserves in the current version of the contract.

Code Location:

contracts/protocol/PunkGateway.sol:

- Line 191 IERC20Upgradeable(reserveAsset).transfer(onBehalfOf, amount);
- Line 232 IERC20Upgradeable(reserve).transferFrom(msg.sender, address(this), amount);
- Line 255 IERC20Upgradeable(reserve).transferFrom(msg.sender, address(this), bidPrice);
- Line 269 IERC20Upgradeable(loan.reserveAsset).transferFrom(msg.sender, address(this), (amount + bidFine));
- Line 291 IERC20Upgradeable(loan.reserveAsset).transferFrom(msg.sender, address(this), amount);
- Line 316 IERC20Upgradeable(reserve).transferFrom(msg.sender, address(this), amount);

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

It is recommended to use OpenZeppelin's SafeERC20Upgradeable wrapper to transfer ERC20 tokens.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit e0e48ea by using OpenZeppelin's SafeERC20Upgradeable wrapper.

3.4 (HAL-04) UNINITIALIZED IMPLEMENTATION CONTRACTS - LOW

Description:

Multiple contracts are using the Initializable module from OpenZeppelin, and the implementations of these contracts 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.

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

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

Listing 6: Initialization Example

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

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit e0e48ea by marking upgradeable contracts as initialized when deployed.

3.5 (HAL-05) IMPROPER PRICE MANAGER INITIALIZATION - INFORMATIONAL

Description:

It was identified that the initialize function in the NFTOracle contract does not configure the account specified in the _admin parameter as a price manager. Not setting the _admin parameter as a price manager increases the risk of accidental misconfiguration.

Code Location:

Listing 7: contracts/protocol/NFTOracle.sol (Line 82)

```
82   function initialize(address _admin, address
↳ _lendPoolConfigurator) public initializer {
83     require(_admin != address(0), Errors.INVALID_ZERO_ADDRESS);
84     __Ownable_init();
85     isPriceManager[_msgSender()] = true;
86     isPriceManager[_lendPoolConfigurator] = true;
87   }
```

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

Consider removing the _admin parameter or adding the specified address to the group of price managers.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit b1803ba by adding the specified address to the group of price managers.

3.6 (HAL-06) DISCOUNT PERCENTAGE HAS NO UPPER AND LOWER BOUNDS - INFORMATIONAL

Description:

It was identified that the setLockeyDiscountPercentage function in the LockeyManager contract does not validate its discountPercentage parameter.

This parameter determines what percentage of the original price the NFT can be bought out (e.g., 9700 means 97%). The lack of lower and upper bounds of this parameter raises the likelihood of accidentally configuring an invalid value.

Note that not configuring the discount percentage causes the contracts to revert when Lockey owners try to call the buyout functions.

Code Location:

Listing 8: contracts/protocol/LockeyManager.sol

```
37   function setLockeyDiscountPercentage(uint256 discountPercentage)
↳  external onlyPoolAdmin {
38     _lockeyDiscount = discountPercentage;
39   }
```

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

Consider adding validation to the setLockeyDiscountPercentage function and setting a non-zero default value for the _lockeyDiscount state variable.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 787a80b by adding validation to the setLockeyDiscountPercentage function and setting a non-zero default value for the _lockeyDiscount state variable.

3.7 (HAL-07) REDUNDANT LOANID VALIDATION - INFORMATIONAL

Description:

It was identified that the buyout and buyoutETH functions in the PunkGateway contract and the buyoutETH function in the WETHGateway contract validate the loanId parameter multiple times during execution.

Code Location:

The buyoutETH function validates the loanId parameter before calling the buyoutETH function of the WETHGateway gateway:

Listing 9: contracts/protocol/PunkGateway.sol (Lines 459-462,464)

```
456   function buyoutETH(uint256 punkIndex, address onBehalfOf)
 ↳ external payable override nonReentrant {
457     _checkValidCallerAndOnBehalfOf(onBehalfOf);
458
459     ILendPoolLoan cachedPoolLoan = _getLendPoolLoan();
460
461     uint256 loanId = cachedPoolLoan.getCollateralLoanId(address(
 ↳ wrappedPunks), punkIndex);
462     require(loanId != 0, "PunkGateway: no loan with such punkIndex
 ↳ ");
463
464     _wethGateway.buyoutETH{value: msg.value}(address(wrappedPunks)
 ↳ , punkIndex, onBehalfOf);
465
466     _withdrawPunk(punkIndex, onBehalfOf);
467   }
```

