Competitive Security Assessment

TakerProtocol

Sep 11th, 2023

Secure3

secure3.io

Summary 4

Overview 5

Audit Scope 6

Code Assessment Findings 10

TPL-1:Malicious actor can steal PUNK from `PunkGateway` contract due to insufficient checks 13

TPL-2:Malicious actor can drain `WETH_GATEWAY` balance 16

TPL-3:Incorrect permission validation results in loss of funds in `AirdropFlashClaimReceiver.sol::executeOperation()` function 19

TPL-4:Oracle implementation for ERC1155 collateral valudation is flawed, making the code impossible to support ERC1155 asset 25

TPL-5:The hard-coded value causes a DOS issue with a revert on the `executeOperation()` function of `AirdropFlashClaimReceiver`. 28

TPL-6:Use safeTransfer instead of transfer in TERC721 contract claimERC20Airdrop function 30

TPL-7: potential reorg attack in UserFlashclaimRegistry contract _createReceiver function 31

TPL-8:PriceOracle will use the wrong price if the Chainlink registry returns price outside min/max range 32

TPL-9:`safeMint` is not used when depositing to the pool 34

TPL-10:weird ERC20 tokens will result in liquidity issues and protocol insolvency 36

TPL-11:An off-by-one error in the UserNftConfiguration::setUsingAsCollateral() library leads to one valuable NFT becoming inoperable. 38

TPL-12:Liquidator is not able to liquidate using ETH in `PunkGateway` if not exact ETH is passed 41

TPL-13:AidropDistribution does not comply with chainlink VRF security guide 43

TPL-14:If the price is not updated in time, the user's funds may be damaged in `TakerOracleGetter::getReserveAssetPrice()` function 46

TPL-15:Logical error in `WETHGateway::repay()` function 49

TPL-16:Centralized Risk With Coin Transfer in lendingPool::claimAidrop 54

TPL-17:Possible lost `msg.value` in `WETHGateway` and `PunkGateway` contract `liquidateWithWeth` & `transfer` function 58

TPL-18:price feed staleness in "TakerOracleGetter" contract "getReserveAssetPrice" function 59

TPL-19:Array length should be checked in AirdropFlashClaimReceiver contract executeOperation function 61

TPL-20:Input validation should be added in "TERC20" contract "burn" function 62

TPL-21:Recorded events are inaccurate in `DepositExecutor::deposit()` function 64

TPL-22:Remove checks for scenarios which will be impossible in WETHGateway::repay() 66

TPL-23:Usage of Builtin Symbols in `ILendingPool::getTNFTProxyAddress()` function 71

TPL-24:Missing Zero Address Check in `LendingPool::initialize()` function 72

TPL-25:Gas Optimization: in `AirdropFlashClaimReceiver` and `PunkGateway` contract 73

Disclaimer 75

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

Project Detail

Project Name	TakerProtocol
Platform & Language	Solidity
Codebase	• https://github.com/takerprotocol/taker-lending • audit commit - 272096004abe93b209b9ae5d4aef4e0070302803 • final commit - 7dbbf966ddf3d078e9f9ec204f67e12e85fabef9
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Code Vulnerability Review Summary

Vulnerability Level	Total	Reported	Acknowledged	Fixed	Mitigated	Declined
Critical	5	0	0	3	1	1
Medium	11	0	2	8	0	1
Low	8	0	0	6	0	2
Informational	1	0	0	1	0	0

Audit Scope

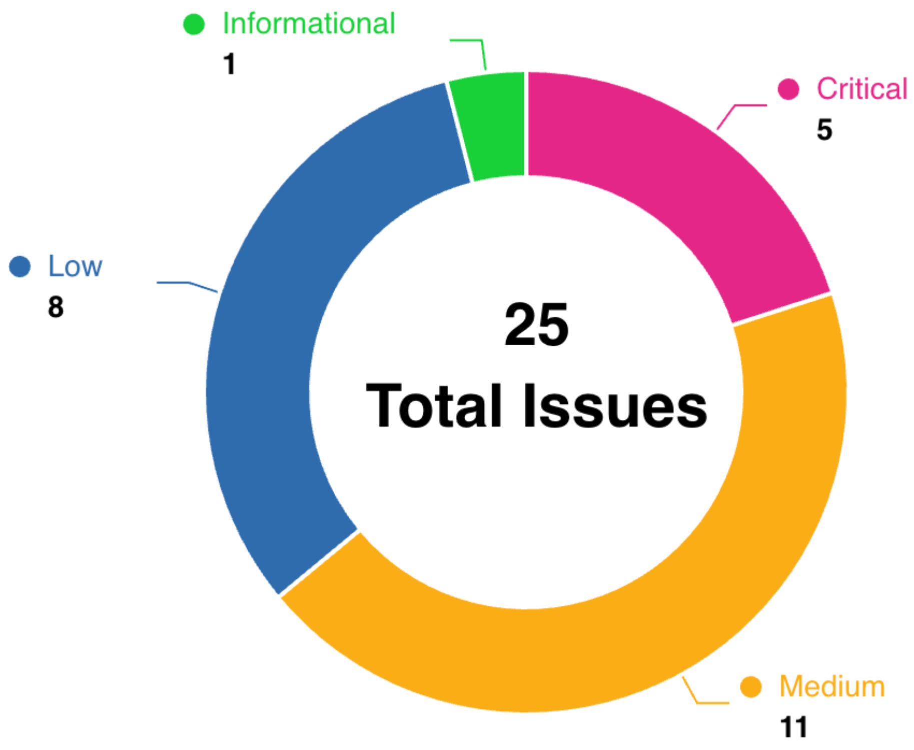
File	SHA256 Hash
contracts/pool/lendingpool/LendingPool.sol	d1197ee4d7a5c2487c0975189f44aa426dadeb2757f8723932892be0c88a7dad
contracts/pool/lendingpool/LendingPoolConfigurator.sol	70305294c36c019f8da76c437fb7b6eb403f64aaf7973cae6c454c6a16c91ac2
contracts/airdrop/AirdropDistribution.sol	08a85cf90d574545a9b35f050a0a25f0bda5f7510b87678163db0762b2757001
contracts/libraries/core/DepositExecutor.sol	01725bd6c69048496d73e10975467bc98720b8a5aa0ba68d6a50d38c0c5298e2
contracts/airdrop/AirdropFlashClaimReceiver.sol	9810b6a2b1cabbf9e8cfde09e4e95e3dbaf7a7132d4c258c704253f720dedb5a
contracts/libraries/core/UserVariableCalculator.sol	0e7b8135d0eafaddeddde1e32851ab50aae6d06a14fae83f23118c33ab70cda5
contracts/libraries/types/ReserveConfiguration.sol	b278ccb1809f3f974b89bdfc38872094893e548053319a07ed459b0eee361eb3
contracts/libraries/types/Reserve.sol	d2f82759a17b49eeeb7de03431808e36316d2545551a2d185a3c91844f43c353
contracts/interfaces/ILendingPool.sol	aa658e0a51745081ab5ec40f616f0192fe46c0d09729e039774cb12699ba4e18
contracts/libraries/core/Validator.sol	5adbc3c1d69dc57bf806f5649ad8eab93e3f2ddec10da737ee70ca17717e4a47
contracts/libraries/core/LiquidationExecutor.sol	813cc003a55086f0f63772ea2dbaf6003fa367903b1e47004ef1ce45cc098bae
contracts/libraries/core/ReserveVariableCalculator.sol	27f294c4bf343dd849c24a4a1d40fa436efe145ab7dc6df97c220e951d90fea6
contracts/tokens/CertificateTokens/TERC721.sol	7f624e2b40303347d32aa79dbe69264635a8424494e6c3a6ca9ef71882d4fdd9
contracts/pool/gateways/PunkGateway.sol	56fa896f6e403715005ba74eb146a999d8bc28d022c2f1a558c4fe07cffeb688
contracts/pool/gateways/WETHGateway.sol	68caa5aea91d8f237806c14e4f58793856e7a748b977120829caff1d794e9c60

contracts/tokens/CertificateTokens/TERC1155.sol	**f50a3ae825f867d151d5408087799a3cd4966f10d54c1cac0cae6bed8c7e37d2**
contracts/tokens/InterestBearingTokens/DebtToken.sol	**82da432cb8b8389f18b1f12c1ebc8fb67c5034ed3d60587a4d3bb08b0b18e1a5**
contracts/interfaces/configuration/ILendingPoolConfigurator.sol	**aead60ec3e0825ff935c71185c3bd3f9e74c575521c3415b55c01ed23c1cd993**
contracts/libraries/core/BorrowExecutor.sol	**b0b6788b671b2b95a2cb0eb3e8fd0e0c74d0addfe78782bc599ea7dfd2393cda**
contracts/libraries/core/FlashClaimExecutor.sol	**313e32a899bbcdbc8361284d84d1a3a6b7a48d4ed82e270a0f1b715c3c2de3b2**
contracts/tokens/InterestBearingTokens/TToken.sol	**cb8340767a01d8e038f3010b275f1f09240e3cda5f8a1492f15345b22782760b**
contracts/libraries/types/UserConfiguration.sol	**8371660199dc8bd1024559bda10a19d3acd1522abb432717b55e040623e1f85d**
contracts/tokens/CertificateTokens/TERC20.sol	**d501ebc9094b311d6cf1169d46b13dbf6d0663243f630bd4327e40af11056289**
contracts/libraries/types/NFTReserve.sol	**c669e3e0f2b3629695b8b9297c76a6b13203a8aa31fdff601a32be6006ce3c36**
contracts/configuration/TakerAddressesProvider.sol	**cb79a68ac27960f9f8a68d2a539f9524c1d1425a6ba0ee767555e542859a02fb**
contracts/libraries/Errors.sol	**7a4fc81440c3c0b8a6f09ded14ed1ef2180475713ee8f8486a936aecb7737bba**
contracts/tokens/InterestBearingTokens/ScaledERC20.sol	**2f2183d1a44679bff76bfcf5546862c761f492702bca6a9229dd592b56d1b7a8**
contracts/pool/lendingpool/InterestRateCalculator.sol	**1eba232b63607f274cd6c50fbd52afcb24246d4154cda005b44b416378639f62**
contracts/libraries/math/InterestCalculator.sol	**c313ea644951430599e52f601cfe699c5144572aaf74b699f2b5ad4fea065a43**
contracts/interfaces/oracle/AggregatorV3Interface.sol	**871fe26cee6992c4180cc133314a794a230c7e56bcdf9ea53c058dd9bb500ae6**
contracts/libraries/types/UserNftConfiguration.sol	**0345a5347eecad8d4091e80308aa8bb50b2283357966388ebe1e9e1ffdc635b5**
contracts/oracle/TakerOracleGetter.sol	**8abea92de7a1a77cbcd2c5aa4e41fe715efd45e0b892d27ef8112c33b2841f63**

