Tokemak - Reserve and Controllers

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **April 4th, 2022 - April 19th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	04/18/2022	Francisco González
0.2	Draft Review	04/19/2022	Gabi Urrutia
1.0	Remediation Plan	07/04/2022	Francisco González
1.1	Remediation Plan Review	07/05/2022	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	halborn@protonmail.com
Steven Walbroehl	Halborn	halborn@protonmail.com
Gabi Urrutia	Halborn	halborn@protonmail.com
Roberto Reigada	Halborn	halborn@protonmail.com
Francisco González	Halborn	halborn@protonmail.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Tokemak engaged Halborn to conduct a security audit on their Reserve and Controller smart contracts beginning on April 4th, 2022, and ending on April 19th, 2022. The security assessment was scoped to the smart contracts provided in the Tokemak GitHub repository: Tokemak/tokemak-smart-contracts.

1.2 AUDIT SUMMARY

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

The purpose of this audit to achieve the following:

- Ensure that all functions in the protocol smart contracts are intended.
- Identify potential security issues in scoped smart contracts.

In summary, Halborn identified some security risks that were mostly addressed by the Tokemak 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 bridge code and can quickly identify items

that do not follow security best practices. The following phases and associated tools were used throughout the term of 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 hotspots 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.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

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

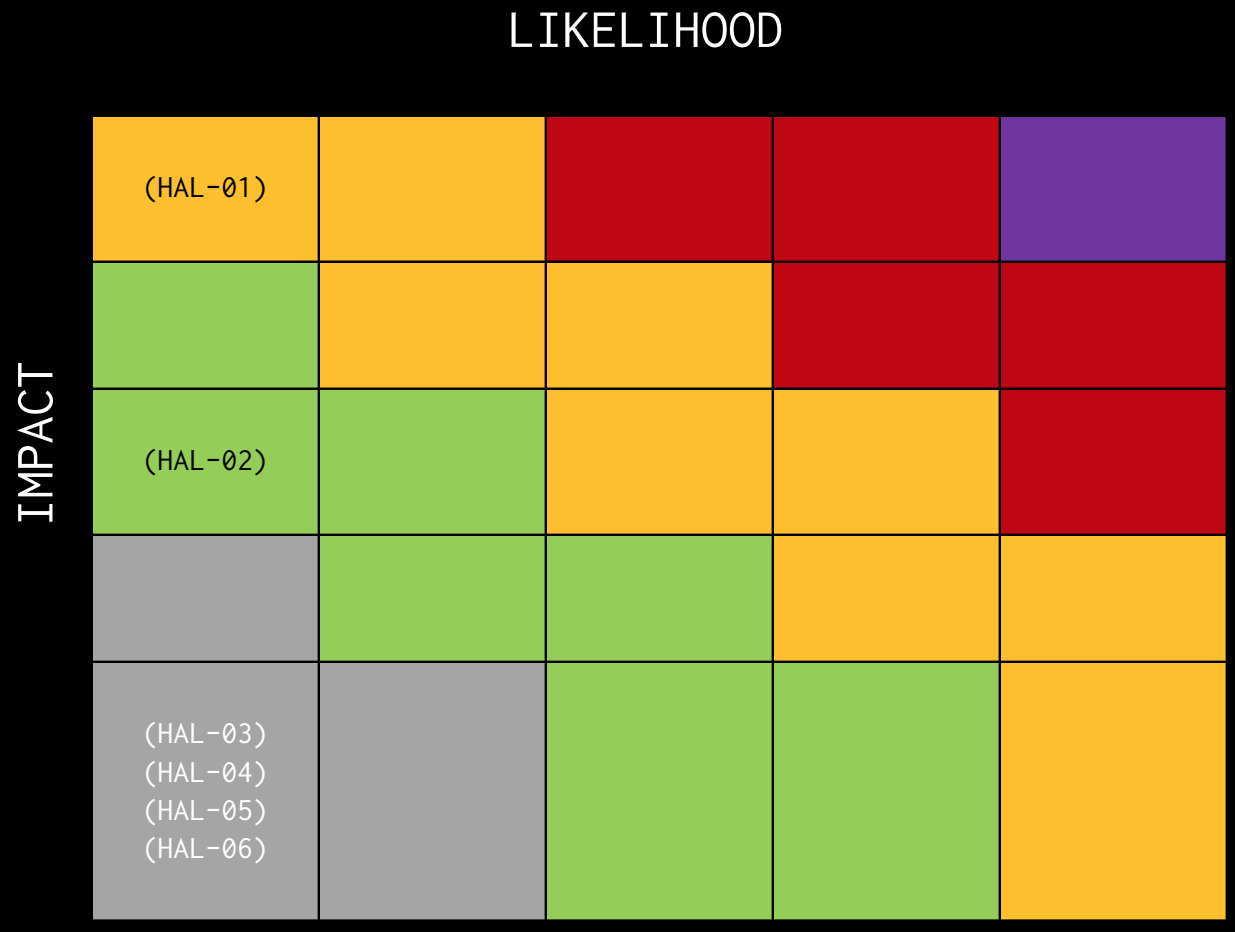
1.4 SCOPE

IN-SCOPE:
The security assessment was scoped to the following smart contracts:

- controllers/ConvexController.sol
- controllers/WethController.sol
- controllers/CurveControllerETH.sol
- controllers/PoolTransferController.sol
- pools/PCAPool.sol
- pools/PCAEthPool.sol
- pools/Pool.sol (controlledBurn() and registerBurner() functions)
- Initial Commit ID: f27dd74d8e56b221eaf11185dbb5acc7ad1642a3
- Remediations Commit ID: f737f0423d7408d72c06f8410b94d4691b8c66c4

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	1	4



EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - UNCHECKED TOKEN ADDRESS	Medium	SOLVED - 07/04/2022