However, a similar validation is also performed in the buyoutETH function in the WETHGateway contract:

Listing 10: contracts/protocol/WETHGateway.sol (Lines 322-326)

```
319   function buyoutETH(address nftAsset, uint256 nftTokenId, address
 ↳  onBehalfOf) external payable override nonReentrant {
320     _checkValidCallerAndOnBehalfOf(onBehalfOf);
321
322     ILendPool cachedPool = _getLendPool();
323     ILendPoolLoan cachedPoolLoan = _getLendPoolLoan();
324
325     uint256 loanId = cachedPoolLoan.getCollateralLoanId(nftAsset,
 ↳ nftTokenId);
326     require(loanId > 0, "collateral loan id not exist");
327
328     DataTypes.LoanData memory loan = cachedPoolLoan.getLoan(loanId
 ↳ );
```

Note that the allowlists used in the _checkValidCallerAndOnBehalfOf functions are independent of each other.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider removing the redundant validation from the functions to save gas.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 787a80b by removing the redundant validation from the functions.

3.8 (HAL-08) MISSING EVENTS FOR CONTRACT OPERATIONS - INFORMATIONAL

Description:

It was identified that in the LockeyManager contract, the setLockeyDiscountPercentage function does not emit any events. As a result, blockchain monitoring systems might not be able to timely detect suspicious behaviors.

Code Location:

Listing 11: contracts/protocol/LockeyManager.sol

```
37   function setLockeyDiscountPercentage(uint256 discountPercentage)
↳  external onlyPoolAdmin {
38     _lockeyDiscount = discountPercentage;
39   }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Adding events for all important operations is recommended to help monitor the contracts and detect suspicious behavior. A monitoring system that tracks relevant events would allow the timely detection of compromised system components.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 787a80b by emitting an event in the setLockeyDiscountPercentage function.

3.9 (HAL-09) UNUSED LIBRARIES - INFORMATIONAL

Description:

Multiple unused library imports were identified in the contracts:

contracts/protocol/LendPool.sol:
- IERC20Upgradeable
- SafeERC20Upgradeable

contracts/protocol/LendPoolConfigurator.sol:
- IERC20Upgradeable

contracts/protocol/InterestRate.sol:
- IERC20Upgradeable

contracts/protocol/IncentivizedERC20.sol:
- IERC20Upgradeable
- ILendPoolAddressesProvider

contracts/libraries/logic/ValidationLogic.sol:
- IInterestRate
- IERC20Upgradeable
- SafeERC20Upgradeable

contracts/libraries/logic/LiquidateLogic.sol:
- IERC721MetadataUpgradeable
- IInterestRate
- IReserveOracleGetter

contracts/libraries/logic/BorrowLogic.sol:
- IERC721EnumerableUpgradeable
- MathUtils
- ReserveConfiguration
- INFTOracleGetter
- IReserveOracleGetter

- IInterestRate

Unused imports decrease the readability of the contracts.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to review the contracts and remove any unnecessary imports from them.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 787a80b by removing the unnecessary imports from the contracts.

3.10 (HAL-10) MISLEADING REVERT MESSAGE - INFORMATIONAL

Description:

It was found that the executeBuyout function had a misleading revert error message. It is only possible to buy out unhealthy loans, but the error message says otherwise.

Code Location:

Listing 12: contracts/libraries/logic/LiquidateLogic.sol (Line 666)

```
664     require(
665       healthFactor <= GenericLogic.
 ↳ HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
666       Errors.VL_HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD
667     );
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to use the VL_HEALTH_FACTOR_HIGHER_THAN_LIQUIDATION_THRESHOLD error message instead of VL_HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD .

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 787a80b by reverting with the correct error message.

3.11 (HAL-11) MISTAKENLY SENT ERC-20 TOKENS CANNOT BE RECOVERED FROM THE CONTRACTS - INFORMATIONAL

Description:

The LockeyManager contract is missing functions to sweep/recover accidental ERC-20 transfers:

```
>>> contract_lockeyManager.signatures
{
    'getLockeyDiscountPercentage': "0x9903b724",
    'setLockeyDiscountPercentage': "0xc755347b"
}
```

Accidentally, sent ERC-20 tokens will be locked in the contracts.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider adding a function to sweep accidental ERC-20 transfers to contracts.