contracts/airdrop/UserFlashclaimRegistry.sol	**f3b2f3b30788ccc34ca565eeda3e5906b229d7d3ee5cf3a600e71a15467823aa**
contracts/libraries/math/WadRayMath.sol	**65d67fb2b6581fb2fe26064d5291a7799e7314afeeda69da8a82e686ab806e31**
contracts/interfaces/tokens/ITERC721.sol	**9224c73ae3020135249e22ca7d7abd63ece0eef10d9e784b1ad600007faeca31**
contracts/tokens/CertificateTokens/AirdropReceiver.sol	**6169c546f20871143b5c9376e9e66567a4c7ab9841f63a9755f9fe43cf0da311**
contracts/configuration/TakerAddressesProviderRegistry.sol	**1bf39b40953e61bdccc7e693c847f6ba7fe86da4c92791ce9d06e35cd082d352**
contracts/pool/lendingpool/LendingPoolStorage.sol	**90bf531649b124474e78271d232e8a02dda699de1c3d779ea98c40b24f48ba17**
contracts/interfaces/tokens/IInitializableToken.sol	**911c4b066521a36b74a010140bbd2a23df2d9465d0af1280511b7bd3cdd81ec6**
contracts/interfaces/IWETHGateway.sol	**df83209d69039a419767b8ec1adfdcc50508466c74ab7ad3759b87ad5571eb1d**
contracts/libraries/proxy/TakerUpgradeableProxy.sol	**57f7a96ffe9ec29fed4ffd2aeca1be168e89ccd48ad230c1a708d60a43e80331**
contracts/interfaces/tokens/IScaledERC20.sol	**88d96734d80b6bebae1173b478bf8424893858f431f1c984b16904bec7d38414**
contracts/interfaces/configuration/ITakerAddressesProvider.sol	**a16cebc00c4488378561066673d642efacdda455e829f1ad79882335bb5f3baa**
contracts/interfaces/tokens/ITToken.sol	**62cab155a0fd5c6347902c9d0d826ee6e48d6bfc01486208058dcdeecbf1daba**
contracts/libraries/math/PercentageMath.sol	**31d294d103ef8d36c6ddecd3666ed5b34174602abf847a45be2cc0642c993302**
contracts/interfaces/tokens/IDebtToken.sol	**8bc7269eaf415e36889b33d8386218ce0499260a0a8f9c2d27f54251d6b9d8b7**
contracts/interfaces/tokens/IWETH.sol	**380d68eafe0f24e9fbd431829a8375dc4eb612082c51434191ca547227980183**
contracts/interfaces/IPunkGateway.sol	**0efa6cb3d88e3bcf3262e3fa01ff2eb60f60a87b99946873559ac0999d8fcf85**
contracts/interfaces/airdrop/IFlashClaimReceiver.sol	**8466410448784e99104077e0d7920b18a49aa0fb8c0a19b4335e418aa2b9ea77**

contracts/interfaces/tokens/ITERC20.sol	**be0c679167c779e7b379e978eed6e578a3c3b18bbe46ff15661737e283313ec1**
contracts/interfaces/IIncentivesController.sol	**3b81e694449d5e5ba14edf29081120094ac50d768b198449b15a53f17952ae0f**
contracts/interfaces/IInterestRateCalculator.sol	**e954d3d7961cba58d7bbfe0e07005de4cc32038c55b0391d1aa1f555ce88afb3**
contracts/interfaces/configuration/ITakerAddressesProviderRegistry.sol	**abfaf7cbd21825a18ed52b4da6edd2d332ad7c8dc2dd233d3dda6cda95d8c78e**
contracts/interfaces/tokens/ITERC1155.sol	**7cf5d9c1a33243f0d5a87187c5cc6c6f90304d1ec689c88cd12195f5a726188c**
contracts/interfaces/tokens/IWPunk.sol	**36d00c6e5e2407e6badd948a3b6c5359c5612682c4e78f2783fb0562a98b5fb1**
contracts/interfaces/oracle/IPriceOracleGetter.sol	**d01f0b9bf155fdaaf11f663ad8c328db061097a72610797b3eec691452139017**
contracts/interfaces/tokens/IPunk.sol	**731efec493d63fdd247880a929bd945553c268511f177a33fa432a27ce99851d**
contracts/interfaces/airdrop/IUserFlashclaimRegistry.sol	**0b3d049deb7620caf9dd738ae66d6dcb4d1134a4d8355cc36e46333f6d56d361**
contracts/libraries/proxy/TakerProxyAdmin.sol	**25869ce21ee2b6ac414076e387d77e3a5970b4b7f4a701a42226e4082b9b63c5**

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
TPL-1	Malicious actor can steal PUNK from `PunkGateway` contract due to insufficient checks	Logical	Critical	Fixed	SerSomeone
TPL-2	Malicious actor can drain `WETH_GATEWAY` balance	Logical	Critical	Fixed	SerSomeone, LiRiu
TPL-3	Incorrect permission validation results in loss of funds in `AirdropFlashClaimReceiver.sol::executeOperation()` function	Logical	Critical	Declined	Xi_Zi

TPL-4	Oracle implementation for ERC1155 collateral valudation is flawed, making the code impossible to support ERC1155 asset	Logical	Critical	Mitigated	ladboy233
TPL-5	The hard-coded value causes a DOS issue with a revert on the `executeOperation()` function of `AirdropFlashClaimReceiver`.	DOS	Critical	Fixed	grep-er
TPL-6	Use safeTransfer instead of transfer in TERC721 contract claimERC20Airdrop function	Logical	Medium	Fixed	ginlee
TPL-7	potential reorg attack in UserFlashclaimRegistry contract _createReceiver function	Logical	Medium	Fixed	ginlee
TPL-8	PriceOracle will use the wrong price if the Chainlink registry returns price outside min/max range	Oracle Manipulation	Medium	Acknowledged	ginlee, ladboy233
TPL-9	`safeMint` is not used when depositing to the pool	Logical	Medium	Fixed	SerSomeone
TPL-10	weird ERC20 tokens will result in liquidity issues and protocol insolvency	Logical	Medium	Fixed	SerSomeone
TPL-11	An off-by-one error in the UserNftConfiguration::setUsingAsCollateral() library leads to one valuable NFT becoming inoperable.	Logical	Medium	Fixed	grep-er
TPL-12	Liquidator is not able to liquidate using ETH in `PunkGateway` if not exact ETH is passed	Logical	Medium	Fixed	SerSomeone
TPL-13	AidropDistribution does not comply with chainlink VRF security guide	Race Condition	Medium	Acknowledged	ladboy233
TPL-14	If the price is not updated in time, the user's funds may be damaged in `TakerOracleGetter::getReserveAssetPrice()` function	Oracle Manipulation	Medium	Fixed	ladboy233, Xi_Zi

TPL-15	Logical error in `WETHGateway::repay()` function	Logical	Medium	Fixed	Xi_Zi
TPL-16	Centralized Risk With Coin Transfer in lendingPool::claimAidrop	Privilege Related	Medium	Declined	Xi_Zi
TPL-17	Possible lost `msg.value` in `WETHGateway` and `PunkGateway` contract `liquidateWithWeth` & `transfer` function	Logical	Low	Declined	ginlee
TPL-18	price feed staleness in "TakerOracleGetter" contract "getReserveAssetPrice" function	Oracle Manipulation	Low	Fixed	ginlee, ladboy233
TPL-19	Array length should be checked in AirdropFlashClaimReceiver contract executeOperation function	Logical	Low	Fixed	ginlee
TPL-20	Input validation should be added in "TERC20" contract "burn" function	Logical	Low	Declined	ginlee
TPL-21	Recorded events are inaccurate in `DepositExecutor::deposit()` function	Code Style	Low	Fixed	Xi_Zi
TPL-22	Remove checks for scenarios which will be impossible in WETHGateway::repay()	Logical	Low	Fixed	grep-er, Xi_Zi
TPL-23	Usage of Builtin Symbols in `ILendingPool::getTNFTProxyAddress()` function	Language Specific	Low	Fixed	Xi_Zi
TPL-24	Missing Zero Address Check in `LendingPool::initialize()` function	Code Style	Low	Fixed	Xi_Zi
TPL-25	Gas Optimization: in `AirdropFlashClaimReceiver` and `PunkGateway` contract	Gas Optimization	Informational	Fixed	Xi_Zi

TPL-1:Malicious actor can steal PUNK from `PunkGateway` contract due to insufficient checks

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	SerSomeone

Code Reference

- code/contracts/pool/gateways/PunkGateway.sol#L75
- code/contracts/pool/gateways/PunkGateway.sol#L112

```
75:function withdraw(address pool, uint256[] calldata punkIndexes, address to) external override {

112:function liquidateWithWeth(address pool, uint256 punkIndex, address user) external override {
```

Description

SerSomeone : The `PunkGateway` has a recovery function that can be called only by the owner of the contract to transfer tokens in the contract.

```
  /// @inheritdoc IPunkGateway
  function transfer(address token, address to, uint256 amount) external onlyOwner {
    IERC20(token).safeTransfer(to, amount);
  }

  /// @inheritdoc IPunkGateway
  function transferPunk(address to, uint256 index) external onlyOwner {
    IPunk(PUNK).transferPunk(to, index);
  }

  /// @inheritdoc IPunkGateway
  function transferERC721(address token, address to, uint256 id) external onlyOwner {
    IERC721(token).safeTransferFrom(address(this), to, id);
  }
```

However, a malicious actor can manipulate the `withdraw` and `liquidateWithWeth` by supplying a hacker controlled `pool` address that will return hacker controlled addresses and in the end burn the `WPUNK` and transfer the `PUNK` to the hacker.