HAL02 - EXPERIMENTAL FEATURES ENABLED	Low	RISK ACCEPTED
HAL03 - MULTIPLE ROLE-BASED ACCESS CONTROL MECHANISMS IMPLEMENTED	Informational	ACKNOWLEDGED
HAL04 - MULTIPLE INITIALIZATION CRITERIA	Informational	SOLVED - 07/04/2022
HAL05 - UNDERLYING TOKEN NOT ENFORCED	Informational	SOLVED - 07/04/2022
HAL06 - CONFUSING VARIABLE NAMING	Informational	SOLVED - 07/04/2022

FINDINGS & TECH DETAILS

3.1 (HAL-01) UNCHECKED TOKEN ADDRESS - MEDIUM

Description:

The WethController.sol contract is based on WETH token contract to wrap and unwrap ETH. This address is provided to the contract when it is deployed.

However, this address is never checked to be the legitimate WETH token, and any malicious administrator (with ADMIN_ROLE) could deploy this contract with a custom WETH contract and drain the ETH balance from the Manager.sol contract.

Code Location:

Listing 1: WethController.sol (Lines 12,18,21)

```
 9 contract WethController is BaseController {
10     using SafeMath for uint256;
11
12     IWETH public immutable weth;
13
14     constructor (
15         address manager,
16         address accessControl,
17         address registry,
18         IWETH wethContract
19     ) public BaseController(manager, accessControl, registry) {
20         require(address(wethContract) != address(0));
21         weth = wethContract;
22     }
```

Please note that weth address is never required to be registered in registry or accessControl contracts in the same fashion is checked in other controllers:

Listing 2: CurveControllerETH.sol (Line 57)

```
45      function deploy(
46          address poolAddress,
47          uint256[N_COINS] calldata amounts,
48          uint256 minMintAmount
49      ) external payable onlyManager onlyAddLiquidity {
50          address lpTokenAddress = _getLPToken(poolAddress);
51          uint256 amountsLength = amounts.length;
52
53          for (uint256 i = 0; i < amountsLength; i++) {
54              if (amounts[i] > 0) {
55                  address coin = IStableSwapPoolETH(poolAddress).
↳ coins(i);
56
57                  require(addressRegistry.checkAddress(coin, 0), "
↳ INVALID_COIN");
58
59                  uint256 balance = _getBalance(coin);
60
61                  require(balance >= amounts[i], "
↳ INSUFFICIENT_BALANCE");
62
63                  if (coin != ETH_REGISTRY_ADDRESS) {
64                      _approve(IERC20(coin), poolAddress, amounts[i
↳ ]);
65                  }
66              }
67          }
```