Remediation Plan:

SOLVED: The Unlockd team solved the issue in commit 787a80b by adding a function to sweep accidental ERC-20 transfers.

AUTOMATED TESTING

4.1 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:

Issues identified by Slither that were relevant to the scope of the current security assessment:

contracts/protocol/LendPool.sol

```
LiquidateLogic.executeBuyout(ILendPoolAddressesProvider,mapping(address => DataTypes.ReserveData),mapping(address => DataTypes.N
ftData),mapping(address => mapping(uint256 => DataTypes.NftConfigurationMap)),DataTypes.ExecuteBuyoutParams).vars (contracts/lib
raries/logic/LiquidateLogic.sol#637) is a local variable never initialized
Reentrancy in LiquidateLogic.executeBuyout(ILendPoolAddressesProvider,mapping(address => DataTypes.ReserveData),mapping(address
=> DataTypes.NftData),mapping(address => mapping(uint256 => DataTypes.NftConfigurationMap)),DataTypes.ExecuteBuyoutParams) (cont
racts/libraries/logic/LiquidateLogic.sol#628-804):
        External calls:
        - reserveData.updateState() (contracts/libraries/logic/LiquidateLogic.sol#712)
        - ILendPoolLoan(vars.poolLoan).buyoutLoan(loanData.bidderAddress,vars.loanId,nftData.uNftAddress,vars.borrowAmount,reser
veData.variableBorrowIndex,params.amount) (contracts/libraries/logic/LiquidateLogic.sol#726-733)
        - IDebtToken(reserveData.debtTokenAddress).burn(loanData.borrower,vars.borrowAmount,reserveData.variableBorrowIndex) (co
ntracts/libraries/logic/LiquidateLogic.sol#735-739)
        - IERC20Upgradeable(loanData.reserveAsset).safeTransferFrom(vars.initiator,address(this),params.amount) (contracts/libra
ries/logic/LiquidateLogic.sol#745)
        - IERC20Upgradeable(loanData.reserveAsset).safeTransfer(reserveData.uTokenAddress,vars.borrowAmount) (contracts/librarie
s/logic/LiquidateLogic.sol#748)
        - IUToken(reserveData.uTokenAddress).depositReserves(vars.borrowAmount) (contracts/libraries/logic/LiquidateLogic.sol#75
1)
        - IERC20Upgradeable(loanData.reserveAsset).safeTransfer(loanData.bidderAddress,loanData.bidPrice) (contracts/libraries/l
ogic/LiquidateLogic.sol#757)
        - IERC20Upgradeable(loanData.reserveAsset).safeTransfer(loanData.firstBidderAddress,vars.bidFine) (contracts/libraries/l
ogic/LiquidateLogic.sol#761)
        - IERC20Upgradeable(loanData.reserveAsset).safeTransfer(loanData.firstBidderAddress,vars.remainAmount) (contracts/librar
ies/logic/LiquidateLogic.sol#764)
        - IERC20Upgradeable(loanData.reserveAsset).safeTransfer(loanData.borrower,vars.remainAmount) (contracts/libraries/logic/
LiquidateLogic.sol#771)
        - (success) = address(loanData.nftAsset).call(abi.encodeWithSignature(safeTransferFrom(address,address,uint256),address(
this),vars.onBehalfOf,params.nftTokenId)) (contracts/libraries/logic/LiquidateLogic.sol#776-783)
        - IERC721Upgradeable(loanData.nftAsset).safeTransferFrom(address(this),IUToken(reserveData.uTokenAddress).RESERVE_TREASU
RY_ADDRESS(),params.nftTokenId) (contracts/libraries/logic/LiquidateLogic.sol#786-790)
        Event emitted after the call(s):
        - Buyout(vars.initiator,loanData.reserveAsset,params.amount,vars.borrowAmount,loanData.nftAsset,loanData.nftTokenId,loan
Data.borrower,vars.onBehalfOf,vars.loanId) (contracts/libraries/logic/LiquidateLogic.sol#792-802)
Reentrancy in SupplyLogic.executeDeposit(mapping(address => DataTypes.ReserveData),DataTypes.ExecuteDepositParams) (contracts/li
```