Explained:

```
    function liquidateWithWeth(address pool, uint256 punkIndex, address user) external override {
      UserVariableCalculator.StateVar memory stateVar = ILendingPool(pool).getUserState(user);
      uint256 price = IPriceOracleGetter(ILendingPool(pool).ADDRESS_PROVIDER().getPriceOracle())
        .getReserveAssetPrice(address(PUNK));
      // Only suppport liquidate nft, so do not need to do decimal converts
      uint256 liquidateAmount = (stateVar.totalDebtInEth * price) / stateVar.totalCollateralInEth;
      IERC20(address(WETH_GATEWAY.WETH())).safeTransferFrom(
        msg.sender,
        address(this),
        liquidateAmount
      );

      ILendingPool(pool).liquidate(
        address(WPUNK),
        punkIndex,
        address(WETH_GATEWAY.WETH()),
        user,
        address(this),
        false
      );
      WPUNK.burn(punkIndex);
      PUNK.transferPunk(msg.sender, punkIndex);
    }
```

Above a hacker controlled `pool` can abide to the `ILendingPool` interface and return a hacker controlled `StateVar` and `priceOracle`. Since the hacker controls the `StateVar` and `oracle` he also controls the `liquidateAmount` which can be set to `0`. Then the hacker can make `pool.liquidate` return true without doing and the `WPUNK` will be burned and the `PUNK` will be sent to the hacker address.

```
    function withdraw(address pool, uint256[] calldata punkIndexes, address to) external override {
      IERC721 tWPunk = IERC721(ILendingPool(pool).getNftReserveData(address(WPUNK)).tNFTAddress);

      address[] memory nfts = new address[](punkIndexes.length);
      uint256[] memory amounts = new uint256[](punkIndexes.length);
      for (uint256 i = 0; i < punkIndexes.length; i++) {
        tWPunk.safeTransferFrom(msg.sender, address(this), punkIndexes[i]);
        nfts[i] = address(WPUNK);
        amounts[i] = 1;
      }
      ILendingPool(pool).withdrawNFTs(nfts, punkIndexes, amounts, address(this));
      for (uint256 i = 0; i < punkIndexes.length; i++) {
        WPUNK.burn(punkIndexes[i]);
        PUNK.transferPunk(to, punkIndexes[i]);
      }
    }
```

Above a hacker controlled `pool` can abide to the `ILendingPool` interface and return a hacker controlled `tWPunk` The hacker can make both `tWPunk.safeTransferFrom` and `pool.withdrawNFTs` return true without doing and the `WPUNK`s will be burned and the `PUNK`s will be sent to the hacker address.

Recommendation

SerSomeone : There needs to be a validation of the pool address in the gateway. Consider when calling `approvePool` to add the pool to a whitelist.

Client Response

Fixed

TPL-2:Malicious actor can drain `WETH_GATEWAY` balance

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	SerSomeone, LiRiu

Code Reference

- code/contracts/pool/gateways/WETHGateway.sol#L51-L64
- code/contracts/pool/gateways/WETHGateway.sol#L67
- code/contracts/pool/gateways/WETHGateway.sol#L52

```
51:/// @inheritdoc IWETHGateway
52:  function withdraw(address pool, uint256 amount, address to) external override {
53:    ITToken tWETH = ITToken(ILendingPool(pool).getReserveData(address(WETH)).tTokenAddress);
54:    uint256 balance = tWETH.compoundedBalanceOf(msg.sender);
55:    uint256 withdrawAmount = amount;
56:
57:    if (amount == type(uint256).max) {
58:      withdrawAmount = balance;
59:    }
60:    tWETH.transferFrom(msg.sender, address(this), withdrawAmount);
61:    ILendingPool(pool).withdraw(address(WETH), withdrawAmount, address(this));
62:    WETH.withdraw(withdrawAmount);
63:    _transferETH(to, withdrawAmount);
64:  }

52:function withdraw(address pool, uint256 amount, address to) external override {

67:function borrow(address pool, uint256 amount) external override {
```

Description

SerSomeone : There is no check if the pool exists in the `WETH_GATEWAY` and therefore a hacker can call `withdraw` or `borrow` functions to drain the contracts balance

```
  function withdraw(address pool, uint256 amount, address to) external override {
    ITToken tWETH = ITToken(ILendingPool(pool).getReserveData(address(WETH)).tTokenAddress);
    uint256 balance = tWETH.compoundedBalanceOf(msg.sender);
    uint256 withdrawAmount = amount;

    if (amount == type(uint256).max) {
      withdrawAmount = balance;
    }
    tWETH.transferFrom(msg.sender, address(this), withdrawAmount);
    ILendingPool(pool).withdraw(address(WETH), withdrawAmount, address(this));
    WETH.withdraw(withdrawAmount);
    _transferETH(to, withdrawAmount);
  }

  /// @inheritdoc IWETHGateway
  function borrow(address pool, uint256 amount) external override { // @audit hacker can drain funds from the gateway
    ILendingPool(pool).borrow(address(WETH), amount, msg.sender);
    WETH.withdraw(amount);
    _transferETH(msg.sender, amount);
  }
```

As can be seen above:

- in the `borrow` function if a hacker controlled `pool` that returns true and does nothing on `borrow` then a hacker can transfer any `amount` to himself (balance of the contract)
- In the `withdraw` function if a hacker controlled `pool` that returns true and does nothing on `withdraw` and a hacker control `tWETH` is called then a hacker can transfer any `amount` to himself (balance of the contract)

A hacker can monitor the balance of the contract or front-run the `onlyOwner` `transfer` for an automatic and guaranted profit

```
  /// @inheritdoc IWETHGateway
  function transfer(address token, address to, uint256 amount) external onlyOwner {
    if (token == address(0)) {
      _transferETH(to, amount);
    } else {
      IERC20(token).safeTransfer(to, amount);
    }
  }
```

LiRiu : WETHGateway has implemented the `withdraw` function for redeeming ETH collateral.

```
  function withdraw(address pool, uint256 amount, address to) external override {
    ITToken tWETH = ITToken(ILendingPool(pool).getReserveData(address(WETH)).tTokenAddress);
    uint256 balance = tWETH.compoundedBalanceOf(msg.sender);
    uint256 withdrawAmount = amount;

    if (amount == type(uint256).max) {
      withdrawAmount = balance;
    }
    tWETH.transferFrom(msg.sender, address(this), withdrawAmount);
    ILendingPool(pool).withdraw(address(WETH), withdrawAmount, address(this));
    WETH.withdraw(withdrawAmount);
    _transferETH(to, withdrawAmount);
  }
```

The function lacks a check for the pool parameter, which allows the `ITToken tWETH` contract to be manipulated.

An attacker can construct a malicious tWETH contract that renders all code before `WETH.withdraw(withdrawAmount)` ineffective without throwing an exception.

For the attacker, only the following code is actually executed:

```
WETH.withdraw(withdrawAmount);
_transferETH(to, withdrawAmount);
```

Therefore, attackers can steal ETH from the contract without any cost.

Recommendation

SerSomeone : Validate that the `pool` address is a real pool deployed by the protocol.
LiRiu : It is recommended to add a whitelist for pools' address.

```
  function withdraw(address pool, uint256 amount, address to) external override {
    require(inWhiteList(pool), "Invalid Pool");
    ...
  }
```

Client Response

Fixed

TPL-3:Incorrect permission validation results in loss of funds in `AirdropFlashClaimReceiver.sol::executeOperation()` function

Category	Severity	Client Response	Contributor
Logical	Critical	Declined	Xi_Zi

Code Reference

- code/contracts/airdrop/AirdropFlashClaimReceiver.sol#L68-L164

```
function executeOperation(
    address[] calldata nftAssets,
    uint256[][] calldata nftTokenIds,
    address initiator,
    address operator,
    bytes calldata params
  ) external override returns (bool) {
    require(nftTokenIds.length > 0, "empty token list");
    require(initiator == owner(), "not contract owner");

    ExecuteOperationLocalVars memory vars;

    // decode parameters
    (
      vars.airdropTokenTypes,
      vars.airdropTokenAddresses,
      vars.airdropTokenIds,
      vars.airdropContract,
      vars.airdropParams,
      vars.ethValue
    ) = abi.decode(params, (uint256[], address[], uint256[], address, bytes, uint256));

    // airdrop token list can be empty, no need transfer immediately after call method
    // require(vars.airdropTokenTypes.length > 0, "invalid airdrop token type");
    require(vars.airdropTokenAddresses.length == vars.airdropTokenTypes.length, "invalid airdrop token address length");
    require(vars.airdropTokenIds.length == vars.airdropTokenTypes.length, "invalid airdrop token id length");

    require(vars.airdropContract != address(0), "invalid airdrop contract address");
    require(vars.airdropParams.length >= 4, "invalid airdrop parameters");

    // allow operator transfer borrowed nfts back to TERC721
    for (uint256 idIdx = 0; idIdx < nftAssets.length; idIdx++) {
       IERC721Upgradeable(nftAssets[idIdx]).setApprovalForAll(operator, true);
     }

     // call project aidrop contract
     AddressUpgradeable.functionCallWithValue(
       vars.airdropContract,
       vars.airdropParams,
       vars.ethValue,
       "call airdrop method failed"
     );

     vars.airdropKeyHash = getClaimKeyHash(initiator, nftAssets, nftTokenIds, params);
     airdropClaimRecords[vars.airdropKeyHash] = true;

     // transfer airdrop tokens to owner
     for (uint256 typeIndex = 0; typeIndex < vars.airdropTokenTypes.length; typeIndex++) {
       require(vars.airdropTokenAddresses[typeIndex] != address(0), "invalid airdrop token address");

       if (vars.airdropTokenTypes[typeIndex] == 1) {
         // ERC20
         vars.airdropBalance = IERC20Upgradeable(vars.airdropTokenAddresses[typeIndex]).balanceOf(address(this));
         if (vars.airdropBalance > 0) {
           IERC20Upgradeable(vars.airdropTokenAddresses[typeIndex]).transfer(initiator, vars.airdropBalance);
         }
       } else if (vars.airdropTokenTypes[typeIndex] == 2) {
         // ERC721 with Enumerate
         vars.airdropBalance = IERC721Upgradeable(vars.airdropTokenAddresses[typeIndex]).balanceOf(address(this));
         for (uint256 i = 0; i < vars.airdropBalance; i++) {
           vars.airdropTokenId = IERC721EnumerableUpgradeable(vars.airdropTokenAddresses[typeIndex]).tokenOfOwnerByIndex(
             address(this),
             0
           );
           IERC721EnumerableUpgradeable(vars.airdropTokenAddresses[typeIndex]).safeTransferFrom(
             address(this),
             initiator,
             vars.airdropTokenId
           );
         }
       } else if (vars.airdropTokenTypes[typeIndex] == 3) {
         // ERC1155
         vars.airdropBalance = IERC1155Upgradeable(vars.airdropTokenAddresses[typeIndex]).balanceOf(
           address(this),
           vars.airdropTokenIds[typeIndex]
         );
         IERC1155Upgradeable(vars.airdropTokenAddresses[typeIndex]).safeTransferFrom(
           address(this),
           initiator,
           vars.airdropTokenIds[typeIndex],
           vars.airdropBalance,
           new bytes(0)
         );
       } else if (vars.airdropTokenTypes[typeIndex] == 4) {
         // ERC721 without Enumerate but can know the droped token id
         IERC721EnumerableUpgradeable(vars.airdropTokenAddresses[typeIndex]).safeTransferFrom(
           address(this),
           initiator,
           vars.airdropTokenIds[typeIndex]
         );
       } else if (vars.airdropTokenTypes[typeIndex] == 5) {
         // ERC721 without Enumerate and can not know the droped token id
       }
     }

     return true;
   }
```