Risk Level:

Likelihood - 1
Impact - 5

Recommendation:

It is recommended to ensure that the addresses provided are legitimate. To prevent attacks like this from happening, make sure the provided address is registered in the Address Registry, or even consider hardcoding the WETH contract address.

Remediation Plan:

SOLVED: The Tokemak Team solved this issue by removing the previous weth address declaration and configuring the query from AddressRegistry contract instead.

Listing 3: WethController.sol (Lines 12,19)

```
 9 contract WethController is BaseController {
10     using SafeMath for uint256;
11
12     IWETH public immutable weth;
13
14     constructor (
15         address manager,
16         address accessControl,
17         address registry
18     ) public BaseController(manager, accessControl, registry) {
19         weth = IWETH(IAddressRegistry(registry).weth());
20     }
```

Remediation Pull Request ID: https://github.com/Tokemak/tokemak-smart-contracts/pull/357

3.2 (HAL-02) EXPERIMENTAL FEATURES ENABLED - LOW

Using experimental features could be dangerous on live deployments. The experimental ABI encoder does not handle non-integer values shorter than 32 bytes properly. This applies to bytesNN types, bool, enum and other types when they are part of an array or a struct and encoded directly from storage. This means these storage references have to be used directly inside abi.encode(. . .) as arguments in external function calls or in event data without prior assignment to a local variable. Using return does not trigger the bug. The types bytesNN and bool will result in corrupted data, while enum might lead to an invalid revert.

Also, arrays with elements shorter than 32 bytes may not be handled correctly, even if the base type is an integer type. Encoding such arrays in the way described above can lead to other data in the encoding being overwritten if the number of elements encoded is not a multiple of the number of elements that fit a single slot. If nothing follows the array in the encoding (note that dynamically-sized arrays are always encoded after statically-sized arrays with statically-sized content), or if only a single array is encoded, no other data is overwritten. There are known bugs that are publicly released while using this feature. However, the bug only manifests itself when all the following conditions are met:

1. Storage data involving arrays or structs is sent directly to an external function call, to abi.encode or to event data without prior assignment to a local (memory) variable.
2. There is an array that contains elements with size less than 32 bytes or a struct that has elements that share a storage slot or members of type bytesNN shorter than 32 bytes.

In addition to that, the code would not be affected in the following cases:

1. If all the structs or arrays only use uint256 or int256 types.

2. If only integer types (that may be shorter) are used and only encode at most one array at a time.
3. If only such data is returned and is not used in abi.encode, external calls or event data.

ABIEncoderV2 is enabled to be able to pass a struct type into a function, both web3 and in another contract. Naturally, any bug can have wildly varying consequences depending on the program control flow, but it is expected that this is more likely to lead to malfunction than exploitability. The bug, when triggered, will under certain circumstances send corrupt parameters on method invocations to other contracts.

Reference:

Solidity Optimizer and ABIEncoderV2 Bug

Code Location:

Listing 4

```
1 Pool.sol:3:pragma experimental ABIEncoderV2;
2 CurveControllerETH.sol:4:pragma experimental ABIEncoderV2;
3 ConvexController.sol:3:pragma experimental ABIEncoderV2;
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

When possible, do not use experimental features in the final live deployment. Validate and check that all the above conditions are true for integers and arrays (i.e., all using uint256).

Remediation Plan:

RISK ACCEPTED: The Tokemak Team accepted the risk of this issue.

3.3 (HAL-03) MULTIPLE ROLE-BASED ACCESS CONTROL MECHANISMS IMPLEMENTED - INFORMATIONAL

Description:

It has been detected that Role-Based Access Control functionalities have been implemented multiple times.

For example, controllers like ConvexController.sol will call hasRole() function from the _accessControl address supplied in the deployment to check for permissions, meanwhile contracts like Manager.sol also implement RBAC locally.

Code Location:

Listing 5: Pool.sol