contracts/protocol/LendPoolLoan.sol

```
Reentrancy in LendPoolLoan.buyoutLoan(address,uint256,address,uint256,uint256,uint256) (contracts/protocol/LendPoolLoan.sol#240-278):
        External calls:
        - IUNFT(uNftAddress).burn(loan.nftTokenId) (contracts/protocol/LendPoolLoan.sol#265)
        - IERC721Upgradeable(loan.nftAsset).safeTransferFrom(address(this),_msgSender(),loan.nftTokenId) (contracts/protocol/LendPoolLoan.sol#267)
        Event emitted after the call(s):
        - LoanBoughtOut(initiator,loanId,loan.nftAsset,loan.nftTokenId,loan.bidBorrowAmount,borrowIndex,buyoutAmount) (contracts/protocol/LendPoolLoan.sol#269-277)
Reentrancy in LendPoolLoan.createLoan(address,address,address,uint256,address,address,uint256,uint256) (contracts/protocol/LendPoolLoan.sol#64-107):
        External calls:
        - IERC721Upgradeable(nftAsset).safeTransferFrom(_msgSender(),address(this),nftTokenId) (contracts/protocol/LendPoolLoan.sol#86)
        - IUNFT(uNftAddress).mint(onBehalfOf,nftTokenId) (contracts/protocol/LendPoolLoan.sol#88)
        Event emitted after the call(s):
        - LoanCreated(initiator,onBehalfOf,loanId,nftAsset,nftTokenId,reserveAsset,amount,borrowIndex) (contracts/protocol/LendPoolLoan.sol#104)
```

contracts/protocol/PunkGateway.sol

```
PunkGateway.buyout(uint256,uint256,address) (contracts/protocol/PunkGateway.sol#309-325) ignores return value by IERC20Upgradeable(reserve).transferFrom(msg.sender,address(this),amount) (contracts/protocol/PunkGateway.sol#320)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
```

contracts/protocol/LockeyManager.sol

```
LockeyManager.setLockeyDiscountPercentage(uint256) (contracts/protocol/LockeyManager.sol#37-39) should emit an event for:
        - _lockeyDiscount = discountPercentage (contracts/protocol/LockeyManager.sol#38)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-arithmetic
```

Slither found no major issues relevant to the scope of the current security assessment.

- Slither found no major issues relevant to the scope of the current security assessment.
- The reentrancy vulnerabilities were checked individually, and they are all false positives.
- The issue of ignoring the return value of the transfer function is covered in the report and considered low risk in the protocol.

4.2 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/LendPoolLoan.sol

Report for contracts/protocol/LendPoolLoan.sol
https://dashboard.mythx.io/#/console/analyses/65a1050e-a798-4fc1-b31c-70a5d8e50092

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

contracts/protocol/PunkGateway.sol

Report for contracts/protocol/PunkGateway.sol
https://dashboard.mythx.io/#/console/analyses/991b3e5a-b51b-425b-b33c-c55b86184139

Line	SWC Title	Severity	Short Description
56	(SWC-107) Reentrancy	Low	Write to persistent state following external call
77	(SWC-123) Requirement Violation	Low	Requirement violation.
78	(SWC-107) Reentrancy	Low	Read of persistent state following external call.
78	(SWC-113) DoS with Failed Call	Medium	Multiple calls are executed in the same transaction.
159	(SWC-113) DoS with Failed Call	Low	Multiple calls are executed in the same transaction.
424	(SWC-113) DoS with Failed Call	Low	Multiple calls are executed in the same transaction.
492	(SWC-123) Requirement Violation	Low	Requirement violation.

contracts/protocol/WETHGateway.sol

Report for contracts/protocol/WETHGateway.sol
https://dashboard.mythx.io/#/console/analyses/151bbf07-1a3e-4bf3-895a-5f43c359b395

Line	SWC Title	Severity	Short Description
47	(SWC-107) Reentrancy	Low	Write to persistent state following external call
137	(SWC-113) DoS with Failed Call	Low	Multiple calls are executed in the same transaction.
138	(SWC-107) Reentrancy	Low	Read of persistent state following external call
138	(SWC-113) DoS with Failed Call	Low	Multiple calls are executed in the same transaction.
138	(SWC-107) Reentrancy	Low	A call to a user-supplied address is executed.
359	(SWC-123) Requirement Violation	Low	Requirement violation.

- MythX identified no issues relevant to the scope of the current security assessment.

THANK YOU FOR CHOOSING

// HALBORN