Description

Xi_Zi : In AirdropFlashClaimReceiver. Sol: : executeOperation function, the parameters of the initiator, operator and params are controlled by an attacker, `require(initiator == owner(), "not contract owner"); 'It can be bypassed and then passed' IERC721Upgradeable(nftAssets[idIdx]).setApprovalForAll(operator, true); 'Authorizing the nft to the operator address controlled by the attacker can result in damage to funds and subsequent execution of incoming params via functionCallWithValue to execute operations related to the malicious contract

```
function executeOperation(
    address[] calldata nftAssets,
    uint256[][] calldata nftTokenIds,
    address initiator,
    address operator,
    bytes calldata params
  ) external override returns (bool) {
    require(nftTokenIds.length > 0, "empty token list");
    require(initiator == owner(), "not contract owner");

        ExecuteOperationLocalVars memory vars;

    // decode parameters
    (
      vars.airdropTokenTypes,
      vars.airdropTokenAddresses,
      vars.airdropTokenIds,
      vars.airdropContract,
      vars.airdropParams,
      vars.ethValue
    ) = abi.decode(params, (uint256[], address[], uint256[], address, bytes, uint256));

    . . .
    for (uint256 idIdx = 0; idIdx < nftAssets.length; idIdx++) {
      IERC721Upgradeable(nftAssets[idIdx]).setApprovalForAll(operator, true);
    }

    // call project aidrop contract
    AddressUpgradeable.functionCallWithValue(
      vars.airdropContract,
      vars.airdropParams,
      vars.ethValue,
      "call airdrop method failed"
    );
   . . .
```

Recommendation

Xi_Zi : To fix this vulnerability, proper authentication and validation should be implemented in the executeOperation function to ensure that only legitimate users can claim airdrop tokens.

Client Response

Declined. The contract AirdropFlashClaimReceiver.sol utilizes the flash claim mechanism to borrow underlying assets from the operate contract before acquiring airdropped assets through the functionCallWithValue function. Therefore, 1. Attackers cannot obtain the underlying assets by forging the operate contract, which means they cannot acquire the airdropped assets. 2. Attackers can forge the operate contract to deceive this contract and gain authorization over the airdropped assets within the contract. However, this contract does not retain the assets because the airdropped assets obtained by the contract will be transferred to the user's address in a single transaction, preventing sandwich attacks.

TPL-4:Oracle implementation for ERC1155 collateral valudation is flawed, making the code impossible to support ERC1155 asset

Category	Severity	Client Response	Contributor
Logical	Critical	Mitigated	ladboy233

Code Reference

- code/contracts/libraries/core/DepositExecutor.sol#L135
- code/contracts/libraries/types/NFTReserve.sol#L93

```
93:amountInEth *= IERC1155(nft.tNFTAddress).balanceOf(user, 0);

135:ITERC1155(tNFT).mint(to, tokenIds[i], amounts[i]);
```

Description

ladboy233 : One of the use case of the protocol is to support the ERC1155 asset as collateral

As we can see, when deposit ERC1155, the corresponding tokenId is minted

```
 if (tokenType == ReserveConfigurator.TokenType.ERC20) {
          IERC20(nft).safeTransferFrom(msg.sender, tNFT, amounts[i]);
          ITERC20(tNFT).mint(to, amounts[i]);
        } else if (tokenType == ReserveConfigurator.TokenType.ERC721) {
          IERC721(nft).safeTransferFrom(msg.sender, tNFT, tokenIds[i]);
          ITERC721(tNFT).mint(to, tokenIds[i]);
        } else {
          IERC1155(nft).safeTransferFrom(msg.sender, tNFT, tokenIds[i], amounts[i], "");
          ITERC1155(tNFT).mint(to, tokenIds[i], amounts[i]);
        }
```

this calls:

```
    function mint(
      address to,
      uint256 tokenId,
      uint256 amount
    ) external override onlyLendingPool {
      _mint(to, tokenId, amount, "");
      _totalSupply += amount;
    }
```

However, when evaluating the collateral worth of the NFT

the function NFTReserve#getUserLiquidityETH is eventually called

```
    function getUserLiquidityETH(
      ReserveData storage nft,
      address underlying,
      address user,
      address oracle
    ) internal view returns (uint256) {
      ReserveConfiguration configuration = nft.configuration;
      ReserveConfigurator.TokenType tokenType = configuration.getTokenType();
      uint256 amountInEth = IPriceOracleGetter(oracle).getReserveAssetPrice(underlying);
      if (tokenType == ReserveConfigurator.TokenType.ERC1155) {
        amountInEth *= IERC1155(nft.tNFTAddress).balanceOf(user, 0);
      } else {
        amountInEth *= IERC721(nft.tNFTAddress).balanceOf(user);
      }
      unchecked {
        amountInEth /= 10 ** configuration.getDecimals();
      }
      return amountInEth;
    }
```

As we can see:

```
   amountInEth *= IERC1155(nft.tNFTAddress).balanceOf(user, 0);
```

the tokenId is hardcoded to 0 instead of a parameter tokenId that is passed in

in most case, the minted token id of the ERC1155 TNFT id is not 0, this just means that the deposited ERC1155 will not count towards the health factor and leads to massively wrong under-evaluation of the collateral worth, which can leads to false liquidation

Recommendation

ladboy233 : replace 0 to tokenid in amountInEth *= IERC1155(nft.tNFTAddress).balanceOf(user, 'tokenId');

Client Response

Mitigated. Currently, the project does not support staking ERC1155 tokens. The issue will be addressed when there is a need to support staking ERC1155 tokens.

TPL-5:The hard-coded value causes a DOS issue with a revert on the `executeOperation()` function of `AirdropFlashClaimReceiver`.

Category	Severity	Client Response	Contributor
DOS	**Critical**	**Fixed**	**grep-er**

Code Reference

- code/contracts/airdrop/AirdropFlashClaimReceiver.sol#L127
- code/contracts/airdrop/AirdropFlashClaimReceiver.sol#L130

```
127:for (uint256 i = 0; i < vars.airdropBalance; i++) {

130:0
```

Description

grep-er : Summary:

The `AirdropFlashClaimReceiver::executeOperation()` function reverts if there are more than 1 ERC721 tokens. All tokens except the first ERC721 are locked permanently in the contract because the index is hardcoded to `0`.

Proof of Concept (POC):

1. In the `executeOperation()` function, when `vars.airdropTokenTypes[typeIndex] = 2`, the total count of ERC721 NFTs in this contract is checked.
2. If `vars.airdropBalance` is greater than 1, it transfers the same `token_id` as the one at index 0.
3. However, since the NFT at index 0 is already transferred, the operation reverts. This is due to the hardcoding of index `0` in the `tokenOfOwnerByIndex` function.

```
        for (uint256 i = 0; i < vars.airdropBalance; i++) {
          vars.airdropTokenId = IERC721EnumerableUpgradeable(vars.airdropTokenAddresses[typeIndex]).
tokenOfOwnerByIndex(
            address(this),
            0//@audit reverts if vars.airdropBalance>1 as 0th index is hard coded so vars.airdropBAl
ance becomes useless.
          );
          IERC721EnumerableUpgradeable(vars.airdropTokenAddresses[typeIndex]).safeTransferFrom(
            address(this),
            initiator,
            vars.airdropTokenId
          );
        }
      }
```

Recommendation

grep-er :

```
        for (uint256 i = 0; i < vars.airdropBalance; i++) {
          vars.airdropTokenId = IERC721EnumerableUpgradeable(vars.airdropTokenAddresses[typeIndex]).
tokenOfOwnerByIndex(
            address(this),
--            0
++            i
          );
          IERC721EnumerableUpgradeable(vars.airdropTokenAddresses[typeIndex]).safeTransferFrom(
            address(this),
            initiator,
            vars.airdropTokenId
          );
        }
      }
```

Client Response

Fixed

TPL-6:Use safeTransfer instead of transfer in TERC721 contract claimERC20Airdrop function

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	ginlee

Code Reference

- code/contracts/tokens/CertificateTokens/TERC721.sol#L121
- code/contracts/airdrop/AirdropFlashClaimReceiver.sol#L122
- code/contracts/airdrop/AirdropFlashClaimReceiver.sol#L201

```
121:IERC20Upgradeable(token).transfer(to, amount);

122:IERC20Upgradeable(vars.airdropTokenAddresses[typeIndex]).transfer(initiator, vars.airdropBalance);

201:IERC20Upgradeable(token).transfer(to, amount);
```

Description

ginlee : In TERC721 contract claimERC20Airdrop function and AirdropFlashClaimReceiver contract executeOperation & transferERC20 function, transfer is used instead of safeTransfer

```
IERC20Upgradeable(vars.airdropTokenAddresses[typeIndex]).transfer(initiator, vars.airdropBalance)
IERC20Upgradeable(token).transfer(to, amount)
IERC20Upgradeable(vars.airdropTokenAddresses[typeIndex]).transfer(initiator, vars.airdropBalance)
```

The ERC20.transfer() functions return a boolean value indicating success. This parameter needs to be checked for success. Some tokens do not revert if the transfer failed but return false instead.