```
90     function registerBurner(address burner, bool allowedBurner)
↳ external override onlyOwner {
91       require(burner != address(0), "INVALID_ADDRESS");
92       registeredBurners[burner] = allowedBurner;
93
94       emit BurnerRegistered(burner, allowedBurner);
95     }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to follow a uniform pattern when developing functionalities across different contracts. This is considered a good

practice, and having functionalities like RBAC implemented only once will significantly ease any further code modification or role management operation that might be necessary.

Remediation Plan:

ACKNOWLEDGED: The Tokemak Team acknowledged this issue and confirmed that Controllers are deployed with the accessControl variable set to Manager address.

3.4 (HAL-04) MULTIPLE INITIALIZATION CRITERIA - INFORMATIONAL

Description:

It has been observed that contracts such as PCAPool.sol and PCAEthPool.sol declare an constructor() function using the initializer modifier, which can also be found in the initialize() function. This function is usually declared to avoid manual contract deployment instead of using a proxy, rendering the contracts unusable since it would be initialized with null values. However, the constructor has not been declared in previous contracts such as Pool.sol.

Code Location:

Listing 6: PCAPool.sol (Lines 20,26)

```
20   constructor() public initializer {}
21
22   function initialize(
23     ILiquidityPool _pool,
24     string memory _name,
25     string memory _symbol
26   ) external initializer {
27     require(address(_pool) != address(0), "ZERO_ADDRESS");
28
29     __Context_init_unchained();
30     __Ownable_init_unchained();
31     __Pausable_init_unchained();
32     __ReentrancyGuard_init_unchained();
33     __ERC20_init_unchained(_name, _symbol);
34     __ERC20Pausable_init_unchained();
35
36     pool = _pool;
37     underlyer = pool.underlyer();
38     require(address(underlyer) != address(0), "POOL_DNE");
39   }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to follow a uniform pattern when defining deploying and initialization criteria along smart contracts. This is considered a good practice and will improve code readability and usability.

Remediation Plan:

SOLVED: The Tokemak Team solved this issue by standardizing the initialization criteria, adding the missing constructor to Pool.sol and EthPool.sol contracts.

Listing 7: Pool.sol (Line 69)

```
67     //@custom:oz-upgrades-unsafe-allow constructor
68     //solhint-disable-next-line no-empty-blocks
69     constructor() public initializer {}
```

Remediation Pull Request ID: https://github.com/Tokemak/tokemak-smart-contracts/pull/355

3.5 (HAL-05) UNDERLYING TOKEN NOT ENFORCED - INFORMATIONAL

Description:

It has been detected that PCAEthPool.sol contract does not enforce that the underlying token is wETH, which could cause every ETH transaction with this pool to fail if a pool with an underlying token different from wETH is mistakenly used during the initialization:

Code Location:

Listing 8: PCAEthPool.sol (Line 40)

```
25  function initialize(
26    ILiquidityEthPool _pool,
27    string memory _name,
28    string memory _symbol
29  ) external initializer {
30    require(address(_pool) != address(0), "ZERO_ADDRESS");
31
32    __Context_init_unchained();
33    __Ownable_init_unchained();
34    __Pausable_init_unchained();
35    __ReentrancyGuard_init_unchained();
36    __ERC20_init_unchained(_name, _symbol);
37    __ERC20Pausable_init_unchained();
38
39    pool = _pool;
40    weth = ERC20(pool.underlyer());
41    require(address(weth) != address(0), "POOL_DNE");
42  }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Apparently, this smart contract is designed to work with wETH as an underlying token. In that case, it is recommended to check that pool.underlyer() is wETH actual direction to prevent deploying a nonfunctional smart contract.

Remediation Plan:

SOLVED: The Tokemak Team solved this issue by using the wETH token address defined in AdressRegistry.sol instead of setting it again when initializing the contract.

Listing 9: EthPool.sol (Line 86)

```
67          weth = IWETH(IAddressRegistry(_addressRegistry).weth());
68
```

Remediation Pull Request ID: https://github.com/Tokemak/tokemak-smart-contracts/pull/358

3.6 (HAL-06) CONFUSING VARIABLE NAMING - INFORMATIONAL

Description:

It has been observed that lptoken and lpToken variables are used in ConvexController.sol contract. These variables are used to set the Curve LP token to deposit and to store the underlying LP token from the selected pool.

Although lptoken is only used once in the contract, having similar variable names could lead to development or usage errors and decrease code's readability.

Code Location:

Listing 10: ConvexController.sol (Lines 51,52)

```
46     ) external onlyManager onlyAddLiquidity {
47         require(addressRegistry.checkAddress(lpToken, 0), "
↳ INVALID_LP_TOKEN");
48         require(staking != address(0), "INVALID_STAKING_ADDRESS");
49         require(amount > 0, "INVALID_AMOUNT");
50
51         (address lptoken, , , address crvRewards, , ) = BOOSTER.
↳ poolInfo(poolId);
52         require(lpToken == lptoken, "POOL_ID_LP_TOKEN_MISMATCH");
53         require(staking == crvRewards, "POOL_ID_STAKING_MISMATCH")
↳ ;
54
55         _approve(IERC20(lpToken), amount);
56
57         uint256 beforeBalance = IConvexBaseRewards(staking).
↳ balanceOf(address(this));
58
59         bool success = BOOSTER.deposit(poolId, amount, true);
60         require(success, "DEPOSIT_AND_STAKE_FAILED");
61
62         uint256 balanceChange = IConvexBaseRewards(staking).
↳ balanceOf(address(this)).sub(
```

```
63                 beforeBalance
64             );
65             require(balanceChange == amount, "BALANCE_MUST_INCREASE");
66         }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is considered a good practice to follow a uniform naming criteria, avoiding the usage of similar variable names.

Remediation Plan:

SOLVED: The Tokemak Team solved this issue by maintaining the name lpToken, and changing lptoken to poolLpToken in the ConvexController.sol contract.

Listing 11: ConvexController.sol (Lines 51,52)

```
51             (address poolLpToken, , , address crvRewards, , ) =
↳ BOOSTER.poolInfo(poolId);
52             require(lpToken == poolLpToken, "POOL_ID_LP_TOKEN_MISMATCH
↳ ");
53             require(staking == crvRewards, "POOL_ID_STAKING_MISMATCH")
↳ ;
```

Remediation Pull Request ID: https://github.com/Tokemak/tokemak-smart-contracts/pull/356

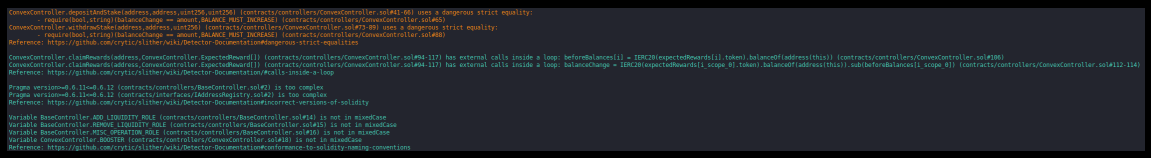
AUTOMATED TESTING

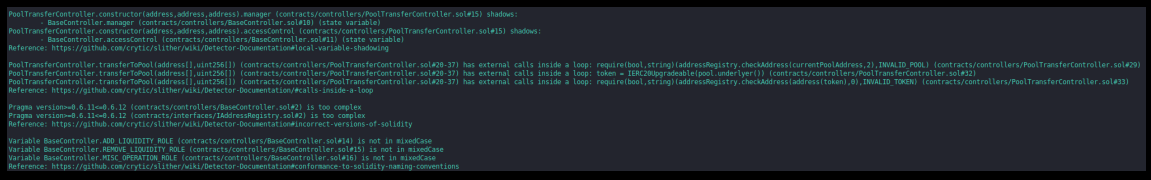
4.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped contract. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their abi and binary formats. 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.

Slither Results:





AUTOMATED TESTING