Recommendation

ginlee : Use OpenZeppelin's SafeERC20 versions with the safeTransfer functions that handle the return value check as well as non-standard-compliant tokens

Client Response

Fixed

TPL-7: potential reorg attack in UserFlashclaimRegistry contract _createReceiver function

Category	Severity	Client Response	Contributor
Logical	**Medium**	**Fixed**	**ginlee**

Code Reference

- code/contracts/airdrop/UserFlashclaimRegistry.sol#L57

```
57:address payable receiver = payable(receiverImplemention.clone());
```

Description

ginlee : The clone function generates a random seed based on the current timestamp and uses this seed to create a new contract instance. However, since the timestamp can be predicted by attackers, they can modify the timestamp by executing a Reorg attack and recreate the same seed in the future, resulting in the creation of the same contract address. This means that attackers can deploy another contract at the same address before the contract creation, allowing them to execute malicious operations, such as stealing funds, on the same address An attacker can steal funds via a reorg attack if a contract is funded within a few blocks of being created inside a factory

Recommendation

ginlee : Utilize cloneDeterministic rather than clone To address this issue, Solidity introduced the cloneDeterministic function. This function allows specifying an explicit seed parameter during contract creation instead of using the timestamp as a random seed. By using an explicit seed, the contract's address and state become predictable and are not affected by the timestamp cloneDeterministic uses the opcode and a salt to deterministically deploy the clone. Using the same implementation and salt multiple times will revert since the clones cannot be deployed twice at the same address When using cloneDeterministic function, be careful of this issue below: https://github.com/code-423n4/2023-04-caviar-findings/issues/419

Client Response

Fixed

TPL-8:PriceOracle will use the wrong price if the Chainlink registry returns price outside min/max range

Category	Severity	Client Response	Contributor
Oracle Manipulation	Medium	Acknowledged	ginlee, ladboy233

Code Reference

- code/contracts/oracle/TakerOracleGetter.sol#L50

```
50:(, int256 price, , , ) = _tokenAggregators[asset].latestRoundData();
```

Description

ginlee : Chainlink aggregators have a built in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value (i.e. LUNA crash) the price of the oracle will continue to return the minPrice instead of the actual price of the asset. This would allow user to continue borrowing with the asset but at the wrong price. This is exactly what happened to Venus on BSC when LUNA imploded. The wrong price may be returned in the event of a market crash, an adversary will then be able to borrow against the wrong price and incur bad debt to the protocol
ladboy233 : Chainlink aggregators have a built in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value (i.e. LUNA crash) the price of the oracle will continue to return the minPrice instead of the actual price of the asset. This would allow user to continue borrowing with the asset but at the wrong price. This is exactly what happened to Venus on BSC when LUNA imploded.

```
  function getReserveAssetPrice(address asset) public view returns (uint256) {
    if (asset == weth) {
      return 1e18;
    }
    if (prices[asset] != 0) {
      return prices[asset];
    }
    require(address(_tokenAggregators[asset]) != address(0), "no price exists");
    (, int256 price, , , ) = _tokenAggregators[asset].latestRoundData();
    return uint256(price);
  }
```

the code does not validate if the returned price is within the accpetable range

Recommendation

ginlee :

```
  require(price >= minPrice && price <= maxPrice, "invalid price");
```

use the proper minPrice and maxPrice for each asset

Also, please refer to this article for further use of chainlink interface, such as if contract will be deployed in layer 2, please check the sequencer is not down before asking for price https://0xmacro.com/blog/how-to-consume-chainlink-price-feeds-safely/

ladboy233 : Implement the proper check for each asset. It must revert in the case of bad price.

Client Response

Acknowledged

TPL-9: `safeMint` is not used when depositing to the pool

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	SerSomeone

Code Reference

- code/contracts/tokens/InterestBearingTokens/ScaledERC20.sol#L74
- code/contracts/libraries/core/DepositExecutor.sol#L129
- code/contracts/libraries/core/DepositExecutor.sol#L90

```
74:super._mint(to, scaledTokenAmount);

90:ITToken(tToken).mint(to, amount, reserve.liquidityIndex);

129:ITERC20(tNFT).mint(to, amounts[i]);
```

Description

SerSomeone : According to EIP-721 it is necessary to call the receiving contract when an NFT is minted/transfered

As per the EIP-721:

`A wallet/broker/auction application MUST implement the wallet interface if it will accept safe transfers.`

Reference: https://eips.ethereum.org/EIPS/eip-721

The use of `mint` can be seen in `depositNFTs` and `deposit`

```
    function deposit(
      Reserve.ReserveData storage reserve,
      uint256 amount,
      address asset,
      address to
    ) external {
  -------
      ITToken(tToken).mint(to, amount, reserve.liquidityIndex); // @audit safemint not used
  -------
    }
```

`ITToken.mint()` ends up with the following logic (which is not `safeMint`):

```
    function _mintWithScaling(
      address to,
      uint256 amount,
      uint256 scaleFactor
    ) internal returns (bool) {
  -----
      super._mint(to, scaledTokenAmount);
  -----
    }
```

This is also discouraged by OpenZeppelin: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol#L276

Users might lose their NFT since it will bypass accounting

Recommendation

SerSomeone : use `_safeMint` instead and add a reentrancyGuard since this opens up reentrancy attack vector

Client Response

Fixed

TPL-10:weird ERC20 tokens will result in liquidity issues and protocol insolvency

Category	Severity	Client Response	Contributor
Logical	**Medium**	**Fixed**	**SerSomeone**

Code Reference

- code/contracts/libraries/core/DepositExecutor.sol#L89
- code/contracts/libraries/core/DepositExecutor.sol#L180

```
89:IERC20(asset).safeTransferFrom(msg.sender, tToken, amount);

180:ITToken(tToken).burn(msg.sender, to, withdrawAmount, reserve.liquidityIndex);
```

Description

SerSomeone : Its important to state that weird ERC20 tokens are not uncommon and should be addressed. The protocol addresses incorrect compliance of return values by using SafeERC20

HOWEVER - balance checks are not included in cases of fee-on-transfer or rebasing tokens are used https://github.com/d-xo/weird-erc20#fee-on-transfer

For example, `USDT` has a fee-on-transfer capability built in that can be enabled at any time `USDC` can be updated to include it.

If a fee-on-transfer token is deposited - the sender will have more `tTokens` then the amount of assets received in the pool

```
  function deposit(
    Reserve.ReserveData storage reserve,
    uint256 amount,
    address asset,
    address to
  ) external {
------
    IERC20(asset).safeTransferFrom(msg.sender, tToken, amount);
    ITToken(tToken).mint(to, amount, reserve.liquidityIndex);
------ // @audit When fee-on-transfer tokens are used more tToken will be minted then the received assets
  }
```

This will create a liquidity issue where users can withdraw more then allowed effectively withdrawing other users liquidity.

Recommendation

SerSomeone : Change the logic in `deposit` to include the actual received assets.

```
uint256 balanceBefore = IERC20(asset).balanceOf(address(this));
IERC20(asset).safeTransferFrom(msg.sender, tToken, amount);
uint256 receivedAmount = IERC20(asset).balanceOf(address(this)) - balanceBefore;

ITToken(tToken).mint(to, receivedAmount, reserve.liquidityIndex);
```

Client Response

Fixed

TPL-11:An off-by-one error in the UserNftConfiguration::setUsingAsCollateral() library leads to one valuable NFT becoming inoperable.

Category	Severity	Client Response	Contributor
Logical	**Medium**	**Fixed**	**grep-er**

Code Reference

- code/contracts/libraries/types/UserNftConfiguration.sol#L31
- code/contracts/libraries/types/UserNftConfiguration.sol#L55

```
31:function setUsingAsCollateral(

55:function isUsingAsCollateral(
```

Description

grep-er : On function `setUsingAsCollateral()` The checks for index start from `0 < reserveIdx && reserveIdx < ReserveConfigurator.MAX_NUMBER_NFT_RESERVES` And MAX_NUMBER_NFT_RESERVES is constant = 256

```
function setUsingAsCollateral(
UserNftConfiguration configuration,
uint256 reserveIdx,
bool usingAsCollateral
) internal pure returns (UserNftConfiguration) {
require(
0 < reserveIdx && reserveIdx < ReserveConfigurator.MAX_NUMBER_NFT_RESERVES,// @audit MAX_NUMBER_NFT_RESERVES = 256; declared on ReserveConfiguration library
Errors.genErrMsg(COMPONENT_NAME, Errors.INVALID_INDEX)
);
uint256 position = reserveIdx - 1; //@audit possible values of variable position = [0,154] and can't accommodate 255th nft
uint256 bitMask = 1 << position;
if (usingAsCollateral) {
return UserNftConfiguration.wrap(UserNftConfiguration.unwrap(configuration) | bitMask);
} else {
return UserNftConfiguration.wrap(UserNftConfiguration.unwrap(configuration) & ~bitMask);
}
}
```

As Shown in above code possible values of variable `position` = [0,254] and can't accommodate 255th index or 256th nft

Same problem is also with `isUsingAsCollateral()` on same contract.

Recommendation

grep-er :

```
require(

--0 < reserveIdx && reserveIdx < ReserveConfigurator.MAX_NUMBER_NFT_RESERVES,
++0 < reserveIdx && reserveIdx <= ReserveConfigurator.MAX_NUMBER_NFT_RESERVES,
Errors.genErrMsg(COMPONENT_NAME, Errors.INVALID_INDEX)

);
```

Client Response

Fixed

TPL-12:Liquidator is not able to liquidate using ETH in `PunkGateway` if not exact ETH is passed

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	SerSomeone

Code Reference

- code/contracts/pool/gateways/PunkGateway.sol#L155

```
155:revert("Receive not allowed");
```

Description

SerSomeone : The `PunkGateway` allows a liquidator to liquidate using ETH by calling the `liquidateWithEth` function

```
    function liquidateWithEth(
      address pool,
      uint256 punkIndex,
      address user
    ) external payable override {
      uint256 returnEth = WETH_GATEWAY.liquidate{value: msg.value}(
        pool,
        address(WPUNK),
        punkIndex,
        user,
        false
      );
      WPUNK.burn(punkIndex);
      PUNK.transferPunk(msg.sender, punkIndex);
      (bool success, ) = msg.sender.call{value: returnEth}("");
      require(success, Errors.genErrMsg(COMPONENT, Errors.NOT_PUNK_OWNER));
    }
```

The function calls the `WETH_GATEWAY.liquidate` to perform the liquidation using ETH. As can be seen `WETH_GATEWAY.liquidate` returns an amount of ETH that was refunded to the caller (in this case the PunkGateway) and then `liquidateWithEth` will return the remaining funds to `msg.sender`.

`WETH_GATEWAY.liquidate`:

```
    function liquidate(
      address pool,
      address nft,
      uint256 tokenId,
      address user,
      bool receiveTNFT
    ) external payable override returns (uint256) {
  --------
      // return remaining ETH
      uint256 returnEth = 0;
      if (msg.value > liquidateAmount) {
        returnEth = msg.value - liquidateAmount;
        _transferETH(msg.sender, returnEth);
      }
      return returnEth;
    }
```

HOWEVER - `PunkGateway` is not able to receive ETH:

```
    receive() external payable {
      revert("Receive not allowed"); // @audit will not work with liquidateWithEth
    }
```

Therefore the function will revert

Recommendation

SerSomeone : Change the `receive` function in `PunkGateway` to accept funds from `WETH_GATEWAY`

```
    receive() external payable {
      require(msg.sender == address(WETH_GATEWAY), "PUNKGW_RECEIVE_NOT_ALLOWED");
    }
```

Client Response

Fixed

TPL-13:AidropDistribution does not comply with chainlink VRF security guide

Category	Severity	Client Response	Contributor
Race Condition	Medium	Acknowledged	ladboy233

Code Reference

- code/contracts/airdrop/AirdropDistribution.sol#L212

```
212:function requestVRFRandomWords(uint256 airdropId) public onlyOwner nonReentrant {
```

Description

ladboy233 : https://docs.chain.link/vrf/v2/security/#choose-a-safe-block-confirmation-time-which-will-vary-between-blockchains

The AirdropDistriution.sol has a feature to let admin request randomness seed using chainlink VRF but the implementation does not comply with a few security consideration

1. To use the chianlink VRF, the admin needs to pay the fee using LINK token

according to security guide

Use VRFConsumerBaseV2 in your contract, to interact with the VRF service

Use VRFv2WrapperConsumer.sol in your contract, to interact with the VRF service

The contract aidropDistribution implements none.

2. the security guide does not recommend re-request randomness

https://docs.chain.link/vrf/v2/security/#do-not-re-request-randomness

```
    function requestVRFRandomWords(uint256 airdropId) public onlyOwner nonReentrant {
       AirdropData storage data = airdropDatas[airdropId];
       require(data.airdropId != 0, "invalid airdrop id");
       require(data.claimType == CLAIM_TYPE_RANDOM, "claim type not random");

       data.vrfRequestId = vrfCoordinator.requestRandomWords(
         vrfKeyHash,
         vrfSubscriptionId,
         vrfRequestConfirmations,
         vrfCallbackGasLimit,
         vrfNumWords
       );

       vrfAllRequestIds.push(data.vrfRequestId);

       vrfReqIdToAirdropIds[data.vrfRequestId] = airdropId;
     }
```

let us examine the sequence action below:

1. the admin request VRF, and has the vrfRrequestId 1000, the request is pending and does not fulfilled yet
2. the admin request VRF again, has the vrfRequestId 1010, the request is pending and does not fufilled yet

Any re-request of randomness is an incorrect use of VRFv2. Doing so would give the VRF service provider the option to withhold a VRF fulfillment if the outcome is not favorable to them and wait for the re-request in the hopes that they get a better outcome, similar to the considerations with block confirmation time.

the VRF fulfillment provider can pick and choose which random seed map to which vrfRequestId, he choose to map a result to id 1000 or id 1010, which undermine the fairness of the randomness

3 Don't accept bids/bets/inputs after you have made a randomness request

https://docs.chain.link/vrf/v2/security/#dont-accept-bidsbetsinputs-after-you-have-made-a-randomness-request

however, while the randomness request is still pending

whenever an outcome in your contract depends on some user-supplied inputs and randomness, the contract should not accept any additional user-supplied inputs after it submits the randomness request.

Otherwise, the cryptoeconomic security properties may be violated by an attacker that can rewrite the chain.

the vars.airdropTokenBalance is considered user-supplied and can be manipulated

```
 vars.airdropTokenBalance = IERC721Upgradeable(data.airdropTokenAddress).balanceOf(address(this));
        vars.randomIndex = _calcRandomIndex(randomWord, vars.airdropTokenBalance);
        vars.airdropTokenId = IERC721EnumerableUpgradeable(data.airdropTokenAddress).tokenOfOwnerByIndex(
          address(this),
          vars.randomIndex
        );
```

Recommendation

ladboy233 : Make sure the implementatoin comply with chainlink VRF guide to make sure the randomness generation process is fair

also, make sure only one random request is pending and do not re-request randomness request until the previous randomness request is filled

make sure block claimERC721 and claimERC1155 if there is a pending randomness request

Client Response

Acknowledged

TPL-14:If the price is not updated in time, the user's funds may be damaged in `TakerOracleGetter::getReserveAssetPrice()` function

Category	Severity	Client Response	Contributor
Oracle Manipulation	Medium	Fixed	ladboy233, Xi_Zi

Code Reference

- code/contracts/oracle/TakerOracleGetter.sol#L42-L57
- code/contracts/oracle/TakerOracleGetter.sol#L46

```
42:function getReserveAssetPrice(address asset) public view returns (uint256) {
43:    if (asset == weth) {
44:      return 1e18;
45:    }
46:    if (prices[asset] != 0) {
47:      return prices[asset];
48:    }
49:    require(address(_tokenAggregators[asset]) != address(0), "no price exists");
50:    (, int256 price, , , ) = _tokenAggregators[asset].latestRoundData();
51:    return uint256(price);
52:  }
53:
54:  function setPrice(address asset, uint256 price) external onlyRole(PRICE_UPDATER) {
55:    prices[asset] = price;
56:    emit NewPrice(asset, price);
57:  }

46:if (prices[asset] != 0) {
```

Description

ladboy233 : Hardcode asset price oracle is very dangerous while the underlying asset depegs

```
    function getReserveAssetPrice(address asset) public view returns (uint256) {
      if (asset == weth) {
        return 1e18;
      }
      if (prices[asset] != 0) {
        return prices[asset];
      }
      require(address(_tokenAggregators[asset]) != address(0), "no price exists");
      (, int256 price, , , ) = _tokenAggregators[asset].latestRoundData();
      return uint256(price);
    }
```

the oracle price has the option to let admin hardocde asset price

```
        if (asset == weth) {
        return 1e18;
      }
      if (prices[asset] != 0) {
        return prices[asset];
      }
```

one of the use case is hardcode the stable price or other token price that are use ETH as collateral to mint such as frxETH (frax ETH or CToken ETH from compound)

but when the underlying asset depegs in case of hack or extreme market condition,

it is very likely the fast will drop too fast for the admin to consistently update the price

and the because the oracle is used to value the asset worth of the collateral, the depg of the hardcoded asset will leads to liuqidation / over borrowing.

Xi_Zi : The setPrice function allows an address with the PRICE_UPDATER role to set custom prices for different assets. If an asset's price is set using the setPrice function and then remains unchanged for an extended period, it can result in inaccurate calculations of asset values throughout the contract. This is because the code primarily relies on the getReserveAssetPrice function to fetch asset prices, and custom prices set through setPrice may not be updated in a timely manner.

```
function getReserveAssetPrice(address asset) public view returns (uint256) {
    if (asset == weth) {
      return 1e18;
    }
    if (prices[asset] != 0) {
      return prices[asset];
    }
    require(address(_tokenAggregators[asset]) != address(0), "no price exists");
    (, int256 price, , , ) = _tokenAggregators[asset].latestRoundData();
    return uint256(price);
  }

  function setPrice(address asset, uint256 price) external onlyRole(PRICE_UPDATER) {
    prices[asset] = price;
    emit NewPrice(asset, price);
  }
```

The exploitation of this vulnerability could lead to significant financial losses for users. Borrowers might find themselves unable to manage their positions accurately due to incorrect calculations of collateral values. Lenders might also be affected, as they rely on accurate collateral values to ensure safety margins for their loans.

Recommendation

ladboy233 : Do not hardcode asset price when fetching price from oracle, Uniswap V3 TWAP is another option other than chainlink

Xi_Zi : To prevent this vulnerability, the contract should implement mechanisms to ensure that custom prices set through the setPrice function are updated promptly and accurately across all relevant calculations within the contract. Additionally, there should be proper checks and balances for the use of the PRICE_UPDATER role and monitoring of price updates.

Client Response

Fixed

TPL-15:Logical error in `WETHGateway::repay()` function

Category	Severity	Client Response	Contributor
Logical	**Medium**	**Fixed**	**Xi_Zi**

Code Reference

- code/contracts/pool/gateways/WETHGateway.sol#L74-L95
- code/contracts/pool/lendingpool/LendingPool.sol#L148-L149
- code/contracts/libraries/core/BorrowExecutor.sol#L111-L139

```
function repay(address pool, address to) external payable override {
    IDebtToken wethDebt = IDebtToken(
      ILendingPool(pool).getReserveData(address(WETH)).debtTokenAddress
    );
    uint256 debt = wethDebt.compoundedBalanceOf(msg.sender);
    uint256 repayAmount = debt;
    if (msg.value < debt) {
      repayAmount = msg.value;
    }
    require(
      msg.value >= repayAmount,
      Errors.genErrMsg(COMPONENT, Errors.INSUFFICIENT_REAPY_ETH_BALANCE)
    );
    WETH.deposit{value: repayAmount}();
    WETH.approve(pool, repayAmount);
    ILendingPool(pool).repay(address(WETH), repayAmount, to);

    // return remaining ETH
    if (msg.value > repayAmount) {
      _transferETH(msg.sender, msg.value - repayAmount);
    }
  }

function repay(
    Reserve.ReserveData storage reserve,
    mapping(address => UserConfiguration) storage userConfigs,
    address asset,
    uint256 amount,
    address to
  ) external returns (uint256) {
    reserve.updateState();
    uint256 debt = IDebtToken(reserve.debtTokenAddress).compoundedBalanceOf(to);

    Validator.validateRepay(reserve, amount, debt);

    if (amount > debt) {
      amount = debt;
    }
    IDebtToken(reserve.debtTokenAddress).burn(to, amount, reserve.debtIndex);

    address tToken = reserve.tTokenAddress;
    reserve.updateInterestRates(asset, tToken);

    if (debt - amount == 0) {
      userConfigs[to] = userConfigs[to].setBorrowing(reserve.id, false);
    }

    IERC20(asset).safeTransferFrom(msg.sender, tToken, amount);
    emit Repaid(asset, msg.sender, to, amount);

    return amount;
  }

function repay(address asset, uint256 amount, address to) external override returns (uint256) {
    return BorrowExecutor.repay(_reserves[asset], _userConfigs, asset, amount, to);
```

Description

Xi_Zi : In the WETHGateway::repay(address pool, address to) function call, the to address is passed to repay operation, but in the function, the debt is calculated according to msg.sender.

```
function repay(address pool, address to) external payable override {
    IDebtToken wethDebt = IDebtToken(
      ILendingPool(pool).getReserveData(address(WETH)).debtTokenAddress
    );
    uint256 debt = wethDebt.compoundedBalanceOf(msg.sender);
    uint256 repayAmount = debt;
    . . .
    ILendingPool(pool).repay(address(WETH), repayAmount, to);
    . . .
  }
```

The pool::repay function is passed using the repayAmount of msg.sender, and the to address is not necessarily msg.sender,

```
  function repay(address asset, uint256 amount, address to) external override returns (uint256) {
      return BorrowExecutor.repay(_reserves[asset], _userConfigs, asset, amount, to);
    }

  function repay(
      Reserve.ReserveData storage reserve,
      mapping(address => UserConfiguration) storage userConfigs,
      address asset,
      uint256 amount,
      address to
    ) external returns (uint256) {
      reserve.updateState();
      uint256 debt = IDebtToken(reserve.debtTokenAddress).compoundedBalanceOf(to);

      Validator.validateRepay(reserve, amount, debt);

      if (amount > debt) {
        amount = debt;
      }
      IDebtToken(reserve.debtTokenAddress).burn(to, amount, reserve.debtIndex);

      address tToken = reserve.tTokenAddress;
      reserve.updateInterestRates(asset, tToken);

      if (debt - amount == 0) {
        userConfigs[to] = userConfigs[to].setBorrowing(reserve.id, false);
      }

      IERC20(asset).safeTransferFrom(msg.sender, tToken, amount);
      emit Repaid(asset, msg.sender, to, amount);

      return amount;
    }
```

Then the debt to the address will be calculated and the related debt token will be burned to repay. If the address of msg.sender and to are inconsistent, the amount of debts of the two is also inconsistent, and the related logic is incorrect, which may lead to an error.