```
WethController.wrap(uint256) (contracts/controllers/WethController.sol#27-37) sends eth to arbitrary user
        Dangerous calls:
        - weth.deposit{value: amount}() (contracts/controllers/WethController.sol#33)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations

WethController.unwrap(uint256) (contracts/controllers/WethController.sol#42-54) uses a dangerous strict equality:
        - require(bool,string)(balanceBeforeEther.add(amount) == balanceAfterEther,INCORRECT_ETH_AMOUNT) (contracts/controllers/WethController.sol#53)
WethController.wrap(uint256) (contracts/controllers/WethController.sol#27-37) uses a dangerous strict equality:
        - require(bool,string)(balanceBefore.add(amount) == balanceAfter,INCORRECT_WETH_AMOUNT) (contracts/controllers/WethController.sol#36)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities

WethController.constructor(address,address,address,IWETH).manager (contracts/controllers/WethController.sol#15) shadows:
        - BaseController.manager (contracts/controllers/BaseController.sol#10) (state variable)
WethController.constructor(address,address,address,IWETH).accessControl (contracts/controllers/WethController.sol#16) shadows:
        - BaseController.accessControl (contracts/controllers/BaseController.sol#11) (state variable)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing

Pragma version>=0.6.11<=0.6.12 (contracts/controllers/BaseController.sol#2) is too complex
Pragma version>=0.6.11<=0.6.12 (contracts/interfaces/IAddressRegistry.sol#2) is too complex
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Variable BaseController.ADD_LIQUIDITY_ROLE (contracts/controllers/BaseController.sol#14) is not in mixedCase
Variable BaseController.REMOVE_LIQUIDITY_ROLE (contracts/controllers/BaseController.sol#15) is not in mixedCase
Variable BaseController.MISC_OPERATION_ROLE (contracts/controllers/BaseController.sol#16) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
```

- High and Medium vulnerabilities flagged by Slither were checked individually, and they are all false positives.
- No major issues found by Slither.

4.2 AUTOMATED SECURITY SCAN

MYTHX:

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

Results:

PCAPool.sol

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

PCAEthPool.sol

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

ConvexController.sol

Line	SWC Title	Severity	Short Description
103	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
104	(SWC-110) Assert Violation	Unknown	Out of bounds array access
105	(SWC-110) Assert Violation	Unknown	Out of bounds array access
106	(SWC-110) Assert Violation	Unknown	Out of bounds array access
111	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
112	(SWC-110) Assert Violation	Unknown	Out of bounds array access
113	(SWC-110) Assert Violation	Unknown	Out of bounds array access
115	(SWC-110) Assert Violation	Unknown	Out of bounds array access

CurveControllerEth.sol

Line	SWC Title	Severity	Short Description
53	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
54	(SWC-110) Assert Violation	Unknown	Out of bounds array access
61	(SWC-110) Assert Violation	Unknown	Out of bounds array access
64	(SWC-110) Assert Violation	Unknown	Out of bounds array access
70	(SWC-110) Assert Violation	Unknown	Out of bounds array access
190	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
192	(SWC-110) Assert Violation	Unknown	Out of bounds array access
202	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
203	(SWC-110) Assert Violation	Unknown	Out of bounds array access
204	(SWC-110) Assert Violation	Unknown	Out of bounds array access

PoolTransferController.sol

Line	SWC Title	Severity	Short Description
24	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
25	(SWC-110) Assert Violation	Unknown	Out of bounds array access
26	(SWC-110) Assert Violation	Unknown	Out of bounds array access

- No issues were found by MythX for Pool.sol and WEthController.sol
- DoS with Failed Call flagged by MythX is a false positive.
- Assert violations flagged by MythX are false positives.
- Requirement violations flagged by MythX are false positives.
- Reentrancies flagged by MythX are false positives.
- Usage of “++” is safely used.
- No major issues found by MythX.

THANK YOU FOR CHOOSING

// HALBORN