Recommendation

Xi_Zi : It is recommended to unify the msg.sender and TO addresses when repay, or optimize the related logic

Client Response

Fixed

TPL-16:Centralized Risk With Coin Transfer in lendingPool::claimAidrop

Category	Severity	Client Response	Contributor
Privilege Related	Medium	Declined	Xi_Zi

Code Reference

- code/contracts/pool/lendingpool/LendingPool.sol#L565-L597

```
function claimERC20Airdrop(
    address nftAssert,
    address token,
    address to,
    uint256 amount
  ) external override nonReentrant onlyLendingPoolConfigurator {
    address tNFTAddress = _nftReserves[nftAssert].tNFTAddress;
    ITERC721(tNFTAddress).claimERC20Airdrop(token, to, amount);
  }

  /// @inheritdoc ILendingPool
  function claimERC721Airdrop(
    address nftAssert,
    address token,
    address to,
    uint256[] calldata ids
  ) external override nonReentrant onlyLendingPoolConfigurator {
    address tNFTAddress = _nftReserves[nftAssert].tNFTAddress;
    ITERC721(tNFTAddress).claimERC721Airdrop(token, to, ids);
  }

  /// @inheritdoc ILendingPool
  function claimERC1155Airdrop(
    address nftAssert,
    address token,
    address to,
    uint256[] calldata ids,
    uint256[] calldata amounts,
    bytes calldata data
  ) external override nonReentrant onlyLendingPoolConfigurator {
    address tNFTAddress = _nftReserves[nftAssert].tNFTAddress;
    ITERC721(tNFTAddress).claimERC1155Airdrop(token, to, ids, amounts, data);
  }
```

Description

Xi_Zi : IIn the lendingPool contract, claimERC721Airdrop, claimERC20Airdrop, claimERC1155Airdrop can directly call claim airdrop in the corresponding contract to transfer funds to the TO address. But only the privileged accounts onlyLendingPoolConfigurator can call, there is a strong centralized risk, which means that the contract is controlled by a single address. If the address is compromised, the contract will be compromised.

```
  function claimERC20Airdrop(
    address nftAssert,
    address token,
    address to,
    uint256 amount
  ) external override nonReentrant onlyLendingPoolConfigurator {
    address tNFTAddress = _nftReserves[nftAssert].tNFTAddress;
    ITERC721(tNFTAddress).claimERC20Airdrop(token, to, amount);
  }

  /// @inheritdoc ILendingPool
  function claimERC721Airdrop(
    address nftAssert,
    address token,
    address to,
    uint256[] calldata ids
  ) external override nonReentrant onlyLendingPoolConfigurator {
    address tNFTAddress = _nftReserves[nftAssert].tNFTAddress;
    ITERC721(tNFTAddress).claimERC721Airdrop(token, to, ids);
  }

  /// @inheritdoc ILendingPool
  function claimERC1155Airdrop(
    address nftAssert,
    address token,
    address to,
    uint256[] calldata ids,
    uint256[] calldata amounts,
    bytes calldata data
  ) external override nonReentrant onlyLendingPoolConfigurator {
    address tNFTAddress = _nftReserves[nftAssert].tNFTAddress;
    ITERC721(tNFTAddress).claimERC1155Airdrop(token, to, ids, amounts, data);
  }



function claimERC20Airdrop(
    address token,
    address to,
    uint256 amount
  ) external override onlyLendingPool {
    require(token != UNDERLYING_ASSET, "TERC721: token can not be underlying asset");
    require(token != address(this), "TERC721: token can not be self address");
IERC20Upgradeable(token).transfer(to, amount);
    emit ClaimERC20Airdrop(token, to, amount);
  }

  function claimERC721Airdrop(
    address token,
    address to,
    uint256[] calldata ids
  ) external override onlyLendingPool {
    require(token != UNDERLYING_ASSET, "TERC721: token can not be underlying asset");
    require(token != address(this), "TERC721: token can not be self address");
    for (uint256 i = 0; i < ids.length; i++) {
      IERC721Upgradeable(token).safeTransferFrom(address(this), to, ids[i]);
    }
    emit ClaimERC721Airdrop(token, to, ids);
  }

  function claimERC1155Airdrop(
    address token,
    address to,
    uint256[] calldata ids,
    uint256[] calldata amounts,
    bytes calldata data
  ) external override onlyLendingPool {
    require(token != UNDERLYING_ASSET, "TERC721: token can not be underlying asset");
    require(token != address(this), "TERC721: token can not be self address");
    IERC1155Upgradeable(token).safeBatchTransferFrom(address(this), to, ids, amounts, data);
    emit ClaimERC1155Airdrop(token, to, ids, amounts, data);
  }
```

Recommendation

Xi_Zi : Avoid using centralized risk contracts.

Client Response

Declined. The code is used for conducting a snapshot airdrop. In a snapshot airdrop, the assets are distributed to the holders based on their holdings at a specific snapshot time. In this case, the holder is the TNFT contract within the protocol. Therefore, the protocol administrator needs to claim the airdrop on behalf of the TNFT contract and then distribute the assets accordingly.

TPL-17:Possible lost `msg.value` in `WETHGateway` and `PunkGateway` contract `liquidateWithWeth` & `transfer` function

Category	Severity	Client Response	Contributor
Logical	Low	Declined	ginlee

Code Reference

- code/contracts/pool/gateways/PunkGateway.sol#L118-L122
- code/contracts/pool/gateways/WETHGateway.sol#L128-L134

```
118:IERC20(address(WETH_GATEWAY.WETH())).safeTransferFrom(
119:      msg.sender,
120:      address(this),
121:      liquidateAmount
122:    );

128:function transfer(address token, address to, uint256 amount) external onlyOwner {
129:    if (token == address(0)) {
130:      _transferETH(to, amount);
131:    } else {
132:      IERC20(token).safeTransfer(to, amount);
133:    }
134:  }
```

Description

ginlee : When transferring ERC20 tokens, it's crucial to ensure that the user doesn't send msg.value during that transaction. Otherwise, the value they send will be lost.

Recommendation

ginlee : Add a require(msg.value == 0) check for the above conditions

```
require(msg.value == 0, "ETH sent along ERC20 Tokens");
IERC20(token).safeTransfer(to, amount);
```

Client Response

Declined

TPL-18:price feed staleness in "TakerOracleGetter" contract "getReserveAssetPrice" function

Category	Severity	Client Response	Contributor
Oracle Manipulation	Low	Fixed	ginlee, ladboy233

Code Reference

- code/contracts/oracle/TakerOracleGetter.sol#L42-L52

```
42:function getReserveAssetPrice(address asset) public view returns (uint256) {
43:    if (asset == weth) {
44:      return 1e18;
45:    }
46:    if (prices[asset] != 0) {
47:      return prices[asset];
48:    }
49:    require(address(_tokenAggregators[asset]) != address(0), "no price exists");
50:    (, int256 price, , , ) = _tokenAggregators[asset].latestRoundData();
51:    return uint256(price);
52:  }
```

Description

ginlee :

```
require(address(_tokenAggregators[asset]) != address(0), "no price exists");
    (, int256 price, , , ) = _tokenAggregators[asset].latestRoundData();
    return uint256(price);
```

it does not check for price feed staleness. It has happened before - a feed stops updating the price and returns a stale one

ladboy233 : Chainlink's latestRoundData is used here to retrieve price feed data; however, there is insufficient protection against price staleness.

Return arguments other than int256 answer are necessary to determine the validity of the returned price, as it is possible for an outdated price to be received. See here for reasons why a price feed might stop updating.

The return value updatedAt contains the timestamp at which the received price was last updated, and can be used to ensure that the price is not outdated. See more information about latestRoundID in the Chainlink docs. Inaccurate price data can lead to functions not working as expected and/or loss of funds.

if the stale price is used to value the collateral, the user can get liquidated wrongly ,or the user can overborrow and create bad debt for protocol

Recommendation

ginlee : validate that no more than 1 hour(or any value you want to set) has passed from the updatedAt timestamp value returned from latestRoundData, otherwise the transaction will revert.

```
  require (updatedAt >= block.timestamp - 3600, "stale price")
```

Also, a backup oracle option is suggested

ladboy233 : Add a check for the updatedAt returned value from latestRoundData.

Client Response

Fixed

TPL-19:Array length should be checked in AirdropFlashClaimReceiver contract executeOperation function

Category	Severity	Client Response	Contributor
Logical	**Low**	**Fixed**	**ginlee**

Code Reference

- code/contracts/airdrop/AirdropFlashClaimReceiver.sol#L69-L70

```
69:address[] calldata nftAssets,
70:    uint256[][] calldata nftTokenIds,
```

Description

ginlee : Function executeOperation takes two dynamic arrays as param, nftAssets length should be equal to nftTokenIds length, but this function doesn't have input validation. Before calling this function, it is necessary to ensure that the lengths of these two arrays are equal so that NFT contract addresses can be correctly matched with their corresponding token IDs. If the lengths are not equal, it may lead to erroneous NFT operations and even result in contract execution failure.

Recommendation

ginlee : add input validation for two dynamic arrays

```
require(nftAssets.length == nftTokenIds.length, "nftAssets and nftTokenIds lengths do not match");
```

Client Response

Fixed

TPL-20:Input validation should be added in "TERC20" contract "burn" function

Category	Severity	Client Response	Contributor
Logical	Low	Declined	ginlee

Code Reference

- code/contracts/tokens/CertificateTokens/TERC20.sol#L58-L69

```
function burn(
    address from,
    address to,
    uint256 amount
  ) external override onlyLendingPool {
    uint256 burnAmount = amount;
    if (amount == type(uint256).max) {
      burnAmount = balanceOf(from);
    }
    _burn(from, burnAmount);
    ERC20Upgradeable(UNDERLYING_ASSET).transfer(to, burnAmount);
  }
```

Description

ginlee : There is no sanity check for from and to, although the onlyLendingPool modifier ensures that only specific authorized contracts (Lending Pool contracts) can call this function, adding the condition from != to can further enhance security and prevent accidental situations where the same address is mistakenly passed. If the from address is equal to the to address, although the onlyLendingPool modifier prevents unauthorized contracts or addresses from calling this function, the contract will actually burn the tokens in the same address without implementing a token transfer. This could lead to permanent loss of tokens, as the tokens remain in the same address, but the balance is reduced accordingly

Recommendation

ginlee :

```
      require(from != to, "Invalid transfer: from and to addresses are the same");
```

add check for from and to

Client Response

Declined

TPL-21:Recorded events are inaccurate in `DepositExecutor::deposit()` function

Category	Severity	Client Response	Contributor
Code Style	Low	Fixed	Xi_Zi

Code Reference

- code/contracts/libraries/core/DepositExecutor.sol#L94

```
94:emit Deposited(asset, msg.sender, to, amount);
```

Description

Xi_Zi : In the DepositExecutor contract deposit function, the event record is implemented according to msg.sender. However, if the msg.sender of the user coming from WETHGateway::deposit is a WETHGateway contract, tx.origin is a good option here to accurately record the deposit status of all users

```
function deposit(
    Reserve.ReserveData storage reserve,
    uint256 amount,
    address asset,
    address to
  ) external {
    Validator.performBaseCheck(reserve.configuration, amount, true);

    address tToken = reserve.tTokenAddress;
    uint256 currLiqIdx = reserve.updateState();

    uint256 depositCap = reserve.configuration.getDepositCap();
    require(
      depositCap == 0 ||
        (IERC20(tToken).totalSupply().rayMul(currLiqIdx) + amount) <=
        depositCap * (10 ** reserve.configuration.getDecimals()),
      Errors.genErrMsg(NAME, Errors.DEPOSIT_CAP_EXCEEDED)
    );
    IERC20(asset).safeTransferFrom(msg.sender, tToken, amount);
    ITToken(tToken).mint(to, amount, reserve.liquidityIndex);

    reserve.updateInterestRates(asset, tToken);

    emit Deposited(asset, msg.sender, to, amount);
  }
```

Recommendation

Xi_Zi : It is recommended to use tx.origin record, which can accurately record the deposit situation of all users.

```
emit Deposited(asset, tx.origin, to, amount);
```

Client Response

Fixed

TPL-22:Remove checks for scenarios which will be impossible in WETHGateway::repay()

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	grep-er, Xi_Zi

Code Reference

- code/contracts/pool/gateways/WETHGateway.sol#L74-L95
- code/contracts/pool/gateways/WETHGateway.sol#L83

```
74:function repay(address pool, address to) external payable override {
75:    IDebtToken wethDebt = IDebtToken(
76:      ILendingPool(pool).getReserveData(address(WETH)).debtTokenAddress
77:    );
78:    uint256 debt = wethDebt.compoundedBalanceOf(msg.sender);
79:    uint256 repayAmount = debt;
80:    if (msg.value < debt) {
81:      repayAmount = msg.value;
82:    }
83:    require(
84:      msg.value >= repayAmount,
85:      Errors.genErrMsg(COMPONENT, Errors.INSUFFICIENT_REAPY_ETH_BALANCE)
86:    );
87:    WETH.deposit{value: repayAmount}();
88:    WETH.approve(pool, repayAmount);
89:    ILendingPool(pool).repay(address(WETH), repayAmount, to);
90:
91:    // return remaining ETH
92:    if (msg.value > repayAmount) {
93:      _transferETH(msg.sender, msg.value - repayAmount);
94:    }
95:  }

83:require(
```

Description

grep-er : Summary: As on the line above it checks if `msg.value < debt` and if it is then it makes `repayAmount = msg.value;`

- So the next scenario where it is require to have `msg.value >= repayAmount` is already always true so the check of no use.

```
function repay(address pool, address to) external payable override {
IDebtToken wethDebt = IDebtToken(
ILendingPool(pool).getReserveData(address(WETH)).debtTokenAddress
);
uint256 debt = wethDebt.compoundedBalanceOf(msg.sender);
uint256 repayAmount = debt;
if (msg.value < debt) {
repayAmount = msg.value;
}
require(
msg.value >= repayAmount,
Errors.genErrMsg(COMPONENT, Errors.INSUFFICIENT_REAPY_ETH_BALANCE)//@audit irrelvent code snippet as it does nothign as it will never be used
);
WETH.deposit{value: repayAmount}();
WETH.approve(pool, repayAmount);
ILendingPool(pool).repay(address(WETH), repayAmount, to);

// return remaining ETH
if (msg.value > repayAmount) {
_transferETH(msg.sender, msg.value - repayAmount);
}
```

```
  }
```

Xi_Zi : In the WETHGateway::repay() function, `require(msg.value >= repayAmount,Errors.genErrMsg(COMPONENT,Errors.INSUFFICIENT_REAPY_ETH_BALANCE));` The validation `msg.value >= repayAmount` will always be true as per the current logic. This might lead to unnecessary gas consumption and potential confusion for anyone reading the code. While this doesn't necessarily create a security risk, it might be a code smell or an indication of a potential logic error.

```
  function repay(address pool, address to) external payable override {
      IDebtToken wethDebt = IDebtToken(
        ILendingPool(pool).getReserveData(address(WETH)).debtTokenAddress
      );
      uint256 debt = wethDebt.compoundedBalanceOf(msg.sender);
      uint256 repayAmount = debt;
      if (msg.value < debt) {  // if  (repayAmount <= debt )
        repayAmount = msg.value;
      }
      require(
        msg.value >= repayAmount,
        Errors.genErrMsg(COMPONENT, Errors.INSUFFICIENT_REAPY_ETH_BALANCE)
      );
    . . .
    }
```

Recommendation

grep-er :

```
uint256 debt = wethDebt.compoundedBalanceOf(msg.sender);

uint256 repayAmount = debt;

if (msg.value < debt) {

repayAmount = msg.value;

}

-- require(

-- msg.value >= repayAmount,

-- Errors.genErrMsg(COMPONENT, Errors.INSUFFICIENT_REAPY_ETH_BALANCE)//@audit irrelvent code snippet as it does nothign as it will never be used

-- );
```

Xi_Zi : Suggested deletion `require(msg.value >= repayAmount,Errors.genErrMsg(COMPONENT,Errors.INSUFFICIENT_REAPY_ETH_BALANCE));` , Because the conditions are always true

Client Response

Fixed

TPL-23:Usage of Builtin Symbols in `ILendingPool::getTNFTProxyAddress()` function

Category	Severity	Client Response	Contributor
Language Specific	**Low**	**Fixed**	**Xi_Zi**

Code Reference

- code/contracts/interfaces/ILendingPool.sol#L440-L442

```
440:function getTNFTProxyAddress(
441:    address assert
442:  ) external view returns (address);
```

Description

Xi_Zi : In `ILendingPool::getTNFTProxyAddress()` function , The parameter name is' assert 'and assert is Builtin Symbols，Builtin symbols may lead to unexpected results.

```
    function getTNFTProxyAddress(
      address assert
    ) external view returns (address);
```

Recommendation

Xi_Zi : It is proposed to be modified as

```
  function getTNFTProxyAddress(
      address asset
    ) external view returns (address);
```

Client Response

Fixed

TPL-24:Missing Zero Address Check in `LendingPool::initialize()` function

Category	Severity	Client Response	Contributor
Code Style	Low	Fixed	Xi_Zi

Code Reference

- code/contracts/pool/lendingpool/LendingPool.sol#L55-L58

```
55:function initialize(ITakerAddressesProvider provider) public initializer {
56:    ADDRESS_PROVIDER = provider;
57:    __ReentrancyGuard_init();
58:  }
```

Description

Xi_Zi : In LendingPool::initialize() function ,this Function is lack of zero address check in important operation,which may cause some unexpected result.

```
  function initialize(ITakerAddressesProvider provider) public initializer {
    ADDRESS_PROVIDER = provider;
    __ReentrancyGuard_init();
  }
```

Recommendation

Xi_Zi : Add check of zero address in important operation.

Client Response

Fixed

TPL-25:Gas Optimization: in `AirdropFlashClaimReceiver` and `PunkGateway` contract

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Fixed	Xi_Zi

Code Reference

- code/contracts/pool/gateways/PunkGateway.sol#L58-L61
- code/contracts/airdrop/AirdropFlashClaimReceiver.sol#L115

```
58:function deposit(address pool, uint256[] calldata punkIndexes, address to) external override {
59:    address[] memory nfts = new address[](punkIndexes.length);
60:    uint256[] memory amounts = new uint256[](punkIndexes.length);
61:    for (uint256 i = 0; i < punkIndexes.length; i++) {

115:for (uint256 typeIndex = 0; typeIndex < vars.airdropTokenTypes.length; typeIndex++) {
```

Description

Xi_Zi : In the deposit function of the PunkGateway.sol contract , located at lines 61, it is recommended to store the length of the punkIndexes array in a separate variable outside the loop. This modification can help optimize gas consumption by avoiding repeated length calculations within the loop. Similarly also AirdropFlashClaimReceiver executeOperation and so on.

```
function deposit(address pool, uint256[] calldata punkIndexes, address to) external override {
    address[] memory nfts = new address[](punkIndexes.length);
    uint256[] memory amounts = new uint256[](punkIndexes.length);
    for (uint256 i = 0; i < punkIndexes.length; i++) {

    }

  }
```

Recommendation

Xi_Zi : To optimize gas consumption, it is suggested to store the length of the punkIndexes array in a separate variable before the loop. This can be done as follows:

```
uint length = punkIndexes.length;
for (uint i = 0; i < length; i++) {

}
```

Client Response

Fixed

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.